NebulaGraph Database 手册

master

NebulaGraph

2023 Vesoft Inc.

Table of contents

1. 欢迎阅读 NebulaGraph master 文档

Note

本文档更新时间2025-8-4，GitHub commit e09b867。该版本主色系为"桑色"，色号为 #55295B。

1.1 快速开始

- 快速开始
- 部署要求
- nGQL 命令汇总
- FAQ
- 生态工具
- Academy 课程
- 在线体验

1.2 最新发布

- NebulaGraph master
- NebulaGraph Dashboard Community
- NebulaGraph Studio

1.3 其他资料

- 学习路径
- 引用 NebulaGraph
- 论坛
- 主页
- 系列视频
- 英文文档

1.4 图例说明

Note

额外的信息或者操作相关的提醒等。

Caution

可能会产生不良影响，例如导致性能下降或引发已知的小问题。

Warning

可能导致严重后果，例如数据丢失、系统崩溃。

Danger

可能导致极其严重的后果，例如系统损坏、信息泄露。

Compatibility

nGQL 与 openCypher 的兼容性或 nGQL 当前版本与历史版本的兼容性。

Enterpriseonly

描述社区版和企业版的差异。

1.5 修改文档中的错误

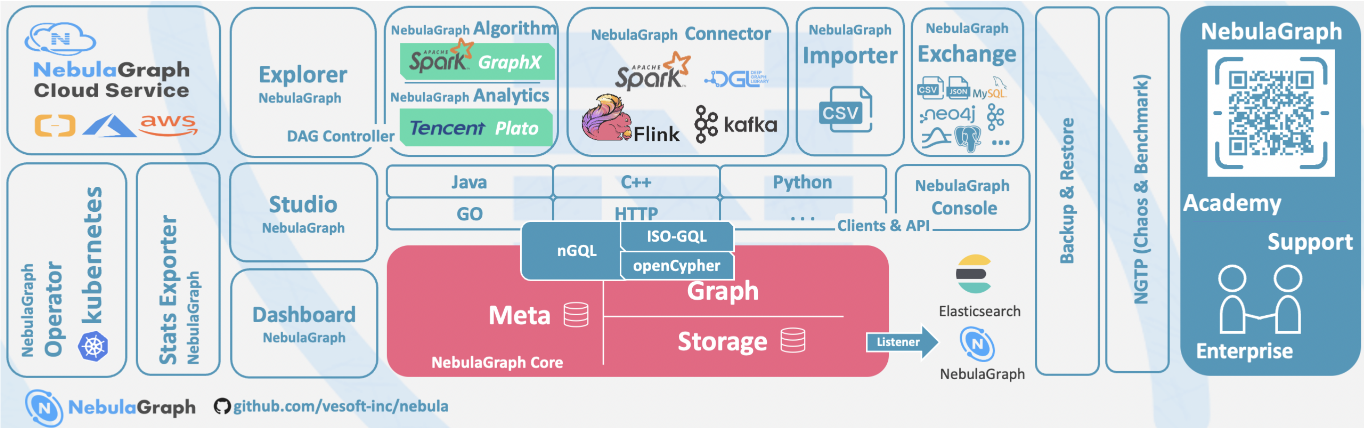
NebulaGraph 文档以 Markdown 语言编写。单击文档标题右上侧的铅笔图标即可提交修改建议。

最后更新: 2025年8月4日

2. 简介

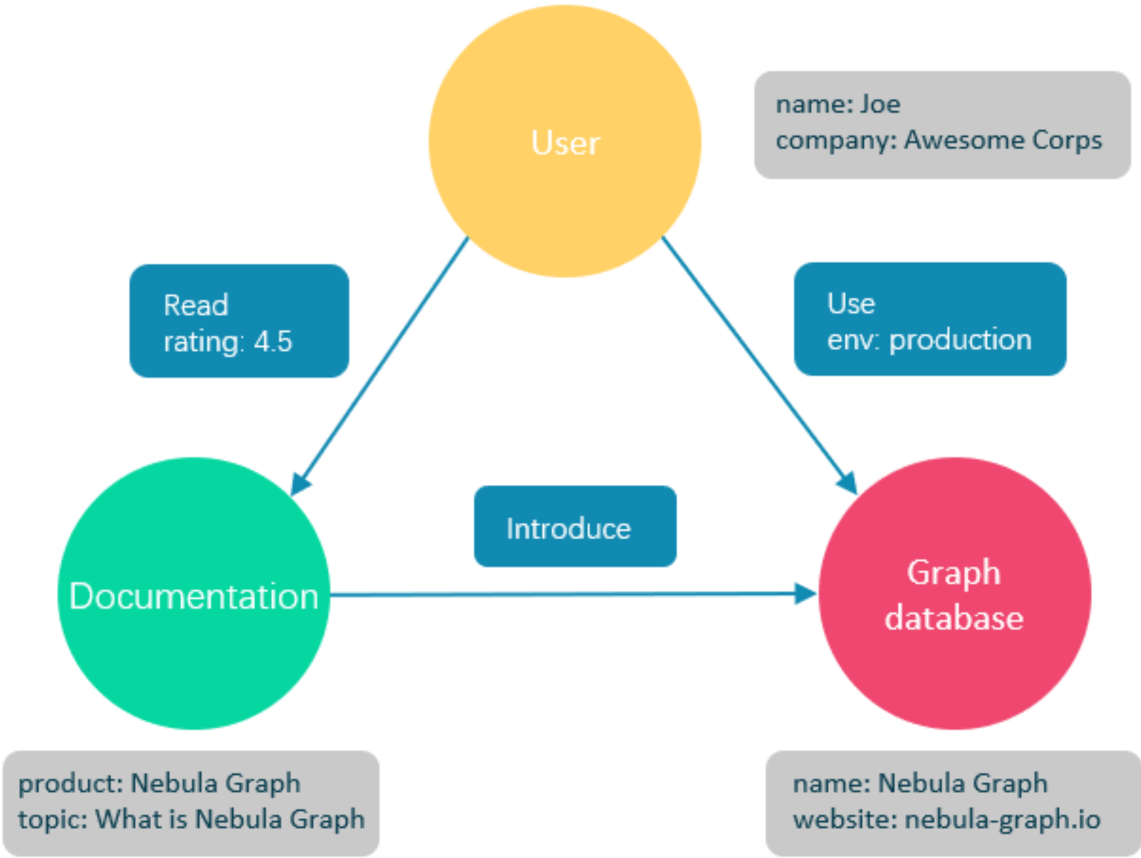
2.1 什么是 NebulaGraph

NebulaGraph 是一款开源的、分布式的、易扩展的原生图数据库，能够承载包含数千亿个点和数万亿条边的超大规模数据集，并且提供毫秒级查询。



2.1.1 什么是图数据库

图数据库是专门存储庞大的图形网络并从中检索信息的数据库。它可以将图中的数据高效存储为点（Vertex）和边（Edge），还可以将属性（Property）附加到点和边上。



图数据库适合存储大多数从现实抽象出的数据类型。世界上几乎所有领域的事物都有内在联系，像关系型数据库这样的建模系统会提取实体之间的关系，并将关系单独存储到表和列中，而实体的类型和属性存储在其他列甚至其他表中，这使得数据管理费时费力。

NebulaGraph 作为一个典型的图数据库，可以将丰富的关系通过边及其类型和属性自然地呈现。

2.1.2 NebulaGraph 的优势

开源

NebulaGraph 是在 Apache 2.0 条款下开发的。越来越多的人，如数据库开发人员、数据科学家、安全专家、算法工程师，都参与到 NebulaGraph 的设计和开发中来，欢迎访问 NebulaGraph GitHub 主页参与开源项目。

高性能

基于图数据库的特性使用 C++ 编写的 NebulaGraph，可以提供毫秒级查询。众多数据库中，NebulaGraph 在图数据服务领域展现了卓越的性能，数据规模越大，NebulaGraph 优势就越大。详情请参见 NebulaGraph benchmarking 页面。

易扩展

NebulaGraph 采用 shared-nothing 架构，支持在不停止数据库服务的情况下扩缩容。

易开发

NebulaGraph 提供 Java、Python、C++ 和 Go 等流行编程语言的客户端，更多客户端仍在开发中。详情请参见 NebulaGraph clients。

高可靠访问控制

NebulaGraph 支持严格的角色访问控制和 LDAP（Lightweight Directory Access Protocol）等外部认证服务，能够有效提高数据安全性。详情请参见验证和授权。

生态多样化

NebulaGraph 开放了越来越多的原生工具，例如 NebulaGraph Studio、NebulaGraph Console、NebulaGraph Exchange 等，更多工具可以查看生态工具概览。

此外，NebulaGraph 还具备与 Spark、Flink、HBase 等产品整合的能力，在这个充满挑战与机遇的时代，大大增强了自身的竞争力。

兼容 **openCypher** 查询语言

NebulaGraph 查询语言，简称为 nGQL，是一种声明性的、部分兼容 openCypher 的文本查询语言，易于理解和使用。详细语法请参见 nGQL 指南。

面向未来硬件，读写平衡

闪存型设备有着极高的性能，并且价格快速下降，NebulaGraph 是一个面向 SSD 设计的产品，相比于基于 HDD + 大内存的产品，更适合面向未来的硬件趋势，也更容易做到读写平衡。

灵活数据建模

用户可以轻松地在 NebulaGraph 中建立数据模型，不必将数据强制转换为关系表。而且可以自由增加、更新和删除属性。详情请参见数据模型。

广受欢迎

腾讯、美团、京东、快手、360 等科技巨头都在使用 NebulaGraph 。详情请参见 NebulaGraph 官网。

2.1.3 适用场景

NebulaGraph 可用于各种基于图的业务场景。为节约转换各类数据到关系型数据库的时间，以及避免复杂查询，建议使用 NebulaGraph 。

欺诈检测

金融机构必须仔细研究大量的交易信息，才能检测出潜在的金融欺诈行为，并了解某个欺诈行为和设备的内在关联。这种场景可以通过图来建模，然后借助 NebulaGraph，可以很容易地检测出诈骗团伙或其他复杂诈骗行为。

实时推荐

NebulaGraph 能够及时处理访问者产生的实时信息，并且精准推送文章、视频、产品和服务。

知识图谱

自然语言可以转化为知识图谱，存储在 NebulaGraph 中。用自然语言组织的问题可以通过智能问答系统中的语义解析器进行解析并重新组织，然后从知识图谱中检索出问题的可能答案，提供给提问人。

社交网络

人际关系信息是典型的图数据，NebulaGraph 可以轻松处理数十亿人和数万亿人际关系的社交网络信息，并在海量并发的情况下，提供快速的好友推荐和工作岗位查询。

2.1.4 视频

用户也可以通过视频了解什么是图数据。

- NebulaGraph 介绍视频（01 分 39 秒）

2.1.5 主题演讲

查看演讲快速了解图数据库概况。

2.1.6 相关链接

- 官方网站
- 文档首页
- 博客首页
- 论坛
- GitHub

最后更新: 2025年8月4日

2.2 数据模型

本文介绍 NebulaGraph 的数据模型。数据模型是一种组织数据并说明它们如何相互关联的模型。

2.2.1 数据模型

NebulaGraph 数据模型使用 6 种基本的数据模型：

- 图空间（Space）

  图空间用于隔离不同团队或者项目的数据。不同图空间的数据是相互隔离的，可以指定不同的存储副本数、权限、分片等。

- 点（Vertex）

  点用来保存实体对象，特点如下：

- 点是用点标识符（`VID`）标识的。`VID` 在同一图空间中唯一。VID 是一个 int64，或者 fixed_string(N)。
- 点可以有 0 到多个 Tag。

> Compatibility
>
> NebulaGraph 2.x 及以下版本中的点必须包含至少一个 Tag。

- 边（Edge）

  边是用来连接点的，表示两个点之间的关系或行为，特点如下：

- 两点之间可以有多条边。
- 边是有方向的，不存在无向边。
- 四元组 `<起点 VID、Edge type、边排序值 (rank)、终点 VID>` 用于唯一标识一条边。边没有 EID。
- 一条边有且仅有一个 Edge type。
- 一条边有且仅有一个 Rank，类型为 int64，默认值为 0。

> 关于
>
> Rank 可以用来区分 Edge type、起始点、目的点都相同的边。该值完全由用户自己指定。
> 读取时必须自行取得全部的 Rank 值后排序过滤和拼接。
> 不支持诸如 `next()`，`pre()`，`head()`，`tail()`，`max()`，`min()`，`lessThan()`，`moreThan()` 等函数功能，也不能通过创建索引加速访问或者条件过滤。

- 标签（Tag）

  Tag 由一组事先预定义的属性构成。

- 边类型（Edge type）

  Edge type 由一组事先预定义的属性构成。

- 属性（Property）

  属性是指以键值对（Key-value pair）形式表示的信息。

> Note
>
> Tag 和 Edge type 的作用，类似于关系型数据库中“点表”和“边表”的表结构。

2.2.2 有向属性图

NebulaGraph 使用有向属性图模型，指点和边构成的图，这些边是有方向的，点和边都可以有属性。

下表为篮球运动员数据集的结构示例，包括两种类型的点（player、team）和两种类型的边（serve、follow）。

类型	名称	属性名（数据类型）	说明
Tag	player	name (string) age (int)	表示球员。属性 `name` 表示姓名，`age` 表示年龄。
Tag	team	name (string)	表示球队。属性 `name` 表示球队名。
Edge type	serve	start_year (int) end_year (int)	表示球员效力球队的行为。该行为将球员和球队联系起来，方向是从球员到球队。属性 `start_year` 表示效力的起始年份，`end_year` 表示效力的结束年份。
Edge type	follow	degree (int)	表示推特上球员关注另一个球员的行为。该行为将两个球员联系起来，方向是从一个球员到另一个球员。属性 `degree` 表示喜爱程度。

> Note
>
> NebulaGraph 中没有无向边，只支持有向边。

> Compatibility
>
> 由于 NebulaGraph master 的数据模型中，允许存在"悬挂边"，因此在增删时，用户需自行保证“一条边所对应的起点和终点”的存在性。详见 INSERT VERTEX、DELETE VERTEX、INSERT EDGE、DELETE EDGE。
>
> 不支持 openCypher 中的 MERGE 语句。

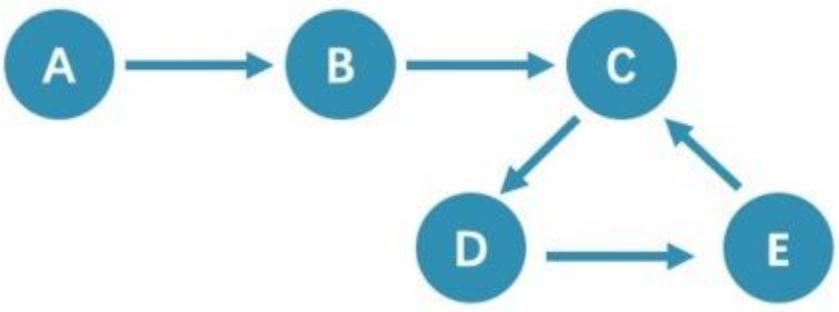
最后更新: 2025年8月4日

2.3 路径

图论中一个非常重要的概念是路径，路径是指一个有限或无限的边序列，这些边连接着一系列的点。

路径的类型分为三种：`walk`、`trail`、`path`。关于路径的详细说明，请参见维基百科。

本文以下图为例进行简单介绍。



2.3.1 walk

`walk` 类型的路径由有限或无限的边序列构成。遍历时点和边可以重复。

查看示例图，由于 C、D、E 构成了一个环，因此该图包含无限个路径，例如 `A->B->C->D->E`、`A->B->C->D->E->C`、`A->B->C->D->E->C->D`。

> Note
>
> `GO` 语句采用的是 `walk` 类型路径。

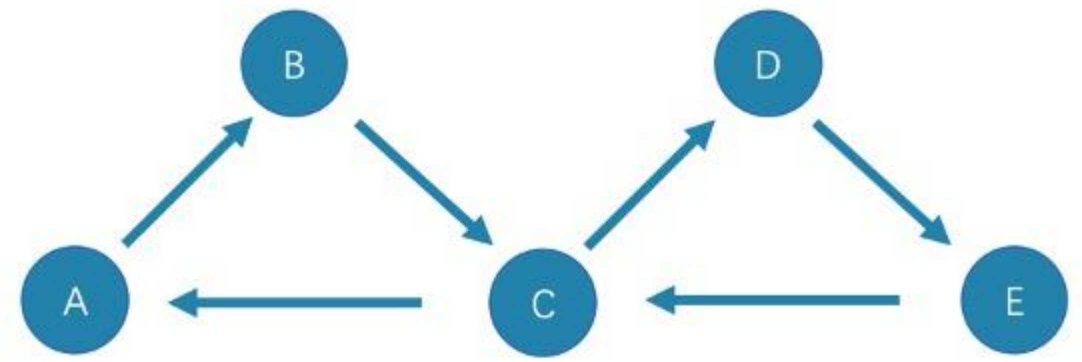
2.3.2 trail

`trail` 类型的路径由有限的边序列构成。遍历时只有点可以重复，边不可以重复。柯尼斯堡七桥问题的路径类型就是 `trail`。

查看示例图，由于边不可以重复，所以该图包含有限个路径，最长路径由 5 条边组成：`A->B->C->D->E->C`。

> Note
>
> `MATCH`、`FIND PATH` 和 `GET SUBGRAPH` 语句采用的是 `trail` 类型路径。

在 trail 类型中，还有 `cycle` 和 `circuit` 两种特殊的路径类型，以下图为例对这两种特殊的路径类型进行介绍。



- cycle

  `cycle` 是封闭的 `trail` 类型的路径，遍历时边不可以重复，起点和终点重复，并且没有其他点重复。在此示例图中，最长路径由三条边组成：`A->B->C->A` 或 `C->D->E->C`.

- circuit

  `circuit` 也是封闭的 `trail` 类型的路径，遍历时边不可以重复，除起点和终点重复外，可能存在其他点重复。在此示例图中，最长路径为：`A->B->C->D->E->C->A`。

2.3.3 path

`path` 类型的路径由有限的边序列构成。遍历时点和边都不可以重复。

查看示例图，由于点和边都不可以重复，所以该图包含有限个路径，最长路径由 4 条边组成：`A->B->C->D->E`。

2.3.4 视频

用户也可以观看视频了解路径的相关概念。

Path（03 分 09 秒）

最后更新: 2025年8月4日

2.4 点 VID

在一个图空间中，一个点由点的 ID 唯一标识，即 VID 或 Vertex ID。

2.4.1 VID 的特点

- VID 数据类型只可以为定长字符串 `FIXED_STRING(<N>)` 或 `INT64`。一个图空间只能选用其中一种 VID 类型。
- VID 在一个图空间中必须唯一，其作用类似于关系型数据库中的主键（索引+唯一约束）。但不同图空间中的 VID 是完全独立无关的。
- 点 VID 的生成方式必须由用户自行指定，系统不提供自增 ID 或者 UUID。
- VID 相同的点，会被认为是同一个点。例如：
- VID 相当于一个实体的唯一标号，例如一个人的身份证号。Tag 相当于实体所拥有的类型，例如"滴滴司机"和"老板"。不同的 Tag 又相应定义了两组不同的属性，例如"驾照号、驾龄、接单量、接单小号"和"工号、薪水、债务额度、商务电话"。
- 同时操作相同 VID 并且相同 Tag 的两条 `INSERT` 语句（均无 `IF NOT EXISTS` 参数），晚写入的 `INSERT` 会覆盖先写入的。
- 同时操作包含相同 VID 但是两个不同 `TAG A` 和 `TAG B` 的两条 `INSERT` 语句，对 `TAG A` 的操作不会影响 `TAG B`。
- VID 通常会被（LSM-tree 方式）索引并缓存在内存中，因此直接访问 VID 的性能最高。

2.4.2 VID 使用建议

- NebulaGraph 1.x 只支持 VID 类型为 `INT64`，从 2.x 开始支持 `INT64` 和 `FIXED_STRING(<N>)`。在 `CREATE SPACE` 中通过参数 `vid_type` 可以指定 VID 类型。
- 可以使用 `id()` 函数，指定或引用该点的 VID。
- 可以使用 `LOOKUP` 或者 `MATCH` 语句，来通过属性索引查找对应的 VID。
- 性能上，直接通过 VID 找到点的语句性能最高，例如 `DELETE xxx WHERE id(xxx) == "player100"`，或者 `GO FROM "player100"` 等语句。通过属性先查找 VID，再进行图操作的性能会变差，例如 `LOOKUP | GO FROM $-.ids` 等语句，相比前者多了一次内存或硬盘的随机读（`LOOKUP`）以及一次序列化（`|`）。

2.4.3 VID 生成建议

VID 的生成工作完全交给应用端，有一些通用的建议：

- （最优）通过有唯一性的主键或者属性来直接作为 VID；属性访问依赖于 VID;
- 通过有唯一性的属性组合来生成 VID，属性访问依赖于属性索引。
- 通过 snowflake 等算法生成 VID，属性访问依赖于属性索引。
- 如果个别记录的主键特别长，但绝大多数记录的主键都很短的情况，不要将 `FIXED_STRING(<N>)` 的 `N` 设置成超大，这会浪费大量内存和硬盘，也会降低性能。此时可通过 BASE64，MD5，hash 编码加拼接的方式来生成。
- 如果用 hash 方式生成 int64 VID：在有 10 亿个点的情况下，发生 hash 冲突的概率大约是 1/10。边的数量与碰撞的概率无关。

2.4.4 定义和修改 VID 与其数据类型

VID 的数据类型必须在创建图空间时定义，且一旦定义无法修改。

VID 必须在插入点时设置，且一旦设置无法修改。

2.4.5 "查询起始点"(`start vid`) 与全局扫描

绝大多数情况下，NebulaGraph 的查询语句（`MATCH`、`GO`、`LOOKUP`）的执行计划，必须要通过一定方式找到查询起始点的 VID（`start vid`）。

定位 `start vid` 只有两种方式：

1. 例如 `GO FROM "player100" OVER` 是在语句中显式的指明 `start vid` 是 "player100"；
2. 例如 `LOOKUP ON player WHERE player.name == "Tony Parker"` 或者 `MATCH (v:player {name:"Tony Parker"})`，是通过属性 `player.name` 的索引来定位到 `start vid`；

最后更新: 2025年8月4日

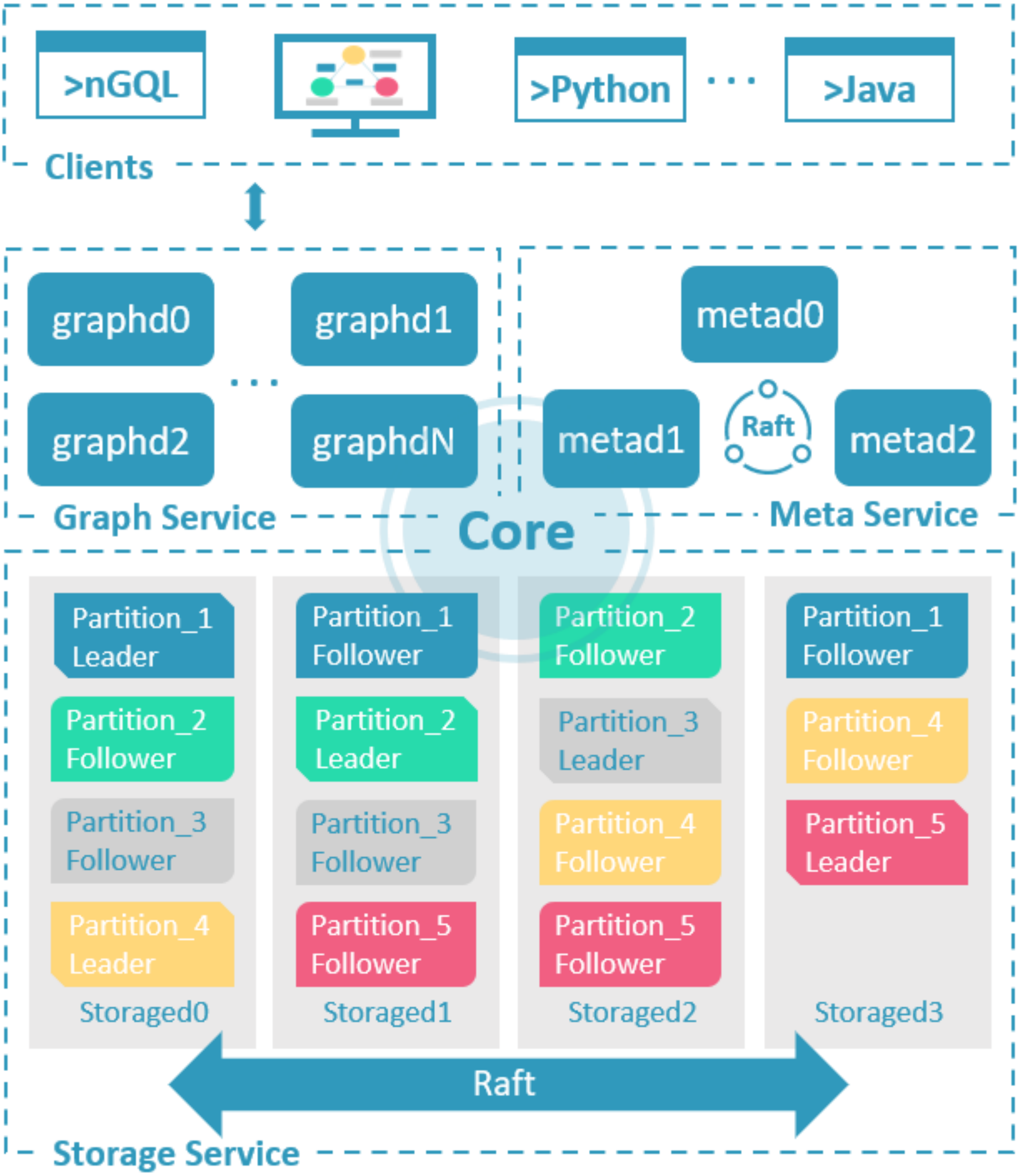
2.5 服务架构

2.5.1 NebulaGraph 架构总览

NebulaGraph 由三种服务构成：Graph 服务、Meta 服务和 Storage 服务，是一种存储与计算分离的架构。

每个服务都有可执行的二进制文件和对应进程，用户可以使用这些二进制文件在一个或多个计算机上部署 NebulaGraph 集群。

下图展示了 NebulaGraph 集群的经典架构。



Meta 服务

在 NebulaGraph 架构中，Meta 服务是由 nebula-metad 进程提供的，负责数据管理，例如 Schema 操作、集群管理和用户权限管理等。

Meta 服务的详细说明，请参见 Meta 服务。

Graph 服务和 Storage 服务

NebulaGraph 采用计算存储分离架构。Graph 服务负责处理计算请求，Storage 服务负责存储数据。它们由不同的进程提供，Graph 服务是由 nebula-graphd 进程提供，Storage 服务是由 nebula-storaged 进程提供。计算存储分离架构的优势如下：

- 易扩展

  分布式架构保证了 Graph 服务和 Storage 服务的灵活性，方便扩容和缩容。

- 高可用

  如果提供 Graph 服务的服务器有一部分出现故障，其余服务器可以继续为客户端提供服务，而且 Storage 服务存储的数据不会丢失。服务恢复速度较快，甚至能做到用户无感知。

- 节约成本

  计算存储分离架构能够提高资源利用率，而且可根据业务需求灵活控制成本。

- 更多可能性

  基于分离架构的特性，Graph 服务将可以在更多类型的存储引擎上单独运行，Storage 服务也可以为多种目的计算引擎提供服务。

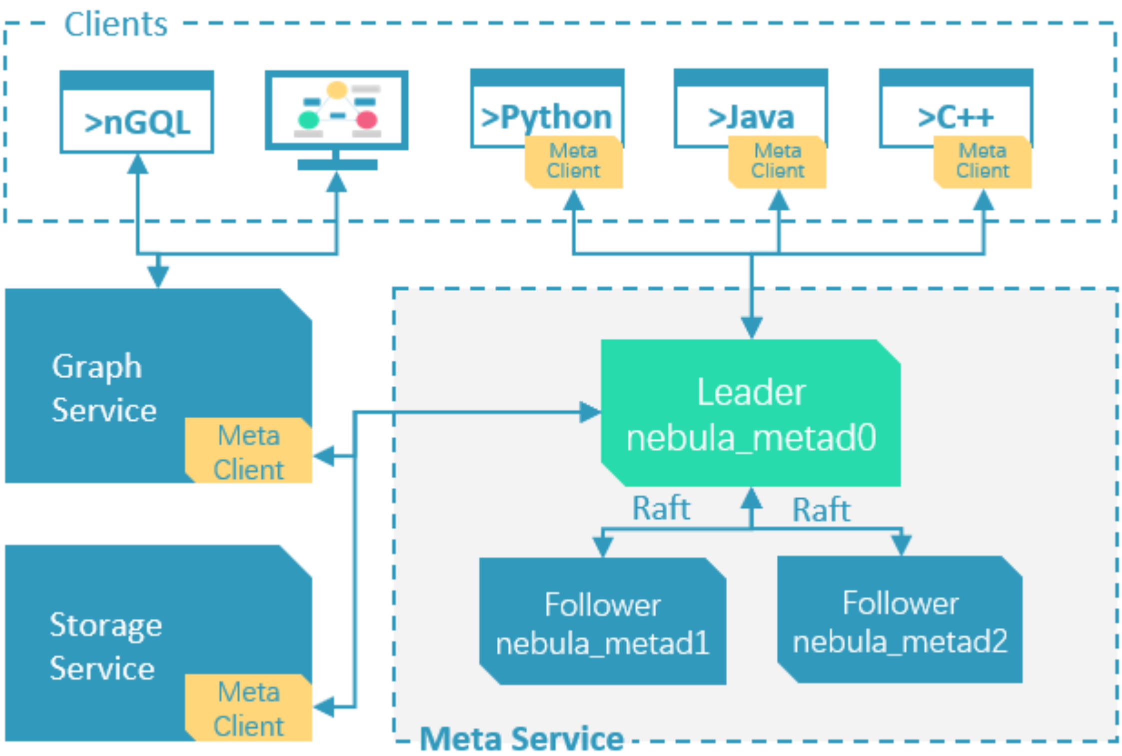
Graph 服务和 Storage 服务的详细说明，请参见 Graph 服务和 Storage 服务。

最后更新: 2025年8月4日

2.5.2 Meta 服务

本文介绍 Meta 服务的架构和功能。

Meta 服务架构



Meta 服务是由 nebula-metad 进程提供的，用户可以根据场景配置 nebula-metad 进程数量：

- 测试环境中，用户可以在 NebulaGraph 集群中部署 1 个或 3 个 nebula-metad 进程。如果要部署 3 个，用户可以将它们部署在 1 台机器上，或者分别部署在不同的机器上。
- 生产环境中，建议在 NebulaGraph 集群中部署 3 个 nebula-metad 进程。请将这些进程部署在不同的机器上以保证高可用。

所有 nebula-metad 进程构成了基于 Raft 协议的集群，其中一个进程是 leader，其他进程都是 follower。

leader 是由多数派选举出来，只有 leader 能够对客户端或其他组件提供服务，其他 follower 作为候补，如果 leader 出现故障，会在所有 follower 中选举出新的 leader。

> Note
>
> leader 和 follower 的数据通过 Raft 协议保持一致，因此 leader 故障和选举新 leader 不会导致数据不一致。更多关于 Raft 的介绍见 Storage 服务。

Meta 服务功能

管理用户账号

Meta 服务中存储了用户的账号和权限信息，当客户端通过账号发送请求给 Meta 服务，Meta 服务会检查账号信息，以及该账号是否有对应的请求权限。

更多 NebulaGraph 的访问控制说明，请参见身份验证。

管理分片

Meta 服务负责存储和管理分片的位置信息，并且保证分片的负载均衡。

管理图空间

NebulaGraph 支持多个图空间，不同图空间内的数据是安全隔离的。Meta 服务存储所有图空间的元数据（非完整数据），并跟踪数据的变更，例如增加或删除图空间。

管理 SCHEMA 信息

NebulaGraph 是强类型图数据库，它的 Schema 包括 Tag、Edge type、Tag 属性和 Edge type 属性。

Meta 服务中存储了 Schema 信息，同时还负责 Schema 的添加、修改和删除，并记录它们的版本。

更多 NebulaGraph 的 Schema 信息，请参见数据模型。

管理 TTL 信息

Meta 服务存储 TTL（Time To Live）定义信息，可以用于设置数据生命周期。数据过期后，会由 Storage 服务进行处理，具体过程参见 TTL。

管理作业

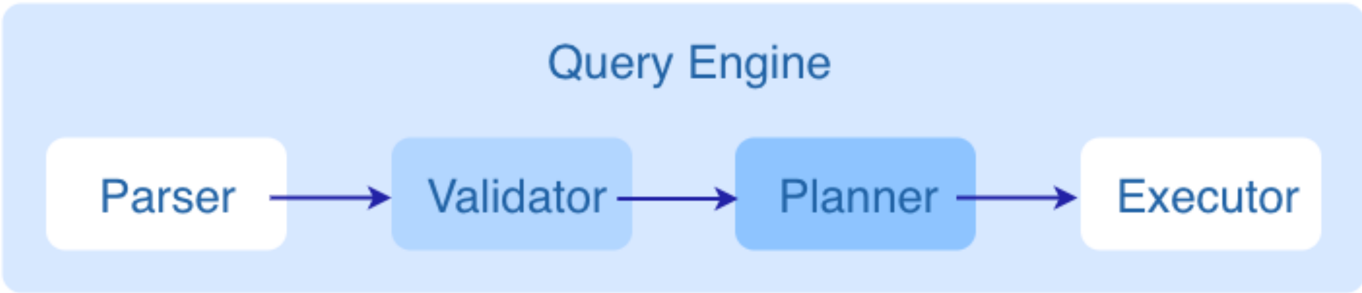
Meta 服务中的作业管理模块负责作业的创建、排队、查询和删除。

最后更新: 2025年8月4日

2.5.3 Graph 服务

Graph 服务主要负责处理查询请求，包括解析查询语句、校验语句、生成执行计划以及按照执行计划执行四个大步骤，本文将基于这些步骤介绍 Graph 服务。

Graph 服务架构



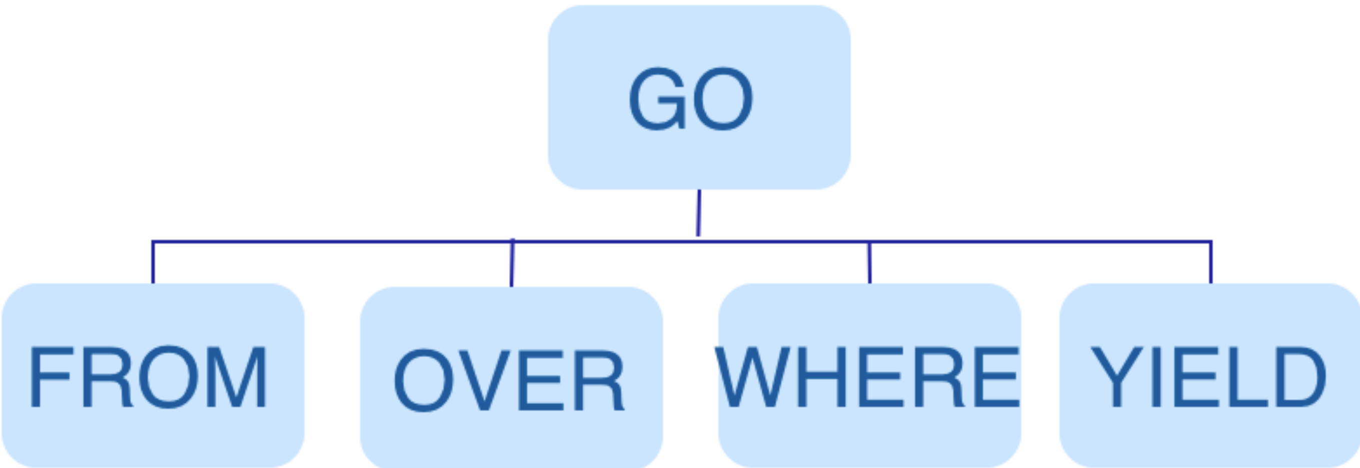
查询请求发送到 Graph 服务后，会由如下模块依次处理：

1. Parser：词法语法解析模块。
2. Validator：语义校验模块。
3. Planner：执行计划与优化器模块。
4. Executor：执行引擎模块。

Parser

Parser 模块收到请求后，通过 Flex（词法分析工具）和 Bison（语法分析工具）生成的词法语法解析器，将语句转换为抽象语法树（AST），在语法解析阶段会拦截不符合语法规则的语句。

例如 `GO FROM "Tim" OVER like WHERE properties(edge).likeness > 8.0 YIELD dst(edge)` 语句转换的 AST 如下。



Validator

Validator 模块对生成的 AST 进行语义校验，主要包括：

- 校验元数据信息

  校验语句中的元数据信息是否正确。

  例如解析 `OVER`、 `WHERE` 和 `YIELD` 语句时，会查找 Schema 校验 Edge type、Tag 的信息是否存在，或者插入数据时校验插入的数据类型和 Schema 中的是否一致。

- 校验上下文引用信息

  校验引用的变量是否存在或者引用的属性是否属于变量。

  例如语句 `$var = GO FROM "Tim" OVER like YIELD dst(edge) AS ID; GO FROM $var.ID OVER serve YIELD dst(edge)`，Validator 模块首先会检查变量 `var` 是否定义，其次再检查属性 `ID` 是否属于变量 `var`。

- 校验类型推断

  推断表达式的结果类型，并根据子句校验类型是否正确。

  例如 `WHERE` 子句要求结果是 `bool`、 `null` 或者 `empty`。

- 校验 `*` 代表的信息

  查询语句中包含 `*` 时，校验子句时需要将 `*` 涉及的 Schema 都进行校验。

  例如语句 `GO FROM "Tim" OVER * YIELD dst(edge), properties(edge).likeness, dst(edge)`，校验 `OVER` 子句时需要校验所有的 Edge type，如果 Edge type 包含 `like` 和 `serve`，该语句会展开为 `GO FROM "Tim" OVER like,serve YIELD dst(edge), properties(edge).likeness, dst(edge)`。

- 校验输入输出

  校验管道符（|）前后的一致性。

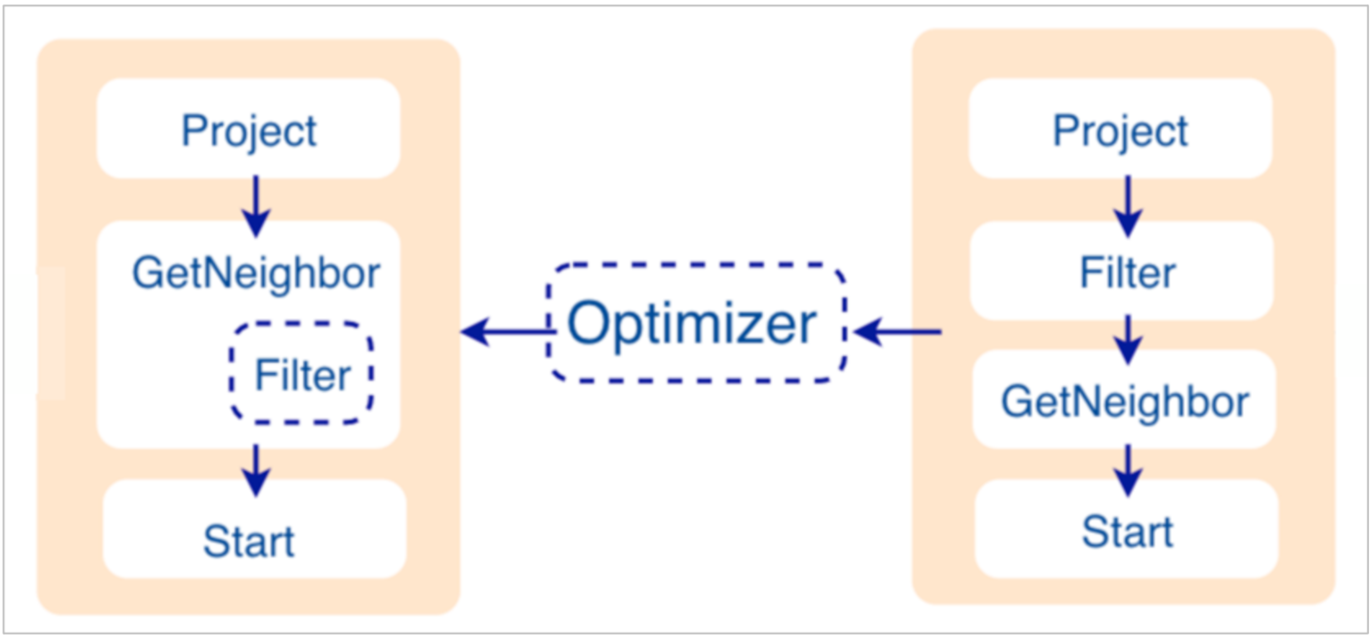
  例如语句 `GO FROM "Tim" OVER like YIELD dst(edge) AS ID | GO FROM $-.ID OVER serve YIELD dst(edge)`，Validator 模块会校验 `$-.ID` 在管道符左侧是否已经定义。

校验完成后，Validator 模块还会生成一个默认可执行，但是未进行优化的执行计划，存储在目录 `src/planner` 内。

Planner

如果配置文件 `nebula-graphd.conf` 中 `enable_optimizer` 设置为 `false`，Planner 模块不会优化 Validator 模块生成的执行计划，而是直接交给 Executor 模块执行。

如果配置文件 `nebula-graphd.conf` 中 `enable_optimizer` 设置为 `true`，Planner 模块会对 Validator 模块生成的执行计划进行优化。如下图所示。



- 优化前

  如上图右侧未优化的执行计划，每个节点依赖另一个节点，例如根节点 `Project` 依赖 `Filter`、`Filter` 依赖 `GetNeighbor`，最终找到叶子节点 `Start`，才能开始执行（并非真正执行）。

  在这个过程中，每个节点会有对应的输入变量和输出变量，这些变量存储在一个哈希表中。由于执行计划不是真正执行，所以哈希表中每个 key 的 value 值都为空（除了 `Start` 节点，起始数据会存储在该节点的输入变量中）。哈希表定义在仓库 `nebula-graph` 内的 `src/context/ExecutionContext.cpp` 中。

  例如哈希表的名称为 `ResultMap`，在建立 `Filter` 这个节点时，定义该节点从 `ResultMap["GN1"]` 中读取数据，然后将结果存储在 `ResultMap["Filter2"]` 中，依次类推，将每个节点的输入输出都确定好。

- 优化过程

  Planner 模块目前的优化方式是 RBO（rule-based optimization），即预定义优化规则，然后对 Validator 模块生成的默认执行计划进行优化。新的优化规则 CBO（cost-based optimization）正在开发中。优化代码存储在仓库 `nebula-graph` 的目录 `src/optimizer/` 内。

  RBO 是一个自底向上的探索过程，即对于每个规则而言，都会由执行计划的根节点（示例是 `Project`）开始，一步步向下探索到最底层的节点，在过程中查看是否可以匹配规则。

  如上图所示，探索到节点 `Filter` 时，发现依赖的节点是 `GetNeighbor`，匹配预先定义的规则，就会将 `Filter` 融入到 `GetNeighbor` 中，然后移除节点 `Filter`，继续匹配下一个规则。在执行阶段，当算子 `GetNeighbor` 调用 Storage 服务的接口获取一个点的邻边时，Storage 服务内部会直接将不符合条件的边过滤掉，这样可以极大地减少传输的数据量，该优化称为过滤下推。

Executor

Executor 模块包含调度器（Scheduler）和执行器（Executor），通过调度器调度执行计划，让执行器根据执行计划生成对应的执行算子，从叶子节点开始执行，直到根节点结束。如下图所示。

每一个执行计划节点都一一对应一个执行算子，节点的输入输出在优化执行计划时已经确定，每个算子只需要拿到输入变量中的值进行计算，最后将计算结果放入对应的输出变量中即可，所以只需要从节点 `Start` 一步步执行，最后一个算子的输出变量会作为最终结果返回给客户端。

代码结构

NebulaGraph 的代码层次结构如下：

```
|--src
  |--graph
    |--context  //校验期和执行期上下文
    |--executor  //执行算子
    |--gc     //垃圾收集器
    |--optimizer //优化规则
    |--planner  //执行计划结构
    |--scheduler //调度器
    |--service  //对外服务管理
    |--session  //会话管理
    |--stats   //运行指标
    |--util    //基础组件
    |--validator //语句校验
    |--visitor  //visitor表达式
```

视频

用户也可以通过视频全方位了解 NebulaGraph 的查询引擎。

- nMeetup・上海 ｜全面解析 Query Engine（33 分 30 秒）

最后更新: 2025年8月4日

2.5.4 Storage 服务

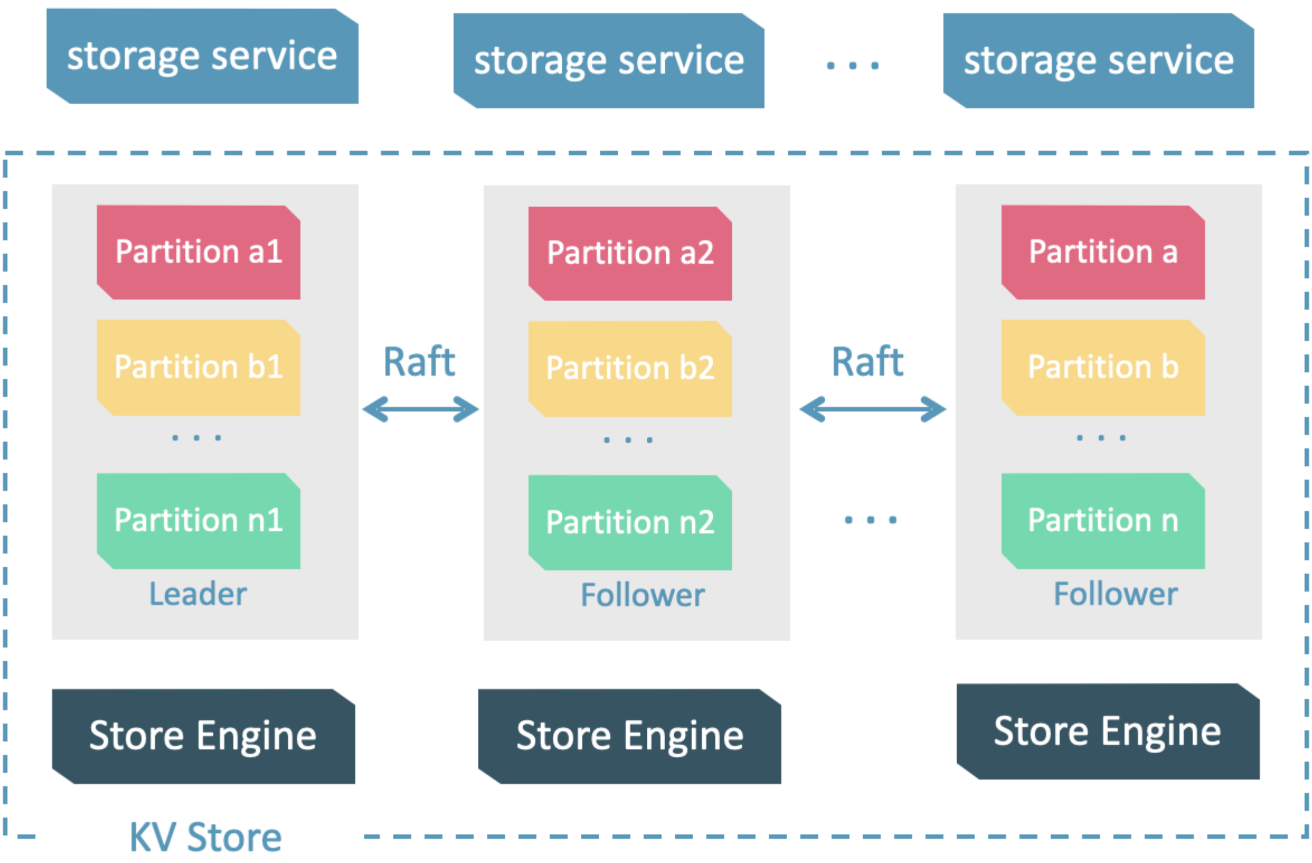
NebulaGraph 的存储包含两个部分，一个是 Meta 相关的存储，称为 Meta 服务，在前文已有介绍。

另一个是具体数据相关的存储，称为 Storage 服务。其运行在 nebula-storaged 进程中。本文仅介绍 Storage 服务的架构设计。

优势

- 高性能（自研 KVStore）
- 易水平扩展（Shared-nothing 架构，不依赖 NAS 等硬件设备）
- 强一致性（Raft）
- 高可用性（Raft）
- 支持向第三方系统进行同步（例如全文索引）

Storage 服务架构



Storage 服务是由 nebula-storaged 进程提供的，用户可以根据场景配置 nebula-storaged 进程数量，例如测试环境 1 个，生产环境 3 个。

所有 nebula-storaged 进程构成了基于 Raft 协议的集群，整个服务架构可以分为三层，从上到下依次为：

- Storage interface 层

  Storage 服务的最上层，定义了一系列和图相关的 API。API 请求会在这一层被翻译成一组针对分片的 KV 操作，例如：

- `getNeighbors`：查询一批点的出边或者入边，返回边以及对应的属性，并且支持条件过滤。
- `insert vertex/edge`：插入一条点或者边及其属性。
- `getProps`：获取一个点或者一条边的属性。

  正是这一层的存在，使得 Storage 服务变成了真正的图存储，否则 Storage 服务只是一个 KV 存储服务。

- Consensus 层

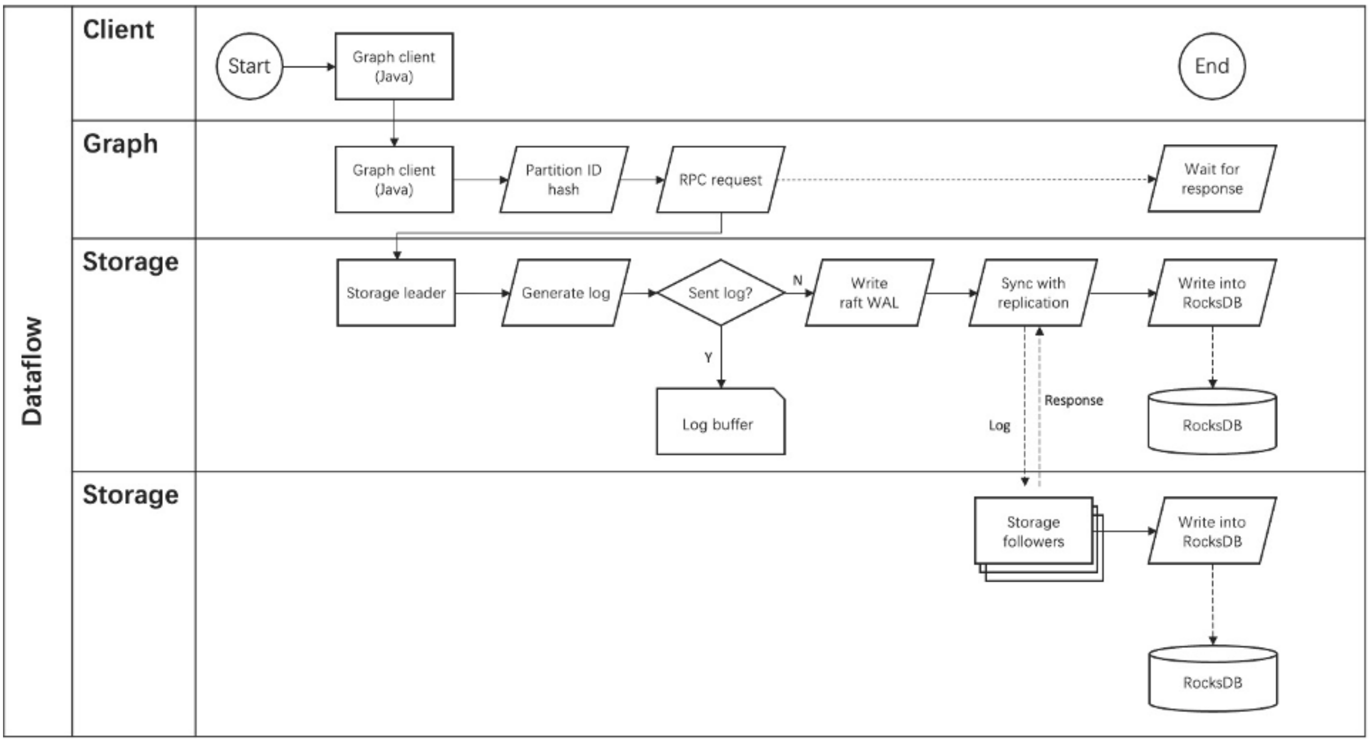
  Storage 服务的中间层，实现了 Multi Group Raft，保证强一致性和高可用性。

- Store Engine 层

  Storage 服务的最底层，是一个单机版本地存储引擎，提供对本地数据的 `get`、`put`、`scan` 等操作。相关接口存储在 `KVStore.h` 和 `KVEngine.h` 文件，用户可以根据业务需求定制开发相关的本地存储插件。

下文将基于架构介绍 Storage 服务的部分特性。

Storage 写入流程



KVStore

NebulaGraph 使用自行开发的 KVStore，而不是其他开源 KVStore，原因如下：

- 需要高性能 KVStore。
- 需要以库的形式提供，实现高效计算下推。对于强 Schema 的 NebulaGraph 来说，计算下推时如何提供 Schema 信息，是高效的关键。
- 需要数据强一致性。

基于上述原因，NebulaGraph 使用 RocksDB 作为本地存储引擎，实现了自己的 KVStore，有如下优势：

- 对于多硬盘机器，NebulaGraph 只需配置多个不同的数据目录即可充分利用多硬盘的并发能力。

- 由 Meta 服务统一管理所有 Storage 服务，可以根据所有分片的分布情况和状态，手动进行负载均衡。

> Note
>
> 不支持自动负载均衡是为了防止自动数据搬迁影响线上业务。

- 定制预写日志（WAL），每个分片都有自己的 WAL。

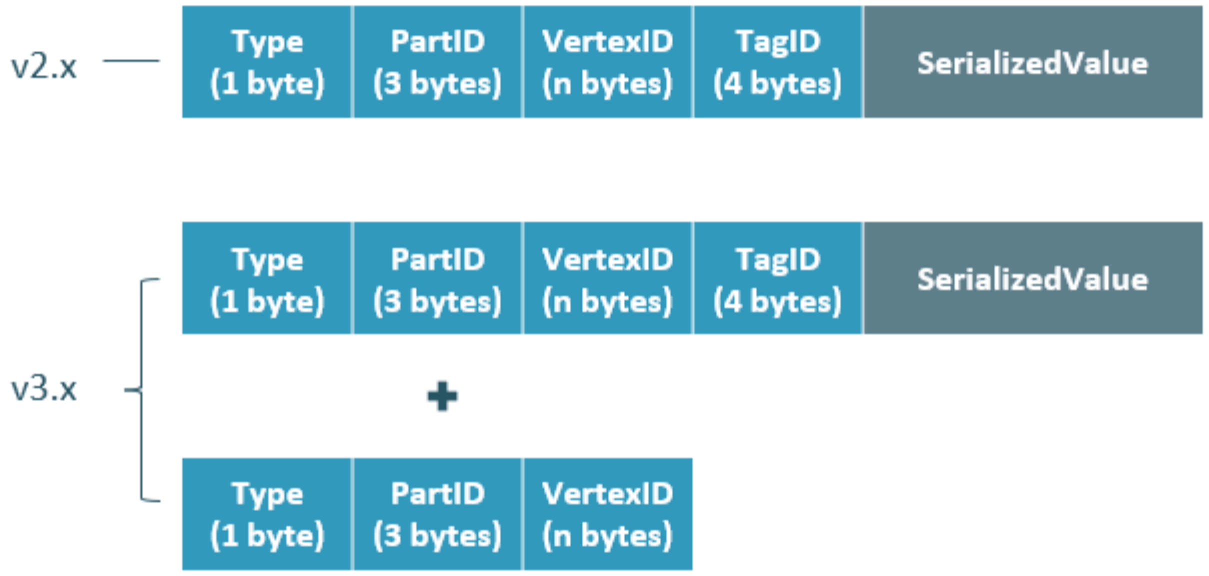
- 支持多个图空间，不同图空间相互隔离，每个图空间可以设置自己的分片数和副本数。

数据存储格式

图存储的主要数据是点和边，NebulaGraph 将点和边的信息存储为 key，同时将点和边的属性信息存储在 value 中，以便更高效地使用属性过滤。

- 点数据存储格式

  相比 NebulaGraph 2.x 版本，3.x 版本在开启无 Tag 的点配置后，每个点多了一个不含 TagID 字段并且无 value 的 key。



字段	说明
`Type`	key 类型。长度为 1 字节。
`PartID`	数据分片编号。长度为 3 字节。此字段主要用于 Storage 负载均衡（balance）时方便根据前缀扫描整个分片的数据。
`VertexID`	点 ID。当点 ID 类型为 int 时，长度为 8 字节；当点 ID 类型为 string 时，长度为创建图空间时指定的 `fixed_string` 长度。
`TagID`	点关联的 Tag ID。长度为 4 字节。
`SerializedValue`	序列化的 value，用于保存点的属性信息。

- 边数据存储格式

字段	说明
`Type`	key 类型。长度为 1 字节。
`PartID`	数据分片编号。长度为 3 字节。此字段主要用于 Storage 负载均衡（balance）时方便根据前缀扫描整个分片的数据。
`VertexID`	点 ID。前一个 `VertexID` 在出边里表示起始点 ID，在入边里表示目的点 ID；后一个 `VertexID` 出边里表示目的点 ID，在入边里表示起始点 ID。
`Edge type`	边的类型。大于 0 表示出边，小于 0 表示入边。长度为 4 字节。
`Rank`	用来处理两点之间有多个同类型边的情况。用户可以根据自己的需求进行设置，例如存放交易时间、交易流水号等。长度为 8 字节，
`PlaceHolder`	预留字段。长度为 1 字节。
`SerializedValue`	序列化的 value，用于保存边的属性信息。

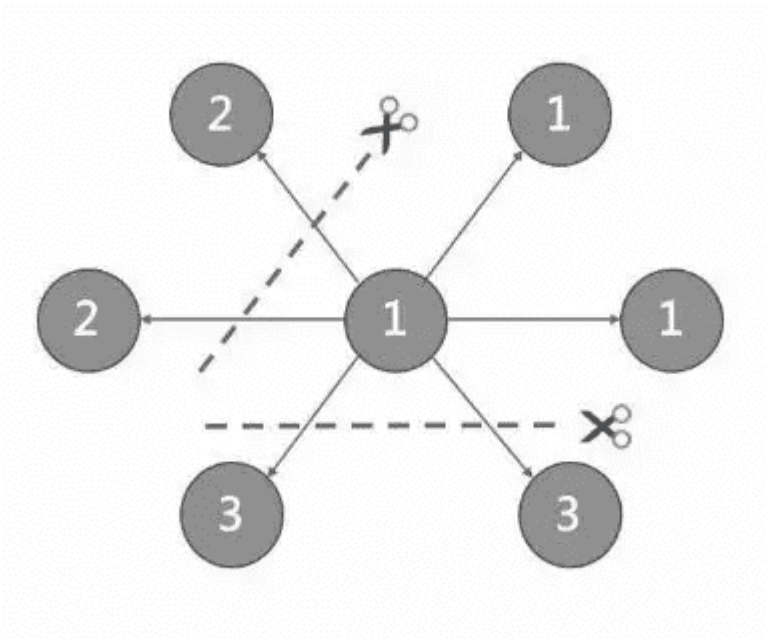
属性说明

NebulaGraph 使用强类型 Schema。

对于点或边的属性信息，NebulaGraph 会将属性信息编码后按顺序存储。由于定长属性的长度是固定的，查询时可以根据偏移量快速查询。在解码之前，需要先从 Meta 服务中查询具体的 Schema 信息（并缓存）。同时为了支持在线变更 Schema，在编码属性时，会加入对应的 Schema 版本信息。

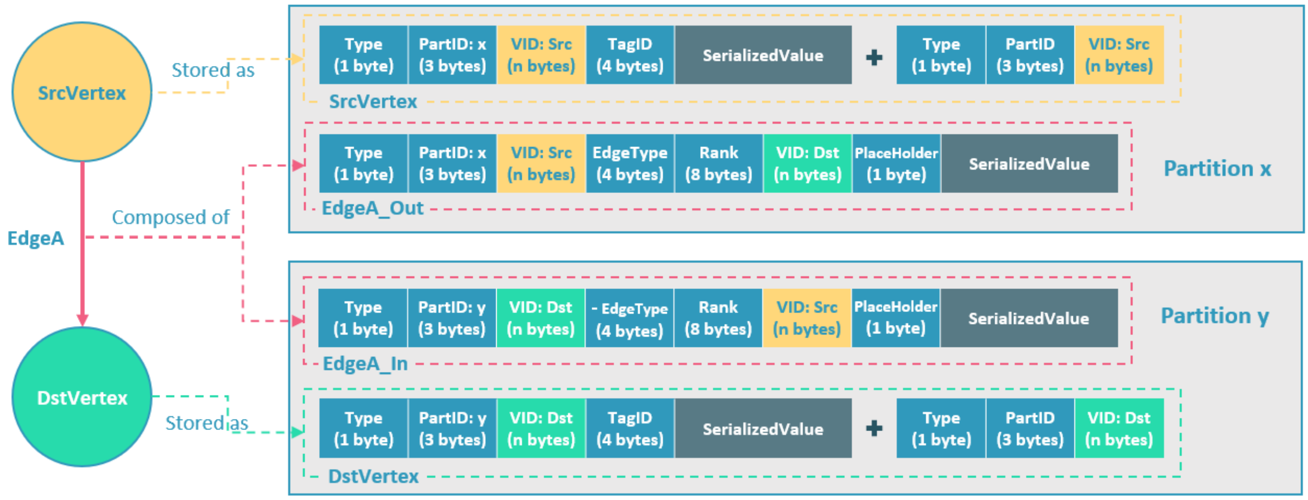
数据分片

由于超大规模关系网络的节点数量高达百亿到千亿，而边的数量更会高达万亿，即使仅存储点和边两者也远大于一般服务器的容量。因此需要有方法将图元素切割，并存储在不同逻辑分片（Partition）上。NebulaGraph 采用边分割的方式。



切边与存储放大

NebulaGraph 中逻辑上的一条边对应着硬盘上的两个键值对（key-value pair），在边的数量和属性较多时，存储放大现象较明显。边的存储方式如下图所示。



上图以最简单的两个点和一条边为例，起点 SrcVertex 通过边 EdgeA 连接目的点 DstVertex，形成路径 `(SrcVertex)-[EdgeA]->(DstVertex)`。这两个点和一条边会以 6 个键值对的形式保存在存储层的两个不同分片，即 Partition x 和 Partition y 中，详细说明如下：

- 点 SrcVertex 的键值保存在 Partition x 中。
- 边 EdgeA 的第一份键值，这里用 EdgeA_Out 表示，与 SrcVertex 一同保存在 Partition x 中。key 的字段有 Type、PartID（x）、VID（Src，即点 SrcVertex 的 ID）、EdgeType（符号为正，代表边方向为出）、Rank（0）、VID（Dst，即点 DstVertex 的 ID）和 PlaceHolder。SerializedValue 即 Value，是序列化的边属性。
- 点 DstVertex 的键值保存在 Partition y 中。
- 边 EdgeA 的第二份键值，这里用 EdgeA_In 表示，与 DstVertex 一同保存在 Partition y 中。key 的字段有 Type、PartID（y）、VID（Dst，即点 DstVertex 的 ID）、EdgeType（符号为负，代表边方向为入）、Rank（0）、VID（Src，即点 SrcVertex 的 ID）和 PlaceHolder。SerializedValue 即 Value，是序列化的边属性，与 EdgeA_Out 中该部分的完全相同。

EdgeA_Out 和 EdgeA_In 以方向相反的两条边的形式存在于存储层，二者组合成了逻辑上的一条边 EdgeA。EdgeA_Out 用于从起点开始的遍历请求，例如 `(a)-[]->()`；EdgeA_In 用于指向目的点的遍历请求，或者说从目的点开始，沿着边的方向逆序进行的遍历请求，例如例如 `()-[]->(a)`。

如 EdgeA_Out 和 EdgeA_In 一样，NebulaGraph 冗余了存储每条边的信息，导致存储边所需的实际空间翻倍。因为边对应的 key 占用的硬盘空间较小，但 value 占用的空间与属性值的长度和数量成正比，所以，当边的属性值较大或数量较多时候，硬盘空间占用量会比较大。

分片算法

分片策略采用静态 Hash 的方式，即对点 VID 进行取模操作，同一个点的所有 Tag、出边和入边信息都会存储到同一个分片，这种方式极大地提升了查询效率。

> Note
>
> 创建图空间时需指定分片数量，分片数量设置后无法修改，建议设置时提前满足业务将来的扩容需求。

多机集群部署时，分片分布在集群内的不同机器上。分片数量在 CREATE SPACE 语句中指定，此后不可更改。

如果需要将某些点放置在相同的分片（例如在一台机器上），可以参考公式或代码。

下文用简单代码说明 VID 和分片的关系。

```
// 如果 ID 长度为 8，为了兼容 1.0，将数据类型视为 int64。
uint64_t vid = 0;
if (id.size() == 8) {
    memcpy(static_cast<void*>(&vid), id.data(), 8);
} else {
    MurmurHash2 hash;
    vid = hash(id.data());
}
PartitionID pId = vid % numParts + 1;
```

简单来说，上述代码是将一个固定的字符串进行哈希计算，转换成数据类型为 int64 的数字（int64 数字的哈希计算结果是数字本身），将数字取模，然后加 1，即：

```
pId = vid % numParts + 1;
```

示例的部分参数说明如下。

参数	说明
`%`	取模运算。
`numParts`	`VID` 所在图空间的分片数，即 CREATE SPACE 语句中的 `partition_num` 值。
`pId`	`VID` 所在分片的 ID。

例如有 100 个分片，`VID` 为 1、101 和 1001 的三个点将会存储在相同的分片。分片 ID 和机器地址之间的映射是随机的，所以不能假定任何两个分片位于同一台机器上。

Raft

关于 RAFT 的简单介绍

分布式系统中，同一份数据通常会有多个副本，这样即使少数副本发生故障，系统仍可正常运行。这就需要一定的技术手段来保证多个副本之间的一致性。

基本原理：Raft 就是一种用于保证多副本一致性的协议。Raft 采用多个副本之间竞选的方式，赢得”超过半数”副本投票的（候选）副本成为 Leader，由 Leader 代表所有副本对外提供服务；其他 Follower 作为备份。当该 Leader 出现异常后（通信故障、运维命令等），其余 Follower 进行新一轮选举，投票出一个新的 Leader。Leader 和 Follower 之间通过心跳的方式相互探测是否存活，并以 Raft-wal 的方式写入硬盘，超过多个心跳仍无响应的副本会被认为发生故障。

> Note
>
> 因为 Raft-wal 需要定期写硬盘，如果硬盘写能力瓶颈会导致 Raft 心跳失败，导致重新发起选举。硬盘 IO 严重堵塞情况下，会导致长期无法选举出 Leader。

读写流程：对于客户端的每个写入请求，Leader 会将该写入以 Raft-wal 的方式，将该条同步给其他 Follower，并只有在“超过半数”副本都成功收到 Raft-wal 后，才会返回客户端该写入成功。对于客户端的每个读取请求，都直接访问 Leader，而 Follower 并不参与读请求服务。

故障流程：场景 1：考虑一个配置为单副本（图空间）的集群；如果系统只有一个副本时，其自身就是 Leader；如果其发生故障，系统将完全不可用。场景 2：考虑一个配置为 3 副本（图空间）的集群；如果系统有 3 个副本，其中一个副本是 Leader，其他 2 个副本是 Follower；即使原 Leader 发生故障，剩下两个副本仍可投票出一个新的 Leader（以及一个 Follower），此时系统仍可使用；但是当这 2 个副本中任一者再次发生故障后，由于投票人数不足，系统将完全不可用。

> Note
>
> Raft 多副本的方式与 HDFS 多副本的方式是不同的，Raft 基于“多数派”投票，因此副本数量不能是偶数。

MULTI GROUP RAFT

由于 Storage 服务需要支持集群分布式架构，所以基于 Raft 协议实现了 Multi Group Raft，即每个分片的所有副本共同组成一个 Raft group，其中一个副本是 leader，其他副本是 follower，从而实现强一致性和高可用性。Raft 的部分实现如下。

由于 Raft 日志不允许空洞，NebulaGraph 使用 Multi Group Raft 缓解此问题，分片数量较多时，可以有效提高 NebulaGraph 的性能。但是分片数量太多会增加开销，例如 Raft group 内部存储的状态信息、WAL 文件，或者负载过低时的批量操作。

实现 Multi Group Raft 有 2 个关键点：

- 共享 Transport 层

  每一个 Raft group 内部都需要向对应的 peer 发送消息，如果不能共享 Transport 层，会导致连接的开销巨大。

- 共享线程池

  如果不共享一组线程池，会造成系统的线程数过多，导致大量的上下文切换开销。

批量（BATCH）操作

NebulaGraph 中，每个分片都是串行写日志，为了提高吞吐，写日志时需要做批量操作，但是由于 NebulaGraph 利用 WAL 实现一些特殊功能，需要对批量操作进行分组，这是 NebulaGraph 的特色。

例如无锁 CAS 操作需要之前的 WAL 全部提交后才能执行，如果一个批量写入的 WAL 里包含了 CAS 类型的 WAL，就需要拆分成粒度更小的几个组，还要保证这几组 WAL 串行提交。

LEADER 切换（TRANSFER LEADERSHIP）

leader 切换对于负载均衡至关重要，当把某个分片从一台机器迁移到另一台机器时，首先会检查分片是不是 leader，如果是的话，需要先切换 leader，数据迁移完毕之后，通常还要重新均衡 leader 分布。

对于 leader 来说，提交 leader 切换命令时，就会放弃自己的 leader 身份，当 follower 收到 leader 切换命令时，就会发起选举。

成员变更

为了避免脑裂，当一个 Raft group 的成员发生变化时，需要有一个中间状态，该状态下新旧 group 的多数派需要有重叠的部分，这样就防止了新的 group 或旧的 group 单方面做出决定。为了更加简化，Diego Ongaro 在自己的博士论文中提出每次只增减一个 peer 的方式，以保证新旧 group 的多数派总是有重叠。 NebulaGraph 也采用了这个方式，只不过增加成员和移除成员的实现有所区别。具体实现方式请参见 Raft Part class 里 addPeer/removePeer 的实现。

与 HDFS 的区别

Storage 服务基于 Raft 协议实现的分布式架构，与 HDFS 的分布式架构有一些区别。例如：

- Storage 服务本身通过 Raft 协议保证一致性，副本数量通常为奇数，方便进行选举 leader，而 HDFS 存储具体数据的 DataNode 需要通过 NameNode 保证一致性，对副本数量没有要求。
- Storage 服务只有 leader 副本提供读写服务，而 HDFS 的所有副本都可以提供读写服务。
- Storage 服务无法修改副本数量，只能在创建图空间时指定副本数量，而 HDFS 可以调整副本数量。
- Storage 服务是直接访问文件系统，而 HDFS 的上层（例如 HBase）需要先访问 HDFS，再访问到文件系统，远程过程调用（RPC）次数更多。

总而言之，Storage 服务更加轻量级，精简了一些功能，架构没有 HDFS 复杂，可以有效提高小块存储的读写性能。

最后更新: 2025年8月4日

3. 快速入门

3.1 基于 Docker 快速部署

NebulaGraph 提供了基于 Docker 的快速部署方式，可以在几分钟内完成部署。

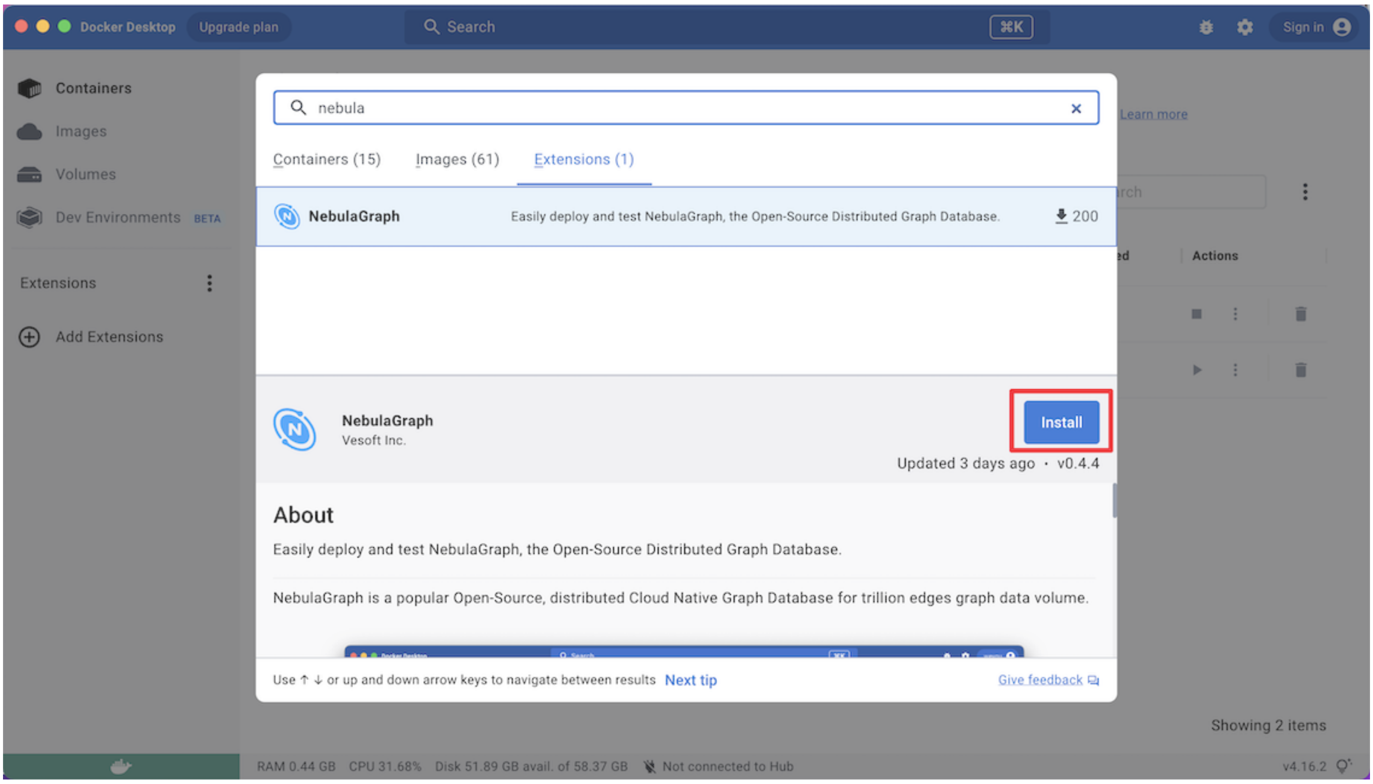
使用 **Docker Desktop**　　使用 **Docker Compose**

按照以下步骤可以快速在 Docker Desktop 中部署 NebulaGraph 。

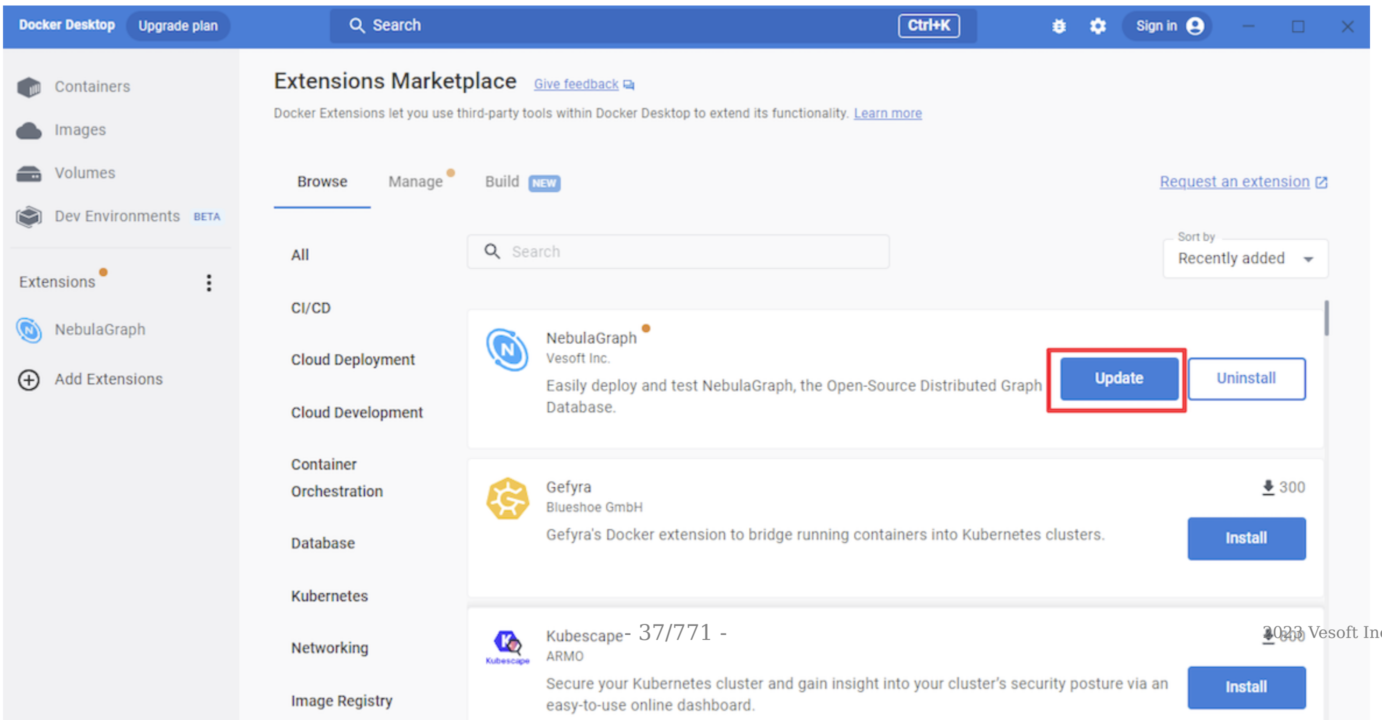
1. 安装 Docker Desktop。

> **Caution**
>
> 不建议在 Windows 上使用 Docker Desktop，因为 Windows 上的 Docker Desktop 性能较差。详情请参见 #12401。如果在 Windows 端安装 Docker Desktop 需安装 WSL 2。

2. 在仪表盘中单击 `Extensions` 或 `Add Extensions` 打开Extensions Marketplace 搜索 NebulaGraph ，也可以点击 NebulaGraph 在 Docker Desktop 打开。
3. 导航到 NebulaGraph 的扩展市场。
4. 点击 `Install` 下载 NebulaGraph 。



5. 在有更新的时候，可以点击 `Update` 更新到最新版本。



最后更新: 2025年8月4日

3.2 从云开始（免费试用）

用户可以在阿里云云市场上免费试用 NebulaGraph 。无需下载和安装任何软件，只需简单的点击操作，即可在云端体验到 NebulaGraph 的强大功能。立即单击免费试用，探索 NebulaGraph 的魅力。

最后更新: 2025年8月4日

3.3 本地部署

3.3.1 安装 NebulaGraph

RPM 和 DEB 是 Linux 系统下常见的两种安装包格式，本文介绍如何使用 RPM 或 DEB 文件在一台机器上快速安装 NebulaGraph 。

> Note
>
> 部署 NebulaGraph 集群的方式参见使用 RPM/DEB 包部署集群。

前提条件

- 安装 `wget` 工具。

下载安装包

> Note
>
> - 当前仅支持在 Linux 系统下安装 NebulaGraph，且仅支持 CentOS 7.x、CentOS 8.x、Ubuntu 16.04、Ubuntu 18.04、Ubuntu 20.04 操作系统。
> - 如果用户使用的是国产化的 Linux 操作系统，请安装企业版 NebulaGraph 。

阿里云 **OSS** 下载

- 下载 release 版本

  URL 格式如下：

```
//Centos 7
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.el7.x86_64.rpm

//Centos 8
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.el8.x86_64.rpm

//Ubuntu 1604
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.ubuntu1604.amd64.deb

//Ubuntu 1804
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.ubuntu1804.amd64.deb

//Ubuntu 2004
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.ubuntu2004.amd64.deb
```

  例如要下载适用于 `Centos 7.5` 的 `master` 安装包：

```
wget https://oss-cdn.nebula-graph.com.cn/package/master/nebula-graph-master.el7.x86_64.rpm
wget https://oss-cdn.nebula-graph.com.cn/package/master/nebula-graph-master.el7.x86_64.rpm.sha256sum.txt
```

  下载适用于 `ubuntu 1804` 的 `master` 安装包：

```
wget https://oss-cdn.nebula-graph.com.cn/package/master/nebula-graph-master.ubuntu1804.amd64.deb
wget https://oss-cdn.nebula-graph.com.cn/package/master/nebula-graph-master.ubuntu1804.amd64.deb.sha256sum.txt
```

- 下载日常开发版本 (nightly)

  Danger

  - nightly 版本通常用于测试新功能、新特性，请不要在生产环境中使用 nightly 版本。
  - nightly 版本不保证每日都能完整发布，也不保证是否会更改文件名。

  URL 格式如下：

```
//Centos 7
https://oss-cdn.nebula-graph.com.cn/package/nightly/<yyyy.mm.dd>/nebula-graph-<yyyy.mm.dd>-nightly.el7.x86_64.rpm

//Centos 8
https://oss-cdn.nebula-graph.com.cn/package/nightly/<yyyy.mm.dd>/nebula-graph-<yyyy.mm.dd>-nightly.el8.x86_64.rpm

//Ubuntu 1604
https://oss-cdn.nebula-graph.com.cn/package/nightly/<yyyy.mm.dd>/nebula-graph-<yyyy.mm.dd>-nightly.ubuntu1604.amd64.deb

//Ubuntu 1804
https://oss-cdn.nebula-graph.com.cn/package/nightly/<yyyy.mm.dd>/nebula-graph-<yyyy.mm.dd>-nightly.ubuntu1804.amd64.deb

//Ubuntu 2004
https://oss-cdn.nebula-graph.com.cn/package/nightly/<yyyy.mm.dd>/nebula-graph-<yyyy.mm.dd>-nightly.ubuntu2004.amd64.deb
```

  例如要下载 `2021.11.24` 适用于 `Centos 7.5` 的 `2.x` 安装包：

```
wget https://oss-cdn.nebula-graph.com.cn/package/nightly/2021.11.24/nebula-graph-2021.11.24-nightly.el7.x86_64.rpm
wget https://oss-cdn.nebula-graph.com.cn/package/nightly/2021.11.24/nebula-graph-2021.11.24-nightly.el7.x86_64.rpm.sha256sum.txt
```

  要下载 `2021.11.24` 适用于 `Ubuntu 1804` 的 `2.x` 安装包：

```
wget https://oss-cdn.nebula-graph.com.cn/package/nightly/2021.11.24/nebula-graph-2021.11.24-nightly.ubuntu1804.amd64.deb
wget https://oss-cdn.nebula-graph.com.cn/package/nightly/2021.11.24/nebula-graph-2021.11.24-nightly.ubuntu1804.amd64.deb.sha256sum.txt
```

安装 NebulaGraph

- 安装 RPM 包

```
$ sudo rpm -ivh --prefix=<installation_path> <package_name>
```

  `--prefix` 为可选项，用于指定安装路径。如不设置，系统会将 NebulaGraph 安装到默认路径 `/usr/local/nebula/`。

  例如，要在默认路径下安装master版本的 RPM 包，运行如下命令：

```
sudo rpm -ivh nebula-graph-master.el7.x86_64.rpm
```

- 安装 DEB 包

```
$ sudo dpkg -i <package_name>
```

  Note

  使用 DEB 包安装 NebulaGraph 时不支持自定义安装路径。默认安装路径为 `/usr/local/nebula/`。

  例如安装master版本的 DEB 包：

```
sudo dpkg -i nebula-graph-master.ubuntu1804.amd64.deb
```

后续操作

- 启动 NebulaGraph

- 连接 NebulaGraph

最后更新: 2025年8月4日

3.3.2 启动 NebulaGraph 服务

NebulaGraph 支持通过脚本管理服务。

使用脚本管理服务

使用脚本 `nebula.service` 管理服务，包括启动、停止、重启、中止和查看。

> Note
>
> `nebula.service` 的默认路径是 `/usr/local/nebula/scripts` ，如果修改过安装路径，请使用实际路径。

语法

```
$ sudo /usr/local/nebula/scripts/nebula.service
[-v] [-c <config_file_path>]
<start | stop | restart | kill | status>
<metad | graphd | storaged | all>
```

参数	说明
`-v`	显示详细调试信息。
`-c`	指定配置文件路径，默认路径为 `/usr/local/nebula/etc/` 。
`start`	启动服务。
`stop`	停止服务。
`restart`	重启服务。
`kill`	中止服务。
`status`	查看服务状态。
`metad`	管理 Meta 服务。
`graphd`	管理 Graph 服务。
`storaged`	管理 Storage 服务。
`all`	管理所有服务。

启动 **NebulaGraph** 服务

执行如下命令启动服务：

```
$ sudo /usr/local/nebula/scripts/nebula.service start all
[INFO] Starting nebula-metad...
[INFO] Done
[INFO] Starting nebula-graphd...
[INFO] Done
[INFO] Starting nebula-storaged...
[INFO] Done
```

停止 **NebulaGraph** 服务

> Danger
>
> 请勿使用 `kill -9` 命令强制终止进程，否则可能较小概率出现数据丢失。

执行如下命令停止 NebulaGraph 服务：

```
$ sudo /usr/local/nebula/scripts/nebula.service stop all
[INFO] Stopping nebula-metad...
[INFO] Done
[INFO] Stopping nebula-graphd...
[INFO] Done
[INFO] Stopping nebula-storaged...
[INFO] Done
```

查看 NebulaGraph 服务

执行如下命令查看 NebulaGraph 服务状态：

```
$ sudo /usr/local/nebula/scripts/nebula.service status all
```

- 如果返回如下结果，表示 NebulaGraph 服务正常运行。

```
[INFO] nebula-metad(33fd35e): Running as 29020, Listening on 9559
[INFO] nebula-graphd(33fd35e): Running as 29095, Listening on 9669
[WARN] nebula-storaged after v3.0.0 will not start service until it is added to cluster.
[WARN] See Manage Storage hosts:ADD HOSTS in https://docs.nebula-graph.io/
[INFO] nebula-storaged(33fd35e): Running as 29147, Listening on 9779
```

> Note
>
> 正常启动 NebulaGraph 后，`nebula-storaged` 进程的端口显示红色。这是因为 `nebula-storaged` 在启动流程中会等待 `nebula-metad` 添加当前 Storage 服务，当前 Storage 服务收到 Ready 信号后才会正式启动服务。从 3.0.0 版本开始，在配置文件中添加的 Storage 节点无法直接读写，配置文件的作用仅仅是将 Storage 节点注册至 Meta 服务中。必须使用 `ADD HOSTS` 命令后，才能正常读写 Storage 节点。更多信息，参见管理 Storage 主机。

- 如果返回类似如下结果，表示 NebulaGraph 服务异常，可以根据异常服务信息进一步排查，或者在 NebulaGraph 社区寻求帮助。

```
[INFO] nebula-metad: Running as 25600, Listening on 9559
[INFO] nebula-graphd: Exited
[INFO] nebula-storaged: Running as 25646, Listening on 9779
```

NebulaGraph 服务由 Meta 服务、Graph 服务和 Storage 服务共同提供，这三种服务的配置文件都保存在安装目录的 `etc` 目录内，默认路径为 `/usr/local/nebula/etc/`，用户可以检查相应的配置文件排查问题。

下一步

- 连接 NebulaGraph

最后更新: 2025年8月4日

3.3.3 连接 NebulaGraph

本文介绍如何使用原生命令行客户端 NebulaGraph Console 连接 NebulaGraph 。

> **Caution**
>
> 首次连接到 NebulaGraph 后，必须先注册 Storage 服务，才能正常查询数据。

NebulaGraph 支持多种类型的客户端，包括命令行客户端、可视化界面客户端和流行编程语言客户端。详情参见客户端列表。

前提条件

- NebulaGraph 服务已启动。
- 运行 NebulaGraph Console 的机器和运行 NebulaGraph 的服务器网络互通。
- NebulaGraph Console 的版本兼容 NebulaGraph 的版本。

> **Note**
>
> 版本相同的 NebulaGraph Console 和 NebulaGraph 兼容程度最高，版本不同的 NebulaGraph Console 连接 NebulaGraph 时，可能会有兼容问题，或者无法连接并报错 `incompatible version between client and server` 。

操作步骤

1. 在 NebulaGraph Console 下载页面，确认需要的版本，单击 Assets。

> **Note**
>
> 建议选择最新版本。

2. 在 Assets 区域找到机器运行所需的二进制文件，下载文件到机器上。
3. （可选）为方便使用，重命名文件为 `nebula-console` 。

> **Note**
>
> 在 Windows 系统中，请重命名为 `nebula-console.exe` 。

4. 在运行 NebulaGraph Console 的机器上执行如下命令，为用户授予 nebula-console 文件的执行权限。

> **Note**
>
> Windows 系统请跳过此步骤。

```
$ chmod 111 nebula-console
```

5. 在命令行界面中，切换工作目录至 nebula-console 文件所在目录。

6. 执行如下命令连接 NebulaGraph 。

- Linux 或 macOS

```
$ ./nebula-console -addr <ip> -port <port> -u <username> -p <password>
  [-t 120] [-e "nGQL_statement" | -f filename.nGQL]
```

- Windows

```
> nebula-console.exe -addr <ip> -port <port> -u <username> -p <password>
  [-t 120] [-e "nGQL_statement" | -f filename.nGQL]
```

参数说明如下。

参数	说明
`-h/-help`	显示帮助菜单。
`-addr/-address`	设置要连接的 Graph 服务的 IP 或主机名。默认地址为 127.0.0.1。
`-P/-port`	设置要连接的 Graph 服务的端口。默认端口为 9669。
`-u/-user`	设置 NebulaGraph 账号的用户名。未启用身份认证时，可以使用任意已存在的用户名（默认为 `root` ）。
`-p/-password`	设置用户名对应的密码。未启用身份认证时，密码可以填写任意字符。
`-t/-timeout`	设置整数类型的连接超时时间。单位为毫秒，默认值为 120。
`-e/-eval`	设置字符串类型的 nGQL 语句。连接成功后会执行一次该语句并返回结果，然后自动断开连接。
`-f/-file`	设置存储 nGQL 语句的文件的路径。连接成功后会执行该文件内的 nGQL 语句并返回结果，执行完毕后自动断开连接。
`-enable_ssl`	连接 NebulaGraph 时使用 SSL 加密。
`-ssl_root_ca_path`	指定 CA 证书的存储路径。
`-ssl_cert_path`	指定 CRT 证书的存储路径。
`-ssl_private_key_path`	指定私钥文件的存储路径。

更多参数参见项目仓库。

最后更新: 2025年8月4日

3.3.4 注册 Storage 服务

首次连接到 NebulaGraph 后，需要先添加 Storage 主机，并确认主机都处于在线状态。

> **历史版本兼容性**
>
> 从 NebulaGraph 3.0.0 版本开始，必须先使用 `ADD HOSTS` 添加主机，才能正常通过 Storage 服务读写数据。

前提条件

已连接 NebulaGraph 服务。

操作步骤

1. 添加 Storage 主机。

   执行如下命令添加主机：

```
ADD HOSTS <ip>:<port> [,<ip>:<port> ...];
```

   示例：

```
nebula> ADD HOSTS 192.168.10.100:9779, 192.168.10.101:9779, 192.168.10.102:9779;
```

> **Caution**
>
> 请确保添加的主机 IP 和配置文件 `nebula-storaged.conf` 中 `local_ip` 配置的 IP 一致，否则会导致添加 Storage 主机失败。关于配置文件的详情，参见配置管理。

2. 检查主机状态，确认全部在线。

```
nebula> SHOW HOSTS;
+------------------+------+----------+--------------+---------------------  +------------------------+---------+
| Host             | Port | Status   | Leader count | Leader distribution |   Partition distribution | Version |
+------------------+------+----------+--------------+---------------------  +------------------------+---------+
| "192.168.10.100" | 9779 | "ONLINE" | 0            | "No valid partition" | "No   valid partition"   | "master" |
| "192.168.10.101" | 9779 | "ONLINE" | 0            | "No valid partition" | "No   valid partition"   | "master" |
| "192.168.10.102" | 9779 | "ONLINE" | 0            | "No valid partition" | "No   valid partition"   | "master" |
+------------------+------+----------+--------------+---------------------  +------------------------+---------+
```

   在返回结果的 Status 列，可以看到所有 Storage 主机都在线。

最后更新: 2025年8月4日

3.3.5 使用常用 nGQL（CRUD 命令）

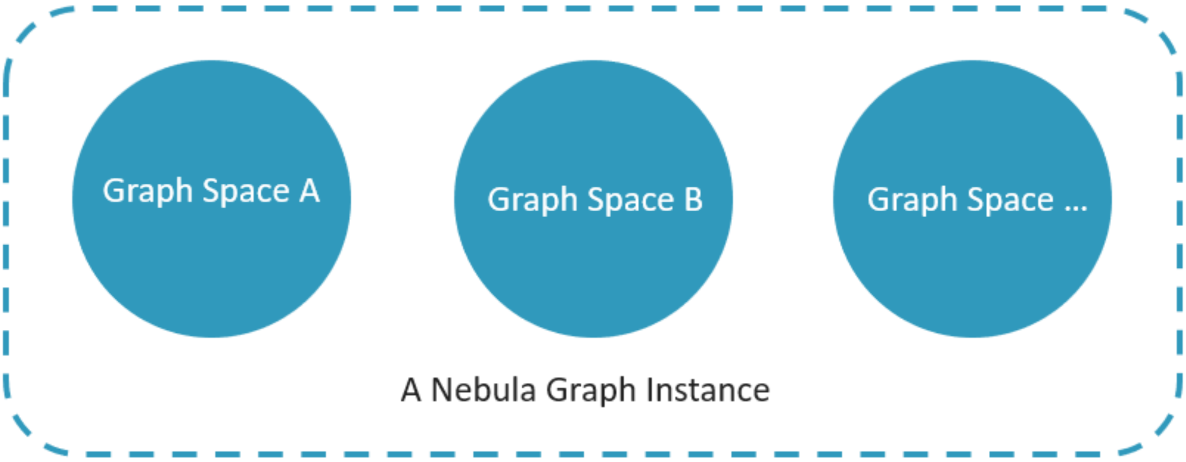
本文介绍 NebulaGraph 查询语言的基础语法，包括用于 Schema 创建和常用增删改查操作的语句。

如需了解更多语句的用法，参见 nGQL 指南。

使用说明

图空间和 SCHEMA

一个 NebulaGraph 实例由一个或多个图空间组成。每个图空间都是物理隔离的，用户可以在同一个实例中使用不同的图空间存储不同的数据集。

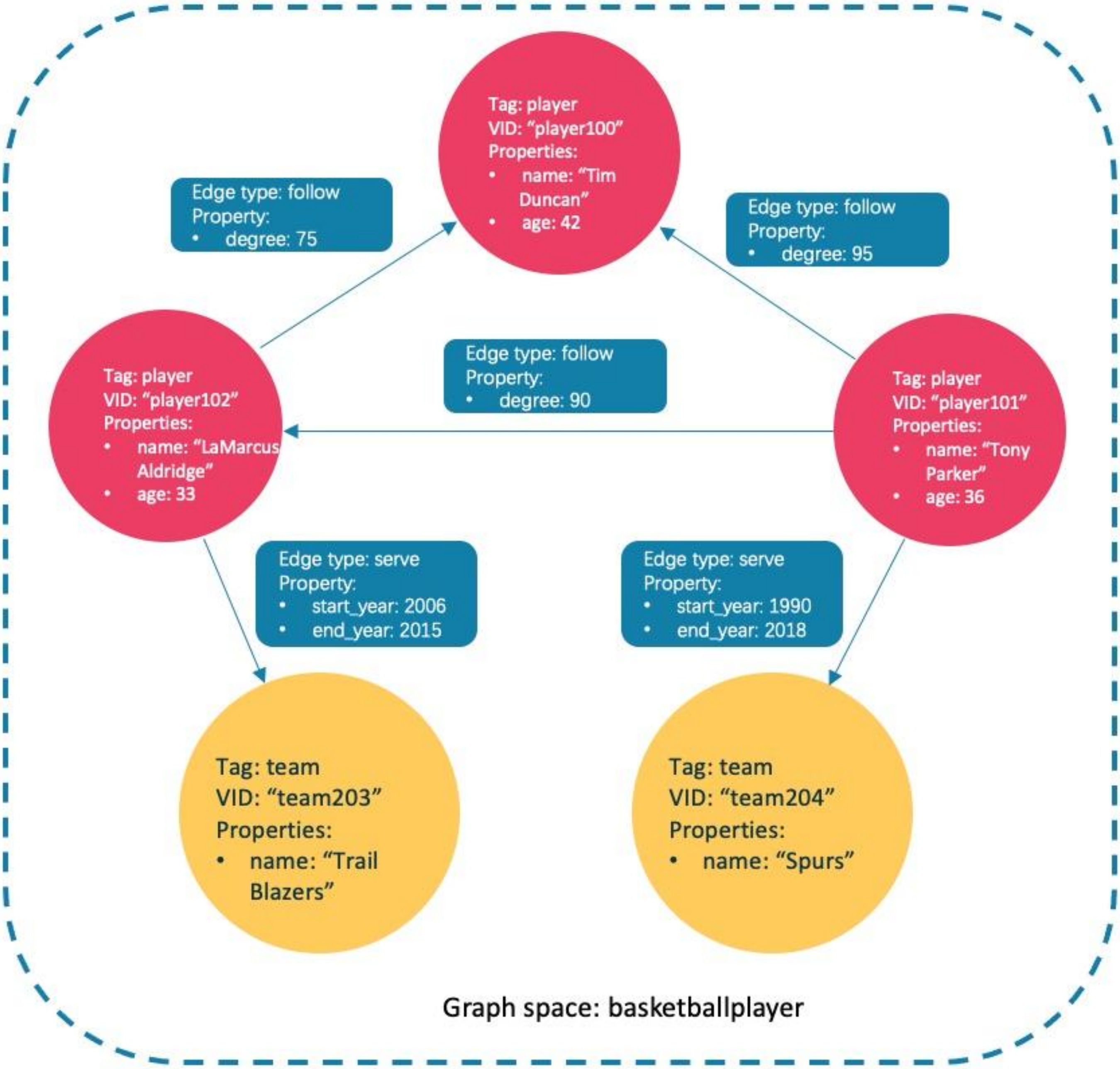


为了在图空间中插入数据，需要为图数据库定义一个 Schema。 NebulaGraph 的 Schema 是由如下几部分组成。

组成部分	说明
点（Vertex）	表示现实世界中的实体。一个点可以有 0 到多个标签。
标签（Tag）	点的类型，定义了一组描述点类型的属性。
边（Edge）	表示两个点之间有方向的关系。
边类型（Edge type）	边的类型，定义了一组描述边的类型的属性。

更多信息，请参见数据结构。

本文将使用下图的数据集演示基础操作的语法。


Tag: player
VID: "player100"
Properties:
• name: "Tim Duncan"
• age: 42
Edge type: follow
Property:
• degree: 75
Edge type: follow
Property:
• degree: 95
Edge type: follow
Property:
• degree: 90
Tag: player
VID: "player102"
Properties:
• name: "LaMarcus Aldridge"
• age: 33
Tag: player
VID: "player101"
Properties:
• name: "Tony Parker"
• age: 36
Edge type: serve
Property:
• start_year: 2006
• end_year: 2015
Edge type: serve
Property:
• start_year: 1990
• end_year: 2018
Tag: team
VID: "team203"
Properties:
• name: "Trail Blazers"
Tag: team
VID: "team204"
Properties:
• name: "Spurs"
Graph space: basketballplayer

异步实现创建和修改

> Caution
>
> 在 NebulaGraph 中，下列创建和修改操作是异步实现的。要在下一个心跳周期之后才能生效，否则访问会报错。为确保数据同步，后续操作能顺利进行，请等待 2 个心跳周期（20 秒）。

- `CREATE SPACE`
- `CREATE TAG`
- `CREATE EDGE`
- `ALTER TAG`
- `ALTER EDGE`
- `CREATE TAG INDEX`
- `CREATE EDGE INDEX`

> Note
>
> 默认心跳周期是 10 秒。修改心跳周期参数 `heartbeat_interval_secs`，请参见配置简介。

第一步：创建和选择图空间

NGQL 语法

- 创建图空间

```
CREATE SPACE [IF NOT EXISTS] <graph_space_name> (
[partition_num = <partition_number>,]
[replica_factor = <replica_number>,]
vid_type = {FIXED_STRING(<N>) | INT64}
)

[COMMENT = '<comment>'];
```

参数详情请参见 CREATE SPACE。

- 列出创建成功的图空间

```
nebula> SHOW SPACES;
```

- 选择已创建的图空间

```
USE <graph_space_name>;
```

示例

1. 执行如下语句创建名为 `basketballplayer` 的图空间。

```
nebula> CREATE SPACE basketballplayer(partition_num=15, replica_factor=1, vid_type=fixed_string(30));
```

> Note
>
> 如果报错提示 `[ERROR (-1005)]: Host not enough!`，请检查是否已添加 Storage 主机。

2. 执行命令 `SHOW HOSTS` 检查分片的分布情况，确保平衡分布。

```
nebula> SHOW HOSTS;
+-------------+-----------+-----------+--------------+----------------------------------+------------------------+---------+
| Host        | Port      | Status    | Leader count | Leader distribution              | Partition distribution | Version |
```

```
+------------+-----------+----------+-------------+---------------------------------+-----------------------+---------+
| "storaged0" | 9779      | "ONLINE"  | 5           | "basketballplayer:5"            | "basketballplayer:5"   | "master" |
| "storaged1" | 9779      | "ONLINE"  | 5           | "basketballplayer:5"            | "basketballplayer:5"   | "master" |
| "storaged2" | 9779      | "ONLINE"  | 5           | "basketballplayer:5"            | "basketballplayer:5"   | "master" |
+------------+-----------+----------+-------------+---------------------------------+-----------------------+---------+
```

如果 Leader distribution 分布不均匀，请执行命令 `BALANCE LEADER` 重新分配。更多信息，请参见 Storage 负载均衡。

3. 选择图空间 `basketballplayer` 。

```
nebula[(none)]> USE basketballplayer;
```

用户可以执行命令 `SHOW SPACES` 查看创建的图空间。

```
nebula> SHOW SPACES;
+--------------------+
| Name               |
+--------------------+
| "basketballplayer" |
+--------------------+
```

第二步：创建 **Tag** 和 **Edge type**

NGQL 语法

```
CREATE {TAG | EDGE} [IF NOT EXISTS] {<tag_name> | <edge_type_name>}
    (
      <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']
      [{, <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']} ...]
    )
    [TTL_DURATION = <ttl_duration>]
    [TTL_COL = <prop_name>]
    [COMMENT = '<comment>'];
```

参数详情请参见 CREATE TAG 和 CREATE EDGE。

示例

创建 Tag: `player` 和 `team` ，以及 Edge type: `follow` 和 `serve` 。说明如下表。

名称	类型	属性
player	Tag	name (string), age (int)
team	Tag	name (string)
follow	Edge type	degree (int)
serve	Edge type	start_year (int), end_year (int)

```
nebula> CREATE TAG player(name string, age int);

nebula> CREATE TAG team(name string);

nebula> CREATE EDGE follow(degree int);

nebula> CREATE EDGE serve(start_year int, end_year int);
```

第三步：插入数据

用户可以使用 `INSERT` 语句，基于现有的 Tag 插入点，或者基于现有的 Edge type 插入边。

NGQL 语法

- 插入点

```
INSERT VERTEX [IF NOT EXISTS] [tag_props, [tag_props] ...]
VALUES <vid>: ([prop_value_list])

tag_props:
  tag_name ([prop_name_list])

prop_name_list:
   [prop_name [, prop_name] ...]
```

```
prop_value_list:
   [prop_value [, prop_value] ...]
```

`vid` 是 Vertex ID 的缩写，`vid` 在一个图空间中是唯一的。参数详情请参见 INSERT VERTEX。

- 插入边

```
INSERT EDGE [IF NOT EXISTS] <edge_type> ( <prop_name_list> ) VALUES
<src_vid> -> <dst_vid>[@<rank>] : ( <prop_value_list> )
[, <src_vid> -> <dst_vid>[@<rank>] : ( <prop_value_list> ), ...];

<prop_name_list> ::=
[ <prop_name> [, <prop_name> ] ...]

<prop_value_list> ::=
[ <prop_value> [, <prop_value> ] ...]
```

参数详情请参见 INSERT EDGE。

示例

- 插入代表球员和球队的点。

```
nebula> INSERT VERTEX player(name, age) VALUES "player100":("Tim Duncan", 42);

nebula> INSERT VERTEX player(name, age) VALUES "player101":("Tony Parker", 36);

nebula> INSERT VERTEX player(name, age) VALUES "player102":("LaMarcus Aldridge", 33);

nebula> INSERT VERTEX team(name) VALUES "team203":("Trail Blazers"), "team204":("Spurs");
```

- 插入代表球员和球队之间关系的边。

```
nebula> INSERT EDGE follow(degree) VALUES "player101" -> "player100":(95);

nebula> INSERT EDGE follow(degree) VALUES "player101" -> "player102":(90);

nebula> INSERT EDGE follow(degree) VALUES "player102" -> "player100":(75);

nebula> INSERT EDGE serve(start_year, end_year) VALUES "player101" -> "team204":(1999, 2018),"player102" -> "team203":(2006,  2015);
```

第四步：查询数据

- GO 语句可以根据指定的条件遍历数据库。`GO` 语句从一个或多个点开始，沿着一条或多条边遍历，返回 `YIELD` 子句中指定的信息。
- FETCH 语句可以获得点或边的属性。
- LOOKUP 语句是基于索引的，和 `WHERE` 子句一起使用，查找符合特定条件的数据。
- MATCH 语句是查询图数据最常用的，可以灵活的描述各种图模式，但是它依赖索引去匹配 NebulaGraph 中的数据模型，性能也还需要调优。

NGQL 语法

- `GO`

```
GO [[<M> TO] <N> {STEP|STEPS}] FROM <vertex_list>
OVER <edge_type_list> [{REVERSELY | BIDIRECT}]
[ WHERE <conditions> ]
YIELD [DISTINCT] <return_list>
[{ SAMPLE <sample_list> | <limit_by_list_clause> }]
[| GROUP BY {<col_name> | expression> | <position>} YIELD <col_name>]
```

```
[| ORDER BY <expression> [{ASC | DESC}]]
[| LIMIT [<offset>,] <number_rows>];
```

- `FETCH`
- 查询 Tag 属性

```
FETCH PROP ON {<tag_name>[, tag_name ...] | *}
<vid> [, vid ...]
YIELD <return_list> [AS <alias>];
```

- 查询边属性

```
FETCH PROP ON <edge_type> <src_vid> -> <dst_vid>[@<rank>] [, <src_vid> -> <dst_vid> ...]
YIELD <output>;
```

- `LOOKUP`

```
LOOKUP ON {<vertex_tag> | <edge_type>}
[WHERE <expression> [AND <expression> ...]]
YIELD <return_list> [AS <alias>];

<return_list>
    <prop_name> [AS <col_alias>] [, <prop_name> [AS <prop_alias>] ...];
```

- `MATCH`

```
MATCH <pattern> [<clause_1>]  RETURN <output>  [<clause_2>];
```

`GO` 语句示例

- 从 VID 为 `player101` 的球员开始，沿着边 `follow` 找到连接的球员。

```
nebula> GO FROM "player101" OVER follow YIELD id($$);
+-------------+
| id($$)      |
+-------------+
| "player100" |
```

```
| "player102" |
+-------------+
```

- 从 VID 为 `player101` 的球员开始，沿着边 `follow` 查找年龄大于或等于 35 岁的球员，并返回他们的姓名和年龄，同时重命名对应的列。

```
nebula> GO FROM "player101" OVER follow WHERE properties($$).age >= 35 \
        YIELD properties($$).name AS Teammate, properties($$).age AS Age;
+-----------------+-----+
| Teammate        | Age |
+-----------------+-----+
| "Tim Duncan"    | 42  |
+-----------------+-----+
```

子句/符号	说明
`YIELD`	指定该查询需要返回的值或结果。
`$$`	表示边的终点。
`\`	表示换行继续输入。

- 从 VID 为 `player101` 的球员开始，沿着边 `follow` 查找连接的球员，然后检索这些球员的球队。为了合并这两个查询请求，可以使用管道符或临时变量。

- 使用管道符

```
nebula> GO FROM "player101" OVER follow YIELD dst(edge) AS id | \
        GO FROM $-.id OVER serve YIELD properties($$).name AS Team, \
        properties($^).name AS Player;
+-----------------+---------------------+
| Team            | Player              |
+-----------------+---------------------+
| "Trail Blazers" | "LaMarcus Aldridge" |
+-----------------+---------------------+
```

子句/符号	说明
`$^`	表示边的起点。
`|`	组合多个查询的管道符，将前一个查询的结果集用于后一个查询。
`$-`	表示管道符前面的查询输出的结果集。

- 使用临时变量

> Note
>
> 当复合语句作为一个整体提交给服务器时，其中的临时变量会在语句结束时被释放。

```
nebula> $var = GO FROM "player101" OVER follow YIELD dst(edge) AS id; \
        GO FROM $var.id OVER serve YIELD properties($$).name AS Team, \
        properties($^).name AS Player;
+-----------------+---------------------+
| Team            | Player              |
+-----------------+---------------------+
| "Trail Blazers" | "LaMarcus Aldridge" |
+-----------------+---------------------+
```

`FETCH` 语句示例

查询 VID 为 `player100` 的球员的属性。

```
nebula> FETCH PROP ON player "player100" YIELD properties(vertex);
+-------------------------------+
| properties(VERTEX)            |
+-------------------------------+
| {age: 42, name: "Tim Duncan"} |
+-------------------------------+
```

> Note
>
> `LOOKUP` 和 `MATCH` 的示例在下文的索引 部分查看。

其他操作

修改点和边

用户可以使用 `UPDATE` 语句或 `UPSERT` 语句修改现有数据。

`UPSERT` 是 `UPDATE` 和 `INSERT` 的结合体。当使用 `UPSERT` 更新一个点或边，如果它不存在，数据库会自动插入一个新的点或边。

> Note
>
> 每个 partition 内部， `UPSERT` 操作是一个串行操作，所以执行速度比执行 `INSERT` 或 `UPDATE` 慢很多。其仅在多个 partition 之间有并发。

nGQL 语法

- `UPDATE` 点

```
UPDATE VERTEX <vid> SET <properties to be updated>
[WHEN <condition>] [YIELD <columns>];
```

- `UPDATE` 边

```
UPDATE EDGE ON <edge_type> <source vid> -> <destination vid> [@rank]
SET <properties to be updated> [WHEN <condition>] [YIELD <columns to be output>];
```

- `UPSERT` 点或边

```
UPSERT {VERTEX <vid> | EDGE <edge_type>} SET <update_columns>
[WHEN <condition>] [YIELD <columns>];
```

示例

- 用 `UPDATE` 修改 VID 为 `player100` 的球员的 `name` 属性，然后用 `FETCH` 语句检查结果。

```
nebula> UPDATE VERTEX "player100" SET player.name = "Tim";

nebula> FETCH PROP ON player "player100" YIELD properties(vertex);
+------------------------+
| properties(VERTEX)     |
+------------------------+
```

```
| {age: 42, name: "Tim"} |
+------------------------+
```

- 用 `UPDATE` 修改某条边的 `degree` 属性，然后用 `FETCH` 检查结果。

```
nebula> UPDATE EDGE ON follow "player101" -> "player100" SET degree = 96;

nebula> FETCH PROP ON follow "player101" -> "player100" YIELD properties(edge);
+------------------+
| properties(EDGE) |
+------------------+
| {degree: 96}     |
+------------------+
```

- 用 `INSERT` 插入一个 VID 为 `player111` 的点，然后用 `UPSERT` 更新它。

```
nebula> INSERT VERTEX player(name,age) VALUES "player111":("David West", 38);

nebula> UPSERT VERTEX "player111" SET player.name = "David", player.age = $^.player.age + 11 \
        WHEN $^.player.name == "David West" AND $^.player.age > 20 \
        YIELD $^.player.name AS Name, $^.player.age AS Age;
+---------+-----+
| Name    | Age |
+---------+-----+
| "David" | 49  |
+---------+-----+
```

删除点和边

nGQL 语法

- 删除点

```
DELETE VERTEX <vid1>[, <vid2>...]
```

- 删除边

```
DELETE EDGE <edge_type> <src_vid> -> <dst_vid>[@<rank>]
[, <src_vid> -> <dst_vid>...]
```

示例

- 删除点

```
nebula> DELETE VERTEX "player111", "team203";
```

- 删除边

```
nebula> DELETE EDGE follow "player101" -> "team204";
```

使用索引

用户可以通过 CREATE INDEX 语句为 Tag 和 Edge type 增加索引。

> **Caution**
>
> `MATCH` 和 `LOOKUP` 语句的执行都依赖索引，但是索引会导致写性能大幅降低。请不要随意在生产环境中使用索引，除非很清楚使用索引对业务的影响。
>
> 必须为“已写入但未构建索引”的数据重建索引，否则无法在 `MATCH` 和 `LOOKUP` 语句中返回这些数据。参见重建索引。

nGQL 语法

- 创建索引

```
CREATE {TAG | EDGE} INDEX [IF NOT EXISTS] <index_name>
ON {<tag_name> | <edge_name>} ([<prop_name_list>]) [COMMENT = '<comment>'];
```

- 重建索引

```
REBUILD {TAG | EDGE} INDEX <index_name>;
```

> Note
>
> 为没有指定长度的变量属性创建索引时，需要指定索引长度。在 utf-8 编码中，一个中文字符占 3 字节，请根据变量属性长度设置合适的索引长度。例如 10 个中文字符，索引长度需要为 30。详情请参见创建索引。

基于索引的 **LOOKUP** 和 **MATCH** 示例

确保 `LOOKUP` 或 `MATCH` 有一个索引可用。如果没有，请先创建索引。

找到 Tag 为 `player` 的点的信息，它的 `name` 属性值为 `Tony Parker`。

```
// 为 name 属性创建索引 player_index_1。
nebula> CREATE TAG INDEX IF NOT EXISTS player_index_1 ON player(name(20));

// 重建索引确保能对已存在数据生效。
nebula> REBUILD TAG INDEX player_index_1
+------------+
| New Job Id |
+------------+
| 31         |
+------------+

// 使用 LOOKUP 语句检索点的属性。
nebula> LOOKUP ON player WHERE player.name == "Tony Parker" \
        YIELD properties(vertex).name AS name, properties(vertex).age AS age;
+---------------+-----+
| name          | age |
+---------------+-----+
| "Tony Parker" | 36  |
+---------------+-----+

// 使用 MATCH 语句检索点的属性。
nebula> MATCH (v:player{name:"Tony Parker"}) RETURN v;
+-----------------------------------------------------+
| v                                                   |
+-----------------------------------------------------+
| ("player101" :player{age: 36, name: "Tony Parker"}) |
+-----------------------------------------------------+
```

最后更新: 2025年8月4日

3.4 nGQL 命令汇总

3.4.1 函数

- 数学函数

函数	说明
double abs(double x)	返回 x 的绝对值。
double floor(double x)	返回小于或等于 x 的最大整数。
double ceil(double x)	返回大于或等于 x 的最小整数。
double round(double x)	返回离 x 最近的整数值，如果 x 恰好在中间，则返回离 0 较远的整数。
double sqrt(double x)	返回 x 的平方根。
double cbrt(double x)	返回 x 的立方根。
double hypot(double x, double y)	返回直角三角形（直角边长为 x 和 y）的斜边长。
double pow(double x, double y)	返回 x^y 的值。
double exp(double x)	返回 e^x 的值。
double exp2(double x)	返回 2^x 的值。
double log(double x)	返回以自然数 e 为底 x 的对数。
double log2(double x)	返回以 2 为底 x 的对数。
double log10(double x)	返回以 10 为底 x 的对数。
double sin(double x)	返回 x 的正弦值。
double asin(double x)	返回 x 的反正弦值。
double cos(double x)	返回 x 的余弦值。
double acos(double x)	返回 x 的反余弦值。
double tan(double x)	返回 x 的正切值。
double atan(double x)	返回 x 的反正切值。
double rand()	返回 [0,1) 内的随机浮点数。
int rand32(int min, int max)	返回 `[min, max)` 内的一个随机 32 位整数。 用户可以只传入一个参数，该参数会判定为 `max`，此时 `min` 默认为 `0`。 如果不传入参数，此时会从带符号的 32 位 int 范围内随机返回。
int rand64(int min, int max)	返回 `[min, max)` 内的一个随机 64 位整数。 用户可以只传入一个参数，该参数会判定为 `max`，此时 `min` 默认为 `0`。 如果不传入参数，此时会从带符号的 64 位 int 范围内随机返回。
bit_and()	逐位做 AND 操作。
bit_or()	逐位做 OR 操作。
bit_xor()	逐位做 XOR 操作。
int size()	返回列表或映射中元素的数量，或字符串的长度。
int range(int start, int end, int step)	返回 `[start,end]` 中指定步长的值组成的列表。步长 `step` 默认为 1。
int sign(double x)	返回 x 的正负号。 如果 x 为 `0`，则返回 `0`。 如果 x 为负数，则返回 `-1`。 如果 x 为正数，则返回 `1`。
double e()	返回自然对数的底 e（2.718281828459045）。
double pi()	返回数学常数π（3.141592653589793）。
double radians()	将角度转换为弧度。 `radians(180)` 返回 `3.141592653589793`。

- 聚合函数

函数	说明
avg()	返回参数的平均值。
count()	语法：`count({expr | *})` 。 `count()` 返回总行数（包括 NULL）。 `count(expr)` 返回满足表达式的非空值的总数。 `count()` 和 `size()` 是不同的。
max()	返回参数的最大值。
min()	返回参数的最小值。
collect()	`collect()` 函数返回一个符合表达式返回结果的列表。该函数可以将多条记录或值合并进一个列表，实现数据聚合。
std()	返回参数的总体标准差。
sum()	返回参数的和。

- 字符串函数

函数	说明
int strcasecmp(string a, string b)	比较两个字符串（不区分大小写）。当 a=b 时，返回 0，当 a>b 是，返回大于 0 的数，当 a<b 时，返回小于 0 的数。
string lower(string a)	返回小写形式的字符串。
string toLower(string a)	和 `lower()` 相同。
string upper(string a)	返回大写形式的字符串。
string toUpper(string a)	和 `upper()` 相同。
int length(a)	返回给定字符串的长度或路径的长度，单位分别是字节和跳数。
string trim(string a)	删除字符串头部和尾部的空格。
string ltrim(string a)	删除字符串头部的空格。
string rtrim(string a)	删除字符串尾部的空格。
string left(string a, int count)	返回字符串左侧 `count` 个字符组成的子字符串。如果 `count` 超过字符串 a 的长度，则返回字符串 a。
string right(string a, int count)	返回字符串右侧 `count` 个字符组成的子字符串。如果 `count` 超过字符串 a 的长度，则返回字符串 a。
string lpad(string a, int size, string letters)	在字符串 a 的左侧填充 `letters` 字符串，并返回 `size` 长度的字符串。
string rpad(string a, int size, string letters)	在字符串 a 的右侧填充 `letters` 字符串，并返回 `size` 长度的字符串。
string substr(string a, int pos, int count)	从字符串 a 的指定位置 `pos` 开始（不包括 `pos` 位置的字符），提取右侧的 `count` 个字符，组成新的字符串并返回。
string substring(string a, int pos, int count)	和 `substr()` 相同。
string reverse(string)	逆序返回字符串。
string replace(string a, string b, string c)	将字符串 a 中的子字符串 b 替换为字符串 c。
list split(string a, string b)	在子字符串 b 处拆分字符串 a，返回一个字符串列表。
concat()	`concat()` 函数至少需要两个或以上字符串参数，并将所有参数连接成一个字符串。 语法：`concat(string1,string2,...)`
concat_ws()	`concat_ws()` 函数将两个或以上字符串参数与预定义的分隔符（separator）相连接。
extract()	`extract()` 从指定字符串中提取符合正则表达式的子字符串。
json_extract()	`json_extract()` 将指定 JSON 字符串转换为 map 类型。

- 日期时间函数

函数	说明
int now()	根据当前系统返回当前时间戳。
timestamp timestamp()	根据当前系统返回当前时间戳。
date date()	根据当前系统返回当前日期（UTC 时间）。
time time()	根据当前系统返回当前时间（UTC 时间）。
datetime datetime()	根据当前系统返回当前日期和时间（UTC 时间）。

- Schema 相关函数
- 原生 nGQL 语句适用

函数	说明
id(vertex)	返回点 ID。数据类型和点 ID 的类型保持一致。
map properties(vertex)	返回点的所有属性。
map properties(edge)	返回边的所有属性。
string type(edge)	返回边的 Edge type。
src(edge)	返回边的起始点 ID。数据类型和点 ID 的类型保持一致。
dst(edge)	返回边的目的点 ID。数据类型和点 ID 的类型保持一致。
int rank(edge)	返回边的 rank。
vertex	返回点的信息。包括点 ID、Tag、属性和值。
edge	返回边的信息。包括 Edge type、起始点 ID、目的点 ID、rank、属性和值。
vertices	返回子图中的点的信息。详情参见 GET SUBGRAPH。
edges	返回子图中的边的信息。详情参见 GET SUBGRAPH。
path	返回路径信息。详情参见 FIND PATH。

- openCypher 兼容语句适用

函数	说明
id(<vertex>)	返回点 ID。数据类型和点 ID 的类型保持一致。
list tags(<vertex>)	返回点的 Tag，与 labels() 作用相同。
list labels(<vertex>)	返回点的 Tag，与 tags() 作用相同，用于兼容 openCypher 语法。
map properties(<vertex_or_edge>)	返回点或边的所有属性。
string type(<edge>)	返回边的 Edge type。
src(<edge>)	返回边的起始点 ID。数据类型和点 ID 的类型保持一致。
dst(<edge>)	返回边的目的点 ID。数据类型和点 ID 的类型保持一致。
vertex startNode(<path>)	获取一条边或一条路径并返回它的起始点 ID。
string endNode(<path>)	获取一条边或一条路径并返回它的目的点 ID。
int rank(<edge>)	返回边的 rank。

- 列表函数

函数	说明
keys(expr)	返回一个列表，包含字符串形式的点、边或映射的所有属性。
labels(vertex)	返回点的 Tag 列表。
nodes(path)	返回路径中所有点的列表。
range(start, end [, step])	返回 `[start,end]` 范围内固定步长的列表，默认步长 `step` 为 1。
relationships(path)	返回路径中所有关系的列表。
reverse(list)	返回将原列表逆序排列的新列表。
tail(list)	返回不包含原列表第一个元素的新列表。
head(list)	返回列表的第一个元素。
last(list)	返回列表的最后一个元素。
reduce()	将表达式逐个应用于列表中的元素，然后和累加器中的当前结果累加，最后返回完整结果。

- 类型转换函数

函数	说明
bool toBoolean()	将字符串转换为布尔。
float toFloat()	将整数或字符串转换为浮点数。
string toString()	将任意数据类型转换为字符串类型。
int toInteger()	将浮点或字符串转换为整数。
set toSet()	将列表或集合转换为集合。
int hash()	`hash()` 函数返回参数的哈希值。其参数可以是数字、字符串、列表、布尔值、NULL 等类型的值，或者计算结果为这些类型的表达式。

- 谓词函数

谓词函数只返回 `true` 或 `false` ，通常用于 `WHERE` 子句中。

```
<predicate>(<variable> IN <list> WHERE <condition>)
```

函数	说明
exists()	如果指定的属性在点、边或映射中存在，则返回 `true` ，否则返回 `false` 。
any()	如果指定的谓词适用于列表中的至少一个元素，则返回 `true` ，否则返回 `false` 。
all()	如果指定的谓词适用于列表中的每个元素，则返回 `true` ，否则返回 `false` 。
none()	如果指定的谓词不适用于列表中的任何一个元素，则返回 `true` ，否则返回 `false` 。
single()	如果指定的谓词适用于列表中的唯一一个元素，则返回 `true` ，否则返回 `false` 。

- 条件表达式函数

函数	说明
CASE	使用条件来过滤 nGQL 查询语句的结果，常用于 `YIELD` 和 `RETURN` 子句中。 `CASE` 表达式会遍历所有条件，并在满足第一个条件时停止读取后续条件，然后返回结果。 如果不满足任何条件，将通过 `ELSE` 子句返回结果。如果没有 `ELSE` 子句且不满足任何条件，则返回 `NULL` 。
coalesce()	返回所有表达式中第一个非空元素。

3.4.2 通用查询语句

- MATCH

```
MATCH <pattern> [<clause_1>]  RETURN <output>  [<clause_2>];
```

模式	示例	说明
匹配点	`(v)`	用户可以在一对括号中使用自定义变量来表示模式中的点。例如 `(v)` 。
匹配 Tag	`MATCH (v:player) RETURN v`	用户可以在点的右侧用 `:<tag_name>` 表示模式中的 Tag。
匹配多 Tag	`MATCH (v:player:team) RETURN v`	用户可以用英文冒号（:）匹配多 Tag 的点。

匹配点的属性	`MATCH (v:player{name:"Tim Duncan"}) RETURN v`
	
	`MATCH (v) WITH v, properties(v) as props, keys(properties(v)) as kk WHERE [i in kk where props[i] == "Tim Duncan"] RETURN v`
		用户可以在 Tag 的右侧用 `{<prop_name>: <prop_value>}` 表示模式中点的属性；或者不指定 Tag 直接匹配点的属性。

匹配单点 ID	`MATCH (v) WHERE id(v) == 'player101' RETURN v`	用户可以使用点 ID 去匹配点。 `id()` 函数可以检索点的 ID。
匹配多点 ID	`MATCH (v:player { name: 'Tim Duncan' })--(v2) WHERE id(v2) IN ["player101", "player102"] RETURN v2`	要匹配多个点的 ID，可以用 `WHERE id(v) IN [vid_list]` 。
匹配连接的点	`MATCH (v:player{name:"Tim Duncan"})--(v2) RETURN v2.player.name AS Name`	用户可以使用 `--` 符号表示两个方向的边，并匹配这些边连接的点。用户可以在 `--` 符号上增加 `<` 或 `>` 符号指定边的方向。
匹配路径	`MATCH p=(v:player{name:"Tim Duncan"})-->(v2) RETURN p`	连接起来的点和边构成了路径。用户可以使用自定义变量命名路径。
匹配边	`MATCH (v:player{name:"Tim Duncan"})-[e]-(v2) RETURN e` `MATCH ()<-[e]-()   RETURN e`	除了用 `--` 、 `-->` 、 `<--` 表示未命名的边之外，用户还可以在方括号中使用自定义变量命名边。例如 `-[e]-` 。
匹配 Edge type	`MATCH ()-[e:follow]->() RETURN e`	和点一样，用户可以用 `:<edge_type>` 表示模式中的 Edge type，例如 `-[e:follow]-` 。

匹配边的属性	`MATCH (v:player{name:"Tim Duncan"})-[e:follow{degree:95}]->(v2) RETURN e`
	
	`MATCH ()-[e]->() WITH e, properties(e) as props, keys(properties(e)) as kk WHERE [i in kk where props[i] == 90] RETURN e`
		用户可以用 `{<prop_name>: <prop_value>}` 表示模式中 Edge type 的属性，例如 `[e:follow{likeness:95}]` ；或者不指定 Edge type 直接匹配边的属性。

匹配多个 Edge type	`MATCH (v:player{name:"Tim Duncan"})-[e:follow | :serve]->(v2) RETURN e`	使用 `|` 可以匹配多个 Edge type，例如 `[e:follow | :serve]` 。第一个 Edge type 前的英文冒号（:）不可省略，后续 Edge type 前的英文冒号可以省略，例如 `[e:follow | serve]` 。
匹配多条边	`MATCH (v:player{name:"Tim Duncan"})-[]->(v2)<-[e:serve]-(v3) RETURN v2, v3`	用户可以扩展模式，匹配路径中的多条边。
匹配定长路径	`MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*2]->(v2) RETURN DISTINCT v2 AS Friends`	用户可以在模式中使用 `:<edge_type>*<hop>` 匹配定长路径。 `hop` 必须是一个非负整数。 `e` 的数据类型是列表。
匹配变长路径	`MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*1..3]->(v2) RETURN v2 AS Friends`	`minHop` ：可选项。表示路径的最小长度。 `minHop` 必须是一个非负整数，默认值为 1。 `maxHop` ：可选项。表示路径的最大长度。 `maxHop` 必须是一个非负整数，默认为无穷大。 `e` 的数据类型是列表。
匹配多个 Edge type 的变长路径	`MATCH p=(v:player{name:"Tim Duncan"})-[e:follow | serve*2]->(v2) RETURN DISTINCT v2`	用户可以在变长或定长模式中指定多个 Edge type。 `hop` 、 `minHop` 和 `maxHop` 对所有 Edge type 都生效。 `e` 的数据类型是列表。
检索点或边的信息	`MATCH (v:player{name:"Tim Duncan"}) RETURN v` `MATCH (v:player{name:"Tim Duncan"})-[e]->(v2) RETURN e`	使用 `RETURN {<vertex_name> | <edge_name>}` 检索点或边的所有信息。
检索点 ID	`MATCH (v:player{name:"Tim Duncan"}) RETURN id(v)`	使用 `id()` 函数检索点 ID。
检索 Tag		

模式	示例	说明
	`MATCH (v:player{name:"Tim Duncan"}) RETURN labels(v)`	使用 `labels()` 函数检索点上的 Tag 列表。 检索列表 `labels(v)` 中的第 N 个元素，可以使用 `labels(v)[n-1]` 。
检索点或边的单个属性	`MATCH (v:player{name:"Tim Duncan"}) RETURN v.player.age`	使用 `RETURN {<vertex_name> | <edge_name>}.<property>` 检索单个属性。 使用 `AS` 设置属性的别名。
检索点或边的所有属性	`MATCH p=(v:player{name:"Tim Duncan"})-[]->(v2) RETURN properties(v2)`	使用 `properties()` 函数检索点或边的所有属性。
检索 Edge type	`MATCH p=(v:player{name:"Tim Duncan"})-[e]->() RETURN DISTINCT type(e)`	使用 `type()` 函数检索匹配的 Edge type。
检索路径	`MATCH p=(v:player{name:"Tim Duncan"})-[*3]->() RETURN p`	使用 `RETURN <path_name>` 检索匹配路径的所有信息。
检索路径中的点	`MATCH p=(v:player{name:"Tim Duncan"})-[]->(v2) RETURN nodes(p)`	使用 `nodes()` 函数检索路径中的所有点。
检索路径中的边	`MATCH p=(v:player{name:"Tim Duncan"})-[]->(v2) RETURN relationships(p)`	使用 `relationships()` 函数检索路径中的所有边。
检索路径长度	`MATCH p=(v:player{name:"Tim Duncan"})-[*..2]->(v2) RETURN p AS Paths, length(p) AS Length`	使用 `length()` 函数检索路径的长度。

- OPTIONAL MATCH

模式	示例	说明
作为 `MATCH` 语句的可选项去匹配图数据库中的模式	`MATCH (m)-[]->(n) WHERE id(m)=="player100" OPTIONAL MATCH (n)-[]->(l) RETURN id(m),id(n),id(l)`	如果图数据库中没有对应的模式，对应的列返回 `NULL` 。

- LOOKUP

```
LOOKUP ON {<vertex_tag> | <edge_type>}
[WHERE <expression> [AND <expression> ...]]
YIELD <return_list> [AS <alias>]
```

模式	示例	说明
检索点	`LOOKUP ON player WHERE player.name == "Tony Parker" YIELD player.name AS name, player.age AS age`	返回 Tag 为 `player` 且 `name` 为 `Tony Parker` 的点。
检索边	`LOOKUP ON follow WHERE follow.degree == 90 YIELD follow.degree`	返回 Edge type 为 `follow` 且 `degree` 为 `90` 的边。
通过 Tag 列出所有点	`LOOKUP ON player YIELD properties(vertex),id(vertex)`	查找所有 Tag 为 `player` 的点 VID。
通过 Edge type 列出边	`LOOKUP ON follow YIELD edge AS e`	查找 Edge type 为 `follow` 的所有边的信息。
统计点	`LOOKUP ON player YIELD id(vertex)| YIELD COUNT(*) AS Player_Count`	统计 Tag 为 `player` 的点。
统计边	`LOOKUP ON follow YIELD edge as e | YIELD COUNT(*) AS Like_Count`	统计 Edge type 为 `follow` 的边。

- GO

```
GO [[<M> TO] <N> {STEP|STEPS} ] FROM <vertex_list>
OVER <edge_type_list> [{REVERSELY | BIDIRECT}]
[ WHERE <conditions> ]
YIELD [DISTINCT] <return_list>
[{SAMPLE <sample_list> | LIMIT <limit_list>}]
[| GROUP BY {col_name | expr | position} YIELD <col_name>]
[| ORDER BY <expression> [{ASC | DESC}]]
[| LIMIT [<offset_value>,] <number_rows>]
```

示例	说明
`GO FROM "player102" OVER serve YIELD dst(edge)`	返回 player102 所属队伍。
`GO 2 STEPS FROM "player102" OVER follow YIELD dst(edge)`	返回距离 player102 两跳的朋友。
`GO FROM "player100", "player102" OVER serve WHERE properties(edge).start_year > 1995 YIELD DISTINCT properties($$).name AS team_name, properties(edge).start_year AS start_year, properties($^).name AS player_name`	添加过滤条件。
`GO FROM "player100" OVER follow, serve YIELD properties(edge).degree, properties(edge).start_year`	遍历多个 Edge type。属性没有值时，会显示 `NULL`。
`GO FROM "player100" OVER follow REVERSELY YIELD src(edge) AS destination`	返回 player100 入方向的邻居点。
`GO FROM "player100" OVER follow REVERSELY YIELD src(edge) AS id | GO FROM $-.id OVER serve WHERE properties($^).age > 20 YIELD properties($^).name AS FriendOf, properties($$).name AS Team`	查询 player100 的朋友和朋友所属队伍。
`GO FROM "player102" OVER follow YIELD dst(edge) AS both`	返回 player102 所有邻居点。
`GO 2 STEPS FROM "player100" OVER follow YIELD src(edge) AS src, dst(edge) AS dst, properties($$).age AS age | GROUP BY $-.dst YIELD $-.dst AS dst, collect_set($-.src) AS src, collect($-.age) AS age`	根据年龄分组。

- FETCH

- 获取点的属性值

```
FETCH PROP ON {<tag_name>[, tag_name ...] | *}
<vid> [, vid ...]
YIELD <return_list> [AS <alias>]
```

示例	说明
`FETCH PROP ON player "player100" YIELD properties(vertex)`	在 `FETCH` 语句中指定 Tag 获取对应点的属性值。
`FETCH PROP ON player "player100" YIELD player.name AS name`	使用 `YIELD` 子句指定返回的属性。
`FETCH PROP ON player "player101", "player102", "player103" YIELD properties(vertex)`	指定多个点 ID 获取多个点的属性值，点之间用英文逗号（,）分隔。
`FETCH PROP ON player, t1 "player100", "player103" YIELD properties(vertex)`	在 `FETCH` 语句中指定多个 Tag 获取属性值。Tag 之间用英文逗号（,）分隔。
`FETCH PROP ON * "player100", "player106", "team200" YIELD properties(vertex)`	在 `FETCH` 语句中使用 `*` 获取当前图空间所有标签里，点的属性值。

- 获取边的属性值

```
FETCH PROP ON <edge_type> <src_vid> -> <dst_vid>[@<rank>] [, <src_vid> -> <dst_vid> ...]
YIELD <output>;
```

示例	说明
`FETCH PROP ON serve "player100" -> "team204" YIELD properties(edge)`	获取连接 player100 和 team204 的边 serve 的所有属性值。
`FETCH PROP ON serve "player100" -> "team204" YIELD serve.start_year`	使用 `YIELD` 子句指定返回的属性。
`FETCH PROP ON serve "player100" -> "team204", "player133" -> "team202" YIELD properties(edge)`	指定多个边模式 (`<src_vid> -> <dst_vid>[@<rank>]`) 获取多个边的属性值。模式之间用英文逗号（,）分隔。
`FETCH PROP ON serve "player100" -> "team204"@1 YIELD properties(edge)`	要获取 rank 不为 0 的边，请在 FETCH 语句中设置 rank。
`GO FROM "player101" OVER follow YIELD follow._src AS s, follow._dst AS d | FETCH PROP ON follow $-.s -> $-.d YIELD follow.degree`	返回从点 player101 开始的 follow 边的 degree 值。
`$var = GO FROM "player101" OVER follow YIELD follow._src AS s, follow._dst AS d; FETCH PROP ON follow $var.s -> $var.d YIELD follow.degree`	自定义变量构建查询。

- SHOW

语句	语法	示例	说明
SHOW CHARSET	`SHOW CHARSET`	`SHOW CHARSET`	显示当前的字符集。
SHOW COLLATION	`SHOW COLLATION`	`SHOW COLLATION`	显示当前的排序规则。
SHOW CREATE SPACE	`SHOW CREATE SPACE <space_name>`	`SHOW CREATE SPACE basketballplayer`	显示指定图空间的创建语句。
SHOW CREATE TAG/EDGE	`SHOW CREATE {TAG <tag_name> | EDGE <edge_name>}`	`SHOW CREATE TAG player`	显示指定 Tag/Edge type 的基本信息。
SHOW HOSTS	`SHOW HOSTS [GRAPH | STORAGE | META]`	`SHOW HOSTS` `SHOW HOSTS GRAPH`	显示 Graph、Storage、Meta 服务主机信息、版本信息。
SHOW INDEX STATUS	`SHOW {TAG | EDGE} INDEX STATUS`	`SHOW TAG INDEX STATUS`	重建原生索引的作业状态，以便确定重建索引是否成功。
SHOW INDEXES	`SHOW {TAG | EDGE} INDEXES`	`SHOW TAG INDEXES`	列出当前图空间内的所有 Tag 和 Edge type（包括属性）的索引。
SHOW PARTS	`SHOW PARTS [<part_id>]`	`SHOW PARTS`	显示图空间中指定分片或所有分片的信息。
SHOW ROLES	`SHOW ROLES IN <space_name>`	`SHOW ROLES in basketballplayer`	显示分配给用户的角色信息。
SHOW SNAPSHOTS	`SHOW SNAPSHOTS`	`SHOW SNAPSHOTS`	显示所有快照信息。
SHOW SPACES	`SHOW SPACES`	`SHOW SPACES`	显示现存的图空间。
SHOW STATS	`SHOW STATS`	`SHOW STATS`	显示最近 `STATS` 作业收集的图空间统计信息。
SHOW TAGS/EDGES	`SHOW TAGS | EDGES`	`SHOW TAGS`、`SHOW EDGES`	显示当前图空间内的所有 Tag/Edge type。
SHOW USERS	`SHOW USERS`	`SHOW USERS`	显示用户信息。
SHOW SESSIONS	`SHOW SESSIONS`	`SHOW SESSIONS`	显示所有会话信息。
SHOW SESSIONS	`SHOW SESSION <Session_Id>`	`SHOW SESSION 1623304491050858`	指定会话 ID 进行查看。
SHOW QUERIES	`SHOW [ALL] QUERIES`	`SHOW QUERIES`	查看当前 Session 中正在执行的查询请求信息。
SHOW META LEADER	`SHOW META LEADER`	`SHOW META LEADER`	显示当前 Meta 集群的 leader 信息。

3.4.3 子句和选项

子句	语法	示例	说明
GROUP BY	GROUP BY <var> YIELD <var>, <aggregation_function(var)>	GO FROM "player100" OVER follow BIDIRECT YIELD $$.player.name as Name | GROUP BY $-.Name YIELD $-.Name as Player, count(*) AS Name_Count	查找所有连接到 player100 的点，并根据他们的姓名进行分组，返回姓名的出现次数。
LIMIT	YIELD <var> [| LIMIT [<offset_value>,] <number_rows>]	GO FROM "player100" OVER follow REVERSELY YIELD $$.player.name AS Friend, $$.player.age AS Age | ORDER BY $-.Age, $-.Friend | LIMIT 1, 3	从排序结果中返回第 2 行开始的 3 行数据。
SKIP	RETURN <var> [SKIP <offset>] [LIMIT <number_rows>]	MATCH (v:player{name:"Tim Duncan"}) --> (v2) RETURN v2.player.name AS Name, v2.player.age AS Age ORDER BY Age DESC SKIP 1	用户可以单独使用 SKIP <offset> 设置偏移量，后面不需要添加 LIMIT <number_rows> 。
SAMPLE	<go_statement> SAMPLE <sample_list>;	GO 3 STEPS FROM "player100" OVER * YIELD properties($$).name AS NAME, properties($$).age AS Age SAMPLE [1,2,3];	在结果集中均匀取样并返回指定数量的数据。
ORDER BY	<YIELD clause> ORDER BY <expression> [ASC | DESC] [, <expression> [ASC | DESC] ...]	FETCH PROP ON player "player100", "player101", "player102", "player103" YIELD player.age AS age, player.name AS name | ORDER BY $-.age ASC, $-.name DESC	ORDER BY 子句指定输出结果的排序规则。
RETURN	RETURN {<vertex_name>|<edge_name>|<vertex_name>.<property>|<edge_name>.<property>|...}	MATCH (v:player) RETURN v.player.name, v.player.age LIMIT 3	返回点的属性为 name 和 age 的前三行值。
TTL	CREATE TAG <tag_name>(<property_name_1> <property_value_1>, <property_name_2> <property_value_2>, ...) ttl_duration= <value_int>, ttl_col = <property_name>	CREATE TAG t2(a int, b int, c string) ttl_duration= 100, ttl_col = "a"	创建 Tag 并设置 TTL 选项。
WHERE	WHERE {<vertex|edge_alias>.<property_name> {>|==|<|...} <value>...}	MATCH (v:player) WHERE v.player.name == "Tim Duncan" XOR (v.player.age < 30 AND v.player.name == "Yao Ming") OR NOT (v.player.name == "Yao Ming" OR v.player.name == "Tim Duncan") RETURN v.player.name, v.player.age	WHERE 子句可以根据条件过滤输出结果，通常用于 GO 和 LOOKUP 语句， MATCH 和 WITH 语句。
YIELD	YIELD [DISTINCT] <col> [AS <alias>] [, <col> [AS <alias>] ...] [WHERE <conditions>];	GO FROM "player100" OVER follow YIELD dst(edge) AS ID | FETCH PROP ON player $-.ID YIELD player.age AS Age \| YIELD AVG($-.Age) as Avg_age, count(*)as Num_friends	查找 player100 关注的 player，并计算他们的平均年龄。
WITH	MATCH $expressions WITH {nodes()|labels()|...}	MATCH p=(v:player{name:"Tim Duncan"})--() WITH nodes(p) AS n UNWIND n AS n1 RETURN DISTINCT n1	WITH 子句可以获取并处理查询前半部分的结果，并将处理结果作为输入传递给查询的后半部分。
UNWIND	UNWIND <list> AS <alias> <RETURN clause>	UNWIND [1,2,3] AS n RETURN n	拆分列表。

3.4.4 图空间语句

语句	语法	示例	说明
CREATE SPACE	CREATE SPACE [IF NOT EXISTS] <graph_space_name> ([partition_num = <partition_number>,] [replica_factor = <replica_number>,] vid_type = {FIXED_STRING(<N>) | INT[64]}) [COMMENT = '<comment>']	CREATE SPACE my_space_1 (vid_type=FIXED_STRING(30))	创建一个新的图空间。
CREATE SPACE	CREATE SPACE <new_graph_space_name> AS <old_graph_space_name>	CREATE SPACE my_space_4 as my_space_3	克隆现有图空间的 Schema。
USE	USE <graph_space_name>	USE space1	指定一个图空间，或切换到另一个图空间，将其作为后续查询的工作空间。
SHOW SPACES	SHOW SPACES	SHOW SPACES	列出 NebulaGraph 示例中的所有图空间。
DESCRIBE SPACE	DESC[RIBE] SPACE <graph_space_name>	DESCRIBE SPACE basketballplayer	显示指定图空间的信息。
CLEAR SPACE	CLEAR SPACE [IF EXISTS] <graph_space_name>	清空图空间中的点和边，但不会删除图空间本身以及其中的 Schema 信息。	
DROP SPACE	DROP SPACE [IF EXISTS] <graph_space_name>	DROP SPACE basketballplayer	删除指定图空间的所有内容。

3.4.5 TAG 语句

语句	语法	示例	说明
CREATE TAG	CREATE TAG [IF NOT EXISTS] <tag_name> (<prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>'] [{, <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']} ...]) [TTL_DURATION = <ttl_duration>] [TTL_COL = <prop_name>] [COMMENT = '<comment>']	CREATE TAG woman(name string, age int, married bool, salary double, create_time timestamp) TTL_DURATION = 100, TTL_COL = "create_time"	通过指定名称创建一个 Tag。
DROP TAG	DROP TAG [IF EXISTS] <tag_name>	DROP TAG test;	删除当前工作空间内所有点上的指定 Tag。
ALTER TAG	ALTER TAG <tag_name> <alter_definition> [, alter_definition] ...] [ttl_definition [, ttl_definition] ...] [COMMENT = '<comment>']	ALTER TAG t1 ADD (p3 int, p4 string)	修改 Tag 的结构。例如增删属性、修改数据类型，也可以为属性设置、修改 TTL（Time-To-Live)。
SHOW TAGS	SHOW TAGS	SHOW TAGS	显示当前图空间内的所有 Tag 名称。
DESCRIBE TAG	DESC[RIBE] TAG <tag_name>	DESCRIBE TAG player	查看指定 Tag 的详细信息，例如字段名称、数据类型等。
DELETE TAG	DELETE TAG <tag_name_list> FROM <VID>	DELETE TAG test1 FROM "test"	删除指定点上的指定 Tag。

3.4.6 Edge type 语句

语句	语法	示例	说明
CREATE EDGE	CREATE EDGE [IF NOT EXISTS] <edge_type_name> (<prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>'] [{, <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']} ...]) [TTL_DURATION = <ttl_duration>] [TTL_COL = <prop_name>] [COMMENT = '<comment>']	CREATE EDGE e1(p1 string, p2 int, p3 timestamp) TTL_DURATION = 100, TTL_COL = "p2"	指定名称创建一个 Edge type。
DROP EDGE	DROP EDGE [IF EXISTS] <edge_type_name>	DROP EDGE e1	删除当前工作空间内的指定 Edge type。
ALTER EDGE	ALTER EDGE <edge_type_name> <alter_definition> [, alter_definition] ...] [ttl_definition [, ttl_definition] ...] [COMMENT = '<comment>']	ALTER EDGE e1 ADD (p3 int, p4 string)	修改 Edge type 的结构。
SHOW EDGES	SHOW EDGES	SHOW EDGES	显示当前图空间内的所有 Edge type 名称。
DESCRIBE EDGE	DESC[RIBE] EDGE <edge_type_name>	DESCRIBE EDGE follow	查看指定 Edge type 的详细信息，例如字段名称、数据类型等。

3.4.7 点语句

语句	语法	示例	说明
INSERT VERTEX	INSERT VERTEX [IF NOT EXISTS] [tag_props, [tag_props] ...] VALUES <vid>: ([prop_value_list])	INSERT VERTEX t2 (name, age) VALUES "13":("n3", 12), "14":("n4", 8)	在 NebulaGraph 实例的指定图空间中插入一个或多个点。
DELETE VERTEX	DELETE VERTEX <vid> [, <vid> ...]	DELETE VERTEX "team1"	删除点，以及点关联的出边和入边。
UPDATE VERTEX	UPDATE VERTEX ON <tag_name> <vid> SET <update_prop> [WHEN <condition>] [YIELD <output>]	UPDATE VERTEX ON player "player101" SET age = age + 2	修改点上 Tag 的属性值。
UPSERT VERTEX	UPSERT VERTEX ON <tag> <vid> SET <update_prop> [WHEN <condition>] [YIELD <output>]	UPSERT VERTEX ON player "player667" SET age = 31	结合 UPDATE 和 INSERT，如果点存在，会修改点的属性值；如果点不存在，会插入新的点。

3.4.8 边语句

语句	语法	示例	说明
INSERT EDGE	`INSERT EDGE [IF NOT EXISTS] <edge_type> ( <prop_name_list> ) VALUES <src_vid> -> <dst_vid>[@<rank>] : ( <prop_value_list> ) [, <src_vid> -> <dst_vid>[@<rank>] : ( <prop_value_list> ), ...]`	`INSERT EDGE e2 (name, age) VALUES "11"->"13": ("n1", 1)`	在 NebulaGraph 实例的指定图空间中插入一条或多条边。
DELETE EDGE	`DELETE EDGE <edge_type> <src_vid> -> <dst_vid>[@<rank>] [, <src_vid> -> <dst_vid>[@<rank>] ...]`	`DELETE EDGE serve "player100" -> "team204"@0`	删除边。一次可以删除一条或多条边。
UPDATE EDGE	`UPDATE EDGE ON <edge_type> <src_vid> -> <dst_vid> [@<rank>] SET <update_prop> [WHEN <condition>] [YIELD <output>]`	`UPDATE EDGE ON serve "player100" -> "team204"@0 SET start_year = start_year + 1`	修改边上 Edge type 的属性。
UPSERT EDGE	`UPSERT EDGE ON <edge_type> <src_vid> -> <dst_vid> [@rank] SET <update_prop> [WHEN <condition>] [YIELD <properties>]`	`UPSERT EDGE on serve "player666" -> "team200"@0 SET end_year = 2021`	结合 `UPDATE` 和 `INSERT`，如果边存在，会更新边的属性；如果边不存在，会插入新的边。

3.4.9 索引

- 原生索引

索引配合 `LOOKUP` 和 `MATCH` 语句使用。

语句	语法	示例	说明
CREATE INDEX	`CREATE {TAG | EDGE} INDEX [IF NOT EXISTS] <index_name> ON {<tag_name> | <edge_name>} ([<prop_name_list>]) [COMMENT = '<comment>']`	`CREATE TAG INDEX player_index on player()`	对 Tag、EdgeType 或其属性创建原生索引。
SHOW CREATE INDEX	`SHOW CREATE {TAG | EDGE} INDEX <index_name>`	`show create tag index index_2`	创建 Tag 或者 Edge type 时使用的 nGQL 语句，其中包含索引的详细信息，例如其关联的属性。
SHOW INDEXES	`SHOW {TAG | EDGE} INDEXES`	`SHOW TAG INDEXES`	列出当前图空间内的所有 Tag 和 Edge type（包括属性）的索引。
DESCRIBE INDEX	`DESCRIBE {TAG | EDGE} INDEX <index_name>`	`DESCRIBE TAG INDEX player_index_0`	查看指定索引的信息，包括索引的属性名称（Field）和数据类型（Type）。
REBUILD INDEX	`REBUILD {TAG | EDGE} INDEX [<index_name_list>]`	`REBUILD TAG INDEX single_person_index`	重建索引。索引功能不会自动对其创建之前已存在的存量数据生效。
SHOW INDEX STATUS	`SHOW {TAG | EDGE} INDEX STATUS`	`SHOW TAG INDEX STATUS`	查看索引名称和对应的状态。
DROP INDEX	`DROP {TAG | EDGE} INDEX [IF EXISTS] <index_name>`	`DROP TAG INDEX player_index_0`	删除当前图空间中已存在的索引。

- 全文索引

语法	示例	说明
`SIGN IN TEXT SERVICE [(<elastic_ip:port> [,<username>, <password>]), (<elastic_ip:port>), ...]`	`SIGN IN TEXT SERVICE (127.0.0.1:9200)`	NebulaGraph 的全文索引是基于 Elasticsearch 实现，部署 Elasticsearch 集群之后，可以使用 `SIGN IN` 语句登录 Elasticsearch 客户端。
`SHOW TEXT SEARCH CLIENTS`	`SHOW TEXT SEARCH CLIENTS`	列出文本搜索客户端。
`SIGN OUT TEXT SERVICE`	`SIGN OUT TEXT SERVICE`	退出所有文本搜索客户端。
`CREATE FULLTEXT {TAG | EDGE} INDEX <index_name> ON {<tag_name> | <edge_name>} (<prop_name> [,<prop_name>]...) [ANALYZER="<analyzer_name>"]`	`CREATE FULLTEXT TAG INDEX nebula_index_1 ON player(name)`	创建全文索引。
`SHOW FULLTEXT INDEXES`	`SHOW FULLTEXT INDEXES`	显示全文索引。
`REBUILD FULLTEXT INDEX`	`REBUILD FULLTEXT INDEX`	重建全文索引。
`DROP FULLTEXT INDEX <index_name>`	`DROP FULLTEXT INDEX nebula_index_1`	删除全文索引。
`LOOKUP ON {<tag> | <edge_type>} WHERE ES_QUERY(<index_name>, "<text>") YIELD <return_list> [| LIMIT [<offset>,] <number_rows>]`	`LOOKUP ON player WHERE ES_QUERY(fulltext_index_1,"Chris") YIELD id(vertex)`	使用查询选项。

3.4.10 子图和路径

<table>
<tr><th>类型</th><th>语法</th><th>示例</th><th>说明</th></tr>
<tr><td>子图</td><td>GET SUBGRAPH [WITH PROP] [<step_count> {STEP|STEPS}] FROM {<vid>, <vid>...} [{IN | OUT | BOTH} <edge_type>, <edge_type>...] YIELD [VERTICES AS <vertex_alias>] [,EDGES AS <edge_alias>]</td><td>GET SUBGRAPH 1 STEPS FROM "player100" YIELD VERTICES AS nodes, EDGES AS relationships</td><td>指定 Edge type 的起始点可以到达的点和边的信息，返回子图信息。</td></tr>
<tr><td>路径</td><td>FIND { SHORTEST | ALL | NOLOOP } PATH [WITH PROP] FROM <vertex_id_list> TO <vertex_id_list> OVER <edge_type_list> [REVERSELY | BIDIRECT] [<WHERE clause>] [UPTO <N> {STEP|STEPS}] YIELD path as <alias> [| ORDER BY $-.path] [| LIMIT <M>]</td><td>FIND SHORTEST PATH FROM "player102" TO "team204" OVER * YIELD path as p</td><td>查找指定起始点和目的点之间的路径。返回的路径格式类似于 (<vertex_id>)-[:<edge_type_name>@<rank>]->(<vertex_id>)。</td></tr>
</table>

3.4.11 查询调优

<table>
<tr><th>类型</th><th>语法</th><th>示例</th><th>说明</th></tr>
<tr><td>EXPLAIN</td><td>EXPLAIN [format="row" | "dot"] <your_nGQL_statement></td><td>EXPLAIN format="row" SHOW TAGS
EXPLAIN format="dot" SHOW TAGS</td><td>输出 nGQL 语句的执行计划，但不会执行 nGQL 语句。</td></tr>
<tr><td>PROFILE</td><td>PROFILE [format="row" | "dot"] <your_nGQL_statement></td><td>PROFILE format="row" SHOW TAGS
EXPLAIN format="dot" SHOW TAGS</td><td>执行 nGQL 语句，然后输出执行计划和执行概要。</td></tr>
</table>

3.4.12 运维

- SUBMIT JOB BALANCE

语法	说明
`SUBMIT JOB BALANCE LEADER`	启动任务均衡分布所有图空间中的 leader。该命令会返回任务 ID。

- 作业管理

语法	说明
`SUBMIT JOB COMPACT`	触发 RocksDB 的长耗时 `compact` 操作。
`SUBMIT JOB FLUSH`	将内存中的 RocksDB memfile 写入硬盘。
`SUBMIT JOB STATS`	启动一个作业，该作业对当前图空间进行统计。作业完成后，用户可以使用 `SHOW STATS` 语句列出统计结果。
`SHOW JOB <job_id>`	显示当前图空间内指定作业和相关任务的信息。Meta 服务将 `SUBMIT JOB` 请求解析为多个任务，然后分配给进程 nebula-storaged。
`SHOW JOBS`	列出当前图空间内所有未过期的作业。
`STOP JOB`	停止当前图空间内未完成的作业。
`RECOVER JOB`	重新执行当前图空间内失败的作业，并返回已恢复的作业数量。

- 终止查询

语法	示例	说明
`KILL QUERY (session=<session_id>, plan=<plan_id>)`	`KILL QUERY(SESSION=1625553545984255,PLAN=163)`	在一个会话中执行命令终止另一个会话中的查询。

- 终止会话

语法	示例	说明
`KILL {SESSION|SESSIONS} <SessionId>`	`KILL SESSION 1672887983842984`	终止单个会话。
`SHOW SESSIONS | YIELD $-.SessionId AS sid [WHERE <filter_clause>] | KILL {SESSION|SESSIONS} $-.sid`	`SHOW SESSIONS | YIELD $-.SessionId AS sid, $-.CreateTime as CreateTime | ORDER BY $-.CreateTime ASC | LIMIT 2 | KILL SESSIONS $-.sid`	终止多个会话。
`SHOW SESSIONS | KILL SESSIONS $-.SessionId`	`SHOW SESSIONS | KILL SESSIONS $-.SessionId`	终止所有会话。

最后更新: 2025年8月4日

4. nGQL 指南

4.1 nGQL 概述

4.1.1 什么是 nGQL

nGQL是 NebulaGraph 使用的的声明式图查询语言，支持灵活高效的图模式，而且 nGQL 是为开发和运维人员设计的类 SQL 查询语言，易于学习。

nGQL 是一个进行中的项目，会持续发布新特性和优化，因此可能会出现语法和实际操作不一致的问题，如果遇到此类问题，请提交 issue 通知 NebulaGraph 团队。 NebulaGraph 3.0 及更新版本正在支持 openCypher 9。

nGQL 可以做什么

- 支持图遍历
- 支持模式匹配
- 支持聚合
- 支持修改图
- 支持访问控制
- 支持聚合查询
- 支持索引
- 支持大部分 openCypher 9 图查询语法（不支持修改和控制语法）

示例数据 Basketballplayer

用户可以下载 NebulaGraph 示例数据 basketballplayer 文件，然后使用 NebulaGraph Console，使用选项 `-f` 执行脚本。

> Note
>
> 导入示例数据前，确保已执行 `ADD HOSTS` 命令将 Storage 主机增加至集群中。更多信息，请参见管理 Storage 主机。

占位标识符和占位符值

NebulaGraph 查询语言 nGQL 参照以下标准设计：

- (Draft) ISO/IEC JTC1 N14279 SC 32 - Database_Languages - GQL
- (Draft) ISO/IEC JTC1 SC32 N3228 - SQL_Property_Graph_Queries - SQLPGQ
- OpenCypher 9

在模板代码中，任何非关键字、字面值或标点符号的标记都是占位符标识符或占位符值。

本文中 nGQL 语法符号的说明如下。

符号	含义
< >	语法元素的名称。
:	定义元素的公式。
[ ]	可选元素。
{ }	显式的指定元素。
|	所有可选的元素。
...	可以重复多次。

例如创建点的 nGQL 语法：

```
INSERT VERTEX [IF NOT EXISTS] [tag_props, [tag_props] ...]
VALUES <vid>: ([prop_value_list])

tag_props:
  tag_name ([prop_name_list])

prop_name_list:
   [prop_name [, prop_name] ...]

prop_value_list:
   [prop_value [, prop_value] ...]
```

示例语句：

```
nebula> CREATE TAG IF NOT EXISTS player(name string, age int);
```

关于 **openCypher** 兼容性

原生 **NGQL** 和 **OPENCYPHER** 的关系

原生 nGQL 是由 NebulaGraph 自行创造和实现的图查询语言。openCypher 是由 openCypher Implementers Group 组织所开源和维护的图查询语言，最新版本为 openCypher 9。

由于 nGQL 语言部分兼容了 openCypher，这个部分在本文中称为 openCypher 兼容语句。

> Note
>
> `nGQL 语言` = `原生 nGQL 语句` + `openCypher 兼容语句`

NGQL 完全兼容 **OPENCYPHER 9** 吗？

不。

> openCypher 兼容性
>
> nGQL 设计目标仅为兼容部分的 DQL 语句(match, optional match, with等)。
>
> 不计划兼容任何 DDL，DML，DCL；
>
> 不计划兼容 Bolt 协议；
>
> 不计划兼容 APOC 与 GDS。
>
> 在本文搜索 "compatibility" 或者 “兼容性” 查看具体不兼容的细节。
>
> 在 Issues 中已经列出已知的兼容错误。如果发现这种类型的新问题，请提交问题并附带 `incompatible` 标签。

NGQL 和 OPENCYPHER 9 的 主要差异有哪些?

类别	openCypher 9	nGQL
Schema	弱 Schema	强 Schema
相等运算符	`=`	`==`
数学求幂	`^`	使用 `pow(x, y)` 替代 `^`。
边 Rank	无此概念	用 `@rank` 设置。
语句	-	不支持 openCypher 9 的所有 DML 语句（如 `CREATE`、 `MERGE` 等），不支持所有的 DCL，和支持部分 `MATCH`，`OPTIONAL MATCH` 语法和函数。
语句文本换行	换行符	`\` + 换行符
Label 与 Tag 是不同的概念	Label 用于寻找点（点的索引）。	Tag 用于定义点的一种类型及相应的属性，无索引功能。
预编译与参数化查询	支持	仅支持参数化查询。

Compatibility

请注意 openCypher 9 和 Cypher 在语法和许可上有不同：

1. Cypher 要求所有 Cypher 语句必须“显式地在一个事务中”执行，而 openCypher 没有这样的要求。另外，nGQL 没有事务及隔离性。
2. Cypher 企业版功能有多种的约束（constraints），包括 Unique node property constraints、Node property existence constraints、Relationship property existence constraints、Node key constraints。 OpenCypher 标准中没有约束。 而 nGQL 是强 Schema 系统，前述的约束大多通过 Schema 定义可实现（包括 NOT NULL），唯一不能支持的功能是“属性值唯一性”（UNIQUE constraint）。
3. Cypher 有 APoC，openCypher 9 没有 APoC。Cypher 有 Bolt 协议支持要求， openCypher 9 没有。

哪里可以找到更多 NGQL 的示例?

用户可以在 GitHub 的 features 目录内查看超过 2500 条 nGQL 示例。

features 目录内包含很多 `.features` 格式的文件，每个文件都记录了使用 nGQL 的场景和示例。例如：

```
Feature: Basic match

  Background:
    Given a graph with space named "basketballplayer"

  Scenario: Single node
    When executing query:
      """
      MATCH (v:player {name: "Yao Ming"}) RETURN v;
      """
    Then the result should be, in any order, with relax comparison:
      | v                                                  |
      | ("player133" :player{age: 38, name: "Yao Ming"}) |

  Scenario: One step
    When executing query:
      """
      MATCH (v1:player{name: "LeBron James"}) -[r]-> (v2)
      RETURN type(r) AS Type, v2.player.name AS Name
      """
    Then the result should be, in any order:

      | Type     | Name        |
      | "follow" | "Ray Allen" |
      | "serve"  | "Lakers"    |
      | "serve"  | "Heat"      |
      | "serve"  | "Cavaliers" |

Feature:  Comparison of where clause

  Background:
    Given a graph with space named "basketballplayer"

    Scenario: push edge props filter down
      When profiling query:
```

```
        """
        GO FROM "player100" OVER follow
        WHERE properties(edge).degree IN [v IN [95,99] WHERE v > 0]
        YIELD dst(edge), properties(edge).degree
        """
      Then the result should be, in any order:
        | follow._dst | follow.degree |
        | "player101" | 95            |
        | "player125" | 95            |
      And the execution plan should be:
        | id | name         | dependencies | operator info                                                      |
        | 0  | Project      | 1            |                                                                    |
        | 1  | GetNeighbors | 2            | {"filter": "(properties(edge).degree IN [v IN [95,99] WHERE (v>0)])"} |
        | 2  | Start        |              |                                                                    |
```

示例中的关键字说明如下。

关键字	说明
`Feature`	描述当前文档的主题。
`Background`	描述当前文档的背景信息。
`Given`	描述执行示例语句的前提条件。
`Scenario`	描述具体场景。如果场景之前有 `@skip` 标识，表示这个场景下示例语句可能无法正常工作，请不要在生产环境中使用该示例语句。
`When`	描述要执行的 nGQL 示例语句。可以是 `executing query` 或 `profiling query` 。
`Then`	描述执行 `When` 内语句的预期返回结果。如果返回结果和文档不同，请提交 issue 通知 NebulaGraph 团队。
`And`	描述执行 `When` 内语句的副作用或执行计划。
`@skip`	跳过这个示例。通常表示测试代码还没有准备好。

欢迎增加更多 tck case，在 CI/CD 中自动回归所使用的语句。

是否支持 TINKERPOP GREMLIN?

不支持。也没有计划。

是否支持 W3C 的 RDF（SPARQL） 或 GRAPHQL 等?

不支持。也没有计划。

NebulaGraph 的数据模型是属性图，是一个强 Schema 系统，不支持 RDF 标准。

nGQL 也不支持 SPARQL 和 GraphQL。

最后更新: 2025年8月4日

4.1.2 模式

模式（pattern）和图模式匹配，是图查询语言的核心功能，本文介绍 NebulaGraph 设计的各种模式，部分还未实现。

单点模式

点用一对括号来描述，通常包含一个名称。例如：

```
(a)
```

示例为一个简单的模式，描述了单个点，并使用变量 `a` 命名该点。

多点关联模式

多个点通过边相连是常见的结构，模式用箭头来描述两个点之间的边。例如：

```
(a)-[]->(b)
```

示例为一个简单的数据结构：两个点和一条连接两个点的边，两个点分别为 `a` 和 `b`，边是有方向的，从 `a` 到 `b`。

这种描述点和边的方式可以扩展到任意数量的点和边，例如：

```
(a)-[]->(b)<-[]-(c)
```

这样的一系列点和边称为 `路径`（path）。

只有在涉及某个点时，才需要命名这个点。如果不涉及这个点，则可以省略名称，例如：

```
(a)-[]->()<-[]-(c)
```

Tag 模式

> Note
>
> nGQL 中的 `Tag` 概念与 openCypher 中的 `Label` 有一些不同。例如，必须创建一个 `Tag` 之后才能使用它，而且 `Tag` 还定义了属性的类型。

模式除了简单地描述图中的点之外，还可以描述点的 Tag。例如：

```
(a:User)-[]->(b)
```

模式也可以描述有多个 Tag 的点，例如：

```
(a:User:Admin)-[]->(b)
```

属性模式

点和边是图的基本结构。nGQL 在这两种结构上都可以增加属性，方便实现更丰富的模型。

在模式中，属性的表示方式为：用花括号括起一些键值对，用英文逗号分隔，并且需要指定属性所属的 Tag 或者 Edge type。

例如一个点有两个属性：

```
(a:player{name: "Tim Duncan", age: 42})
```

在这个点上可以有一条边是：

```
(a)-[e:follow{degree: 95}]->(b)
```

边模式

描述一条边最简单的方法是使用箭头连接两个点。

可以用以下方式描述边以及它的方向性。如果不关心边的方向，可以省略箭头，例如：

```
(a)-[]-(b)
```

和点一样，边也可以命名。一对方括号用于分隔箭头，变量放在两者之间。例如：

```
(a)-[r]->(b)
```

和点上的 Tag 一样，边也可以有类型。描述边的类型，例如：

```
(a)-[r:REL_TYPE]->(b)
```

和点上的 Tag 不同，一条边只能有一种 Edge type。但是如果我们想描述多个可选 Edge type，可以用管道符号（|）将可选值分开，例如：

```
(a)-[r:TYPE1|TYPE2]->(b)
```

和点一样，边的名称可以省略，例如：

```
(a)-[:REL_TYPE]->(b)
```

变长模式

在图中指定边的长度来描述多条边（以及中间的点）组成的一条长路径，不需要使用多个点和边来描述。例如：

```
(a)-[*2]->(b)
```

该模式描述了 3 点 2 边组成的图，它们都在一条路径上（长度为 2），等价于：

```
(a)-[]->()-[]->(b)
```

也可以指定长度范围，这样的边模式称为 `variable-length edges`，例如：

```
(a)-[*3..5]->(b)
```

`*3..5` 表示最小长度为 3，最大长度为 5。

该模式描述了 4 点 3 边、5 点 4 边或 6 点 5 边组成的图。

也可以指定长度范围的上限或下限，也可以两个都不指定，例如：

```
(a)-[*..5]->(b)  // 最小长度为 1，最大长度为 5。
(a)-[*3..]->(b)  // 最小长度为 3，最大长度为无穷大。
(a)-[*]->(b)     // 最小长度为 1，最大长度为无穷大。
```

路径变量

一系列连接的点和边称为`路径`。nGQL 允许使用变量来命名路径，例如：

```
p = (a)-[*3..5]->(b) // 从 a 到 b， 最小长度为 3，最大长度为 5 的路径
```

可以在 MATCH 语句中使用路径变量。

最后更新: 2025年8月4日

4.1.3 注释

本文介绍 nGQL 中的注释方式。

Examples

```
nebula> RETURN 1+1;     # 这条注释延续到行尾。
nebula> RETURN 1+1;     // 这条注释延续到行尾。
nebula> RETURN 1 /* 这是一条行内注释 */ + 1 == 2;
nebula> RETURN 11 +            \
/* 多行注释       \
用反斜线来换行。   \
*/ 12;
```

> Note
>
> - nGQL 语句中的反斜线（\）代表换行。
> - 如果 nGQL 语句以 `#` 或者 `//` 开头，不会执行该行命令并且会返回提示 `StatementEmpty`。

openCypher 兼容性

- 在 nGQL 中，用户必须在行末使用反斜线（\）来换行，即使是在使用 `/* */` 符号的多行注释内。
- 在 openCypher 中不需要使用反斜线换行。

```
/* openCypher 风格：
这条注释
延续了不止
一行 */
MATCH (n:label)
RETURN n;
```

```
/* 原生 nGQL 风格：  \
这条注释       \
延续了不止     \
一行 */       \
MATCH (n:tag) \
RETURN n;
```

最后更新: 2025年8月4日

4.1.4 大小写区分

标识符区分大小写

以下语句会出现错误，因为 `my_space` 和 `MY_SPACE` 是两个不同的图空间。

```
nebula> CREATE SPACE IF NOT EXISTS my_space (vid_type=FIXED_STRING(30));
nebula> use MY_SPACE;
[ERROR (-1005)]: SpaceNotFound:
```

关键字不区分大小写

以下语句是等价的，因为 `show` 和 `spaces` 是关键字。

```
nebula> show spaces;
nebula> SHOW SPACES;
nebula> SHOW spaces;
nebula> show SPACES;
```

函数不区分大小写

函数名称不区分大小写，例如 `count()` 、 `COUNT()` 、 `couNT()` 是等价的。

```
nebula> WITH [NULL, 1, 1, 2, 2] As a \
        UNWIND a AS b \
        RETURN count(b), COUNT(*), couNT(DISTINCT b);
+----------+----------+-------------------+
| count(b) | COUNT(*) | couNT(distinct b) |
+----------+----------+-------------------+
| 4        | 5        | 2                 |
+----------+----------+-------------------+
```

最后更新: 2025年8月4日

4.1.5 关键字

关键字是 nGQL 中具有特殊含义的词，例如 `CREATE TAG` 语句中的 `CREATE` 和 `TAG`。需要经过特殊处理才能作为标识符使用的关键字被称为保留关键字，而能够直接作为标识符使用的这部分关键字被称为非保留关键字。

不建议在创建 Schema 时使用关键字。如果必须使用关键字，请注意一下规则：

- 使用保留关键字或特殊字符作为标识符时，必须用反引号（`）包围，例如 `AND`。否则，会返回语义错误，创建失败。

- 使用非保留关键字作为标识符时：
- 若其包含大写字母，必须用反引号（`）包围，例如 `Comment`。否则，虽然能创建成功，系统会自动将标识符转化为全小写字母，也即 `comment`。
- 若其为全小写字母，可以不使用反引号（`）包围。

```
nebula> CREATE TAG TAG(name string);
[ERROR (-1004)]: SyntaxError: syntax error near `TAG'

nebula> CREATE TAG `TAG` (name string);
Execution succeeded

nebula> CREATE TAG SPACE(name string);
Execution succeeded

nebula> CREATE TAG 中文(简体 string);
Execution succeeded

nebula> CREATE TAG `￥%特殊 字符&*+-*/` (`q~！（）=  wer` string);
Execution succeeded
```

保留关键字

```
ACROSS
ADD
ALTER
AND
AS
ASC
ASCENDING
BALANCE
BOOL
BY
CASE
CHANGE
COMPACT
CREATE
DATE
DATETIME
DELETE
DESC
DESCENDING
DESCRIBE
DISTINCT
DOUBLE
DOWNLOAD
DROP
DURATION
EDGE
EDGES
EXISTS
EXPLAIN
FALSE
FETCH
FIND
FIXED_STRING
FLOAT
FLUSH
FROM
GEOGRAPHY
GET
GO
GRANT
IF
IGNORE_EXISTED_INDEX
IN
INDEX
INDEXES
INGEST
INSERT
INT
INT16
```

```
INT32
INT64
INT8
INTERSECT
IS
JOIN
LEFT
LIST
LOOKUP
MAP
MATCH
MINUS
NO
NOT
NULL
OF
ON
OR
ORDER
OVER
OVERWRITE
PATH
PROP
REBUILD
RECOVER
REMOVE
RESTART
RETURN
REVERSELY
REVOKE
SET
SHOW
STEP
STEPS
STOP
STRING
SUBMIT
TAG
TAGS
TIME
TIMESTAMP
TO
TRUE
UNION
UNWIND
UPDATE
UPSERT
UPTO
USE
VERTEX
VERTICES
WHEN
WHERE
WITH
XOR
YIELD
```

非保留关键字

```
ACCOUNT
ADMIN
AGENT
ALL
ALLSHORTESTPATHS
ANALYZER
ANY
ATOMIC_EDGE
AUTO
BASIC
BIDIRECT
BOTH
CHARSET
CLEAR
CLIENTS
COLLATE
COLLATION
COMMENT
CONFIGS
CONTAINS
DATA
DBA
DEFAULT
DIVIDE
DRAINER
DRAINERS
ELASTICSEARCH
ELSE
END
ENDS
ES_QUERY
FORCE
```

```
FORMAT
FULLTEXT
GOD
GRANTS
GRAPH
GROUP
GROUPS
GUEST
HDFS
HOST
HOSTS
HTTP
HTTPS
INTO
IP
JOB
JOBS
KILL
LEADER
LIMIT
LINESTRING
LISTENER
LOCAL
MERGE
META
NEW
NOLOOP
NONE
OFFSET
OPTIONAL
OUT
PART
PARTITION_NUM
PARTS
PASSWORD
PLAN
POINT
POLYGON
PROFILE
QUERIES
QUERY
READ
REDUCE
RENAME
REPLICA_FACTOR
RESET
ROLE
ROLES
S2_MAX_CELLS
S2_MAX_LEVEL
SAMPLE
SEARCH
SERVICE
SESSION
SESSIONS
SHORTEST
SHORTESTPATH
SIGN
SINGLE
SKIP
SNAPSHOT
SNAPSHOTS
SPACE
SPACES
STARTS
STATS
STATUS
STORAGE
SUBGRAPH
SYNC
TEXT
TEXT_SEARCH
THEN
TOP
TTL_COL
TTL_DURATION
USER
USERS
UUID
VALUE
VALUES
VARIABLES
VID_TYPE
WHITELIST
WRITE
ZONE
ZONES
```

最后更新: 2025年8月4日

4.1.6 nGQL 风格指南

nGQL 没有严格的构建格式要求，但根据恰当而统一的风格创建 nGQL 语句有利于提高可读性、避免歧义。在同一组织或项目中使用相同的 nGQL 风格有利于降低维护成本，规避因格式混乱或误解造成的问题。本文为写作 nGQL 语句提供了风格参考。

> **Compatibility**
>
> nGQL 风格与 Cypher Style Guide 不同。

换行

1. 换行写子句。

   不推荐：

   ```
   GO FROM "player100" OVER follow REVERSELY YIELD src(edge) AS id;
   ```

   推荐：

   ```
   GO FROM "player100" \
   OVER follow REVERSELY \
   YIELD src(edge) AS id;
   ```

2. 换行写复合语句中的不同语句。

   不推荐：

   ```
   GO FROM "player100" OVER follow REVERSELY YIELD src(edge) AS id | GO FROM $-.id \
   OVER serve WHERE properties($^).age > 20 YIELD properties($^).name AS FriendOf, properties($$).name AS Team;
   ```

   推荐：

   ```
   GO FROM "player100" \
   OVER follow REVERSELY \
   YIELD src(edge) AS id | \
   GO FROM $-.id OVER serve \
   WHERE properties($^).age > 20 \
   YIELD properties($^).name AS FriendOf, properties($$).name AS Team;
   ```

3. 子句长度超过 80 个字符时，在合适的位置换行。

   不推荐：

   ```
   MATCH (v:player{name:"Tim Duncan"})-[e]->(v2) \
   WHERE (v2.player.name STARTS WITH "Y" AND v2.player.age > 35 AND v2.player.age < v.player.age) OR (v2.player.name STARTS WITH "T" AND v2.player.age < 45 AND v2.player.age > v.player.age) \
   RETURN v2;
   ```

   推荐：

   ```
   MATCH (v:player{name:"Tim Duncan"})-[e]->(v2) \
   WHERE (v2.player.name STARTS WITH "Y" AND v2.player.age > 35 AND v2.player.age < v.player.age) \
   OR (v2.player.name STARTS WITH "T" AND v2.player.age < 45 AND v2.player.age > v.player.age) \
   RETURN v2;
   ```

> **Note**
>
> 即使子句不超过 80 个字符，如需换行后有助于理解，也可将子句再次分行。

标识符命名

在 nGQL 语句中，关键字、标点符号、空格以外的字符内容都是标识符。推荐的标识符命名方式如下。

1. 使用单数名词命名 Tag，用原型动词或动词短语构成 Edge type。

   不推荐：

   ```
   MATCH p=(v:players)-[e:are_following]-(v2) \
   RETURN nodes(p);
   ```

   推荐：

   ```
   MATCH p=(v:player)-[e:follow]-(v2) \
   RETURN nodes(p);
   ```

2. 标识符用蛇形命名法，以下划线（_）连接单词，且所有字母小写。

   不推荐：

   ```
   MATCH (v:basketballTeam) \
   RETURN v;
   ```

   推荐：

   ```
   MATCH (v:basketball_team) \
   RETURN v;
   ```

3. 语法关键词大写，变量小写。

   不推荐：

   ```
   match (V:player) return V limit 5;
   ```

   推荐：

   ```
   MATCH (v:player) RETURN v LIMIT 5;
   ```

Pattern

1. 分行写 Pattern 时，在表示边的箭头右侧换行，而不是左侧。

   不推荐：

   ```
   MATCH (v:player{name: "Tim Duncan", age: 42}) \
   -[e:follow]->()-[e2:serve]->()<--(v2) \
   RETURN v, e, v2;
   ```

   推荐：

   ```
   MATCH (v:player{name: "Tim Duncan", age: 42})-[e:follow]-> \
   ()-[e2:serve]->()<--(v2) \
   RETURN v, e, v2;
   ```

2. 将无需查询的点和边匿名化。

   不推荐：

   ```
   MATCH (v:player)-[e:follow]->(v2) \
   RETURN v;
   ```

   推荐：

   ```
   MATCH (v:player)-[:follow]->() \
   RETURN v;
   ```

3. 将非匿名点放在匿名点的前面。

   不推荐：

```
MATCH ()-[:follow]->(v) \
RETURN v;
```

推荐：

```
MATCH (v)<-[:follow]-() \
RETURN v;
```

字符串

字符串用双引号包围。

不推荐：

```
RETURN 'Hello Nebula!';
```

推荐：

```
RETURN "Hello Nebula!\"123\"";
```

> Note
>
> 字符串中需要嵌套单引号或双引号时，用反斜线（\）转义。例如：
>
> ```
> RETURN "\"The database is amazing,\" the user says.";
> ```

结束语句

1. 用英文分号（;）结束 nGQL 语句。

   不推荐：

   ```
   FETCH PROP ON player "player100" YIELD properties(vertex)
   ```

   推荐：

   ```
   FETCH PROP ON player "player100" YIELD properties(vertex);
   ```

2. 使用管道符（|）分隔的复合语句，仅在最后一行末用英文分号结尾。在管道符前使用英文分号会导致语句执行失败。

   不支持：

   ```
   GO FROM "player100" \
   OVER follow \
   YIELD dst(edge) AS id; | \
   GO FROM $-.id \
   OVER serve \
   YIELD properties($$).name AS Team, properties($^).name AS Player;
   ```

   支持：

   ```
   GO FROM "player100" \
   OVER follow \
   YIELD dst(edge) AS id | \
   GO FROM $-.id \
   OVER serve \
   YIELD properties($$).name AS Team, properties($^).name AS Player;
   ```

3. 在包含自定义变量的复合语句中，用英文分号结束定义变量的语句。不按规则加分号或使用管道符结束该语句会导致执行失败。

   不支持：

   ```
   $var = GO FROM "player100" \
   OVER follow \
   YIELD dst(edge) AS id \
   GO FROM $var.id \
   OVER serve \
   YIELD properties($$).name AS Team, properties($^).name AS Player;
   ```

也不支持：

```
$var = GO FROM "player100" \
OVER follow \
YIELD dst(edge) AS id | \
GO FROM $var.id \
OVER serve \
YIELD properties($$).name AS Team, properties($^).name AS Player;
```

支持：

```
$var = GO FROM "player100" \
OVER follow \
YIELD dst(edge) AS id; \
GO FROM $var.id \
OVER serve \
YIELD properties($$).name AS Team, properties($^).name AS Player;
```

最后更新: 2025年8月4日

4.2 数据类型

4.2.1 数值

nGQL 支持整数和浮点数。

整数

nGQL 支持带符号的 64 位整数（INT64）、32 位整数（INT32）、16 位整数（INT16）和 8 位整数（INT8）。

类型	声明关键字	范围
INT64	`INT64` 或 `INT`	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
INT32	`INT32`	-2,147,483,648 ~ 2,147,483,647
INT16	`INT16`	-32,768 ~ 32,767
INT8	`INT8`	-128 ~ 127

浮点数

nGQL 支持单精度浮点（FLOAT）和双精度浮点（DOUBLE）。

类型	声明关键字	范围	精度
FLOAT	`FLOAT`	3.4E +/- 38	6~7 位
DOUBLE	`DOUBLE`	1.7E +/- 308	15~16 位

nGQL 支持科学计数法，例如 `1e2`、`1.1e2`、`.3e4`、`1.e4`、`-1234E-10`。

Note

不支持 MySQL 中的 DECIMAL 数据类型。

数值的读写

在写入和读取不同类型的数据时，nGQL 的行为遵守以下规则：

数值类型	设置为 VID	设置为属性类型	读取该类型的属性值得到的类型
INT64	支持	支持	INT64
INT32	不支持	支持	INT64
INT16	不支持	支持	INT64
INT8	不支持	支持	INT64
FLOAT	不支持	支持	DOUBLE
DOUBLE	不支持	支持	DOUBLE

例如，nGQL 不支持设置 INT8 类型的 VID，但支持将 TAG 或 Edge type 的某个属性类型设置为 INT8。当使用 nGQL 语句读取 INT8 类型的属性时，获取到的值的类型为 INT64。

- NebulaGraph 支持写入多种进制的数值：
- 十进制，例如 `123456` 。
- 十六进制，例如 `0x1e240` 。
- 八进制，例如 `0361100` 。

  但 NebulaGraph 会将写入的非十进制数值解析为十进制的值保存。读取到的值为十进制。

  例如，属性 `score` 的类型为 `INT` ，通过 INSERT 语句为其赋值 `0xb` ，使用 FETCH 等语句查询该属性值获取到的结果是 `11` ，即将十六进制的 `0xb` 转换为十进制后的值。

- 将 FLOAT/DOUBLE 类型的数值插入 INT 类型的列，会将数值四舍五入取整。

最后更新: 2025年8月4日

4.2.2 布尔

NebulaGraph 使用关键字 `BOOL` 声明布尔数据类型，可选值为 `true` 或 `false` 。

nGQL 支持以如下方式使用布尔值：

- 将属性值的数据类型定义为布尔。
- 在 `WHERE` 子句中用布尔值作为判断条件。

最后更新: 2025年8月4日

4.2.3 字符串

NebulaGraph 支持定长字符串和变长字符串。

声明与表示方式

nGQL 中的字符串声明方式如下：

- 使用关键字 `STRING` 声明变长字符串。
- 使用关键字 `FIXED_STRING(<length>)` 声明定长字符串， `<length>` 为字符串长度，例如 `FIXED_STRING(32)` 。

字符串的表示方式为用双引号或单引号包裹，例如 `"Hello, Cooper"` 或 `'Hello, Cooper'` 。

字符串读写

nGQL 支持以如下方式使用字符串：

- 将 VID 的数据类型定义为定长字符串。
- 将变长字符串设置为 Schema 名称，包括图空间、Tag、Edge type 和属性的名称。
- 将属性值的数据类型定义为定长或变长字符串。

例如：

- 将属性值的类型定义为定长字符串

```
nebula> CREATE TAG IF NOT EXISTS t1 (p1 FIXED_STRING(10));
```

- 将属性值的类型定义为变长字符串

```
nebula> CREATE TAG IF NOT EXISTS t2 (p2 STRING);
```

如果尝试写入的定长字符串超出长度限制：

- 当该定长字符串为属性值时，写入会成功，NebulaGraph 将截断字符串，仅存入符合长度限制的部分。
- 当该定长字符串为 VID 时，写入会失败，NebulaGraph 将报错。

转义字符

反斜杠（ `\` ）在字符串中为转义字符，用于表示特殊字符。

例如，如果想在一个字符串中包含一个双引号(`"`), 不能直接写 `"Hello "world""` ，因为这会引起语法错误。相反，需要使用反斜杠（ `\` ）来转义双引号，如 `"Hello \"world\""` 。

```
nebula> RETURN "Hello \"world\""
+-----------------+
| "Hello "world"" |
+-----------------+
| "Hello "world"" |
+-----------------+
```

同样，反斜杠本身也需要转义，因为它是一个特殊字符。例如，要在字符串中包含一个反斜杠，需要写成 `"Hello \\ world"` 。

```
nebula> RETURN "Hello \\ world"
+-----------------+
| "Hello \ world" |
+-----------------+
| "Hello \ world" |
+-----------------+
```

更多转义字符的示例，请参见 Escape character examples。

OpenCypher 兼容性

openCypher、Cypher 和 nGQL 之间有一些细微区别，例如下面 openCypher 的示例，不能将单引号替换为双引号。

```
# File: Literals.feature
Feature: Literals

Background:
    Given any graph
 Scenario: Return a single-quoted string
    When executing query:
      """
      RETURN '' AS literal
      """
    Then the result should be, in any order:
      | literal |
      | ''      |    # Note: it should return single-quotes as openCypher required.
    And no side effects
```

Cypher 的返回结果同时支持单引号和双引号，nGQL 遵循 Cypher 的方式。

```
nebula > YIELD '' AS quote1, "" AS quote2, "'" AS quote3, '"' AS quote4
+--------+--------+--------+--------+
| quote1 | quote2 | quote3 | quote4 |
+--------+--------+--------+--------+
| ""     | ""     | "'"    | """    |
+--------+--------+--------+--------+
```

最后更新: 2025年8月4日

4.2.4 日期和时间类型

本文介绍日期和时间的类型，包括 `DATE` 、 `TIME` 、 `DATETIME` 、 `TIMESTAMP` 和 `DURATION` 。

注意事项

- 在插入时间类型的属性值时，NebulaGraph 会根据配置文件中 `timezone_name` 参数指定的时区，将该 `DATE` 、 `TIME` 、 `DATETIME` 转换成相应的世界协调时间（UTC）时间。

> Note
>
> 如需修改当前时区，请同时修改所有服务的配置文件中的 `timezone_name` 参数。

- 函数 `date()` 、 `time()` 和 `datetime()` 可以指定时区进行转换，例如 `datetime("2017-03-04 22:30:40.003000+08:00")` 或 `datetime("2017-03-04T22:30:40.003000[Asia/Shanghai]")` 。
- 函数 `date()` 、 `time()` 、 `datetime()` 和 `timestamp()` 可以用空值获取当前的日期或时间。
- 函数 `date()` 、 `time()` 、 `datetime()` 和 `duration()` 可以用属性名称获取自身的某一个具体属性值，例如 `date().month` 获取当前月份、 `time("02:59:40").minute` 获取传入时间的分钟数。
- 进行时间运算时建议使用 `duration()` 计算时刻的偏移。此外还支持 `date()` 和 `date()` 的加减、 `timestamp()` 和 `timestamp()` 的加减。
- 设置时间的年份为负数时，需要使用 Map 类型数据。

openCypher 兼容性

- 支持年、月、日、时、分、秒、毫秒、微秒，不支持纳秒。
- 不支持函数 `localdatetime()` 。
- 不支持大部分字符串时间格式，支持 `YYYY-MM-DDThh:mm:ss` 和 `YYYY-MM-DD hh:mm:ss` 。
- 支持单个数字的字符串时间格式，例如 `time("1:1:1")` 。

DATE

`DATE` 包含日期，但是不包含时间。 NebulaGraph 检索和显示 `DATE` 的格式为 `YYYY-MM-DD` 。支持的范围是 `-32768-01-01` 到 `32767-12-31` 。

`date()` 支持的属性名称包括 `year` 、 `month` 和 `day` 。 `date()` 支持输入 `YYYY` 、 `YYYY-MM` 或 `YYYY-MM-DD` ，未输入的月份或日期默认为 `01` 。

```
nebula> RETURN DATE({year:-123, month:12, day:3});
+------------------------------------+
| date({year:-(123),month:12,day:3}) |
+------------------------------------+
| -123-12-03                         |
+------------------------------------+

nebula> RETURN DATE("23333");
+---------------+
| date("23333") |
+---------------+
| 23333-01-01   |
+---------------+

nebula> RETURN DATE("2023-12-12") - DATE("2023-12-11");
+-----------------------------------------+
| (date("2023-12-12")-date("2023-12-11")) |
+-----------------------------------------+
| 1                                       |
+-----------------------------------------+
```

TIME

`TIME` 包含时间，但是不包含日期。 NebulaGraph 检索和显示 `TIME` 的格式为 `hh:mm:ss.msmsmsususus` 。支持的范围是 `00:00:00.000000` 到 `23:59:59.999999` 。

`time()` 支持的属性名称包括 `hour` 、 `minute` 和 `second` 。

DATETIME

`DATETIME` 包含日期和时间。 NebulaGraph 检索和显示 `DATETIME` 的格式为 `YYYY-MM-DDThh:mm:ss.msmsmsususus` 。支持的范围是 `-32768-01-01T00:00:00.000000` 到 `32767-12-31T23:59:59.999999` 。

- `datetime()` 支持的属性名称包括 `year` 、 `month` 、 `day` 、 `hour` 、 `minute` 和 `second` 。
- `datetime()` 可将 `TIMESTAMP` 类型的日期值转换成 `DATETIME` 类型的日期值。 `TIMESTAMP` 类型的日期值取值范围： `0~9223372036` 。
- `datetime()` 支持 `int` 类型的参数，该 `int` 参数表示时间戳。

```
# 获取当前时间。
nebula> RETURN datetime();
+----------------------------+
| datetime()                 |
+----------------------------+
| 2022-08-29T06:37:08.933000 |
+----------------------------+

# 获取当前时间的小时。
nebula> RETURN datetime().hour;
+-----------------+
| datetime().hour |
+-----------------+
| 6               |
+-----------------+

# 将时间戳转换成 DATETIME 类型的格式。
nebula> RETURN datetime(timestamp(1625469277));
+---------------------------------+
| datetime(timestamp(1625469277)) |
+---------------------------------+
| 2021-07-05T07:14:37.000000      |
+---------------------------------+

nebula> RETURN datetime(1625469277);
+----------------------------+
| datetime(1625469277)       |
+----------------------------+
| 2021-07-05T07:14:37.000000 |
+----------------------------+
```

TIMESTAMP

`TIMESTAMP` 包含日期和时间。支持的范围是 UTC 时间的 `1970-01-01T00:00:01` 到 `2262-04-11T23:47:16` 。

`TIMESTAMP` 还有以下特点：

- 以时间戳形式存储和显示。例如 `1615974839` ，表示 `2021-03-17T17:53:59` 。
- 查询 `TIMESTAMP` 类型属性值的方式包括时间戳整数和 `timestamp()` 函数。
- 插入 `TIMESTAMP` 类型属性值的方式包括时间戳整数、 `timestamp()` 函数和 `now()` 函数。
- `timestamp()` 函数支持传入空值获取当前时间戳；同时支持传入整数以标识该整数为时间戳，整数取值： `0~9223372036` 。
- `timestamp()` 函数可将 `DATETIME` 类型的日期值转换成 `TIMESTAMP` 类型的日期值，且传入的 `DATETIME` 类型的日期值为 `string` 类型。
- 底层存储的数据格式为 64 位 int。

```
# 传入当前时间。
nebula> RETURN timestamp();
+-------------+
| timestamp() |
+-------------+
| 1625469277  |
+-------------+

# 传入指定时间。
nebula> RETURN timestamp("2022-01-05T06:18:43");
+----------------------------------+
```

```
| timestamp("2022-01-05T06:18:43") |
+----------------------------------+
| 1641363523                       |
+----------------------------------+

# 传入 datetime()。
nebula> RETURN timestamp(datetime("2022-08-29T07:53:10.939000"));
+---------------------------------------------------+
| timestamp(datetime("2022-08-29T07:53:10.939000")) |
+---------------------------------------------------+
| 1661759590                                        |
+---------------------------------------------------+
```

> Note
>
> 传入 `timestamp()` 函数的时间字符串不支持包含毫秒和微秒，但是通过 `timestamp(datetime())` 传入的时间字符串支持包含毫秒和微秒。

DURATION

`DURATION` 是一段连续的时间，由 `years`、 `months`、 `days`、 `hours`、 `minutes`、 `seconds` 六个Key自由组合成的Map类型数据表示。例如 `duration({years: 12, months: 5, days: 14, hours: 16, minutes: 12, seconds: 70})`。

`DURATION` 还有以下特点：

- 不支持为 `DURATION` 类型数据创建索引。
- 可以用于对指定时间进行计算。

示例

1. 创建 Tag，名称为 `date1`，包含 `DATE`、 `TIME` 和 `DATETIME` 三种类型。

```
nebula> CREATE TAG IF NOT EXISTS date1(p1 date, p2 time, p3 datetime);
```

2. 插入点，名称为 `test1`。

```
nebula> INSERT VERTEX date1(p1, p2, p3) VALUES "test1":(date("2021-03-17"), time("17:53:59"), datetime("2017-03-04T22:30:40.003000[Asia/Shanghai]"));
```

3. 查询 `test1` 的属性 `p1` 是否为 `2021-03-17`。

```
nebula> MATCH (v:date1) RETURN v.date1.p1 == date("2021-03-17");
+----------------------------------+
| (v.date1.p1==date("2021-03-17")) |
+----------------------------------+
| true                             |
+----------------------------------+
```

4. 获取 `test1` 的属性 `p1` 的月份。

```
nebula> CREATE TAG INDEX IF NOT EXISTS date1_index ON date1(p1);
nebula> REBUILD TAG INDEX date1_index;
nebula> MATCH (v:date1) RETURN v.date1.p1.month;
+------------------+
| v.date1.p1.month |
+------------------+
| 3                |
+------------------+
```

5. 查找 Tag `date1` 中属性 `p3` 小于 `2023-01-01T00:00:00.000000` 的值。

```
nebula> MATCH (v:date1)  \
WHERE v.date1.p3 < datetime("2023-01-01T00:00:00.000000") \
RETURN v.date1.p3;
+----------------------------+
| v.date1.p3                 |
+----------------------------+
| 2017-03-04T14:30:40.003000 |
+----------------------------+
```

6. 创建 Tag，名称为 `school`，包含 `TIMESTAMP` 类型。

```
nebula> CREATE TAG IF NOT EXISTS school(name string , found_time timestamp);
```

7. 插入点，名称为 `DUT`，存储时间为 `"1988-03-01T08:00:00"`。

```
# 时间戳形式插入，1988-03-01T08:00:00 对应的时间戳为 573177600，转换为 UTC 时间为 573206400。
nebula> INSERT VERTEX school(name, found_time) VALUES "DUT":("DUT", 573206400);

# 日期和时间格式插入。
nebula> INSERT VERTEX school(name, found_time) VALUES "DUT":("DUT", timestamp("1988-03-01T08:00:00"));
```

8. 插入点，名称为 `dut`，用 `now()` 或 `timestamp()` 函数存储时间。

```
# 用 now() 函数存储时间
nebula> INSERT VERTEX school(name, found_time) VALUES "dut":("dut", now());

# 用 timestamp() 函数存储时间
nebula> INSERT VERTEX school(name, found_time) VALUES "dut":("dut", timestamp());
```

还可以使用 `WITH` 语句设置具体日期时间或进行计算，例如：

```
nebula> WITH time({hour: 12, minute: 31, second: 14, millisecond:111, microsecond: 222}) AS d RETURN d;
+-----------------+
| d               |
+-----------------+
| 12:31:14.111222 |
+-----------------+

nebula> WITH date({year: 1984, month: 10, day: 11}) AS x RETURN x + 1;
+------------+
| (x+1)      |
+------------+
| 1984-10-12 |
+------------+

nebula> WITH date('1984-10-11') as x, duration({years: 12, days: 14, hours: 99, minutes: 12}) as d \
        RETURN x + d AS sum, x - d AS diff;
+------------+------------+
| sum        | diff       |
+------------+------------+
| 1996-10-29 | 1972-09-23 |
+------------+------------+
```

最后更新: 2025年8月4日

4.2.5 NULL

默认情况下，插入点或边时，属性值可以为 `NULL`，用户也可以设置属性值不允许为 `NULL`（`NOT NULL`），即插入点或边时必须设置该属性的值，除非创建属性时已经设置默认值。

NULL 的逻辑操作

`AND`、`OR`、`XOR` 和 `NOT` 的真值表如下。

a	b	a AND b	a OR b	a XOR b	NOT a
false	false	false	false	false	true
false	null	false	null	null	true
false	true	false	true	true	true
true	false	false	true	true	false
true	null	null	true	null	false
true	true	true	true	false	false
null	false	false	null	null	null
null	null	null	null	null	null
null	true	null	true	null	null

OpenCypher 兼容性

NebulaGraph 中，NULL 的比较和操作与 openCypher 不同，后续也可能会有变化。

NULL 的比较

NebulaGraph 中，NULL 的比较操作不兼容 openCypher。

NULL 的操作和返回

NebulaGraph 中，对 NULL 的操作以及返回结果不兼容 openCypher。

示例

使用 NOT NULL

创建 Tag，名称为 `player`，指定属性 `name` 为 `NOT NULL`。

```
nebula> CREATE TAG IF NOT EXISTS player(name string NOT NULL, age int);
```

使用 `SHOW` 命令查看创建 Tag 语句，属性 `name` 为 `NOT NULL`，属性 `age` 为默认的 `NULL`。

```
nebula> SHOW CREATE TAG player;
+-----------+-----------------------------------+
| Tag       | Create Tag                        |
+-----------+-----------------------------------+
| "student" | "CREATE TAG `player` (           |
|           |  `name` string NOT NULL,          |
|           |  `age` int64 NULL                 |
|           | ) ttl_duration = 0, ttl_col = """ |
+-----------+-----------------------------------+
```

插入点 `Kobe`，属性 `age` 可以为 `NULL`。

```
nebula> INSERT VERTEX player(name, age) VALUES "Kobe":("Kobe",null);
```

使用 **NOT NULL** 并设置默认值

创建 Tag，名称为 `player`，指定属性 `age` 为 `NOT NULL`，并设置默认值 `18`。

```
nebula> CREATE TAG IF NOT EXISTS player(name string, age int NOT NULL DEFAULT 18);
```

插入点 `Kobe`，只设置属性 `name`。

```
nebula> INSERT VERTEX player(name) VALUES "Kobe":("Kobe");
```

查询点 `Kobe`，属性 `age` 为默认值 `18`。

```
nebula> FETCH PROP ON player "Kobe" YIELD properties(vertex);
+--------------------------+
| properties(VERTEX)       |
+--------------------------+
| {age: 18, name: "Kobe"}  |
+--------------------------+
```

最后更新: 2025年8月4日

4.2.6 列表

列表（List）是复合数据类型，一个列表是一组元素的序列，可以通过元素在序列中的位置访问列表中的元素。

列表用左方括号（[）和右方括号（]）包裹多个元素，各个元素之间用英文逗号（,）隔开。元素前后的空格在列表中被忽略，因此可以使用换行符、制表符和空格调整格式。

OpenCypher 兼容性

复合数据类型（例如 List、Set、Map）不能存储为点或边的属性。

列表操作

对列表进行操作可以使用预设的列表函数，也可以使用下标表达式过滤列表内的元素。

下标表达式语法

```
[M]
[M..N]
[M..]
[..N]
```

nGQL 的下标支持从前往后查询，从 0 开始，0 表示第一个元素，1 表示第二个元素，以此类推；也支持从后往前查询，从-1 开始，-1 表示最后一个元素，-2 表示倒数第二个元素，以此类推。

- [M]：表示下标为 M 的元素。
- [M..N]：表示 `M ≤ 下标 < N` 的元素。`N` 为 0 时，返回为空。
- [M..]：表示 `M ≤ 下标` 的元素。
- [..N]：表示 `下标 < N` 的元素。`N` 为 0 时，返回为空。

> Note
>
> - 越界的下标返回为空，未越界的可以正常返回。
> - `M ≥ N` 时，返回为空。
> - 查询单个元素时，如果 `M` 为 null，返回报错 `BAD_TYPE`；范围查询时，`M` 或 `N` 为 null，返回为 `null`。

示例

```
# 返回列表 [1,2,3]
nebula> RETURN list[1, 2, 3] AS a;
+-----------+
| a         |
+-----------+
| [1, 2, 3] |
+-----------+

# 返回列表 [1,2,3,4,5] 中位置下标为 3 的元素。列表的位置下标是从 0 开始，因此返回的元素为 4。
nebula> RETURN range(1,5)[3];
+---------------+
| range(1,5)[3] |
+---------------+
| 4             |
+---------------+

# 返回列表 [1,2,3,4,5] 中位置下标为-2 的元素。列表的最后一个元素的位置下标是-1，因此-2 是指倒数第二个元素，即 4。
nebula> RETURN range(1,5)[-2];
+------------------+
| range(1,5)[-(2)] |
+------------------+
| 4                |
+------------------+

# 返回列表 [1,2,3,4,5] 中下标位置从 0 到 3（不包括 3）的元素。
nebula> RETURN range(1,5)[0..3];
+------------------+
| range(1,5)[0..3] |
+------------------+
```

```
| [1, 2, 3]        |
+------------------+

# 返回列表 [1,2,3,4,5] 中位置下标大于 2 的元素。
nebula> RETURN range(1,5)[3..] AS a;
+--------+
| a      |
+--------+
| [4, 5] |
+--------+

# 返回列表内下标小于 3 的元素。
nebula> WITH list[1, 2, 3, 4, 5] AS a \
        RETURN a[..3] AS r;
+-----------+
| r         |
+-----------+
| [1, 2, 3] |
+-----------+

# 筛选列表 [1,2,3,4,5] 中大于 2 的元素，将这些元素分别做运算并返回。
nebula> RETURN [n IN range(1,5) WHERE n > 2 | n + 10] AS a;
+--------------+
| a            |
+--------------+
| [13, 14, 15] |
+--------------+

# 返回列表内第一个至倒数第二个（包括）的元素。
nebula> YIELD list[1, 2, 3][0..-1] AS a;
+--------+
| a      |
+--------+
| [1, 2] |
+--------+

# 返回列表内倒数第三个至倒数第一个（不包括）的元素。
nebula> YIELD list[1, 2, 3, 4, 5][-3..-1] AS a;
+--------+
| a      |
+--------+
| [3, 4] |
+--------+

# 设置变量，返回列表内下标为 1、2 的元素。
nebula> $var = YIELD 1 AS f, 3 AS t; \
        YIELD list[1, 2, 3][$var.f..$var.t] AS a;
+--------+
| a      |
+--------+
| [2, 3] |
+--------+

# 越界的下标返回为空，未越界的可以正常返回。
nebula> RETURN list[1, 2, 3, 4, 5] [0..10] AS a;
+-----------------+
| a               |
+-----------------+
| [1, 2, 3, 4, 5] |
+-----------------+

nebula> RETURN list[1, 2, 3] [-5..5] AS a;
+-----------+
| a         |
+-----------+
| [1, 2, 3] |
+-----------+

# [0..0] 时返回为空。
nebula> RETURN list[1, 2, 3, 4, 5] [0..0] AS a;
+----+
| a  |
+----+
| [] |
+----+

# M ≥ N 时，返回为空。
nebula> RETURN list[1, 2, 3, 4, 5] [3..1] AS a;
+----+
| a  |
+----+
| [] |
+----+

# 范围查询时，下标有 null 时，返回为 null。
nebula> WITH list[1,2,3] AS a \
        RETURN a[0..null] as r;
+----------+
| r        |
+----------+
| __NULL__ |
+----------+

# 将列表 [1,2,3,4,5] 中的元素分别做运算，然后将列表去掉表头并返回。
```

```
nebula> RETURN tail([n IN range(1, 5) | 2 * n - 10]) AS a;
+-----------------+
| a               |
+-----------------+
| [-6, -4, -2, 0] |
+-----------------+

# 将列表 [1,2,3] 中的元素判断为真，然后返回。
nebula> RETURN [n IN range(1, 3) WHERE true | n] AS r;
+-----------+
| r         |
+-----------+
| [1, 2, 3] |
+-----------+

# 返回列表 [1,2,3] 的长度。
nebula> RETURN size(list[1,2,3]);
+---------------+
| size([1,2,3]) |
+---------------+
| 3             |
+---------------+

# 将列表 [92,90] 中的元素做运算，然后在 where 子句中进行条件判断。
nebula> GO FROM "player100" OVER follow WHERE properties(edge).degree NOT IN [x IN [92, 90] | x + $$.player.age] \
        YIELD dst(edge) AS id, properties(edge).degree AS degree;
+-------------+--------+
| id          | degree |
+-------------+--------+
| "player101" | 95     |
| "player102" | 90     |
+-------------+--------+

# 将 MATCH 语句的查询结果作为列表中的元素进行运算并返回。
nebula> MATCH p = (n:player{name:"Tim Duncan"})-[:follow]->(m) \
        RETURN [n IN nodes(p) | n.player.age + 100] AS r;
+------------+
| r          |
+------------+
| [142, 136] |
| [142, 141] |
+------------+
```

OpenCypher 兼容性

- 在 openCypher 中，查询越界元素时返回 `null`，而在 nGQL 中，查询单个越界元素时返回 `OUT_OF_RANGE`。

```
nebula> RETURN range(0,5)[-12];
+-------------------+
| range(0,5)[-(12)] |
+-------------------+
| OUT_OF_RANGE      |
+-------------------+
```

- 复合数据类型（例如 set、map、list）不能存储为点或边的属性。
- 建议修改图建模方式：将复合数据类型建模为点的邻边，而不是该点的自身属性，每条邻边可以动态增删，并且可以设置邻边的 Rank 值来控制邻边的顺序。

- List 中不支持 pattern，例如 `[(src)-[]->(m) | m.name]`。

最后更新: 2025年8月4日

4.2.7 集合

集合（Set）是复合数据类型，集合中是一组元素，与列表（List）不同的是，集合中的元素是无序的，且不允许重复。

集合用左花括号（{）和右花括号（}）包裹多个元素，各个元素之间用英文逗号（,）隔开。元素前后的空格在集合中被忽略，因此可以使用换行符、制表符和空格调整格式。

OpenCypher 兼容性

- 复合数据类型（例如 List、Set、Map）不能存储为点或边的属性。
- 在 OpenCypher 中，集合不是一个数据类型，而在 nGQL 中，用户可以使用集合。

示例

```
# 返回集合 {1,2,3}。
nebula> RETURN set{1, 2, 3} AS a;
+-----------+
| a         |
+-----------+
| {3, 2, 1} |
+-----------+

# 返回集合 {1,2,1}，因为集合不允许重复元素，会返回 {1,2}，且顺序是无序的。
nebula> RETURN set{1, 2, 1} AS a;
+--------+
| a      |
+--------+
| {2, 1} |
+--------+

# 判断集合中是否有指定元素 1。
nebula> RETURN 1 IN set{1, 2} AS a;
+------+
| a    |
+------+
| true |
+------+

# 计算集合中的元素数量。
nebula> YIELD size(set{1, 2, 1}) AS a;
+---+
| a |
+---+
| 2 |
+---+

# 返回目标点属性值组成的集合。
nebula> GO FROM "player100" OVER follow \
        YIELD set{properties($$).name,properties($$).age} as a;
+-----------------------+
| a                     |
+-----------------------+
| {36, "Tony Parker"}   |
| {41, "Manu Ginobili"} |
+-----------------------+
```

最后更新: 2025年8月4日

4.2.8 映射

映射（Map）是复合数据类型。一个映射是一组键值对（Key-Value）的无序集合。在映射中，Key 是字符串类型，Value 可以是任何数据类型。用户可以通过 `map['<key>']` 的方法获取映射中的元素。

映射用左花括号（{）和右花括号（}）包裹多个键值对，各个键值对之间用英文逗号（,）隔开。键值对前后的空格在映射中被忽略，因此可以使用换行符、制表符和空格调整格式。

OpenCypher 兼容性

- 复合数据类型（例如 List、Set、Map）不能存储为点或边的属性。
- 不支持映射投影（map projection）。

示例

```
# 返回简单的映射。
nebula> YIELD map{key1: 'Value1', Key2: 'Value2'} as a;
+----------------------------------+
| a                                |
+----------------------------------+
| {Key2: "Value2", key1: "Value1"} |
+----------------------------------+

# 返回列表类型的映射。
nebula> YIELD map{listKey: [{inner: 'Map1'}, {inner: 'Map2'}]} as a;
+-----------------------------------------------+
| a                                             |
+-----------------------------------------------+
| {listKey: [{inner: "Map1"}, {inner: "Map2"}]} |
+-----------------------------------------------+

# 返回混合类型的映射。
nebula> RETURN map{a: LIST[1,2], b: SET{1,2,1}, c: "hee"} as a;
+----------------------------------+
| a                                |
+----------------------------------+
| {a: [1, 2], b: {2, 1}, c: "hee"} |
+----------------------------------+

# 返回映射中的指定元素。
nebula> RETURN map{a: LIST[1,2], b: SET{1,2,1}, c: "hee"}["b"] AS b;
+--------+
| b      |
+--------+
| {2, 1} |
+--------+

# 判断映射中是否有指定key，暂不支持判断value。
nebula> RETURN "a" IN MAP{a:1, b:2} AS a;
+------+
| a    |
+------+
| true |
+------+
```

最后更新: 2025年8月4日

4.2.9 类型转换

类型转换是指将表达式的类型转换为另一个类型。

NebulaGraph 支持显式地转换类型。详情参见类型转换函数。

示例

```
# 将列表拆分，并各自转换为布尔值显示。
nebula> UNWIND [true, false, 'true', 'false', NULL] AS b \
        RETURN toBoolean(b) AS b;
+----------+
| b        |
+----------+
| true     |
+----------+
| false    |
+----------+
| true     |
+----------+
| false    |
+----------+
| __NULL__ |
+----------+

# 将整数或字符串转换为浮点数。
nebula> RETURN toFloat(1), toFloat('1.3'), toFloat('1e3'), toFloat('not a number');
+------------+----------------+----------------+------------------------+
| toFloat(1) | toFloat("1.3") | toFloat("1e3") | toFloat("not a number") |
+------------+----------------+----------------+------------------------+
| 1.0        | 1.3            | 1000.0         | __NULL__               |
+------------+----------------+----------------+------------------------+
```

最后更新: 2025年8月4日

4.2.10 地理空间

地理空间（GEOGRAPHY）是由经纬度构成的表示地理空间信息的数据类型。 NebulaGraph 当前支持简单地理要素中的 Point、LineString 和 Polygon 三种地理形状。支持 SQL-MM 3 中的部分核心 geo 解析、构造、格式设置、转换、谓词和度量等函数。

GEOGRAPHY

GEOGRAPHY 的基本类型是点，由经纬度确定一个点，例如 `"POINT(3 8)"` 表示经度为 `3°`，纬度为 `8°`。多个点可以构成线段或多边形。

类型	示例	说明
Point	`"POINT(3 8)"`	点类型
LineString	`"LINESTRING(3 8, 4.7 73.23)"`	线段类型
Polygon	`"POLYGON((0 1, 1 2, 2 3, 0 1))"`	多边形类型

Note

请勿直接插入上述类型的 GEOGRAPHY 数据，例如 `INSERT VERTEX any_shape(geo) VALUES "1":("POINT(1 1)")`，需要使用 geo 函数指定数据类型后才能插入，例如 `INSERT VERTEX any_shape(geo) VALUES "1":(ST_GeogFromText("POINT(1 1)"));`。

示例

```
//创建 Tag，允许存储任意形状地理空间数据类型。
nebula> CREATE TAG IF NOT EXISTS any_shape(geo geography);

//创建 Tag，只允许存储点形状地理空间数据类型。
nebula> CREATE TAG IF NOT EXISTS only_point(geo geography(point));

//创建 Tag，只允许存储线段形状地理空间数据类型。
nebula> CREATE TAG IF NOT EXISTS only_linestring(geo geography(linestring));

//创建 Tag，只允许存储多边形形状地理空间数据类型。
nebula> CREATE TAG IF NOT EXISTS only_polygon(geo geography(polygon));

//创建 Edge type，允许存储任意形状地理空间数据类型。
nebula> CREATE EDGE IF NOT EXISTS any_shape_edge(geo geography);

//创建存储多边形地理空间的点。
nebula> INSERT VERTEX any_shape(geo) VALUES "103":(ST_GeogFromText("POLYGON((0 1, 1 2, 2 3, 0 1))"));

//创建存储多边形地理空间的边。
nebula> INSERT EDGE any_shape_edge(geo) VALUES "201"->"302":(ST_GeogFromText("POLYGON((0 1, 1 2, 2 3, 0 1))"));

//查询点 103 的属性 geo。
nebula> FETCH PROP ON any_shape "103" YIELD ST_ASText(any_shape.geo);
+---------------------------------+
| ST_ASText(any_shape.geo)        |
+---------------------------------+
| "POLYGON((0 1, 1 2, 2 3, 0 1))" |
+---------------------------------+

//查询边 201->302 的属性 geo。
nebula> FETCH PROP ON any_shape_edge "201"->"302" YIELD ST_ASText(any_shape_edge.geo);
+---------------------------------+
| ST_ASText(any_shape_edge.geo)   |
+---------------------------------+
| "POLYGON((0 1, 1 2, 2 3, 0 1))" |
+---------------------------------+

//为 geo 属性创建索引并使用 LOOKUP 查询。
nebula> CREATE TAG INDEX IF NOT EXISTS any_shape_geo_index ON any_shape(geo);
nebula> REBUILD TAG INDEX any_shape_geo_index;
nebula> LOOKUP ON any_shape YIELD ST_ASText(any_shape.geo);
+---------------------------------+
| ST_ASText(any_shape.geo)        |
+---------------------------------+
| "POLYGON((0 1, 1 2, 2 3, 0 1))" |
+---------------------------------+
```

为 geo 属性创建索引时，还可以指定 geo 索引的参数。说明如下。

参数	默认值	说明
`s2_max_level`	`30`	S2 cell 用于填充的最大等级。取值： `1` ~ `30` 。设置为小于默认值时，意味着会使用较大的单元格进行填充。
`s2_max_cells`	`8`	S2 cell 用于填充的最大数量，可以限制填充时的工作量。取值： `1` ~ `30` 。对于复杂形状的区域（例如细矩形），可以使用更大的值。

Note

指定如上两个参数对 Point 类型属性没有影响，Point 类型属性的 `s2_max_level` 强制为 `30` 。

```
nebula> CREATE TAG INDEX IF NOT EXISTS any_shape_geo_index ON any_shape(geo) with (s2_max_level=30, s2_max_cells=8);
```

更多索引说明请参见索引介绍。

最后更新: 2025年8月4日

4.3 运算符

4.3.1 比较符

NebulaGraph 支持的比较符如下。

符号	说明
`==`	相等
`!=`, `<>`	不等于
`>`	大于
`>=`	大于等于
`<`	小于
`<=`	小于等于
`IS NULL`	为 NULL
`IS NOT NULL`	不为 NULL
`IS EMPTY`	不存在
`IS NOT EMPTY`	存在

比较操作的结果是 `true` 或者 `false`。

> Note
>
> - 比较不同类型的值通常没有定义，结果可能是 `NULL` 或其它。
> - `EMPTY` 当前仅用于判断，不支持函数或者运算操作，包括且不限于 `GROUP BY`、`count()`、`sum()`、`max()`、`hash()`、`collect()`、`+`、`*`。

OpenCypher 兼容性

openCypher 中没有 `EMPTY`，因此不支持在 MATCH 语句中使用 `EMPTY`。

示例

`==`

字符串比较时，会区分大小写。不同类型的值不相等。

> Note
>
> nGQL 中的相等符号是 `==`，openCypher 中的相等符号是 `=`。

```
nebula> RETURN 'A' == 'a', toUpper('A') == toUpper('a'), toLower('A') == toLower('a');
+------------+------------------------------+------------------------------+
| ("A"=="a") | (toUpper("A")==toUpper("a")) | (toLower("A")==toLower("a")) |
+------------+------------------------------+------------------------------+
| false      | true                         | true                         |
+------------+------------------------------+------------------------------+

nebula> RETURN '2' == 2, toInteger('2') == 2;
+----------+---------------------+
| ("2"==2) | (toInteger("2")==2) |
+----------+---------------------+
```

```
| false    | true                |
+----------+---------------------+
```

`>`

```
nebula> RETURN 3 > 2;
+-------+
| (3>2) |
+-------+
| true  |
+-------+

nebula> WITH 4 AS one, 3 AS two \
        RETURN one > two AS result;
+--------+
| result |
+--------+
| true   |
+--------+
```

`>=`

```
nebula> RETURN 2 >= "2", 2 >= 2;
+----------+--------+
| (2>="2") | (2>=2) |
+----------+--------+
| __NULL__ | true   |
+----------+--------+
```

`<`

```
nebula> YIELD 2.0 < 1.9;
+---------+
| (2<1.9) |
+---------+
| false   |
+---------+
```

`<=`

```
nebula> YIELD 0.11 <= 0.11;
+--------------+
| (0.11<=0.11) |
+--------------+
| true         |
+--------------+
```

`!=`

```
nebula> YIELD 1 != '1';
+----------+
| (1!="1") |
+----------+
| true     |
+----------+
```

`IS [NOT] NULL`

```
# 返回关于 null 的一些判断结果。
nebula> RETURN null IS NULL AS value1, null == null AS value2, null != null AS value3;
+--------+----------+----------+
| value1 | value2   | value3   |
+--------+----------+----------+
| true   | __NULL__ | __NULL__ |
+--------+----------+----------+

# 返回关于 NULL 的一些属性信息。
nebula> RETURN length(NULL), size(NULL), count(NULL), NULL IS NULL, NULL IS NOT NULL, sin(NULL), NULL + NULL, [1, NULL] IS NULL;
+--------------+------------+-------------+--------------+------------------+-----------+-------------+------------------+
| length(NULL) | size(NULL) | count(NULL) | NULL IS NULL | NULL IS NOT NULL | sin(NULL) | (NULL+NULL) | [1,NULL] IS NULL |
+--------------+------------+-------------+--------------+------------------+-----------+-------------+------------------+
| __NULL__     | __NULL__   | 0           | true         | false            | __NULL__  | __NULL__    | false            |
+--------------+------------+-------------+--------------+------------------+-----------+-------------+------------------+

# 创建 map 数据，判断 map 的 name 属性是否不为 NULL。
nebula> WITH {name: null} AS `map` \
        RETURN `map`.name IS NOT NULL;
+----------------------+
| map.name IS NOT NULL |
+----------------------+
| false                |
+----------------------+

# 创建 map1、map2、map3 的数据，并判断返回其 name 属性是否为 NULL。
nebula> WITH {name: 'Mats', name2: 'Pontus'} AS map1, \
```

```
        {name: null} AS map2, {notName: 0, notName2: null } AS map3 \
        RETURN map1.name IS NULL, map2.name IS NOT NULL, map3.name IS NULL;
+-------------------+-----------------------+-------------------+
| map1.name IS NULL | map2.name IS NOT NULL | map3.name IS NULL |
+-------------------+-----------------------+-------------------+
| false             | false                 | true              |
+-------------------+-----------------------+-------------------+

# 查询所有 Tag 为 player 的点数据，判断并返回点的 age 属性是否为 NULL、name 属性是否不为 NULL、empty 属性是否为 NULL。
nebula> MATCH (n:player) \
        RETURN n.player.age IS NULL, n.player.name IS NOT NULL, n.player.empty IS NULL;
+----------------------+---------------------------+------------------------+
| n.player.age IS NULL | n.player.name IS NOT NULL | n.player.empty IS NULL |
+----------------------+---------------------------+------------------------+
| false                | true                      | true                   |
| false                | true                      | true                   |
...
```

`IS [NOT] EMPTY`

```
# 判断 null 是否不存在。
nebula> RETURN null IS EMPTY;
+---------------+
| NULL IS EMPTY |
+---------------+
| false         |
+---------------+

# 判断 a 字符串是否存在。
nebula> RETURN "a" IS NOT EMPTY;
+------------------+
| "a" IS NOT EMPTY |
+------------------+
| true             |
+------------------+

# 遍历所有 player100 指向的目的点，并返回目的点数据 name 属性存在的点 id。
nebula> GO FROM "player100" OVER * WHERE properties($$).name IS NOT EMPTY YIELD dst(edge);
+-------------+
| dst(EDGE)   |
+-------------+
| "team204"   |
| "player101" |
| "player125" |
+-------------+
```

最后更新: 2025年8月4日

4.3.2 布尔符

NebulaGraph 支持的布尔符如下。

符号	说明
AND	逻辑与
OR	逻辑或
NOT	逻辑非
XOR	逻辑异或

对于以上运算的优先级，请参见运算优先级。

对于带有 `NULL` 的逻辑运算，请参见 NULL。

Note

非 0 数字不能转换为布尔值。

最后更新: 2025年8月4日

4.3.3 管道符

nGQL 支持使用管道符（|）将多个查询组合起来。

openCypher 兼容性

管道符仅适用于原生 nGQL。

语法

nGQL 和 SQL 之间的一个主要区别是子查询的组成方式。

- 在 SQL 中，子查询是嵌套在查询语句中的。
- 在 nGQL 中，子查询是通过类似 shell 中的管道符（`|`）实现的。

示例

```
nebula> GO FROM "player100" OVER follow \
        YIELD dst(edge) AS dstid, properties($$).name AS Name | \
        GO FROM $-.dstid OVER follow YIELD dst(edge);

+-------------+
| dst(EDGE)   |
+-------------+
| "player100" |
| "player102" |
| "player125" |
| "player100" |
+-------------+
```

必须在 `YIELD` 子句中为需要的返回结果设置别名，才能在管道符右侧使用引用符 `$-`，例如示例中的 `$-.dstid`。

性能提示

NebulaGraph 中的管道对性能有影响，以 `A | B` 为例，体现在以下几个方面：

1. 管道是同步操作。也即需要管道之前的子句 `A` 执行完毕后，数据才能整体进入管道子句。
2. 如果 `A` 发大量数据给 `|`，整个查询请求的总体时延可能会非常大。此时可以尝试拆分这个语句：

a. 应用程序发送 `A`，

b. 将收到的返回结果在应用程序拆分，

c. 并发发送给多个 graphd，

d. 每个 graphd 执行部分 B。

这样通常比单个 graphd 执行完整地 `A | B` 要快很多。

最后更新: 2025年8月4日

4.3.4 集合运算符

合并多个请求时，可以使用集合运算符，包括 `UNION`、`UNION ALL`、`INTERSECT` 和 `MINUS`。

所有集合运算符的优先级相同，如果一个 nGQL 语句中有多个集合运算符，NebulaGraph 会从左到右进行计算，除非用括号指定顺序。

> Caution
>
> 集合运算符前后的查询语句中定义的变量名及顺序必需保持一致，例如 `RETURN a,b,c UNION RETURN a,b,c` 中的 `a,b,c` 的名称及顺序需要保持一致。

UNION、UNION DISTINCT、UNION ALL

```
<left> UNION [DISTINCT | ALL] <right> [ UNION [DISTINCT | ALL] <right> ...]
```

- 运算符 `UNION DISTINCT` （或使用缩写 `UNION`）返回两个集合 A 和 B 的并集，不包含重复的元素。
- 运算符 `UNION ALL` 返回两个集合 A 和 B 的并集，包含重复的元素。
- `left` 和 `right` 必须有相同数量的列和数据类型。如果需要转换数据类型，请参见类型转换。

示例

```
# 返回两个查询结果的并集，不包含重复的元素。
nebula> GO FROM "player102" OVER follow YIELD dst(edge) \
        UNION \
        GO FROM "player100" OVER follow YIELD dst(edge);
+-------------+
| dst(EDGE)   |
+-------------+
| "player100" |
| "player101" |
| "player125" |
+-------------+

# 查询 Tag 为 player 的点，根据名称排序后获取前 3 条数据，并与数组合并返回，不包含重复的元素。
nebula> MATCH (v:player) \
        WITH v.player.name AS n \
        RETURN n ORDER BY n LIMIT 3 \
        UNION \
        UNWIND ["Tony Parker", "Ben Simmons"] AS n \
        RETURN n;
+---------------------+
| n                   |
+---------------------+
| "Amar'e Stoudemire" |
| "Aron Baynes"       |
| "Ben Simmons"       |
| "Tony Parker"       |
+---------------------+

# 返回两个查询结果的并集，包含重复的元素。
nebula> GO FROM "player102" OVER follow YIELD dst(edge) \
        UNION ALL \
        GO FROM "player100" OVER follow YIELD dst(edge);
+-------------+
| dst(EDGE)   |
+-------------+
| "player100" |
| "player101" |
| "player101" |
| "player125" |
+-------------+

# 查询 Tag 为 player 的点，根据名称排序后获取前 3 条数据，并与数组合并返回，包含重复的元素。
nebula> MATCH (v:player) \
        WITH v.player.name AS n \
        RETURN n ORDER BY n LIMIT 3 \
        UNION ALL \
        UNWIND ["Tony Parker", "Ben Simmons"] AS n \
        RETURN n;
+---------------------+
| n                   |
+---------------------+
| "Amar'e Stoudemire" |
| "Aron Baynes"       |
| "Ben Simmons"       |
| "Tony Parker"       |
| "Ben Simmons"       |
+---------------------+
```

```
# UNION 也可以和 YIELD 语句一起使用，去重时会检查每一行的所有列，每列都相同时才会去重。
nebula> GO FROM "player102" OVER follow \
        YIELD dst(edge) AS id, properties(edge).degree AS Degree, properties($$).age AS Age \
        UNION /* DISTINCT */ \
        GO FROM "player100" OVER follow \
        YIELD dst(edge) AS id, properties(edge).degree AS Degree, properties($$).age AS Age;
+-------------+--------+-----+
| id          | Degree | Age |
+-------------+--------+-----+
| "player100" | 75     | 42  |
| "player101" | 75     | 36  |
| "player101" | 95     | 36  |
| "player125" | 95     | 41  |
+-------------+--------+-----+
```

INTERSECT

```
<left> INTERSECT <right>
```

- 运算符 `INTERSECT` 返回两个集合 A 和 B 的交集。
- `left` 和 `right` 必须有相同数量的列和数据类型。如果需要转换数据类型，请参见类型转换。

示例

```
# 返回两个查询结果的交集。
nebula> GO FROM "player102" OVER follow \
        YIELD dst(edge) AS id, properties(edge).degree AS Degree, properties($$).age AS Age \
        INTERSECT \
        GO FROM "player100" OVER follow \
        YIELD dst(edge) AS id, properties(edge).degree AS Degree, properties($$).age AS Age;
+----+--------+-----+
| id | Degree | Age |
+----+--------+-----+
+----+--------+-----+

# 返回 player102 的邻居和边数据与 player100 的邻居和边数据之间的交集。
nebula> MATCH (v:player)-[e:follow]->(v2) \
        WHERE id(v) == "player102" \
        RETURN id(v2) As id, e.degree As Degree, v2.player.age AS Age \
        INTERSECT \
        MATCH (v:player)-[e:follow]->(v2) \
        WHERE id(v) == "player100" \
        RETURN id(v2) As id, e.degree As Degree, v2.player.age AS Age;
+----+--------+-----+
| id | Degree | Age |
+----+--------+-----+
+----+--------+-----+

# 返回 [1,2] 与 [1,2,3,4] 的交集。
nebula> UNWIND [1,2] AS a RETURN a \
        INTERSECT \
        UNWIND [1,2,3,4] AS a \
        RETURN a;
+---+
| a |
+---+
| 1 |
| 2 |
+---+
```

MINUS

```
<left> MINUS <right>
```

运算符 `MINUS` 返回两个集合 A 和 B 的差异，即 `A-B`。请注意 `left` 和 `right` 的顺序，`A-B` 表示在集合 A 中，但是不在集合 B 中的元素。

示例

```
# 返回在第一个查询结果中，但是不在第二个查询结果中的元素。
nebula> GO FROM "player100" OVER follow YIELD dst(edge) \
        MINUS \
        GO FROM "player102" OVER follow YIELD dst(edge);
+-------------+
| dst(EDGE)   |
+-------------+
| "player125" |
+-------------+

# 返回在 player102 的邻居中，但不在 player100 的邻居中的元素。
nebula> GO FROM "player102" OVER follow YIELD dst(edge) AS id \
        MINUS \
```

```
        GO FROM "player100" OVER follow YIELD dst(edge) AS id;
+-------------+
| id          |
+-------------+
| "player100" |
+-------------+

# 返回在 player102 的邻居中，但不在 player100 的邻居中的元素。
nebula> MATCH (v:player)-[e:follow]->(v2) \
        WHERE id(v) =="player102" \
        RETURN id(v2) AS id\
        MINUS \
        MATCH (v:player)-[e:follow]->(v2) \
        WHERE id(v) =="player100" \
        RETURN id(v2) AS id;
+-------------+
| id          |
+-------------+
| "player100" |
+-------------+

# 返回 [1,2,3] 中不与 4 相同的元素。
nebula> UNWIND [1,2,3] AS a RETURN a \
        MINUS \
        WITH 4 AS a \
        RETURN a;
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
+---+
```

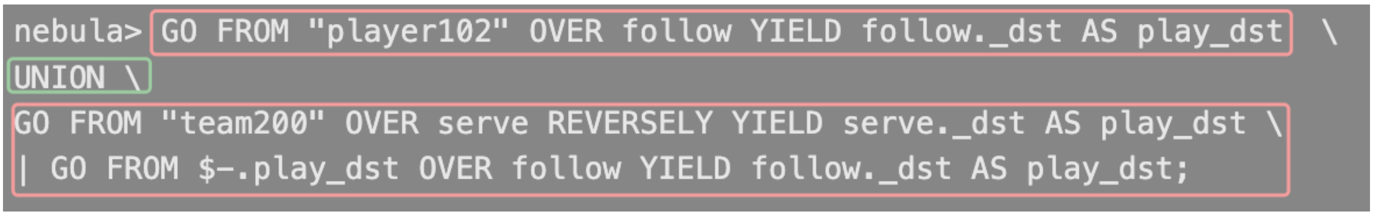
集合运算符和管道符的优先级

当查询包含集合运算符和管道符（|）时，管道符的优先级高。例如 `GO FROM 1 UNION GO FROM 2 | GO FROM 3` 相当于 `GO FROM 1 UNION (GO FROM 2 | GO FROM 3)`。

示例

```
nebula> GO FROM "player102" OVER follow \
        YIELD dst(edge) AS play_dst  \
        UNION \
        GO FROM "team200" OVER serve REVERSELY \
        YIELD src(edge) AS play_src \
        | GO FROM $-.play_src OVER follow YIELD dst(edge) AS play_dst;

+-------------+
| play_dst    |
+-------------+
| "player100" |
| "player101" |
| "player117" |
| "player105" |
+-------------+
```



该查询会先执行红框内的语句，然后执行绿框的 `UNION` 操作。

圆括号可以修改执行的优先级，例如：

```
nebula> (GO FROM "player102" OVER follow \
        YIELD dst(edge) AS play_dst  \
        UNION \
        GO FROM "team200" OVER serve REVERSELY \
        YIELD src(edge) AS play_dst) \
        | GO FROM $-.play_dst OVER follow YIELD dst(edge) AS play_dst;
```

该查询中，圆括号包裹的部分先执行，即先执行 `UNION` 操作，再将结果结合管道符进行下一步操作。

最后更新: 2025年8月4日

4.3.5 字符串运算符

NebulaGraph 支持使用字符串运算符进行连接、搜索、匹配运算。支持的运算符如下。

名称	说明
+	连接字符串。
CONTAINS	在字符串中执行搜索。
(NOT) IN	字符串是否匹配某个值。
(NOT) STARTS WITH	在字符串的开头执行匹配。
(NOT) ENDS WITH	在字符串的结尾执行匹配。
正则表达式	通过正则表达式匹配字符串。

> Note
>
> 所有搜索或匹配都区分大小写。

示例

`+`

```
# 返回连接后的字符串。
nebula> RETURN 'a' + 'b';
+-----------+
| ("a"+"b") |
+-----------+
| "ab"      |
+-----------+
nebula> UNWIND 'a' AS a UNWIND 'b' AS b RETURN a + b;
+-------+
| (a+b) |
+-------+
| "ab"  |
+-------+
```

`CONTAINS`

`CONTAINS` 要求待运算的左右两边都是字符串类型。

```
# 查询返回 id 值为 player101，且目标点属性 name 中含有"ets"的数据。
nebula> MATCH (s:player)-[e:serve]->(t:team) WHERE id(s) == "player101" \
        AND t.team.name CONTAINS "ets" RETURN s.player.name, e.start_year, e.end_year, t.team.name;
+---------------+--------------+------------+-------------+
| s.player.name | e.start_year | e.end_year | t.team.name |
+---------------+--------------+------------+-------------+
| "Tony Parker" | 2018         | 2019       | "Hornets"   |
+---------------+--------------+------------+-------------+

# 查询返回 player101 的边属性 start_year 转为字符串后含有"19"，且点属性 name 中含有"ny"的数据。
nebula> GO FROM "player101" OVER serve WHERE (STRING)properties(edge).start_year CONTAINS "19" AND \
        properties($^).name CONTAINS "ny" \
        YIELD properties($^).name, properties(edge).start_year, properties(edge).end_year, properties($$).name;
+--------------------+----------------------------+--------------------------+--------------------+
| properties($^).name | properties(EDGE).start_year | properties(EDGE).end_year | properties($$).name |
+--------------------+----------------------------+--------------------------+--------------------+
| "Tony Parker"       | 1999                        | 2018                      | "Spurs"             |
+--------------------+----------------------------+--------------------------+--------------------+

# 查询返回 palyer101 指向的目标点属性 name 中不含有"ets"的数据。
nebula> GO FROM "player101" OVER serve WHERE !(properties($$).name CONTAINS "ets") \
        YIELD properties($^).name, properties(edge).start_year, properties(edge).end_year, properties($$).name;
+--------------------+----------------------------+--------------------------+--------------------+
| properties($^).name | properties(EDGE).start_year | properties(EDGE).end_year | properties($$).name |
+--------------------+----------------------------+--------------------------+--------------------+
| "Tony Parker"       | 1999                        | 2018                      | "Spurs"             |
+--------------------+----------------------------+--------------------------+--------------------+
```

`(NOT) IN`

```
# 返回列表中是否含有某个值。
nebula> RETURN  1 IN [1,2,3], "Yao" NOT IN ["Yi", "Tim", "Kobe"], NULL IN ["Yi", "Tim", "Kobe"];
+----------------+------------------------------------+-------------------------------+
| (1 IN [1,2,3]) | ("Yao" NOT IN ["Yi","Tim","Kobe"]) | (NULL IN ["Yi","Tim","Kobe"]) |
+----------------+------------------------------------+-------------------------------+
| true           | true                               | __NULL__                      |
+----------------+------------------------------------+-------------------------------+
```

`(NOT) STARTS WITH`

```
# 返回是否以某个字符串开头。
nebula> RETURN 'apple' STARTS WITH 'app', 'apple' STARTS WITH 'a', 'apple' STARTS WITH toUpper('a');
+-----------------------------+---------------------------+------------------------------------+
| ("apple" STARTS WITH "app") | ("apple" STARTS WITH "a") | ("apple" STARTS WITH toUpper("a")) |
+-----------------------------+---------------------------+------------------------------------+
| true                        | true                      | false                              |
+-----------------------------+---------------------------+------------------------------------+

nebula> RETURN 'apple' STARTS WITH 'b','apple' NOT STARTS WITH 'app';
+---------------------------+---------------------------------+
| ("apple" STARTS WITH "b") | ("apple" NOT STARTS WITH "app") |
+---------------------------+---------------------------------+
| false                     | false                           |
+---------------------------+---------------------------------+
```

`(NOT) ENDS WITH`

```
# 返回是否以某个字符串结尾。
nebula> RETURN 'apple' ENDS WITH 'app', 'apple' ENDS WITH 'e', 'apple' ENDS WITH 'E', 'apple' ENDS WITH 'b';
+---------------------------+-------------------------+-------------------------+-------------------------+
| ("apple" ENDS WITH "app") | ("apple" ENDS WITH "e") | ("apple" ENDS WITH "E") | ("apple" ENDS WITH "b") |
+---------------------------+-------------------------+-------------------------+-------------------------+
| false                     | true                    | false                   | false                   |
+---------------------------+-------------------------+-------------------------+-------------------------+
```

正则表达式

Note

当前仅 opencypher 兼容语句（`MATCH`、`WITH` 等）支持正则表达式，原生 nGQL 语句（`FETCH`、`GO`、`LOOKUP` 等）不支持正则表达式。

NebulaGraph 支持使用正则表达式进行过滤，正则表达式的语法是继承自 `std::regex`，用户可以使用语法 `=~ '<regexp>'` 进行正则表达式匹配。例如：

```
# 返回是否匹配正则表达式。
nebula> RETURN "384748.39" =~ "\\d+(\\.\\d{2})?";
+--------------------------------+
| ("384748.39"=~"\d+(\.\d{2})?") |
+--------------------------------+
| true                           |
+--------------------------------+

# 查询返回符合正则表达式的点数据。
nebula> MATCH (v:player) WHERE v.player.name =~ 'Tony.*' RETURN v.player.name;
+---------------+
| v.player.name |
+---------------+
| "Tony Parker" |
+---------------+
```

最后更新: 2025年8月4日

4.3.6 列表运算符

NebulaGraph 支持使用列表（List）运算符进行运算。支持的运算符如下。

名称	说明
+	连接列表。
IN	元素是否存在于列表中。
[]	使用下标操作符访问列表中的元素。

示例

```
# 返回连接后的列表。
nebula> YIELD [1,2,3,4,5]+[6,7] AS myList;
+-----------------------+
| myList                |
+-----------------------+
| [1, 2, 3, 4, 5, 6, 7] |
+-----------------------+

# 返回列表中的元素数量。
nebula> RETURN size([NULL, 1, 2]);
+------------------+
| size([NULL,1,2]) |
+------------------+
| 3                |
+------------------+

# 返回 NULL 是否存在列表中。
nebula> RETURN NULL IN [NULL, 1];
+--------------------+
| (NULL IN [NULL,1]) |
+--------------------+
| __NULL__           |
+--------------------+

# 返回在列表[2, 3, 4, 5]中存在，且也在列表[2, 3, 8]中存在的元素，并拆分显示。
nebula> WITH [2, 3, 4, 5] AS numberlist \
    UNWIND numberlist AS number \
    WITH number \
    WHERE number IN [2, 3, 8] \
    RETURN number;
+--------+
| number |
+--------+
| 2      |
| 3      |
+--------+

# 返回列表中下标为 1 的元素。
nebula> WITH ['Anne', 'John', 'Bill', 'Diane', 'Eve'] AS names RETURN names[1] AS result;
+--------+
| result |
+--------+
| "John" |
+--------+
```

最后更新: 2025年8月4日

4.3.7 算术运算符

NebulaGraph 支持的算术运算符如下。

符号	说明
`+`	加法
`-`	减法
`*`	乘法
`/`	除法
`%`	取模
`-`	负数符号

示例

```
# 返回 1+2 的结果。
nebula> RETURN 1+2 AS result;
+--------+
| result |
+--------+
| 3      |
+--------+

# 返回 -10+5 的结果。
nebula> RETURN -10+5 AS result;
+--------+
| result |
+--------+
| -5     |
+--------+

# 返回 (3*8)%5 的结果。
nebula> RETURN (3*8)%5 AS result;
+--------+
| result |
+--------+
| 4      |
+--------+
```

最后更新: 2025年8月4日

4.3.8 运算符优先级

nGQL 运算符的优先级从高到低排列如下（同一行的运算符优先级相同）：

- `-` （负数）
- `!` 、 `NOT`
- `*` 、 `/` 、 `%`
- `-` 、 `+`
- `==` 、 `>=` 、 `>` 、 `<=` 、 `<` 、 `<>` 、 `!=`
- `AND`
- `OR` 、 `XOR`
- `=` （赋值）

如果表达式中有相同优先级的运算符，运算是从左到右进行，只有赋值操作是例外（从右到左运算）。

运算符的优先级决定运算的顺序，要显式修改运算顺序，可以使用圆括号。

示例

```
nebula> RETURN 2+3*5;
+-----------+
| (2+(3*5)) |
+-----------+
| 17        |
+-----------+

nebula> RETURN (2+3)*5;
+-----------+
| ((2+3)*5) |
+-----------+
| 25        |
+-----------+
```

openCypher 兼容性

在 openCypher 中，比较操作可以任意连接，例如 `x < y <= z` 等价于 `x < y AND y <= z` 。

在 nGQL 中， `x < y <= z` 等价于 `(x < y) <= z` ， `(x < y)` 的结果是一个布尔值，再将布尔值和 `z` 比较，最终结果是 `NULL` 。

最后更新: 2025年8月4日

4.4 函数和表达式

4.4.1 内置数学函数

本文介绍 NebulaGraph 支持的数学函数。

abs()

abs() 返回指定数字的绝对值。

语法：`abs(<expression>)`

- `expression`：结果的数据类型为 double 的表达式。
- 返回类型：double。

示例：

```
nebula> RETURN abs(-10);
+------------+
| abs(-(10)) |
+------------+
| 10         |
+------------+

nebula> RETURN abs(5-6);
+------------+
| abs((5-6)) |
+------------+
| 1          |
+------------+
```

floor()

floor() 返回小于或等于指定数字的最大整数。

语法：`floor(<expression>)`

- `expression`：结果的数据类型为 double 的表达式。
- 返回类型：double。

示例：

```
nebula> RETURN floor(9.9);
+------------+
| floor(9.9) |
+------------+
| 9.0        |
+------------+
```

ceil()

ceil() 返回大于或等于指定数字的最小整数。

语法：`ceil(<expression>)`

- `expression`：结果的数据类型为 double 的表达式。
- 返回类型：double。

示例：

```
nebula> RETURN ceil(9.1);
+-----------+
| ceil(9.1) |
+-----------+
```

```
| 10.0      |
+-----------+
```

round()

round() 返回指定数字四舍五入后的值。极端情况下请注意浮点数的精度问题。

语法： `round(<expression>, <precision>, <mode>)`

- `expression`： 结果的数据类型为 double 的表达式。
- `precision`： 代表精度的整数。小于 0 时，在小数点左侧做四舍五入。
- `mode`： 代表舍入模式的字符串。可选参数。可选值如下：
- `DOWN`： 向零舍入。
- `CEILING`： 向正无穷舍入。
- `FLOOR`： 向负无穷舍入。
- `HALF_UP`： 基于精度向最接近的值舍入，如果最接近的值有两个，则取远离零的值。
- `HALF_DOWN`： 基于精度向最接近的值舍入，如果最接近的值有两个，则取靠近零的值。
- `HALF_EVEN`： 基于精度向最接近的值舍入，如果最接近的值有两个，则取最接近的偶数值。
- 返回类型：double。

示例：

```
nebula> RETURN round(314.15926, 2);
+--------------------+
| round(314.15926,2) |
+--------------------+
| 314.16             |
+--------------------+

nebula> RETURN round(314.15926, -1);
+-----------------------+
| round(314.15926,-(1)) |
+-----------------------+
| 310.0                 |
+-----------------------+

nebula> RETURN round(314.15926, 2, "DOWN");
+---------------------------+
| round(314.15926,2,"DOWN") |
+---------------------------+
| 314.15                    |
+---------------------------+

nebula> RETURN round(314.15926, 2, "CEILING");
+------------------------------+
| round(314.15926,2,"CEILING") |
+------------------------------+
| 314.16                       |
+------------------------------+

nebula> RETURN round(314.15926, 2, "FLOOR");
+----------------------------+
| round(314.15926,2,"FLOOR") |
+----------------------------+
| 314.15                     |
+----------------------------+

nebula> RETURN round(314.15, 1, "HALF_UP");
+---------------------------+
| round(314.15,1,"HALF_UP") |
+---------------------------+
| 314.2                     |
+---------------------------+

nebula> RETURN round(314.15, 1, "HALF_DOWN");
+-----------------------------+
| round(314.15,1,"HALF_DOWN") |
+-----------------------------+
| 314.1                       |
+-----------------------------+

nebula> RETURN round(314.15, 1, "HALF_EVEN");
+-----------------------------+
| round(314.15,1,"HALF_EVEN") |
+-----------------------------+
```

```
| 314.2                       |
+-----------------------------+
```

sqrt()

sqrt() 返回指定数字的平方根。

语法： `sqrt(<expression>)`

- `expression`：结果的数据类型为 double 的表达式。
- 返回类型：double。

示例：

```
nebula> RETURN sqrt(9);
+---------+
| sqrt(9) |
+---------+
| 3.0     |
+---------+
```

cbrt()

cbrt() 返回指定数字的立方根。

语法： `cbrt(<expression>)`

- `expression`：结果的数据类型为 double 的表达式。
- 返回类型：double。

示例：

```
nebula> RETURN cbrt(8);
+---------+
| cbrt(8) |
+---------+
| 2.0     |
+---------+
```

hypot()

hypot() 返回直角三角形的斜边长。

语法： `hypot(<expression_x>,<expression_y>)`

- `expression_x`、`expression_y`：结果的数据类型为 double 的表达式。表示直角三角形的边长 `x` 和 `y`。
- 返回类型：double。

示例：

```
nebula> RETURN hypot(3,2*2);
+----------------+
| hypot(3,(2*2)) |
+----------------+
| 5.0            |
+----------------+
```

pow()

pow() 返回指定数字的幂（x^y）。

语法：`pow(<expression_x>,<expression_y>,)`

- `expression_x`：结果的数据类型为 double 的表达式。表示底数 `x`。
- `expression_y`：结果的数据类型为 double 的表达式。表示指数 `y`。
- 返回类型：double。

示例：

```
nebula> RETURN pow(3,3);
+----------+
| pow(3,3) |
+----------+
| 27       |
+----------+
```

exp()

exp() 返回自然常数 e 的幂（e^X）。

语法：`exp(<expression>)`

- `expression`：结果的数据类型为 double 的表达式。表示指数 `x`。
- 返回类型：double。

示例：

```
nebula> RETURN exp(2);
+------------------+
| exp(2)           |
+------------------+
| 7.38905609893065 |
+------------------+
```

exp2()

exp2() 返回2的幂（2^X）。

语法：`exp2(<expression>)`

- `expression`：结果的数据类型为 double 的表达式。表示指数 `x`。
- 返回类型：double。

示例：

```
nebula> RETURN exp2(3);
+---------+
| exp2(3) |
+---------+
| 8.0     |
+---------+
```

log()

log() 返回以自然数 e 为底的对数（\(log_{e}{N}\)）。

语法：`log(<expression>)`

- `expression`：结果的数据类型为 double 的表达式。表示真数 `N`。
- 返回类型：double。

示例：

```
nebula> RETURN log(8);
+--------------------+
| log(8)             |
```

```
+--------------------+
| 2.0794415416798357 |
+--------------------+
```

log2()

log2() 返回以 2 为底的对数（\(log_{2}{N}\)）。

语法： `log2(<expression>)`

- `expression`：结果的数据类型为 double 的表达式。表示真数 `N`。
- 返回类型：double。

示例：

```
nebula> RETURN log2(8);
+---------+
| log2(8) |
+---------+
| 3.0     |
+---------+
```

log10()

log10() 返回以 10 为底的对数（\(log_{10}{N}\)）。

语法： `log10(<expression>)`

- `expression`：结果的数据类型为 double 的表达式。表示真数 `N`。
- 返回类型：double。

示例：

```
nebula> RETURN log10(100);
+------------+
| log10(100) |
+------------+
| 2.0        |
+------------+
```

sin()

sin() 返回指定数字的正弦值。可以使用函数 `radians()` 将角度转化为弧度。

语法： `sin(<expression>)`

- `expression`：结果的数据类型为 double 的表达式。
- 返回类型：double。

示例：

```
nebula> RETURN sin(3);
+--------------------+
| sin(3)             |
+--------------------+
| 0.1411200080598672 |
+--------------------+
```

asin()

asin() 返回指定数字的反正弦值。可以使用函数 `radians()` 将角度转化为弧度。

语法：`asin(<expression>)`

- `expression`：结果的数据类型为 double 的表达式。
- 返回类型：double。

示例：

```
nebula> RETURN asin(0.5);
+--------------------+
| asin(0.5)          |
+--------------------+
| 0.5235987755982989 |
+--------------------+
```

cos()

cos() 返回指定数字的余弦值。可以使用函数 `radians()` 将角度转化为弧度。

语法：`cos(<expression>)`

- `expression`：结果的数据类型为 double 的表达式。
- 返回类型：double。

示例：

```
nebula> RETURN cos(0.5);
+--------------------+
| cos(0.5)           |
+--------------------+
| 0.8775825618903728 |
+--------------------+
```

acos()

acos() 返回指定数字的反余弦值。可以使用函数 `radians()` 将角度转化为弧度。

语法：`acos(<expression>)`

- `expression`：结果的数据类型为 double 的表达式。
- 返回类型：double。

示例：

```
nebula> RETURN acos(0.5);
+--------------------+
| acos(0.5)          |
+--------------------+
| 1.0471975511965979 |
+--------------------+
```

tan()

tan() 返回指定数字的正切值。可以使用函数 `radians()` 将角度转化为弧度。

语法：`tan(<expression>)`

- `expression`：结果的数据类型为 double 的表达式。
- 返回类型：double。

示例：

```
nebula> RETURN tan(0.5);
+--------------------+
| tan(0.5)           |
+--------------------+
```

```
| 0.5463024898437905 |
+--------------------+
```

atan()

atan() 返回指定数字的反正切值。可以使用函数 `radians()` 将角度转化为弧度。

语法：`atan(<expression>)`

- `expression`：结果的数据类型为 double 的表达式。
- 返回类型：double。

示例：

```
nebula> RETURN atan(0.5);
+--------------------+
| atan(0.5)          |
+--------------------+
| 0.4636476090008061 |
+--------------------+
```

rand()

rand() 返回 [0,1) 内的随机浮点数。

语法：`rand()`

- 返回类型：double。

示例：

```
nebula> RETURN rand();
+--------------------+
| rand()             |
+--------------------+
| 0.6545837172298736 |
+--------------------+
```

rand32()

rand32() 返回指定范围（`[min, max)`）内的随机 32 位整数。

语法：`rand32(<expression_min>,<expression_max>)`

- `expression_min`：结果的数据类型为 int 的表达式。表示最小值 `min`。
- `expression_max`：结果的数据类型为 int 的表达式。表示最大值 `max`。
- 返回类型：int。
- 用户可以只传入一个参数，该参数会判定为 `max`，此时 `min` 默认为 `0`。如果不传入参数，此时会从带符号的 32 位 int 范围内随机返回。

示例：

```
nebula> RETURN rand32(1,100);
+---------------+
| rand32(1,100) |
+---------------+
| 63            |
+---------------+
```

rand64()

rand64() 返回指定范围（`[min, max)`）内的随机 64 位整数。

语法：`rand64(<expression_min>,<expression_max>)`

- `expression_min`：结果的数据类型为 int 的表达式。表示最小值 `min`。
- `expression_max`：结果的数据类型为 int 的表达式。表示最大值 `max`。
- 返回类型：int。
- 用户可以只传入一个参数，该参数会判定为 `max`，此时 `min` 默认为 `0`。如果不传入参数，此时会从带符号的 64 位 int 范围内随机返回。

示例：

```
nebula> RETURN rand64(1,100);
+---------------+
| rand64(1,100) |
+---------------+
| 34            |
+---------------+
```

bit_and()

bit_and() 返回按位进行 AND 运算后的结果。

语法：`bit_and(<expression_1>,<expression_2>)`

- `expression_1`、`expression_2`：结果的数据类型为 int 的表达式。
- 返回类型：int。

示例：

```
nebula> RETURN bit_and(5,6);
+--------------+
| bit_and(5,6) |
+--------------+
| 4            |
+--------------+
```

bit_or()

bit_or() 返回按位进行 OR 运算后的结果。

语法：`bit_or(<expression_1>,<expression_2>)`

- `expression_1`、`expression_2`：结果的数据类型为 int 的表达式。
- 返回类型：int。

示例：

```
nebula> RETURN bit_or(5,6);
+-------------+
| bit_or(5,6) |
+-------------+
| 7           |
+-------------+
```

bit_xor()

bit_xor() 返回按位进行 XOR 运算后的结果。

语法：`bit_xor(<expression_1>,<expression_2>)`

- `expression_1`、`expression_2`：结果的数据类型为 int 的表达式。
- 返回类型：int。

示例：

```
nebula> RETURN bit_xor(5,6);
+-------------+
| bit_xor(5,6) |
+-------------+
| 3           |
+-------------+
```

size()

size() 返回列表或映射中元素的数量，或者返回字符串的长度。

语法： `size({<expression>|<string>})`

- `expression`： 列表或映射的表达式。
- `string`： 指定的字符串。
- 返回类型：int。

示例：

```
nebula> RETURN size([1,2,3,4]);
+-----------------+
| size([1,2,3,4]) |
+-----------------+
| 4               |
+-----------------+
```

```
nebula> RETURN size("basketballplayer") as size;
+------+
| size |
+------+
| 16   |
+------+
```

range()

range() 返回指定范围（ `[start,end]` ）中指定步长的值组成的列表。

语法： `range(<expression_start>,<expression_end>[,<expression_step>])`

- `expression_start`： 结果的数据类型为 int 的表达式。表示起始值 `start` 。
- `expression_end`： 结果的数据类型为 int 的表达式。表示结束值 `end` 。
- `expression_step`： 结果的数据类型为 int 的表达式。表示步长 `step` ，默认值为 `1` 。
- 返回类型：list。

示例：

```
nebula> RETURN range(1,3*3,2);
+------------------+
| range(1,(3*3),2) |
+------------------+
| [1, 3, 5, 7, 9]  |
+------------------+
```

sign()

sign() 返回指定数字的正负号。如果数字为 `0` ，则返回 `0` ；如果数字为负数，则返回 `-1` ；如果数字为正数，则返回 `1` 。

语法： `sign(<expression>)`

- `expression`： 结果的数据类型为 double 的表达式。
- 返回类型：int。

示例：

```
nebula> RETURN sign(10);
+----------+
| sign(10) |
+----------+
| 1        |
+----------+
```

e()

e() 返回自然对数的底 e（2.718281828459045）。

语法：`e()`

- 返回类型：double。

示例：

```
nebula> RETURN e();
+-------------------+
| e()               |
+-------------------+
| 2.718281828459045 |
+-------------------+
```

pi()

pi() 返回数学常数π（3.141592653589793）。

语法：`pi()`

- 返回类型：double。

示例：

```
nebula> RETURN pi();
+-------------------+
| pi()              |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

radians()

radians() 返回指定角度的弧度。

语法：`radians(<angle>)`

- 返回类型：double。

示例：

```
nebula> RETURN radians(180);
+-------------------+
| radians(180)      |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

最后更新: 2025年8月4日

4.4.2 聚合函数

本文介绍 NebulaGraph 支持的聚合函数。

avg()

avg() 返回参数的平均值。

语法：`avg(<expression>)`

- 返回类型：double。

示例：

```
# 返回 Tag player 的 age 属性的平均值。
nebula> MATCH (v:player) RETURN avg(v.player.age);
+--------------------+
| avg(v.player.age)  |
+--------------------+
| 33.294117647058826 |
+--------------------+
```

count()

count() 返回参数的数量。

- （原生 nGQL）用户可以同时使用 `count()` 和 `GROUP BY` 对传参进行分组和计数，再使用 `YIELD` 返回结果。
- （openCypher 方式）用户可以使用 `count()` 对指定的值进行计数，再使用 `RETURN` 返回结果。不需要使用 `GROUP BY`。

语法：`count({<expression> | *})`

- count(*) 返回总行数（包括 NULL）。
- 返回类型：int。

示例：

```
# 将列表拆分，并返回拆分后行的数量，列表元素的数量，拆分后将行去重后的数量。
nebula> WITH [NULL, 1, 1, 2, 2] As a UNWIND a AS b \
        RETURN count(b), count(*), count(DISTINCT b);
+----------+----------+-------------------+
| count(b) | count(*) | count(distinct b) |
+----------+----------+-------------------+
| 4        | 5        | 2                 |
+----------+----------+-------------------+
```

```
# 返回 player101 follow 的人，以及 follow player101 的人，即双向查询。
# 使用`count()`和`GROUP BY`进行分组和计数。
nebula> GO FROM "player101" OVER follow BIDIRECT \
        YIELD properties($$).name AS Name \
        | GROUP BY $-.Name YIELD $-.Name, count(*);
+---------------------+----------+
| $-.Name             | count(*) |
+---------------------+----------+
| "LaMarcus Aldridge" | 2        |
| "Tim Duncan"        | 2        |
| "Marco Belinelli"   | 1        |
| "Manu Ginobili"     | 1        |
| "Boris Diaw"        | 1        |
| "Dejounte Murray"   | 1        |
+---------------------+----------+

# 使用`count()`进行计数。
nebula> MATCH (v1:player)-[:follow]-(v2:player) \
        WHERE id(v1)== "player101" \
        RETURN v2.player.name AS Name, count(*) as cnt ORDER BY cnt DESC;
+---------------------+-----+
| Name                | cnt |
+---------------------+-----+
| "LaMarcus Aldridge" | 2   |
| "Tim Duncan"        | 2   |
| "Boris Diaw"        | 1   |
| "Manu Ginobili"     | 1   |
| "Dejounte Murray"   | 1   |
```

```
| "Marco Belinelli"   | 1   |
+--------------------+-----+
```

上述示例的返回结果有两列：

- `$-.Name`：查询结果包含的姓名。
- `count(*)`：姓名出现的次数。

因为测试数据集 `basketballplayer` 中没有重复的姓名，`count(*)` 列中数字 `2` 表示该行的人和 `player101` 是互相 `follow` 的关系。

```
# 方法一：统计数据库中的年龄分布情况。
nebula> LOOKUP ON player \
        YIELD player.age As playerage \
        | GROUP BY $-.playerage \
        YIELD $-.playerage as age, count(*) AS number \
        | ORDER BY $-.number DESC, $-.age DESC;
+-----+--------+
| age | number |
+-----+--------+
| 34  | 4      |
| 33  | 4      |
| 30  | 4      |
| 29  | 4      |
| 38  | 3      |
+-----+--------+
...

# 方法二：统计数据库中的年龄分布情况。
nebula> MATCH (n:player) \
        RETURN n.player.age as age, count(*) as number \
        ORDER BY number DESC, age DESC;
+-----+--------+
| age | number |
+-----+--------+
| 34  | 4      |
| 33  | 4      |
| 30  | 4      |
| 29  | 4      |
| 38  | 3      |
+-----+--------+
...
```

```
# 统计 Tim Duncan 关联的边数。
nebula> MATCH (v:player{name:"Tim Duncan"}) -[e]- (v2) \
        RETURN count(e);
+----------+
| count(e) |
+----------+
| 13       |
+----------+


# 多跳查询，统计 Tim Duncan 关联的边数，返回两列（不去重和去重）。
nebula> MATCH (n:player {name : "Tim Duncan"})-[]->(friend:player)-[]->(fof:player) \
        RETURN count(fof), count(DISTINCT fof);
+------------+---------------------+
| count(fof) | count(distinct fof) |
+------------+---------------------+
| 4          | 3                   |
+------------+---------------------+
```

max()

max() 返回参数的最大值。

语法：`max(<expression>)`

- 返回类型：与原参数相同。

示例：

```
# 返回 Tag player 的 age 属性的最大值。
nebula> MATCH (v:player) RETURN max(v.player.age);
+-------------------+
| max(v.player.age) |
+-------------------+
| 47                |
+-------------------+
```

min()

min() 返回参数的最小值。

语法：`min(<expression>)`

- 返回类型：与原参数相同。

示例：

```
# 返回 Tag player 的 age 属性的最小值。
nebula> MATCH (v:player) RETURN min(v.player.age);
+-------------------+
| min(v.player.age) |
+-------------------+
| 20                |
+-------------------+
```

collect()

collect() 返回一个符合表达式返回结果的列表。该函数可以将多条记录或值合并进一个列表，实现数据聚合。

语法：`collect(<expression>)`

- 返回类型：list。

示例：

```
# 将列表拆分为单独的行。
nebula> UNWIND [1, 2, 1] AS a \
        RETURN a;
+---+
| a |
+---+
| 1 |
| 2 |
| 1 |
+---+

# 将列表拆分为单独的行记录后再合并为一个列表返回。
nebula> UNWIND [1, 2, 1] AS a \
        RETURN collect(a);
+------------+
| collect(a) |
+------------+
| [1, 2, 1]  |
+------------+

# 统计并返回列表中相同元素的数量。
nebula> UNWIND [1, 2, 1] AS a \
        RETURN a, collect(a), size(collect(a));
+---+------------+------------------+
| a | collect(a) | size(collect(a)) |
+---+------------+------------------+
| 2 | [2]        | 1                |
| 1 | [1, 1]     | 2                |
+---+------------+------------------+

# 降序排列，限制输出行数为 3，然后将结果输出到列表中。
nebula> UNWIND ["c", "b", "a", "d" ] AS p \
        WITH p AS q \
        ORDER BY q DESC LIMIT 3 \
        RETURN collect(q);
+-----------------+
| collect(q)      |
+-----------------+
| ["d", "c", "b"] |
+-----------------+

# 将列表拆分后去重，再合并为新的列表返回。
nebula> WITH [1, 1, 2, 2] AS coll \
        UNWIND coll AS x \
        WITH DISTINCT x \
        RETURN collect(x) AS ss;
+--------+
| ss     |
+--------+
| [1, 2] |
+--------+

# 将 Tag player 的 age 属性值合并为列表返回。
nebula> MATCH (n:player) \
        RETURN collect(n.player.age);
+---------------------------------------------------------------+
```

```
| collect(n.player.age)                                          |
+----------------------------------------------------------------+
| [32, 32, 34, 29, 41, 40, 33, 25, 40, 37, ...
...

# 基于年龄聚合姓名。
nebula> MATCH (n:player) \
        RETURN n.player.age AS age, collect(n.player.name);
+-----+--------------------------------------------------------------------------+
| age | collect(n.player.name)                                                   |
+-----+--------------------------------------------------------------------------+
| 24  | ["Giannis Antetokounmpo"]                                                |
| 20  | ["Luka Doncic"]                                                          |
| 25  | ["Joel Embiid", "Kyle Anderson"]                                         |
+-----+--------------------------------------------------------------------------+
...

# 将 player100 的目的点 name 属性值聚合，结果合并为列表返回。
nebula> GO FROM "player100" OVER serve \
        YIELD properties($$).name AS name \
        | GROUP BY $-.name \
        YIELD collect($-.name) AS name;
+-----------+
| name      |
+-----------+
| ["Spurs"] |
+-----------+

# 将 Tag player 的 age 属性值聚合，结果合并为列表返回。
nebula> LOOKUP ON player \
        YIELD player.age As playerage \
        | GROUP BY $-.playerage \
        YIELD collect($-.playerage) AS playerage;
+------------------+
| playerage        |
+------------------+
| [22]             |
| [47]             |
| [43]             |
| [25, 25]         |
+------------------+
...
```

std()

std() 返回参数的总体标准差。

语法：`std(<expression>)`

- 返回类型：double。

示例：

```
# 返回 Tag player 的 age 属性值的标准差。
nebula> MATCH (v:player) RETURN std(v.player.age);
+-------------------+
| std(v.player.age) |
+-------------------+
| 6.423895701687502 |
+-------------------+
```

sum()

sum() 返回参数的和。

语法：`sum(<expression>)`

- 返回类型：与原参数相同。

示例：

```
# 返回 Tag player 的 age 属性值的总和。
nebula> MATCH (v:player) RETURN sum(v.player.age);
+-------------------+
| sum(v.player.age) |
+-------------------+
| 1698              |
+-------------------+
```

聚合示例

```
nebula>  GO FROM "player100" OVER follow YIELD dst(edge) AS dst, properties($$).age AS age \
         | GROUP BY $-.dst \
         YIELD \
         $-.dst AS dst, \
         toInteger((sum($-.age)/count($-.age)))+avg(distinct $-.age+1)+1 AS statistics;
+-------------+------------+
| dst         | statistics |
+-------------+------------+
| "player125" | 84.0       |
| "player101" | 74.0       |
+-------------+------------+
```

最后更新: 2025年8月4日

4.4.3 内置字符串函数

本文介绍 NebulaGraph 支持的字符串函数。

注意事项

- 字符串的表示方式为用双引号或单引号包裹。
- 和 SQL 一样，nGQL 的字符索引（位置）从 `1` 开始。但是 C 语言的字符索引是从 `0` 开始的。

strcasecmp()

strcasecmp() 比较两个字符串（不区分大小写）。

语法：`strcasecmp(<string_a>,<string_b>)`

- `string_a`、`string_b`：待比较的字符串。
- 返回类型：int。
- 当 `string_a = string_b` 时，返回 `0`，当 `string_a > string_b` 是，返回大于 `0` 的数，当 `string_a < string_b` 时，返回小于 `0` 的数。

示例：

```
# 比较字符串 "a" 和 "aa"
nebula> RETURN strcasecmp("a","aa");
+----------------------+
| strcasecmp("a","aa") |
+----------------------+
| -97                  |
+----------------------+
```

lower() 和 toLower()

lower() 和 toLower() 都可以返回指定字符串的小写形式。

语法：`lower(<string>)`、`toLower(<string>)`

- `string`：指定的字符串。
- 返回类型：string。

示例：

```
# 返回 "Basketball_Player" 小写形式
nebula> RETURN lower("Basketball_Player");
+----------------------------+
| lower("Basketball_Player") |
+----------------------------+
| "basketball_player"        |
+----------------------------+
```

upper() 和 toUpper()

upper() 和 toUpper() 都可以返回指定字符串的大写形式。

语法：`upper(<string>)`、`toUpper(<string>)`

- `string`：指定的字符串。
- 返回类型：string。

示例：

```
# 返回 "Basketball_Player" 大写形式
nebula> RETURN upper("Basketball_Player");
```

```
+----------------------------+
| upper("Basketball_Player") |
+----------------------------+
| "BASKETBALL_PLAYER"        |
+----------------------------+
```

length()

length() 返回：

- 指定字符串的长度，单位：字节。
- 路径的长度，单位：跳。

语法： `length({<string>|<path>})`

- `string`：指定的字符串。
- `path`：指定的路径，使用变量表示。
- 返回类型：int。

示例：

```
# 返回字符串 "basketball" 的长度
nebula> RETURN length("basketball");
+----------------------+
| length("basketball") |
+----------------------+
| 10                   |
+----------------------+
```

```
# 返回 p 到 v2 路径的长度
nebula> MATCH p=(v:player{name:"Tim Duncan"})-->(v2) return length(p);
+-----------+
| length(p) |
+-----------+
| 1         |
| 1         |
| 1         |
+-----------+
```

trim()

trim() 删除指定字符串头部和尾部的空格。

语法： `trim(<string>)`

- `string`：指定的字符串。
- 返回类型：string。

示例：

```
# 裁剪掉字符串 " basketball player " 头部和尾部的空格
nebula> RETURN trim(" basketball player ");
+-----------------------------+
| trim(" basketball player ") |
+-----------------------------+
| "basketball player"         |
+-----------------------------+
```

ltrim()

ltrim() 删除字符串头部的空格。

语法： `ltrim(<string>)`

- `string`：指定的字符串。
- 返回类型：string。

示例：

```
# 裁剪掉字符串 " basketball player " 头部的空格
nebula> RETURN ltrim(" basketball player ");
+-----------------------------+
| ltrim(" basketball player ") |
+-----------------------------+
| "basketball player "        |
+-----------------------------+
```

rtrim()

rtrim() 删除字符串尾部的空格。

语法： `rtrim(<string>)`

- `string`：指定的字符串。
- 返回类型：string。

示例：

```
# 裁剪掉字符串 " basketball player " 尾部的空格
nebula> RETURN rtrim(" basketball player ");
+-----------------------------+
| rtrim(" basketball player ") |
+-----------------------------+
| " basketball player"        |
+-----------------------------+
```

left()

left() 返回指定字符串头部若干个字符组成的子字符串。

语法： `left(<string>,<count>)`

- `string`：指定的字符串。
- `count`：指定从头部开始的字符数量。如果 `count` 超过字符串的长度，则返回字符串本身。
- 返回类型：string。

示例：

```
# 返回字符串 "basketball_player" 从头部开始的 6 个字符组成的字符串
nebula> RETURN left("basketball_player",6);
+----------------------------+
| left("basketball_player",6) |
+----------------------------+
| "basket"                   |
+----------------------------+
```

right()

right() 返回指定字符串尾部若干个字符组成的子字符串。

语法： `right(<string>,<count>)`

- `string`：指定的字符串。
- `count`：指定从尾部开始的字符数量。如果 `count` 超过字符串的长度，则返回字符串本身。
- 返回类型：string。

示例：

```
# 返回字符串 "basketball_player" 从尾部 6 个字符组成的字符串
nebula> RETURN right("basketball_player",6);
+-----------------------------+
| right("basketball_player",6) |
+-----------------------------+
```

```
| "player"                    |
+-----------------------------+
```

lpad()

lpad() 在指定字符串的头部填充字符串至指定长度，并返回结果字符串。

语法：`lpad(<string>,<count>,<letters>)`

- `string`：指定的字符串。
- `count`：指定从尾部开始将要返回的字符数量。如果 `count` 少于 `string` 字符串的长度，则只返回 `string` 字符串从前到后的 `count` 个字符。
- `letters`：从头部填充的字符串。
- 返回类型：string。

示例：

```
# 在字符串 "abcd" 头部填充 "b" 字符，延长字符串长度至 10
nebula> RETURN lpad("abcd",10,"b");
+---------------------+
| lpad("abcd",10,"b") |
+---------------------+
| "bbbbbbabcd"        |
+---------------------+

# 返回字符串 "abcd" 的前三个字符
nebula> RETURN lpad("abcd",3,"b");
+--------------------+
| lpad("abcd",3,"b") |
+--------------------+
| "abc"              |
+--------------------+
```

rpad()

rpad() 在指定字符串的尾部填充字符串至指定长度，并返回结果字符串。

语法：`rpad(<string>,<count>,<letters>)`

- `string`：指定的字符串。
- `count`：指定从头部开始将要返回的字符数量。如果 `count` 少于 `string` 字符串的长度，则只返回 `string` 字符串从前到后的 `count` 个字符。
- `letters`：从尾部填充的字符串。
- 返回类型：string。

示例：

```
# 在字符串 "abcd" 尾部填充 "b" 字符，延长字符串长度至 10
nebula> RETURN rpad("abcd",10,"b");
+---------------------+
| rpad("abcd",10,"b") |
+---------------------+
| "abcdbbbbbb"        |
+---------------------+

# 返回字符串 "abcd" 的前三个字符
nebula> RETURN rpad("abcd",3,"b");
+--------------------+
| rpad("abcd",3,"b") |
+--------------------+
| "abc"              |
+--------------------+
```

substr() 和 substring()

substr() 和 substring() 从指定字符串的指定位置开始（不包括开始位置的字符），提取后面的若干个字符，组成新的字符串并返回。

语法： `substr(<string>,<pos>,<count>)` 、 `substring(<string>,<pos>,<count>)`

- `string`： 指定的字符串。
- `pos`： 指定开始的位置，即字符索引，数据类型为 int。
- `count`： 指定从开始位置往后提取的字符数量。
- 返回类型：string。

SUBSTR() 和 **SUBSTRING()** 的返回说明

- 如果 `pos` 为 0，表示从指定字符串头部开始提取（包括第一个字符）。
- 如果 `pos` 大于最大字符索引，则返回空字符串。
- 如果 `pos` 是负数，则返回 `BAD_DATA` 。
- 如果省略 `count` ，则返回从 `pos` 位置开始到字符串末尾的子字符串。
- 如果 `count` 为 0，则返回空字符串。
- 使用 `NULL` 作为任何参数会出现错误。

> openCypher 兼容性
>
> 在 openCypher 中，如果字符串 `a` 为 `null` ，会返回 `null` 。

示例：

```
# 返回字符串 "abcdefg" 从下标 2 开始的 4 个字符组成的子字符串
nebula> RETURN substr("abcdefg",2,4);
+----------------------+
| substr("abcdefg",2,4) |
+----------------------+
| "cdef"               |
+----------------------+

# 返回字符串 "abcdefg" 从下标 0 开始的 4 个字符组成的子字符串
nebula> RETURN substr("abcdefg",0,4);
+----------------------+
| substr("abcdefg",0,4) |
+----------------------+
| "abcd"               |
+----------------------+

# 返回字符串 "abcdefg" 从下标 0 开始到结尾的字符组成的子字符串
nebula> RETURN substr("abcdefg",2);
+--------------------+
| substr("abcdefg",2) |
+--------------------+
| "cdefg"            |
+--------------------+
```

reverse()

reverse() 逆序返回指定的字符串。

语法： `reverse(<string>)`

- `string`： 指定的字符串。
- 返回类型：string。

示例：

```
# 逆序返回字符串 "abcdefg"
nebula> RETURN reverse("abcdefg");
+-------------------+
| reverse("abcdefg") |
+-------------------+
```

```
| "gfedcba"          |
+--------------------+
```

replace()

replace() 将指定字符串中的子字符串 a 替换为字符串 b。

语法：`replace(<string>,<substr_a>,<string_b>)`

- `string`：指定的字符串。
- `substr_a`：子字符串 a。
- `string_b`：字符串 b。
- 返回类型：string。

示例：

```
nebula> RETURN replace("abcdefg","cd","AAAAA");
+---------------------------------+
| replace("abcdefg","cd","AAAAA") |
+---------------------------------+
| "abAAAAAefg"                    |
+---------------------------------+
```

split()

split() 将子字符串 b 识别为分隔符，分隔指定字符串，并返回分隔后的字符串列表。

语法：`split(<string>,<substr>)`

- `string`：指定的字符串。
- `substr`：子字符串 b。
- 返回类型：list。

示例：

```
nebula> RETURN split("basketballplayer","a");
+-------------------------------+
| split("basketballplayer","a") |
+-------------------------------+
| ["b", "sketb", "llpl", "yer"] |
+-------------------------------+
```

concat()

concat() 返回所有参数连接成的字符串。

语法：`concat(<string1>,<string2>,...)`

- 函数至少需要两个或以上字符串参数。如果字符串参数只有一个，则返回该字符串参数本身。
- 如果任何一个的字符串参数为 `NULL`，则 `concat()` 函数返回值为 `NULL`。
- 返回类型：string。

示例：

```
//拼接 1，2，3
nebula> RETURN concat("1","2","3") AS r;
+-------+
| r     |
+-------+
| "123" |
+-------+

//字符串参数有 NULL
nebula> RETURN concat("1","2",NULL) AS r;
```

```
+----------+
| r        |
+----------+
| __NULL__ |
+----------+

nebula> GO FROM "player100" over follow \
        YIELD concat(src(edge), properties($^).age, properties($$).name, properties(edge).degree) AS A;
+------------------------------+
| A                            |
+------------------------------+
| "player10042Tony Parker95"   |
| "player10042Manu Ginobili95" |
+------------------------------+
```

concat_ws()

concat_ws() 返回用分隔符（separator）连接的所有字符串。

语法：`concat_ws(<separator>,<string1>,<string2>,... )`

- 函数至少需要两个或以上字符串参数。
- 如果分隔符为 `NULL` 时，`concat_ws()` 函数才返回 `NULL`。
- 如果分隔符不为 `NULL`，字符串参数只有一个，则返回该字符串参数本身。
- 字符串参数存在 `NULL` 值时，忽略 `NULL` 值，继续连接下一个参数。

示例：

```
//分隔符为 +，连接 a，b，c。
nebula> RETURN concat_ws("+","a","b","c") AS r;
+---------+
| r       |
+---------+
| "a+b+c" |
+---------+

//分隔符为 NULL。
neubla> RETURN concat_ws(NULL,"a","b","c") AS r;
+----------+
| r        |
+----------+
| __NULL__ |
+----------+

//分隔符为 +，字符串参数有 NULL。
nebula> RETURN concat_ws("+","a",NULL,"b","c") AS r;
+---------+
| r       |
+---------+
| "a+b+c" |
+---------+

//分隔符为 +。字符串参数只有一个
nebula> RETURN concat_ws("+","a") AS r;
+-----+
| r   |
+-----+
| "a" |
+-----+

nebula> GO FROM "player100" over follow \
        YIELD concat_ws(" ",src(edge), properties($^).age, properties($$).name, properties(edge).degree) AS A;
+---------------------------------+
| A                               |
+---------------------------------+
| "player100 42 Tony Parker 95"   |
| "player100 42 Manu Ginobili 95" |
+---------------------------------+
```

extract()

extract() 从指定字符串中提取符合正则表达式的子字符串。

语法：`extract(<string>,"<regular_expression>")`

- `string`：指定的字符串。
- `regular_expression`：正则表达式。
- 返回类型：list。

示例：

```
nebula> MATCH (a:player)-[b:serve]-(c:team{name: "Lakers"}) \
        WHERE a.player.age > 45 \
        RETURN extract(a.player.name, "\\w+") AS result;
+----------------------------+
| result                     |
+----------------------------+
| ["Shaquille", "O", "Neal"] |
+----------------------------+

nebula> MATCH (a:player)-[b:serve]-(c:team{name: "Lakers"}) \
        WHERE a.player.age > 45 \
        RETURN extract(a.player.name, "hello") AS result;
+--------+
| result |
+--------+
| []     |
+--------+
```

json_extract() 函数

json_extract() 将指定 JSON 字符串转换为 map 类型。

语法：`extract(<string>)`

- `string`：指定字符串，为 JSON 格式。
- 返回类型：map。

> Caution
>
> - 目前仅支持 Bool、Double、Int、String 和 NULL 类型数据。
> - 仅支持深度为 1 的 Map 嵌套，如果嵌套深度为 2 及以上，嵌套项保留为空。

示例：

```
nebula> YIELD json_extract('{"a": 1, "b": {}, "c": {"d": true}}') AS result;
+-----------------------------+
| result                      |
+-----------------------------+
| {a: 1, b: {}, c: {d: true}} |
+-----------------------------+
```

最后更新: 2025年8月4日

4.4.4 内置日期时间函数

NebulaGraph 支持以下内置日期时间函数。

函数	说明
int now()	根据当前系统返回当前时间戳。
timestamp timestamp()	根据当前系统返回当前时间戳。
date date()	根据当前系统返回当前日期（UTC 时间）。
time time()	根据当前系统返回当前时间（UTC 时间）。
datetime datetime()	根据当前系统返回当前日期和时间（UTC 时间）。
map duration()	持续时间。可以用于对指定时间进行计算。

详细信息参见日期和时间类型。

示例

```
nebula>  RETURN now(), timestamp(), date(), time(), datetime();
+------------+-------------+------------+-----------------+----------------------------+
| now()      | timestamp() | date()     | time()          | datetime()                 |
+------------+-------------+------------+-----------------+----------------------------+
| 1640057560 | 1640057560  | 2021-12-21 | 03:32:40.351000 | 2021-12-21T03:32:40.351000 |
+------------+-------------+------------+-----------------+----------------------------+
```

最后更新: 2025年8月4日

4.4.5 Schema 相关函数

本文介绍 NebulaGraph 支持的 Schema 相关的函数。

Schema 相关的函数分为两类：

- 适用于原生 nGQL 语句
- 适用于 openCypher 兼容语句

原生 nGQL 语句适用

原生 nGQL 语句的 `YIELD` 和 `WHERE` 子句中可以使用如下介绍的函数。

> Note
>
> 由于 vertex、edge、vertices、edges、path 属于关键字，使用时需要用 `AS <alias>` 设置别名才能正常使用。例如 `GO FROM "player100" OVER follow YIELD edge AS e;`。

ID(VERTEX)

id(vertex) 返回点 ID。

语法：`id(vertex)`

- 返回类型：和点 ID 的类型保持一致。

示例：

```
nebula> LOOKUP ON player WHERE player.age  > 45 YIELD id(vertex);
+-------------+
| id(VERTEX)  |
+-------------+
| "player144" |
| "player140" |
+-------------+
```

PROPERTIES(VERTEX)

properties(vertex) 返回点的所有属性。

语法：`properties(vertex)`

- 返回类型：map。

示例：

```
nebula> LOOKUP ON player WHERE player.age  > 45 \
        YIELD properties(vertex);
+-------------------------------------+
| properties(VERTEX)                  |
+-------------------------------------+
| {age: 47, name: "Shaquille O'Neal"} |
| {age: 46, name: "Grant Hill"}       |
+-------------------------------------+
```

用户也可以使用属性引用符（`$^` 和 `$$`）替代函数 `properties()` 中的 `vertex` 参数来获取点的所有属性。

- `$^` 表示探索开始时的点数据。例如 `GO FROM "player100" OVER follow reversely YIELD properties($^)` 中，`$^` 指 `player100` 这个点。
- `$$` 表示探索结束的终点数据。

`properties($^)` 和 `properties($$)` 一般用于 `GO` 语句中。更多信息，请参见属性引用符。

Caution

用户可以通过 `properties().<property_name>` 来获取点的指定属性。但是不建议使用这种方式获取指定属性，因为 `properties()` 函数返回所有属性，这样会降低查询性能。

PROPERTIES(EDGE)

properties(edge) 返回边的所有属性。

语法： `properties(edge)`

- 返回类型：map。

示例：

```
nebula> GO FROM "player100" OVER follow \
        YIELD properties(edge);
+------------------+
| properties(EDGE) |
+------------------+
| {degree: 95}     |
| {degree: 95}     |
+------------------+
```

Warning

用户可以通过 `properties(edge).<property_name>` 来获取边的指定属性。但是不建议使用这种方式获取指定属性，因为 `properties(edge)` 函数返回边的所有属性，这样会降低查询性能。

TYPE(EDGE)

type(edge) 返回边的 Edge type。

语法： `type(edge)`

- 返回类型：string。

示例：

```
nebula> GO FROM "player100" OVER follow \
        YIELD src(edge), dst(edge), type(edge), rank(edge);
+-------------+-------------+------------+------------+
| src(EDGE)   | dst(EDGE)   | type(EDGE) | rank(EDGE) |
+-------------+-------------+------------+------------+
| "player100" | "player101" | "follow"   | 0          |
| "player100" | "player125" | "follow"   | 0          |
+-------------+-------------+------------+------------+
```

SRC(EDGE)

src(edge) 返回边的起始点 ID。

语法： `src(edge)`

- 返回类型：和点 ID 的类型保持一致。

示例：

```
nebula> GO FROM "player100" OVER follow \
        YIELD src(edge), dst(edge);
+-------------+-------------+
| src(EDGE)   | dst(EDGE)   |
+-------------+-------------+
| "player100" | "player101" |
| "player100" | "player125" |
+-------------+-------------+
```

> Note
>
> src(edge) 和 properties(`$^`) 查找起始点的语义不同。src(edge) 始终表示图数据库中边的起始点 ID，而 properties(`$^`) 表示探索开始时的点数据，例如示例中 `GO FROM "player100"` 中 `player100` 这个点的数据。

DST(EDGE)

dst(edge) 返回边的目的点 ID。

语法： `dst(edge)`

- 返回类型：和点 ID 的类型保持一致。

示例：

```
nebula> GO FROM "player100" OVER follow \
        YIELD src(edge), dst(edge);
+-------------+-------------+
| src(EDGE)   | dst(EDGE)   |
+-------------+-------------+
| "player100" | "player101" |
| "player100" | "player125" |
+-------------+-------------+
```

> Note
>
> dst(edge) 始终表示图数据库中边的目的点 ID。

RANK(EDGE)

rank(edge) 返回边的 rank。

语法： `rank(edge)`

- 返回类型：int。

示例：

```
nebula> GO FROM "player100" OVER follow \
        YIELD src(edge), dst(edge), rank(edge);
+-------------+-------------+------------+
| src(EDGE)   | dst(EDGE)   | rank(EDGE) |
+-------------+-------------+------------+
| "player100" | "player101" | 0          |
| "player100" | "player125" | 0          |
+-------------+-------------+------------+
```

VERTEX

vertex 返回点的信息。包括点 ID、Tag、属性和值。需要用 `AS <alias>` 设置别名。

语法： `vertex`

示例：

```
nebula> LOOKUP ON player WHERE player.age  > 45 YIELD vertex AS v;
+-----------------------------------------------------------+
| v                                                         |
+-----------------------------------------------------------+
| ("player144" :player{age: 47, name: "Shaquille O'Neal"}) |
| ("player140" :player{age: 46, name: "Grant Hill"})       |
+-----------------------------------------------------------+
```

EDGE

edge 返回边的信息。包括 Edge type、起始点 ID、目的点 ID、rank、属性和值。需要用 `AS <alias>` 设置别名。

语法： `edge`

示例：

```
nebula> GO FROM "player100" OVER follow YIELD edge AS e;
+----------------------------------------------------+
| e                                                  |
+----------------------------------------------------+
| [:follow "player100"->"player101" @0 {degree: 95}] |
| [:follow "player100"->"player125" @0 {degree: 95}] |
+----------------------------------------------------+
```

VERTICES

vertices 返回子图中的点的信息。详情参见 GET SUBGRAPH。

EDGES

edges 返回子图中的边的信息。详情参见 GET SUBGRAPH。

PATH

path 返回路径信息。详情参见 FIND PATH。

openCypher 兼容语句适用

openCypher 兼容语句的 `RETURN` 和 `WHERE` 子句中可以使用如下介绍的函数。

ID()

id() 返回点 ID。

语法： `id(<vertex>)`

- 返回类型：和点 ID 的类型保持一致。

示例：

```
nebula> MATCH (v:player) RETURN id(v);
+-------------+
| id(v)       |
+-------------+
| "player129" |
| "player115" |
| "player106" |
| "player102" |
...
```

TAGS() 和LABELS()

tags() 和labels() 返回点的 Tag。

语法： `tags(<vertex>)` 、 `labels(<vertex>)`

- 返回类型：list。

示例：

```
nebula> MATCH (v) WHERE id(v) == "player100" \
        RETURN tags(v);
+------------+
| tags(v)    |
+------------+
| ["player"] |
+------------+
```

PROPERTIES()

properties() 返回点或边的所有属性。

语法： `properties(<vertex_or_edge>)`

- 返回类型：map。

示例：

```
nebula> MATCH (v:player)-[e:follow]-() RETURN properties(v),properties(e);
+---------------------------------------+---------------+
| properties(v)                         | properties(e) |
+---------------------------------------+---------------+
| {age: 31, name: "Stephen Curry"}      | {degree: 90}  |
| {age: 47, name: "Shaquille O'Neal"}   | {degree: 100} |
| {age: 34, name: "LeBron James"}       | {degree: 13}  |
...
```

TYPE()

type() 返回边的 Edge type。

语法：`type(<edge>)`

- 返回类型：string。

示例：

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->() \
        RETURN type(e);
+----------+
| type(e)  |
+----------+
| "serve"  |
| "follow" |
| "follow" |
+----------+
```

TYPEID()

typeid() 返回边的 Edge type 的内部 ID 值，可以用正负来判断方向。

语法：`typeid(<edge>)`

- 返回类型：int。

示例：

```
nebula> MATCH (v:player)-[e:follow]-(v2) RETURN e,typeid(e), \
        CASE WHEN typeid(e) > 0 \
        THEN "正向" ELSE "反向" END AS direction \
        LIMIT 5;
+----------------------------------------------------+-----------+-----------+
| e                                                  | typeid(e) | direction |
+----------------------------------------------------+-----------+-----------+
| [:follow "player127"->"player114" @0 {degree: 90}] | 5         | "正向"    |
| [:follow "player127"->"player148" @0 {degree: 70}] | 5         | "正向"    |
| [:follow "player148"->"player127" @0 {degree: 80}] | -5        | "反向"    |
| [:follow "player147"->"player136" @0 {degree: 90}] | 5         | "正向"    |
| [:follow "player136"->"player147" @0 {degree: 90}] | -5        | "反向"    |
+----------------------------------------------------+-----------+-----------+
```

SRC()

src() 返回边的起始点 ID。

语法：`src(<edge>)`

- 返回类型：和点 ID 的类型保持一致。

示例：

```
nebula> MATCH ()-[e]->(v:player{name:"Tim Duncan"}) \
        RETURN src(e);
+-------------+
| src(e)      |
+-------------+
| "player125" |
| "player113" |
| "player102" |
...
```

DST()

dst() 返回边的目的点 ID。

语法： `dst(<edge>)`

- 返回类型：和点 ID 的类型保持一致。

示例：

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->() \
        RETURN dst(e);
+-------------+
| dst(e)      |
+-------------+
| "team204"   |
| "player101" |
| "player125" |
+-------------+
```

STARTNODE()

startNode() 获取一条路径并返回它的起始点信息，包括点 ID、Tag、属性和值。

语法： `startNode(<path>)`

示例：

```
nebula> MATCH p = (a :player {name : "Tim Duncan"})-[r:serve]-(t) \
        RETURN startNode(p);
+----------------------------------------------------+
| startNode(p)                                       |
+----------------------------------------------------+
| ("player100" :player{age: 42, name: "Tim Duncan"}) |
+----------------------------------------------------+
```

ENDNODE()

endNode() 获取一条路径并返回它的目的点信息，包括点 ID、Tag、属性和值。

语法： `endNode(<path>)`

示例：

```
nebula> MATCH p = (a :player {name : "Tim Duncan"})-[r:serve]-(t) \
        RETURN endNode(p);
+----------------------------------+
| endNode(p)                       |
+----------------------------------+
| ("team204" :team{name: "Spurs"}) |
+----------------------------------+
```

RANK()

rank() 返回边的 rank。

语法： `rank(<edge>)`

- 返回类型：int。

示例：

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->() \
        RETURN rank(e);
+---------+
| rank(e) |
+---------+
| 0       |
| 0       |
| 0       |
+---------+
```

最后更新: 2025年8月4日

4.4.6 列表函数

本文介绍 NebulaGraph 支持的列表（List）函数。部分列表函数在原生 nGQL 语句和 openCypher 兼容语句中的语法不同。

注意事项

- 和 SQL 一样，nGQL 的字符索引（位置）从 `1` 开始。但是 C 语言的字符索引是从 `0` 开始的。

通用

RANGE()

range() 返回指定整数范围 `[start,end]` 内固定步长的列表。

语法：`range(start, end [, step])`

- `step`：可选参数。步长。默认为 `1`。
- 返回类型：list。

示例：

```
nebula> RETURN range(1,9,2);
+-----------------+
| range(1,9,2)    |
+-----------------+
| [1, 3, 5, 7, 9] |
+-----------------+
```

REVERSE()

reverse() 返回将原列表逆序排列的新列表。

语法：`reverse(<list>)`

- 返回类型：list。

示例：

```
nebula> WITH [NULL, 4923, 'abc', 521, 487] AS ids \
        RETURN reverse(ids);
+-----------------------------------+
| reverse(ids)                      |
+-----------------------------------+
| [487, 521, "abc", 4923, __NULL__] |
+-----------------------------------+
```

TAIL()

tail() 返回不包含原列表第一个元素的新列表。

语法：`tail(<list>)`

- 返回类型：list。

示例：

```
nebula> WITH [NULL, 4923, 'abc', 521, 487] AS ids \
        RETURN tail(ids);
+-------------------------+
| tail(ids)               |
+-------------------------+
| [4923, "abc", 521, 487] |
+-------------------------+
```

HEAD()

head() 返回列表的第一个元素。

语法： `head(<list>)`

- 返回类型：与原列表内的元素类型相同。

示例：

```
nebula> WITH [NULL, 4923, 'abc', 521, 487] AS ids \
        RETURN head(ids);
+-----------+
| head(ids) |
+-----------+
| __NULL__  |
+-----------+
```

LAST()

last() 返回列表的最后一个元素。

语法： `last(<list>)`

- 返回类型：与原列表内的元素类型相同。

示例：

```
nebula> WITH [NULL, 4923, 'abc', 521, 487] AS ids \
        RETURN last(ids);
+-----------+
| last(ids) |
+-----------+
| 487       |
+-----------+
```

REDUCE()

reduce() 将表达式逐个应用于列表中的元素，然后和累加器中的当前结果累加，最后返回完整结果。该函数将遍历给定列表中的每个元素 e，在 e 上运行表达式并和累加器的当前结果累加，将新的结果存储在累加器中。这个函数类似于函数式语言（如 Lisp 和 Scala）中的 fold 或 reduce 方法。

> openCypher 兼容性
>
> 在 openCypher 中， `reduce()` 函数没有定义。nGQL 使用了 Cypher 方式实现 `reduce()` 函数。

语法： `reduce(<accumulator> = <initial>, <variable> IN <list> | <expression>)`

- `accumulator` ：在遍历列表时保存累加结果。
- `initial` ：为 `accumulator` 提供初始值的表达式或值。
- `variable` ：为列表引入一个变量，决定使用列表中的哪个元素。
- `list` ：列表或列表表达式。
- `expression` ：该表达式将对列表中的每个元素运行一次，并将结果累加至 `accumulator` 。
- 返回类型：取决于提供的参数，以及表达式的语义。

示例：

```
nebula> RETURN reduce(totalNum = -4 * 5, n IN [1, 2] | totalNum + n * 2) AS r;
+-----+
| r   |
+-----+
| -14 |
+-----+

nebula> MATCH p = (n:player{name:"LeBron James"})<-[:follow]-(m) \
        RETURN  nodes(p)[0].player.age AS src1,  nodes(p)[1].player.age AS dst2,  \
        reduce(totalAge = 100, n IN nodes(p) | totalAge + n.player.age) AS sum;
+------+------+-----+
| src1 | dst2 | sum |
+------+------+-----+
| 34   | 31   | 165 |
| 34   | 29   | 163 |
| 34   | 33   | 167 |
```

```
| 34   | 26   | 160 |
| 34   | 34   | 168 |
| 34   | 37   | 171 |
+------+------+-----+

nebula> LOOKUP ON player WHERE player.name == "Tony Parker" YIELD id(vertex) AS VertexID \
        |  GO FROM $-.VertexID over follow \
        WHERE properties(edge).degree != reduce(totalNum = 5, n IN range(1, 3) | properties($$).age + totalNum + n) \
        YIELD properties($$).name AS id, properties($$).age AS age, properties(edge).degree AS degree;
+---------------------+-----+--------+
| id                  | age | degree |
+---------------------+-----+--------+
| "Tim Duncan"        | 42  | 95     |
| "LaMarcus Aldridge" | 33  | 90     |
| "Manu Ginobili"     | 41  | 95     |
+---------------------+-----+--------+
```

原生 **nGQL** 语句适用

KEYS()

keys() 返回一个列表，包含字符串形式的点、边的所有属性。

语法：`keys({vertex | edge})`

- 返回类型：list。

示例：

```
nebula> LOOKUP ON player \
        WHERE player.age  > 45 \
        YIELD keys(vertex);
+-----------------+
| keys(VERTEX)    |
+-----------------+
| ["age", "name"] |
| ["age", "name"] |
+-----------------+
```

LABELS()

labels() 返回点的 Tag 列表。

语法：`labels(verte)`

- 返回类型：list。

示例：

```
nebula> FETCH PROP ON * "player101", "player102", "team204" \
        YIELD labels(vertex);
+----------------+
| labels(VERTEX) |
+----------------+
| ["player"]     |
| ["player"]     |
| ["team"]       |
+----------------+
```

openCypher 兼容语句适用

KEYS()

keys() 返回一个列表，包含字符串形式的点、边或映射的所有属性。

语法：`keys(<vertex_or_edge>)`

- 返回类型：list。

示例：

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->() \
        RETURN keys(e);
+----------------------------+
| keys(e)                    |
+----------------------------+
| ["end_year", "start_year"] |
```

```
| ["degree"]                 |
| ["degree"]                 |
+----------------------------+
```

LABELS()

labels() 返回点的 Tag 列表。

语法： `labels(<vertex>)`

- 返回类型：list。

示例：

```
nebula> MATCH (v)-[e:serve]->() \
        WHERE id(v)=="player100" \
        RETURN labels(v);
+------------+
| labels(v)  |
+------------+
| ["player"] |
+------------+
```

NODES()

nodes() 返回路径中所有点的列表。包括点 ID、Tag、属性和值。

语法： `nodes(<path>)`

- 返回类型：list。

示例：

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-->(v2) \
        RETURN nodes(p);
+-------------------------------------------------------------------------------------------------------------+
| nodes(p)                                                                                                    |
+-------------------------------------------------------------------------------------------------------------+
| [("player100" :player{age: 42, name: "Tim Duncan"}), ("team204" :team{name: "Spurs"})]                      |
| [("player100" :player{age: 42, name: "Tim Duncan"}), ("player101" :player{age: 36, name: "Tony Parker"})]   |
| [("player100" :player{age: 42, name: "Tim Duncan"}), ("player125" :player{age: 41, name: "Manu Ginobili"})] |
+-------------------------------------------------------------------------------------------------------------+
```

RELATIONSHIPS()

relationships() 返回路径中所有关系的列表。

语法： `relationships(<path>)`

- 返回类型：list。

示例：

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-->(v2) \
        RETURN relationships(p);
+-------------------------------------------------------------------------+
| relationships(p)                                                        |
+-------------------------------------------------------------------------+
| [[:serve "player100"->"team204" @0 {end_year: 2016, start_year: 1997}]] |
| [[:follow "player100"->"player101" @0 {degree: 95}]]                    |
| [[:follow "player100"->"player125" @0 {degree: 95}]]                    |
+-------------------------------------------------------------------------+
```

最后更新: 2025年8月4日

4.4.7 类型转换函数

本文介绍 NebulaGraph 支持的类型转换函数。

toBoolean()

toBoolean() 将字符串转换为布尔。

语法：`toBoolean(<value>)`

- 返回类型：bool。

示例：

```
nebula> UNWIND [true, false, 'true', 'false', NULL] AS b \
        RETURN toBoolean(b) AS b;
+----------+
| b        |
+----------+
| true     |
| false    |
| true     |
| false    |
| __NULL__ |
+----------+
```

toFloat()

toFloat() 将整数或字符串转换为浮点数。

语法：`toFloat(<value>)`

- 返回类型：float。

示例：

```
nebula> RETURN toFloat(1), toFloat('1.3'), toFloat('1e3'), toFloat('not a number');
+------------+----------------+----------------+-------------------------+
| toFloat(1) | toFloat("1.3") | toFloat("1e3") | toFloat("not a number") |
+------------+----------------+----------------+-------------------------+
| 1.0        | 1.3            | 1000.0         | __NULL__                |
+------------+----------------+----------------+-------------------------+
```

toString()

toString() 将任意非复合数据类型数据转换为字符串类型。

语法：`toString(<value>)`

- 返回类型：string。

示例：

```
nebula> RETURN toString(9669) AS int2str, toString(null) AS null2str;
+---------+----------+
| int2str | null2str |
+---------+----------+
| "9669"  | __NULL__ |
+---------+----------+
```

toInteger()

toInteger() 将浮点或字符串转换为整数。

语法：`toInteger(<value>)`

- 返回类型：int。

示例：

```
nebula> RETURN toInteger(1), toInteger('1'), toInteger('1e3'), toInteger('not a number');
+--------------+----------------+------------------+---------------------------+
| toInteger(1) | toInteger("1") | toInteger("1e3") | toInteger("not a number") |
+--------------+----------------+------------------+---------------------------+
| 1            | 1              | 1000             | __NULL__                  |
+--------------+----------------+------------------+---------------------------+
```

toSet()

toSet() 将列表或集合转换为集合。

语法： `toSet(<value>)`

- 返回类型：set。

示例：

```
nebula> RETURN toSet(list[1,2,3,1,2]) AS list2set;
+-----------+
| list2set  |
+-----------+
| {3, 1, 2} |
+-----------+
```

hash()

hash() 返回参数的哈希值。其参数可以是数字、字符串、列表、布尔值、NULL 等类型的值，或者计算结果为这些类型的表达式。

`hash()` 函数采用 MurmurHash2 算法，种子（seed）为 `0xc70f6907UL` 。用户可以在 `MurmurHash2.h` 中查看其源代码。

在 Java 中的调用方式如下：

```
MurmurHash2.hash64("to_be_hashed".getBytes(),"to_be_hashed".getBytes().length, 0xc70f6907)
```

语法： `hash(<string>)`

- 返回类型：int。

示例：

```
nebula> RETURN hash("abcde");
+--------------------+
| hash("abcde")      |
+--------------------+
| 811036730794841393 |
+--------------------+

nebula> YIELD hash([1,2,3]);
+----------------+
| hash([1,2,3])  |
+----------------+
| 11093822460243 |
+----------------+

nebula> YIELD hash(NULL);
+------------+
| hash(NULL) |
+------------+
| -1         |
+------------+

nebula> YIELD hash(toLower("HELLO NEBULA"));
+-------------------------------+
| hash(toLower("HELLO NEBULA")) |
+-------------------------------+
| -8481157362655072082          |
+-------------------------------+
```

最后更新: 2025年8月4日

4.4.8 条件表达式函数

本文介绍 NebulaGraph 支持的条件表达式函数。

CASE

`CASE` 表达式使用条件来过滤传参。和 openCypher 一样，nGQL 提供两种形式的 `CASE` 表达式：简单形式和通用形式。

`CASE` 表达式会遍历所有条件，并在满足第一个条件时停止读取后续条件，然后返回结果。如果不满足任何条件，将通过 `ELSE` 子句返回结果。如果没有 `ELSE` 子句且不满足任何条件，则返回 `NULL`。

简单形式

- 语法

```
CASE <comparer>
WHEN <value> THEN <result>
[WHEN ...]
[ELSE <default>]
END
```

> Caution
>
> `CASE` 表达式一定要用 `END` 结尾。

参数	说明
`comparer`	用于与 `value` 进行比较的值或者有效表达式。
`value`	和 `comparer` 进行比较，如果匹配，则满足此条件。
`result`	如果 `value` 匹配 `comparer`，则返回该 `result`。
`default`	如果没有条件匹配，则返回该 `default`。

- 示例

```
nebula> RETURN \
        CASE 2+3 \
        WHEN 4 THEN 0 \
        WHEN 5 THEN 1 \
        ELSE -1 \
        END \
        AS result;
+--------+
| result |
+--------+
| 1      |
+--------+
```

```
nebula> GO FROM "player100" OVER follow \
        YIELD properties($$).name AS Name, \
        CASE properties($$).age > 35 \
        WHEN true THEN "Yes" \
        WHEN false THEN "No" \
        ELSE "Nah" \
        END \
        AS Age_above_35;
+-----------------+--------------+
| Name            | Age_above_35 |
+-----------------+--------------+
| "Tony Parker"   | "Yes"        |
| "Manu Ginobili" | "Yes"        |
+-----------------+--------------+
```

通用形式

- 语法

```
CASE
WHEN <condition> THEN <result>
[WHEN ...]
[ELSE <default>]
END
```

参数	说明
`condition`	如果条件 `condition` 为 true，表示满足此条件。
`result`	`condition` 为 true，则返回此 `result`。
`default`	如果没有条件匹配，则返回该 `default`。

- 示例

```
nebula> YIELD \
        CASE WHEN 4 > 5 THEN 0 \
        WHEN 3+4==7 THEN 1 \
        ELSE 2 \
        END \
        AS result;
+--------+
| result |
+--------+
| 1      |
+--------+
```

```
nebula> MATCH (v:player) WHERE v.player.age > 30 \
        RETURN v.player.name AS Name,  \
        CASE \
        WHEN v.player.name STARTS WITH "T" THEN "Yes" \
        ELSE "No" \
        END \
        AS Starts_with_T;
+---------------------+---------------+
| Name                | Starts_with_T |
+---------------------+---------------+
| "Tim Duncan"        | "Yes"         |
| "LaMarcus Aldridge" | "No"          |
| "Tony Parker"       | "Yes"         |
+---------------------+---------------+
```

简单形式和通用形式的区别

为了避免误用简单形式和通用形式，用户需要了解它们的差异。请参见如下示例：

```
nebula> GO FROM "player100" OVER follow \
        YIELD properties($$).name AS Name, properties($$).age AS Age, \
        CASE properties($$).age \
        WHEN properties($$).age > 35 THEN "Yes" \
        ELSE "No" \
        END \
        AS Age_above_35;
+-----------------+-----+--------------+
| Name            | Age | Age_above_35 |
+-----------------+-----+--------------+
| "Tony Parker"   | 36  | "No"         |
| "Manu Ginobili" | 41  | "No"         |
+-----------------+-----+--------------+
```

示例本意为当玩家年龄大于 35 时输出 `Yes`。但是查看输出结果，年龄为 36 时输出的却是 `No`。

这是因为查询使用了简单形式的 `CASE` 表达式，比较对象是 `$$.player.age` 和 `$$.player.age > 35`。当年龄为 36 时：

- `$$.player.age` 的值为 `36`，数据类型为 int。
- `$$.player.age > 35` 的值为 `true`，数据类型为 boolean。

这两种数据类型无法匹配，不满足条件，因此返回 `No`。

coalesce()

coalesce() 返回所有表达式中第一个非空元素。

语法： `coalesce(<expression_1>[,<expression_2>...])`

- 返回类型：与原元素类型相同。

示例：

```
nebula> RETURN coalesce(null,[1,2,3]) as result;
+-----------+
| result    |
+-----------+
| [1, 2, 3] |
+-----------+

nebula> RETURN coalesce(null) as result;
+----------+
| result   |
+----------+
| __NULL__ |
+----------+
```

最后更新: 2025年8月4日

4.4.9 谓词函数

谓词函数只返回 `true` 或 `false` ，通常用于 `WHERE` 子句中。

NebulaGraph 支持以下谓词函数。

函数	说明
exists()	如果指定的属性在点、边或映射中存在，则返回 `true` ，否则返回 `false` 。
any()	如果指定的谓词适用于列表中的至少一个元素，则返回 `true` ，否则返回 `false` 。
all()	如果指定的谓词适用于列表中的每个元素，则返回 `true` ，否则返回 `false` 。
none()	如果指定的谓词不适用于列表中的任何一个元素，则返回 `true` ，否则返回 `false` 。
single()	如果指定的谓词适用于列表中的唯一一个元素，则返回 `true` ，否则返回 `false` 。

> Note
>
> 如果列表为空，或者列表中的所有元素都为空，则返回 NULL。

> Compatibility
>
> 在 openCypher 中只定义了函数 `exists()` ，其他几个函数依赖于具体实现。

语法

```
<predicate>(<variable> IN <list> WHERE <condition>)
```

示例

```
nebula> RETURN any(n IN [1, 2, 3, 4, 5, NULL] \
        WHERE n > 2) AS r;
+------+
| r    |
+------+
| true |
+------+

nebula> RETURN single(n IN range(1, 5) \
        WHERE n == 3) AS r;
+------+
| r    |
+------+
| true |
+------+

nebula> RETURN none(n IN range(1, 3) \
        WHERE n == 0) AS r;
+------+
| r    |
+------+
| true |
+------+

nebula> WITH [1, 2, 3, 4, 5, NULL] AS a \
        RETURN any(n IN a WHERE n > 2);
+-------------------------+
| any(n IN a WHERE (n>2)) |
+-------------------------+
| true                    |
+-------------------------+

nebula> MATCH p = (n:player{name:"LeBron James"})<-[:follow]-(m) \
        RETURN nodes(p)[0].player.name AS n1, nodes(p)[1].player.name AS n2, \
        all(n IN nodes(p) WHERE n.player.name NOT STARTS WITH "D") AS b;
+----------------+-------------------+-------+
| n1             | n2                | b     |
+----------------+-------------------+-------+
| "LeBron James" | "Danny Green"     | false |
```

```
| "LeBron James" | "Dejounte Murray" | false |
| "LeBron James" | "Chris Paul"      | true  |
| "LeBron James" | "Kyrie Irving"    | true  |
| "LeBron James" | "Carmelo Anthony" | true  |
| "LeBron James" | "Dwyane Wade"     | false |
+----------------+-------------------+-------+

nebula> MATCH p = (n:player{name:"LeBron James"})-[:follow]->(m) \
        RETURN single(n IN nodes(p) WHERE n.player.age > 40) AS b;
+------+
| b    |
+------+
| true |
+------+

nebula> MATCH (n:player) \
        RETURN exists(n.player.id), n IS NOT NULL;
+---------------------+---------------+
| exists(n.player.id) | n IS NOT NULL |
+---------------------+---------------+
| false               | true          |
...

nebula> MATCH (n:player) \
        WHERE exists(n['name']) \
        RETURN n;
+----------------------------------------------------------------+
| n                                                              |
+----------------------------------------------------------------+
| ("player105" :player{age: 31, name: "Danny Green"})           |
| ("player109" :player{age: 34, name: "Tiago Splitter"})        |
| ("player111" :player{age: 38, name: "David West"})            |
...
```

最后更新: 2025年8月4日

4.4.10 geo 函数

geo 函数用于生成地理空间（GEOGRAPHY）数据类型的值或对其执行操作。

关于地理空间数据类型说明请参见地理空间。

函数说明

函数	返回类型	说明
ST_Point(longitude, latitude)	`GEOGRAPHY`	创建包含一个点的地理空间。
ST_GeogFromText(wkt_string)	`GEOGRAPHY`	返回与传入的 WKT 字符串形式相对应的 GEOGRAPHY。
ST_ASText(geography)	`STRING`	返回传入的 GEOGRAPHY 的 WKT 字符串形式。
ST_Centroid(geography)	`GEOGRAPHY`	以单点 GEOGRAPHY 的形式返回传入的 GEOGRAPHY 的形心。
ST_ISValid(geography)	`BOOL`	返回传入的 GEOGRAPHY 是否有效。
ST_Intersects(geography_1, geography_2)	`BOOL`	返回传入的两个 GEOGRAPHY 是否有交集。
ST_Covers(geography_1, geography_2)	`BOOL`	返回 geography_1 是否完全包含 geography_2。如果 geography_2 中没有位于 geography_1 外部的点，返回 True。
ST_CoveredBy(geography_1, geography_2)	`BOOL`	返回 geography_2 是否完全包含 geography_1。如果 geography_1 中没有位于 geography_2 外部的点，返回 True。
ST_DWithin(geography_1, geography_2, distance)	`BOOL`	如果 geography_1 中至少有一个点与 geography_2 中的一个点的距离小于或等于 distance 参数（以米为单位）指定的距离，则返回 True。
ST_Distance(geography_1, geography_2)	`FLOAT`	返回两个非空 GEOGRAPHY 之间的最短距离（以米为单位）。
S2_CellIdFromPoint(point_geography)	`INT`	返回覆盖点 GEOGRAPHY 的 S2 单元ID。
S2_CoveringCellIds(geography)	`ARRAY<INT64>`	返回覆盖传入的 GEOGRAPHY 的 S2 单元 ID 的数组。

示例

```
nebula> RETURN ST_ASText(ST_Point(1,1));
+------------------------+
| ST_ASText(ST_Point(1,1)) |
+------------------------+
| "POINT(1 1)"           |
+------------------------+

nebula> RETURN ST_ASText(ST_GeogFromText("POINT(3 8)"));
+---------------------------------------+
| ST_ASText(ST_GeogFromText("POINT(3 8)")) |
+---------------------------------------+
| "POINT(3 8)"                          |
+---------------------------------------+

nebula> RETURN ST_ASTEXT(ST_Centroid(ST_GeogFromText("LineString(0 1,1 0)")));
+---------------------------------------------------------+
| ST_ASTEXT(ST_Centroid(ST_GeogFromText("LineString(0 1,1 0)"))) |
+---------------------------------------------------------+
| "POINT(0.5000380800773782 0.5000190382261059)"          |
+---------------------------------------------------------+

nebula> RETURN ST_ISValid(ST_GeogFromText("POINT(3 8)"));
+----------------------------------------+
| ST_ISValid(ST_GeogFromText("POINT(3 8)")) |
+----------------------------------------+
| true                                   |
+----------------------------------------+

nebula> RETURN ST_Intersects(ST_GeogFromText("LineString(0 1,1 0)"),ST_GeogFromText("LineString(0 0,1 1)"));
+----------------------------------------------------------------------------+
| ST_Intersects(ST_GeogFromText("LineString(0 1,1 0)"),ST_GeogFromText("LineString(0 0,1 1)")) |
+----------------------------------------------------------------------------+
| true                                                                       |
+----------------------------------------------------------------------------+

nebula> RETURN ST_Covers(ST_GeogFromText("POLYGON((0 0,10 0,10 10,0 10,0 0))"),ST_Point(1,2));
+--------------------------------------------------------------------+
```

```
| ST_Covers(ST_GeogFromText("POLYGON((0 0,10 0,10 10,0 10,0 0))"),ST_Point(1,2)) |
+--------------------------------------------------------------------------------+
| true                                                                           |
+--------------------------------------------------------------------------------+

nebula> RETURN ST_CoveredBy(ST_Point(1,2),ST_GeogFromText("POLYGON((0 0,10 0,10 10,0 10,0 0))"));
+-----------------------------------------------------------------------------------+
| ST_CoveredBy(ST_Point(1,2),ST_GeogFromText("POLYGON((0 0,10 0,10 10,0 10,0 0))")) |
+-----------------------------------------------------------------------------------+
| true                                                                              |
+-----------------------------------------------------------------------------------+

nebula> RETURN ST_dwithin(ST_GeogFromText("Point(0 0)"),ST_GeogFromText("Point(10 10)"),20000000000.0);
+---------------------------------------------------------------------------------------+
| ST_dwithin(ST_GeogFromText("Point(0 0)"),ST_GeogFromText("Point(10 10)"),20000000000) |
+---------------------------------------------------------------------------------------+
| true                                                                                  |
+---------------------------------------------------------------------------------------+

nebula> RETURN ST_Distance(ST_GeogFromText("Point(0 0)"),ST_GeogFromText("Point(10 10)"));
+----------------------------------------------------------------------------+
| ST_Distance(ST_GeogFromText("Point(0 0)"),ST_GeogFromText("Point(10 10)")) |
+----------------------------------------------------------------------------+
| 1.5685230187677438e+06                                                     |
+----------------------------------------------------------------------------+

nebula> RETURN S2_CellIdFromPoint(ST_GeogFromText("Point(1 1)"));
+---------------------------------------------------+
| S2_CellIdFromPoint(ST_GeogFromText("Point(1 1)")) |
+---------------------------------------------------+
| 1153277837650709461                               |
+---------------------------------------------------+

nebula> RETURN S2_CoveringCellIds(ST_GeogFromText("POLYGON((0 1, 1 2, 2 3, 0 1))"));
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| S2_CoveringCellIds(ST_GeogFromText("POLYGON((0 1, 1 2, 2 3, 0 1))"))                                                                                                     |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| [1152391494368201343, 1153466862374223872, 1153554823304445952, 1153836298281156608, 1153959443583467520, 1154240918560178176, 1160503736791990272, 1160591697722212352] |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
```

最后更新: 2025年8月4日

4.5 通用查询语句

4.5.1 NebulaGraph 查询语句概述

本文介绍 NebulaGraph 的通用的查询语句分类及各类语句的使用场景。

背景信息

NebulaGraph 的数据以点和边的形式存储。每个点可以有 0 或多个标签（Tag）；每条边有且仅有一个边类型（Edge Type）。标签定义点的类型以及描述点的属性；边类型定义边的类型以及描述边的属性。在查询时，可以通过指定点的标签或边的类型来限定查询的范围。更多信息，请参见数据模型。

查询语句分类

NebulaGraph 的核心查询语句可分为：

- FETCH PROP ON
- LOOKUP ON
- GO
- MATCH
- FIND PATH
- GET SUBGRAPH
- SHOW

`FETCH PROP ON` 和 `LOOKUP ON` 更多用于基础的数据查询；`GO` 和 `MATCH` 用于更复杂的查询和图数据遍历；`FIND PATH` 和 `GET SUBGRAPH` 用于图数据的路径查询和子图查询；`SHOW` 用于获取数据库的元数据信息。

用法及使用场景

FETCH PROP ON

用法：用于获取指定点或边的属性。

场景：已知具体的点 ID 或边 ID，并想获取其属性。

说明：

- 必需指定点或边的 ID。
- 必需指定点或边所属的标签或边类型。
- 使用 `YIELD` 子句指定返回的属性。

示例：

```
FETCH PROP ON player "player100" YIELD properties(vertex);
              --+---  ----+-----       -------+----------
                |         |                   |
                |         |                   |
                |         |                   +--------- 返回点的 player TAG 下所有属性
                |         |
                |         +----------------- 从 "player100" 这个点获取
                |
                +--------------------------- 获取 player 这个 TAG 下的属性
```

更多信息，请参见 FETCH PROP ON。

LOOKUP ON

用法：用于基于索引查询点或边 ID。

场景：根据属性值查找点或边的 ID。

说明：

- 必需预先定义索引。
- 必需指定点或边所属的标签或边类型。
- 使用 `YIELD` 子句指定返回的 ID。

示例：

```
LOOKUP ON player WHERE player.name == "Tony Parker" YIELD id(vertex);
          --+--- -----------------+--------------       ---+------
            |                     |                        |
            |                     |                        |
            |                     |                        +---- 返回查到点的 VID
            |                     |
            |                     +----------- 过滤条件是属性 name 的值
            |
            +---------------------------------- 根据点的 TAG player 查询
```

更多信息，请参见 LOOKUP ON。

GO

用法：用于基于给定的点进行图遍历，按需返回起始点、边或目标点的信息。可以指定遍历的深度、边的类型、方向等。

场景：复杂的图遍历，比如找到某个点的朋友、朋友的朋友等。

说明：

- 结合属性引用符(`$^` 和 `$$`)来返回起始点或目标点的属性，例如 `YIELD $^.player.name` 。
- 结合函数 `properties($^)` 和 `properties($$)` 来返回起始点或目标点的所有属性；或者在函数中指定属性名，来返回指定的属性，例如 `YIELD properties($^).name` 。
- 结合函数 `src(edge)` 和 `dst(edge)` 来返回边的起始点或目标点 ID，例如 `YIELD src(edge)` 。

示例：

```
GO 3 STEPS FROM "player102" OVER follow YIELD dst(edge);
-----+---       --+-------       -+----       ---+-----
     |            |               |              |
     |            |               |              |
     |            |               |              +--------- 返回最后一跳边的终点
     |            |               |
     |            |               +------ 从 follow 这个边的出方向探索
     |            |
     |            +--------------------- 起点是 "player102"
     |
     +---------------------------------- 探索 3 步
```

更多信息，请参见 GO。

MATCH

用法：用于执行复杂的图模式匹配查询。

场景：复杂的图模式匹配时使用，比如寻找满足特定模式的点和边的组合。

说明：

MATCH 语句兼容 OpenCypher 的语法，但是有一些差异：

- 使用 `==` 表达相等判断而不是 `=` ，例如 `WHERE player.name == "Tony Parker"` 。
- 引用点的属性时，需要指定点的标签，例如 `YIELD player.name` 。
- 新增 `WHERE id(v) == "player100"` 语法。
- 必须使用 `RETURN` 子句指定返回的信息。

示例：

```
MATCH (v:player{name:"Tim Duncan"})--(v2:player) \
        RETURN v2.player.name AS Name;
```

更多信息，请参见 MATCH。

FIND PATH

用法：用于查询给定的起始点和目标点之间的所有路径；或者查询路径中的点和边的属性时使用。

场景：查询两个点之间的所有路径。

说明：必须使用 `YIELD` 子句指定返回信息。

示例：

```
FIND SHORTEST PATH FROM "player102" TO "team204" OVER * YIELD path AS p;
-------+-----           -------+---------------- ---+--       ----+----
       |                       |                    |             |
       |                       |                    |             |
       |                       |                    |             +---------- 返回路径为 p 列
       |                       |                    |
       |                       |                    +----------- 经由所有类型的边的出方向探索
       |                       |
       |                       |
       |                       +------------------ 从给定的起点、终点 VID
       |
       +--------------------------- 查找最短路径
```

更多信息，请参见 FIND PATH。

GET SUBGRAPH

用法：提取满足特定条件的图的一部分；查询子图中的点和边的属性。

场景：分析图的局部结构或特定区域时非常有用，比如提取某个人的社交网络子图，或者提取某个区域的交通网络。

说明：必须使用 `YIELD` 子句指定返回信息。

示例：

```
GET SUBGRAPH 5 STEPS FROM "player101" YIELD VERTICES AS nodes, EDGES AS relationships;
             -----+- -----+--------        ------------------------+----------------
                  |       |                                        |
                  |       |                                        |
                  |       +------- 从 "player101" 开始出发            +------------ 返回所有的点、边
                  |
                  +---------------- 获取 5 步的探索
```

更多信息，请参见 GET SUBGRAPH。

SHOW

SHOW 语句主要用于获取数据库的元数据信息，而不是用于获取存储在数据库中的实际数据内容。这类语句通常用于查询数据库的结构和配置，如查看现有的图空间、标签、边类型、索引等。

语句	语法	示例	说明
SHOW CHARSET	`SHOW CHARSET`	`SHOW CHARSET`	显示当前的字符集。
SHOW COLLATION	`SHOW COLLATION`	`SHOW COLLATION`	显示当前的排序规则。
SHOW CREATE SPACE	`SHOW CREATE SPACE <space_name>`	`SHOW CREATE SPACE basketballplayer`	显示指定图空间的创建语句。
SHOW CREATE TAG/EDGE	`SHOW CREATE {TAG <tag_name> | EDGE <edge_name>}`	`SHOW CREATE TAG player`	显示指定 Tag/Edge type 的基本信息。
SHOW HOSTS	`SHOW HOSTS [GRAPH | STORAGE | META]`	`SHOW HOSTS` `SHOW HOSTS GRAPH`	显示 Graph、Storage、Meta 服务主机信息、版本信息。
SHOW INDEX STATUS	`SHOW {TAG | EDGE} INDEX STATUS`	`SHOW TAG INDEX STATUS`	重建原生索引的作业状态，以便确定重建索引是否成功。
SHOW INDEXES	`SHOW {TAG | EDGE} INDEXES`	`SHOW TAG INDEXES`	列出当前图空间内的所有 Tag 和 Edge type（包括属性）的索引。
SHOW PARTS	`SHOW PARTS [<part_id>]`	`SHOW PARTS`	显示图空间中指定分片或所有分片的信息。
SHOW ROLES	`SHOW ROLES IN <space_name>`	`SHOW ROLES in basketballplayer`	显示分配给用户的角色信息。
SHOW SNAPSHOTS	`SHOW SNAPSHOTS`	`SHOW SNAPSHOTS`	显示所有快照信息。
SHOW SPACES	`SHOW SPACES`	`SHOW SPACES`	显示现存的图空间。
SHOW STATS	`SHOW STATS`	`SHOW STATS`	显示最近 `STATS` 作业收集的图空间统计信息。
SHOW TAGS/EDGES	`SHOW TAGS | EDGES`	`SHOW TAGS`、`SHOW EDGES`	显示当前图空间内的所有 Tag/Edge type。
SHOW USERS	`SHOW USERS`	`SHOW USERS`	显示用户信息。
SHOW SESSIONS	`SHOW SESSIONS`	`SHOW SESSIONS`	显示所有会话信息。
SHOW SESSIONS	`SHOW SESSION <Session_Id>`	`SHOW SESSION 1623304491050858`	指定会话 ID 进行查看。
SHOW QUERIES	`SHOW [ALL] QUERIES`	`SHOW QUERIES`	查看当前 Session 中正在执行的查询请求信息。
SHOW META LEADER	`SHOW META LEADER`	`SHOW META LEADER`	显示当前 Meta 集群的 leader 信息。

复合查询

NebulaGraph 的查询语句可以组合使用，以实现更复杂的查询。

复合语句中如需引用子查询的结果，需要为该结果设置别名，并使用管道符 `|` 传递给下一个子查询，同时在下一个子查询中使用 `$-` 引用该结果的别名。详情参见管道符。

示例：

```
nebula> GO FROM "player100" OVER follow \
        YIELD dst(edge) AS dstid, properties($$).name AS Name | \
        GO FROM $-.dstid OVER follow YIELD dst(edge);

+-------------+
| dst(EDGE)   |
+-------------+
| "player100" |
```

```
| "player102" |
| "player125" |
| "player100" |
+-------------+
```

管道符 | 仅适用于 nGQL，不适用于 OpenCypher 语句，即不能在 MATCH 语句中使用管道符。如果需要使用 MATCH 语句进行复合查询，可以使用 WITH 子句。

示例：

```
nebula> MATCH (v:player)-->(v2:player) \
        WITH DISTINCT v2 AS v2, v2.player.age AS Age \
        ORDER BY Age \
        WHERE Age<25 \
        RETURN v2.player.name AS Name, Age;
+----------------------+-----+
| Name                 | Age |
+----------------------+-----+
| "Luka Doncic"        | 20  |
| "Ben Simmons"        | 22  |
| "Kristaps Porzingis" | 23  |
+----------------------+-----+
```

更多信息

- nGQL 简明教程
- nGQL 语句汇总

最后更新: 2025年8月4日

4.5.2 MATCH

`MATCH` 语句提供基于模式（Pattern）匹配的搜索功能，其通过定义一个或多个模式，允许在 NebulaGraph 中查找与模式匹配的数据。在检索到匹配的数据后，用户可以使用 RETURN 子句将其作为结果返回。

在本文中，我们将使用名为 basketballplayer 的测试数据集来演示 MATCH 语句的使用。

语法

`MATCH` 语句的语法相较于其他查询语句（如 `GO` 和 `LOOKUP`）更具灵活性。在进行查询时，`MATCH` 语句使用的路径类型是 `trail`，这意味着点可以重复出现，但边不能重复。

`MATCH` 语法的基本结构如下：

```
MATCH <pattern> [<clause_1>]  RETURN <output>  [<clause_2>];
```

- `pattern`： `MATCH` 语句支持匹配一个或多个模式，多个模式之间用英文逗号（,）分隔。例如 `(a)-[]->(b),(c)-[]->(d)`。Pattern 的详细说明请参见模式。
- `clause_1`： 支持 `WHERE`、`WITH`、`UNWIND`、`OPTIONAL MATCH` 子句，也可以使用 `MATCH` 作为子句。
- `output`： 定义需要返回输出结果的列表名称。可以使用 `AS` 设置列表的别名。
- `clause_2`： 支持 `ORDER BY`、`LIMIT` 子句。

历史版本兼容性

- 从 3.5.0 版本开始，`MATCH` 语句支持全表扫描，即在不使用任何索引或者过滤条件的情况下可遍历图中点或边。在此之前的版本中，MATCH 语句在某些情况下需要索引才能执行查询或者需要使用 `LIMIT` 限制输出结果数量。
- 从 3.0.0 版本开始，为了区别不同 Tag 的属性，返回属性时必须额外指定 Tag 名称。即从 `RETURN <变量名>.<属性名>` 改为 `RETURN <变量名>.<Tag名>.<属性名>`。

使用说明

- 尽量避免执行全表扫描，因为这可能导致查询性能下降；并且如果在进行全表扫描时内存不足，可能会导致查询失败，系统会提示报错。建议使用具有过滤条件或指定 Tag、边类型的查询，例如 `MATCH (v:player) RETURN v.player.name AS Name` 语句中的 `v:player` 和 `v.player.name`。
- 可为 Tag、Edge type 或 Tag、Edge type 的某个属性创建索引，以提高查询性能。例如，用户可以为 `player` Tag 创建索引，或者为 `player` Tag 的 `name` 属性创建索引。有关索引的使用及注意事项，请参见使用索引必读。
- 目前 MATCH 语句无法查询到悬挂边。

使用模式

匹配点

用户可以在一对括号中使用自定义变量来表示模式中的点。例如 `(v)`。

```
# 匹配全图空间中的任意 3 个点
nebula> MATCH (v) \
        RETURN v \
        LIMIT 3;
+-----------------------------------------------------------+
| v                                                         |
+-----------------------------------------------------------+
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) |
| ("player106" :player{age: 25, name: "Kyle Anderson"})     |
| ("player115" :player{age: 40, name: "Kobe Bryant"})       |
+-----------------------------------------------------------+
```

匹配 TAG

> **历史版本兼容性**
>
> 在 NebulaGraph 3.0.0 之前，匹配 Tag 的前提是 Tag 本身有索引或者 Tag 的某个属性有索引，否则，用户无法基于该 Tag 执行 `MATCH` 语句。从 NebulaGraph 3.0.0 开始，匹配 Tag 可以不创建索引，但需要使用 `LIMIT` 限制输出结果数量。从 NebulaGraph 3.5.0 开始，`MATCH` 语句支持全表扫描，无需为 Tag 或 Tag 的某个属性创建索引，或者使用 `LIMIT` 限制输出结果数量，即可执行 `MATCH` 语句。

用户可以在点的右侧用 `:<tag_name>` 表示模式中的 Tag。

```
# 匹配全图空间中所有 Tag 为 player 的点
nebula> MATCH (v:player) \
        RETURN v;
+-----------------------------------------------------------+
| v                                                         |
+-----------------------------------------------------------+
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) |
| ("player106" :player{age: 25, name: "Kyle Anderson"})     |
| ("player115" :player{age: 40, name: "Kobe Bryant"})       |
...
```

需要匹配拥有多个 Tag 的点，可以用英文冒号（:）。

```
# 创建包含 name 属性和 age 属性的 Tag actor。
nebula> CREATE TAG actor (name string, age int);
# 插入 Tag actor 到点 player100。
nebula> INSERT VERTEX actor(name, age) VALUES "player100":("Tim Duncan", 42);
# 匹配 Tag 为 player 和 actor 的点。
nebula> MATCH (v:player:actor) \
        RETURN v;
+----------------------------------------------------------------------------------------+
| v                                                                                      |
+----------------------------------------------------------------------------------------+
| ("player100" :actor{age: 42, name: "Tim Duncan"} :player{age: 42, name: "Tim Duncan"}) |
+----------------------------------------------------------------------------------------+
```

匹配点的属性

用户可以在 Tag 的右侧用 `{<prop_name>: <prop_value>}` 表示模式中点的属性。

```
# 使用属性 name 搜索匹配的点。
nebula> MATCH (v:player{name:"Tim Duncan"}) \
        RETURN v;
+----------------------------------------------------+
| v                                                  |
+----------------------------------------------------+
| ("player100" :player{age: 42, name: "Tim Duncan"}) |
+----------------------------------------------------+
```

使用 `WHERE` 子句也可以实现相同的操作：

```
# 查找类型为 player，名字为 Tim Duncan 的点。
nebula> MATCH (v:player) \
        WHERE v.player.name == "Tim Duncan" \
        RETURN v;
+----------------------------------------------------+
| v                                                  |
+----------------------------------------------------+
| ("player100" :player{age: 42, name: "Tim Duncan"}) |
+----------------------------------------------------+
```

> **openCypher 兼容性**
>
> 在 openCypher 9 中，`=` 是相等运算符，在 nGQL 中，`==` 是相等运算符，`=` 是赋值运算符。

使用 `WHERE` 子句直接匹配点的属性。

```
# 匹配属性中值等于 Tim Duncan 的点。
nebula> MATCH (v) \
        WITH v, properties(v) as props, keys(properties(v)) as kk \
        WHERE [i in kk where props[i] == "Tim Duncan"] \
        RETURN v;
+----------------------------------------------------+
| v                                                  |
```

```
+----------------------------------------------------+
| ("player100" :player{age: 42, name: "Tim Duncan"}) |
+----------------------------------------------------+

# 匹配 name 属性值存在于 names 列表内的起点，并返回起点和终点的数据。
nebula> WITH ['Tim Duncan', 'Yao Ming'] AS names \
        MATCH (v1:player)-->(v2:player) \
        WHERE v1.player.name in names \
        RETURN v1, v2;
+----------------------------------------------------+----------------------------------------------------------+
| v1                                                 | v2                                                       |
+----------------------------------------------------+----------------------------------------------------------+
| ("player133" :player{age: 38, name: "Yao Ming"})   | ("player114" :player{age: 39, name: "Tracy McGrady"})    |
| ("player133" :player{age: 38, name: "Yao Ming"})   | ("player144" :player{age: 47, name: "Shaquille O'Neal"}) |
| ("player100" :player{age: 42, name: "Tim Duncan"}) | ("player101" :player{age: 36, name: "Tony Parker"})      |
| ("player100" :player{age: 42, name: "Tim Duncan"}) | ("player125" :player{age: 41, name: "Manu Ginobili"})    |
+----------------------------------------------------+----------------------------------------------------------+
```

匹配点 ID

用户可以使用点 ID 去匹配点。 `id()` 函数可以检索点的 ID。

```
# 查找 ID 为 “player101” 的点。（注：ID 全局唯一）。
nebula> MATCH (v) \
        WHERE id(v) == 'player101' \
        RETURN v;
+-----------------------------------------------------+
| v                                                   |
+-----------------------------------------------------+
| ("player101" :player{age: 36, name: "Tony Parker"}) |
+-----------------------------------------------------+
```

要匹配多个点的 ID，可以用 `WHERE id(v) IN [vid_list]` 或者 `WHERE id(v) IN {vid_list}` 。

```
# 查找与 `Tim Duncan` 直接相连的点，并且这些点的 ID 必须是 `player101` 或 `player102`。
nebula> MATCH (v:player { name: 'Tim Duncan' })--(v2) \
        WHERE id(v2) IN ["player101", "player102"] \
        RETURN v2;
+-----------------------------------------------------------+
| v2                                                        |
+-----------------------------------------------------------+
| ("player101" :player{age: 36, name: "Tony Parker"})       |
| ("player101" :player{age: 36, name: "Tony Parker"})       |
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) |
+-----------------------------------------------------------+

# 查找 ID 为 player100 和 player101 的点，并返回 name 属性。
nebula> MATCH (v) WHERE id(v) IN {"player100", "player101"} \
        RETURN v.player.name AS name;
+---------------+
| name          |
+---------------+
| "Tony Parker" |
| "Tim Duncan"  |
+---------------+
```

匹配连接的点

用户可以使用 `--` 符号表示两个方向的边，并匹配这些边连接的点。

> **历史版本兼容性**
>
> 在 nGQL 1.x 中， `--` 符号用于行内注释，从 nGQL 2.x 起， `--` 符号表示出边或入边，不再用于注释。

```
# name 属性值为 Tim Duncan 的点为 v，与 v 相连接的点为 v2，查找 v2 并返回其 name 属性值。
nebula> MATCH (v:player{name:"Tim Duncan"})--(v2:player) \
        RETURN v2.player.name AS Name;
+---------------------+
| Name                |
+---------------------+
| "Manu Ginobili"     |
| "Manu Ginobili"     |
| "Dejounte Murray"   |
...
```

用户可以在 `--` 符号上增加 `<` 或 `>` 符号指定边的方向。

```
# `-->` 表示边从 v 开始，指向 v2。对于点 v 来说是出边，对于点 v2 来说是入边。
nebula> MATCH (v:player{name:"Tim Duncan"})-->(v2:player) \
        RETURN v2.player.name AS Name;
+-----------------+
```

```
| Name           |
+----------------+
| "Tony Parker"  |
| "Manu Ginobili" |
+----------------+
```

如果需要判断目标点，可以使用 `CASE` 表达式。

```
# name 属性值为 Tim Duncan 的点为 v，与 v 相连接的点为 v2，查找 v2 并判断，如果 v2.team.name 的值不为空则返回 v2.team.name 属性值，如果 v2.player.name 的值不为空则返回 v2.player.name 属性值。
nebula> MATCH (v:player{name:"Tim Duncan"})--(v2) \
        RETURN \
        CASE WHEN v2.team.name IS NOT NULL \
        THEN v2.team.name  \
        WHEN v2.player.name IS NOT NULL \
        THEN v2.player.name END AS Name;

+-------------------+
| Name              |
+-------------------+
| "Manu Ginobili"   |
| "Manu Ginobili"   |
| "Spurs"           |
| "Dejounte Murray" |
...
```

如果需要扩展模式，可以增加更多点和边。

```
# name 属性值为 Tim Duncan 的点为 v，指向点 v2，点 v3 也指向点 v2，返回 v3 的 name 属性值。
nebula> MATCH (v:player{name:"Tim Duncan"})-->(v2)<--(v3) \
        RETURN v3.player.name AS Name;
+-------------------+
| Name              |
+-------------------+
| "Dejounte Murray"   |
| "LaMarcus Aldridge" |
| "Marco Belinelli"   |
...
```

如果不需要引用点，可以省略括号中表示点的变量。

```
# 查找 name 属性值为 Tim Duncan 的点 v， 点 v3 与 v 指向同一个点，返回 v3 的 name 属性值。
nebula> MATCH (v:player{name:"Tim Duncan"})-->()<--(v3) \
        RETURN v3.player.name AS Name;
+-------------------+
| Name              |
+-------------------+
| "Dejounte Murray"   |
| "LaMarcus Aldridge" |
| "Marco Belinelli"   |
...
```

匹配路径

连接起来的点和边构成了路径。用户可以使用自定义变量命名路径。

```
# 设置路径 p，其模式为 name 属性值为 Tim Duncan 的点 v 指向相邻的点 v2。返回所有符合条件的路径。
nebula> MATCH p=(v:player{name:"Tim Duncan"})-->(v2) \
        RETURN p;
+--------------------------------------------------------------------------------------------------------------------------------------+
| p                                                                                                                                    |
+--------------------------------------------------------------------------------------------------------------------------------------+
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:serve@0 {end_year: 2016, start_year: 1997}]->("team204" :team{name: "Spurs"})> |
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})>   |
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player125" :player{age: 41, name: "Manu Ginobili"})> |
+--------------------------------------------------------------------------------------------------------------------------------------+
```

openCypher 兼容性

在 nGQL 中，`@` 符号表示边的 rank，在 openCypher 中，没有 rank 概念。

匹配边

```
# 匹配对应边并返回 3 条数据。
nebula> MATCH ()<-[e]-() \
        RETURN e \
        LIMIT 3;
+----------------------------------------------------+
| e                                                  |
+----------------------------------------------------+
```

```
| [:follow "player101"->"player102" @0 {degree: 90}] |
| [:follow "player103"->"player102" @0 {degree: 70}] |
| [:follow "player135"->"player102" @0 {degree: 80}] |
+----------------------------------------------------+
```

匹配 EDGE TYPE

和点一样，用户可以用 `:<edge_type>` 表示模式中的 Edge type，例如 `-[e:follow]-`。

> **历史版本兼容性**
>
> 在 NebulaGraph 3.0.0 之前，匹配 Edge Type 的前提是 Edge Type 本身有对应属性的索引，否则，用户无法基于 Edge Type 执行 `MATCH` 语句。从 NebulaGraph 3.0.0 开始，匹配 Edge Type 可以不创建索引，但需要使用 `LIMIT` 限制输出结果数量，并且必须指定边的方向。从 NebulaGraph 3.5.0 开始，无需为 Edge Type 创建索引或者使用 `LIMIT` 限制输出结果数量，即可使用 `MATCH` 语句匹配边。

```
# 匹配所有 edge type 为 follow 的边。
nebula> MATCH ()-[e:follow]->() \
        RETURN e;
+----------------------------------------------------+
| e                                                  |
+----------------------------------------------------+
| [:follow "player102"->"player100" @0 {degree: 75}] |
| [:follow "player102"->"player101" @0 {degree: 75}] |
| [:follow "player129"->"player116" @0 {degree: 90}] |
...
```

匹配边的属性

用户可以用 `{<prop_name>: <prop_value>}` 表示模式中 Edge type 的属性，例如 `[e:follow{likeness:95}]`。

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e:follow{degree:95}]->(v2) \
        RETURN e;
+--------------------------------------------------------+
| e                                                      |
+--------------------------------------------------------+
| [:follow "player100"->"player101" @0 {degree: 95}]     |
| [:follow "player100"->"player125" @0 {degree: 95}]     |
+--------------------------------------------------------+
```

使用 `WHERE` 子句直接匹配边的属性。

```
nebula> MATCH ()-[e]->() \
        WITH e, properties(e) as props, keys(properties(e)) as kk \
        WHERE [i in kk where props[i] == 90] \
        RETURN e;
+----------------------------------------------------+
| e                                                  |
+----------------------------------------------------+
| [:follow "player125"->"player100" @0 {degree: 90}] |
| [:follow "player140"->"player114" @0 {degree: 90}] |
| [:follow "player133"->"player144" @0 {degree: 90}] |
| [:follow "player133"->"player114" @0 {degree: 90}] |
...
+----------------------------------------------------+
```

匹配多个 EDGE TYPE

使用 `|` 可以匹配多个 Edge type，例如 `[e:follow|:serve]`。第一个 Edge type 前的英文冒号（:）不可省略，后续 Edge type 前的英文冒号可以省略，例如 `[e:follow|serve]`。

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e:follow|:serve]->(v2) \
        RETURN e;
+---------------------------------------------------------------------------+
| e                                                                         |
+---------------------------------------------------------------------------+
| [:follow "player100"->"player101" @0 {degree: 95}]                        |
| [:follow "player100"->"player125" @0 {degree: 95}]                        |
| [:serve "player100"->"team204" @0 {end_year: 2016, start_year: 1997}]     |
+---------------------------------------------------------------------------+
```

匹配多条边

用户可以扩展模式，匹配路径中的多条边。

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[]->(v2)<-[e:serve]-(v3) \
        RETURN v2, v3;
+---------------------------------+-----------------------------------------------------------+
```

```
| v2                             | v3                                                       |
+--------------------------------+----------------------------------------------------------+
| ("team204" :team{name: "Spurs"}) | ("player104" :player{age: 32, name: "Marco Belinelli"})   |
| ("team204" :team{name: "Spurs"}) | ("player101" :player{age: 36, name: "Tony Parker"})       |
| ("team204" :team{name: "Spurs"}) | ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) |
...
```

匹配定长路径

用户可以在模式中使用 `:<edge_type>*<hop>` 匹配定长路径。 `hop` 必须是一个非负整数。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*2]->(v2) \
        RETURN DISTINCT v2 AS Friends;
+-----------------------------------------------------------+
| Friends                                                   |
+-----------------------------------------------------------+
| ("player100" :player{age: 42, name: "Tim Duncan"})        |
| ("player125" :player{age: 41, name: "Manu Ginobili"})     |
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) |
+-----------------------------------------------------------+
```

如果 `hop` 为 0，模式会匹配路径上的起始点。

```
nebula> MATCH (v:player{name:"Tim Duncan"}) -[*0]-> (v2) \
        RETURN v2;
+----------------------------------------------------+
| v2                                                 |
+----------------------------------------------------+
| ("player100" :player{age: 42, name: "Tim Duncan"}) |
+----------------------------------------------------+
```

Note

在对匹配的多跳边进行过滤时，如对 `-[e:follow*2]->` 中的 `e` 进行过滤，此时的 `e` 不再是单条边的数据类型，而是一个包含多条边的列表，例如：

以下语句可以运行但是没有返回数据，因为 `e` 是一个列表，没有 `.degree` 的属性。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*2]->(v2) \
        WHERE e.degree > 1 \
        RETURN DISTINCT v2 AS Friends;
```

这是正确的表达：

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*2]->(v2) \
        WHERE ALL(e_ in e WHERE e_.degree > 0) \
        RETURN DISTINCT v2 AS Friends;
```

进一步，这是表达对多跳边的第一跳的边属性过滤的表达：

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*2]->(v2) \
        WHERE e[0].degree > 98 \
        RETURN DISTINCT v2 AS Friends;
```

匹配变长路径

用户可以在模式中使用 `:<edge_type>*[minHop..maxHop]` 匹配变长路径。

Caution

如果未设置 `maxHop` 可能会导致 graph 服务 OOM，请谨慎执行该命令。

参数	说明
`minHop`	可选项。表示路径的最小长度。 `minHop` 必须是一个非负整数，默认值为 1。
`maxHop`	可选项。表示路径的最大长度。 `maxHop` 必须是一个非负整数，默认值为无穷大。

如果未指定 `minHop` 和 `maxHop`，仅设置了 `:<edge_type>*`，则二者都应用默认值，即 `minHop` 为 1， `maxHop` 为无穷大。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*]->(v2) \
        RETURN v2 AS Friends;
```

```
+-----------------------------------------------------------+
| Friends                                                   |
+-----------------------------------------------------------+
| ("player125" :player{age: 41, name: "Manu Ginobili"})     |
| ("player101" :player{age: 36, name: "Tony Parker"})       |
...

nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*1..3]->(v2) \
        RETURN v2 AS Friends;
+-----------------------------------------------------------+
| Friends                                                   |
+-----------------------------------------------------------+
| ("player101" :player{age: 36, name: "Tony Parker"})       |
| ("player125" :player{age: 41, name: "Manu Ginobili"})     |
| ("player100" :player{age: 42, name: "Tim Duncan"})        |
...

nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*1..]->(v2) \
        RETURN v2 AS Friends;
+-----------------------------------------------------------+
| Friends                                                   |
+-----------------------------------------------------------+
| ("player125" :player{age: 41, name: "Manu Ginobili"})     |
| ("player101" :player{age: 36, name: "Tony Parker"})       |
| ("player100" :player{age: 42, name: "Tim Duncan"})        |
...
```

用户可以使用 `DISTINCT` 关键字聚合重复结果。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*1..3]->(v2:player) \
        RETURN DISTINCT v2 AS Friends, count(v2);
+-----------------------------------------------------------+-----------+
| Friends                                                   | count(v2) |
+-----------------------------------------------------------+-----------+
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) | 1         |
| ("player100" :player{age: 42, name: "Tim Duncan"})        | 4         |
| ("player101" :player{age: 36, name: "Tony Parker"})       | 3         |
| ("player125" :player{age: 41, name: "Manu Ginobili"})     | 3         |
+-----------------------------------------------------------+-----------+
```

如果 `minHop` 为 `0`，模式会匹配路径上的起始点。例如，与上个示例相比，下面的示例设置 `minHop` 为 `0`。此时，因为表示 `"Tim Duncan"` 的点是路径的起始点，所以它在结果集中的计数为 5，比在上个示例的结果中多计一次。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow*0..3]->(v2:player) \
        RETURN DISTINCT v2 AS Friends, count(v2);
+-----------------------------------------------------------+-----------+
| Friends                                                   | count(v2) |
+-----------------------------------------------------------+-----------+
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) | 1         |
| ("player100" :player{age: 42, name: "Tim Duncan"})        | 5         |
| ("player125" :player{age: 41, name: "Manu Ginobili"})     | 3         |
| ("player101" :player{age: 36, name: "Tony Parker"})       | 3         |
+-----------------------------------------------------------+-----------+
```

Note

当在模式中使用变量 `e` 匹配定长或者变长路径时，例如 `-[e:follow*0..3]->`，不支持在其他模式中引用 `e`。例如，不支持以下语句：

```
nebula> MATCH (v:player)-[e:like*1..3]->(n) \
        WHERE (n)-[e*1..4]->(:player) \
        RETURN v;
```

匹配多个 EDGE TYPE 的变长路径

用户可以在变长或定长模式中指定多个 Edge type。`hop`、`minHop` 和 `maxHop` 对所有 Edge type 都生效。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e:follow|serve*2]->(v2) \
        RETURN DISTINCT v2;
+-----------------------------------------------------------+
| v2                                                        |
+-----------------------------------------------------------+
| ("team204" :team{name: "Spurs"})                          |
| ("player100" :player{age: 42, name: "Tim Duncan"})        |
| ("team215" :team{name: "Hornets"})                        |
| ("player125" :player{age: 41, name: "Manu Ginobili"})     |
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) |
+-----------------------------------------------------------+
```

匹配多个模式

用户可以用英文逗号（,）分隔多个模式。

```
nebula> CREATE TAG INDEX IF NOT EXISTS team_index ON team(name(20));
nebula> REBUILD TAG INDEX team_index;
nebula> MATCH (v1:player{name:"Tim Duncan"}), (v2:team{name:"Spurs"}) \
        RETURN v1,v2;
+----------------------------------------------------+-----------------------------------+
| v1                                                 | v2                                |
+----------------------------------------------------+-----------------------------------+
| ("player100" :player{age: 42, name: "Tim Duncan"}) | ("team204" :team{name: "Spurs"}) |
+----------------------------------------------------+-----------------------------------+
```

匹配最短路径

用户可以使用 `allShortestPaths` 返回起始点到目标点的所有最短路径。

```
nebula> MATCH p = allShortestPaths((a:player{name:"Tim Duncan"})-[e*..5]-(b:player{name:"Tony Parker"})) \
        RETURN p;
+------------------------------------------------------------------------------------------------------------------------------------+
| p                                                                                                                                  |
+------------------------------------------------------------------------------------------------------------------------------------+
| <("player100" :player{age: 42, name: "Tim Duncan"})<-[:follow@0 {degree: 95}]-("player101" :player{age: 36, name: "Tony Parker"})> |
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})> |
+------------------------------------------------------------------------------------------------------------------------------------+
```

用户可以使用 `shortestPath` 返回起始点到目标点的任意一条最短路径。

```
nebula> MATCH p = shortestPath((a:player{name:"Tim Duncan"})-[e*..5]-(b:player{name:"Tony Parker"})) \
        RETURN p;
+------------------------------------------------------------------------------------------------------------------------------------+
| p                                                                                                                                  |
+------------------------------------------------------------------------------------------------------------------------------------+
| <("player100" :player{age: 42, name: "Tim Duncan"})<-[:follow@0 {degree: 95}]-("player101" :player{age: 36, name: "Tony Parker"})> |
+------------------------------------------------------------------------------------------------------------------------------------+
```

多MATCH检索

不同的模式有不同的筛选条件时，可以使用多 `MATCH`，会返回模式完全匹配的行。

```
nebula> MATCH (m)-[]->(n) WHERE id(m)=="player100" \
        MATCH (n)-[]->(l) WHERE id(n)=="player125" \
        RETURN id(m),id(n),id(l);
+-------------+-------------+-------------+
| id(m)       | id(n)       | id(l)       |
+-------------+-------------+-------------+
| "player100" | "player125" | "team204"   |
| "player100" | "player125" | "player100" |
+-------------+-------------+-------------+
```

OPTIONAL MATCH检索

参见OPTIONAL MATCH。

> **Caution**
>
> NebulaGraph master 中 `MATCH` 语句的性能和资源占用得到了优化.但对性能要求较高时，仍建议使用 `GO`, `LOOKUP`, `|` 和 `FETCH` 等来替代 `MATCH`。

最后更新: 2025年8月4日

4.5.3 OPTIONAL MATCH

Caution

目前 `OPTIONAL MATCH` 为 Beta 功能，后续可能会有一定优化调整。

`OPTIONAL MATCH` 通常与 `MATCH` 语句一起使用，作为 `MATCH` 语句的可选项去匹配命中的模式，如果没有命中对应的模式，对应的列返回 `NULL`。

openCypher 兼容性

本文操作仅适用于 nGQL 中的 openCypher 方式。

使用限制

`OPTIONAL MATCH` 子句中暂不支持使用 `WHERE` 子句。

示例

`MATCH` 语句中使用 `OPTIONAL MATCH` 的示例如下：

```
# 找出特定节点 m（ID为 "player100" 的节点）直接连接的节点，以及这些节点 n 直连的下一层节点 l（OPTIONAL MATCH (n)-[]->(l)）， 如果没有找到这样的节点 l，查询也会继续， l 的返回值是 null。
nebula> MATCH (m)-[]->(n) WHERE id(m)=="player100" \
        OPTIONAL MATCH (n)-[]->(l) \
        RETURN id(m),id(n),id(l);
+-------------+-------------+-------------+
| id(m)       | id(n)       | id(l)       |
+-------------+-------------+-------------+
| "player100" | "team204"   | __NULL__    |
| "player100" | "player101" | "team204"   |
| "player100" | "player101" | "team215"   |
| "player100" | "player101" | "player100" |
| "player100" | "player101" | "player102" |
| "player100" | "player101" | "player125" |
| "player100" | "player125" | "team204"   |
| "player100" | "player125" | "player100" |
+-------------+-------------+-------------+
```

而使用多 `MATCH`，不使用 `OPTIONAL MATCH` 时，会返回模式完全匹配的行。示例如下：

```
# 找出特定节点 m（ID为 "player100" 的节点）直接连接的节点，以及这些节点 n 直连的下一层节点 l，节点 l 必须存在。
nebula> MATCH (m)-[]->(n) WHERE id(m)=="player100" \
        MATCH (n)-[]->(l) \
        RETURN id(m),id(n),id(l);
+-------------+-------------+-------------+
| id(m)       | id(n)       | id(l)       |
+-------------+-------------+-------------+
| "player100" | "player101" | "team204"   |
| "player100" | "player101" | "team215"   |
| "player100" | "player101" | "player100" |
| "player100" | "player101" | "player102" |
| "player100" | "player101" | "player125" |
| "player100" | "player125" | "team204"   |
| "player100" | "player125" | "player100" |
+-------------+-------------+-------------+
```

最后更新: 2025年8月4日

4.5.4 LOOKUP

`LOOKUP` 根据索引遍历数据。用户可以使用 `LOOKUP` 实现如下功能：

- 根据 `WHERE` 子句搜索特定数据。
- 通过 Tag 列出点：检索指定 Tag 的所有点 ID。
- 通过 Edge type 列出边：检索指定 Edge type 的所有边的起始点、目的点和 rank。
- 统计包含指定 Tag 的点或属于指定 Edge type 的边的数量。

OpenCypher 兼容性

本文操作仅适用于原生 nGQL。

注意事项

- 索引会导致写性能大幅降低。请不要随意在生产环境中使用索引，除非很清楚使用索引对业务的影响。
- 通过 `Explain` 命令查看选择的索引。

> 历史版本兼容性
>
> 在 2.5.0 版本之前，如果用 `LOOKUP` 语句基于指定属性查询时该属性没有索引，系统将报错，而不会使用其它索引。

前提条件

请确保 `LOOKUP` 语句有至少一个索引可用。

如果已经存在相关的点、边或属性，必须在新创建索引后重建索引，才能使其生效。

语法

```
LOOKUP ON {<vertex_tag> | <edge_type>}
[WHERE <expression> [AND <expression> ...]]
YIELD [DISTINCT] <return_list> [AS <alias>]
[<clause>];

<return_list>
    <prop_name> [AS <col_alias>] [, <prop_name> [AS <prop_alias>] ...];
```

- `WHERE <expression>`：指定遍历的过滤条件，还可以结合布尔运算符 AND 和 OR 一起使用。详情请参见 WHERE。
- `YIELD`：定义需要返回的输出。详情请参见 `YIELD`。
- `DISTINCT`：聚合输出结果，返回去重后的结果集。
- `AS`：设置别名。
- `clause`：支持 `ORDER BY`、`LIMIT` 子句。

WHERE 语句限制

在 `LOOKUP` 语句中使用 `WHERE` 子句，不支持如下操作：

- `$-` 和 `$^` 。
- 在关系表达式中，不支持运算符两边都有字段名，例如 `tagName.prop1 > tagName.prop2` 。
- 不支持运算表达式和函数表达式中嵌套 AliasProp 表达式。
- 不支持 XOR 运算符。
- 不支持除 `STARTS WITH` 之外的字符串操作。
- 不支持过滤 `rank()` 。
- 不支持图模式.

检索点

返回 Tag 为 `player` 且 `name` 为 `Tony Parker` 的点。

```
# 创建 Tag 为 player 的复合属性索引 index_player。
nebula> CREATE TAG INDEX IF NOT EXISTS index_player ON player(name(30), age);

# 重建复合属性索引 index_player，返回任务 id。
nebula> REBUILD TAG INDEX index_player;
+------------+
| New Job Id |
+------------+
| 15         |
+------------+

# 返回所有 name 等于 Tony Parker 的点数据的 ID。
nebula> LOOKUP ON player \
        WHERE player.name == "Tony Parker" \
        YIELD id(vertex);
+---------------+
| id(VERTEX)    |
+---------------+
| "player101"   |
+---------------+

# 查询所有 name 等于 Tony Parker 的点数据，返回 name 和 age 属性值。
nebula> LOOKUP ON player \
        WHERE player.name == "Tony Parker" \
        YIELD properties(vertex).name AS name, properties(vertex).age AS age;
+---------------+-----+
| name          | age |
+---------------+-----+
| "Tony Parker" | 36  |
+---------------+-----+

# 查询所有 age 大于 45 的点数据，并返回点数据的 ID。
nebula> LOOKUP ON player \
        WHERE player.age  > 45 \
        YIELD id(vertex);
+-------------+
| id(VERTEX)  |
+-------------+
| "player144" |
| "player140" |
+-------------+

# 查询所有 name 以 B 开头，且 age 在 22 和 30 之间的点数据，并返回 name 和 age 属性值。
nebula> LOOKUP ON player \
        WHERE player.name STARTS WITH "B" \
        AND player.age IN [22,30] \
        YIELD properties(vertex).name, properties(vertex).age;
+-------------------------+------------------------+
| properties(VERTEX).name | properties(VERTEX).age |
+-------------------------+------------------------+
| "Ben Simmons"           | 22                     |
| "Blake Griffin"         | 30                     |
+-------------------------+------------------------+

# 查询所有 name 等于 Kobe Bryant 的点数据，返回点的 ID 和 name 属性值，并以此为起始点进行遍历，返回遍历结果。
nebula> LOOKUP ON player \
        WHERE player.name == "Kobe Bryant"\
        YIELD id(vertex) AS VertexID, properties(vertex).name AS name |\
        GO FROM $-.VertexID OVER serve \
        YIELD $-.name, properties(edge).start_year, properties(edge).end_year, properties($$).name;
+---------------+-----------------------------+---------------------------+---------------------+
| $-.name       | properties(EDGE).start_year | properties(EDGE).end_year | properties($$).name |
+---------------+-----------------------------+---------------------------+---------------------+
| "Kobe Bryant" | 1996                        | 2016                      | "Lakers"            |
+---------------+-----------------------------+---------------------------+---------------------+
```

检索边

返回 Edge type 为 `follow` 且 `degree` 为 `90` 的边。

```
# 创建 Edge type 为 follow，属性 degree 的索引 index_follow。
nebula> CREATE EDGE INDEX IF NOT EXISTS index_follow ON follow(degree);

# 重建属性索引 index_follow，返回任务 id。
nebula> REBUILD EDGE INDEX index_follow;
+------------+
| New Job Id |
+------------+
| 62         |
+------------+

# 查询并返回所有 degree 等于 90 的边。
nebula> LOOKUP ON follow \
        WHERE follow.degree == 90 YIELD edge AS e;
+----------------------------------------------------+
| e                                                  |
+----------------------------------------------------+
| [:follow "player109"->"player125" @0 {degree: 90}] |
| [:follow "player118"->"player120" @0 {degree: 90}] |
| [:follow "player118"->"player131" @0 {degree: 90}] |
...

# 查询所有 degree 等于 90 的边，并返回 degree 属性值。
nebula> LOOKUP ON follow \
        WHERE follow.degree == 90 \
        YIELD properties(edge).degree;
+-------------------------+
| properties(EDGE).degree |
+-------------------------+
| 90                      |
| 90                      |
...

# 根据 degree 属性值升序排列，并返回前 10 条 degree 属性值。
nebula> LOOKUP ON follow \
        YIELD properties(edge).degree as degree \
        | ORDER BY $-.degree \
        | LIMIT 10;
+--------+
| degree |
+--------+
| -1     |
| -1     |
| 9      |
| 10     |
| 13     |
| 50     |
| 55     |
| 60     |
| 70     |
| 70     |
+--------+

# 查询所有 degree 等于 60 的边，返回边的目的点 ID 和边的 degree 属性值，并为此起始点进行遍历，返回遍历结果。
nebula> LOOKUP ON follow \
        WHERE follow.degree == 60 \
        YIELD dst(edge) AS DstVID, properties(edge).degree AS Degree |\
        GO FROM $-.DstVID OVER serve \
        YIELD $-.DstVID, properties(edge).start_year, properties(edge).end_year, properties($$).name;
+-------------+-----------------------------+---------------------------+---------------------+
| $-.DstVID   | properties(EDGE).start_year | properties(EDGE).end_year | properties($$).name |
+-------------+-----------------------------+---------------------------+---------------------+
| "player105" | 2010                        | 2018                      | "Spurs"             |
| "player105" | 2009                        | 2010                      | "Cavaliers"         |
| "player105" | 2018                        | 2019                      | "Raptors"           |
+-------------+-----------------------------+---------------------------+---------------------+
```

通过 **Tag** 列出所有的对应的点/通过 **Edge type** 列出边

如果需要通过 Tag 列出所有的点，或通过 Edge type 列出边，则 Tag、Edge type 或属性上必须有至少一个索引。

例如一个 Tag `player` 有属性 `name` 和 `age`，为了遍历所有包含 Tag `player` 的点 ID，Tag `player`、属性 `name` 或属性 `age` 中必须有一个已经创建索引。

- 查找所有 Tag 为 `player` 的点 VID。

```
# 创建名为 player 的 Tag。
nebula> CREATE TAG IF NOT EXISTS player(name string,age int);

# 创建 Tag 为 player 的索引 player_index。
nebula> CREATE TAG INDEX IF NOT EXISTS player_index on player();

# 重建索引 player_index，返回任务 id。
nebula> REBUILD TAG INDEX player_index;
+------------+
| New Job Id |
+------------+
| 66         |
+------------+

# 插入两个点数据。
nebula> INSERT VERTEX player(name,age) \
        VALUES "player100":("Tim Duncan", 42), "player101":("Tony Parker", 36);

# 列出所有的 player。类似于 MATCH (n:player) RETURN id(n) /*, n */。
nebula> LOOKUP ON player YIELD id(vertex);
+-------------+
| id(VERTEX)  |
+-------------+
| "player100" |
| "player101" |
...

# 从结果中返回最前面的 4 行数据。
nebula> LOOKUP ON player YIELD id(vertex) | LIMIT 4;
+-------------+
| id(VERTEX)  |
+-------------+
| "player105" |
| "player109" |
| "player111" |
| "player118" |
+-------------+
```

- 查找 Edge type 为 `follow` 的所有边的信息。

```
# 创建名为 follow 的 Edge type。
nebula> CREATE EDGE IF NOT EXISTS follow(degree int);

# 创建 Edge type 为 follow 的索引 follow_index。
nebula> CREATE EDGE INDEX IF NOT EXISTS follow_index on follow();

# 重建索引 follow_index。
nebula> REBUILD EDGE INDEX follow_index;
+------------+
| New Job Id |
+------------+
| 88         |
+------------+

# 插入边数据。
nebula> INSERT EDGE follow(degree) \
        VALUES "player100"->"player101":(95);

# 列出所有的 follow 边。类似于 MATCH (s)-[e:follow]->(d) RETURN id(s), rank(e), id(d) /*, type(e) */。
nebula)> LOOKUP ON follow YIELD edge AS e;
+----------------------------------------------------+
| e                                                  |
+----------------------------------------------------+
| [:follow "player105"->"player100" @0 {degree: 70}]  |
| [:follow "player105"->"player116" @0 {degree: 80}]  |
| [:follow "player109"->"player100" @0 {degree: 80}]  |
...
```

统计点或边

统计 Tag 为 `player` 的点和 Edge type 为 `follow` 的边。

```
# 统计 Tag 为 player 的点总数。
nebula> LOOKUP ON player YIELD id(vertex)|\
        YIELD COUNT(*) AS Player_Number;
+---------------+
| Player_Number |
+---------------+
| 51            |
+---------------+
```

```
# 统计 Edge type 为 follow 的边总数。
nebula> LOOKUP ON follow YIELD edge AS e| \
        YIELD COUNT(*) AS Follow_Number;
+---------------+
| Follow_Number |
+---------------+
| 81            |
+---------------+
```

Note

使用 `SHOW STATS` 命令也可以统计点和边。

最后更新: 2025年8月4日

4.5.5 GO

`GO` 语句是 NebulaGraph 图数据库中用于从给定起始点开始遍历图的语句。`GO` 语句采用的路径类型是 `walk`，即遍历时点和边都可以重复。

openCypher 兼容性

本文操作仅适用于原生 nGQL。

语法

```
GO [[<M> TO] <N> {STEP|STEPS} ] FROM <vertex_list>
OVER <edge_type_list> [{REVERSELY | BIDIRECT}]
[ WHERE <conditions> ]
YIELD [DISTINCT] <return_list>
[{ SAMPLE <sample_list> | <limit_by_list_clause> }]
[| GROUP BY {<col_name> | expression> | <position>} YIELD <col_name>]
[| ORDER BY <expression> [{ASC | DESC}]]
[| LIMIT [<offset>,] <number_rows>];

<vertex_list> ::=
    <vid> [, <vid> ...]

<edge_type_list> ::=
   <edge_type> [, <edge_type> ...]
   | *
```

```
<return_list> ::=
    <col_name> [AS <col_alias>] [, <col_name> [AS <col_alias>] ...]
```

- `<N> {STEP|STEPS}`：指定跳数。如果没有指定跳数，默认值 `N` 为 `1`。如果 `N` 为 `0`，NebulaGraph 不会检索任何边。
- `M TO N {STEP|STEPS}`：遍历 `M~N` 跳的边。如果 `M` 为 `0`，输出结果和 `M` 为 `1` 相同，即 `GO 0 TO 2` 和 `GO 1 TO 2` 是相同的。
- `<vertex_list>`：用逗号分隔的点 ID 列表。
- `<edge_type_list>`：遍历的 Edge type 列表。
- `REVERSELY | BIDIRECT`：默认情况下检索的是 `<vertex_list>` 的出边（正向），`REVERSELY` 表示反向，即检索入边；`BIDIRECT` 为双向，即检索正向和反向。可通过 `YIELD` 返回 `<edge_type>._type` 字段判断方向，其正数为正向，负数为反向。
- `WHERE <conditions>`：指定遍历的过滤条件。用户可以在起始点、目的点和边使用 `WHERE` 子句，还可以结合 `AND`、`OR`、`NOT`、`XOR` 一起使用。详情参见 WHERE。

> Note
>
> - 遍历多个 Edge type 时，`WHERE` 子句有一些限制。例如不支持 `WHERE edge1.prop1 > edge2.prop2`。
> - GO 语句执行时先遍历所有的点，然后再根据过滤器条件进行过滤。

- `YIELD [DISTINCT] <return_list>`：定义需要返回的输出。`<return_list>` 建议使用 Schema 相关函数指定返回信息，当前支持 `src(edge)`、`dst(edge)`、`type(edge)` 等，暂不支持嵌套函数。详情参见 YIELD。
- `SAMPLE <sample_list>`：用于在结果集中取样。详情参见 SAMPLE。
- `<limit_by_list_clause>`：用于在遍历过程中逐步限制输出数量。详情参见 LIMIT。
- `GROUP BY`：根据指定属性的值将输出分组。详情参见 GROUP BY。分组后需要再次使用 `YIELD` 定义需要返回的输出。
- `ORDER BY`：指定输出结果的排序规则。详情参见 ORDER BY。

> Note
>
> 没有指定排序规则时，输出结果的顺序不是固定的。

- `LIMIT [<offset>,] <number_rows>]`：限制输出结果的行数。详情参见 LIMIT。

使用说明

- GO 语句中的 `WHERE` 和 `YIELD` 子句通常结合属性引用符(`$^` 和 `$$`)或函数 `properties($^)` 和 `properties($$)` 指定点的属性；使用函数 `properties(edge)` 指定边的属性。用法参见属性引用符和 Schema 相关函数。
- GO 复合语句中如需引用子查询的结果，需要为该结果设置别名，并使用管道符 `|` 传递给下一个子查询，同时在下一个子查询中使用 `$-` 引用该结果的别名。详情参见管道符。
- 当查询属性没有值时，返回结果显示 `NULL`。

场景及示例

查询起始点的直接邻居点

场景：查询某个点的直接相邻点，例如查询一个人所属队伍。

```
# 返回 player102 所属队伍。
nebula> GO FROM "player102" OVER serve YIELD dst(edge);
+-----------+
| dst(EDGE) |
+-----------+
| "team203" |
```

```
| "team204" |
+-----------+
```

查询指定跳数内的点

场景：查询一个点在指定跳数内的所有点，例如查询一个人两跳内的朋友。

```
# 返回距离 player102 两跳的朋友。
nebula> GO 2 STEPS FROM "player102" OVER follow YIELD dst(edge);
+-------------+
| dst(EDGE)   |
+-------------+
| "player101" |
| "player125" |
| "player100" |
| "player102" |
| "player125" |
+-------------+
```

```
# 查询 player100 1~2 跳内的朋友。
nebula> GO 1 TO 2 STEPS FROM "player100" OVER follow \
        YIELD dst(edge) AS destination;
+-------------+
| destination |
+-------------+
| "player101" |
| "player125" |
...

# 该 MATCH 查询与上一个 GO 查询具有相同的语义。
nebula> MATCH (v) -[e:follow*1..2]->(v2) \
        WHERE id(v) == "player100" \
        RETURN id(v2) AS destination;
+-------------+
| destination |
+-------------+
| "player100" |
| "player102" |
...
```

添加过滤条件

场景：查询满足特定条件的点和边，例如查询起始点和目的点之间具有特定属性的边。

```
# 使用 WHERE 添加过滤条件。
nebula> GO FROM "player100", "player102" OVER serve \
        WHERE properties(edge).start_year > 1995 \
        YIELD DISTINCT properties($$).name AS team_name, properties(edge).start_year AS start_year, properties($^).name AS player_name;

+-----------------+------------+---------------------+
| team_name       | start_year | player_name         |
+-----------------+------------+---------------------+
| "Spurs"         | 1997       | "Tim Duncan"        |
| "Trail Blazers" | 2006       | "LaMarcus Aldridge" |
| "Spurs"         | 2015       | "LaMarcus Aldridge" |
+-----------------+------------+---------------------+
```

查询所有边

场景：查询起始点关联的所有边。

```
# 返回 player102 关联的所有边。
nebula> GO FROM "player102" OVER * BIDIRECT YIELD edge AS e;
+-----------------------------------------------------------------------+
| e                                                                     |
+-----------------------------------------------------------------------+
| [:follow "player101"->"player102" @0 {degree: 90}]                    |
| [:follow "player103"->"player102" @0 {degree: 70}]                    |
| [:follow "player135"->"player102" @0 {degree: 80}]                    |
| [:follow "player102"->"player100" @0 {degree: 75}]                    |
| [:follow "player102"->"player101" @0 {degree: 75}]                    |
| [:serve "player102"->"team203" @0 {end_year: 2015, start_year: 2006}] |
| [:serve "player102"->"team204" @0 {end_year: 2019, start_year: 2015}] |
+-----------------------------------------------------------------------+
```

查询多个 **EDGE TYPE**

场景：查询起始点关联的多个边类型可以通过设置多个Edge Type实现，也可以通过设置 `*` 关联所有的边类型。

```
# 遍历多个 Edge type。
nebula> GO FROM "player100" OVER follow, serve \
        YIELD properties(edge).degree, properties(edge).start_year;
+-----------------------+---------------------------+
| properties(EDGE).degree | properties(EDGE).start_year |
```

```
+------------------------+----------------------------+
| 95                     | __NULL__                   |
| 95                     | __NULL__                   |
| __NULL__               | 1997                       |
+------------------------+----------------------------+
```

查询入边方向的点

```
# 返回关注 player100 的邻居点。
nebula> GO FROM "player100" OVER follow REVERSELY \
        YIELD src(edge) AS destination;
+-------------+
| destination |
+-------------+
| "player101" |
| "player102" |
...

# 该 MATCH 查询与上一个 GO 查询具有相同的语义。
nebula> MATCH (v)<-[e:follow]- (v2) WHERE id(v) == 'player100' \
        RETURN id(v2) AS destination;
+-------------+
| destination |
+-------------+
| "player101" |
| "player102" |
...
```

子查询作为起始点

场景：使用子查询的结果作为图遍历的起始点。

```
# 查询 player100 的朋友和朋友所属队伍。
nebula> GO FROM "player100" OVER follow REVERSELY \
        YIELD src(edge) AS id | \
        GO FROM $-.id OVER serve \
        WHERE properties($^).age > 20 \
        YIELD properties($^).name AS FriendOf, properties($$).name AS Team;
+---------------------+-----------------+
| FriendOf            | Team            |
+---------------------+-----------------+
| "Boris Diaw"        | "Spurs"         |
| "Boris Diaw"        | "Jazz"          |
| "Boris Diaw"        | "Suns"          |
...

# 该 MATCH 查询与上一个 GO 查询具有相同的语义。
nebula> MATCH (v)<-[e:follow]- (v2)-[e2:serve]->(v3)  \
        WHERE id(v) == 'player100' \
        RETURN v2.player.name AS FriendOf, v3.team.name AS Team;
+---------------------+-----------------+
| FriendOf            | Team            |
+---------------------+-----------------+
| "Boris Diaw"        | "Spurs"         |
| "Boris Diaw"        | "Jazz"          |
| "Boris Diaw"        | "Suns"          |
...
```

使用 **GROUP BY** 分组

场景：使用 `GROUP BY` 分组，然后使用 `YIELD` 返回分组后的结果。

```
# 根据年龄分组。
nebula> GO 2 STEPS FROM "player100" OVER follow \
        YIELD src(edge) AS src, dst(edge) AS dst, properties($$).age AS age \
        | GROUP BY $-.dst \
        YIELD $-.dst AS dst, collect_set($-.src) AS src, collect($-.age) AS age;
+-------------+----------------------------+----------+
| dst         | src                        | age      |
+-------------+----------------------------+----------+
| "player125" | {"player101"}              | [41]     |
| "player100" | {"player125", "player101"} | [42, 42] |
| "player102" | {"player101"}              | [33]     |
+-------------+----------------------------+----------+
```

使用 **ORDER BY** 和 **LIMIT** 排序和限制输出结果

```
# 分组并限制输出结果的行数。
nebula> $a = GO FROM "player100" OVER follow YIELD src(edge) AS src, dst(edge) AS dst; \
        GO 2 STEPS FROM $a.dst OVER follow \
        YIELD $a.src AS src, $a.dst, src(edge), dst(edge) \
        | ORDER BY $-.src | OFFSET 1 LIMIT 2;
+-------------+-------------+-------------+-------------+
| src         | $a.dst      | src(EDGE)   | dst(EDGE)   |
+-------------+-------------+-------------+-------------+
| "player100" | "player101" | "player100" | "player101" |
```

```
| "player100" | "player125" | "player100" | "player125" |
+-------------+-------------+-------------+-------------+
```

其他用法

```
# 在多个边上通过 IS NOT EMPTY 进行判断。
nebula> GO FROM "player100" OVER follow WHERE properties($$).name IS NOT EMPTY YIELD dst(edge);
+-------------+
| dst(EDGE)   |
+-------------+
| "player125" |
| "player101" |
+-------------+
```

最后更新: 2025年8月4日

4.5.6 FETCH

`FETCH` 可以获取指定点或边的属性值。

openCypher 兼容性

本文操作仅适用于原生 nGQL。

获取点的属性值

语法

```
FETCH PROP ON {<tag_name>[, tag_name ...] | *}
<vid> [, vid ...]
YIELD [DISTINCT] <return_list> [AS <alias>];
```

参数	说明
`tag_name`	Tag 名称。
`*`	表示当前图空间中的所有 Tag。
`vid`	点 ID。
`YIELD`	定义需要返回的输出。详情请参见 `YIELD`。
`AS`	设置别名。

基于 TAG 获取点的属性值

在 `FETCH` 语句中指定 Tag 获取对应点的属性值。

```
# 获取 Tag 为 player，且 ID 为 player100 的点数据的属性值。
nebula> FETCH PROP ON player "player100" YIELD properties(vertex);
+-------------------------------+
| properties(VERTEX)            |
+-------------------------------+
| {age: 42, name: "Tim Duncan"} |
+-------------------------------+
```

获取点的指定属性值

使用 `YIELD` 子句指定返回的属性。

```
# 获取 Tag 为 player，且 ID 为 player100 的点数据 name 的属性值。
nebula> FETCH PROP ON player "player100" \
        YIELD properties(vertex).name AS name;
+--------------+
| name         |
+--------------+
| "Tim Duncan" |
+--------------+
```

获取多个点的属性值

指定多个点 ID 获取多个点的属性值，点之间用英文逗号（,）分隔。

```
# 获取 Tag 为 player，且 ID 为 player101，player102，player103 三个点数据的属性值。
nebula> FETCH PROP ON player "player101", "player102", "player103" YIELD properties(vertex);
+--------------------------------------+
| properties(VERTEX)                   |
+--------------------------------------+
| {age: 33, name: "LaMarcus Aldridge"} |
| {age: 36, name: "Tony Parker"}       |
| {age: 32, name: "Rudy Gay"}          |
+--------------------------------------+
```

基于多个 TAG 获取点的属性值

在 `FETCH` 语句中指定多个 Tag 获取属性值。Tag 之间用英文逗号（,）分隔。

```
# 创建新 Tag t1。
nebula> CREATE TAG IF NOT EXISTS t1(a string, b int);

# 为点 player100 添加 Tag t1。
nebula> INSERT VERTEX t1(a, b) VALUES "player100":("Hello", 100);

# 基于 Tag player 和 t1 获取点 player100 上的属性值。
nebula> FETCH PROP ON player, t1 "player100" YIELD vertex AS v;
+----------------------------------------------------------------------------+
| v                                                                          |
+----------------------------------------------------------------------------+
| ("player100" :player{age: 42, name: "Tim Duncan"} :t1{a: "Hello", b: 100}) |
+----------------------------------------------------------------------------+
```

用户可以在 `FETCH` 语句中组合多个 Tag 和多个点。

```
# 获取 Tag 为 player 或 t1，且 ID 为 player100，player103 两个点数据的属性值。
nebula> FETCH PROP ON player, t1 "player100", "player103" YIELD vertex AS v;
+----------------------------------------------------------------------------+
| v                                                                          |
+----------------------------------------------------------------------------+
| ("player100" :player{age: 42, name: "Tim Duncan"} :t1{a: "Hello", b: 100}) |
| ("player103" :player{age: 32, name: "Rudy Gay"})                           |
+----------------------------------------------------------------------------+
```

在所有标签中获取点的属性值

在 `FETCH` 语句中使用 `*` 获取当前图空间所有标签里，点的属性值。

```
# 获取当前图空间所有标签里 ID 为 player100，player106，team200 的点数据属性值。
nebula> FETCH PROP ON * "player100", "player106", "team200" YIELD vertex AS v;
+----------------------------------------------------------------------------+
| v                                                                          |
+----------------------------------------------------------------------------+
| ("player100" :player{age: 42, name: "Tim Duncan"} :t1{a: "Hello", b: 100}) |
| ("player106" :player{age: 25, name: "Kyle Anderson"})                      |
| ("team200" :team{name: "Warriors"})                                        |
+----------------------------------------------------------------------------+
```

获取边的属性值

语法

```
FETCH PROP ON <edge_type> <src_vid> -> <dst_vid>[@<rank>] [, <src_vid> -> <dst_vid> ...]
YIELD <output>;
```

参数	说明
`edge_type`	Edge type 名称。
`src_vid`	起始点 ID，表示边的起点。
`dst_vid`	目的点 ID，表示边的终点。
`rank`	边的 rank。可选参数，默认值为 `0`。起始点、目的点、Edge type 和 rank 可以唯一确定一条边。
`YIELD`	定义需要返回的输出。详情请参见 `YIELD`。

获取边的所有属性值

```
# 获取连接 player100 和 team204 的边 serve 的所有属性值。
nebula> FETCH PROP ON serve "player100" -> "team204" YIELD properties(edge);
+------------------------------------+
| properties(EDGE)                   |
+------------------------------------+
| {end_year: 2016, start_year: 1997} |
+------------------------------------+
```

获取边的指定属性值

使用 `YIELD` 子句指定返回的属性。

```
# 获取连接 player100 和 team204 的边 serve 的 start_year 属性值。
nebula> FETCH PROP ON serve "player100" -> "team204"    \
        YIELD properties(edge).start_year;
+-----------------------------+
| properties(EDGE).start_year |
+-----------------------------+
```

```
| 1997                        |
+-----------------------------+
```

获取多条边的属性值

指定多个边模式 (`<src_vid> -> <dst_vid>[@<rank>]`) 获取多个边的属性值。模式之间用英文逗号（,）分隔。

```
# 获取连接 player100 和 team204 的边，连接 player133 和 team202 的边 serve 的所有属性值。
nebula> FETCH PROP ON serve "player100" -> "team204", "player133" -> "team202" YIELD edge AS e;
+-----------------------------------------------------------------------+
| e                                                                     |
+-----------------------------------------------------------------------+
| [:serve "player100"->"team204" @0 {end_year: 2016, start_year: 1997}] |
| [:serve "player133"->"team202" @0 {end_year: 2011, start_year: 2002}] |
+-----------------------------------------------------------------------+
```

基于 **RANK** 获取属性值

如果有多条边，起始点、目的点和 Edge type 都相同，可以通过指定 rank 获取正确的边属性值。

```
# 插入不同属性值、不同 rank 的边。
nebula> insert edge serve(start_year,end_year) \
        values "player100"->"team204"@1:(1998, 2017);

nebula> insert edge serve(start_year,end_year) \
        values "player100"->"team204"@2:(1990, 2018);

# 默认返回 rank 为 0 的边。
nebula> FETCH PROP ON serve "player100" -> "team204" YIELD edge AS e;
+-----------------------------------------------------------------------+
| e                                                                     |
+-----------------------------------------------------------------------+
| [:serve "player100"->"team204" @0 {end_year: 2016, start_year: 1997}] |
+-----------------------------------------------------------------------+

# 要获取 rank 不为 0 的边，请在 FETCH 语句中设置 rank。
nebula> FETCH PROP ON serve "player100" -> "team204"@1 YIELD edge AS e;
+-----------------------------------------------------------------------+
| e                                                                     |
+-----------------------------------------------------------------------+
| [:serve "player100"->"team204" @1 {end_year: 2017, start_year: 1998}] |
+-----------------------------------------------------------------------+
```

复合语句中使用 **FETCH**

将 `FETCH` 与原生 nGQL 结合使用是一种常见的方式，例如和 `GO` 一起。

```
# 返回从点 player101 开始的 follow 边的 degree 值。src(edge) 获取边的起始点 ID，dst(edge) 获取边的目的点 ID。
nebula> GO FROM "player101" OVER follow \
        YIELD src(edge) AS s, dst(edge) AS d \
        | FETCH PROP ON follow $-.s -> $-.d \
        YIELD properties(edge).degree;
+-------------------------+
| properties(EDGE).degree |
+-------------------------+
| 95                      |
| 90                      |
| 95                      |
+-------------------------+
```

用户也可以通过自定义变量构建类似的查询。

```
# 返回从点 player101 开始的 follow 边的 degree 值。src(edge) 获取边的起始点 ID，dst(edge) 获取边的目的点 ID。
nebula> $var = GO FROM "player101" OVER follow \
        YIELD src(edge) AS s, dst(edge) AS d; \
        FETCH PROP ON follow $var.s -> $var.d \
        YIELD properties(edge).degree;
+-------------------------+
| properties(EDGE).degree |
+-------------------------+
| 95                      |
| 90                      |
| 95                      |
+-------------------------+
```

更多复合语句的详情，请参见复合查询（子句结构）。

最后更新: 2025年8月4日

4.5.7 SHOW

SHOW CHARSET

`SHOW CHARSET` 语句显示当前的字符集。

目前可用的字符集为 `utf8` 和 `utf8mb4`。默认字符集为 `utf8`。 NebulaGraph 扩展 `uft8` 支持四字节字符，因此 `utf8` 和 `utf8mb4` 是等价的。

语法

```
SHOW CHARSET;
```

示例

```
nebula> SHOW CHARSET;
+---------+-----------------+-------------------+--------+
| Charset | Description     | Default collation | Maxlen |
+---------+-----------------+-------------------+--------+
| "utf8"  | "UTF-8 Unicode" | "utf8_bin"        | 4      |
+---------+-----------------+-------------------+--------+
```

参数	说明
`Charset`	字符集名称。
`Description`	字符集说明。
`Default collation`	默认排序规则。
`Maxlen`	存储一个字符所需的最大字节数。

最后更新: 2025年8月4日

SHOW COLLATION

`SHOW COLLATION` 语句显示当前的排序规则。

目前可用的排序规则为 `utf8_bin` 和 `utf8mb4_bin` 。

- 当字符集为 `utf8` ，默认排序规则为 `utf8_bin` 。
- 当字符集为 `utf8mb4` ，默认排序规则为 `utf8mb4_bin` 。

语法

```
SHOW COLLATION;
```

示例

```
nebula> SHOW COLLATION;
+------------+---------+
| Collation  | Charset |
+------------+---------+
| "utf8_bin" | "utf8"  |
+------------+---------+
```

参数	说明
`Collation`	排序规则名称。
`Charset`	与排序规则关联的字符集名称。

最后更新: 2025年8月4日

SHOW CREATE SPACE

`SHOW CREATE SPACE` 语句显示指定图空间的创建语句。

图空间的更多详细信息，请参见 CREATE SPACE。

语法

```
SHOW CREATE SPACE <space_name>;
```

示例

```
nebula> SHOW CREATE SPACE basketballplayer;
+--------------------+--------------------------------------------------------------------------------------------------------------------------------+
| Space              | Create Space                                                                                                                   |
+--------------------+--------------------------------------------------------------------------------------------------------------------------------+
| "basketballplayer" | "CREATE SPACE `basketballplayer` (partition_num = 10, replica_factor = 1, charset = utf8, collate = utf8_bin, vid_type = FIXED_STRING(32))" |
+--------------------+--------------------------------------------------------------------------------------------------------------------------------+
```

最后更新: 2025年8月4日

SHOW CREATE TAG/EDGE

`SHOW CREATE TAG` 语句显示指定 Tag 的基本信息。Tag 的更多详细信息，请参见 CREATE TAG。

`SHOW CREATE EDGE` 语句显示指定 Edge type 的基本信息。Edge type 的更多详细信息，请参见 CREATE EDGE。

语法

```
SHOW CREATE {TAG <tag_name> | EDGE <edge_name>};
```

示例

```
nebula> SHOW CREATE TAG player;
+----------+-----------------------------------+
| Tag      | Create Tag                        |
+----------+-----------------------------------+
| "player" | "CREATE TAG `player` (            |
|          |  `name` string NULL,              |
|          |  `age` int64 NULL                 |
|          | ) ttl_duration = 0, ttl_col = """ |
+----------+-----------------------------------+

nebula> SHOW CREATE EDGE follow;
+----------+-----------------------------------+
| Edge     | Create Edge                       |
+----------+-----------------------------------+
| "follow" | "CREATE EDGE `follow` (           |
|          |  `degree` int64 NULL              |
|          | ) ttl_duration = 0, ttl_col = """ |
+----------+-----------------------------------+
```

最后更新: 2025年8月4日

SHOW HOSTS

`SHOW HOSTS` 语句可以显示集群信息，包括端口、状态、leader、分片、版本等信息，或者指定显示 Graph、Storage、Meta 服务主机信息。

语法

```
SHOW HOSTS [GRAPH | STORAGE | META];
```

> Note
>
> 对于使用源码安装的 NebulaGraph，执行添加了服务名的命令后，输出的信息中不显示版本信息。

示例

```
nebula> SHOW HOSTS;
+-------------+-------+----------+--------------+----------------------------------+------------------------------+---------+
| Host        | Port  | Status   | Leader count | Leader distribution              | Partition distribution       | Version |
+-------------+-------+----------+--------------+----------------------------------+------------------------------+---------+
| "storaged0" | 9779  | "ONLINE" | 8            | "docs:5, basketballplayer:3"     | "docs:5, basketballplayer:3" | "master" |
| "storaged1" | 9779  | "ONLINE" | 9            | "basketballplayer:4, docs:5"     | "docs:5, basketballplayer:4" | "master" |
| "storaged2" | 9779  | "ONLINE" | 8            | "basketballplayer:3, docs:5"     | "docs:5, basketballplayer:3" | "master" |
+-------------+-------+----------+--------------+----------------------------------+------------------------------+---------+

nebula> SHOW HOSTS GRAPH;
+-----------+------+----------+---------+--------------+---------+
| Host      | Port | Status   | Role    | Git Info Sha | Version |
+-----------+------+----------+---------+--------------+---------+
| "graphd"  | 9669 | "ONLINE" | "GRAPH" | "3ba41bd"    | "master" |
| "graphd1" | 9669 | "ONLINE" | "GRAPH" | "3ba41bd"    | "master" |
| "graphd2" | 9669 | "ONLINE" | "GRAPH" | "3ba41bd"    | "master" |
+-----------+------+----------+---------+--------------+---------+

nebula> SHOW HOSTS STORAGE;
+-------------+------+----------+-----------+--------------+---------+
| Host        | Port | Status   | Role      | Git Info Sha | Version |
+-------------+------+----------+-----------+--------------+---------+
| "storaged0" | 9779 | "ONLINE" | "STORAGE" | "3ba41bd"    | "master" |
| "storaged1" | 9779 | "ONLINE" | "STORAGE" | "3ba41bd"    | "master" |
| "storaged2" | 9779 | "ONLINE" | "STORAGE" | "3ba41bd"    | "master" |
+-------------+------+----------+-----------+--------------+---------+

nebula> SHOW HOSTS META;
+----------+------+----------+--------+--------------+---------+
| Host     | Port | Status   | Role   | Git Info Sha | Version |
+----------+------+----------+--------+--------------+---------+
| "metad2" | 9559 | "ONLINE" | "META" | "3ba41bd"    | "master" |
| "metad0" | 9559 | "ONLINE" | "META" | "3ba41bd"    | "master" |
| "metad1" | 9559 | "ONLINE" | "META" | "3ba41bd"    | "master" |
+----------+------+----------+--------+--------------+---------+
```

最后更新: 2025年8月4日

SHOW INDEX STATUS

`SHOW INDEX STATUS` 语句显示重建原生索引的作业状态，以便确定重建索引是否成功。

语法

```
SHOW {TAG | EDGE} INDEX STATUS;
```

示例

```
nebula> SHOW TAG INDEX STATUS;
+------------------------------------+--------------+
| Name                               | Index Status |
+------------------------------------+--------------+
| "date1_index"                      | "FINISHED"   |
| "basketballplayer_all_tag_indexes" | "FINISHED"   |
| "any_shape_geo_index"              | "FINISHED"   |
+------------------------------------+--------------+

nebula> SHOW EDGE INDEX STATUS;
+----------------+--------------+
| Name           | Index Status |
+----------------+--------------+
| "follow_index" | "FINISHED"   |
+----------------+--------------+
```

相关文档

- 管理作业
- REBUILD NATIVE INDEX

最后更新: 2025年8月4日

SHOW INDEXES

`SHOW INDEXES` 语句可以列出当前图空间内的所有 Tag 和 Edge type（包括属性）的索引。

语法

```
SHOW {TAG | EDGE} INDEXES;
```

示例

```
nebula> SHOW TAG INDEXES;
+------------------+----------+----------+
| Index Name       | By Tag   | Columns  |
+------------------+----------+----------+
| "player_index_0" | "player" | []       |
| "player_index_1" | "player" | ["name"] |
+------------------+----------+----------+

nebula> SHOW EDGE INDEXES;
+----------------+----------+---------+
| Index Name     | By Edge  | Columns |
+----------------+----------+---------+
| "follow_index" | "follow" | []      |
+----------------+----------+---------+
```

历史版本兼容性

NebulaGraph 2.0.1 中， `SHOW TAG/EDGE INDEXES` 语句仅返回 `Names` 。

最后更新: 2025年8月4日

SHOW PARTS

`SHOW PARTS` 语句显示图空间中指定分片或所有分片的信息。

语法

```
SHOW PARTS [<part_id>];
```

示例

```
nebula> SHOW PARTS;
+--------------+--------------------+--------------------+-------+
| Partition ID | Leader             | Peers              | Losts |
+--------------+--------------------+--------------------+-------+
| 1            | "192.168.2.1:9779" | "192.168.2.1:9779" | ""    |
| 2            | "192.168.2.2:9779" | "192.168.2.2:9779" | ""    |
| 3            | "192.168.2.3:9779" | "192.168.2.3:9779" | ""    |
| 4            | "192.168.2.1:9779" | "192.168.2.1:9779" | ""    |
| 5            | "192.168.2.2:9779" | "192.168.2.2:9779" | ""    |
| 6            | "192.168.2.3:9779" | "192.168.2.3:9779" | ""    |
| 7            | "192.168.2.1:9779" | "192.168.2.1:9779" | ""    |
| 8            | "192.168.2.2:9779" | "192.168.2.2:9779" | ""    |
| 9            | "192.168.2.3:9779" | "192.168.2.3:9779" | ""    |
| 10           | "192.168.2.1:9779" | "192.168.2.1:9779" | ""    |
+--------------+--------------------+--------------------+-------+

nebula> SHOW PARTS 1;
+--------------+--------------------+--------------------+-------+
| Partition ID | Leader             | Peers              | Losts |
+--------------+--------------------+--------------------+-------+
| 1            | "192.168.2.1:9779" | "192.168.2.1:9779" | ""    |
+--------------+--------------------+--------------------+-------+
```

返回结果的说明如下：

参数	说明
`Partition ID`	存储分片的 ID。
`Leader`	分片对应的 Raft leader 副本的信息，包括 IP（或主机名）与服务端口。
`Peers`	分片对应的所有副本（leader 与 follower）的信息，包括 IP（或主机名）与服务端口。
`Losts`	分片对应的处于离线状态的副本信息，包括 IP（或主机名）和服务端口。

最后更新: 2025年8月4日

SHOW ROLES

`SHOW ROLES` 语句显示分配给用户的角色信息。

根据登录的用户角色，返回的结果也有所不同：

- 如果登录的用户角色是 `GOD`，或者有权访问该图空间的 `ADMIN`，则返回该图空间内除 `GOD` 之外的所有用户角色信息。
- 如果登录的用户角色是有权访问该图空间 `DBA`、 `USER` 或 `GUEST`，则返回自身的角色信息。
- 如果登录的用户角色没有权限访问该图空间，则返回权限错误。

关于角色的详情请参见内置角色权限。

语法

```
SHOW ROLES IN <space_name>;
```

示例

```
nebula> SHOW ROLES in basketballplayer;
+---------+-----------+
| Account | Role Type |
+---------+-----------+
| "user1" | "ADMIN"   |
+---------+-----------+
```

最后更新: 2025年8月4日

SHOW SNAPSHOTS

`SHOW SNAPSHOTS` 语句显示所有快照信息。

快照的使用方式请参见管理快照。

角色要求

只有 `GOD` 角色的用户（即 `root`）才能执行 `SHOW SNAPSHOTS` 语句。

语法

```
SHOW SNAPSHOTS;
```

示例

```
nebula> SHOW SNAPSHOTS;
+--------------------------------+---------+------------------------------------------------------+
| Name                           | Status  | Hosts                                                |
+--------------------------------+---------+------------------------------------------------------+
| "SNAPSHOT_2020_12_16_11_13_55" | "VALID" | "storaged0:9779, storaged1:9779, storaged2:9779"     |
| "SNAPSHOT_2020_12_16_11_14_10" | "VALID" | "storaged0:9779, storaged1:9779, storaged2:9779"     |
+--------------------------------+---------+------------------------------------------------------+
```

最后更新: 2025年8月4日

SHOW SPACES

`SHOW SPACES` 语句显示现存的图空间。

如何创建图空间，请参见 CREATE SPACE。

语法

```
SHOW SPACES;
```

示例

```
nebula> SHOW SPACES;
+--------------------+
| Name               |
+--------------------+
| "docs"             |
| "basketballplayer" |
+--------------------+
```

最后更新: 2025年8月4日

SHOW STATS

`SHOW STATS` 语句显示最近一次 `SUBMIT JOB STATS` 作业收集的图空间统计信息。

图空间统计信息包含：

- 点的总数
- 边的总数
- 每个 Tag 关联的点的总数
- 每个 Edge type 关联的边的总数

> **Warning**
>
> `SHOW STATS` 返回的不是实时数据。因为返回的数据是最近一次 SUBMIT JOB STATS 作业收集的数据。返回的数据可能包含 TTL 过期数据，该过期数据会在下次执行 Compaction 操作时被删除并不纳入统计。

前提条件

在需要查看统计信息的图空间中执行 `SUBMIT JOB STATS`。详情请参见 SUBMIT JOB STATS。

> **Caution**
>
> `SHOW STATS` 的结果取决于最后一次执行的 `SUBMIT JOB STATS`。如果发生过新的写入或者更改，必须再次执行 `SUBMIT JOB STATS`，否则统计数据有错误。

语法

```
SHOW STATS;
```

示例

```
# 选择图空间。
nebula> USE basketballplayer;

# 执行 SUBMIT JOB STATS。
nebula> SUBMIT JOB STATS;
+------------+
| New Job Id |
+------------+
| 98         |
+------------+

# 确认作业执行成功。
nebula> SHOW JOB 98;
+----------------+---------------+------------+----------------------------+----------------------------+-------------+
| Job Id(TaskId) | Command(Dest) | Status     | Start Time                 | Stop Time                  | Error Code  |
+----------------+---------------+------------+----------------------------+----------------------------+-------------+
| 98             | "STATS"       | "FINISHED" | 2021-11-01T09:33:21.000000 | 2021-11-01T09:33:21.000000 | "SUCCEEDED" |
| 0              | "storaged2"   | "FINISHED" | 2021-11-01T09:33:21.000000 | 2021-11-01T09:33:21.000000 | "SUCCEEDED" |
| 1              | "storaged0"   | "FINISHED" | 2021-11-01T09:33:21.000000 | 2021-11-01T09:33:21.000000 | "SUCCEEDED" |
| 2              | "storaged1"   | "FINISHED" | 2021-11-01T09:33:21.000000 | 2021-11-01T09:33:21.000000 | "SUCCEEDED" |
| "Total:3"      | "Succeeded:3" | "Failed:0" | "In Progress:0"            | ""                         | ""          |
+----------------+---------------+------------+----------------------------+----------------------------+-------------+

# 显示图空间统计信息。
nebula> SHOW STATS;
+---------+------------+-------+
| Type    | Name       | Count |
+---------+------------+-------+
| "Tag"   | "player"   | 51    |
| "Tag"   | "team"     | 30    |
| "Edge"  | "follow"   | 81    |
| "Edge"  | "serve"    | 152   |
| "Space" | "vertices" | 81    |
| "Space" | "edges"    | 233   |
+---------+------------+-------+
```

最后更新: 2025年8月4日

SHOW TAGS/EDGES

`SHOW TAGS` 语句显示当前图空间内的所有 Tag。

`SHOW EDGES` 语句显示当前图空间内的所有 Edge type。

语法

```
SHOW {TAGS | EDGES};
```

示例

```
nebula> SHOW TAGS;
+----------+
| Name     |
+----------+
| "player" |
| "star"   |
| "team"   |
+----------+

nebula> SHOW EDGES;
+----------+
| Name     |
+----------+
| "follow" |
| "serve"  |
+----------+
```

最后更新: 2025年8月4日

SHOW USERS

`SHOW USERS` 语句显示用户信息。

角色要求

只有 `GOD` 角色的用户（即 `root`）才能执行 `SHOW USERS` 语句。

语法

```
SHOW USERS;
```

示例

```
nebula> SHOW USERS;
+---------+-----------------+
| Account | IP Whitelist    |
+---------+-----------------+
| "root"  | ""              |
| "user1" | ""              |
| "user2" | "192.168.10.10" |
+---------+-----------------+
```

最后更新: 2025年8月4日

SHOW SESSIONS

登录 NebulaGraph 数据库时，会创建对应会话，用户可以查询会话信息。

注意事项

- 执行 `exit` 退出登录时，客户端会调用 API `release`，释放会话并清除会话信息。如果没有正常退出，且没有在配置文件 nebula-graphd.conf 设置空闲会话超时时间（`session_idle_timeout_secs`），会话不会自动释放。对于未自动释放的会话，需要手动删除指定会话，详情参见终止会话。
- `SHOW SESSIONS` 查询所有 Graph 服务上的会话信息。
- `SHOW LOCAL SESSIONS` 从当前连接的 Graph 服务获取会话信息，不会查询其他 Graph 服务上的会话信息。
- `SHOW SESSION <Session_Id>` 查询指定 Session ID 的会话信息。

语法

```
SHOW [LOCAL] SESSIONS;
SHOW SESSION <Session_Id>;
```

示例

```
nebula> SHOW SESSIONS;
+------------------+----------+--------------------+----------------------------+----------------------------+------------------+----------+--------------------+
| SessionId        | UserName | SpaceName          | CreateTime                 | UpdateTime                 | GraphAddr        | Timezone | ClientIp           |
+------------------+----------+--------------------+----------------------------+----------------------------+------------------+----------+--------------------+
| 1651220858102296 | "root"   | "basketballplayer" | 2022-04-29T08:27:38.102296 | 2022-04-29T08:50:46.282921 | "127.0.0.1:9669" | 0        | "127.0.0.1" |
| 1651199330300991 | "root"   | "basketballplayer" | 2022-04-29T02:28:50.300991 | 2022-04-29T08:16:28.339038 | "127.0.0.1:9669" | 0        | "127.0.0.1" |
| 1651112899847744 | "root"   | "basketballplayer" | 2022-04-28T02:28:19.847744 | 2022-04-28T08:17:44.470210 | "127.0.0.1:9669" | 0        | "127.0.0.1" |
| 1651041092662100 | "root"   | "basketballplayer" | 2022-04-27T06:31:32.662100 | 2022-04-27T07:01:25.200978 | "127.0.0.1:9669" | 0        | "127.0.0.1" |
| 1650959429593975 | "root"   | "basketballplayer" | 2022-04-26T07:50:29.593975 | 2022-04-26T07:51:47.184810 | "127.0.0.1:9669" | 0        | "127.0.0.1" |
| 1650958897679595 | "root"   | ""                 | 2022-04-26T07:41:37.679595 | 2022-04-26T07:41:37.683802 | "127.0.0.1:9669" | 0        | "127.0.0.1" |
+------------------+----------+--------------------+----------------------------+----------------------------+------------------+----------+--------------------+

nebula> SHOW SESSION 1635254859271703;
+------------------+----------+--------------------+----------------------------+----------------------------+------------------+----------+--------------------+
| SessionId        | UserName | SpaceName          | CreateTime                 | UpdateTime                 | GraphAddr        | Timezone | ClientIp           |
+------------------+----------+--------------------+----------------------------+----------------------------+------------------+----------+--------------------+
| 1651220858102296 | "root"   | "basketballplayer" | 2022-04-29T08:27:38.102296 | 2022-04-29T08:50:54.254384 | "127.0.0.1:9669" | 0        | "127.0.0.1" |
+------------------+----------+--------------------+----------------------------+----------------------------+------------------+----------+--------------------+
```

参数	说明
`SessionId`	会话 ID，唯一标识一个会话。
`UserName`	会话的登录用户名称。
`SpaceName`	用户当前所使用的图空间。刚登录时为空（`""`）。
`CreateTime`	会话的创建时间，即用户认证登录的时间。时区为配置文件中 `timezone_name` 指定的时区。
`UpdateTime`	用户有执行操作时，会更新此时间。时区为配置文件中 `timezone_name` 指定的时区。
`GraphAddr`	会话的 Graph 服务 IP（或主机名）和端口。
`Timezone`	保留参数，暂无意义。
`ClientIp`	会话的客户端 IP 或主机名。

最后更新: 2025年8月4日

SHOW QUERIES

`SHOW QUERIES` 语句可以查看当前 Session 中正在执行的查询请求信息。

> Note
>
> 如果需要终止查询，请参见终止查询。

注意事项

- `SHOW LOCAL QUERIES` 从本地缓存获取当前 Session 中查询的状态，几乎没有延迟。
- `SHOW QUERIES` 从 Meta 服务获取所有 Session 中的查询信息。这些信息会根据参数 `session_reclaim_interval_secs` 定义的周期同步到 Meta 服务，因此在客户端获取到的信息可能属于上个同步周期。

语法

```
SHOW [LOCAL] QUERIES;
```

示例

```
nebula> SHOW LOCAL QUERIES;
+------------------+-----------------+--------+--------------------+----------------------------+----------------+-----------+-----------------------+
| SessionID        | ExecutionPlanID | User   | Host               | StartTime                  | DurationInUSec | Status    | Query                 |
+------------------+-----------------+--------+--------------------+----------------------------+----------------+-----------+-----------------------+
| 1625463842921750 | 46              | "root" | ""192.168.x.x":9669" | 2021-07-05T05:44:19.502903 | 0              | "RUNNING" | "SHOW LOCAL QUERIES;" |
+------------------+-----------------+--------+--------------------+----------------------------+----------------+-----------+-----------------------+

nebula> SHOW QUERIES;
+------------------+-----------------+---------+----------------------+----------------------------+----------------+-----------+---------------------------------------------------------+
| SessionID        | ExecutionPlanID | User    | Host                 | StartTime                  | DurationInUSec | Status    | Query                                                   |
+------------------+-----------------+---------+----------------------+----------------------------+----------------+-----------+---------------------------------------------------------+
| 1625456037718757 | 54              | "user1" | ""192.168.x.x":9669" | 2021-07-05T05:51:08.691318 | 1504502        | "RUNNING" | "MATCH p=(v:player)-[*1..4]-(v2) RETURN v2 AS Friends;" |
+------------------+-----------------+---------+----------------------+----------------------------+----------------+-----------+---------------------------------------------------------+

# 返回耗时 TOP 10 的查询。
nebula> SHOW QUERIES | ORDER BY $-.DurationInUSec DESC | LIMIT 10;
+------------------+-----------------+---------+----------------------+----------------------------+----------------+-----------+-------------------------------------------------------+
| SessionID        | ExecutionPlanID | User    | Host                 | StartTime                  | DurationInUSec | Status    | Query                                                 |
+------------------+-----------------+---------+----------------------+----------------------------+----------------+-----------+-------------------------------------------------------+
| 1625471375320831 | 98              | "user2" | ""192.168.x.x":9669" | 2021-07-05T07:50:24.461779 | 2608176        | "RUNNING" | "MATCH (v:player)-[*1..4]-(v2) RETURN v2 AS Friends;" |
| 1625456037718757 | 99              | "user1" | ""192.168.x.x":9669" | 2021-07-05T07:50:24.910616 | 2159333        | "RUNNING" | "MATCH (v:player)-[*1..4]-(v2) RETURN v2 AS Friends;" |
+------------------+-----------------+---------+----------------------+----------------------------+----------------+-----------+-------------------------------------------------------+
```

参数说明如下。

参数	说明
`SessionID`	会话 ID。
`ExecutionPlanID`	执行计划 ID。
`User`	执行查询的用户名。
`Host`	用户连接的服务器地址和端口。
`StartTime`	执行查询的开始时间。
`DurationInUSec`	执行查询的持续时长。单位：微秒。
`Status`	查询的当前状态。
`Query`	查询语句。

最后更新: 2025年8月4日

SHOW META LEADER

`SHOW META LEADER` 语句显示当前 Meta 集群的 leader 信息。

关于 Meta 服务的详细说明请参见 Meta 服务。

语法

```
SHOW META LEADER;
```

示例

```
nebula> SHOW META LEADER;
+------------------+---------------------------+
| Meta Leader      | secs from last heart beat |
+------------------+---------------------------+
| "127.0.0.1:9559" | 3                         |
+------------------+---------------------------+
```

参数	说明
`Meta Leader`	Meta 集群的 leader 信息，包括 leader 所在服务器的 IP（或主机名）和端口。
`secs from last heart beat`	距离上次心跳的时间间隔。单位：秒。

最后更新: 2025年8月4日

4.5.8 FIND PATH

`FIND PATH` 语句查找指定起始点和目的点之间的路径。

> Note
>
> 用户可在配置文件 `nebula-graphd.conf` 中添加 `num_operator_threads` 参数提高 `FIND PATH` 的查询性能。`num_operator_threads` 的取值为 `2` ~ `10`，该值不能超过 Graph 服务所在机器的 CPU 核心个数，建议设置为 Graph 服务所在机器的 CPU 核心个数。关于配置文件的详细信息，参见 Graph 服务配置。

语法

```
FIND { SHORTEST | SINGLE SHORTEST | ALL | NOLOOP } PATH [WITH PROP] FROM <vertex_id_list> TO <vertex_id_list>
OVER <edge_type_list> [REVERSELY | BIDIRECT]
[<WHERE clause>] [UPTO <N> {STEP|STEPS}]
YIELD path as <alias>
[| ORDER BY $-.path] [| LIMIT <M>];

<vertex_id_list> ::=
    [vertex_id [, vertex_id] ...]
```

- `SHORTEST`：查找所有最短路径。
- `SINGLE SHORTEST`：查找所有最短路径，随机返回其中一条。
- `ALL`：查找所有路径。
- `NOLOOP`：查找非循环路径。
- `WITH PROP`：展示点和边的属性。不添加本参数则隐藏属性。
- `<vertex_id_list>`：点 ID 列表。多个点用英文逗号（,）分隔。支持 `$-` 和 `$var`。
- `<edge_type_list>`：Edge type 列表。多个 Edge type 用英文逗号（,）分隔。`*` 表示所有 Edge type。
- `REVERSELY | BIDIRECT`：`REVERSELY` 表示反向，`BIDIRECT` 表示双向。
- `<WHERE clause>`：可以使用 `WHERE` 子句过滤边属性。
- `UPTO <N> {STEP|STEPS}`：路径的最大跳数。默认值为 `5`。
- `ORDER BY $-.path`：将返回结果进行排序。排序规则参见 Path。
- `LIMIT <M>`：指定返回的最大行数。

> Note
>
> `FIND PATH` 语句检索的路径类型为 `trail`，即检索的路径只有点可以重复，边不可以重复。详情请参见路径。

限制

- 指定起始点和目的点的列表后，会返回起始点和目的点所有组合的路径。
- 搜索所有路径时可能会出现循环。
- 使用 `WHERE` 子句时只能过滤边属性，暂不支持过滤点属性，且不支持函数。
- graphd 是单进程查询，会占用很多内存。

示例

返回的路径格式类似于 `(<vertex_id>)-[:<edge_type_name>@<rank>]->(<vertex_id)` 。

```
# 查找并返回 player102 到 team204 的最短路径。
nebula> FIND SHORTEST PATH FROM "player102" TO "team204" OVER * YIELD path AS p;
+--------------------------------------------+
| p                                          |
+--------------------------------------------+
| <("player102")-[:serve@0 {}]->("team204")> |
+--------------------------------------------+
```

```
# 查找并返回带属性值的 team204 到 player100 的最短反向路径。
nebula> FIND SHORTEST PATH WITH PROP FROM "team204" TO "player100" OVER * REVERSELY YIELD path AS p;
+--------------------------------------------------------------------------------------------------------------------------------------+
| p                                                                                                                                    |
+--------------------------------------------------------------------------------------------------------------------------------------+
| <("team204" :team{name: "Spurs"})<-[:serve@0 {end_year: 2016, start_year: 1997}]-("player100" :player{age: 42, name: "Tim Duncan"})> |
+--------------------------------------------------------------------------------------------------------------------------------------+
```

```
# 查找并返回起点为 player100,player130 而终点为 player132,player133 的 18 跳之内双向最短路径。
nebula> FIND SHORTEST PATH FROM "player100", "player130" TO "player132", "player133" OVER * BIDIRECT UPTO 18 STEPS YIELD path as p;
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
+
|
p
|
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
+
| <("player100")<-[:follow@0 {}]-("player144")<-[:follow@0 {}]-
("player133")>                                                                                                   |
| <("player100")-[:serve@0 {}]->("team204")<-[:serve@0 {}]-("player138")-[:serve@0 {}]->("team225")<-[:serve@0 {}]-
("player132")>                                                         |
| <("player130")-[:serve@0 {}]->("team219")<-[:serve@0 {}]-("player112")-[:serve@0 {}]->("team204")<-[:serve@0 {}]-("player114")<-[:follow@0 {}]-
("player133")>                             |
| <("player130")-[:serve@0 {}]->("team219")<-[:serve@0 {}]-("player109")-[:serve@0 {}]->("team204")<-[:serve@0 {}]-("player114")<-[:follow@0 {}]-
("player133")>                             |
| <("player130")-[:serve@0 {}]->("team219")<-[:serve@0 {}]-("player104")-[:serve@20182019 {}]->("team204")<-[:serve@0 {}]-("player114")<-[:follow@0 {}]-
("player133")>                       |
| ...
|
| <("player130")-[:serve@0 {}]->("team219")<-[:serve@0 {}]-("player112")-[:serve@0 {}]->("team204")<-[:serve@0 {}]-("player138")-[:serve@0 {}]->("team225")<-[:serve@0 {}]-
("player132")>       |
| <("player130")-[:serve@0 {}]->("team219")<-[:serve@0 {}]-("player109")-[:serve@0 {}]->("team204")<-[:serve@0 {}]-("player138")-[:serve@0 {}]->("team225")<-[:serve@0 {}]-
("player132")>       |
| ...
|
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
+
```

```
# 查找 player100 和 player130 分别与 player132 和 player133 之间跳数最长为 18 步的最短路径。
nebula> FIND SINGLE SHORTEST PATH FROM "player100", "player130" TO "player132", "player133" OVER * BIDIRECT UPTO 18 STEPS YIELD path as p;
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| p                                                                                                                                                                                 |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| <("player100")<-[:follow@0 {}]-("player144")<-[:follow@0 {}]-("player133")>                                                                                                       |
| <("player100")-[:serve@0 {}]->("team204")<-[:serve@0 {}]-("player138")-[:serve@0 {}]->("team225")<-[:serve@0 {}]-("player132")>                                                   |
| <("player130")-[:serve@0 {}]->("team219")<-[:serve@0 {}]-("player112")-[:serve@0 {}]->("team204")<-[:serve@0 {}]-("player114")<-[:follow@0 {}]-("player133")>                     |
| <("player130")-[:serve@0 {}]->("team219")<-[:serve@0 {}]-("player112")-[:serve@0 {}]->("team204")<-[:serve@0 {}]-("player138")-[:serve@0 {}]->("team225")<-[:serve@0 {}]-("player132")> |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
```

```
# 查找所有从 player100 到 team204，并且 degree 为空或者大于等于 0 的路径。
nebula> FIND ALL PATH FROM "player100" TO "team204" OVER * WHERE follow.degree is EMPTY or follow.degree >=0 YIELD path AS p;
+------------------------------------------------------------------------------+
| p                                                                            |
+------------------------------------------------------------------------------+
| <("player100")-[:serve@0 {}]->("team204")>                                   |
| <("player100")-[:follow@0 {}]->("player125")-[:serve@0 {}]->("team204")>     |
| <("player100")-[:follow@0 {}]->("player101")-[:serve@0 {}]->("team204")>     |
| ...                                                                          |
+------------------------------------------------------------------------------+
```

```
# 查找所有从 player100 到 team204 无环路径。
nebula> FIND NOLOOP PATH FROM "player100" TO "team204" OVER * YIELD path AS p;
+--------------------------------------------------------------------------------------------------------+
| p                                                                                                      |
+--------------------------------------------------------------------------------------------------------+
| <("player100")-[:serve@0 {}]->("team204")>                                                             |
| <("player100")-[:follow@0 {}]->("player125")-[:serve@0 {}]->("team204")>                               |
| <("player100")-[:follow@0 {}]->("player101")-[:serve@0 {}]->("team204")>                               |
| <("player100")-[:follow@0 {}]->("player101")-[:follow@0 {}]->("player125")-[:serve@0 {}]->("team204")> |
| <("player100")-[:follow@0 {}]->("player101")-[:follow@0 {}]->("player102")-[:serve@0 {}]->("team204")> |
| ...                                                                                                    |
+--------------------------------------------------------------------------------------------------------+
```

FAQ

是否支持 WHERE 子句，以实现图遍历过程中的条件过滤?

支持使用 `WHERE` 子句过滤，但只能过滤边属性，不支持过滤点属性。

如示例中的 `WHERE follow.degree is EMPTY or follow.degree >= 0` 。

最后更新: 2025年8月4日

4.5.9 GET SUBGRAPH

`GET SUBGRAPH` 语句查询并返回一个通过从指定点出发对图进行游走而生成的子图。在 `GET SUBGRAPH` 语句中，用户可以指定游走的步数以及游走所经过的边的类型或方向。

语法

```
GET SUBGRAPH [WITH PROP] [<step_count> {STEP|STEPS}] FROM {<vid>, <vid>...}
[{IN | OUT | BOTH} <edge_type>, <edge_type>...]
[WHERE <expression> [AND <expression> ...]]
YIELD [VERTICES AS <vertex_alias>] [, EDGES AS <edge_alias>];
```

- `WITH PROP`：展示属性。不添加本参数则隐藏属性。
- `step_count`：指定从起始点开始的跳数，返回从 0 到 `step_count` 跳的子图。必须是非负整数。默认值为 1。
- `vid`：指定起始点 ID。
- `edge_type`：指定 Edge type。可以用 `IN`、`OUT` 和 `BOTH` 来指定起始点上该 Edge type 的方向。默认为 `BOTH`。
- `WHERE`：指定遍历的过滤条件，可以结合布尔运算符 AND 使用。
- `YIELD`：定义需要返回的输出。可以仅返回点或边。必须设置别名。

> Note
>
> `GET SUBGRAPH` 语句检索的路径类型为 `trail`，即检索的路径只有点可以重复，边不可以重复。详情请参见路径。

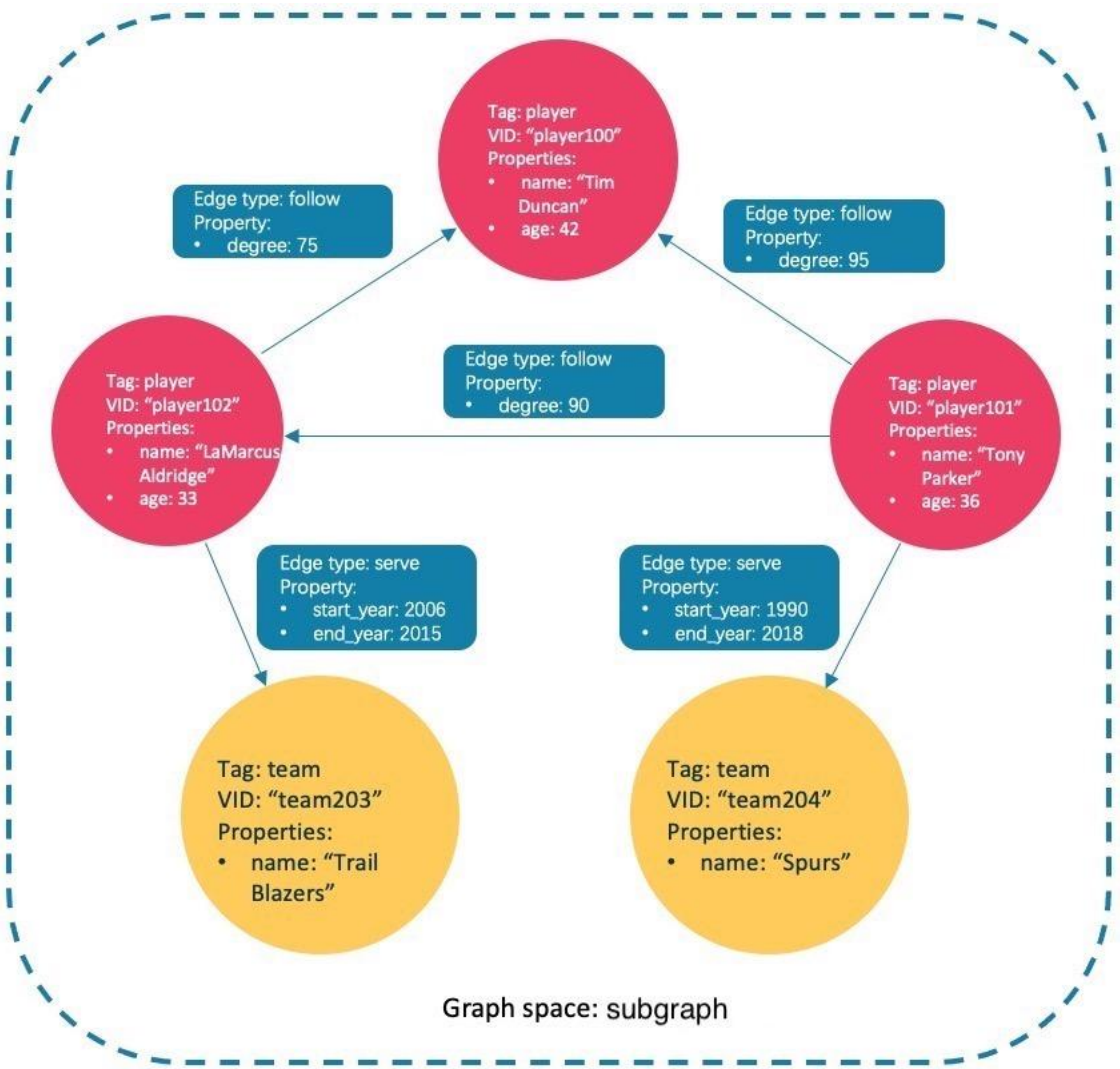
WHERE 语句限制

在 `GET SUBGRAPH` 语句中使用 `WHERE` 子句，注意以下限制：

- 仅支持 `AND` 运算符。
- 仅支持过滤目的点，点的格式为 `$$.tagName.propName`。
- 支持过滤边，边的格式为 `edge_type.propName`。
- 支持数学函数、聚合函数、字符串函数、日期时间函数、列表函数中的通用函数和类型转化函数。
- 不支持聚合函数、Schema 相关函数、条件表达式函数、谓词函数、geo 函数和自定义函数，列表函数中除通用函数以外的函数。

示例

以下面的示例图进行演示。



插入测试数据：

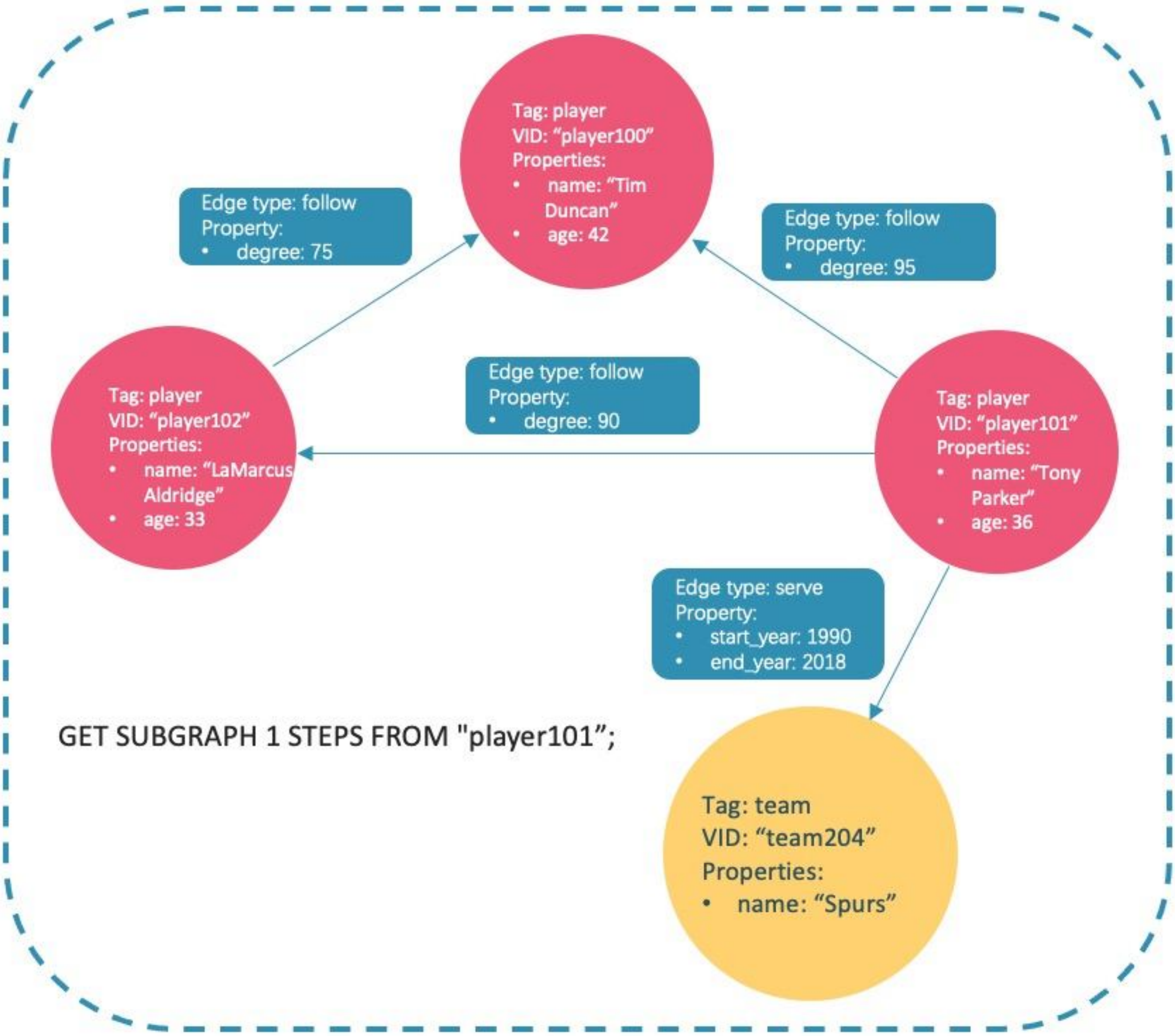
```
nebula> CREATE SPACE IF NOT EXISTS subgraph(partition_num=15, replica_factor=1, vid_type=fixed_string(30));
nebula> USE subgraph;
nebula> CREATE TAG IF NOT EXISTS player(name string, age int);
nebula> CREATE TAG IF NOT EXISTS team(name string);
nebula> CREATE EDGE IF NOT EXISTS follow(degree int);
nebula> CREATE EDGE IF NOT EXISTS serve(start_year int, end_year int);
nebula> INSERT VERTEX player(name, age) VALUES "player100":("Tim Duncan", 42);
nebula> INSERT VERTEX player(name, age) VALUES "player101":("Tony Parker", 36);
nebula> INSERT VERTEX player(name, age) VALUES "player102":("LaMarcus Aldridge", 33);
nebula> INSERT VERTEX team(name) VALUES "team203":("Trail Blazers"), "team204":("Spurs");
nebula> INSERT EDGE follow(degree) VALUES "player101" -> "player100":(95);
nebula> INSERT EDGE follow(degree) VALUES "player101" -> "player102":(90);
```

```
nebula> INSERT EDGE follow(degree) VALUES "player102" -> "player100":(75);
nebula> INSERT EDGE serve(start_year, end_year) VALUES "player101" -> "team204":(1999, 2018),"player102" -> "team203":(2006,  2015);
```

- 查询从点 `player101` 开始、0~1 跳、所有 Edge type 的子图。

```
nebula> GET SUBGRAPH 1 STEPS FROM "player101" YIELD VERTICES AS nodes, EDGES AS relationships;
+-------------------------------------------------------------------------
+-------------------------------------------------------------------------------------------------------------------------+
| nodes                                                                    |
relationships                                                                                                         |
+-------------------------------------------------------------------------
+-------------------------------------------------------------------------------------------------------------------------+
| [("player101" :player{})]                                                | [[:serve "player101"->"team204" @0 {}], [:follow "player101"->"player100" @0 {}], [:follow "player101"-
>"player102" @0 {}]] |
| [("team204" :team{}), ("player100" :player{}), ("player102" :player{})] | [[:follow "player102"->"player100" @0
{}]]                                                                                                                 |
+-------------------------------------------------------------------------
+-------------------------------------------------------------------------------------------------------------------------+
```

返回的子图如下。



- 查询从点 `player101` 开始、0~1 跳、`follow` 类型的入边的子图。

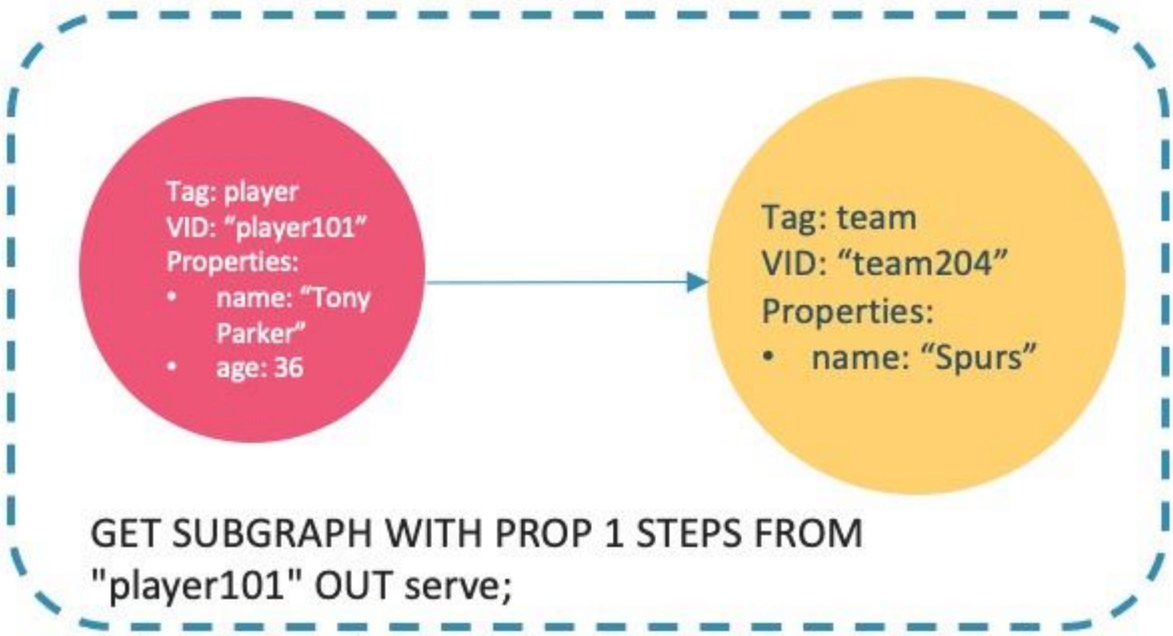
```
nebula> GET SUBGRAPH 1 STEPS FROM "player101" IN follow YIELD VERTICES AS nodes, EDGES AS relationships;
+--------------------------+---------------+
| nodes                    | relationships |
+--------------------------+---------------+
| [("player101" :player{})] | []            |
+--------------------------+---------------+
```

因为 `player101` 没有 `follow` 类型的入边。所以仅返回点 `player101`。

- 查询从点 `player101` 开始、0~1 跳、`serve` 类型的出边的子图，同时展示边的属性。

```
nebula> GET SUBGRAPH WITH PROP 1 STEPS FROM "player101" OUT serve YIELD VERTICES AS nodes, EDGES AS relationships;
+-------------------------------------------------------+-------------------------------------------------------------------------+
| nodes                                                 | relationships                                                           |
+-------------------------------------------------------+-------------------------------------------------------------------------+
| [("player101" :player{age: 36, name: "Tony Parker"})] | [[:serve "player101"->"team204" @0 {end_year: 2018, start_year: 1999}]] |
| [("team204" :team{name: "Spurs"})]                    | []                                                                      |
+-------------------------------------------------------+-------------------------------------------------------------------------+
```

返回的子图如下。



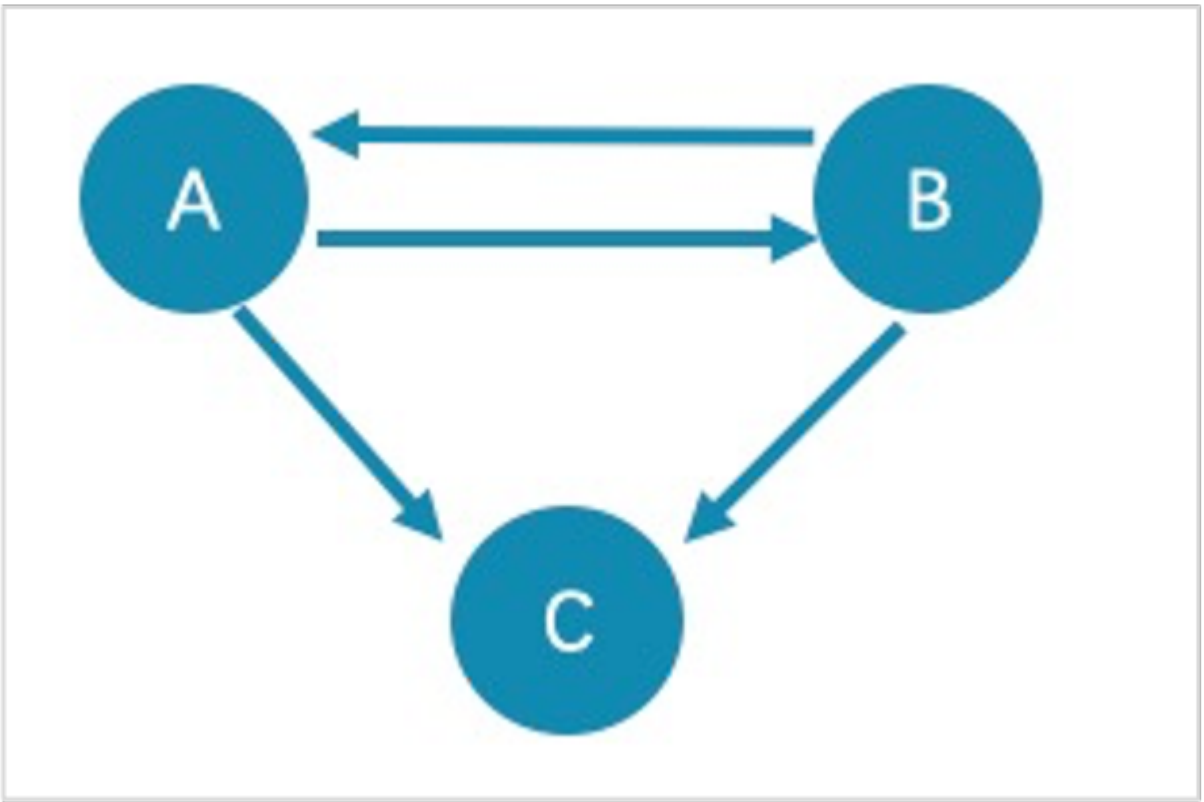
- 查询从点 `player101` 开始、0~2 跳、`follow` 类型边 `degree` 大于 90，年龄大于 30 的子图，同时展示边的属性。

```
nebula> GET SUBGRAPH WITH PROP 2 STEPS FROM "player101" \
    WHERE follow.degree > 90 AND $$.player.age > 30 \
    YIELD VERTICES AS nodes, EDGES AS relationships;
+-------------------------------------------------------+------------------------------------------------------+
| nodes                                                 | relationships                                        |
+-------------------------------------------------------+------------------------------------------------------+
| [("player101" :player{age: 36, name: "Tony Parker"})] | [[:follow "player101"->"player100" @0 {degree: 95}]] |
| [("player100" :player{age: 42, name: "Tim Duncan"})]  | []                                                   |
+-------------------------------------------------------+------------------------------------------------------+
```

FAQ

为什么返回结果中会出现超出 `STEP_COUNT` 跳数之外的关系？

为了展示子图的完整性，会在满足条件的所有点上额外查询一跳。例如下图。



- 用 `GET SUBGRAPH 1 STEPS FROM "A";` 查询的满足结果的路径是 `A->B`、`B->A` 和 `A->C`，为了子图的完整性，会在满足结果的点上额外查询一跳，即 `B->C`。
- 用 `GET SUBGRAPH 1 STEPS FROM "A" IN follow;` 查询的满足结果的路径是 `B->A`，在满足结果的点上额外查询一跳，即 `A->B`。

如果只是查询满足条件的路径或点，建议使用 MATCH 或 GO 语句。例如：

```
nebula> MATCH p= (v:player) -- (v2) WHERE id(v)=="A" RETURN p;

nebula> GO 1 STEPS FROM "A" OVER follow YIELD src(edge),dst(edge);
```

为什么返回结果中会出现低于 **STEP_COUNT** 跳数的关系?

查询到没有多余子图数据时会停止查询，且不会返回空值。

```
# 查询从点 player101 开始、0~100 跳、follow 类型的出边的子图。
nebula> GET SUBGRAPH 100 STEPS FROM "player101" OUT follow YIELD VERTICES AS nodes, EDGES AS relationships;
+----------------------------------------------------+-------------------------------------------------------------------------------------------+
| nodes                                              | relationships                                                                             |
+----------------------------------------------------+-------------------------------------------------------------------------------------------+
| [("player101" :player{})]                          | [[:follow "player101"->"player100" @0 {}], [:follow "player101"->"player102" @0 {}]] |
| [("player100" :player{}), ("player102" :player{})] | [[:follow "player102"->"player100" @0 {}]]                                                |
+----------------------------------------------------+-------------------------------------------------------------------------------------------+
```

最后更新: 2025年8月4日

4.6 子句和选项

4.6.1 GROUP BY

`GROUP BY` 子句可以用于聚合数据。

openCypher 兼容性

本文操作仅适用于原生 nGQL。

用户也可以使用 openCypher 方式的 count() 函数聚合数据。

```
nebula>  MATCH (v:player)<-[:follow]-(:player) RETURN v.player.name AS Name, count(*) as cnt ORDER BY cnt DESC;
+---------------------+-----+
| Name                | cnt |
+---------------------+-----+
| "Tim Duncan"        | 10  |
| "LeBron James"      | 6   |
| "Tony Parker"       | 5   |
| "Chris Paul"        | 4   |
| "Manu Ginobili"     | 4   |
+---------------------+-----+
...
```

语法

`GROUP BY` 子句可以聚合相同值的行，然后进行计数、排序和计算等操作。

`GROUP BY` 子句可以在管道符（|）之后和 `YIELD` 子句之前使用。

```
| GROUP BY <var> YIELD <var>, <aggregation_function(var)>
```

`aggregation_function()` 函数支持 `avg()`、`sum()`、`max()`、`min()`、`count()`、`collect()`、`std()`。

示例

```
# 查找所有连接到 player100 的点，并根据他们的姓名进行分组，返回姓名的出现次数。
nebula> GO FROM "player100" OVER follow BIDIRECT \
        YIELD properties($$).name as Name \
        | GROUP BY $-.Name \
        YIELD $-.Name as Player, count(*) AS Name_Count;
+---------------------+------------+
| Player              | Name_Count |
+---------------------+------------+
| "Shaquille O'Neal"  | 1          |
| "Tiago Splitter"    | 1          |
| "Manu Ginobili"     | 2          |
| "Boris Diaw"        | 1          |
| "LaMarcus Aldridge" | 1          |
| "Tony Parker"       | 2          |
| "Marco Belinelli"   | 1          |
| "Dejounte Murray"   | 1          |
| "Danny Green"       | 1          |
| "Aron Baynes"       | 1          |
+---------------------+------------+
```

```
# 查找所有连接到 player100 的点，并根据起始点进行分组，返回 degree 的总和。
nebula> GO FROM "player100" OVER follow \
        YIELD src(edge) AS player, properties(edge).degree AS degree \
        | GROUP BY $-.player \
        YIELD sum($-.degree);
+----------------+
| sum($-.degree) |
+----------------+
| 190            |
+----------------+
```

`sum()` 函数详情请参见内置数学函数。

隐式分组

在上述 nGQL 语句中明确写出 `GROUP BY` 并起到分组字段作用的用法称为 `GROUP BY` 显示用法；而在 openCypher 语句中 `GROUP BY` 的用法是隐式的，即在语句中不用写出 `GROUP BY` 也可起到分组字段的作用。nGQL 语句中显示地 `GROUP BY` 用法与 openCypher 语句中的隐式地 `GROUP BY` 用法相同，并且 nGQL 语句兼容 openCypher 的用法，即也支持隐式地使用 `GROUP BY` 。有关 `GROUP BY` 的隐式用法，请参见Stack Overflow。

例如：查询 34 岁以上的球员中完全重叠服役的区间。

```
nebula> LOOKUP ON player WHERE player.age > 34 YIELD id(vertex) AS v | \
        GO FROM $-.v OVER serve YIELD serve.start_year AS start_year, serve.end_year AS end_year | \
        YIELD $-.start_year, $-.end_year, count(*) AS count | \
        ORDER BY $-.count DESC | LIMIT 5;
+---------------+-------------+-------+
| $-.start_year | $-.end_year | count |
+---------------+-------------+-------+
| 2018          | 2019        | 3     |
| 2007          | 2012        | 2     |
| 1998          | 2004        | 2     |
| 2017          | 2018        | 2     |
| 2010          | 2011        | 2     |
+---------------+-------------+-------+
```

最后更新: 2025年8月4日

4.6.2 LIMIT

`LIMIT` 子句限制输出结果的行数。`LIMIT` 在原生 nGQL 语句和 openCypher 兼容语句中的用法有所不同。

- 在原生 nGQL 语句中，一般需要在 `LIMIT` 子句前使用管道符，可以直接在 LIMIT 语句后设置或者省略偏移量参数。
- 在 openCypher 兼容语句中，不允许在 `LIMIT` 子句前使用管道符，可以使用 `SKIP` 指明偏移量。

> Note
>
> 在原生 nGQL 或 openCypher 方式中使用 `LIMIT` 时，使用 `ORDER BY` 子句限制输出顺序非常重要，否则会输出一个不可预知的子集。

原生 nGQL 语句中的 LIMIT

在原生 nGQL 中，`LIMIT` 有通用语法和 `GO` 语句中的专属语法。

原生 NGQL 中的通用 LIMIT 语法

原生 nGQL 中的通用 `LIMIT` 语法与 `SQL` 中的 `LIMIT` 原理相同。`LIMIT` 子句接收一个或两个参数，参数的值必须是非负整数，且必须用在管道符之后。语法和说明如下：

```
... | LIMIT [<offset>,] <number_rows>;
```

参数	说明
`offset`	偏移量，即定义从哪一行开始返回。索引从 `0` 开始。默认值为 `0`，表示从第一行开始返回。
`number_rows`	返回的总行数。

示例：

```
# 从结果中返回最前面的 3 行数据。
nebula> LOOKUP ON player YIELD id(vertex)|\
        LIMIT 3;
+-------------+
| id(VERTEX)  |
+-------------+
| "player100" |
| "player101" |
| "player102" |
+-------------+

# 从排序后结果中返回第 2 行开始的 3 行数据。
nebula> GO FROM "player100" OVER follow REVERSELY \
        YIELD properties($$).name AS Friend, properties($$).age AS Age \
        | ORDER BY $-.Age, $-.Friend \
        | LIMIT 1, 3;
+-------------------+-----+
| Friend            | Age |
+-------------------+-----+
| "Danny Green"     | 31  |
| "Aron Baynes"     | 32  |
| "Marco Belinelli" | 32  |
+-------------------+-----+
```

GO 语句中的 LIMIT

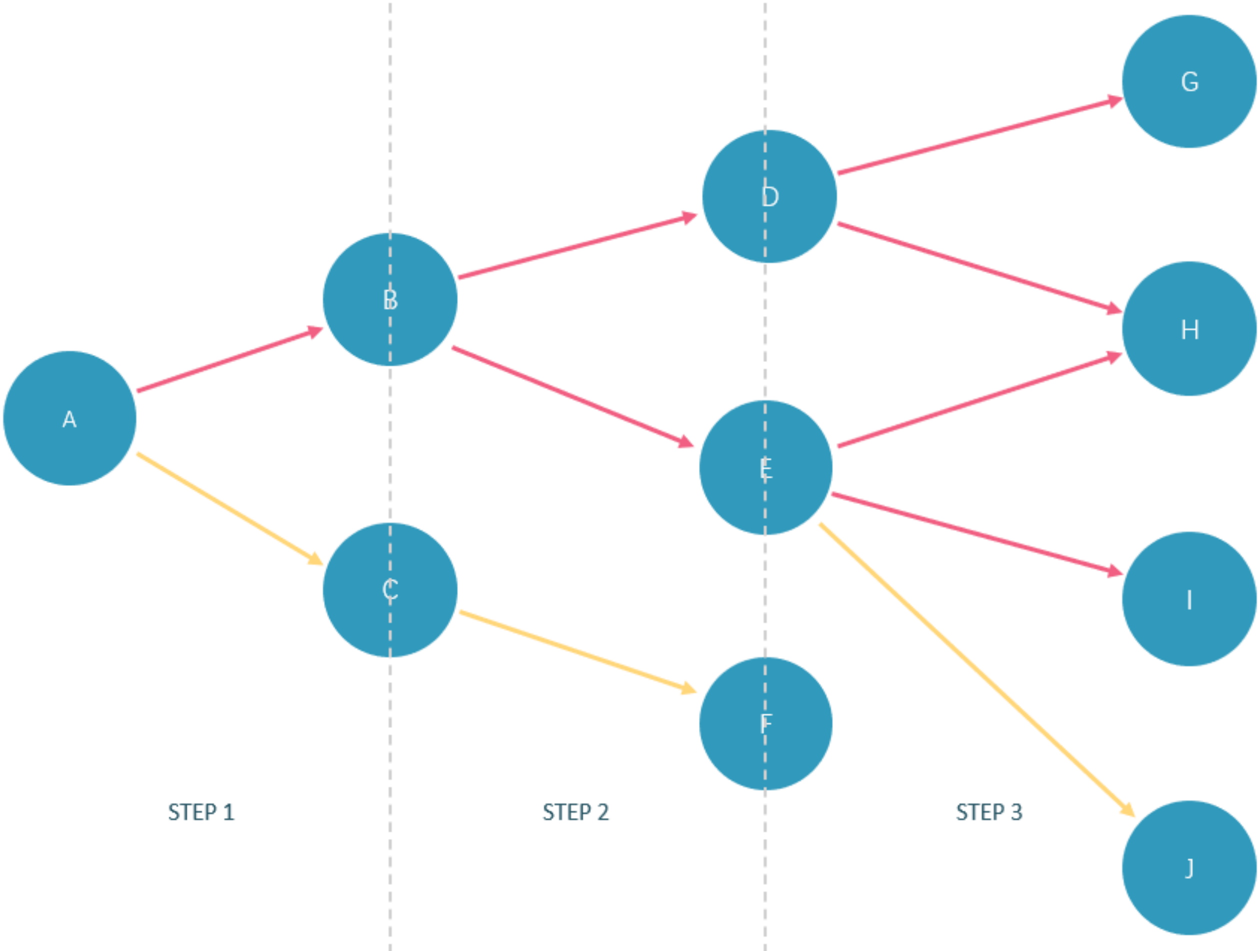
`GO` 语句中的 `LIMIT` 除了支持原生 nGQL 中的通用语法外，还支持根据边限制输出结果数量。

语法：

```
<go_statement> LIMIT <limit_list>;
```

`limit_list` 是一个列表，列表中的元素必须为自然数，且元素数量必须与 `GO` 语句中的 `STEPS` 的最大数相同。下文以 `GO 1 TO 3 STEPS FROM "A" OVER * LIMIT <limit_list>` 为例详细介绍 `LIMIT` 的这种用法。

- 列表 `limit_list` 必须包含 3 个自然数元素，例如 `GO 1 TO 3 STEPS FROM "A" OVER * LIMIT [1,2,4]`。
- `LIMIT [1,2,4]` 中的 `1` 表示系统在第一步时自动选择 1 条边继续遍历，`2` 表示在第二步时选择 2 条边继续遍历，`4` 表示在第三步时选择 4 条边继续遍历。
- 因为 `GO 1 TO 3 STEPS` 表示返回第一到第三步的所有遍历结果，因此下图中所有红色边和它们的原点与目的点都会被这条 `GO` 语句匹配上，而黄色边表示 `GO` 语句遍历时没有选择的路径。如果不是 `GO 1 TO 3 STEPS` 而是 `GO 3 STEPS`，则只会匹配上第三步的红色边和它们两端的点。



在 basketballplayer 数据集中的执行示例如下：

```
nebula> GO 3 STEPS FROM "player100" \
        OVER * \
        YIELD properties($$).name AS NAME, properties($$).age AS Age \
        LIMIT [3,3,3];
+-----------------+----------+
| NAME            | Age      |
+-----------------+----------+
| "Tony Parker"   | 36       |
| "Manu Ginobili" | 41       |
| "Spurs"         | __NULL__ |
+-----------------+----------+

nebula> GO 3 STEPS FROM "player102" OVER * BIDIRECT\
        YIELD dst(edge) \
        LIMIT [rand32(5),rand32(5),rand32(5)];
+-------------+
| dst(EDGE)   |
+-------------+
| "player100" |
| "player100" |
+-------------+
```

openCypher 兼容语句中的 LIMIT

在 `MATCH` 等 openCypher 兼容语句中使用 LIMIT 不需要加管道符。语法和说明如下：

```
... [SKIP <offset>] [LIMIT <number_rows>];
```

参数	说明
`offset`	偏移量，即定义从哪一行开始返回。索引从 `0` 开始。默认值为 `0`，表示从第一行开始返回。
`number_rows`	返回的总行数量。

`offset` 和 `number_rows` 可以使用表达式，但是表达式的结果必须是非负整数。

> Note
>
> 两个整数组成的分数表达式会自动向下取整。例如 `8/6` 向下取整为 1。

单独使用 LIMIT

`LIMIT` 可以单独使用，返回指定数量的结果。

```
nebula> MATCH (v:player) RETURN v.player.name AS Name, v.player.age AS Age \
        ORDER BY Age LIMIT 5;
+-------------------------+-----+
| Name                    | Age |
+-------------------------+-----+
| "Luka Doncic"           | 20  |
| "Ben Simmons"           | 22  |
| "Kristaps Porzingis"    | 23  |
| "Giannis Antetokounmpo" | 24  |
| "Kyle Anderson"         | 25  |
+-------------------------+-----+
```

单独使用 SKIP

`SKIP` 可以单独使用，用于设置偏移量，返回指定位置之后的数据。

```
nebula> MATCH (v:player{name:"Tim Duncan"}) --> (v2) \
        RETURN v2.player.name AS Name, v2.player.age AS Age \
        ORDER BY Age DESC SKIP 1;
+-----------------+-----+
| Name            | Age |
+-----------------+-----+
| "Manu Ginobili" | 41  |
| "Tony Parker"   | 36  |
+-----------------+-----+

nebula> MATCH (v:player{name:"Tim Duncan"}) --> (v2) \
        RETURN v2.player.name AS Name, v2.player.age AS Age \
        ORDER BY Age DESC SKIP 1+1;
+---------------+-----+
| Name          | Age |
+---------------+-----+
| "Tony Parker" | 36  |
+---------------+-----+
```

同时使用 SKIP 与 LIMIT

同时使用 `SKIP` 与 `LIMIT` 可以返回从指定位置开始的指定数量的数据。

```
nebula> MATCH (v:player{name:"Tim Duncan"}) --> (v2) \
        RETURN v2.player.name AS Name, v2.player.age AS Age \
        ORDER BY Age DESC SKIP 1 LIMIT 1;
+-----------------+-----+
| Name            | Age |
+-----------------+-----+
| "Manu Ginobili" | 41  |
+-----------------+-----+
```

最后更新: 2025年8月4日

4.6.3 SAMPLE

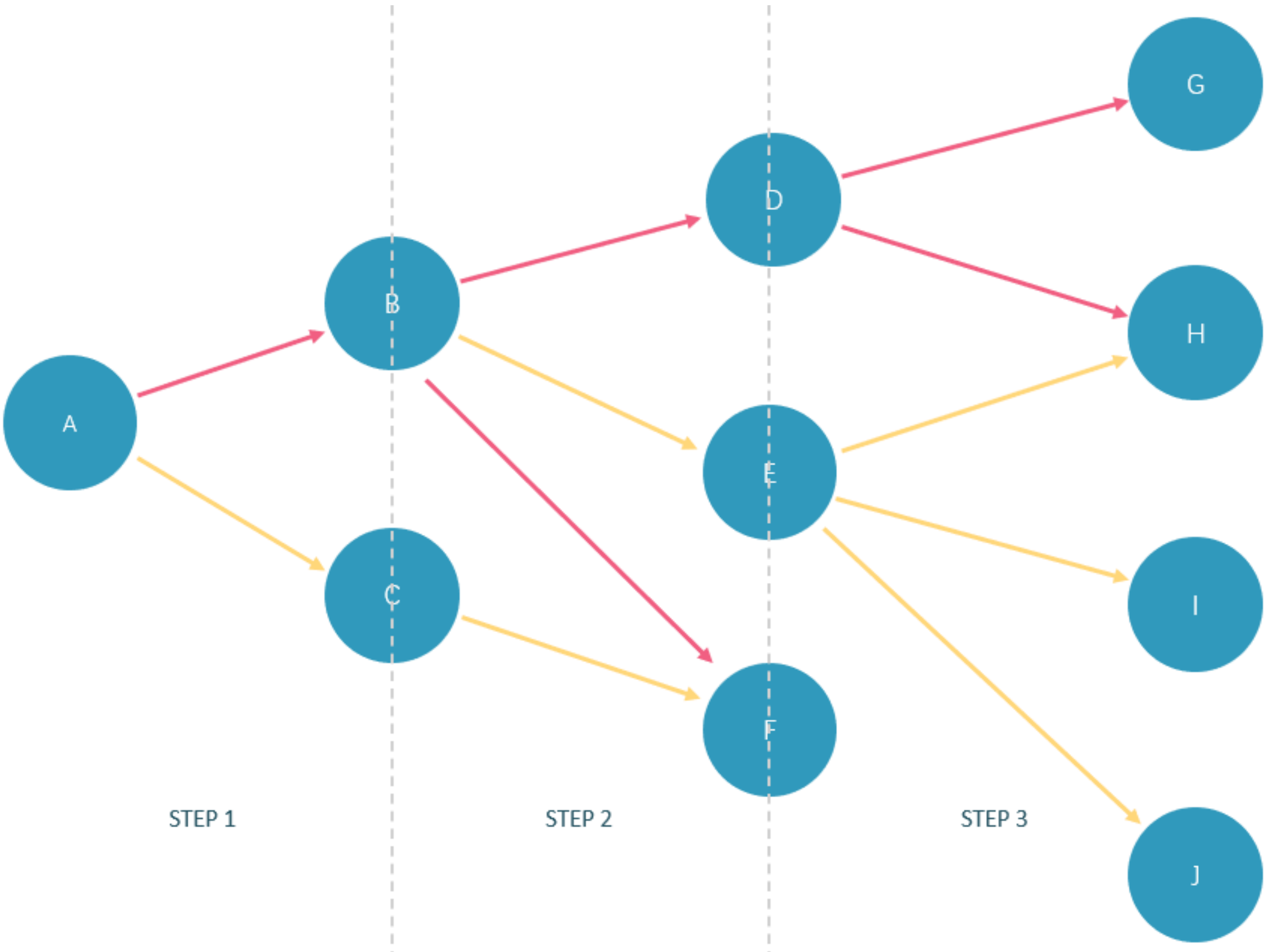
`SAMPLE` 子句用于在结果集中均匀取样并返回指定数量的数据。

`SAMPLE` 仅能在 `GO` 语句中使用，语法如下：

```
<go_statement> SAMPLE <sample_list>;
```

`sample_list` 是一个列表，列表中的元素必须为自然数，且元素数量必须与 `GO` 语句中的 `STEPS` 的最大数相同。下文以 `GO 1 TO 3 STEPS FROM "A" OVER * SAMPLE <sample_list>` 为例详细介绍 `SAMPLE` 的用法。

- 列表 `sample_list` 必须包含 3 个自然数元素，例如 `GO 1 TO 3 STEPS FROM "A" OVER * SAMPLE [1,2,4]`。
- `SAMPLE [1,2,4]` 中的 `1` 表示系统在第一步时自动选择 1 条边继续遍历，`2` 表示在第二步时选择 2 条边继续遍历，`4` 表示在第三步时选择 4 条边继续遍历。如果某一步没有匹配的边或者匹配到的边数量小于指定数量，则按实际数量返回。
- 因为 `GO 1 TO 3 STEPS` 表示返回第一到第三步的所有遍历结果，因此下图中所有红色边和它们的原点与目的点都会被这条 `GO` 语句匹配上，而黄色边表示 `GO` 语句遍历时没有选择的路径。如果不是 `GO 1 TO 3 STEPS` 而是 `GO 3 STEPS`，则只会匹配上第三步的红色边和它们两端的点。



在 basketballplayer 数据集中的执行示例如下：

```
nebula> GO 3 STEPS FROM "player100" \
        OVER * \
        YIELD properties($$).name AS NAME, properties($$).age AS Age \
        SAMPLE [1,2,3];
+-----------------+----------+
| NAME            | Age      |
+-----------------+----------+
| "Tony Parker"   | 36       |
| "Manu Ginobili" | 41       |
| "Spurs"         | __NULL__ |
```

```
+-----------------+----------+

nebula> GO 1 TO 3 STEPS FROM "player100" \
        OVER * \
        YIELD properties($$).name AS NAME, properties($$).age AS Age \
        SAMPLE [2,2,2];
+-----------------+----------+
| NAME            | Age      |
+-----------------+----------+
| "Manu Ginobili" | 41       |
| "Spurs"         | __NULL__ |
| "Tim Duncan"    | 42       |
| "Spurs"         | __NULL__ |
| "Manu Ginobili" | 41       |
| "Spurs"         | __NULL__ |
+-----------------+----------+
```

最后更新: 2025年8月4日

4.6.4 ORDER BY

`ORDER BY` 子句指定输出结果的排序规则。

- 在原生 nGQL 中，必须在 `YIELD` 子句之后使用管道符（|）和 `ORDER BY` 子句。
- 在 openCypher 方式中，不允许使用管道符。在 `RETURN` 子句之后使用 `ORDER BY` 子句。

排序规则分为如下两种：

- `ASC`（默认）: 升序。
- `DESC`: 降序。

原生 **nGQL** 语法

```
<YIELD clause>
| ORDER BY <expression> [ASC | DESC] [, <expression> [ASC | DESC] ...];
```

> Compatibility
>
> 原生 nGQL 语法中，`ORDER BY` 命令后必须使用引用符 `$-.`。但在 2.5.0 之前的版本中不需要。

示例

```
nebula> FETCH PROP ON player "player100", "player101", "player102", "player103" \
        YIELD properties(vertex).age AS age, properties(vertex).name AS name \
        | ORDER BY $-.age ASC, $-.name DESC;
+-----+---------------------+
| age | name                |
+-----+---------------------+
| 32  | "Rudy Gay"          |
| 33  | "LaMarcus Aldridge" |
| 36  | "Tony Parker"       |
| 42  | "Tim Duncan"        |
+-----+---------------------+

nebula> $var = GO FROM "player100" OVER follow \
        YIELD dst(edge) AS dst; \
        ORDER BY $var.dst DESC;
+-------------+
| dst         |
+-------------+
| "player125" |
| "player101" |
+-------------+
```

OpenCypher 方式语法

```
<RETURN clause>
ORDER BY <expression> [ASC | DESC] [, <expression> [ASC | DESC] ...];
```

示例

```
nebula> MATCH (v:player) RETURN v.player.name AS Name, v.player.age AS Age  \
        ORDER BY Name DESC;
+-----------------+-----+
| Name            | Age |
+-----------------+-----+
| "Yao Ming"      | 38  |
| "Vince Carter"  | 42  |
| "Tracy McGrady" | 39  |
| "Tony Parker"   | 36  |
| "Tim Duncan"    | 42  |
+-----------------+-----+
...

# 首先以年龄排序，如果年龄相同，再以姓名排序。
nebula> MATCH (v:player) RETURN v.player.age AS Age, v.player.name AS Name  \
        ORDER BY Age DESC, Name ASC;
+-----+-------------------+
| Age | Name              |
+-----+-------------------+
```

```
| 47  | "Shaquille O'Neal" |
| 46  | "Grant Hill"       |
| 45  | "Jason Kidd"       |
| 45  | "Steve Nash"       |
+-----+-------------------+
...
```

NULL 值的排序

升序排列时，会在输出的最后列出 NULL 值，降序排列时，会在输出的开头列出 NULL 值。

```
nebula> MATCH (v:player{name:"Tim Duncan"}) --> (v2) \
        RETURN v2.player.name AS Name, v2.player.age AS Age  \
        ORDER BY Age;
+-----------------+----------+
| Name            | Age      |
+-----------------+----------+
| "Tony Parker"   | 36       |
| "Manu Ginobili" | 41       |
| __NULL__        | __NULL__ |
+-----------------+----------+

nebula> MATCH (v:player{name:"Tim Duncan"}) --> (v2) \
        RETURN v2.player.name AS Name, v2.player.age AS Age  \
        ORDER BY Age DESC;
+-----------------+----------+
| Name            | Age      |
+-----------------+----------+
| __NULL__        | __NULL__ |
| "Manu Ginobili" | 41       |
| "Tony Parker"   | 36       |
+-----------------+----------+
```

最后更新: 2025年8月4日

4.6.5 RETURN

`RETURN` 子句定义了 nGQL 查询的输出结果。如果需要返回多个字段，用英文逗号（,）分隔。

`RETURN` 可以引导子句或语句：

- `RETURN` 子句可以用于 nGQL 中的 openCypher 方式语句中，例如 `MATCH` 或 `UNWIND`。
- `RETURN` 可以单独使用，输出表达式的结果。

openCypher 兼容性

本文操作仅适用于 nGQL 中的 openCypher 方式。关于原生 nGQL 如何定义输出结果，请参见 `YIELD`。

`RETURN` 不支持如下 openCypher 功能：

- 使用不在英文字母表中的字符作为变量名。例如：

```
MATCH (`点 1`:player) \
RETURN `点 1`;
```

- 设置一个模式，并返回该模式匹配的所有元素。例如：

```
MATCH (v:player) \
RETURN (v)-[e]->(v2);
```

历史版本兼容性

- 在 nGQL 1.x 中，`RETURN` 适用于原生 nGQL，语法为 `RETURN <var_ref> IF <var_ref> IS NOT NULL`。
- 从 nGQL 2.0 开始，`RETURN` 不适用于原生 nGQL。

Map 顺序说明

`RETURN` 返回 Map 时，Key 的顺序是未定义的。

```
nebula> RETURN {age: 32, name: "Marco Belinelli"};
+------------------------------------+
| {age:32,name:"Marco Belinelli"}    |
+------------------------------------+
| {age: 32, name: "Marco Belinelli"} |
+------------------------------------+

nebula> RETURN {zage: 32, name: "Marco Belinelli"};
+-------------------------------------+
| {zage:32,name:"Marco Belinelli"}    |
+-------------------------------------+
| {name: "Marco Belinelli", zage: 32} |
+-------------------------------------+
```

返回点或边

使用 `RETURN {<vertex_name> | <edge_name>}` 返回点或边的所有信息。

```
// 返回点
nebula> MATCH (v:player) \
        RETURN v;
+---------------------------------------------------------------+
| v                                                             |
+---------------------------------------------------------------+
| ("player104" :player{age: 32, name: "Marco Belinelli"})       |
| ("player107" :player{age: 32, name: "Aron Baynes"})           |
| ("player116" :player{age: 34, name: "LeBron James"})          |
| ("player120" :player{age: 29, name: "James Harden"})          |
| ("player125" :player{age: 41, name: "Manu Ginobili"})         |
+---------------------------------------------------------------+
...
```

```
// 返回边
nebula> MATCH (v:player)-[e]->() \
        RETURN e;
+------------------------------------------------------------------------------+
| e                                                                            |
+------------------------------------------------------------------------------+
| [:follow "player104"->"player100" @0 {degree: 55}]                           |
| [:follow "player104"->"player101" @0 {degree: 50}]                           |
| [:follow "player104"->"player105" @0 {degree: 60}]                           |
| [:serve "player104"->"team200" @0 {end_year: 2009, start_year: 2007}]        |
| [:serve "player104"->"team208" @0 {end_year: 2016, start_year: 2015}]        |
+------------------------------------------------------------------------------+
...
```

返回点 ID

使用 `id()` 函数返回点 ID。

```
nebula> MATCH (v:player{name:"Tim Duncan"}) \
        RETURN id(v);
+-------------+
| id(v)       |
+-------------+
| "player100" |
+-------------+
```

返回 Tag

使用 `labels()` 函数返回点上的 Tag 列表。

```
nebula> MATCH (v:player{name:"Tim Duncan"}) \
        RETURN labels(v);
+------------+
| labels(v)  |
+------------+
| ["player"] |
+------------+
```

返回列表 `labels(v)` 中的第 N 个元素，可以使用 `labels(v)[n-1]`。例如下面示例使用 `labels(v)[0]` 检索第一个元素。

```
nebula> MATCH (v:player{name:"Tim Duncan"}) \
        RETURN labels(v)[0];
+--------------+
| labels(v)[0] |
+--------------+
| "player"     |
+--------------+
```

返回属性

返回点的属性时，必需指定属性所属的 Tag，因为点可以有多个 Tag，并且同一个属性名可以在不同的 Tag 上出现。

支持指定点的 Tag 返回该 Tag 的所有属性；也支持指定点的 Tag 和某个属性名，返回该 Tag 的指定属性。

```
// 返回点的属性
nebula> MATCH (v:player) \
        RETURN v.player, v.player.name, v.player.age \
        LIMIT 3;
+--------------------------------------+---------------------+--------------+
| v.player                             | v.player.name       | v.player.age |
+--------------------------------------+---------------------+--------------+
| {age: 33, name: "LaMarcus Aldridge"} | "LaMarcus Aldridge" | 33           |
| {age: 25, name: "Kyle Anderson"}     | "Kyle Anderson"     | 25           |
| {age: 40, name: "Kobe Bryant"}       | "Kobe Bryant"       | 40           |
+--------------------------------------+---------------------+--------------+
```

返回边的属性时，无需指定属性所属的 Edge type，因为边只能有一个 Edge type。

```
// 返回边的属性
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->() \
        RETURN e.start_year, e.degree \
+--------------+----------+
| e.start_year | e.degree |
+--------------+----------+
| __NULL__     | 95       |
| __NULL__     | 95       |
| 1997         | __NULL__ |
+--------------+----------+
```

使用 `properties()` 函数返回点或边的所有属性。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[]->(v2) \
        RETURN properties(v2);
+----------------------------------+
| properties(v2)                   |
+----------------------------------+
| {name: "Spurs"}                  |
| {age: 36, name: "Tony Parker"}   |
| {age: 41, name: "Manu Ginobili"} |
+----------------------------------+
```

返回 **Edge type**

使用 `type()` 函数返回匹配的 Edge type。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[e]->() \
        RETURN DISTINCT type(e);
+----------+
| type(e)  |
+----------+
| "serve"  |
| "follow" |
+----------+
```

返回路径

使用 `RETURN <path_name>` 返回匹配路径的所有信息。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[*3]->() \
        RETURN p;
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
+
|
p
|
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
+
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})-[:follow@0 {degree: 90}]->("player102" :player{age: 33, name: "LaMarcus Aldridge"})-[:serve@0 {end_year: 2019, start_year: 2015}]->("team204" :team{name: "Spurs"})>        |
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})-[:follow@0 {degree: 90}]->("player102" :player{age: 33, name: "LaMarcus Aldridge"})-[:serve@0 {end_year: 2015, start_year: 2006}]->("team203" :team{name: "Trail Blazers"})> |
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})-[:follow@0 {degree: 90}]->("player102" :player{age: 33, name: "LaMarcus Aldridge"})-[:follow@0 {degree: 75}]->("player101" :player{age: 36, name: "Tony Parker"})>          |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
+
...
```

返回路径中的点

使用 `nodes()` 函数返回路径中的所有点。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[]->(v2) \
        RETURN nodes(p);
+-------------------------------------------------------------------------------------------------------------+
| nodes(p)                                                                                                    |
+-------------------------------------------------------------------------------------------------------------+
| [("player100" :player{age: 42, name: "Tim Duncan"}), ("team204" :team{name: "Spurs"})]                     |
| [("player100" :player{age: 42, name: "Tim Duncan"}), ("player101" :player{age: 36, name: "Tony Parker"})]   |
| [("player100" :player{age: 42, name: "Tim Duncan"}), ("player125" :player{age: 41, name: "Manu Ginobili"})] |
+-------------------------------------------------------------------------------------------------------------+
```

返回路径中的边

使用 `relationships()` 函数返回路径中的所有边。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[]->(v2) \
        RETURN relationships(p);
+-------------------------------------------------------------------------+
| relationships(p)                                                        |
+-------------------------------------------------------------------------+
| [[:serve "player100"->"team204" @0 {end_year: 2016, start_year: 1997}]] |
| [[:follow "player100"->"player101" @0 {degree: 95}]]                    |
| [[:follow "player100"->"player125" @0 {degree: 95}]]                    |
+-------------------------------------------------------------------------+
```

返回路径长度

使用 `length()` 函数检索路径的长度。

```
nebula> MATCH p=(v:player{name:"Tim Duncan"})-[*..2]->(v2) \
        RETURN p AS Paths, length(p) AS Length;
+------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
+--------+
|
Paths
| Length |
+------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
+--------+
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:serve@0 {end_year: 2016, start_year: 1997}]->("team204" :team{name:
"Spurs"})>                                                                                 | 1      |
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony
Parker"})>                                                                                   | 1      |
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player125" :player{age: 41, name: "Manu
Ginobili"})>                                                                                 | 1      |
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})-[:serve@0 {end_year: 2018, start_year: 1999}]-
>("team204" :team{name: "Spurs"})>     | 2      |
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})-[:serve@0 {end_year: 2019, start_year: 2018}]-
>("team215" :team{name: "Hornets"})>   | 2      |
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})-[:follow@0 {degree: 95}]->("player100" :player{age: 42,
name: "Tim Duncan"})>        | 2      |
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})-[:follow@0 {degree: 90}]->("player102" :player{age: 33,
name: "LaMarcus Aldridge"})> | 2      |
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})-[:follow@0 {degree: 95}]->("player125" :player{age: 41,
name: "Manu Ginobili"})>     | 2      |
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player125" :player{age: 41, name: "Manu Ginobili"})-[:serve@0 {end_year: 2018, start_year: 2002}]-
>("team204" :team{name: "Spurs"})>   | 2      |
| <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player125" :player{age: 41, name: "Manu Ginobili"})-[:follow@0 {degree: 90}]->("player100" :player{age: 42,
name: "Tim Duncan"})>      | 2      |
+------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
+--------+
```

返回所有元素

使用星号（*）返回匹配模式中的所有元素。

```
nebula> MATCH (v:player{name:"Tim Duncan"}) \
        RETURN *;
+----------------------------------------------------+
| v                                                  |
+----------------------------------------------------+
| ("player100" :player{age: 42, name: "Tim Duncan"}) |
+----------------------------------------------------+

nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->(v2) \
        RETURN *;
+----------------------------------------------------+-----------------------------------------------------------------------+-------------------------------------------------------+
| v                                                  | e                                                                     | v2                                                    |
+----------------------------------------------------+-----------------------------------------------------------------------+-------------------------------------------------------+
| ("player100" :player{age: 42, name: "Tim Duncan"}) | [:follow "player100"->"player101" @0 {degree: 95}]                    | ("player101" :player{age: 36, name: "Tony Parker"})   |
| ("player100" :player{age: 42, name: "Tim Duncan"}) | [:follow "player100"->"player125" @0 {degree: 95}]                    | ("player125" :player{age: 41, name: "Manu Ginobili"}) |
| ("player100" :player{age: 42, name: "Tim Duncan"}) | [:serve "player100"->"team204" @0 {end_year: 2016, start_year: 1997}] | ("team204" :team{name: "Spurs"})                      |
+----------------------------------------------------+-----------------------------------------------------------------------+-------------------------------------------------------+
```

重命名字段

使用语法 `AS <alias>` 重命名输出结果中的字段。

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[:serve]->(v2) \
        RETURN v2.team.name AS Team;
+---------+
| Team    |
+---------+
| "Spurs" |
+---------+

nebula> RETURN "Amber" AS Name;
+---------+
| Name    |
+---------+
| "Amber" |
+---------+
```

返回不存在的属性

如果匹配的结果中，某个属性不存在，会返回 `NULL`。

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->(v2) \
        RETURN v2.player.name, type(e), v2.player.age;
+-----------------+----------+---------------+
| v2.player.name  | type(e)  | v2.player.age |
+-----------------+----------+---------------+
| "Manu Ginobili" | "follow" | 41            |
| __NULL__        | "serve"  | __NULL__      |
| "Tony Parker"   | "follow" | 36            |
+-----------------+----------+---------------+
```

返回表达式结果

`RETURN` 语句可以返回字面量、函数或谓词等表达式的结果。

```
nebula> MATCH (v:player{name:"Tony Parker"})-->(v2:player) \
        RETURN DISTINCT v2.player.name, "Hello"+" graphs!", v2.player.age > 35;
+---------------------+---------------------+--------------------+
| v2.player.name      | ("Hello"+" graphs!") | (v2.player.age>35) |
+---------------------+---------------------+--------------------+
| "LaMarcus Aldridge" | "Hello graphs!"     | false              |
| "Tim Duncan"        | "Hello graphs!"     | true               |
| "Manu Ginobili"     | "Hello graphs!"     | true               |
+---------------------+---------------------+--------------------+

nebula> RETURN 1+1;
+-------+
| (1+1) |
+-------+
| 2     |
+-------+

nebula> RETURN 1- -1;
+----------+
| (1--(1)) |
+----------+
| 2        |
+----------+

nebula> RETURN 3 > 1;
+-------+
| (3>1) |
+-------+
| true  |
+-------+

nebula> RETURN 1+1, rand32(1, 5);
+-------+-------------+
| (1+1) | rand32(1,5) |
+-------+-------------+
| 2     | 1           |
+-------+-------------+
```

返回唯一字段

使用 `DISTINCT` 可以删除结果集中的重复字段。

```
# 未使用 DISTINCT。
nebula> MATCH (v:player{name:"Tony Parker"})--(v2:player) \
        RETURN v2.player.name, v2.player.age;
+---------------------+--------+
| v2.name             | v2.age |
+---------------------+--------+
| "Tim Duncan"        | 42     |
| "LaMarcus Aldridge" | 33     |
| "Marco Belinelli"   | 32     |
| "Boris Diaw"        | 36     |
| "Dejounte Murray"   | 29     |
| "Tim Duncan"        | 42     |
| "LaMarcus Aldridge" | 33     |
| "Manu Ginobili"     | 41     |
+---------------------+--------+

# 使用 DISTINCT。
nebula> MATCH (v:player{name:"Tony Parker"})--(v2:player) \
        RETURN DISTINCT v2.player.name, v2.player.age;
+---------------------+--------+
| v2.name             | v2.age |
+---------------------+--------+
| "Tim Duncan"        | 42     |
| "LaMarcus Aldridge" | 33     |
| "Marco Belinelli"   | 32     |
| "Boris Diaw"        | 36     |
| "Dejounte Murray"   | 29     |
| "Manu Ginobili"     | 41     |
+---------------------+--------+
```

最后更新: 2025年8月4日

4.6.6 TTL

TTL（Time To Live）是一个定义数据生存期的机制。当数据达到其预设的生存时间后，它会被自动从图数据库中删除。这种功能尤其适合于只需要暂时存储的数据，例如临时会话或缓存数据。

openCypher 兼容性

本文操作仅适用于原生 nGQL。

注意事项

- 不能修改带有 TTL 选项的属性的 Schema。

- TTL 和 INDEX 共存问题：
- 如果一个 Tag/Edge type 的其中一属性已有 INDEX，则不能为其设置 TTL，也不能为该 Tag 的其他属性设置 TTL。
- 如果已有 TTL，可以再添加 INDEX。

TTL 选项

nGQL 支持的 TTL 选项如下。

选项	说明
`ttl_col`	指定一个现有的要设置存活时间的属性。属性的数据类型必须是 `int` 或者 `timestamp`。
`ttl_duration`	指定时间戳差值，默认单位：秒。时间戳差值必须为 64 位非负整数。属性值和时间戳差值之和如果小于当前时间戳，属性就会过期。如果 `ttl_duration` 为 `0`，属性永不过期。 可在配置文件 `nebula-storaged.conf`（默认路径 `/usr/local/nightly/etc/`）中设置 `ttl_use_ms` 为 `true` 将默认单位设为毫秒。

> Warning
>
> - 在设置 `ttl_use_ms` 为 `true` 前，请确保没有为属性设置 TTL，否则会因为过期时间缩短，导致数据被错误地删除。
> - 在设置 `ttl_use_ms` 为 `true` 后，即设置 `ttl_duration` 的默认单位为毫秒后，`ttl_col` 的默认单位仍然为秒，它的数据类型必须是 `int`，并且需要手动转换属性值为毫秒。例如设置 `ttl_col` 为 `a`，则需要将 `a` 的值转换为毫秒，如当 `a` 的值为 `now()`，则需要将 `a` 的值设置为 `now() * 1000`。

使用 TTL 选项

在使用 TTL 功能之前，必须先创建一个时间戳或整数属性，并在 TTL 选项中指定了它。数据库不会自动为您创建或管理这个时间戳属性。

在插入时间戳或整数属性的值时，建议使用 `now()` 函数或者当前时间戳来代表当前时间。

TAG 或 EDGE TYPE 已存在

如果 Tag 和 Edge type 已经创建，请使用 `ALTER` 语句更新 Tag 或 Edge type。

```
# 创建 Tag。
nebula> CREATE TAG IF NOT EXISTS t1 (a timestamp);

# ALTER 修改 Tag，添加 TTL 选项。
nebula> ALTER TAG t1 TTL_COL = "a", TTL_DURATION = 5;

# 插入点，插入后 5 秒过期。
nebula> INSERT VERTEX t1(a) VALUES "101":(now());
```

TAG 或 EDGE TYPE 不存在

创建 Tag 或 Edge type 时可以同时设置 TTL 选项。详情请参见 CREATE TAG 和 CREATE EDGE。

```
# 创建 Tag 并设置 TTL 选项。
nebula> CREATE TAG IF NOT EXISTS t2(a int, b int, c string) TTL_DURATION= 100, TTL_COL = "a";
```

```
# 插入点。过期时间戳为 1648197238（1648197138 + 100）。
nebula> INSERT VERTEX t2(a, b, c) VALUES "102":(1648197138, 30, "Hello");
```

属性过期

Caution

- 当为一个 Tag 或 Edge type 的属性设置 TTL 并该属性的值为 `NULL` 时，TTL 功能不会生效，即该属性永不过期。
- 如果为一个 Tag 或 Edge type 新增默认值为 `now()` 的属性并且该属性设置了 TTL，该 Tag 或 Edge type 相关的历史数据不会过期，因为历史数据的该属性值为当前时间戳。

点属性过期

点属性过期有如下影响：

- 如果一个点仅有一个 Tag，点上的一个属性过期，点也会过期。
- 如果一个点有多个 Tag，点上的一个属性过期，和该属性相同 Tag 的其他属性也会过期，但是点不会过期，点上其他 Tag 的属性保持不变。

边属性过期

因为一条边仅有一个 Edge type，边上的一个属性过期，边也会过期。

过期处理

属性过期后，对应的过期数据仍然存储在硬盘上，但是查询时会过滤过期数据。

NebulaGraph 自动删除过期数据后，会在下一次 Compaction 过程中回收硬盘空间。

Note

如果关闭 TTL 选项，上一次 Compaction 之后的过期数据将可以被查询到。

删除存活时间

删除存活时间可以使用如下几种方法：

- 删除设置存活时间的属性。

```
nebula> ALTER TAG t1 DROP (a);
```

- 设置 `ttl_col` 为空字符串。

```
nebula> ALTER TAG t1 TTL_COL = "";
```

- 设置 `ttl_duration` 为 `0`。本操作可以保留 TTL 选项，属性永不过期，且属性的 Schema 无法修改。

```
nebula> ALTER TAG t1 TTL_DURATION = 0;
```

最后更新: 2025年8月4日

4.6.7 WHERE

`WHERE` 子句可以根据条件过滤输出结果。

`WHERE` 子句通常用于如下查询：

- 原生 nGQL，例如 `GO` 和 `LOOKUP` 语句。
- openCypher 方式，例如 `MATCH` 和 `WITH` 语句。

openCypher 兼容性

过滤 Rank 是原生 nGQL 功能。如需在 openCypher 兼容语句中直接获取 Rank 值，可以使用 rank() 函数，例如 `MATCH (:player)-[e:follow]->() RETURN rank(e);`。

基础用法

> **Note**
>
> 下文示例中的 `$$`、`$^` 等是引用符号，详情请参见引用符。

用布尔运算符定义条件

在 `WHERE` 子句中使用布尔运算符 `NOT`、`AND`、`OR` 和 `XOR` 定义条件。关于运算符的优先级，请参见运算符优先级。

```
nebula> MATCH (v:player) \
        WHERE v.player.name == "Tim Duncan" \
        XOR (v.player.age < 30 AND v.player.name == "Yao Ming") \
        OR NOT (v.player.name == "Yao Ming" OR v.player.name == "Tim Duncan") \
        RETURN v.player.name, v.player.age;
+------------------------+-------------+
| v.player.name          | v.player.age |
+------------------------+-------------+
| "Danny Green"          | 31          |
| "Tiago Splitter"       | 34          |
| "David West"           | 38          |
...
```

```
nebula> GO FROM "player100" \
        OVER follow \
        WHERE properties(edge).degree > 90 \
        OR properties($$).age != 33 \
        AND properties($$).name != "Tony Parker" \
        YIELD properties($$);
+----------------------------------+
| properties($$)                   |
+----------------------------------+
| {age: 41, name: "Manu Ginobili"} |
+----------------------------------+
```

过滤属性

在 `WHERE` 子句中使用点或边的属性定义条件。

- 过滤点属性：

```
nebula> MATCH (v:player)-[e]->(v2) \
        WHERE v2.player.age < 25 \
        RETURN v2.player.name, v2.player.age;
+----------------------+---------------+
| v2.player.name       | v2.player.age |
+----------------------+---------------+
| "Ben Simmons"        | 22            |
| "Luka Doncic"        | 20            |
| "Kristaps Porzingis" | 23            |
+----------------------+---------------+
```

```
nebula> GO FROM "player100" OVER follow \
        WHERE $^.player.age >= 42 \
        YIELD dst(edge);
+-------------+
| dst(EDGE)   |
+-------------+
| "player101" |
| "player125" |
+-------------+
```

- 过滤边属性：

```
nebula> MATCH (v:player)-[e]->() \
        WHERE e.start_year < 2000 \
        RETURN DISTINCT v.player.name, v.player.age;
+--------------------+--------------+
| v.player.name      | v.player.age |
+--------------------+--------------+
| "Tony Parker"      | 36           |
| "Tim Duncan"       | 42           |
| "Grant Hill"       | 46           |
...
```

```
nebula> GO FROM "player100" OVER follow \
        WHERE follow.degree > 90 \
        YIELD dst(edge);
+-------------+
| dst(EDGE)   |
+-------------+
| "player101" |
| "player125" |
+-------------+
```

过滤动态计算属性

```
nebula> MATCH (v:player) \
        WHERE v[toLower("AGE")] < 21 \
        RETURN v.player.name, v.player.age;
+---------------+-------+
| v.name        | v.age |
+---------------+-------+
| "Luka Doncic" | 20    |
+---------------+-------+
```

过滤现存属性

```
nebula> MATCH (v:player) \
        WHERE exists(v.player.age) \
        RETURN v.player.name, v.player.age;
+-------------------------+--------------+
| v.player.name           | v.player.age |
+-------------------------+--------------+
| "Danny Green"           | 31           |
| "Tiago Splitter"        | 34           |
| "David West"            | 38           |
...
```

过滤 **RANK**

在 nGQL 中，如果多个边拥有相同的起始点、目的点和属性，则它们的唯一区别是 rank 值。在 `WHERE` 子句中可以使用 rank 过滤边。

```
# 创建测试数据。
nebula> CREATE SPACE IF NOT EXISTS test (vid_type=FIXED_STRING(30));
nebula> USE test;
```

```
nebula> CREATE EDGE IF NOT EXISTS e1(p1 int);
nebula> CREATE TAG IF NOT EXISTS person(p1 int);
nebula> INSERT VERTEX person(p1) VALUES "1":(1);
nebula> INSERT VERTEX person(p1) VALUES "2":(2);
nebula> INSERT EDGE e1(p1) VALUES "1"->"2"@0:(10);
nebula> INSERT EDGE e1(p1) VALUES "1"->"2"@1:(11);
nebula> INSERT EDGE e1(p1) VALUES "1"->"2"@2:(12);
nebula> INSERT EDGE e1(p1) VALUES "1"->"2"@3:(13);
nebula> INSERT EDGE e1(p1) VALUES "1"->"2"@4:(14);
nebula> INSERT EDGE e1(p1) VALUES "1"->"2"@5:(15);
nebula> INSERT EDGE e1(p1) VALUES "1"->"2"@6:(16);

# 通过 rank 过滤边，查找 rank 大于 2 的边。
nebula> GO FROM "1" \
        OVER e1 \
        WHERE rank(edge) > 2 \
        YIELD src(edge), dst(edge), rank(edge) AS Rank, properties(edge).p1 | \
        ORDER BY $-.Rank DESC;
+-----------+-----------+------+---------------------+
| src(EDGE) | dst(EDGE) | Rank | properties(EDGE).p1 |
+-----------+-----------+------+---------------------+
| "1"       | "2"       | 6    | 16                  |
| "1"       | "2"       | 5    | 15                  |
| "1"       | "2"       | 4    | 14                  |
| "1"       | "2"       | 3    | 13                  |
+-----------+-----------+------+---------------------+

# 通过 rank 过滤边，查找 rank 值等于 0 的 follow 边。
nebula> MATCH (v)-[e:follow]->() \
        WHERE rank(e)==0 \
        RETURN *;
+------------------------------------------------------------+-------------------------------------------------------+
| v                                                          | e                                                     |
+------------------------------------------------------------+-------------------------------------------------------+
| ("player142" :player{age: 29, name: "Klay Thompson"})      | [:follow "player142"->"player117" @0 {degree: 90}]  |
| ("player139" :player{age: 34, name: "Marc Gasol"})         | [:follow "player139"->"player138" @0 {degree: 99}]  |
| ("player108" :player{age: 36, name: "Boris Diaw"})         | [:follow "player108"->"player100" @0 {degree: 80}]  |
| ("player108" :player{age: 36, name: "Boris Diaw"})         | [:follow "player108"->"player101" @0 {degree: 80}]  |
...
```

过滤 PATTERN

```
nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->(t) \
        WHERE (v)-[e]->(t:team) \
        RETURN (v)-->();

+-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
+
| (v)-->() = (v)--
>()
|
+-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
+
| [<("player100" :player{age: 42, name: "Tim Duncan"})-[:serve@0 {end_year: 2016, start_year: 1997}]->("team204" :team{name: "Spurs"})>, <("player100" :player{age: 42, name: "Tim Duncan"})-
[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})>, <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player125" :player{age: 41,
name: "Manu Ginobili"})>] |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
+

nebula> MATCH (v:player{name:"Tim Duncan"})-[e]->(t) \
        WHERE NOT (v)-[e]->(t:team) \
        RETURN (v)-->();
+-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
+
| (v)-->() = (v)--
>()
|
+-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
+
| [<("player100" :player{age: 42, name: "Tim Duncan"})-[:serve@0 {end_year: 2016, start_year: 1997}]->("team204" :team{name: "Spurs"})>, <("player100" :player{age: 42, name: "Tim Duncan"})-
[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})>, <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player125" :player{age: 41,
name: "Manu Ginobili"})>] |
| [<("player100" :player{age: 42, name: "Tim Duncan"})-[:serve@0 {end_year: 2016, start_year: 1997}]->("team204" :team{name: "Spurs"})>, <("player100" :player{age: 42, name: "Tim Duncan"})-
[:follow@0 {degree: 95}]->("player101" :player{age: 36, name: "Tony Parker"})>, <("player100" :player{age: 42, name: "Tim Duncan"})-[:follow@0 {degree: 95}]->("player125" :player{age: 41,
name: "Manu Ginobili"})>] |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
+
```

过滤字符串

在 `WHERE` 子句中使用 `STARTS WITH` 、 `ENDS WITH` 或 `CONTAINS` 可以匹配字符串的特定部分。匹配时区分大小写。

`STARTS WITH`

`STARTS WITH` 会从字符串的起始位置开始匹配。

```
# 查询姓名以 T 开头的 player 信息。
nebula> MATCH (v:player) \
```

```
        WHERE v.player.name STARTS WITH "T" \
        RETURN v.player.name, v.player.age;
+------------------+--------------+
| v.player.name    | v.player.age |
+------------------+--------------+
| "Tony Parker"    | 36           |
| "Tiago Splitter" | 34           |
| "Tim Duncan"     | 42           |
| "Tracy McGrady"  | 39           |
+------------------+--------------+
```

如果使用小写 `t` （`STARTS WITH "t"`），会返回空集，因为数据库中没有以小写 `t` 开头的姓名。

```
nebula> MATCH (v:player) \
        WHERE v.player.name STARTS WITH "t" \
        RETURN v.player.name, v.player.age;
+---------------+--------------+
| v.player.name | v.player.age |
+---------------+--------------+
+---------------+--------------+
Empty set (time spent 5080/6474 us)
```

ENDS WITH

`ENDS WITH` 会从字符串的结束位置开始匹配。

```
nebula> MATCH (v:player) \
        WHERE v.player.name ENDS WITH "r" \
        RETURN v.player.name, v.player.age;
+------------------+--------------+
| v.player.name    | v.player.age |
+------------------+--------------+
| "Tony Parker"    | 36           |
| "Tiago Splitter" | 34           |
| "Vince Carter"   | 42           |
+------------------+--------------+
```

CONTAINS

`CONTAINS` 会检查关键字是否匹配字符串的某一部分。

```
nebula> MATCH (v:player) \
        WHERE v.player.name CONTAINS "Pa" \
        RETURN v.player.name, v.player.age;
+---------------+--------------+
| v.player.name | v.player.age |
+---------------+--------------+
| "Paul George" | 28           |
| "Tony Parker" | 36           |
| "Paul Gasol"  | 38           |
| "Chris Paul"  | 33           |
+---------------+--------------+
```

结合 NOT 使用

用户可以结合布尔运算符 `NOT` 一起使用，否定字符串匹配条件。

```
nebula> MATCH (v:player) \
        WHERE NOT v.player.name ENDS WITH "R" \
        RETURN v.player.name, v.player.age;
+-------------------------+--------------+
| v.player.name           | v.player.age |
+-------------------------+--------------+
| "Danny Green"           | 31           |
| "Tiago Splitter"        | 34           |
| "David West"            | 38           |
| "Russell Westbrook"     | 30           |
...
```

过滤列表

匹配列表中的值

使用 `IN` 运算符检查某个值是否在指定列表中。

```
nebula> MATCH (v:player) \
        WHERE v.player.age IN range(20,25) \
        RETURN v.player.name, v.player.age;
+-------------------------+--------------+
| v.player.name           | v.player.age |
+-------------------------+--------------+
```

```
| "Ben Simmons"           | 22           |
| "Giannis Antetokounmpo" | 24           |
| "Kyle Anderson"         | 25           |
| "Joel Embiid"           | 25           |
| "Kristaps Porzingis"    | 23           |
| "Luka Doncic"           | 20           |
+-------------------------+--------------+

nebula> LOOKUP ON player \
        WHERE player.age IN [25,28]  \
        YIELD properties(vertex).name, properties(vertex).age;
+-------------------------+------------------------+
| properties(VERTEX).name | properties(VERTEX).age |
+-------------------------+------------------------+
| "Kyle Anderson"         | 25                     |
| "Damian Lillard"        | 28                     |
| "Joel Embiid"           | 25                     |
| "Paul George"           | 28                     |
| "Ricky Rubio"           | 28                     |
+-------------------------+------------------------+
```

结合 **NOT** 使用

```
nebula> MATCH (v:player) \
        WHERE v.player.age NOT IN range(20,25) \
        RETURN v.player.name AS Name, v.player.age AS Age \
        ORDER BY Age;
+---------------------+-----+
| Name                | Age |
+---------------------+-----+
| "Kyrie Irving"      | 26  |
| "Cory Joseph"       | 27  |
| "Damian Lillard"    | 28  |
| "Paul George"       | 28  |
| "Ricky Rubio"       | 28  |
...
```

最后更新: 2025年8月4日

4.6.8 YIELD

`YIELD` 定义 nGQL 查询的输出结果。

`YIELD` 可以引导子句或语句：

- `YIELD` 子句用于原生 nGQL 语句中，例如 `GO`、`FETCH` 或 `LOOKUP`，必须通过 `YIELD` 子句定义返回结果。
- `YIELD` 语句可以在独立查询或复合查询中使用。

openCypher 兼容性

本文操作仅适用于原生 nGQL。关于 openCypher 方式如何定义输出结果，请参见 `RETURN`。

`YIELD` 在 nGQL 和 openCypher 中有不同的函数：

- 在 openCypher 中，`YIELD` 用于在 `CALL[...YIELD]` 子句中指定过程调用的输出。

> Note
>
> nGQL 不支持 `CALL[...YIELD]`。

- 在 nGQL 中，`YIELD` 和 openCypher 中的 `RETURN` 类似。

> Note
>
> 下文示例中的 `$$`、`$-` 等是引用符号，详情请参见引用属性。

YIELD 子句

语法

```
YIELD [DISTINCT] <col> [AS <alias>] [, <col> [AS <alias>] ...];
```

参数	说明
`DISTINCT`	聚合输出结果，返回去重后的结果集。
`col`	要返回的字段。如果没有为字段设置别名，返回结果中的列名为 `col`。
`alias`	`col` 的别名。使用关键字 `AS` 进行设置，设置后返回结果中的列名为该别名。

使用 **YIELD** 子句

- `GO` 语句中使用 `YIELD`：

```
nebula> GO FROM "player100" OVER follow \
        YIELD properties($$).name AS Friend, properties($$).age AS Age;
+-----------------+-----+
| Friend          | Age |
+-----------------+-----+
| "Tony Parker"   | 36  |
```

```
| "Manu Ginobili" | 41  |
+-----------------+-----+
```

- `FETCH` 语句中使用 `YIELD`：

```
nebula> FETCH PROP ON player "player100" \
        YIELD properties(vertex).name;
+-------------------------+
| properties(VERTEX).name |
+-------------------------+
| "Tim Duncan"            |
+-------------------------+
```

- `LOOKUP` 语句中使用 `YIELD`：

```
nebula> LOOKUP ON player WHERE player.name == "Tony Parker" \
        YIELD properties(vertex).name, properties(vertex).age;
+-------------------------+------------------------+
| properties(VERTEX).name | properties(VERTEX).age |
+-------------------------+------------------------+
| "Tony Parker"           | 36                     |
+-------------------------+------------------------+
```

YIELD 语句

语法

```
YIELD [DISTINCT] <col> [AS <alias>] [, <col> [AS <alias>] ...]
[WHERE <conditions>];
```

参数	说明
`DISTINCT`	聚合输出结果，返回去重后的结果集。
`col`	要按返回的字段。如果没有为字段设置别名，返回结果中的列名为 `col`。
`alias`	`col` 的别名。使用关键字 `AS` 进行设置，设置后返回结果中的列名为该别名。
`conditions`	在 `WHERE` 子句中设置的过滤条件。详情请参见 `WHERE`。

复合查询中使用 YIELD 语句

在复合查询中，`YIELD` 语句可以接收、过滤、修改之前语句的结果集，然后输出。

```
# 查找 player100 关注的 player，并计算他们的平均年龄。
nebula> GO FROM "player100" OVER follow \
        YIELD dst(edge) AS ID \
        | FETCH PROP ON player $-.ID \
        YIELD properties(vertex).age AS Age \
        | YIELD AVG($-.Age) as Avg_age, count(*)as Num_friends;
+---------+-------------+
| Avg_age | Num_friends |
+---------+-------------+
| 38.5    | 2           |
+---------+-------------+
```

```
# 查找 player101 关注的 player，返回 degree 大于 90 的 player。
nebula> $var1 = GO FROM "player101" OVER follow \
        YIELD properties(edge).degree AS Degree, dst(edge) as ID; \
        YIELD $var1.ID AS ID WHERE $var1.Degree > 90;
+-------------+
| ID          |
+-------------+
| "player100" |
| "player125" |
+-------------+

# 查找 player 中年龄大于 30 且小于 32 的点，返回去掉重复属性的值。
nebula> LOOKUP ON player  \
        WHERE player.age < 32 and player.age >30  \
        YIELD DISTINCT properties(vertex).age as v;
+--------+
| v      |
+--------+
| 31     |
+--------+
```

独立使用 YIELD 语句

`YIELD` 可以计算表达式并返回结果。

```
nebula> YIELD rand32(1, 6);
+-------------+
| rand32(1,6) |
+-------------+
| 3           |
+-------------+

nebula> YIELD "Hel" + "\tlo" AS string1, ", World!" AS string2;
+-------------+------------+
| string1     | string2    |
+-------------+------------+
| "Hel    lo" | ", World!" |
+-------------+------------+

nebula> YIELD hash("Tim") % 100;
+-------------------+
| (hash("Tim")%100) |
+-------------------+
| 42                |
+-------------------+

nebula> YIELD \
      CASE 2+3 \
      WHEN 4 THEN 0 \
      WHEN 5 THEN 1 \
      ELSE -1 \
      END \
      AS result;
+--------+
| result |
+--------+
| 1      |
+--------+

nebula> YIELD 1- -1;
+----------+
| (1--(1)) |
+----------+
| 2        |
+----------+
```

最后更新: 2025年8月4日

4.6.9 WITH

`WITH` 子句可以获取并处理查询前半部分的结果，并将处理结果作为输入传递给查询的后半部分。

openCypher 兼容性

本文操作仅适用于 openCypher 方式。

> Note
>
> 在原生 nGQL 中，有与 `WITH` 类似的管道符，但它们的工作方式不同。不要在 openCypher 方式中使用管道符，也不要在原生 nGQL 中使用 `WITH` 子句。

组成复合查询

使用 `WITH` 子句可以组合语句，将一条语句的输出转换为另一条语句的输入。

示例 **1**

1. 匹配一个路径。
2. 通过 `nodes()` 函数将路径上的所有点输出到一个列表。
3. 将列表拆分为行。
4. 去重后返回点的信息。

```
# 查询 Tag 为 player，name 属性为 Tim Duncan 的点的路径，并返回去重后路径中的所有的点信息列表。
nebula> MATCH p=(v:player{name:"Tim Duncan"})--() \
        WITH nodes(p) AS n \
        UNWIND n AS n1 \
        RETURN DISTINCT n1;
+-----------------------------------------------------------+
| n1                                                        |
+-----------------------------------------------------------+
| ("player100" :player{age: 42, name: "Tim Duncan"})        |
| ("player101" :player{age: 36, name: "Tony Parker"})       |
| ("team204" :team{name: "Spurs"})                          |
| ("player102" :player{age: 33, name: "LaMarcus Aldridge"}) |
| ("player125" :player{age: 41, name: "Manu Ginobili"})     |
| ("player104" :player{age: 32, name: "Marco Belinelli"})   |
| ("player144" :player{age: 47, name: "Shaquille O'Neal"})  |
| ("player105" :player{age: 31, name: "Danny Green"})       |
| ("player113" :player{age: 29, name: "Dejounte Murray"})   |
| ("player107" :player{age: 32, name: "Aron Baynes"})       |
| ("player109" :player{age: 34, name: "Tiago Splitter"})    |
| ("player108" :player{age: 36, name: "Boris Diaw"})        |
+-----------------------------------------------------------+
```

示例 **2**

1. 匹配点 ID 为 `player100` 的点。
2. 通过 `labels()` 函数将点的所有 Tag 输出到一个列表。
3. 将列表拆分为行。
4. 返回结果。

```
# 查询 id 为 player100 的点，并返回点的所有 Tag。
nebula> MATCH (v) \
        WHERE id(v)=="player100" \
        WITH labels(v) AS tags_unf \
        UNWIND tags_unf AS tags_f \
        RETURN tags_f;
+----------+
| tags_f   |
+----------+
| "player" |
+----------+
```

过滤聚合查询

`WITH` 可以在聚合查询中作为过滤器使用。

```
# 查询所有起始点和目的点均有 player 点类型的边，并对所有目的点去重，再根据 age 属性升序排列，返回 age 属性值小于 25 的目的点数据。
nebula> MATCH (v:player)-->(v2:player) \
        WITH DISTINCT v2 AS v2, v2.player.age AS Age \
        ORDER BY Age \
        WHERE Age<25 \
        RETURN v2.player.name AS Name, Age;
+----------------------+-----+
| Name                 | Age |
+----------------------+-----+
| "Luka Doncic"        | 20  |
| "Ben Simmons"        | 22  |
| "Kristaps Porzingis" | 23  |
+----------------------+-----+
```

collect() 之前处理输出

在 `collect()` 函数将输出结果转换为列表之前，可以使用 `WITH` 子句排序和限制输出结果。

```
# 查询所有 Tag 为 player 的点数据，并根据 name 属性倒序排列，将前 3 条结果转换为列表返回。
nebula> MATCH (v:player) \
        WITH v.player.name AS Name \
        ORDER BY Name DESC \
        LIMIT 3 \
        RETURN collect(Name);
+-----------------------------------------------+
| collect(Name)                                 |
+-----------------------------------------------+
| ["Yao Ming", "Vince Carter", "Tracy McGrady"] |
+-----------------------------------------------+
```

结合 **RETURN** 语句使用

在 `WITH` 子句中设置别名，并通过 `RETURN` 子句输出结果。

```
# 判断 3 是否存在于 list 数组中。
nebula> WITH [1, 2, 3] AS `list` RETURN 3 IN `list` AS r;
+------+
| r    |
+------+
| true |
+------+

# 返回 3 和 4 的比较结果。
nebula> WITH 4 AS one, 3 AS two RETURN one > two AS result;
+--------+
| result |
+--------+
| true   |
+--------+
```

最后更新: 2025年8月4日

4.6.10 UNWIND

`UNWIND` 语句可以将列表拆分为单独的行，列表中的每个元素为一行。

`UNWIND` 可以作为单独语句或语句中的子句使用。

UNWIND 语句

语法

```
UNWIND <list> AS <alias> <RETURN clause>;
```

示例

- 拆分列表。

```
nebula> UNWIND [1,2,3] AS n RETURN n;
+---+
| n |
+---+
| 1 |
| 2 |
| 3 |
+---+
```

UNWIND 子句

语法

- 原生 nGQL 语句中使用 `UNWIND` 子句。

> **Note**
>
> 在原生 nGQL 语句中使用 `UNWIND` 子句时，需要用在管道符 `|` 之后，并使用 `$-` 引用管道符之前的变量。如果 `UNWIND` 后使用语句或子句，需要使用管道符 `|` 并且使用 `$-` 引用管道符之前的变量。

```
<statement> | UNWIND $-.<var> AS <alias> <|> <clause>;
```

- openCypher 语句中使用 `UNWIND` 子句。

```
<statement> UNWIND <list> AS <alias> <RETURN clause>；
```

示例

- 在 `UNWIND` 子句中使用 `WITH DISTINCT` 可以将列表中的重复项忽略，返回去重后的结果。

> **Note**
>
> 原生 nGQL 语句不支持 `WITH DISTINCT`。

```
// 拆分列表`[1,1,2,2,3,3]`，删除重复行，排序行，将行转换为列表。
nebula> WITH [1,1,2,2,3,3] AS n \
        UNWIND n AS r \
        WITH DISTINCT r AS r \
        ORDER BY r \
        RETURN collect(r);
+------------+
| collect(r) |
+------------+
```

```
| [1, 2, 3]  |
+-----------+
```

- `MATCH` 语句中使用 `UNWIND`。

```
// 将匹配路径上的顶点输出到列表中，拆分列表，删除重复行，将行转换为列表。
nebula> MATCH p=(v:player{name:"Tim Duncan"})--(v2) \
        WITH nodes(p) AS n \
        UNWIND n AS r \
        WITH DISTINCT r AS r \
        RETURN collect(r);
+--------------------------------------------------------------------------------------------------------------------------+
| collect(r)                                                                                                               |
+--------------------------------------------------------------------------------------------------------------------------+
| [("player100" :player{age: 42, name: "Tim Duncan"}), ("player101" :player{age: 36, name: "Tony Parker"}),              |
|("team204" :team{name: "Spurs"}), ("player102" :player{age: 33, name: "LaMarcus Aldridge"}),                            |
|("player125" :player{age: 41, name: "Manu Ginobili"}), ("player104" :player{age: 32, name: "Marco Belinelli"}),        |
|("player144" :player{age: 47, name: "Shaquile O'Neal"}), ("player105" :player{age: 31, name: "Danny Green"}),          |
|("player113" :player{age: 29, name: "Dejounte Murray"}), ("player107" :player{age: 32, name: "Aron Baynes"}),          |
|("player109" :player{age: 34, name: "Tiago Splitter"}), ("player108" :player{age: 36, name: "Boris Diaw"})]            |
+--------------------------------------------------------------------------------------------------------------------------+
```

- `GO` 语句中使用 `UNWIND`。

```
// 在点列表中查询点关联的边。
nebula> YIELD ['player101', 'player100'] AS a | UNWIND $-.a AS  b | GO FROM $-.b OVER follow YIELD edge AS e;
+----------------------------------------------------+
| e                                                  |
+----------------------------------------------------+
| [:follow "player101"->"player100" @0 {degree: 95}] |
| [:follow "player101"->"player102" @0 {degree: 90}] |
| [:follow "player101"->"player125" @0 {degree: 95}] |
| [:follow "player100"->"player101" @0 {degree: 95}] |
| [:follow "player100"->"player125" @0 {degree: 95}] |
+----------------------------------------------------+
```

- `LOOKUP` 语句中使用 `UNWIND`。

```
// 查询年龄大于 46 岁球员的所有属性，去掉重复属性，并将结果转换为行。
nebula> LOOKUP ON player \
        WHERE player.age > 46 \
        YIELD DISTINCT keys(vertex) as p | UNWIND $-.p as a | YIELD $-.a AS a;
+--------+
| a      |
+--------+
| "age"  |
| "name" |
+--------+
```

- `FETCH` 语句中使用 `UNWIND`。

```
// 查询 player101 点的所有 Tag，并将结果转换为行。
nebula> CREATE TAG hero(like string, height int);
        INSERT VERTEX hero(like, height) VALUES "player101":("deep", 182);
        FETCH PROP ON * "player101" \
        YIELD tags(vertex) as t | UNWIND $-.t as a | YIELD $-.a AS a;
+----------+
| a        |
+----------+
| "hero"   |
| "player" |
+----------+
```

- `GET SUBGRAPH` 语句中使用 `UNWIND`。

```
// 查询从点 player100 开始、0~2 跳、serve 类型的出边和入边的子图，并将结果转换为行。
nebula> GET SUBGRAPH 2 STEPS FROM "player100" BOTH serve \
        YIELD edges as e | UNWIND $-.e as a | YIELD $-.a AS a;
+----------------------------------------------+
| a                                            |
+----------------------------------------------+
| [:serve "player100"->"team204" @0 {}]        |
| [:serve "player101"->"team204" @0 {}]        |
| [:serve "player102"->"team204" @0 {}]        |
| [:serve "player103"->"team204" @0 {}]        |
| [:serve "player105"->"team204" @0 {}]        |
| [:serve "player106"->"team204" @0 {}]        |
| [:serve "player107"->"team204" @0 {}]        |
| [:serve "player108"->"team204" @0 {}]        |
| [:serve "player109"->"team204" @0 {}]        |
| [:serve "player110"->"team204" @0 {}]        |
| [:serve "player111"->"team204" @0 {}]        |
| [:serve "player112"->"team204" @0 {}]        |
| [:serve "player113"->"team204" @0 {}]        |
| [:serve "player114"->"team204" @0 {}]        |
| [:serve "player125"->"team204" @0 {}]        |
| [:serve "player138"->"team204" @0 {}]        |
| [:serve "player104"->"team204" @20132015 {}] |
| [:serve "player104"->"team204" @20182019 {}] |
+----------------------------------------------+
```

- `FIND PATH` 语句中使用 `UNWIND`。

```
// 找出 player101 到 team204 延 serve 类型边的最短路径上的所有点，并将结果转换为行。
nebula> FIND SHORTEST PATH FROM "player101" TO "team204" OVER serve \
        YIELD path as p | YIELD nodes($-.p) AS nodes | UNWIND $-.nodes AS a | YIELD $-.a AS a;
+---------------+
| a             |
+---------------+
| ("player101") |
| ("team204")   |
+---------------+
```

最后更新: 2025年8月4日

4.6.11 INNER JOIN

`INNER JOIN` 是一种连接查询。它基于两个表之间的共同列值进行匹配，并返回满足条件的记录。 `INNER JOIN` 通常用于根据关联列的值来创建一个包含两个表的结果集。在 NebulaGraph 中，可以显示地使用 `INNER JOIN` 子句在两个表之间进行连接查询，以获取更加复杂的查询结果。

> Note
>
> 在 nGQL 语句中， `GO` 的多跳查询隐式地使用了 `INNER JOIN` 子句。例如， `GO 1 TO 2 STEPS FROM "player101" OVER follow YIELD $$.player.name AS name, $$.player.age AS age` 语句中， `GO` 子句隐式地使用了 `INNER JOIN` 子句，将从 `player101` 出发沿 `follow` 边第一步查询的结果列和第二步查询的起点列进行匹配，然后基于匹配结果返回 `name` 和 `age` 。

openCypher 兼容性

`INNER JOIN` 子句仅适用于原生 nGQL 语法。

语法

```
YIELD <column_name_list>
FROM <first_table> INNER JOIN <second_table> ON <join_condition>
```

使用说明

- 必须使用 `YIELD` 子句指定返回的列，并且 `YIELD` 子句需放置在 `INNER JOIN` 子句之前。
- 使用 `FROM` 子句指定要连接的两个表。
- `INNER JOIN` 子句必须包含 `ON` 子句， `ON` 子句指定了连接条件，并且连接条件只支持等值连接（即 `==` ）。
- `<first_table>` 和 `<second_table>` 是要连接的两个表，两表名不能相同。
- 使用自定义变量来指定表名。详情参见使用自定义变量。

使用示例

以下示例介绍如何通过 `INNER JOIN` 关联 nGQL 语句中的两个查询结果。

示例一：通过 `INNER JOIN` 将第一个 `LOOKUP` 查询中的结果 `dst` 列（ `Tony Parker` 的 ID `player101` ）和第二个 `GO` 查询中的结果 `src` 列（ `player101` 和 `player125` 两个 ID）进行匹配。匹配的结果集为两个查询结果中 `dst` 和 `src` 列都包含的 `player101` ，基于这个 `player101` 匹配的结果，然后通过 `YIELD $b.vid AS vid, $a.v AS v, $b.e2 AS e2` 返回最终查询结果。

```
nebula> $a = LOOKUP ON player WHERE player.name == 'Tony Parker' YIELD id(vertex) as dst, vertex AS v; \
        $b = GO FROM 'player101', 'player125' OVER follow YIELD id($^) as src, id($$) as vid, edge AS e2; \
        YIELD $b.vid AS vid, $a.v AS v, $b.e2 AS e2 FROM $a INNER JOIN $b ON $a.dst == $b.src;
+-------------+-----------------------------------------------------+------------------------------------------------------+
| vid         | v                                                   | e2                                                   |
+-------------+-----------------------------------------------------+------------------------------------------------------+
| "player100" | ("player101" :player{age: 36, name: "Tony Parker"}) | [:follow "player101"->"player100" @0 {degree: 95}] |
| "player102" | ("player101" :player{age: 36, name: "Tony Parker"}) | [:follow "player101"->"player102" @0 {degree: 90}] |
| "player125" | ("player101" :player{age: 36, name: "Tony Parker"}) | [:follow "player101"->"player125" @0 {degree: 95}] |
+-------------+-----------------------------------------------------+------------------------------------------------------+
```

示例二：通过 `INNER JOIN` 将第一个 `LOOKUP` 查询中的结果 `src` 列（ `Tony Parker` 的 ID `player101` ）和第二个 `FETCH` 查询中的结果 `src` 列（ `player101` 到 `player100` 的起点 ID `player101` ）进行匹配。匹配的结果集为都包含的 `player101` ，基于这个 `player101` 匹配的结果，然后通过 `YIELD $a.src AS src, $a.v AS v, $b.e AS e` 返回最终查询结果。

```
nebula> $a = LOOKUP ON player WHERE player.name == 'Tony Parker' YIELD id(vertex) as src, vertex AS v; \
        $b = FETCH PROP ON follow 'player101'->'player100' YIELD src(edge) as src, edge as e; \
        YIELD $a.src AS src, $a.v AS v, $b.e AS e FROM $a INNER JOIN $b ON $a.src == $b.src;
+-------------+-----------------------------------------------------+------------------------------------------------------+
| src         | v                                                   | e                                                    |
+-------------+-----------------------------------------------------+------------------------------------------------------+
| "player101" | ("player101" :player{age: 36, name: "Tony Parker"}) | [:follow "player101"->"player100" @0 {degree: 95}] |
+-------------+-----------------------------------------------------+------------------------------------------------------+
```

示例三：使用 `INNER JOIN` 关联 `LOOKUP`、 `GO` 和 `FIND PATH` 子句的查询的过程示例步骤：

1. 通过 `LOOKUP ON` 语句，先对 `player` 表进行查询，找到名为 `Tony Parker` 的球员顶点，并将其 ID 和属性存储到 `$a.src` 和 `v` 列中。
2. 再通过 `GO` 查找从 `$a.src` 节点出发经过 `follow` 边往外走 2-5 步的球员节点，其中要求这些节点对应的 `player` 点年龄大于 30 岁，返回这些节点的 ID，并将它们存储在 `$b.dst` 列中。
3. 然后通过 `FIND ALL PATH` 语句，查找从 `$a.src` 到 `$b.dst` 的沿 `follow` 所有路径，并返回路径 `$c.p` 以及路径中的终点 `$c.dst` 。
4. 最后通过 `FIND SHORTEST PATH` 语句，查找从 `$c.dst` 到 `$a.src` 的最短路径，并返回路径 `$d.p` 和路径起点 `$d.src` 。
5. 使用 `INNER JOIN` 将 3 和 4 中得到的结果联接起来，匹配 `$c.dst` 和 `$d.src` 列，为每个联接匹配的记录返回 `YIELD $c.forward AS forwardPath, $c.dst AS end, $d.p AS backwordPath` 。

```
nebula> $a = LOOKUP ON player WHERE player.name == 'Tony Parker' YIELD id(vertex) as src, vertex AS v; \
        $b = GO 2 TO 5 STEPS FROM $a.src OVER follow WHERE $$.player.age > 30 YIELD id($$) AS dst; \
        $c = (FIND ALL PATH FROM $a.src TO $b.dst OVER follow YIELD path AS p | YIELD $-.p AS forward, id(endNode($-.p)) AS dst); \
        $d = (FIND SHORTEST PATH FROM $c.dst TO $a.src OVER follow YIELD path AS p | YIELD $-.p AS p, id(startNode($-.p)) AS src); \
        YIELD $c.forward AS forwardPath, $c.dst AS end, $d.p AS backwordPath FROM $c INNER JOIN $d ON $c.dst == $d.src;
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------+-------------
+---------------------------------------------------------------------------+
| forwardPath                                                                                                                                                                 | end         |
backwordPath                                                                |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------+-------------
+---------------------------------------------------------------------------+
| <("player101")-[:follow@0 {}]->("player102")>                                                                                                                               | "player102" |
<("player102")-[:follow@0 {}]->("player101")>                               |
| <("player101")-[:follow@0 {}]->("player100")-[:follow@0 {}]->("player101")-[:follow@0 {}]->("player102")>                                                                   | "player102" |
<("player102")-[:follow@0 {}]->("player101")>                               |
| <("player101")-[:follow@0 {}]->("player125")>                                                                                                                               | "player125" |
<("player125")-[:follow@0 {}]->("player100")-[:follow@0 {}]->("player101")> |
| <("player101")-[:follow@0 {}]->("player100")-[:follow@0 {}]->("player125")>                                                                                                 | "player125" |
<("player125")-[:follow@0 {}]->("player100")-[:follow@0 {}]->("player101")> |
| <("player101")-[:follow@0 {}]->("player100")-[:follow@0 {}]->("player101")-[:follow@0 {}]->("player125")>                                                                   | "player125" |
<("player125")-[:follow@0 {}]->("player100")-[:follow@0 {}]->("player101")> |
| <("player101")-[:follow@0 {}]->("player102")-[:follow@0 {}]->("player100")-[:follow@0 {}]->("player125")>                                                                   | "player125" |
<("player125")-[:follow@0 {}]->("player100")-[:follow@0 {}]->("player101")> |
| <("player101")-[:follow@0 {}]->("player102")-[:follow@0 {}]->("player101")-[:follow@0 {}]->("player125")>                                                                   | "player125" |
<("player125")-[:follow@0 {}]->("player100")-[:follow@0 {}]->("player101")> |
...
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------+-------------
+---------------------------------------------------------------------------+
```

最后更新: 2025年8月4日

4.7 变量和复合查询

4.7.1 复合查询（子句结构）

复合查询将来自不同请求的数据放在一起，然后进行过滤、分组或者排序等，最后返回结果。

NebulaGraph 支持三种方式进行复合查询（或子查询）：

- （opencypher 兼容语句）连接各个子句，让它们在彼此之间提供中间结果集。
- （原生 nGQL）多个查询可以合并处理，以英文分号（;）分隔，返回最后一个查询的结果。
- （原生 nGQL）可以用管道符（|）将多个查询连接起来，上一个查询的结果可以作为下一个查询的输入。

OpenCypher 兼容性

在复合查询中，请不要混用 opencypher 兼容语句和原生 nGQL 语句，例如 `MATCH ... | GO ... | YIELD ...`。

- 如果使用 openCypher 兼容语句（`MATCH`、`RETURN`、`WITH` 等），请不要使用管道符或分号组合子句。
- 如果使用原生 nGQL 语句（`FETCH`、`GO`、`LOOKUP` 等），必须使用管道符或分号组合子句。

复合查询不支持事务

例如一个查询由三个子查询 A、B、C 组成，A 是一个读操作，B 是一个计算操作，C 是一个写操作，如果在执行过程中，任何一个操作执行失败，则整个结果是未定义的：没有回滚，而且写入的内容取决于执行程序。

> Note
>
> openCypher 没有事务要求。

示例

- opencypher 兼容语句

```
# 子句连接多个查询。
nebula> MATCH p=(v:player{name:"Tim Duncan"})--() \
        WITH nodes(p) AS n \
```

```
        UNWIND n AS n1 \
        RETURN DISTINCT n1;
```

- 原生 nGQL（分号）

```
# 只返回边。
nebula> SHOW TAGS; SHOW EDGES;

# 插入多个点。
nebula> INSERT VERTEX player(name, age) VALUES "player100":("Tim Duncan", 42); \
        INSERT VERTEX player(name, age) VALUES "player101":("Tony Parker", 36); \
        INSERT VERTEX player(name, age) VALUES "player102":("LaMarcus Aldridge", 33);
```

- 原生 nGQL（管道符）

```
# 管道符连接多个查询。
nebula> GO FROM "player100" OVER follow YIELD dst(edge) AS id | \
        GO FROM $-.id OVER serve YIELD properties($$).name AS Team, \
        properties($^).name AS Player;
+-----------+-----------------+
| Team      | Player          |
+-----------+-----------------+
| "Spurs"   | "Tony Parker"   |
| "Hornets" | "Tony Parker"   |
| "Spurs"   | "Manu Ginobili" |
+-----------+-----------------+
```

最后更新: 2025年8月4日

4.7.2 自定义变量

NebulaGraph 允许将一条语句的结果作为自定义变量传递给另一条语句。

OpenCypher 兼容性

当引用一个变量的点、边或路径，需要先给它命名。例如：

```
nebula> MATCH (v:player{name:"Tim Duncan"}) RETURN v;
+----------------------------------------------------+
| v                                                  |
+----------------------------------------------------+
| ("player100" :player{name: "Tim Duncan", age: 42}) |
+----------------------------------------------------+
```

示例中的 `v` 就是自定义变量。

> Caution
>
> 在同一个 MATCH 语句的模式中，不能重复使用边变量。例如 `e` 不能重复被写在模式 `p=(v1)-[e*2..2]->(v2)-[e*2..2]->(v3)` 中。

原生 nGQL

nGQL 扩展的自定义变量可以表示为 `$var_name`， `var_name` 由字母、数字或下划线（_）构成，不允许使用其他字符。

自定义变量仅在当前执行（本复合查询中）有效，执行结束后变量也会释放，不能在其他客户端、执行、session 中使用之前的自定义变量。

用户可以在复合查询中使用自定义变量。复合查询的详细信息请参见复合查询。

> Note
>
> - 自定义变量区分大小写。
> - 在包含扩展的自定义变量的复合语句中，用英文分号 `;` 结束定义变量的语句。详情参见 nGQL 风格指南。

示例

```
nebula> $var = GO FROM "player100" OVER follow YIELD dst(edge) AS id; \
        GO FROM $var.id OVER serve YIELD properties($$).name AS Team, \
        properties($^).name AS Player;
+-----------+-----------------+
| Team      | Player          |
+-----------+-----------------+
| "Spurs"   | "Tony Parker"   |
| "Hornets" | "Tony Parker"   |
| "Spurs"   | "Manu Ginobili" |
+-----------+-----------------+
```

集合运算与变量语句的范围与运算顺序

当我们需要对集合运算的复合语句进行变量赋值时，需要注意用括号包裹语句的范围，比如下边的例子里 `$var` 赋值的来源是两个语句 `INTERSECT` 之后的输出。

```
$var = ( \
    GO FROM "player100" OVER follow \
      YIELD dst(edge) AS id \
    INTERSECT \
    GO FROM "player100" OVER follow \
      YIELD dst(edge) AS id \
       ); \
    GO FROM $var.id OVER follow YIELD follow.degree AS degree
```

最后更新: 2025年8月4日

4.7.3 引用属性

nGQL 提供属性引用符以允许用户在 `GO` 语句中引用起始点、目的点、边的属性，以及引用复合查询的输出结果。本文将详细介绍如何在 nGQL 中使用这些属性引用符。

openCypher 兼容性

属性引用符仅适用于原生 nGQL。

点属性引用符

引用符	说明
`$^`	引用起始点。
`$$`	引用目的点。

引用语法

```
$^.<tag_name>.<prop_name>  # 起始点属性引用
$$.<tag_name>.<prop_name>   # 目的点属性引用
```

- `tag_name`： 点的 Tag 名称。
- `prop_name`： Tag 内的属性名称。

边属性引用符

引用符	说明
`_src`	边的起始点
`_dst`	边的目的点
`_type`	边的类型内部编码，正负号表示方向：正数为正向边，负数为逆向边
`_rank`	边的 rank 值

引用语法

nGQL 允许用户引用边的属性，包括自定义的边属性和四种内置的边属性。

```
<edge_type>.<prop_name>  # 自定义边属性引用
<edge_type>._src|_dst|_type|_rank  # 内置边属性引用
```

- `edge_type`： Edge type。
- `prop_name`： Edge type 的属性名称。

复合查询中的引用符

引用符	说明
`$-`	引用复合查询中管道符之前的语句输出结果。更多信息请参见管道符。

示例

使用点属性引用符

```
# 返回起始点的 Tag player 的 name 属性值和目的点的 Tag player 的 age 属性值。
nebula> GO FROM "player100" OVER follow YIELD $^.player.name AS startName, $$.player.age AS endAge;
+--------------+--------+
| startName    | endAge |
+--------------+--------+
```

```
| "Tim Duncan" | 36     |
| "Tim Duncan" | 41     |
+--------------+--------+
```

历史版本兼容性

从NebulaGraph 2.6.0 起支持了新的 Schema 相关函数。以上示例在NebulaGraph master 中的近似写法如下：

```
GO FROM "player100" OVER follow YIELD properties($^).name AS startName, properties($$).age AS endAge;
```

NebulaGraph master 兼容新旧语法。

使用边属性引用符

```
# 返回 Edge type follow 的 degree 属性值。
nebula> GO FROM "player100" OVER follow YIELD follow.degree;
+---------------+
| follow.degree |
+---------------+
| 95            |
+---------------+

# 返回 EdgeType 是 follow 的起始点 VID、目的点 VID、EdgeType 编码（正数为正向边，负数为逆向边），和边的 rank 值。
nebula> GO FROM "player100" OVER follow YIELD follow._src, follow._dst, follow._type, follow._rank;
+-------------+-------------+--------------+--------------+
| follow._src | follow._dst | follow._type | follow._rank |
+-------------+-------------+--------------+--------------+
| "player100" | "player101" | 17           | 0            |
| "player100" | "player125" | 17           | 0            |
+-------------+-------------+--------------+--------------+
```

历史版本兼容性

从 NebulaGraph 2.6.0 起支持了新的 Schema 相关函数。以上示例在 NebulaGraph master 中的近似写法如下：

```
GO FROM "player100" OVER follow YIELD properties(edge).degree;
GO FROM "player100" OVER follow YIELD src(edge), dst(edge), type(edge), rank(edge);
```

NebulaGraph master 兼容新旧语法。

复合查询中使用属性引用符

以下复合查询语句示例执行操作如下：

1. 使用 `$-.id` 引用管道符前面的 `GO FROM "player100" OVER follow YIELD dst(edge) AS id` 语句的结果，即返回 `follow` 边类型的目的点 ID。
2. 使用 `properties($^)` 函数获取 `serve` 类型边上起始点的球员的 `name` 属性。
3. 使用 `properties($$)` 函数获取 `serve` 类型边上目的点的团队的 `name` 属性。

```
nebula> GO FROM "player100" OVER follow \
        YIELD dst(edge) AS id | \
        GO FROM $-.id OVER serve \
        YIELD properties($^).name AS Player, properties($$).name AS Team;
+-----------------+-----------+
| Player          | Team      |
+-----------------+-----------+
| "Tony Parker"   | "Spurs"   |
| "Tony Parker"   | "Hornets" |
| "Manu Ginobili" | "Spurs"   |
+-----------------+-----------+
```

最后更新: 2025年8月4日

4.8 图空间语句

4.8.1 CREATE SPACE

图空间是 NebulaGraph 中彼此隔离的图数据集合，与 MySQL 中的 database 概念类似。`CREATE SPACE` 语句可以创建一个新的图空间，或者克隆现有图空间的 Schema。

前提条件

只有 God 角色的用户可以执行 `CREATE SPACE` 语句。详情请参见身份验证。

语法

创建图空间

```
CREATE SPACE [IF NOT EXISTS] <graph_space_name> (
    [partition_num = <partition_number>,]
    [replica_factor = <replica_number>,]
    vid_type = {FIXED_STRING(<N>) | INT[64]}
    )
    [COMMENT = '<comment>'];
```

参数	说明
`IF NOT EXISTS`	检测待创建的图空间是否存在，只有不存在时，才会创建图空间。仅检测图空间的名称，不会检测具体属性。
`<graph_space_name>`	1、在NebulaGraph实例中唯一标识一个图空间。 2、图空间名称设置后无法被修改。 3、默认情况下，仅支持 1~4 字节的 UTF-8 编码字符，包括英文字母（区分大小写）、数字、中文等。不能包含下划线（_）以外的特殊字符，且不能以数字开头。 4、如果需要使用特殊字符、保留关键字或者以数字开头，请使用反引号（`）包围整个名称。反引号中不能包含英文句号（`.`）。详情参见关键字和保留字。 注意： 1、如果以中文为图空间命名，报 `SyntaxError` 错误时，需使用反引号（`）包围中文字符。 2、名称中如需包含反引号（`），使用反斜杠（\）来转义反引号（`），如：\`；如需包含反斜杠（\），反斜杠（\）本身也需转义，如：\\。
`partition_num`	指定图空间的分片数量。建议设置为集群中硬盘数量的 20 倍（HDD 硬盘建议为 2 倍）。例如集群中有 3 个硬盘，建议设置 60 个分片。默认值为 10。
`replica_factor`	指定每个分片的副本数量。建议在生产环境中设置为 3，在测试环境中设置为 1。由于需要基于多数表决，副本数量必须是奇数。默认值为 1。
`vid_type`	必选参数。指定点 ID 的数据类型。可选值为 `FIXED_STRING(<N>)` 和 `INT64`。`INT` 等同于 `INT64`。 `FIXED_STRING(<N>)` 表示数据类型为定长字符串，长度为 `N` 字节，超出长度会报错。例如，UTF-8中，一个中文字符的长度为三个字节，如果设置 `N` 为 12，那么 `vid_type` 为最多 4 个中文字符。 `INT64` 表示数据类型为整数。
`COMMENT`	图空间的描述。最大为 256 字节。默认无描述。

Caution

- 如果将副本数设置为 1，用户将无法使用 SUBMIT JOB BALANCE 命令为 NebulaGraph 的存储服务平衡负载或扩容。

- VID 类型变更与长度限制：
- 在 NebulaGraph 1.x 中，VID 的类型只能为 `INT64`，不支持字符型；在 NebulaGraph 2.x 中，VID 的类型支持 `INT64` 和 `FIXED_STRING(<N>)`。请在创建图空间时指定 VID 类型，使用 `INSERT` 语句时也需要保持一致，否则会报错 VID 类型不匹配 `Wrong vertex id type: 1001`。
- VID 最大长度必须为 `N`，不可任意长度；超过该长度也会报错 `The VID must be a 64-bit integer or a string fitting space vertex id length limit.`。

- 如果出现 `Host not enough!` 的报错，直接原因是线上 Storage Host 的数量少于即将创建的图空间的 `replica_factor` 的数量。此时，可用 `SHOW HOSTS` 指令查看判断，出现的情况有：
- 在集群是单 Storage Host 的情况下试图创建多副本（replica_factor）图空间，这时候只能选择创建单副本数的图空间，或者扩容 Storage Host 之后再创建图空间。
- 新创建的集群里 Storage Host 已经被服务发现，但是尚未执行 `ADD HOSTS` 将其激活，这时候需要通过 Console 连接，执行 `SHOW HOSTS` 获取被发现了的 Storage Host，然后执行相应的 `ADD HOSTS` 激活，待有足够的 Online Storage Host 之后再尝试创建图空间。
- 有部分 Storage Host 处在非 Online 状态，需要进行进一步排查。

历史版本兼容性

2.5.0 之前的 2.x 版本中，`vid_type` 不是必选参数，默认为 `FIXED_STRING(8)`。

Note

`graph_space_name`，`partition_num`，`replica_factor`，`vid_type`，`comment` 设置后就无法改变。除非 `DROP SPACE`，并重新 `CREATE SPACE`。

克隆图空间

```
CREATE SPACE [IF NOT EXISTS] <new_graph_space_name> AS <old_graph_space_name>;
```

参数	说明
`IF NOT EXISTS`	检测待创建的图空间是否存在，只有不存在时，才会克隆图空间。仅检测图空间的名称，不会检测具体属性。
`<new_graph_space_name>`	目标图空间名称。该图空间必须未创建。 默认情况下，仅支持 1~4 字节的 UTF-8 编码字符，包括英文字母（区分大小写）、数字、中文等，但是特殊字符只能使用下划线。不能以数字开头。 如果需要使用特殊字符、保留关键字或者以数字开头，请使用反引号（`）包围整个名称。反引号中不能包含英文句号（`.`）。详情参见关键字和保留字。 创建时会克隆 `<old_graph_space_name>` 图空间的 Schema，包括图空间本身参数（分片数量、副本数量等）、Tag、Edge type 和 原生索引。 注意： 1、如果以中文为图空间命名，报 `SyntaxError` 错误时，需使用反引号（`）包围中文字符。 2、名称中如需包含反引号（`），使用反斜杠（\）来转义反引号（`），如：\`；如需包含反斜杠（\），反斜杠（\）本身也需转义，如：\\。
`<old_graph_space_name>`	原始图空间名称。该图空间必须已存在。

示例

```
# 仅指定 VID 类型，其他选项使用默认值。
nebula> CREATE SPACE IF NOT EXISTS my_space_1 (vid_type=FIXED_STRING(30));

# 指定分片数量、副本数量和 VID 类型。
nebula> CREATE SPACE IF NOT EXISTS my_space_2 (partition_num=15, replica_factor=1, vid_type=FIXED_STRING(30));
```

```
# 指定分片数量、副本数量和 VID 类型，并添加描述。
nebula> CREATE SPACE IF NOT EXISTS my_space_3 (partition_num=15, replica_factor=1, vid_type=FIXED_STRING(30)) comment="测试图空间";

# 克隆图空间。
nebula> CREATE SPACE IF NOT EXISTS my_space_4 as my_space_3;
nebula> SHOW CREATE SPACE my_space_4;
+--------------+------------------------------------------------------------------------------------------------------------------------------------------------+
| Space        | Create Space                                                                                                                                   |
+--------------+------------------------------------------------------------------------------------------------------------------------------------------------+
| "my_space_4" | "CREATE SPACE `my_space_4` (partition_num = 15, replica_factor = 1, charset = utf8, collate = utf8_bin, vid_type = FIXED_STRING(30)) comment = '测试图空间'" |
+--------------+------------------------------------------------------------------------------------------------------------------------------------------------+
```

Caution

立刻尝试使用刚创建的图空间可能会失败。因为创建是异步实现的。为确保数据同步，后续操作能顺利进行，请等待 2 个心跳周期（20 秒）。如果需要修改心跳间隔，请为所有配置文件修改参数 `heartbeat_interval_secs` 。但过短的心跳周期（<5 秒）可能会导致分布式系统中的机器误判对端失联。

检查分片分布情况

在大型集群中，由于启动时间不同，分片的分布可能不均衡。用户可以执行如下命令检查分片的分布情况：

```
nebula> SHOW HOSTS;
+------------+------+----------+--------------+--------------------------------+--------------------------------+---------+
| Host       | Port | Status   | Leader count | Leader distribution            | Partition distribution         | Version |
+------------+------+----------+--------------+--------------------------------+--------------------------------+---------+
| "storaged0" | 9779 | "ONLINE" | 8            | "basketballplayer:3, test:5"   | "basketballplayer:10, test:10" | "master" |
| "storaged1" | 9779 | "ONLINE" | 9            | "basketballplayer:4, test:5"   | "basketballplayer:10, test:10" | "master" |
| "storaged2" | 9779 | "ONLINE" | 3            | "basketballplayer:3"           | "basketballplayer:10, test:10" | "master" |
+------------+------+----------+--------------+--------------------------------+--------------------------------+---------+
```

如果需要均衡负载，请执行如下命令：

```
nebula> BALANCE LEADER;
nebula> SHOW HOSTS;
+------------+------+----------+--------------+--------------------------------+--------------------------------+---------+
| Host       | Port | Status   | Leader count | Leader distribution            | Partition distribution         | Version |
+------------+------+----------+--------------+--------------------------------+--------------------------------+---------+
| "storaged0" | 9779 | "ONLINE" | 7            | "basketballplayer:3, test:4"   | "basketballplayer:10, test:10" | "master" |
| "storaged1" | 9779 | "ONLINE" | 7            | "basketballplayer:4, test:3"   | "basketballplayer:10, test:10" | "master" |
| "storaged2" | 9779 | "ONLINE" | 6            | "basketballplayer:3, test:3"   | "basketballplayer:10, test:10" | "master" |
+------------+------+----------+--------------+--------------------------------+--------------------------------+---------+
```

最后更新: 2025年8月4日

4.8.2 USE

`USE` 语句可以指定一个图空间，或切换到另一个图空间，将其作为后续查询的工作空间。

前提条件

执行 `USE` 语句指定图空间时，需要当前登录的用户拥有指定图空间的权限，否则会报错。

语法

```
USE <graph_space_name>;
```

示例

```
# 创建示例空间。
nebula> CREATE SPACE IF NOT EXISTS space1 (vid_type=FIXED_STRING(30));
nebula> CREATE SPACE IF NOT EXISTS space2 (vid_type=FIXED_STRING(30));

# 指定图空间 space1 作为工作空间。
nebula> USE space1;

# 切换到图空间 space2。检索 space2 时，无法从 space1 读取任何数据，检索的点和边与 space1 无关。
nebula> USE space2;
```

Caution

不能在一条语句中同时操作两个图空间。

与 Fabric Cypher 不同，NebulaGraph 的图空间彼此之间是完全隔离的，将一个图空间作为工作空间后，用户无法访问其他空间。使用新图空间的唯一方法是通过 `USE` 语句切换。而在 Fabric Cypher 中可以在一条语句中（`USE + CALL` 语法）使用两个图空间。

最后更新: 2025年8月4日

4.8.3 SHOW SPACES

`SHOW SPACES` 语句可以列出 NebulaGraph 示例中的所有图空间。

语法

```
SHOW SPACES;
```

示例

```
nebula> SHOW SPACES;
+--------------------+
| Name               |
+--------------------+
| "cba"              |
| "basketballplayer" |
+--------------------+
```

创建图空间请参见 CREATE SPACE。

最后更新: 2025年8月4日

4.8.4 DESCRIBE SPACE

`DESCRIBE SPACE` 语句可以显示指定图空间的信息。

语法

你可以用 `DESC` 作为 `DESCRIBE` 的缩写。

```
DESC[RIBE] SPACE <graph_space_name>;
```

示例

```
nebula> DESCRIBE SPACE basketballplayer;
+----+--------------------+------------------+----------------+---------+------------+--------------------+---------+
| ID | Name               | Partition Number | Replica Factor | Charset | Collate    | Vid Type           | Comment |
+----+--------------------+------------------+----------------+---------+------------+--------------------+---------+
| 1  | "basketballplayer" | 10               | 1              | "utf8"  | "utf8_bin" | "FIXED_STRING(32)" |         |
+----+--------------------+------------------+----------------+---------+------------+--------------------+---------+
```

最后更新: 2025年8月4日

4.8.5 CLEAR SPACE

`CLEAR SPACE` 语句用于清空图空间中的点和边，但不会删除图空间本身以及其中的 Schema 信息。

> Note
>
> 建议在执行 `CLEAR SPACE` 操作之后，立即执行 SUBMIT JOB COMPACT操作以提升查询性能。需要注意的是，COMPACT 操作可能会影响查询性能，建议在业务低峰期（例如凌晨）执行该操作。

权限要求

只有 God 角色的用户可以执行 `CLEAR SPACE` 语句。

注意事项

- 数据清除后，如无备份，无法恢复。使用该功能务必谨慎。
- `CLEAR SPACE` 不是原子性操作。如果执行出错，请重新执行，避免残留数据。
- 图空间中的数据量越大， `CLEAR SPACE` 消耗的时间越长。如果 `CLEAR SPACE` 的执行因客户端连接超时而失败，可以增大 Graph 服务配置中 `storage_client_timeout_ms` 参数的值。
- 在 `CLEAR SPACE` 的执行过程中，向该图空间写入数据的行为不会被自动禁止。这样的写入行为可能导致 `CLEAR SPACE` 清除数据不完全，残留的数据也可能受到损坏。

> Note
>
> NebulaGraph 不支持在运行 `CLEAR SPACE` 的同时禁止写入。

语法

```
CLEAR SPACE [IF EXISTS] <space_name>;
```

参数/选项	说明
`IF EXISTS`	检查待清空的图空间是否存在，如果图空间存在，则继续执行清空操作；如果图空间不存在，则完成执行，并且提示执行成功，不会提示图空间不存在。若不设置该选项，当图空间不存在时， `CLEAR SPACE` 语句会执行失败，系统会报错。
`space_name`	要清除数据的图空间名称。

示例：

```
CLEAR SPACE basketballplayer;
```

保留的数据

图空间中， `CLEAR SPACE` 不会删除的数据包括：

- Tag 信息。
- Edge type 信息。
- 原生索引和全文索引的元数据。

下面的执行示例明确展示了 `CLEAR SPACE` 会删除与保留的数据。

```
# 进入图空间 basketballplayer。
nebula[(none)]> use basketballplayer;
Execution succeeded
```

```
# 查看 Tag 和 Edge type。
nebula[basketballplayer]> SHOW TAGS;
+----------+
| Name     |
+----------+
| "player" |
| "team"   |
+----------+
Got 2 rows

nebula[basketballplayer]> SHOW EDGES;
+----------+
| Name     |
+----------+
| "follow" |
| "serve"  |
+----------+
Got 2 rows

# 统计图空间中的数据。
nebula[basketballplayer]> SUBMIT JOB STATS;
+------------+
| New Job Id |
+------------+
| 4          |
+------------+
Got 1 rows

# 查看统计结果。
nebula[basketballplayer]> SHOW STATS;
+---------+------------+-------+
| Type    | Name       | Count |
+---------+------------+-------+
| "Tag"   | "player"   | 51    |
| "Tag"   | "team"     | 30    |
| "Edge"  | "follow"   | 81    |
| "Edge"  | "serve"    | 152   |
| "Space" | "vertices" | 81    |
| "Space" | "edges"    | 233   |
+---------+------------+-------+
Got 6 rows

# 查看 Tag 索引。
nebula[basketballplayer]> SHOW TAG INDEXES;
+------------------+----------+----------+
| Index Name       | By Tag   | Columns  |
+------------------+----------+----------+
| "player_index_0" | "player" | []       |
| "player_index_1" | "player" | ["name"] |
+------------------+----------+----------+
Got 2 rows

# ---------------------- CLEAR SPACE 分割线 ----------------------
# 执行 CLEAR SPACE 清空图空间 basketballplayer。
nebula[basketballplayer]> CLEAR SPACE basketballplayer;
Execution succeeded

# 更新统计信息。
nebula[basketballplayer]> SUBMIT JOB STATS;
+------------+
| New Job Id |
+------------+
| 5          |
+------------+
Got 1 rows

# 查看统计信息。Tag 和 Edge type 还存在，但所有的点和边都没了。
nebula[basketballplayer]> SHOW STATS;
+---------+------------+-------+
| Type    | Name       | Count |
+---------+------------+-------+
| "Tag"   | "player"   | 0     |
| "Tag"   | "team"     | 0     |
| "Edge"  | "follow"   | 0     |
| "Edge"  | "serve"    | 0     |
| "Space" | "vertices" | 0     |
| "Space" | "edges"    | 0     |
+---------+------------+-------+
Got 6 rows

# 查看 Tag 索引，它们依然存在。
nebula[basketballplayer]> SHOW TAG INDEXES;
+------------------+----------+----------+
| Index Name       | By Tag   | Columns  |
+------------------+----------+----------+
| "player_index_0" | "player" | []       |
| "player_index_1" | "player" | ["name"] |
+------------------+----------+----------+
Got 2 rows (time spent 523/978 us)
```

最后更新: 2025年8月4日

4.8.6 DROP SPACE

`DROP SPACE` 语句用于删除指定图空间以及其中的所有信息。

> Note
>
> `DROP SPACE` 是否删除图空间对应的硬盘数据由 Storage 配置参数 `auto_remove_invalid_space` 决定。`auto_remove_invalid_space` 的默认值为 `true`，表示会删除数据。如需在删除逻辑图空间时保留硬盘数据，将 `auto_remove_invalid_space` 的值修改为 `false`。详情参见 Storage 服务配置。

> Warning
>
> 执行 `DROP SPACE` 后，即使快照中存在该图空间的数据，该图空间的数据也无法恢复。

前提条件

只有 God 角色的用户可以执行 `DROP SPACE` 语句。详情请参见身份验证。

语法

```
DROP SPACE [IF EXISTS] <graph_space_name>;
```

`IF EXISTS` 关键字可以检测待删除的图空间是否存在，只有存在时，才会删除图空间。

> 历史版本兼容性
>
> 在 NebulaGraph 3.1.0 版本前，`DROP SPACE` 语句不会删除硬盘上对应图空间的目录和文件。

> Danger
>
> 请谨慎执行删除图空间操作。

FAQ

问：执行 `DROP SPACE` 语句删除图空间后，为什么磁盘的大小没变化？

答：如果使用 3.1.0 之前版本的 NebulaGraph，`DROP SPACE` 语句仅删除指定的逻辑图空间，不会删除硬盘上对应图空间的目录和文件。如需删除硬盘上的数据，需手动删除相应文件的路径，文件路径为 `<nebula_graph_install_path>/data/storage/nebula/<space_id>`。其中 `<space_id>` 可以通过 `DESCRIBE SPACE {space_name}` 查看。

最后更新: 2025年8月4日

4.9 Tag 语句

4.9.1 CREATE TAG

`CREATE TAG` 语句可以通过指定名称创建一个 Tag。

OpenCypher 兼容性

nGQL 中的 Tag 和 openCypher 中的 Label 相似，但又有所不同，例如它们的创建方式。

- openCypher 中的 Label 需要在 `CREATE` 语句中与点一起创建。
- nGQL 中的 Tag 需要使用 `CREATE TAG` 语句独立创建。Tag 更像是 MySQL 中的表。

前提条件

执行 `CREATE TAG` 语句需要当前登录的用户拥有指定图空间的创建 Tag 权限，否则会报错。

语法

创建 Tag 前，需要先用 `USE` 语句指定工作空间。

```
CREATE TAG [IF NOT EXISTS] <tag_name>
    (
      <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']
      [{, <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']} ...]
    )
    [TTL_DURATION = <ttl_duration>]
```

```
    [TTL_COL = <prop_name>]
    [COMMENT = '<comment>'];
```

参数	说明
`IF NOT EXISTS`	检测待创建的 Tag 是否存在，只有不存在时，才会创建 Tag。仅检测 Tag 的名称，不会检测具体属性。
`<tag_name>`	1、每个图空间内的 Tag 必须是唯一的。 2、Tag 名称设置后无法修改。 3、默认情况下，仅支持 1~4 字节的 UTF-8 编码字符，包括英文字母（区分大小写）、数字、中文等。不能包含下划线（_）以外的特殊字符，且不能以数字开头。 4、如果需要使用特殊字符、保留关键字或者以数字开头，请使用反引号（`）包围整个名称。反引号中不能包含英文句号（`.`）。详情参见关键字和保留字。 注意： 1、如果以中文命名 Tag，报 `SyntaxError` 错误时，需使用反引号（`）包围中文字符。 2、名称中如需包含反引号（`），使用反斜杠（\）来转义反引号（`），如：\`；如需包含反斜杠（\），反斜杠（\）本身也需转义，如：\\。
`<prop_name>`	属性名称。每个 Tag 中的属性名称必须唯一。属性的命名规则与 Tag 相同。
`<data_type>`	属性的数据类型，目前支持数值、布尔、字符串以及日期与时间。
`NULL | NOT NULL`	指定属性值是否支持为 `NULL`。默认值为 `NULL`。指定 `NOT NULL` 且没有指定 `DEFAULT` 值时，插入数据必须指定值；同时指定 `NOT NULL` 和 `DEFAULT` 值时，插入数据如果没有指定值，则默认插入 `DEFAULT` 值。
`DEFAULT`	指定属性的默认值。默认值可以是一个文字值或 NebulaGraph 支持的表达式。如果插入点时没有指定某个属性的值，则使用默认值。
`COMMENT`	对单个属性或 Tag 的描述。最大为 256 字节。默认无描述。
`TTL_DURATION`	指定时间戳差值，单位：秒。时间戳差值必须为 64 位非负整数。属性值和时间戳差值之和如果小于当前时间戳，属性就会过期。默认值为 `0`，表示属性永不过期。
`TTL_COL`	指定要设置存活时间的属性。属性的数据类型必须是 `int` 或者 `timestamp`。一个 Tag 只能指定一个字段为 `TTL_COL`。更多 TTL 的信息请参见 TTL。

示例

```
nebula> CREATE TAG IF NOT EXISTS player(name string, age int);

# 创建没有属性的 Tag。
nebula> CREATE TAG IF NOT EXISTS no_property();

# 创建包含默认值的 Tag。
nebula> CREATE TAG IF NOT EXISTS player_with_default(name string, age int DEFAULT 20);

# 对字段 create_time 设置 TTL 为 100 秒。
nebula> CREATE TAG IF NOT EXISTS woman(name string, age int, \
        married bool, salary double, create_time timestamp) \
        TTL_DURATION = 100, TTL_COL = "create_time";
```

创建 **Tag** 说明

尝试使用新创建的 Tag 可能会失败，因为创建是异步实现的。为确保数据同步，后续操作能顺利进行，请等待 2 个心跳周期（20 秒）。

如果需要修改心跳间隔，请为所有配置文件修改参数 `heartbeat_interval_secs`。

最后更新: 2025年8月4日

4.9.2 DROP TAG

`DROP TAG` 语句可以删除当前工作空间内所有点上的指定 Tag。

点可以有一个或多个 Tag。

- 如果某个点只有一个 Tag，删除这个 Tag 后，用户就无法访问这个点，下次 Compaction 操作时会删除该点，但与该点相邻的边仍然存在——这会造成悬挂边。
- 如果某个点有多个 Tag，删除其中一个 Tag，仍然可以访问这个点，但是无法访问已删除 Tag 所定义的所有属性。

删除 Tag 操作仅删除 Schema 数据，硬盘上的文件或目录不会立刻删除，而是在下一次 Compaction 操作时删除。

> Compatibility
>
> NebulaGraph master 中默认不支持插入无 Tag 的点。如需使用无 Tag 的点，在集群内所有 Graph 服务的配置文件（`nebula-graphd.conf`）中新增 `--graph_use_vertex_key=true`；在所有 Storage 服务的配置文件（`nebula-storaged.conf`）中新增 `--use_vertex_key=true`。

前提条件

- 登录的用户必须拥有对应权限才能执行 `DROP TAG` 语句。详情请参见内置角色权限。
- 确保 Tag 不包含任何索引，否则 `DROP TAG` 时会报冲突错误 `[ERROR (-1005)]: Conflict!`。删除索引请参见 drop index。

语法

```
DROP TAG [IF EXISTS] <tag_name>;
```

- `IF EXISTS`：检测待删除的 Tag 是否存在，只有存在时，才会删除 Tag。
- `tag_name`：指定要删除的 Tag 名称。一次只能删除一个 Tag。

示例

```
nebula> CREATE TAG IF NOT EXISTS test(p1 string, p2 int);
nebula> DROP TAG test;
```

最后更新: 2025年8月4日

4.9.3 ALTER TAG

`ALTER TAG` 语句可以修改 Tag 的结构。例如增删属性、修改数据类型，也可以为属性设置、修改 TTL（Time-To-Live）。

注意事项

- 登录的用户必须拥有对应权限才能执行 `ALTER TAG` 语句。详情请参见内置角色权限。
- 确保要修改的属性不包含索引，否则 `ALTER TAG` 时会报冲突错误 `[ERROR (-1005)]: Conflict!`。删除索引请参见 drop index。
- 确保新增的属性名不与已存在或被删除的属性名同名，否则新增属性会失败。

语法

```
ALTER TAG <tag_name>
    <alter_definition> [[, alter_definition] ...]
    [ttl_definition [, ttl_definition] ... ]
    [COMMENT = '<comment>'];

alter_definition:
| ADD    (prop_name data_type [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>'])
| DROP   (prop_name)
| CHANGE (prop_name data_type [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>'])

ttl_definition:
    TTL_DURATION = ttl_duration, TTL_COL = prop_name
```

- `tag_name`：指定要修改的 Tag 名称。一次只能修改一个 Tag。请确保要修改的 Tag 在当前工作空间中存在，否则会报错。
- 可以在一个 `ALTER TAG` 语句中使用多个 `ADD`、`DROP` 和 `CHANGE` 子句，子句之间用英文逗号（,）分隔。
- 当使用 `ADD` 或 `CHANGE` 指定属性值为 `NOT NULL` 时，必需为该属性指定默认值，即定义 `DEFAULT` 的值。

- 当使用 `CHANGE` 修改属性的数据类型时：
- 仅允许修改 `FIXED_STRING` 和 `INT` 类型的长度为更大的长度，不允许减少长度。
- 仅允许修改 `FIXED_STRING` 类型为 `STRING` 类型、修改 `FLOAT` 类型为 `DOUBLE` 类型。

示例

```
nebula> CREATE TAG IF NOT EXISTS t1 (p1 string, p2 int);
nebula> ALTER TAG t1 ADD (p3 int32, p4 fixed_string(10));
nebula> ALTER TAG t1 TTL_DURATION = 2, TTL_COL = "p2";
nebula> ALTER TAG t1 COMMENT = 'test1';
nebula> ALTER TAG t1 ADD (p5 double NOT NULL DEFAULT 0.4 COMMENT 'p5') COMMENT='test2';
// 将 TAG t1 的 p3 属性类型从 INT32 改为 INT64，p4 属性类型从 FIXED_STRING(10) 改为 STRING。
nebula> ALTER TAG t1 CHANGE (p3 int64, p4 string);
```

修改 **Tag** 说明

尝试使用刚修改的 Tag 可能会失败，因为修改是异步实现的。为确保数据同步，后续操作能顺利进行，请等待 2 个心跳周期（20 秒）。

如果需要修改心跳间隔，请为所有配置文件修改参数 `heartbeat_interval_secs`。

最后更新: 2025年8月4日

4.9.4 SHOW TAGS

`SHOW TAGS` 语句显示当前图空间内的所有 Tag 名称。

执行 `SHOW TAGS` 语句不需要任何权限，但是返回结果由登录的用户权限决定。

语法

```
SHOW TAGS;
```

示例

```
nebula> SHOW TAGS;
+----------+
| Name     |
+----------+
| "player" |
| "team"   |
+----------+
```

最后更新: 2025年8月4日

4.9.5 DESCRIBE TAG

`DESCRIBE TAG` 显示指定 Tag 的详细信息，例如字段名称、数据类型等。

前提条件

登录的用户必须拥有对应权限才能执行 `DESCRIBE TAG` 语句。详情请参见内置角色权限。

语法

```
DESC[RIBE] TAG <tag_name>;
```

`DESCRIBE` 可以缩写为 `DESC` 。

示例

```
nebula> DESCRIBE TAG player;
+--------+----------+-------+---------+---------+
| Field  | Type     | Null  | Default | Comment |
+--------+----------+-------+---------+---------+
| "name" | "string" | "YES" |         |         |
| "age"  | "int64"  | "YES" |         |         |
+--------+----------+-------+---------+---------+
```

最后更新: 2025年8月4日

4.9.6 DELETE TAG

`DELETE TAG` 语句可以删除指定点上的指定 Tag。

前提条件

登录的用户必须拥有对应权限才能执行 `DELETE TAG` 语句。详情请参见内置角色权限。

语法

```
DELETE TAG <tag_name_list> FROM <VID_list>;
```

- `tag_name_list`：指定 Tag 名称。多个 Tag 用英文逗号（,）分隔，也可以用 `*` 表示所有 Tag。
- `VID_list`：指定要删除 Tag 的点 ID。可以指定多个 VID，用英文逗号（,）分隔。

示例

```
nebula> CREATE TAG IF NOT EXISTS test1(p1 string, p2 int);
nebula> CREATE TAG IF NOT EXISTS test2(p3 string, p4 int);
nebula> INSERT VERTEX test1(p1, p2),test2(p3, p4) VALUES "test":("123", 1, "456", 2);
nebula> FETCH PROP ON * "test" YIELD vertex AS v;
+------------------------------------------------------------+
| v                                                          |
+------------------------------------------------------------+
| ("test" :test1{p1: "123", p2: 1} :test2{p3: "456", p4: 2}) |
+------------------------------------------------------------+

nebula> DELETE TAG test1 FROM "test";
nebula> FETCH PROP ON * "test" YIELD vertex AS v;
+-----------------------------------+
| v                                 |
+-----------------------------------+
| ("test" :test2{p3: "456", p4: 2}) |
+-----------------------------------+

nebula> DELETE TAG * FROM "test";
nebula> FETCH PROP ON * "test" YIELD vertex AS v;
+---+
| v |
+---+
+---+
```

Compatibility

- 在 openCypher 中，可以使用 `REMOVE v:LABEL` 语句来移除该点 `v` 的 `LABEL`。
- 相同语意，但不同语法。在 nGQL 中使用 `DELETE TAG`。

最后更新: 2025年8月4日

4.9.7 增加和删除标签

在 openCypher 中，有增加标签（`SET label`）和移除标签（`REMOVE label`）的功能，可以用于加速查询或者标记过程。

在 NebulaGraph 中，可以通过 Tag 变相实现相同操作，创建 Tag 并将 Tag 插入到已有的点上，就可以根据 Tag 名称快速查找点，也可以通过 `DELETE TAG` 删除某些点上不再需要的 Tag。

示例

例如在 basketballplayer 数据集中，部分篮球运动员同时也是球队股东，可以为股东 Tag `shareholder` 创建索引，方便快速查找。如果不再是股东，可以通过 `DELETE TAG` 语句删除相应运动员的股东 Tag。

```
//创建股东 Tag 和索引
nebula> CREATE TAG IF NOT EXISTS shareholder();
nebula> CREATE TAG INDEX IF NOT EXISTS shareholder_tag on shareholder();

//为点添加 Tag
nebula> INSERT VERTEX shareholder() VALUES "player100":();
nebula> INSERT VERTEX shareholder() VALUES "player101":();

//快速查询所有股东
nebula> MATCH (v:shareholder) RETURN v;
+---------------------------------------------------------------------+
| v                                                                   |
+---------------------------------------------------------------------+
| ("player100" :player{age: 42, name: "Tim Duncan"} :shareholder{})  |
| ("player101" :player{age: 36, name: "Tony Parker"} :shareholder{}) |
+---------------------------------------------------------------------+
nebula> LOOKUP ON shareholder YIELD id(vertex);
+-------------+
| id(VERTEX)  |
+-------------+
| "player100" |
| "player101" |
+-------------+

//如果 player100 不再是股东
nebula> DELETE TAG shareholder FROM "player100";
nebula> LOOKUP ON shareholder YIELD id(vertex);
+-------------+
| id(VERTEX)  |
+-------------+
| "player101" |
+-------------+
```

> **Note**
>
> 如果插入测试数据后才创建索引，请用 `REBUILD TAG INDEX <index_name_list>;` 语句重建索引。

最后更新: 2025年8月4日

4.10 Edge type 语句

4.10.1 CREATE EDGE

`CREATE EDGE` 语句可以通过指定名称创建一个 Edge type。

OpenCypher 兼容性

nGQL 中的 Edge type 和 openCypher 中的关系类型相似，但又有所不同，例如它们的创建方式。

- openCypher 中的关系类型需要在 `CREATE` 语句中与点一起创建。
- nGQL 中的 Edge type 需要使用 `CREATE EDGE` 语句独立创建。Edge type 更像是 MySQL 中的表。

前提条件

执行 `CREATE EDGE` 语句需要当前登录的用户拥有指定图空间的创建 Edge type 权限，否则会报错。

语法

创建 Edge type 前，需要先用 `USE` 语句指定工作空间。

```
CREATE EDGE [IF NOT EXISTS] <edge_type_name>
    (
      <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']
      [{, <prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>']} ...]
    )
    [TTL_DURATION = <ttl_duration>]
    [TTL_COL = <prop_name>]
    [COMMENT = '<comment>'];
```

参数	说明
`IF NOT EXISTS`	检测待创建的 Edge type 是否存在，只有不存在时，才会创建 Edge type。仅检测 Edge type 的名称，不会检测具体属性。
`<edge_type_name>`	1、每个图空间内的 Edge type 必须是唯一的。 2、Edge type 名称设置后无法修改。 3、默认情况下，仅支持 1~4 字节的 UTF-8 编码字符，包括英文字母（区分大小写）、数字、中文等。不能包含下划线（_）以外的特殊字符，且不能以数字开头。 4、如果需要使用特殊字符、保留关键字或者以数字开头，请使用反引号（`）包围整个名称。反引号中不能包含英文句号（`.`）。详情参见关键字和保留字。 注意： 1、如果以中文命名 Edge type，报 `SyntaxError` 错误时，需使用反引号（`）包围中文字符。 2、名称中如需包含反引号（`），使用反斜杠（\）来转义反引号（`），如：\`；如需包含反斜杠（\），反斜杠（\）本身也需转义，如：\\。
`<prop_name>`	属性名称。每个 Edge type 中的属性名称必须唯一。属性的命名规则与 Edge type 相同。
`<data_type>`	属性的数据类型，目前支持数值、布尔、字符串以及日期与时间。
`NULL | NOT NULL`	指定属性值是否支持为 `NULL`。默认值为 `NULL`。当指定属性值为 `NOT NULL` 时，必需指定属性的默认值，也就是 `DEFAULT` 的值。
`DEFAULT`	指定属性的默认值。默认值可以是一个文字值或 NebulaGraph 支持的表达式。如果插入边时没有指定某个属性的值，则使用默认值。
`COMMENT`	对单个属性或 Edge type 的描述。最大为 256 字节。默认无描述。
`TTL_DURATION`	指定时间戳差值，单位：秒。时间戳差值必须为 64 位非负整数。属性值和时间戳差值之和如果小于当前时间戳，属性就会过期。默认值为 `0`，表示属性永不过期。
`TTL_COL`	指定要设置存活时间的属性。属性的数据类型必须是 `int` 或者 `timestamp`。一个 Edge type 只能指定一个字段为 `TTL_COL`。更多 TTL 的信息请参见 TTL。

示例

```
nebula> CREATE EDGE IF NOT EXISTS follow(degree int);

# 创建没有属性的 Edge type。
nebula> CREATE EDGE IF NOT EXISTS no_property();

# 创建包含默认值的 Edge type。
nebula> CREATE EDGE IF NOT EXISTS follow_with_default(degree int DEFAULT 20);

# 对字段 p2 设置 TTL 为 100 秒。
nebula> CREATE EDGE IF NOT EXISTS e1(p1 string, p2 int, p3 timestamp) \
        TTL_DURATION = 100, TTL_COL = "p2";
```

最后更新: 2025年8月4日

4.10.2 DROP EDGE

`DROP EDGE` 语句可以删除当前工作空间内的指定 Edge type。

一个边只能有一个 Edge type，删除这个 Edge type 后，用户就无法访问这个边，下次 Compaction 操作时会删除该边。

删除 Edge type 操作仅删除 Schema 数据，硬盘上的文件或目录不会立刻删除，而是在下一次 Compaction 操作时删除。

前提条件

- 登录的用户必须拥有对应权限才能执行 `DROP EDGE` 语句。详情请参见内置角色权限。
- 确保 Edge type 不包含任何索引，否则 `DROP EDGE` 时会报冲突错误 `[ERROR (-1005)]: Conflict!`。删除索引请参见 drop index。

语法

```
DROP EDGE [IF EXISTS] <edge_type_name>
```

- `IF EXISTS`：检测待删除的 Edge type 是否存在，只有存在时，才会删除 Edge type。
- `edge_type_name`：指定要删除的 Edge type 名称。一次只能删除一个 Edge type。

示例

```
nebula> CREATE EDGE IF NOT EXISTS e1(p1 string, p2 int);
nebula> DROP EDGE e1;
```

最后更新: 2025年8月4日

4.10.3 ALTER EDGE

`ALTER EDGE` 语句可以修改 Edge type 的结构。例如增删属性、修改数据类型，也可以为属性设置、修改 TTL（Time-To-Live）。

注意事项

- 登录的用户必须拥有对应权限才能执行 `ALTER EDGE` 语句。详情请参见内置角色权限。
- 确保要修改的属性不包含索引，否则 `ALTER EDGE` 时会报冲突错误 `[ERROR (-1005)]: Conflict!`。删除索引请参见 drop index。
- 确保新增的属性名不与已存在或被删除的属性名同名，否则新增属性会失败。
- 允许增加 `FIXED_STRING` 和 `INT` 类型的长度。
- 允许 `FIXED_STRING` 类型转换为 `STRING` 类型、 `FLOAT` 类型转换为 `DOUBLE` 类型。

语法

```
ALTER EDGE <edge_type_name>
    <alter_definition> [, alter_definition] ...]
    [ttl_definition [, ttl_definition] ... ]
    [COMMENT = '<comment>'];

alter_definition:
| ADD    (prop_name data_type)
| DROP   (prop_name)
| CHANGE (prop_name data_type)

ttl_definition:
    TTL_DURATION = ttl_duration, TTL_COL = prop_name
```

- `edge_type_name`：指定要修改的 Edge type 名称。一次只能修改一个 Edge type。请确保要修改的 Edge type 在当前工作空间中存在，否则会报错。
- 可以在一个 `ALTER EDGE` 语句中使用多个 `ADD`、 `DROP` 和 `CHANGE` 子句，子句之间用英文逗号（,）分隔。
- 当使用 `ADD` 或 `CHANGE` 指定属性值为 `NOT NULL` 时，必需为该属性指定默认值，即定义 `DEFAULT` 的值。

示例

```
nebula> CREATE EDGE IF NOT EXISTS e1(p1 string, p2 int);
nebula> ALTER EDGE e1 ADD (p3 int, p4 string);
nebula> ALTER EDGE e1 TTL_DURATION = 2, TTL_COL = "p2";
nebula> ALTER EDGE e1 COMMENT = 'edge1';
```

修改 **Edge type** 说明

尝试使用刚修改的 Edge type 可能会失败，因为修改是异步实现的。为确保数据同步，后续操作能顺利进行，请等待 2 个心跳周期（20 秒）。

如果需要修改心跳间隔，请为所有配置文件修改参数 `heartbeat_interval_secs`。

最后更新: 2025年8月4日

4.10.4 SHOW EDGES

`SHOW EDGES` 语句显示当前图空间内的所有 Edge type 名称。

执行 `SHOW EDGES` 语句不需要任何权限，但是返回结果由登录的用户权限 决定。

语法

```
SHOW EDGES;
```

示例

```
nebula> SHOW EDGES;
+----------+
| Name     |
+----------+
| "follow" |
| "serve"  |
+----------+
```

最后更新: 2025年8月4日

4.10.5 DESCRIBE EDGE

`DESCRIBE EDGE` 显示指定 Edge type 的详细信息，例如字段名称、数据类型等。

前提条件

登录的用户必须拥有对应权限才能执行 `DESCRIBE EDGE` 语句。详情请参见内置角色权限。

语法

```
DESC[RIBE] EDGE <edge_type_name>
```

`DESCRIBE` 可以缩写为 `DESC`。

示例

```
nebula> DESCRIBE EDGE follow;
+----------+---------+-------+---------+---------+
| Field    | Type    | Null  | Default | Comment |
+----------+---------+-------+---------+---------+
| "degree" | "int64" | "YES" |         |         |
+----------+---------+-------+---------+---------+
```

最后更新: 2025年8月4日

4.11 点语句

4.11.1 INSERT VERTEX

`INSERT VERTEX` 语句可以在 NebulaGraph 实例的指定图空间中插入一个或多个点。

前提条件

执行 `INSERT VERTEX` 语句需要当前登录的用户拥有指定图空间的插入点权限，否则会报错。

语法

```
INSERT VERTEX [IF NOT EXISTS] [tag_props, [tag_props] ...]
VALUES VID: ([prop_value_list])

tag_props:
  tag_name ([prop_name_list])

prop_name_list:
   [prop_name [, prop_name] ...]

prop_value_list:
   [prop_value [, prop_value] ...]
```

- `IF NOT EXISTS`：检测待插入的 VID 是否存在，只有不存在时，才会插入，如果已经存在，不会进行修改。

> Note
>
> - `IF NOT EXISTS` 仅检测 VID + Tag 的值是否相同，不会检测属性值。
> - `IF NOT EXISTS` 会先读取一次数据是否存在，因此对性能会有明显影响。

- `tag_name`：点关联的 Tag（点类型）。Tag 的创建，详情请参见 CREATE TAG。

> Caution
>
> NebulaGraph master 中默认不支持插入无 Tag 的点。如需使用无 Tag 的点，在集群内所有 Graph 服务的配置文件（`nebula-graphd.conf`）中新增 `--graph_use_vertex_key=true`；在所有 Storage 服务的配置文件（`nebula-storaged.conf`）中新增 `--use_vertex_key=true`。插入无 Tag 点的命令示例如 `INSERT VERTEX VALUES "1":();`。

- `property_name`：需要设置的属性名称。
- `vid`：点 ID。在 NebulaGraph master 中支持字符串和整数，需要在创建图空间时设置，详情请参见 CREATE SPACE。
- `property_value`：根据 `prop_name_list` 填写属性值。如果没有填写属性值，而 Tag 中对应的属性设置为 `NOT NULL`，会返回错误。详情请参见 CREATE TAG。

> Caution
>
> `INSERT VERTEX` 与 openCypher 中 `CREATE` 的语意不同：
>
> - `INSERT VERTEX` 语意更接近于 NoSQL(key-value) 方式的 INSERT 语意，或者 SQL 中的 `UPSERT`（`UPDATE` or `INSERT`）。
> - 相同 `VID` 和 `TAG` 的情况下，如果没有使用 `IF NOT EXISTS`，新写入的数据会覆盖旧数据，不存在时会新写入。
> - 相同 `VID` 但不同 `TAG` 的情况下，不同 TAG 对应的记录不会相互覆盖，不存在会新写入。

参考以下示例。

示例

```
# 插入不包含属性的点。
nebula> CREATE TAG IF NOT EXISTS t1();
nebula> INSERT VERTEX t1() VALUES "10":();
```

```
nebula> CREATE TAG IF NOT EXISTS t2 (name string, age int);
nebula> INSERT VERTEX t2 (name, age) VALUES "11":("n1", 12);

# 创建失败，因为"a13"不是 int 类型。
nebula> INSERT VERTEX t2 (name, age) VALUES "12":("n1", "a13");

# 一次插入 2 个点。
nebula> INSERT VERTEX t2 (name, age) VALUES "13":("n3", 12), "14":("n4", 8);
```

```
nebula> CREATE TAG IF NOT EXISTS t3(p1 int);
nebula> CREATE TAG IF NOT EXISTS t4(p2 string);

# 一次插入两个 Tag 的属性到同一个点。
nebula> INSERT VERTEX  t3 (p1), t4(p2) VALUES "21": (321, "hello");
```

一个点可以多次插入属性值，以最后一次为准。

```
# 多次插入属性值。
nebula> INSERT VERTEX t2 (name, age) VALUES "11":("n2", 13);
nebula> INSERT VERTEX t2 (name, age) VALUES "11":("n3", 14);
nebula> INSERT VERTEX t2 (name, age) VALUES "11":("n4", 15);
nebula> FETCH PROP ON t2 "11" YIELD properties(vertex);
+-----------------------+
| properties(VERTEX)    |
+-----------------------+
| {age: 15, name: "n4"} |
+-----------------------+
```

```
nebula> CREATE TAG IF NOT EXISTS t5(p1 fixed_string(5) NOT NULL, p2 int, p3 int DEFAULT NULL);
nebula> INSERT VERTEX t5(p1, p2, p3) VALUES "001":("Abe", 2, 3);

# 插入失败，因为属性 p1 不能为 NULL。
nebula> INSERT VERTEX t5(p1, p2, p3) VALUES "002":(NULL, 4, 5);
[ERROR (-1009)]: SemanticError: No schema found for `t5'

# 属性 p3 为默认值 NULL。
nebula> INSERT VERTEX t5(p1, p2) VALUES "003":("cd", 5);
nebula> FETCH PROP ON t5 "003" YIELD properties(vertex);
+---------------------------------+
| properties(VERTEX)              |
+---------------------------------+
| {p1: "cd", p2: 5, p3: __NULL__} |
+---------------------------------+

# 属性 p1 最大长度为 5，因此会被截断。
nebula> INSERT VERTEX t5(p1, p2) VALUES "004":("shalalalala", 4);
nebula> FETCH PROP on t5 "004" YIELD properties(vertex);
+------------------------------------+
| properties(VERTEX)                 |
+------------------------------------+
| {p1: "shala", p2: 4, p3: __NULL__} |
+------------------------------------+
```

使用 `IF NOT EXISTS` 插入已存在的点时，不会进行修改。

```
# 插入点 1。
nebula> INSERT VERTEX t2 (name, age) VALUES "1":("n2", 13);

# 使用 IF NOT EXISTS 修改点 1，因为点 1 已存在，不会进行修改。
nebula> INSERT VERTEX IF NOT EXISTS t2 (name, age) VALUES "1":("n3", 14);
nebula> FETCH PROP ON t2 "1" YIELD properties(vertex);
+-----------------------+
| properties(VERTEX)    |
+-----------------------+
| {age: 13, name: "n2"} |
+-----------------------+
```

最后更新: 2025年8月4日

4.11.2 DELETE VERTEX

`DELETE VERTEX` 语句可以删除点，但是默认不删除该点关联的出边和入边。

> Compatibility
>
> NebulaGraph 2.x 默认删除点及关联该点的出边和入边，NebulaGraph master 默认只删除点，不删除该点关联的出边和入边，此时将默认存在悬挂边。

`DELETE VERTEX` 语句一次可以删除一个或多个点。用户可以结合管道符一起使用，详情请参见管道符。

> Note
>
> - `DELETE VERTEX` 是直接删除点，不删除关联的边。
> - `DELETE TAG` 是删除指定点上的指定 Tag。

语法

```
DELETE VERTEX <vid> [ , <vid> ... ] [WITH EDGE];
```

- `WITH EDGE`：删除该点关联的出边和入边。

示例

```
# 删除 VID 为 `team1` 的点，不删除该点关联的出边和入边。
nebula> DELETE VERTEX "team1";

# 删除 VID 为 `team1` 的点，并删除该点关联的出边和入边。
nebula> DELETE VERTEX "team1" WITH EDGE;
```

```
# 结合管道符，删除符合条件的点。
nebula> GO FROM "player100" OVER serve WHERE properties(edge).start_year == "2021" YIELD dst(edge) AS id | DELETE VERTEX $-.id;
```

删除过程

NebulaGraph 找到目标点并删除，该目标点的所有邻边（出边和入边）将成为悬挂边。

> Caution
>
> - 不支持原子性删除，如果发生错误请重试，避免出现部分删除的情况。否则会导致悬挂边。
> - 删除超级节点耗时较多，为避免删除完成前连接超时，可以调整 `nebula-graphd.conf` 中的参数 `--storage_client_timeout_ms` 延长超时时间。

悬挂边视频

- NebulaGraph 的悬挂边小科普（2 分 28 秒）

最后更新: 2025年8月4日

4.11.3 UPDATE VERTEX

`UPDATE VERTEX` 语句可以修改点上 Tag 的属性值。

NebulaGraph 支持 CAS（compare and set）操作。

Note

一次只能修改一个 Tag。

语法

```
UPDATE VERTEX ON <tag_name> <vid>
SET <update_prop>
[WHEN <condition>]
[YIELD <output>]
```

参数	是否必须	说明	示例
`ON <tag_name>`	是	指定点的 Tag。要修改的属性必须在这个 Tag 内。	`ON player`
`<vid>`	是	指定要修改的点 ID。	`"player100"`
`SET <update_prop>`	是	指定如何修改属性值。	`SET age = age +1`
`WHEN <condition>`	否	指定过滤条件。如果 `<condition>` 结果为 `false`， `SET` 子句不会生效。	`WHEN name == "Tim"`
`YIELD <output>`	否	指定语句的输出格式。	`YIELD name AS Name`

示例

```
// 查看点"player101"的属性。
nebula> FETCH PROP ON player "player101" YIELD properties(vertex);
+--------------------------------+
| properties(VERTEX)             |
+--------------------------------+
| {age: 36, name: "Tony Parker"} |
+--------------------------------+

// 修改属性 age 的值，并返回 name 和新的 age。
nebula> UPDATE VERTEX ON player "player101" \
        SET age = age + 2 \
        WHEN name == "Tony Parker" \
        YIELD name AS Name, age AS Age;
+---------------+-----+
| Name          | Age |
+---------------+-----+
| "Tony Parker" | 38  |
+---------------+-----+
```

最后更新: 2025年8月4日

4.11.4 UPSERT VERTEX

`UPSERT VERTEX` 语句结合 `UPDATE` 和 `INSERT`，如果点存在，会修改点的属性值；如果点不存在，会插入新的点。

> Note
>
> `UPSERT VERTEX` 一次只能修改一个 Tag。

`UPSERT VERTEX` 性能远低于 `INSERT`，因为 `UPSERT` 是一组分片级别的读取、修改、写入操作。

> Caution
>
> 并发 `UPSERT` 同一个点或边会报错。

语法

```
UPSERT VERTEX ON <tag> <vid>
SET <update_prop>
[WHEN <condition>]
[YIELD <output>]
```

参数	是否必须	说明	示例
`ON <tag>`	是	指定点的 Tag。要修改的属性必须在这个 Tag 内。	`ON player`
`<vid>`	是	指定要修改或插入的点 ID。	`"player100"`
`SET <update_prop>`	是	指定如何修改属性值。	`SET age = age +1`
`WHEN <condition>`	否	指定过滤条件。	`WHEN name == "Tim"`
`YIELD <output>`	否	指定语句的输出格式。	`YIELD name AS Name`

插入不存在的点

如果点不存在，无论 `WHEN` 子句的条件是否满足，都会插入点，同时执行 `SET` 子句，因此新插入的点的属性值取决于：

- `SET` 子句。
- 属性是否有默认值。

例如：

- 要插入的点包含基于 Tag `player` 的属性 `name` 和 `age`。
- `SET` 子句指定 `age=30`。

不同情况下的属性值如下表。

是否满足 `WHEN` 子句条件	属性是否有默认值	`name` 属性值	`age` 属性值
是	是	默认值	`30`
是	否	`NULL`	`30`
否	是	默认值	`30`
否	否	`NULL`	`30`

示例如下：

```
// 查看三个点是否存在，结果 “Empty set” 表示顶点不存在。
nebula> FETCH PROP ON * "player666", "player667", "player668" YIELD properties(vertex);
+--------------------+
| properties(VERTEX) |
+--------------------+
+--------------------+
Empty set

nebula> UPSERT VERTEX ON player "player666" \
        SET age = 30 \
        WHEN name == "Joe" \
        YIELD name AS Name, age AS Age;
+----------+----------+
| Name     | Age      |
+----------+----------+
| __NULL__ | 30       |
+----------+----------+

nebula> UPSERT VERTEX ON player "player666" \
        SET age = 31 \
        WHEN name == "Joe" \
        YIELD name AS Name, age AS Age;
+----------+-----+
| Name     | Age |
+----------+-----+
| __NULL__ | 30  |
+----------+-----+

nebula> UPSERT VERTEX ON player "player667" \
        SET age = 31 \
        YIELD name AS Name, age AS Age;
+----------+-----+
| Name     | Age |
+----------+-----+
| __NULL__ | 31  |
+----------+-----+

nebula> UPSERT VERTEX ON player "player668" \
        SET name = "Amber", age = age + 1 \
        YIELD name AS Name, age AS Age;
+---------+----------+
| Name    | Age      |
+---------+----------+
| "Amber" | __NULL__ |
+---------+----------+
```

上面最后一个示例中，因为 `age` 没有默认值，插入点时， `age` 默认值为 `NULL` ，执行 `age = age + 1` 后仍为 `NULL` 。如果 `age` 有默认值，则 `age = age + 1` 可以正常执行，例如：

```
nebula> CREATE TAG IF NOT EXISTS player_with_default(name string, age int DEFAULT 20);
Execution succeeded

nebula> UPSERT VERTEX ON player_with_default "player101" \
        SET age = age + 1 \
        YIELD name AS Name, age AS Age;

+----------+-----+
| Name     | Age |
+----------+-----+
| __NULL__ | 21  |
+----------+-----+
```

修改存在的点

如果点存在，且满足 `WHEN` 子句的条件，就会修改点的属性值。

```
nebula> FETCH PROP ON player "player101" YIELD properties(vertex);
+--------------------------------+
| properties(VERTEX)             |
+--------------------------------+
| {age: 36, name: "Tony Parker"} |
+--------------------------------+

nebula> UPSERT VERTEX ON player "player101" \
        SET age = age + 2 \
        WHEN name == "Tony Parker" \
        YIELD name AS Name, age AS Age;
+---------------+-----+
| Name          | Age |
+---------------+-----+
| "Tony Parker" | 38  |
+---------------+-----+
```

如果点存在，但是不满足 `WHEN` 子句的条件，修改不会生效。

```
nebula> FETCH PROP ON player "player101" YIELD properties(vertex);
+--------------------------------+
| properties(VERTEX)             |
+--------------------------------+
| {age: 38, name: "Tony Parker"} |
+--------------------------------+

nebula> UPSERT VERTEX ON player "player101" \
        SET age = age + 2 \
        WHEN name == "Someone else" \
        YIELD name AS Name, age AS Age;
+---------------+-----+
| Name          | Age |
+---------------+-----+
| "Tony Parker" | 38  |
+---------------+-----+
```

最后更新: 2025年8月4日

4.12 边语句

4.12.1 INSERT EDGE

`INSERT EDGE` 语句可以在 NebulaGraph 实例的指定图空间中插入一条或多条边。边是有方向的，从起始点（src_vid）到目的点（dst_vid）。

`INSERT EDGE` 的执行方式为覆盖式插入。如果已有 Edge type、起点、终点、rank 都相同的边，则覆盖原边。

语法

```
INSERT EDGE [IF NOT EXISTS] <edge_type> ( <prop_name_list> ) VALUES
<src_vid> -> <dst_vid>[@<rank>] : ( <prop_value_list> )
[, <src_vid> -> <dst_vid>[@<rank>] : ( <prop_value_list> ), ...];

<prop_name_list> ::=
  [ <prop_name> [, <prop_name> ] ...]

<prop_value_list> ::=
  [ <prop_value> [, <prop_value> ] ...]
```

- `IF NOT EXISTS`：用户可以使用 `IF NOT EXISTS` 关键字检测待插入的边是否存在，只有不存在时，才会插入。

> Note
>
> - `IF NOT EXISTS` 仅检测<边的类型、起始点、目的点和 rank>是否存在，不会检测属性值是否重合。
> - `IF NOT EXISTS` 会先读取一次数据是否存在，因此对性能会有明显影响。

- `<edge_type>`：边关联的 Edge type，只能指定一个 Edge type。Edge type 必须提前创建，详情请参见 CREATE EDGE。
- `<prop_name_list>`：需要设置的属性名称列表。
- `src_vid`：起始点 ID，表示边的起点。
- `dst_vid`：目的点 ID，表示边的终点。
- `rank`：可选项。边的 rank 值。数据类型为 `int`。默认值为 `0`。

> openCypher 兼容性
>
> openCypher 中没有 rank 的概念。

- `<prop_value_list>`：根据 `prop_name_list` 填写属性值。如果属性值和 Edge type 中的数据类型不匹配，会返回错误。如果没有填写属性值，而 Edge type 中对应的属性设置为 `NOT NULL`，也会返回错误。详情请参见 CREATE EDGE。

示例

```
# 插入不包含属性的边。
nebula> CREATE EDGE IF NOT EXISTS e1();
nebula> INSERT EDGE e1 () VALUES "10"->"11":();

# 插入 rank 为 1 的边。
nebula> INSERT EDGE e1 () VALUES "10"->"11"@1:();
```

```
nebula> CREATE EDGE IF NOT EXISTS e2 (name string, age int);
nebula> INSERT EDGE e2 (name, age) VALUES "11"->"13":("n1", 1);

# 一次插入 2 条边。
nebula> INSERT EDGE e2 (name, age) VALUES \
        "12"->"13":("n1", 1), "13"->"14":("n2", 2);
```

```
# 创建失败，因为"a13"不是 int 类型。
nebula> INSERT EDGE e2 (name, age) VALUES "11"->"13":("n1", "a13");
```

一条边可以多次插入属性值，以最后一次为准。

```
# 多次插入属性值。
nebula> INSERT EDGE e2 (name, age) VALUES "11"->"13":("n1", 12);
nebula> INSERT EDGE e2 (name, age) VALUES "11"->"13":("n1", 13);
nebula> INSERT EDGE e2 (name, age) VALUES "11"->"13":("n1", 14);
nebula> FETCH PROP ON e2 "11"->"13" YIELD edge AS e;
+-------------------------------------------+
| e                                         |
+-------------------------------------------+
| [:e2 "11"->"13" @0 {age: 14, name: "n1"}] |
+-------------------------------------------+
```

使用 `IF NOT EXISTS` 插入已存在的边时，不会进行修改。

```
# 插入边。
nebula> INSERT EDGE e2 (name, age) VALUES "14"->"15"@1:("n1", 12);
# 使用 IF NOT EXISTS 修改边，因为边已存在，不会进行修改。
nebula> INSERT EDGE IF NOT EXISTS e2 (name, age) VALUES "14"->"15"@1:("n2", 13);
nebula> FETCH PROP ON e2 "14"->"15"@1 YIELD edge AS e;
+-------------------------------------------+
| e                                         |
+-------------------------------------------+
| [:e2 "14"->"15" @1 {age: 12, name: "n1"}] |
+-------------------------------------------+
```

Note

- NebulaGraph master 允许存在悬挂边（Dangling edge）。因此可以在起点或者终点存在前，先写边；此时就可以通过 `<edgetype>._src` 或 `<edgetype>._dst` 获取到（尚未写入的）点 VID（不建议这样使用）。
- 目前还不能保证操作的原子性，如果失败请重试，否则会发生部分写入。此时读取该数据的行为是未定义的。例如写入操作涉及到多个机器时，可能会出现插入单个边的正反向边只写入成功一个，或者插入多个边时只写入成功一部分，此时会返回报错，请重新执行命令。
- 并发写入同一条边会报 `edge conflict` 错误，可稍后重试。
- `边的 INSERT 速度` 大约是点的 INSERT 速度一半。原因是 INSERT 边会对应 storaged 的两个 INSERT，`INSERT 点` 对应 storaged 的一个 INSERT。

最后更新: 2025年8月4日

4.12.2 DELETE EDGE

`DELETE EDGE` 语句可以删除边。一次可以删除一条或多条边。用户可以结合管道符一起使用，详情请参见管道符。

如果需要删除一个点的所有出边，请删除这个点。详情请参见 DELETE VERTEX。

语法

```
DELETE EDGE <edge_type> <src_vid> -> <dst_vid>[@<rank>] [, <src_vid> -> <dst_vid>[@<rank>] ...]
```

> Caution
>
> 如果不指定 rank，则仅仅删除 rank 为 0 的边。需要删除所有的 rank，见如下示例。

示例

```
nebula> DELETE EDGE serve "player100" -> "team204"@0;
```

```
# 结合管道符，删除两点之间同类型的所有rank的边。
nebula> GO FROM "player100" OVER follow \
        WHERE dst(edge) == "player101" \
        YIELD src(edge) AS src, dst(edge) AS dst, rank(edge) AS rank \
        | DELETE EDGE follow $-.src -> $-.dst @ $-.rank;
```

最后更新: 2025年8月4日

4.12.3 UPDATE EDGE

`UPDATE EDGE` 语句可以修改边上 Edge type 的属性。

NebulaGraph 支持 CAS（compare and swap）操作。

语法

```
UPDATE EDGE ON <edge_type>
<src_vid> -> <dst_vid> [@<rank>]
SET <update_prop>
[WHEN <condition>]
[YIELD <output>]
```

参数	是否必须	说明	示例
`ON <edge_type>`	是	指定 Edge type。要修改的属性必须在这个 Edge type 内。	`ON serve`
`<src_vid>`	是	指定边的起始点 ID。	`"player100"`
`<dst_vid>`	是	指定边的目的点 ID。	`"team204"`
`<rank>`	否	指定边的 rank 值。	`10`
`SET <update_prop>`	是	指定如何修改属性值。	`SET start_year = start_year +1`
`WHEN <condition>`	否	指定过滤条件。如果 `<condition>` 结果为 `false`， `SET` 子句不会生效。	`WHEN end_year < 2010`
`YIELD <output>`	否	指定语句的输出格式。	`YIELD start_year AS Start_Year`

示例

```
// 用 GO 语句查看边的属性值。
nebula> GO FROM "player100" \
        OVER serve \
        YIELD properties(edge).start_year, properties(edge).end_year;
+-----------------------------+---------------------------+
| properties(EDGE).start_year | properties(EDGE).end_year |
+-----------------------------+---------------------------+
| 1997                        | 2016                      |
+-----------------------------+---------------------------+

// 修改属性 start_year 的值，并返回 end_year 和新的 start_year。

nebula> UPDATE EDGE ON serve "player100" -> "team204"@0 \
        SET start_year = start_year + 1 \
        WHEN end_year > 2010 \
        YIELD start_year, end_year;
+------------+----------+
| start_year | end_year |
+------------+----------+
| 1998       | 2016     |
+------------+----------+
```

最后更新: 2025年8月4日

4.12.4 UPSERT EDGE

`UPSERT EDGE` 语句结合 `UPDATE` 和 `INSERT`，如果边存在，会更新边的属性；如果边不存在，会插入新的边。

`UPSERT EDGE` 性能远低于 `INSERT`，因为 `UPSERT` 是一组分片级别的读取、修改、写入操作。

> Caution
>
> 并发 `UPSERT` 同一个点或边会报错。

语法

```
UPSERT EDGE ON <edge_type>
<src_vid> -> <dst_vid> [@rank]
SET <update_prop>
[WHEN <condition>]
[YIELD <properties>]
```

参数	是否必须	说明	示例
`ON <edge_type>`	是	指定 Edge type。要修改的属性必须在这个 Edge type 内。	`ON serve`
`<src_vid>`	是	指定边的起始点 ID。	`"player100"`
`<dst_vid>`	是	指定边的目的点 ID。	`"team204"`
`<rank>`	否	指定边的 rank 值。数据类型为 `int`。	`10`
`SET <update_prop>`	是	指定如何修改属性值。	`SET start_year = start_year +1`
`WHEN <condition>`	否	指定过滤条件。	`WHEN end_year < 2010`
`YIELD <output>`	否	指定语句的输出格式。	`YIELD start_year AS Start_Year`

插入不存在的边

如果边不存在，无论 `WHEN` 子句的条件是否满足，都会插入边，同时执行 `SET` 子句，因此新插入的边的属性值取决于：

- `SET` 子句。
- 属性是否有默认值。

例如：

- 要插入的边包含基于 Edge type `serve` 的属性 `start_year` 和 `end_year`。
- `SET` 子句指定 `end_year = 2021`。

不同情况下的属性值如下表。

是否满足 `WHEN` 子句条件	属性是否有默认值	`start_year` 属性值	`end_year` 属性值
是	是	默认值	`2021`
是	否	`NULL`	`2021`
否	是	默认值	`2021`
否	否	`NULL`	`2021`

示例如下：

```
// 查看如下三个点是否有 serve 类型的出边，结果 “Empty set” 表示没有 serve 类型的出边。
nebula> GO FROM "player666", "player667", "player668" \
        OVER serve \
        YIELD properties(edge).start_year, properties(edge).end_year;
+-----------------------------+---------------------------+
| properties(EDGE).start_year | properties(EDGE).end_year |
+-----------------------------+---------------------------+
+-----------------------------+---------------------------+
Empty set

nebula> UPSERT EDGE on serve \
        "player666" -> "team200"@0 \
        SET end_year = 2021 \
        WHEN end_year == 2010 \
        YIELD start_year, end_year;
+------------+----------+
| start_year | end_year |
+------------+----------+
| __NULL__   | 2021     |
+------------+----------+

nebula> UPSERT EDGE on serve \
        "player666" -> "team200"@0 \
        SET end_year = 2022 \
        WHEN end_year == 2010 \
        YIELD start_year, end_year;
+------------+----------+
| start_year | end_year |
+------------+----------+
| __NULL__   | 2021     |
+------------+----------+

nebula> UPSERT EDGE on serve \
        "player667" -> "team200"@0 \
        SET end_year = 2022 \
        YIELD start_year, end_year;
+------------+----------+
| start_year | end_year |
+------------+----------+
| __NULL__   | 2022     |
+------------+----------+

nebula> UPSERT EDGE on serve \
        "player668" -> "team200"@0 \
        SET start_year = 2000, end_year = end_year + 1 \
        YIELD start_year, end_year;
+------------+----------+
| start_year | end_year |
+------------+----------+
| 2000       | __NULL__ |
+------------+----------+
```

上面最后一个示例中，因为 `end_year` 没有默认值，插入边时， `end_year` 默认值为 `NULL` ，执行 `end_year = end_year + 1` 后仍为 `NULL` 。如果 `end_year` 有默认值，则 `end_year = end_year + 1` 可以正常执行，例如：

```
nebula> CREATE EDGE IF NOT EXISTS serve_with_default(start_year int, end_year int DEFAULT 2010);
Execution succeeded

nebula> UPSERT EDGE on serve_with_default \
        "player668" -> "team200" \
        SET end_year = end_year + 1 \
        YIELD start_year, end_year;
+------------+----------+
| start_year | end_year |
+------------+----------+
| __NULL__   | 2011     |
+------------+----------+
```

修改存在的边

如果边存在，且满足 `WHEN` 子句的条件，就会修改边的属性值。

```
nebula> MATCH (v:player{name:"Ben Simmons"})-[e:serve]-(v2) \
        RETURN e;
+-----------------------------------------------------------------------+
| e                                                                     |
+-----------------------------------------------------------------------+
| [:serve "player149"->"team219" @0 {end_year: 2019, start_year: 2016}] |
+-----------------------------------------------------------------------+

nebula> UPSERT EDGE on serve \
        "player149" -> "team219" \
        SET end_year = end_year + 1 \
        WHEN start_year == 2016 \
        YIELD start_year, end_year;
+------------+----------+
| start_year | end_year |
```

```
+------------+----------+
| 2016       | 2020     |
+------------+----------+
```

如果边存在，但是不满足 `WHEN` 子句的条件，修改不会生效。

```
nebula> MATCH (v:player{name:"Ben Simmons"})-[e:serve]-(v2) \
        RETURN e;
+-----------------------------------------------------------------------+
| e                                                                     |
+-----------------------------------------------------------------------+
| [:serve "player149"->"team219" @0 {end_year: 2020, start_year: 2016}] |
+-----------------------------------------------------------------------+

nebula> UPSERT EDGE on serve \
        "player149" -> "team219" \
        SET end_year = end_year + 1 \
        WHEN start_year != 2016 \
        YIELD start_year, end_year;
+------------+----------+
| start_year | end_year |
+------------+----------+
| 2016       | 2020     |
+------------+----------+
```

最后更新: 2025年8月4日

4.13 原生索引

4.13.1 索引介绍

为了提高查询性能，NebulaGraph 支持为点的 Tag 或 Tag 的某个属性，边的 Edge type 或 Edge type 的某个属性创建索引。索引可以基于指定的 Tag、Edge type、属性查询数据，但是索引本身不存储数据，而是存储数据的位置。

NebulaGraph 支持两种类型索引：原生索引和全文索引。

使用说明

- 索引可以提高查询性能，但是会降低写入性能。
- 索引是执行 `LOOKUP` 语句时用于定位到数据的前置条件，如果没有索引，执行 `LOOKUP` 语句会报错。
- 使用索引时，NebulaGraph 会自动选择最优的索引。
- 具有高选择度的索引，即索引列中不同值的记录数与总记录数的比值较高（例如身份证号的比值为 1）可显著提升查询性能；而对于低选择度的索引（例如国家），查询性能可能不会带来显著提升。

原生索引

原生索引可以基于指定的属性查询数据，有如下特点：

- 包括 Tag 索引和 Edge type 索引。
- 必须手动重建索引（`REBUILD INDEX`）。
- 支持创建同一个 Tag 或 Edge type 的多个属性的索引（复合索引），但是不能跨 Tag 或 Edge type。

原生索引操作

- CREATE INDEX
- SHOW CREATE INDEX
- SHOW INDEXES
- DESCRIBE INDEX
- REBUILD INDEX
- SHOW INDEX STATUS
- DROP INDEX
- LOOKUP
- MATCH
- 地理空间索引

全文索引

全文索引是基于 Elasticsearch 来实现的，用于对字符串属性进行前缀搜索、通配符搜索、正则表达式搜索和模糊搜索，有如下特点：

- 只允许创建一个属性的索引。
- 不支持逻辑操作，例如 `AND`、`OR`、`NOT`。

> Note
>
> 如果需要进行整个字符串的匹配，请使用原生索引。

没有 **NULL** 值索引

不支持对值为 NULL 的属性创建索引。

没有唯一索引

在 Cypher 中，可以通过 Constrains 实现属性值的唯一性限制。在 MySQL 中，可以建立唯一索引来限制某字段只有唯一值。在 nGQL 中没有属性的唯一索引（用户自行保证属性值的唯一性）。

数字、日期和时间类型的范围查询

原生索引还支持对数字、日期和时间类型的属性进行范围查询，不支持其他属性类型的范围查询。

最后更新: 2025年8月4日

4.13.2 CREATE INDEX

前提条件

创建索引之前，请确保相关的 Tag 或 Edge type 已经创建。如何创建 Tag 和 Edge type，请参见 CREATE TAG 和 CREATE EDGE。

如何创建全文索引，请参见部署全文索引。

使用索引必读

索引的概念和使用限制都较为复杂。在使用索引前，请务必阅读以下内容。

`CREATE INDEX` 语句用于对 Tag、EdgeType 或其属性创建原生索引。通常分别称为“Tag 索引”、“Edge type 索引”和“属性索引”。

- Tag 索引和 Edge type 索引应用于和 Tag、Edge type 自身相关的查询，例如用 `LOOKUP` 查找有 Tag `player` 的所有点。
- “属性索引”应用于基于属性的查询，例如基于属性 `age` 找到 `age == 19` 的所有的点。

如果已经为 Tag `T` 的属性 `A` 建立过属性索引 `i_TA` （`T` 的索引为 `i_T`），索引之间的可替代关系如下（Edge type 索引同理）：

- 查询引擎可以使用 `i_TA` 来替代 `i_T`。
- 在 `MATCH`、`LOOKUP` 语句中 `i_T` 可以替代 `i_TA` 查找属性。

> 历史版本兼容性
>
> 在此前的版本中，`LOOKUP` 语句中的 Tag 或 Edge type 索引不可替代属性索引用于属性查找。

使用替代索引进行查询虽然能获得相同的结果，但查询性能会根据选择的索引有所区别。

> Caution
>
> 不要任意在生产环境中使用索引，除非很清楚使用索引对业务的影响。索引会导致写性能大幅下降。
>
> 变长字符串的属性索引长度最长为 256；定长字符串的属性索引长度为索引名本身的长度，无限制。长索引会降低 Storage 服务的扫描性能，以及占用更多内存，建议将索引长度设置为和要被索引的最长字符串相同。

操作步骤

如果必须使用索引，通常按照如下步骤：

1. 初次导入数据至 NebulaGraph 。
2. 创建索引。
3. 重建索引。
4. 使用 LOOKUP 或 MATCH 语句查询数据。不需要（也无法）指定使用哪个索引，NebulaGraph 会自动计算。

> Note
>
> 如果先创建索引再导入数据，会因为写性能的下降导致导入速度极慢。
>
> 日常增量写入时保持 `--disable_auto_compaction = false`。
>
> 新创建的索引并不会立刻生效。创建新的索引并尝试立刻使用（例如 `LOOKUP` 或者 `REBUILD INDEX`）通常会失败（报错 `can't find xxx in the space`）。因为创建步骤是异步实现的，NebulaGraph 要在下一个心跳周期才能完成索引的创建。为确保数据同步，后续操作能顺利进行，请等待 2 个心跳周期（20 秒）。如果需要修改心跳间隔，请为所有配置文件修改参数 `heartbeat_interval_secs`。

> **Danger**
>
> 创建索引，或者删除并再次创建同名索引后，必须 `REBUILD INDEX`。否则无法在 `MATCH` 和 `LOOKUP` 语句中返回这些数据。

语法

```
CREATE {TAG | EDGE} INDEX [IF NOT EXISTS] <index_name> ON {<tag_name> | <edge_name>} ([<prop_name_list>]) [COMMENT '<comment>'];
```

参数	说明
`TAG	EDGE`	指定要创建的索引类型。
`IF NOT EXISTS`	检测待创建的索引是否存在，只有不存在时，才会创建索引。
`<index_name>`	1、索引名。索引名在一个图空间中必须是唯一的。推荐的命名方式为 `i_tagName_propName`。 2、默认情况下，仅支持 1~4 字节的 UTF-8 编码字符，包括英文字母（区分大小写）、数字、中文等。不能包含下划线（_）以外的特殊字符，且不能以数字开头。 3、如果需要使用特殊字符、保留关键字或者以数字开头，请使用反引号（`）包围整个名称。反引号中不能包含英文句号（`.`）。详情参见关键字和保留字。 注意： 1、如果以中文为索引命名，报 `SyntaxError` 错误时，需使用反引号（`）包围中文字符。 2、名称中如需包含反引号（`），使用反斜杠（\）来转义反引号（`），如：\`；如需包含反斜杠（\），反斜杠（\）本身也需转义，如：\\。
`<tag_name>	<edge_name>`	指定索引关联的 Tag 或 Edge 名称。
`<prop_name_list>`	为变长字符串属性创建索引时，必须用 `prop_name(length)` 指定索引长度，索引长度最长为 256；为 Tag 或 Edge type 本身创建索引时，忽略 `<prop_name_list>`。
`COMMENT`	索引的描述。最大为 256 字节。默认无描述。

创建 Tag/Edge type 索引

```
nebula> CREATE TAG INDEX player_index on player();
```

```
nebula> CREATE EDGE INDEX follow_index on follow();
```

为 Tag 或 Edge type 创建索引后，用户可以使用 `LOOKUP` 语句查找 `带有该 Tag 的` 所有点的 VID，或者 `所有该类型的边` 的 `对应起始点 VID、目的点 VID、以及 rank`。详情请参见 LOOKUP。

创建单属性索引

```
nebula> CREATE TAG INDEX IF NOT EXISTS player_index_0 on player(name(10));
```

上述示例是为所有包含 Tag `player` 的点创建属性 `name` 的索引，索引长度为 10。即只使用属性 `name` 的前 10 个字符来创建索引。

```
# 变长字符串需要指定索引长度。
nebula> CREATE TAG IF NOT EXISTS var_string(p1 string);
nebula> CREATE TAG INDEX IF NOT EXISTS var ON var_string(p1(10));

# 定长字符串不需要指定索引长度。
nebula> CREATE TAG IF NOT EXISTS fix_string(p1 FIXED_STRING(10));
nebula> CREATE TAG INDEX IF NOT EXISTS fix ON fix_string(p1);
```

```
nebula> CREATE EDGE INDEX IF NOT EXISTS follow_index_0 on follow(degree);
```

创建复合属性索引

`复合属性索引` 用于查找一个 Tag（或者 Edge type）中的多个属性（的组合）。

```
nebula> CREATE TAG INDEX IF NOT EXISTS player_index_1 on player(name(10), age);
```

Caution

不支持跨 Tag 或 Edge type 创建复合索引。

Note

使用复合属性索引时，遵循"最左匹配原则"，必须从复合属性索引的最左侧开始匹配。

最后更新: 2025年8月4日

4.13.3 SHOW INDEXES

`SHOW INDEXES` 语句可以列出当前图空间内的所有 Tag 和 Edge type（包括属性）的索引。

语法

```
SHOW {TAG | EDGE} INDEXES;
```

示例

```
nebula> SHOW TAG INDEXES;
+------------------+--------------+-----------------+
| Index Name       | By Tag       | Columns         |
+------------------+--------------+-----------------+
| "fix"            | "fix_string" | ["p1"]          |
| "player_index_0" | "player"     | ["name"]        |
| "player_index_1" | "player"     | ["name", "age"] |
| "var"            | "var_string" | ["p1"]          |
+------------------+--------------+-----------------+

nebula> SHOW EDGE INDEXES;
+----------------+----------+---------+
| Index Name     | By Edge  | Columns |
+----------------+----------+---------+
| "follow_index" | "follow" | []      |
+----------------+----------+---------+
```

最后更新: 2025年8月4日

4.13.4 SHOW CREATE INDEX

`SHOW CREATE INDEX` 展示创建 Tag 或者 Edge type 时使用的 nGQL 语句，其中包含索引的详细信息，例如其关联的属性。

语法

```
SHOW CREATE {TAG | EDGE} INDEX <index_name>;
```

示例

用户可以先运行 `SHOW TAG INDEXES` 查看有哪些 Tag 索引，然后用 `SHOW CREATE TAG INDEX` 查看指定索引的创建信息。

```
nebula> SHOW TAG INDEXES;
+------------------+----------+----------+
| Index Name       | By Tag   | Columns  |
+------------------+----------+----------+
| "player_index_0" | "player" | []       |
| "player_index_1" | "player" | ["name"] |
+------------------+----------+----------+

nebula> SHOW CREATE TAG INDEX player_index_1;
+------------------+--------------------------------------------------+
| Tag Index Name   | Create Tag Index                                 |
+------------------+--------------------------------------------------+
| "player_index_1" | "CREATE TAG INDEX `player_index_1` ON `player` ( |
|                  |  `name`(20)                                      |
|                  | )"                                               |
+------------------+--------------------------------------------------+
```

Edge type 索引可以用类似的方法查询：

```
nebula> SHOW EDGE INDEXES;
+----------------+----------+---------+
| Index Name     | By Edge  | Columns |
+----------------+----------+---------+
| "follow_index" | "follow" | []      |
+----------------+----------+---------+

nebula> SHOW CREATE EDGE INDEX follow_index;
+-----------------+-------------------------------------------------+
| Edge Index Name | Create Edge Index                               |
+-----------------+-------------------------------------------------+
| "follow_index"  | "CREATE EDGE INDEX `follow_index` ON `follow` ( |
|                 | )"                                              |
+-----------------+-------------------------------------------------+
```

最后更新: 2025年8月4日

4.13.5 DESCRIBE INDEX

`DESCRIBE INDEX` 语句可以查看指定索引的信息，包括索引的属性名称（Field）和数据类型（Type）。

语法

```
DESCRIBE {TAG | EDGE} INDEX <index_name>;
```

示例

```
nebula> DESCRIBE TAG INDEX player_index_0;
+--------+--------------------+
| Field  | Type               |
+--------+--------------------+
| "name" | "fixed_string(30)" |
+--------+--------------------+

nebula> DESCRIBE TAG INDEX player_index_1;
+--------+--------------------+
| Field  | Type               |
+--------+--------------------+
| "name" | "fixed_string(10)" |
| "age"  | "int64"            |
+--------+--------------------+
```

最后更新: 2025年8月4日

4.13.6 REBUILD INDEX

Danger

- 索引功能不会自动对其创建之前已存在的存量数据生效———在索引重建完成之前，无法基于该索引使用 `LOOKUP` 和 `MATCH` 语句查询到存量数据。
- 索引的重建未完成时，依赖索引的查询仅能使用部分索引，因此不能获得准确结果。

请在创建索引后，选择合适的时间为存量数据重建索引。使用索引的详情请参见 CREATE INDEX。

Caution

通过修改配置文件中的 `rebuild_index_part_rate_limit` 和 `rebuild_index_batch_size` 两个参数，可优化重建索引的速度，另外，更大参数可能会导致更高的内存和网络占用，详情请参见 Storage服务配置。

语法

```
REBUILD {TAG | EDGE} INDEX [<index_name_list>];

<index_name_list>::=
    [index_name [, index_name] ...]
```

- 可以一次重建多个索引，索引名称之间用英文逗号（,）分隔。如果没有指定索引名称，将会重建所有索引。
- 重建完成后，用户可以使用命令 `SHOW {TAG | EDGE} INDEX STATUS` 检查索引是否重建完成。详情请参见 SHOW INDEX STATUS。

示例

```
nebula> CREATE TAG IF NOT EXISTS person(name string, age int, gender string, email string);
nebula> CREATE TAG INDEX IF NOT EXISTS single_person_index ON person(name(10));

# 重建索引，返回任务 ID。
nebula> REBUILD TAG INDEX single_person_index;
+------------+
| New Job Id |
+------------+
| 31         |
+------------+

# 查看索引状态。
nebula> SHOW TAG INDEX STATUS;
+----------------------+--------------+
| Name                 | Index Status |
+----------------------+--------------+
| "single_person_index" | "FINISHED"   |
+----------------------+--------------+

# 也可以使用 SHOW JOB <job_id>查看重建索引的任务状态。
nebula> SHOW JOB 31;
+----------------+---------------------+------------+-------------------------+-------------------------+-------------+
| Job Id(TaskId) | Command(Dest)       | Status     | Start Time              | Stop Time               | Error Code  |
+----------------+---------------------+------------+-------------------------+-------------------------+-------------+
| 31             | "REBUILD_TAG_INDEX" | "FINISHED" | 2021-07-07T09:04:24.000 | 2021-07-07T09:04:24.000 | "SUCCEEDED" |
| 0              | "storaged1"         | "FINISHED" | 2021-07-07T09:04:24.000 | 2021-07-07T09:04:28.000 | "SUCCEEDED" |
| 1              | "storaged2"         | "FINISHED" | 2021-07-07T09:04:24.000 | 2021-07-07T09:04:28.000 | "SUCCEEDED" |
| 2              | "storaged0"         | "FINISHED" | 2021-07-07T09:04:24.000 | 2021-07-07T09:04:28.000 | "SUCCEEDED" |
| "Total:3"      | "Succeeded:3"       | "Failed:0" | "In Progress:0"         | ""                      | ""          |
+----------------+---------------------+------------+-------------------------+-------------------------+-------------+
+----------------+---------------------+------------+-------------------------+-------------------------+-------------+
```

NebulaGraph 创建一个任务去重建索引，因此可以根据返回的任务 ID，通过 `SHOW JOB <job_id>` 语句查看任务状态。详情请参见 SHOW JOB。

最后更新: 2025年8月4日

4.13.7 SHOW INDEX STATUS

`SHOW INDEX STATUS` 语句可以查看索引名称和对应作业的状态。

重建索引时的作业状态包括：

- `QUEUE`：队列中
- `RUNNING`：执行中
- `FINISHED`：已完成
- `FAILED`：失败
- `STOPPED`：停止
- `INVALID`：失效

> Note
>
> 如何创建索引请参见 CREATE INDEX。

语法

```
SHOW {TAG | EDGE} INDEX STATUS;
```

示例

```
nebula> SHOW TAG INDEX STATUS;
+----------------------+--------------+
| Name                 | Index Status |
+----------------------+--------------+
| "player_index_0"     | "FINISHED"   |
| "player_index_1"     | "FINISHED"   |
+----------------------+--------------+
```

最后更新: 2025年8月4日

4.13.8 DROP INDEX

`DROP INDEX` 语句可以删除当前图空间中已存在的索引。

前提条件

执行 `DROP INDEX` 语句需要当前登录的用户拥有指定图空间的 `DROP TAG INDEX` 和 `DROP EDGE INDEX` 权限，否则会报错。

语法

```
DROP {TAG | EDGE} INDEX [IF EXISTS] <index_name>;
```

`IF EXISTS`：检测待删除的索引是否存在，只有存在时，才会删除索引。

示例

```
nebula> DROP TAG INDEX player_index_0;
```

最后更新: 2025年8月4日

4.14 全文索引

4.14.1 全文索引限制

本文介绍全文索引的限制，请在使用全文索引前仔细阅读。

> Caution
>
> 3.6.0 版本重做了全文索引功能，不兼容之前版本的全文索引。如果你想继续使用通配符、正则、模糊匹配等方式，有如下 3 种方式：
>
> - 删除原有全文索引，使用新的方式重建全文索引，使用新的查询语法。
> - 删除原有全文索引，直接用 NebulaGraph 的原生索引和字符串运算符。
> - 继续使用之前版本的 NebulaGraph 及其全文索引功能。

全文索引有如下限制：

- 全文索引当前仅支持 `LOOKUP` 语句。
- 全文索引名称只能包含数字、小写字母、下划线。
- 不同图空间内的全文索引名称不能重复。
- 查询默认返回 10 条数据。可以使用 `LIMIT` 子句返回更多数据，最多可以返回 10000 条。可以修改 Elasticsearch 的参数调整最大返回条数。
- 如果 Tag/Edge type 上存在全文索引，无法删除或修改 Tag/Edge type。
- 属性的类型必须为 `STRING` 或 `FIXED_STRING`。
- 全文索引不支持多个 Tag/Edge type 的搜索。
- 全文索引不支持搜索属性值为 `NULL` 的属性。
- 不支持修改 Elasticsearch 中的索引，只能删除重建。
- 不支持修改分词器，需要删除索引数据后重建索引时指定。
- 确保同时启动了 Elasticsearch 集群和 NebulaGraph，否则可能导致 Elasticsearch 集群写入的数据不完整。
- 从写入 NebulaGraph，到写入 listener，再到写入 Elasticsearch 并创建索引可能需要一段时间。如果访问全文索引时返回未找到索引，可检查索引任务的状态。
- 使用 K8s 方式部署的 NebulaGraph 集群不支持自动部署全文索引，但支持手动部署。

最后更新: 2025年8月4日

4.14.2 部署全文索引

NebulaGraph 的全文索引是基于 Elasticsearch 实现，这意味着用户可以使用 Elasticsearch 全文查询语言来检索想要的内容。全文索引由内置的进程管理，当 listener 集群和 Elasticsearch 集群部署后，内置的进程只能为数据类型为定长字符串或变长字符串的属性创建全文索引。

注意事项

使用全文索引前，请确认已经了解全文索引的使用限制。

部署 Elasticsearch 集群

部署 Elasticsearch 集群请参见 Kubernetes 安装 Elasticsearch 或单机安装 Elasticsearch。目前仅支持 7.x 版本的 Elasticsearch。

> Note
>
> 为了支持外网访问 Elasticsearch，请将 `config/elasticsearch.yml` 中的 `network.host` 设置为 `0.0.0.0` 。

用户可以配置 Elasticsearch 来满足业务需求，如果需要定制 Elasticsearch，请参见 Elasticsearch 官方文档。

登录文本搜索客户端

部署 Elasticsearch 集群之后，可以使用 `SIGN IN` 语句登录 Elasticsearch 客户端。必须使用 Elasticsearch 配置文件中的 IP 地址和端口才能正常连接，同时登录多个客户端，请在多个 `elastic_ip:port` 之间用英文逗号（,）分隔。

语法

```
SIGN IN TEXT SERVICE (<elastic_ip:port>, {HTTP | HTTPS} [,"<username>", "<password>"]) [, (<elastic_ip:port>, ...)];
```

示例

```
nebula> SIGN IN TEXT SERVICE (192.168.8.100:9200, HTTP);
```

> Note
>
> Elasticsearch 默认没有用户名和密码，如果设置了用户名和密码，请在 `SIGN IN` 语句中指定。

> Caution
>
> Elasticsearch 客户端只能登录一次，如有修改，需要 `SIGN OUT` 后重新 `SIGN IN` ，且客户端对全局生效，多个图空间共享相同的 Elasticsearch 客户端。

显示文本搜索客户端

`SHOW TEXT SEARCH CLIENTS` 语句可以列出文本搜索客户端。

语法

```
SHOW TEXT SEARCH CLIENTS;
```

示例

```
nebula> SHOW TEXT SEARCH CLIENTS;
+-----------------+-----------------+------+
| Type            | Host            | Port |
+-----------------+-----------------+------+
| "ELASTICSEARCH" | "192.168.8.100" | 9200 |
+-----------------+-----------------+------+
```

退出文本搜索客户端

`SIGN OUT TEXT SERVICE` 语句可以退出所有文本搜索客户端。

语法

```
SIGN OUT TEXT SERVICE;
```

示例

```
nebula> SIGN OUT TEXT SERVICE;
```

最后更新: 2025年8月4日

4.14.3 部署 Raft listener

全文索引的数据是异步写入 Elasticsearch 集群的。流程是通过 Storage 服务的 Raft listener（简称 listener）这个单独部署的进程，从 Storage 服务读取数据，然后将它们写入 Elasticsearch 集群。

前提条件

- 已经了解全文索引的使用限制。
- 已经部署 NebulaGraph 集群。
- 完成部署 Elasticsearch 集群。
- 准备一台或者多台服务器，以部署 Raft listener。

注意事项

- 请保证 NebulaGraph 各服务（Metad、Storaged、Graphd、listener）有相同的版本。
- 请保证 listener 所在机器的时区与 NebulaGraph 各服务所在机器的时区一致。
- 只能为一个图空间一次性添加所有的 listener 机器。尝试向已经存在有 listener 的图空间再添加新 listener 会失败。因此，需在一个命令语句里完整地添加全部的 listener。

部署流程

第一步：安装 LISTENER 服务

listener 服务与 storaged 服务使用相同的二进制文件，但是二者配置文件不同，进程使用端口也不同。我们在所有需要部署 listener 的服务器上都安装 NebulaGraph，但是不启动服务。详情请参见使用 RPM 或 DEB 安装包安装 NebulaGraph 。

第二步：准备 LISTENER 的配置文件

在 `etc` 目录内将 `nebula-storaged-listener.conf.default` 或 `nebula-storaged-listener.conf.production` 去掉后缀命名为 `nebula-storaged-listener.conf` ，然后修改配置内容。

大部分配置与 Storage 服务的配置文件相同，本文仅介绍差异部分。

名称	预设值	说明
`daemonize`	`true`	是否启动守护进程。
`pid_file`	`pids/nebula-storaged-listener.pid`	记录进程 ID 的文件。
`meta_server_addrs`	-	全部 Meta 服务的 IP（或主机名）和端口。多个 Meta 服务用英文逗号（,）分隔。
`local_ip`	-	listener 服务的 IP（或主机名）。请使用真实的 listener 机器 IP 替换 `127.0.0.1`。
`port`	-	listener 服务的 RPC 守护进程监听端口。
`heartbeat_interval_secs`	`10`	Meta 服务的心跳间隔。单位：秒（s）。
`listener_path`	`data/listener`	listener 的 WAL 目录。只允许使用一个目录。
`data_path`	`data`	出于兼容性考虑，可以忽略此参数。填充一个默认值 `data`。
`part_man_type`	`memory`	部件管理器类型，可选值为 `memory` 和 `meta`。
`rocksdb_batch_size`	`4096`	批处理操作的默认保留字节。
`rocksdb_block_cache`	`4`	BlockBasedTable 的默认块缓存大小。单位：兆字节（MB）。
`engine_type`	`rocksdb`	存储引擎类型，例如 `rocksdb`、`memory` 等。
`part_type`	`simple`	部件类型，例如 `simple`、`consensus` 等。

第三步：启动 LISTENER

在需要部署 listener 集群的安装目录下，执行如下命令启动 listener：

```
./bin/nebula-storaged --flagfile etc/nebula-storaged-listener.conf
```

第四步：添加 LISTENER 到 NEBULAGRAPH 集群

用命令行连接到 NebulaGraph，然后执行 `USE <space>` 进入需要创建全文索引的图空间。然后执行如下命令添加 listener：

```
ADD LISTENER ELASTICSEARCH <listener_ip:port> [,<listener_ip:port>, ...]
```

> **Warning**
>
> listener 必须使用真实的 IP 地址。
>
> 请在一个语句里完整地添加所有 listener。例如：
>
> ```
> nebula> ADD LISTENER ELASTICSEARCH 192.168.8.100:9789,192.168.8.101:9789;
> ```

查看 listener

执行 `SHOW LISTENER` 语句可以列出所有的 listener。

示例

```
nebula> SHOW LISTENER;
+--------+-----------------+-----------------------+-------------+
| PartId | Type            | Host                  | Host Status |
+--------+-----------------+-----------------------+-------------+
| 1      | "ELASTICSEARCH" | ""192.168.8.100":9789" | "ONLINE"    |
| 2      | "ELASTICSEARCH" | ""192.168.8.100":9789" | "ONLINE"    |
| 3      | "ELASTICSEARCH" | ""192.168.8.100":9789" | "ONLINE"    |
+--------+-----------------+-----------------------+-------------+
```

删除 listener

执行 `REMOVE LISTENER ELASTICSEARCH` 语句可以删除图空间的所有 listener。

示例

```
nebula> REMOVE LISTENER ELASTICSEARCH;
```

最后更新: 2025年8月4日

4.14.4 全文搜索

全文搜索是基于全文索引对值为字符串类型的属性进行前缀搜索、通配符搜索、正则表达式搜索和模糊搜索。

在 `LOOKUP` 语句中，使用 `WHERE` 子句指定字符串的搜索条件。

前提条件

请确保已经部署全文索引。详情请参见部署全文索引和部署 listener。

注意事项

使用全文索引前，请确认已经了解全文索引的使用限制。

全文本查询

全文本查询使您能够搜索经过分析的文本字段，使用具有严格语法的解析器，根据提供的查询字符串返回内容。详情参见Query string query。

语法

创建全文索引

```
CREATE FULLTEXT {TAG | EDGE} INDEX <index_name> ON {<tag_name> | <edge_name>} (<prop_name> [,<prop_name>]...) [ANALYZER="<analyzer_name>"];
```

- 创建全文索引时支持多属性的复合索引。
- `<analyzer_name>` 为分词器名称。默认为 `standard` 。使用其他分词器（例如 IK Analysis）需要确保已经提前在 Elasticsearch 中安装对应分词器。

显示全文索引

```
SHOW FULLTEXT INDEXES;
```

重建全文索引

```
REBUILD FULLTEXT INDEX;
```

> Caution
>
> 数据量大时，重建全文索引速度较慢，可以修改 Storage 服务的配置文件（ `nebula-storaged.conf` ）中 `snapshot_send_files=false` 。

删除全文索引

```
DROP FULLTEXT INDEX <index_name>;
```

使用查询选项

```
LOOKUP ON {<tag> | <edge_type>} WHERE ES_QUERY(<index_name>, "<text>") YIELD <return_list> [| LIMIT [<offset>,] <number_rows>];

<return_list>
    <prop_name> [AS <prop_alias>] [, <prop_name> [AS <prop_alias>] ...] [, id(vertex)  [AS <prop_alias>]] [, score() AS <score_alias>]
```

- `index_name` ：索引名称。
- `text` ：搜索条件。WHERE 后只能跟一个 ES_QUERY，所有判断条件必须写在 text 里。详细语法请参见Query string syntax。
- `score()` ：对符合条件的点做 N 度扩展计算出的分数。默认值为 `1.0` 。分数越高，匹配程度越高。返回值默认按照分数从高到低排序。详情参见 Search and Scoring in Lucene。

示例

```
//创建图空间。
nebula> CREATE SPACE IF NOT EXISTS basketballplayer (partition_num=3,replica_factor=1, vid_type=fixed_string(30));

//登录文本搜索客户端。
nebula> SIGN IN TEXT SERVICE (192.168.8.100:9200, HTTP);

//检查是否成功登录。
nebula> SHOW TEXT SEARCH CLIENTS;
+-----------------+-----------------+------+
| Type            | Host            | Port |
+-----------------+-----------------+------+
| "ELASTICSEARCH" | "192.168.8.100" | 9200 |
+-----------------+-----------------+------+

//切换图空间。
nebula> USE basketballplayer;

//添加 listener 到 NebulaGraph 集群。
nebula> ADD LISTENER ELASTICSEARCH 192.168.8.100:9789;

//检查是否成功添加 listener，当状态为 Online 时表示成功添加。
nebula> SHOW LISTENER;
+--------+-----------------+------------------------+-------------+
| PartId | Type            | Host                   | Host Status |
+--------+-----------------+------------------------+-------------+
| 1      | "ELASTICSEARCH" | ""192.168.8.100":9789" | "ONLINE"    |
| 2      | "ELASTICSEARCH" | ""192.168.8.100":9789" | "ONLINE"    |
| 3      | "ELASTICSEARCH" | ""192.168.8.100":9789" | "ONLINE"    |
+--------+-----------------+------------------------+-------------+

//创建 Tag。
nebula> CREATE TAG IF NOT EXISTS player(name string, city string);

//创建单属性全文索引。
nebula> CREATE FULLTEXT TAG INDEX fulltext_index_1 ON player(name) ANALYZER="standard";

//创建多属性全文索引。
nebula> CREATE FULLTEXT TAG INDEX fulltext_index_2 ON player(name,city) ANALYZER="standard";

//重建全文索引。
nebula> REBUILD FULLTEXT INDEX;

//查看全文索引。
nebula> SHOW FULLTEXT INDEXES;
+--------------------+-------------+-------------+--------------+------------+
| Name               | Schema Type | Schema Name | Fields       | Analyzer   |
+--------------------+-------------+-------------+--------------+------------+
| "fulltext_index_1" | "Tag"       | "player"    | "name"       | "standard" |
| "fulltext_index_2" | "Tag"       | "player"    | "name, city" | "standard" |
+--------------------+-------------+-------------+--------------+------------+

//插入测试数据。
nebula> INSERT VERTEX player(name, city) VALUES \
  "Russell Westbrook": ("Russell Westbrook", "Los Angeles"), \
  "Chris Paul": ("Chris Paul", "Houston"),\
  "Boris Diaw": ("Boris Diaw", "Houston"),\
  "David West": ("David West", "Philadelphia"),\
  "Danny Green": ("Danny Green", "Philadelphia"),\
  "Tim Duncan": ("Tim Duncan", "New York"),\
  "James Harden": ("James Harden", "New York"),\
  "Tony Parker": ("Tony Parker", "Chicago"),\
  "Aron Baynes": ("Aron Baynes", "Chicago"),\
  "Ben Simmons": ("Ben Simmons", "Phoenix"),\
  "Blake Griffin": ("Blake Griffin", "Phoenix");

//测试查询
nebula> LOOKUP ON player WHERE ES_QUERY(fulltext_index_1,"Chris") YIELD id(vertex);
+--------------+
| id(VERTEX)   |
+--------------+
| "Chris Paul" |
+--------------+

nebula> LOOKUP ON player WHERE ES_QUERY(fulltext_index_1,"Harden") YIELD properties(vertex);
+----------------------------------------------------------------+
| properties(VERTEX)                                             |
+----------------------------------------------------------------+
| {_vid: "James Harden", city: "New York", name: "James Harden"} |
+----------------------------------------------------------------+

nebula> LOOKUP ON player WHERE ES_QUERY(fulltext_index_1,"Da*") YIELD properties(vertex);
+------------------------------------------------------------------+
| properties(VERTEX)                                               |
+------------------------------------------------------------------+
| {_vid: "David West", city: "Philadelphia", name: "David West"}   |
| {_vid: "Danny Green", city: "Philadelphia", name: "Danny Green"} |
+------------------------------------------------------------------+

nebula> LOOKUP ON player WHERE ES_QUERY(fulltext_index_1,"*b*") YIELD id(vertex);
+---------------------+
| id(VERTEX)          |
+---------------------+
| "Russell Westbrook" |
| "Boris Diaw"        |
```

```
| "Aron Baynes"       |
| "Ben Simmons"       |
| "Blake Griffin"     |
+---------------------+

nebula> LOOKUP ON player WHERE ES_QUERY(fulltext_index_1,"*b*") YIELD id(vertex) | LIMIT 2,3;
+-----------------+
| id(VERTEX)      |
+-----------------+
| "Aron Baynes"   |
| "Ben Simmons"   |
| "Blake Griffin" |
+-----------------+

nebula> LOOKUP ON player WHERE ES_QUERY(fulltext_index_1,"*b*") YIELD id(vertex) | YIELD count(*);
+----------+
| count(*) |
+----------+
| 5        |
+----------+

nebula> LOOKUP ON player WHERE ES_QUERY(fulltext_index_1,"*b*") YIELD id(vertex), score() AS score;
+---------------------+-------+
| id(VERTEX)          | score |
+---------------------+-------+
| "Russell Westbrook" | 1.0   |
| "Boris Diaw"        | 1.0   |
| "Aron Baynes"       | 1.0   |
| "Ben Simmons"       | 1.0   |
| "Blake Griffin"     | 1.0   |
+---------------------+-------+

//对于包含 b 的词的文档，它的得分将乘以加权因子 4，而对于包含 c 的词的文档，使用默认加权因子 1。
nebula> LOOKUP ON player WHERE ES_QUERY(fulltext_index_1,"*b*^4 OR *c*") YIELD id(vertex), score() AS score;
+---------------------+-------+
| id(VERTEX)          | score |
+---------------------+-------+
| "Russell Westbrook" | 4.0   |
| "Boris Diaw"        | 4.0   |
| "Aron Baynes"       | 4.0   |
| "Ben Simmons"       | 4.0   |
| "Blake Griffin"     | 4.0   |
| "Chris Paul"        | 1.0   |
| "Tim Duncan"        | 1.0   |
+---------------------+-------+

//使用多属性全文索引查询，此时条件会在索引的所有属性内进行匹配。
nebula> LOOKUP ON player WHERE ES_QUERY(fulltext_index_2,"*h*") YIELD properties(vertex);
+------------------------------------------------------------------+
| properties(VERTEX)                                               |
+------------------------------------------------------------------+
| {_vid: "Chris Paul", city: "Houston", name: "Chris Paul"}        |
| {_vid: "Boris Diaw", city: "Houston", name: "Boris Diaw"}        |
| {_vid: "David West", city: "Philadelphia", name: "David West"}   |
| {_vid: "James Harden", city: "New York", name: "James Harden"}   |
| {_vid: "Tony Parker", city: "Chicago", name: "Tony Parker"}      |
| {_vid: "Aron Baynes", city: "Chicago", name: "Aron Baynes"}      |
| {_vid: "Ben Simmons", city: "Phoenix", name: "Ben Simmons"}      |
| {_vid: "Blake Griffin", city: "Phoenix", name: "Blake Griffin"}  |
| {_vid: "Danny Green", city: "Philadelphia", name: "Danny Green"} |
+------------------------------------------------------------------+

//使用多属性全文索引查询时，可以对不同的属性指定不同的文本进行查询。
nebula> LOOKUP ON player WHERE ES_QUERY(fulltext_index_2,"name:*b* AND city:Houston") YIELD properties(vertex);
+-----------------------------------------------------------+
| properties(VERTEX)                                        |
+-----------------------------------------------------------+
| {_vid: "Boris Diaw", city: "Houston", name: "Boris Diaw"} |
+-----------------------------------------------------------+

//删除单属性全文索引。
nebula> DROP FULLTEXT INDEX fulltext_index_1;
```

最后更新: 2025年8月4日

4.15 查询调优与终止

4.15.1 EXPLAIN 和 PROFILE

`EXPLAIN` 语句输出 nGQL 语句的执行计划，但不会执行 nGQL 语句。

`PROFILE` 语句执行 nGQL 语句，然后输出执行计划和执行概要。用户可以根据执行计划和执行概要优化查询性能。

执行计划

执行计划由 NebulaGraph 查询引擎中的执行计划器决定。

执行计划器将解析后的 nGQL 语句处理为 `action`。 `action` 是最小的执行单元。典型的 `action` 包括获取指定点的所有邻居、获取边的属性、根据条件过滤点或边等。每个 `action` 都被分配给一个 `operator`。

例如 `SHOW TAGS` 语句分为两个 `action`， `operator` 为 `Start` 和 `ShowTags`。更复杂的 `GO` 语句可能会被处理成 10 个以上的 `action`。

语法

- `EXPLAIN`

```
EXPLAIN [format= {"row" | "dot" | "tck"}] <your_nGQL_statement>;
```

- `PROFILE`

```
PROFILE [format= {"row" | "dot" | "tck"}] <your_nGQL_statement>;
```

输出格式

`EXPLAIN` 或 `PROFILE` 语句的输出有三种格式： `row` （默认）、 `dot` 和 `tck`。用户可以使用 `format` 选项修改输出格式。

`row` 格式

`row` 格式将返回信息输出到一个表格中。

- `EXPLAIN`

```
nebula> EXPLAIN format="row" SHOW TAGS;
Execution succeeded (time spent 327/892 us)

Execution Plan

-----+----------+--------------+----------------+----------------------------------------------------------------------
| id | name     | dependencies | profiling data | operator info                                                        |
-----+----------+--------------+----------------+----------------------------------------------------------------------
|  1 | ShowTags | 0            |                | outputVar: [{"colNames":[],"name":"__ShowTags_1","type":"DATASET"}] |
|    |          |              |                | inputVar:                                                            |
-----+----------+--------------+----------------+----------------------------------------------------------------------
|  0 | Start    |              |                | outputVar: [{"colNames":[],"name":"__Start_0","type":"DATASET"}]    |
-----+----------+--------------+----------------+----------------------------------------------------------------------
```

- `PROFILE`

```
nebula> PROFILE format="row" SHOW TAGS;
+--------+
| Name   |
+--------+
| player |
| team   |
+--------+
Got 2 rows (time spent 2038/2728 us)

Execution Plan

-----+----------+--------------+----------------------------------------------------+----------------------------------------------------------------------
| id | name     | dependencies | profiling data                                     | operator info                                                        |
-----+----------+--------------+----------------------------------------------------+----------------------------------------------------------------------
|  1 | ShowTags | 0            | ver: 0, rows: 1, execTime: 42us, totalTime: 1177us | outputVar: [{"colNames":[],"name":"__ShowTags_1","type":"DATASET"}] |
|    |          |              |                                                    | inputVar:                                                            |
-----+----------+--------------+----------------------------------------------------+----------------------------------------------------------------------
|  0 | Start    |              | ver: 0, rows: 0, execTime: 1us, totalTime: 57us    | outputVar: [{"colNames":[],"name":"__Start_0","type":"DATASET"}]    |
-----+----------+--------------+----------------------------------------------------+----------------------------------------------------------------------
```

参数	说明
`id`	`operator` 的 ID。
`name`	`operator` 的名称。
`dependencies`	当前 `operator` 所依赖的 `operator` 的 ID。
`profiling data`	执行概要文件内容。 `ver` 表示 `operator` 的版本； `rows` 表示 `operator` 输出结果的行数； `execTime` 表示执行 `action` 的时间； `totalTime` 表示执行 `action` 的时间、系统调度时间、排队时间的总和。
`operator info`	`operator` 的详细信息。

`dot` 格式

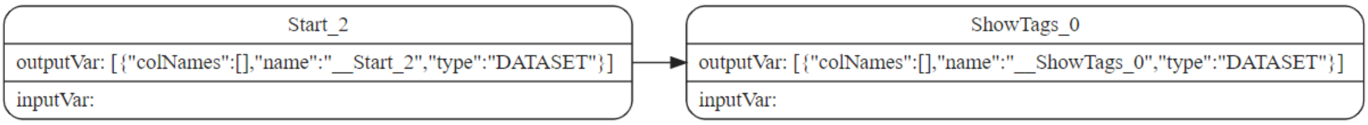
`dot` 格式将返回 DOT 语言的信息，然后用户可以使用 Graphviz 生成计划图。

> **Note**
>
> Graphviz 是一款开源可视化图工具，可以绘制 DOT 语言脚本描述的图。Graphviz 提供一个在线工具，可以预览 DOT 语言文件，并将它们导出为 SVG 或 JSON 等其他格式。详情请参见 Graphviz Online。

```
nebula> EXPLAIN format="dot" SHOW TAGS;
Execution succeeded (time spent 161/665 us)
Execution Plan
---------------------------------------------------------------------------------------------------------------------------------------------  -------------
  plan
---------------------------------------------------------------------------------------------------------------------------------------------  -------------
  digraph exec_plan {
      rankdir=LR;
      "ShowTags_0"[label="ShowTags_0|outputVar: \[\{\"colNames\":\[\],\"name\":\"__ShowTags_0\",\"type\":\"DATASET\"\}\]\l|inputVar:\l",   shape=Mrecord];
      "Start_2"->"ShowTags_0";
      "Start_2"[label="Start_2|outputVar: \[\{\"colNames\":\[\],\"name\":\"__Start_2\",\"type\":\"DATASET\"\}\]\l|inputVar: \l",   shape=Mrecord];
```

```
  }
-----------------------------------------------------------------------------------------------------------------------------   -------------
```

将上述示例的 DOT 语言转换为 Graphviz 图，如下所示。



`tck` 格式

`tck` 格式类似于表格，但是没有边框和行之间的间的分割线。用户可以将结果用在单元测试的测试用例中。 关于 `tck` 格式的测试用例，参见 TCK case。

- `EXPLAIN`

```
nebula> EXPLAIN format="tck" FETCH PROP ON player "player_1","player_2","player_3" YIELD properties(vertex).name as name, properties(vertex).age as age;
Execution succeeded (time spent 261µs/613.718µs)

Execution Plan (optimize time 28 us)


| id | name        | dependencies | profiling data | operator info |
|  2 | Project     | 1            |                |               |
|  1 | GetVertices | 0            |                |               |
|  0 | Start       |              |                |               |


Wed, 22 Mar 2023 23:15:52 CST
```

- `PROFILE`

```
nebula> PROFILE format="tck" FETCH PROP ON player "player_1","player_2","player_3" YIELD properties(vertex).name as name, properties(vertex).age as age;

| name        | age |
| "Piter Park" | 24  |
| "aaa"       | 24  |
| "ccc"       | 24  |

Got 3 rows (time spent 1.474ms/2.19677ms)

Execution Plan (optimize time 41 us)


| id | name        | dependencies | profiling data                                                                                                          | operator info |
|  2 | Project     | 1            | {"rows":3,"version":0}                                                                                                  |               |
|  1 | GetVertices | 0            | {"resp[0]":{"exec":"232(us)","host":"127.0.0.1:9779","total":"758(us)"},"rows":3,"total_rpc":"875(us)","version":0} |               |
|  0 | Start       |              | {"rows":0,"version":0}                                                                                                  |               |


Wed, 22 Mar 2023 23:16:13 CST
```

最后更新: 2025年8月4日

4.15.2 终止查询

`KILL QUERY` 命令可以终止正在执行的查询，常用于终止慢查询。

> **Note**
>
> 仅 God 角色权限可以终止任意查询，其他角色只能终止自己的查询。

语法

```
KILL QUERY (session=<session_id>, plan=<plan_id>);
```

- `session_id`：会话 ID。
- `plan_id`：执行计划 ID。

会话 ID 和执行计划 ID 可以唯一确定一个查询。二者可以通过 SHOW QUERIES 语句获取。

示例

在一个会话中执行命令终止另一个会话中的查询：

```
nebula> KILL QUERY(SESSION=1625553545984255,PLAN=163);
```

另一个会话中的查询会终止，并返回如下信息：

```
[ERROR (-1005)]: ExecutionPlanId[1001] does not exist in current Session.
```

最后更新: 2025年8月4日

4.15.3 终止会话（KILL SESSION）

`KILL SESSION` 语句用于关闭未登出的会话（Session）。

> Note
>
> - 只有 `root` 用户可以终止会话。
> - 执行 `KILL SESSION` 命令后，所有的 Graph 服务同步最新的会话信息需要等待 `2* session_reclaim_interval_secs` 秒，默认等待 `120` 秒。

语法

`KILL SESSION` 语句支持终止单个和多个会话，语法如下：

- 终止单个会话

```
KILL {SESSION|SESSIONS} <SessionId>
```

- `{SESSION|SESSIONS}`：支持 `SESSION` 和 `SESSIONS` 的写法。
- `<SessionId>`：指会话 Session 的 ID。可执行 SHOW SESSIONS 命令查看会话 ID。

- 终止多个会话

```
SHOW SESSIONS
| YIELD $-.SessionId AS sid [WHERE <filter_clause>]
| KILL {SESSION|SESSIONS} $-.sid
```

> Note
>
> `KILL SESSION` 语句支持管道操作，即将 `SHOW SESSIONS` 语句与 `KILL SESSION` 语句结合使用，以终止多个会话。

- `[WHERE <filter_clause>]`：
- 可选项，使用 `WHERE` 子句过滤会话；`<filter_expression>` 指滤过表达式，例如 `WHERE $-.CreateTime < datetime("2022-12-14T18:00:00")`。如果不加该选项，则关闭所有当前会话。
- `WHERE` 子句中支持的过滤项有：`SessionId`、`UserName`、`SpaceName`、`CreateTime`、`UpdateTime`、`GraphAddr`、`Timezone`、`ClientIp`。可以执行 SHOW SESSIONS 命令查看这些过滤项的含义。
- `{SESSION|SESSIONS}`：支持 `SESSION` 和 `SESSIONS` 的写法。

> Caution
>
> 请谨慎使用过滤条件以防误删会话。

示例

- 终止单个会话。

```
nebula> KILL SESSION 1672887983842984
```

- 终止多个会话。
- 终止创建时间小于 `2023-01-05T18:00:00` 的所有会话。

```
nebula> SHOW SESSIONS | YIELD $-.SessionId AS sid WHERE $-.CreateTime < datetime("2023-01-05T18:00:00") | KILL SESSIONS $-.sid
```

- 终止创建时间最早的两个会话。

```
nebula> SHOW SESSIONS | YIELD $-.SessionId AS sid, $-.CreateTime as CreateTime | ORDER BY $-.CreateTime ASC | LIMIT 2 | KILL SESSIONS $-.sid
```

- 终止用户名为 `session_user1` 创建的所有会话。

```
nebula> SHOW SESSIONS | YIELD $-.SessionId as sid WHERE $-.UserName == "session_user1" | KILL SESSIONS $-.sid
```

- 终止所有会话

```
nebula> SHOW SESSIONS | YIELD $-.SessionId as sid | KILL SESSION $-.sid

// 或者
nebula> SHOW SESSIONS | KILL SESSIONS $-.SessionId
```

Caution

终止所有会话时，当前会话也会被终止。请谨慎使用。

最后更新: 2025年8月4日

4.16 作业管理

在 Storage 服务上长期运行的任务称为作业，例如 `COMPACT`、 `FLUSH` 和 `STATS`。 如果图空间的数据量很大，这些作业可能耗时很长。作业管理可以帮助执行、查看、停止和恢复作业。

> Note
>
> 所有作业管理命令都需要先选择图空间后才能执行。

4.16.1 SUBMIT JOB BALANCE LEADER

`SUBMIT JOB BALANCE LEADER` 语句会启动任务均衡分布所有图空间中的 leader。该命令会返回任务 ID。

示例：

```
nebula> SUBMIT JOB BALANCE LEADER;
+------------+
| New Job Id |
+------------+
| 33         |
+------------+
```

4.16.2 SUBMIT JOB COMPACT

`SUBMIT JOB COMPACT` 语句会在当前图空间内触发 RocksDB 的长耗时 `compact` 操作。

`compact` 配置详情请参见 Storage 服务配置。

示例：

```
nebula> SUBMIT JOB COMPACT;
+------------+
| New Job Id |
+------------+
| 40         |
+------------+
```

4.16.3 SUBMIT JOB FLUSH

`SUBMIT JOB FLUSH` 语句将当前图空间内存中的 RocksDB memfile 写入硬盘。

示例：

```
nebula> SUBMIT JOB FLUSH;
+------------+
| New Job Id |
+------------+
| 96         |
+------------+
```

4.16.4 SUBMIT JOB STATS

`SUBMIT JOB STATS` 语句会在当前图空间内启动一个作业，该作业对当前图空间进行统计。作业完成后，用户可以使用 `SHOW STATS` 语句列出统计结果。详情请参见 SHOW STATS。

> Note
>
> 如果存储在 NebulaGraph 中的数据有变化，为了获取最新的统计结果，请重新执行 `SUBMIT JOB STATS`。

示例：

```
nebula> SUBMIT JOB STATS;
+------------+
| New Job Id |
+------------+
| 9          |
+------------+
```

4.16.5 SUBMIT JOB DOWNLOAD/INGEST

`SUBMIT JOB DOWNLOAD HDFS` 和 `SUBMIT JOB INGEST` 命令用于将 SST 文件导入 NebulaGraph 。详情参见导入 SST 文件数据。

`SUBMIT JOB DOWNLOAD HDFS` 语句会下载指定的 HDFS 上的 SST 文件。

`SUBMIT JOB INGEST` 语句会将下载的 SST 文件导入图空间。

示例：

```
nebula> SUBMIT JOB DOWNLOAD HDFS "hdfs://192.168.10.100:9000/sst";
+------------+
| New Job Id |
+------------+
| 10         |
+------------+

nebula> SUBMIT JOB INGEST;
+------------+
| New Job Id |
+------------+
| 11         |
+------------+
```

4.16.6 SHOW JOB

Meta 服务将 `SUBMIT JOB` 请求解析为多个任务，然后分配给进程 nebula-storaged。 `SHOW JOB <job_id>` 语句显示当前图空间内指定作业和相关任务的信息。

`job_id` 在执行 `SUBMIT JOB` 语句时会返回。

示例：

```
nebula> SHOW JOB 8;
+----------------+-----------------+------------+----------------------------+----------------------------+-------------+
| Job Id(TaskId) | Command(Dest)   | Status     | Start Time                 | Stop Time                  | Error Code  |
+----------------+-----------------+------------+----------------------------+----------------------------+-------------+
| 8              | "STATS"         | "FINISHED" | 2022-10-18T08:14:45.000000 | 2022-10-18T08:14:45.000000 | "SUCCEEDED" |
| 0              | "192.168.8.129" | "FINISHED" | 2022-10-18T08:14:45.000000 | 2022-10-18T08:15:13.000000 | "SUCCEEDED" |
| "Total:1"      | "Succeeded:1"   | "Failed:0" | "In Progress:0"            | ""                         | ""          |
+----------------+-----------------+------------+----------------------------+----------------------------+-------------+
```

参数	说明
`Job Id(TaskId)`	第一行显示作业 ID，最后一行显示作业相关的任务总数，其他行显示作业相关的任务 ID。
`Command(Dest)`	第一行显示执行的作业命令名称，最后一行显示作业相关的成功的任务数。其他行显示任务对应的 nebula-storaged 进程。
`Status`	显示作业或任务的状态，最后一行显示作业相关的失败的任务数。详情请参见作业状态。
`Start Time`	显示作业或任务开始执行的时间，最后一行显示作业相关的正在进行的任务数。
`Stop Time`	显示作业或任务结束执行的时间，结束后的状态包括 `FINISHED` 、 `FAILED` 或 `STOPPED` 。
`Error Code`	显示作业或任务的错误码。

作业状态

作业状态的说明如下。

状态	说明
QUEUE	作业或任务在等待队列中。此阶段 `Start Time` 为空。
RUNNING	作业或任务在执行中。 `Start Time` 为该阶段的起始时间。
FINISHED	作业或任务成功完成。 `Stop Time` 为该阶段的起始时间。
FAILED	作业或任务失败。 `Stop Time` 为该阶段的起始时间。
STOPPED	作业或任务停止。 `Stop Time` 为该阶段的起始时间。
REMOVED	作业或任务被删除。

状态转换的说明如下。

```
Queue -- running -- finished -- removed
     \          \                /
      \          \ -- failed -- /
       \          \            /
        \ ---------- stopped -/
```

4.16.7 SHOW JOBS

`SHOW JOBS` 语句列出当前图空间内所有未过期的作业。

作业的默认过期时间为一周。如果需要修改过期时间，请修改 Meta 服务的参数 `job_expired_secs` 。详情请参见 Meta 服务配置。

示例：

```
nebula> SHOW JOBS;
+--------+---------------------+------------+----------------------------+----------------------------+
| Job Id | Command             | Status     | Start Time                 | Stop Time                  |
+--------+---------------------+------------+----------------------------+----------------------------+
| 34     | "STATS"             | "FINISHED" | 2021-11-01T03:32:27.000000 | 2021-11-01T03:32:27.000000 |
| 33     | "FLUSH"             | "FINISHED" | 2021-11-01T03:32:15.000000 | 2021-11-01T03:32:15.000000 |
| 32     | "COMPACT"           | "FINISHED" | 2021-11-01T03:32:06.000000 | 2021-11-01T03:32:06.000000 |
| 31     | "REBUILD_TAG_INDEX" | "FINISHED" | 2021-10-29T05:39:16.000000 | 2021-10-29T05:39:17.000000 |
| 10     | "COMPACT"           | "FINISHED" | 2021-10-26T02:27:05.000000 | 2021-10-26T02:27:05.000000 |
+--------+---------------------+------------+----------------------------+----------------------------+
```

4.16.8 STOP JOB

`STOP JOB <job_id>` 语句可以停止当前图空间内未完成的作业。

示例：

```
nebula> STOP JOB 22;
+---------------+
| Result        |
+---------------+
| "Job stopped" |
+---------------+
```

4.16.9 RECOVER JOB

`RECOVER JOB [<job_id>]` 语句可以重新执行当前图空间内状态为 `FAILED` 、 `STOPPED` 的作业，未指定 `<job_id>` 时，会从最早的作业开始尝试重新执行，并返回已恢复的作业数量。

示例：

```
nebula> RECOVER JOB;
+-------------------+
| Recovered job num |
+-------------------+
```

```
| 5 job recovered   |
+-------------------+
```

4.16.10 FAQ

如何排查作业问题?

`SUBMIT JOB` 操作使用的是 HTTP 端口，请检查 Storage 服务机器上的 HTTP 端口是否正常工作。用户可以执行如下命令调试：

```
curl "http://{storaged-ip}:19779/admin?space={space_name}&op=compact"
```

最后更新: 2025年8月4日

5. 安装部署

5.1 准备编译、安装和运行 NebulaGraph 的环境

本文介绍编译、安装 NebulaGraph 的要求和建议，以及如何预估集群运行所需的资源。

5.1.1 关于存储硬件

NebulaGraph 是针对 NVMe SSD 进行设计和实现的，所有默认参数都是基于 SSD 设备进行调优，要求极高的 IOPS 和极低的 Latency。

- 不建议使用 HDD；因为其 IOPS 性能差，随机寻道延迟高。
- 不要使用远端存储设备（如 NAS 或 SAN），不要外接基于 HDFS 或者 Ceph 的虚拟硬盘。
- 不建议配置独立磁盘冗余阵列（RAID）。NebulaGraph本身提供了多副本机制，如果配置 RAID，会造成资源浪费。
- 使用本地 SSD 设备；或 AWS Provisioned IOPS SSD 或等价云产品。

5.1.2 关于 CPU 架构

从 3.0.2 开始，NebulaGraph 在 Docker Hub 上的 Docker 支持 ARM64 架构。社区用户可以在 ARM macOS 的 Docker Desktop 上或者 ARM Linux Server 上运行容器化的 NebulaGraph 。

> Caution
>
> 不建议在 Windows 上使用 Docker Desktop，因为 Windows 上的 Docker Desktop 性能较差。详情请参见 #12401。

5.1.3 编译源码要求

硬件要求

单个机器的硬件要求如下。

类型	要求
CPU 架构	x86_64
内存	4 GB
硬盘	10 GB，SSD

操作系统要求

当前仅支持在 Linux 系统中编译 NebulaGraph，建议使用内核版本为 `4.15` 及以上版本的 Linux 系统。

> Note
>
> 在内核版本低于要求的 Linux 系统中安装 NebulaGraph 可使用 RPM、DEB 或者 TAR 文件。

软件要求

软件版本需要如下表所示，如果版本不符合要求，请按照安装编译所需软件中的步骤进行操作。

软件名称	版本	备注
glibc	2.17 及以上	执行命令 `ldd --version` 检查版本。
make	任意稳定版本	-
m4	任意稳定版本	-
git	任意稳定版本	-
wget	任意稳定版本	-
unzip	任意稳定版本	-
xz	任意稳定版本	-
readline-devel	任意稳定版本	-
ncurses-devel	任意稳定版本	-
zlib-devel	任意稳定版本	-
g++	8.5.0 及以上	执行命令 `g++ -v` 检查版本。
cmake	3.14.0 及以上	执行命令 `cmake --version` 检查版本。
curl	任意稳定版本	-
redhat-lsb-core	任意稳定版本	-
libstdc++-static	任意稳定版本	仅在 CentOS 8+、RedHat 8+、Fedora 中需要。
libasan	任意稳定版本	仅在 CentOS 8+、RedHat 8+、Fedora 中需要。
bzip2	任意稳定版本	-

其他第三方软件将在安装（cmake）阶段自动下载并安装到 `build` 目录中。

安装编译所需软件

如果部分依赖软件缺失或者版本不满足要求，根据如下步骤手动安装。可根据实际情况删减命令中要安装的软件、跳过无需执行的步骤。

1. 安装依赖包。

- CentOS、RedHat、Fedora 用户请执行如下命令：

```
$ yum update
$ yum install -y make \
        m4 \
        git \
        wget \
        unzip \
        xz \
        readline-devel \
        ncurses-devel \
        zlib-devel \
        gcc \
        gcc-c++ \
        cmake \
        curl \
        redhat-lsb-core \
        bzip2
// 仅 CentOS 8+、RedHat 8+、Fedora 需要安装 libstdc++-static 和 libasan。
$ yum install -y libstdc++-static libasan
```

- Debian 和 Ubuntu 用户请执行如下命令：

```
$ apt-get update
$ apt-get install -y make \
         m4 \
         git \
         wget \
         unzip \
         xz-utils \
         curl \
         lsb-core \
         build-essential \
         libreadline-dev \
         ncurses-dev \
         cmake \
         bzip2
```

2. 检查主机上的 G++ 和 CMake 版本是否正确。版本信息请参见软件要求。

```
$ g++ --version
$ cmake --version
```

如果版本正确，则软件依赖已准备完毕，忽略后续步骤；如果不正确，根据不符合版本要求的软件执行后续步骤。

3. 如果 CMake 或 g++ 版本不符合要求，访问官网以获取符合需要的版本。

5.1.4 测试环境要求

硬件要求

单个机器的硬件要求如下。

类型	要求
CPU 架构	x86_64
CPU 核数	4
内存	8 GB
硬盘	100 GB，SSD

操作系统要求

当前仅支持在 Linux 系统中安装 NebulaGraph，建议在测试环境中使用内核版本为 `3.9` 及以上版本的 Linux 系统。

服务架构建议

进程	建议数量
metad（meta 数据服务进程）	1
storaged（存储服务进程）	⩾1
graphd（查询引擎服务进程）	⩾1

例如单机测试环境，用户可以在机器上部署 1 个 metad、1 个 storaged 和 1 个 graphd 进程。

对于更常见的测试环境，例如三台机器构成的集群，用户可以按照如下方案部署 NebulaGraph 。

机器名称	metad 进程数量	storaged 进程数量	graphd 进程数量
A	1	1	1
B	-	1	1
C	-	1	1

5.1.5 生产环境运行要求

硬件要求

单个机器的硬件要求如下。

类型	要求
CPU 架构	x86_64
CPU 核数	48
内存	256 GB
硬盘	2 * 1.6 TB，NVMe SSD

操作系统要求

当前仅支持在 Linux 系统中安装 NebulaGraph，建议在生产环境中使用内核版本为 `3.9` 及以上版本的 Linux 系统。

用户可以通过调整一些内核参数来提高 NebulaGraph 性能，详情请参见内核配置。

服务架构建议

> **Danger**
>
> 不要跨机房部署单个集群（企业版支持跨机房集群间同步）。

进程	数量
metad（meta 数据服务进程）	3
storaged（存储服务进程）	⩾3
graphd（查询引擎服务进程）	⩾3

有且仅有 3 个 metad 进程，每个 metad 进程会自动创建并维护 meta 数据的一个副本。

storaged 进程的数量不会影响图空间副本的数量。

用户可以在一台机器上部署多个不同进程，例如五台机器构成的集群，用户可以按照如下方案部署 NebulaGraph 。

机器名称	metad 进程数量	storaged 进程数量	graphd 进程数量
A	1	1	1
B	1	1	1
C	1	1	1
D	-	1	1
E	-	1	1

5.1.6 NebulaGraph 资源要求

用户可以预估一个 3 副本 NebulaGraph 集群所需的内存、硬盘空间和分区数量。

资源	单位	计算公式	说明
硬盘空间	Bytes	`点和边的总数` * `属性的平均字节大小` * 7.5 * 120%	由于边存在存储放大现象，所以需要 `点和边的总数` * `属性的平均字节大小` * 7.5 的空间，详情请参见切边与存储放大。
内存	Bytes	[`点和边的总数` * 16 + `RocksDB 实例数量` * (`write_buffer_size` * `max_write_buffer_number`) + `块缓存大小`] * 120%	`点和边的总数` * 16 是 BloomFilter 需要占用的内存空间，`write_buffer_size` 和 `max_write_buffer_number` 是 RocksDB 内存相关参数，详情请参见 MemTable。块缓存大小请参见 Memory usage in RocksDB。
分区数量	-	`集群硬盘数量` * `disk_partition_num_multiplier`	`disk_partition_num_multiplier` 是一个用于衡量硬盘性能的整数，取值范围 2~20。建议在计算 SSD 硬盘的分区数量时使用 20 做参数值，HDD 硬盘使用 2。

- 问题 1：为什么硬盘空间的预估公式中需要乘以 7.5?

  答：因为根据测试值，相比原始数据文件（csv），单副本所占用的空间约为之前的 2.5 倍；另外索引会另外占用硬盘空间，每个被索引的点或边占用 16 字节的内存，索引的占用硬盘空间可按经验估算为：被索引的点或边总数 * 50 字节。

- 问题 2：为什么磁盘空间和内存都要乘以 120%?

  答：额外的 20% 用于缓冲。

- 问题 3：如何获取 RocksDB 实例数量?

  答：对于社区版 NebulaGraph，每个图空间对应一个 RocksDB 实例，并且 `--data_path` 选项（`etc` 目录下的 `nebula-storaged.conf` 文件中）中的每个目录对应一个 RocksDB 实例。即，RocksDB 实例数量 = 图空间总数 * 目录总数。

> **Note**
>
> 用户可以在配置文件 `nebula-storaged.conf` 中添加 `--enable_partitioned_index_filter=true` 来降低 bloom 过滤器占用的内存大小，但是在某些随机寻道（random-seek）的情况下，可能会降低读取性能。

> **Caution**
>
> 每个 RocksDB 实例即使还未写入任何数据时，仍会占用 70M 左右的磁盘空间。一个分区对应一个 RocksDB 实例，当分区设置特别多时，例如 100，图空间创建后即占用了大量磁盘空间。

最后更新: 2025年8月4日

5.2 编译安装

5.2.1 使用源码安装 NebulaGraph

使用源码安装 NebulaGraph 允许自定义编译和安装设置，并测试最新特性。

前提条件

- 准备正确的编译环境。参见软硬件要求和安装三方库依赖包。

> Note
>
> 暂不支持离线编译 NebulaGraph 。

- 待安装 NebulaGraph 的主机可以访问互联网。

安装步骤

1. 克隆 NebulaGraph 的源代码到主机。

- [推荐] 如果需要安装master版本的 NebulaGraph，执行如下命令：

```
$ git clone --branch release-3.6 https://github.com/vesoft-inc/nebula.git
```

- 如果需要安装最新的开发版本用于测试，执行如下命令克隆 master 分支的代码：

```
$ git clone https://github.com/vesoft-inc/nebula.git
```

2. 进入 `nebula/third-party` 目录，安装 NebulaGraph 依赖的第三方库。

```
$ cd nebula/third-party
$ ./install-third-party.sh
```

3. 返回 `nebula` 目录，创建目录 `build` 并进入该目录。

```
$ cd ..
$ mkdir build && cd build
```

4. 使用 CMake 生成 makefile 文件。

> Note
>
> 默认安装路径为 `/usr/local/nebula` ，如果需要修改路径，请在下方命令内增加参数 `-DCMAKE_INSTALL_PREFIX=<installation_path>` 。
>
> 更多 CMake 参数说明，请参见 CMake 参数。

```
$ cmake -DCMAKE_INSTALL_PREFIX=/usr/local/nebula -DENABLE_TESTING=OFF -DCMAKE_BUILD_TYPE=Release ..
```

5. 编译 NebulaGraph 。

> Note
>
> 检查软硬件要求和安装三方库依赖包。

为了适当地加快编译速度，可以使用选项 `-j` 并行编译。并行数量 `N` 建议为\(\min(\text{CPU}核数，\frac{内存 (GB)}{2})\)。

```
$ make -j{N} # E.g., make -j2
```

6. 安装 NebulaGraph 。

```
$ sudo make install
```

> Note
>
> 安装目录下 `etc/` 目录中（默认为 `/usr/local/nebula/etc` ）的配置文件为参考模版，用户可以根据需要创建自己的配置文件。如果要使用 `script` 目录下的脚本，启动、停止、重启、中止和查看服务，配置文件需要命名为 `nebula-graph.conf` ， `nebula-metad.conf` 和 `nebula-storaged.conf` 。

更新 **master** 版本

master 分支的代码更新速度快，如果安装了 master 分支对应的开发版 NebulaGraph，可根据以下步骤更新版本。

1. 在目录 `nebula` 中，执行命令 `git pull upstream master` 更新源码。
2. 在目录 `nebula/build` 中，重新执行 `make -j{N}` 和 `make install` 。

下一步

管理 NebulaGraph 服务

CMake 参数

使用方法

```
$ cmake -D<variable>=<value> ...
```

下文的 CMake 参数可以在配置 (CMake) 阶段用来调整编译设置。

CMAKE_INSTALL_PREFIX

`CMAKE_INSTALL_PREFIX` 指定 NebulaGraph 服务模块、脚本和配置文件的安装路径，默认路径为 `/usr/local/nebula` 。

ENABLE_WERROR

默认值为 `ON` ，表示将所有警告（warning）变为错误（error）。如果有必要，用户可以设置为 `OFF` 。

ENABLE_TESTING

默认值为 `ON` ，表示单元测试服务由 NebulaGraph 服务构建。如果只需要服务模块，可以设置为 `OFF` 。

ENABLE_ASAN

默认值为 `OFF` ，表示关闭内存问题检测工具 ASan（AddressSanitizer）。该工具是为 NebulaGraph 开发者准备的，如果需要开启，可以设置为 `ON` 。

CMAKE_BUILD_TYPE

`CMAKE_BUILD_TYPE` 控制 NebulaGraph 的 build 方法，取值说明如下：

- `Debug`

  `CMAKE_BUILD_TYPE` 的默认值，表示 build 过程中只记录 debug 信息，不使用优化选项。

- `Release`

  build 过程中使用优化选项，不记录 debug 信息。

- `RelWithDebInfo`

  build 过程中既使用优化选项，也记录 debug 信息。

- `MinSizeRel`

  build 过程中仅通过优化选项控制代码大小，不记录 debug 信息。

ENABLE_INCLUDE_WHAT_YOU_USE

默认值为 `OFF`。当其值为 `ON` 且系统中安装了include-what-you-use，系统将在生成 makefile 过程中报告工程源码中包含的冗余头文件。

NEBULA_USE_LINKER

指定链接程序的方式：

- 默认值为 `bfd`，表示使用 ld.bfd 链接程序。
- 如果系统中安装了 lld 链接器, 可设置参数值为 `lld`，表示使用 ld.lld 链接程序。
- 如果系统中安装了 gold 链接器，可设为 `gold`，表示使用 ld.gold 链接程序。

CMAKE_C_COMPILER/CMAKE_CXX_COMPILER

通常情况下，CMake 会自动查找并使用主机上的 C/C++编译器，但是如果编译器没有安装在标准路径，或者想使用其他编译器，请执行如下命令指定目标编译器的安装路径：

```
$ cmake -DCMAKE_C_COMPILER=<path_to_gcc/bin/gcc> -DCMAKE_CXX_COMPILER=<path_to_gcc/bin/g++> ..
$ cmake -DCMAKE_C_COMPILER=<path_to_clang/bin/clang> -DCMAKE_CXX_COMPILER=<path_to_clang/bin/clang++> ..
```

ENABLE_CCACHE

`ENABLE_CCACHE` 默认值为 `ON`，表示使用 Ccache（compiler cache）工具加速编译。

如果想要禁用 ccache，仅仅设置 `ENABLE_CCACHE=OFF` 是不行的，因为在某些平台上，ccache 会代理当前编译器，因此还需要设置环境变量 `export CCACHE_DISABLE=true`，或者在文件 `~/.ccache/ccache.conf` 中添加 `disable=true`。更多信息请参见 ccache official documentation。

NEBULA_THIRDPARTY_ROOT

`NEBULA_THIRDPARTY_ROOT` 指定第三方软件的安装路径，默认路径为 `/opt/vesoft/third-party`。

问题排查

如果出现编译失败，请参考以下建议：

1. 检查操作系统版本是否符合要求、内存和硬盘空间是否足够。
2. 检查 third-party 是否正确安装。
3. 使用 `make -j1` 降低编译并发度。

最后更新: 2025年8月4日

5.2.2 使用 Docker 编译 NebulaGraph

NebulaGraph 的源代码是使用 C++ 编写的，编译 NebulaGraph 需要安装一些依赖，这些依赖可能会与宿主机操作系统的依赖冲突，导致编译失败。为了避免这种情况，我们可以使用 Docker 来编译 NebulaGraph。NebulaGraph 提供整个编译环境的 Docker 镜像，可以帮助我们快速搭建编译环境并编译 NebulaGraph，同时避免了与宿主机操作系统的依赖冲突。本文介绍如何使用 Docker 编译 NebulaGraph 图数据库。

准备工作

在开始编译 NebulaGraph 之前，确保已经完成以下准备工作：

1. 安装 Docker：确保用户的系统已经安装了 Docker。
2. 克隆 NebulaGraph 源代码：将 NebulaGraph 的源代码克隆到本地。你可以使用 Git 命令来克隆代码仓库：

```
git clone --branch release-3.6 https://github.com/vesoft-inc/nebula.git
```

这将会将 NebulaGraph 的源代码克隆到当前目录的 `nebula` 子目录中。

步骤

1. 拉取 NebulaGraph 编译镜像。

```
docker pull vesoft/nebula-dev:ubuntu2004
```

这里我们使用的是 NebulaGraph 的官方编译镜像，版本号为 `ubuntu2004`，你也可以根据需要使用特定的版本号。详情参见nebula-dev-docker。

2. 运行编译容器。

现在，我们可以在 Docker 容器中编译 NebulaGraph。执行以下命令：

```
docker run -ti \
  --security-opt seccomp=unconfined \
  -v "$PWD":/home \
  -w /home \
  --name nebula_dev \
  vesoft/nebula-dev:ubuntu2004 \
  bash
```

- `--security-opt seccomp=unconfined`：为了避免 Docker 容器中的 CMake 编译过程中出现 `Killed` 错误，不限制容器进程可以进行的系统调用。
- `-v "$PWD":/home`：表示当前的 NebulaGraph 代码本地的路径会被挂载到容器内部的 `/home` 目录。
- `-w /home`：将容器的工作目录设置为 `/home`，在容器中运行任何命令都将以该目录作为当前目录。
- `--name nebula_dev`：为容器指定一个名称，方便管理和操作。
- `vesoft/nebula-dev:ubuntu2004`：使用 `vesoft/nebula-dev` 编译镜像的 `ubuntu2004` 版本。
- `bash`：在容器中运行 `bash` 命令，进入容器的交互式终端。

成功运行以上命令后，将会自动进入容器的交互式终端。若退出容器，可执行 `docker exec -ti nebula_dev bash` 重新进入容器。

3. 在容器内编译 NebulaGraph。

a. 进入 NebulaGraph 源代码目录

```
cd nebula
```

b. 创建目录build并进入该目录。

```
mkdir build && cd build
```

c. 使用 CMake 生成 makefile 文件。

```
cmake -DCMAKE_CXX_COMPILER=$TOOLSET_CLANG_DIR/bin/g++ -DCMAKE_C_COMPILER=$TOOLSET_CLANG_DIR/bin/gcc -DENABLE_WERROR=OFF -DCMAKE_BUILD_TYPE=Debug -DENABLE_TESTING=OFF ..
```

有关 CMake 的 更多信息，参见 CMake 参数。

d. 执行编译命令。

```
# 根据服务器的空闲 CPU 核心数，设置合理的并行编译的线程数。比如，如果服务器有 2 个空闲 CPU 核心，可以设置为 make -j2。
make -j2
```

这将会开始编译 NebulaGraph，编译过程可能会花费一些时间，取决于用户的系统性能。

4. 将生产的可执行文件和库文件安装到 `/usr/local/nebula` 目录下。

编译成功后，在容器中的 `/home/nebula/build` 目录下将生成 NebulaGraph 的可执行文件和库文件。为了方便管理，我们可以将这些文件安装到 `/usr/local/nebula` 目录下。执行以下命令：

```
make install
```

完成以上步骤后，NebulaGraph 将会被同步编译并安装到宿主机的 `/usr/local/nebula` 目录下。

后续步骤

- 启动 NebulaGraph 服务
- 连接 NebulaGraph 服务

最后更新: 2025年8月4日

5.3 本地单机安装

5.3.1 使用 RPM 或 DEB 包安装 NebulaGraph

RPM 和 DEB 是 Linux 系统下常见的两种安装包格式，本文介绍如何使用 RPM 或 DEB 文件在一台机器上快速安装 NebulaGraph 。

> Note
>
> 部署 NebulaGraph 集群的方式参见使用 RPM/DEB 包部署集群。

前提条件

- 安装 `wget` 工具。

下载安装包

> Note
>
> - 当前仅支持在 Linux 系统下安装 NebulaGraph，且仅支持 CentOS 7.x、CentOS 8.x、Ubuntu 16.04、Ubuntu 18.04、Ubuntu 20.04 操作系统。
> - 如果用户使用的是国产化的 Linux 操作系统，请安装企业版 NebulaGraph 。

阿里云 **OSS** 下载

- 下载 release 版本

  URL 格式如下：

```
//Centos 7
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.el7.x86_64.rpm

//Centos 8
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.el8.x86_64.rpm

//Ubuntu 1604
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.ubuntu1604.amd64.deb

//Ubuntu 1804
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.ubuntu1804.amd64.deb

//Ubuntu 2004
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.ubuntu2004.amd64.deb
```

  例如要下载适用于 `Centos 7.5` 的 `master` 安装包：

```
wget https://oss-cdn.nebula-graph.com.cn/package/master/nebula-graph-master.el7.x86_64.rpm
wget https://oss-cdn.nebula-graph.com.cn/package/master/nebula-graph-master.el7.x86_64.rpm.sha256sum.txt
```

  下载适用于 `ubuntu 1804` 的 `master` 安装包：

```
wget https://oss-cdn.nebula-graph.com.cn/package/master/nebula-graph-master.ubuntu1804.amd64.deb
wget https://oss-cdn.nebula-graph.com.cn/package/master/nebula-graph-master.ubuntu1804.amd64.deb.sha256sum.txt
```

- 下载日常开发版本 (nightly)

    Danger

    - nightly 版本通常用于测试新功能、新特性，请不要在生产环境中使用 nightly 版本。
    - nightly 版本不保证每日都能完整发布，也不保证是否会更改文件名。

    URL 格式如下：

```
//Centos 7
https://oss-cdn.nebula-graph.com.cn/package/nightly/<yyyy.mm.dd>/nebula-graph-<yyyy.mm.dd>-nightly.el7.x86_64.rpm

//Centos 8
https://oss-cdn.nebula-graph.com.cn/package/nightly/<yyyy.mm.dd>/nebula-graph-<yyyy.mm.dd>-nightly.el8.x86_64.rpm

//Ubuntu 1604
https://oss-cdn.nebula-graph.com.cn/package/nightly/<yyyy.mm.dd>/nebula-graph-<yyyy.mm.dd>-nightly.ubuntu1604.amd64.deb

//Ubuntu 1804
https://oss-cdn.nebula-graph.com.cn/package/nightly/<yyyy.mm.dd>/nebula-graph-<yyyy.mm.dd>-nightly.ubuntu1804.amd64.deb

//Ubuntu 2004
https://oss-cdn.nebula-graph.com.cn/package/nightly/<yyyy.mm.dd>/nebula-graph-<yyyy.mm.dd>-nightly.ubuntu2004.amd64.deb
```

    例如要下载 `2021.11.24` 适用于 `Centos 7.5` 的 `2.x` 安装包：

```
wget https://oss-cdn.nebula-graph.com.cn/package/nightly/2021.11.24/nebula-graph-2021.11.24-nightly.el7.x86_64.rpm
wget https://oss-cdn.nebula-graph.com.cn/package/nightly/2021.11.24/nebula-graph-2021.11.24-nightly.el7.x86_64.rpm.sha256sum.txt
```

    要下载 `2021.11.24` 适用于 `Ubuntu 1804` 的 `2.x` 安装包：

```
wget https://oss-cdn.nebula-graph.com.cn/package/nightly/2021.11.24/nebula-graph-2021.11.24-nightly.ubuntu1804.amd64.deb
wget https://oss-cdn.nebula-graph.com.cn/package/nightly/2021.11.24/nebula-graph-2021.11.24-nightly.ubuntu1804.amd64.deb.sha256sum.txt
```

安装 NebulaGraph

- 安装 RPM 包

```
$ sudo rpm -ivh --prefix=<installation_path> <package_name>
```

    `--prefix` 为可选项，用于指定安装路径。如不设置，系统会将 NebulaGraph 安装到默认路径 `/usr/local/nebula/`。

    例如，要在默认路径下安装master版本的 RPM 包，运行如下命令：

```
sudo rpm -ivh nebula-graph-master.el7.x86_64.rpm
```

- 安装 DEB 包

```
$ sudo dpkg -i <package_name>
```

    Note

    使用 DEB 包安装 NebulaGraph 时不支持自定义安装路径。默认安装路径为 `/usr/local/nebula/`。

    例如安装master版本的 DEB 包：

```
sudo dpkg -i nebula-graph-master.ubuntu1804.amd64.deb
```

后续操作

- 启动 NebulaGraph

- 连接 NebulaGraph

最后更新: 2025年8月4日

5.3.2 使用 tar.gz 文件安装 NebulaGraph

用户可以下载打包好的 tar.gz 文件快速安装 NebulaGraph 。

> Note
>
> - NebulaGraph 从 2.6.0 版本起提供 tar.gz 文件。
> - 当前仅支持在 Linux 系统下安装 NebulaGraph，且仅支持 CentOS 7.x、CentOS 8.x、Ubuntu 16.04、Ubuntu 18.04、Ubuntu 20.04 操作系统。
> - 如果用户使用的是国产化的 Linux 操作系统，请安装 NebulaGraph 。

操作步骤

1. 使用如下地址下载 NebulaGraph 的 tar.gz 文件。

   下载前需将 `<release_version>` 替换为需要下载的版本。

```
//Centos 7
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.el7.x86_64.tar.gz
//Checksum
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.el7.x86_64.tar.gz.sha256sum.txt

//Centos 8
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.el8.x86_64.tar.gz
//Checksum
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.el8.x86_64.tar.gz.sha256sum.txt

//Ubuntu 1604
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.ubuntu1604.amd64.tar.gz
//Checksum
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.ubuntu1604.amd64.tar.gz.sha256sum.txt

//Ubuntu 1804
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.ubuntu1804.amd64.tar.gz
//Checksum
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.ubuntu1804.amd64.tar.gz.sha256sum.txt

//Ubuntu 2004
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.ubuntu2004.amd64.tar.gz
//Checksum
https://oss-cdn.nebula-graph.com.cn/package/<release_version>/nebula-graph-<release_version>.ubuntu2004.amd64.tar.gz.sha256sum.txt
```

   例如，要下载适用于 `CentOS 7.5` 的 NebulaGraph release-3.6 tar.gz 文件，运行以下命令：

```
wget https://oss-cdn.nebula-graph.com.cn/package/master/nebula-graph-master.el7.x86_64.tar.gz
```

2. 解压 tar.gz 文件到 NebulaGraph 安装目录。

```
tar -xvzf <tar.gz_file_name> -C <install_path>
```

- `tar.gz_file_name` 表示 tar.gz 文件的名称。
- `install_path` 表示安装路径。

   例如：

```
tar -xvzf nebula-graph-master.el7.x86_64.tar.gz -C /home/joe/nebula/install
```

3. 修改配置文件名以应用配置。

   进入解压出的目录，将子目录 `etc` 中的文件 `nebula-graphd.conf.default` 、 `nebula-metad.conf.default` 和 `nebula-storaged.conf.default` 重命名，删除 `.default` ，即可应用 NebulaGraph 的默认配置。

> Note
>
> 如需修改更多配置，参见配置管理。

至此，NebulaGraph 安装完毕。

下一步

管理 NebulaGraph 服务

最后更新: 2025年8月4日

5.3.3 存算合并版 NebulaGraph

存算合并版 NebulaGraph 将存储服务（Meta 和 Storage）和计算服务（Graph）合并至一个进程，用于部署在单台机器上。本文介绍存算合并版 NebulaGraph 的使用场景、安装步骤等。

> **Danger**
>
> 存算合并版 NebulaGraph 不用于生产环境。

背景信息

传统的 NebulaGraph 架构由 3 个服务构成，每个服务都有可执行的二进制文件和对应的进程，进程之间通过 RPC 协议进行调用。而在存算合并版 NebulaGraph 中，NebulaGraph 中 3 个服务对应的 3 个进程被合为 1 个进程。

关于 NebulaGraph 的更多信息，参见架构总览。

使用场景

数据规模小，可用性需求不大的场景。例如，受限于机器数量的测试环境或者仅用于验证功能的场景。

使用限制

- 仅支持单副本服务。
- 不支持高可用和可靠性。

环境准备

关于安装存算合并版 NebulaGraph 所需的环境，参见编译 NebulaGraph 源码要求。

安装步骤

目前仅支持使用源码安装存算合并版 NebulaGraph 。其安装步骤与多进程的 NebulaGraph 步骤类似，用户只需在使用 CMake 生成 makefile 文件步骤的命令中添加 `-DENABLE_STANDALONE_VERSION=on` 。示例如下：

```
cmake -DCMAKE_INSTALL_PREFIX=/usr/local/nebula -DENABLE_TESTING=OFF -DENABLE_STANDALONE_VERSION=on -DCMAKE_BUILD_TYPE=Release ..
```

有关具体的安装步骤，参见使用源码安装。

用户完成存算合并版 NebulaGraph 后，可以参见连接服务连接 NebulaGraph 。

配置文件

存算合并版 NebulaGraph 的配置文件的路径默认为 `/usr/local/nebula/etc` 。

用户可执行 `sudo cat nebula-standalone.conf.default` 查看配置文件内容。配置文件参数和描述和多进程的 NebulaGraph 大体一致，除以下参数外：

参数	预设值	说明
`meta_port`	`9559`	Meta 服务的端口号。
`storage_port`	`9779`	Storage 服务的端口号。
`meta_data_path`	`data/meta`	Meta 数据存储路径。

用户可以执行命令查看配置项列表与说明。具体操作，请参见配置管理。

最后更新: 2025年8月4日

5.4 使用 RPM/DEB 包部署 NebulaGraph 多机集群

本文介绍通过 RPM 或 DEB 文件部署集群的示例。

> Note
>
> 用户还可以通过官方工具部署 NebulaGraph 多机集群。详情参见使用生态工具安装集群。

5.4.1 部署方案

机器名称	IP 地址	graphd 进程数量	storaged 进程数量	metad 进程数量
A	192.168.10.111	1	1	1
B	192.168.10.112	1	1	1
C	192.168.10.113	1	1	1
D	192.168.10.114	1	1	-
E	192.168.10.115	1	1	-

5.4.2 前提条件

- 准备 5 台用于部署集群的机器。
- 在集群中通过 NTP 服务同步时间。

5.4.3 手动部署流程

安装 NebulaGraph

在集群的每一台服务器上都安装 NebulaGraph，安装后暂不需要启动服务。安装方式请参见：

- 使用 RPM 或 DEB 包安装 NebulaGraph
- 使用源码安装 NebulaGraph

修改配置文件

修改每个服务器上的 NebulaGraph 配置文件。

NebulaGraph 的所有配置文件均位于安装目录的 `etc` 目录内，包括 `nebula-graphd.conf` 、 `nebula-metad.conf` 和 `nebula-storaged.conf` ，用户可以只修改所需服务的配置文件。各个机器需要修改的配置文件如下。

机器名称	待修改配置文件
A	`nebula-graphd.conf` 、 `nebula-storaged.conf` 、 `nebula-metad.conf`
B	`nebula-graphd.conf` 、 `nebula-storaged.conf` 、 `nebula-metad.conf`
C	`nebula-graphd.conf` 、 `nebula-storaged.conf` 、 `nebula-metad.conf`
D	`nebula-graphd.conf` 、 `nebula-storaged.conf`
E	`nebula-graphd.conf` 、 `nebula-storaged.conf`

用户可以参考如下配置文件的内容，仅展示集群通信的部分设置，未展示的内容为默认设置，便于用户了解集群间各个服务器的关系。

> Note
>
> 主要修改的配置是 `meta_server_addrs`，所有配置文件都需要填写所有 Meta 服务的 IP 地址和端口，同时需要修改 `local_ip` 为机器本身的联网 IP 地址。配置参数的详细说明请参见：
>
> - Meta 服务配置
> - Graph 服务配置
> - Storage 服务配置

- 机器 A 配置
- `nebula-graphd.conf`

```
########## networking ##########
# Comma separated Meta Server Addresses
--meta_server_addrs=192.168.10.111:9559,192.168.10.112:9559,192.168.10.113:9559
# Local IP used to identify the nebula-graphd process.
# Change it to an address other than loopback if the service is distributed or
# will be accessed remotely.
--local_ip=192.168.10.111
# Network device to listen on
--listen_netdev=any
# Port to listen on
--port=9669
```

- `nebula-storaged.conf`

```
########## networking ##########
# Comma separated Meta server addresses
--meta_server_addrs=192.168.10.111:9559,192.168.10.112:9559,192.168.10.113:9559
# Local IP used to identify the nebula-storaged process.
# Change it to an address other than loopback if the service is distributed or
# will be accessed remotely.
--local_ip=192.168.10.111
# Storage daemon listening port
--port=9779
```

- `nebula-metad.conf`

```
########## networking ##########
# Comma separated Meta Server addresses
--meta_server_addrs=192.168.10.111:9559,192.168.10.112:9559,192.168.10.113:9559
# Local IP used to identify the nebula-metad process.
# Change it to an address other than loopback if the service is distributed or
# will be accessed remotely.
--local_ip=192.168.10.111
# Meta daemon listening port
--port=9559
```

- 机器 B 配置

- `nebula-graphd.conf`

```
########## networking ##########
# Comma separated Meta Server Addresses
--meta_server_addrs=192.168.10.111:9559,192.168.10.112:9559,192.168.10.113:9559
# Local IP used to identify the nebula-graphd process.
# Change it to an address other than loopback if the service is distributed or
# will be accessed remotely.
--local_ip=192.168.10.112
# Network device to listen on
--listen_netdev=any
# Port to listen on
--port=9669
```

- `nebula-storaged.conf`

```
########## networking ##########
# Comma separated Meta server addresses
--meta_server_addrs=192.168.10.111:9559,192.168.10.112:9559,192.168.10.113:9559
# Local IP used to identify the nebula-storaged process.
# Change it to an address other than loopback if the service is distributed or
# will be accessed remotely.
--local_ip=192.168.10.112
# Storage daemon listening port
--port=9779
```

- `nebula-metad.conf`

```
########## networking ##########
# Comma separated Meta Server addresses
--meta_server_addrs=192.168.10.111:9559,192.168.10.112:9559,192.168.10.113:9559
# Local IP used to identify the nebula-metad process.
# Change it to an address other than loopback if the service is distributed or
# will be accessed remotely.
--local_ip=192.168.10.112
# Meta daemon listening port
--port=9559
```

- 机器 C 配置

- `nebula-graphd.conf`

```
########## networking ##########
# Comma separated Meta Server Addresses
--meta_server_addrs=192.168.10.111:9559,192.168.10.112:9559,192.168.10.113:9559
# Local IP used to identify the nebula-graphd process.
# Change it to an address other than loopback if the service is distributed or
# will be accessed remotely.
--local_ip=192.168.10.113
# Network device to listen on
--listen_netdev=any
# Port to listen on
--port=9669
```

- `nebula-storaged.conf`

```
########## networking ##########
# Comma separated Meta server addresses
--meta_server_addrs=192.168.10.111:9559,192.168.10.112:9559,192.168.10.113:9559
# Local IP used to identify the nebula-storaged process.
# Change it to an address other than loopback if the service is distributed or
# will be accessed remotely.
--local_ip=192.168.10.113
# Storage daemon listening port
--port=9779
```

- `nebula-metad.conf`

```
########## networking ##########
# Comma separated Meta Server addresses
--meta_server_addrs=192.168.10.111:9559,192.168.10.112:9559,192.168.10.113:9559
# Local IP used to identify the nebula-metad process.
# Change it to an address other than loopback if the service is distributed or
# will be accessed remotely.
--local_ip=192.168.10.113
# Meta daemon listening port
--port=9559
```

- 机器 D 配置

- `nebula-graphd.conf`

```
########## networking ##########
# Comma separated Meta Server Addresses
--meta_server_addrs=192.168.10.111:9559,192.168.10.112:9559,192.168.10.113:9559
# Local IP used to identify the nebula-graphd process.
# Change it to an address other than loopback if the service is distributed or
# will be accessed remotely.
--local_ip=192.168.10.114
# Network device to listen on
--listen_netdev=any
# Port to listen on
--port=9669
```

- `nebula-storaged.conf`

```
########## networking ##########
# Comma separated Meta server addresses
--meta_server_addrs=192.168.10.111:9559,192.168.10.112:9559,192.168.10.113:9559
# Local IP used to identify the nebula-storaged process.
# Change it to an address other than loopback if the service is distributed or
# will be accessed remotely.
--local_ip=192.168.10.114
# Storage daemon listening port
--port=9779
```

- 机器 E 配置

- `nebula-graphd.conf`

```
########## networking ##########
# Comma separated Meta Server Addresses
--meta_server_addrs=192.168.10.111:9559,192.168.10.112:9559,192.168.10.113:9559
# Local IP used to identify the nebula-graphd process.
# Change it to an address other than loopback if the service is distributed or
# will be accessed remotely.
--local_ip=192.168.10.115
# Network device to listen on
--listen_netdev=any
# Port to listen on
--port=9669
```

- `nebula-storaged.conf`

```
########## networking ##########
# Comma separated Meta server addresses
--meta_server_addrs=192.168.10.111:9559,192.168.10.112:9559,192.168.10.113:9559
# Local IP used to identify the nebula-storaged process.
# Change it to an address other than loopback if the service is distributed or
# will be accessed remotely.
--local_ip=192.168.10.115
# Storage daemon listening port
--port=9779
```

启动集群

依次启动各个服务器上的对应进程。

机器名称	待启动的进程
A	graphd、storaged、metad
B	graphd、storaged、metad
C	graphd、storaged、metad
D	graphd、storaged
E	graphd、storaged

启动 NebulaGraph 进程的命令如下：

```
sudo /usr/local/nebula/scripts/nebula.service start <metad|graphd|storaged|all>
```

> **Note**
>
> - 确保每个服务器中的对应进程都已启动，否则服务将启动失败。
> - 当需都启动 graphd、storaged 和 metad 时，可以用 all 代替。
> - `/usr/local/nebula` 是 NebulaGraph 的默认安装路径，如果修改过安装路径，请使用实际路径。更多启停服务的内容，请参见管理 NebulaGraph 服务。

检查集群

安装原生 CLI 客户端 NebulaGraph Console，然后连接任何一个已启动 graphd 进程的机器，添加 Storage 主机，然后执行命令 `SHOW HOSTS` 检查集群状态。例如：

```
$ ./nebula-console --addr 192.168.10.111 --port 9669 -u root -p nebula

2021/05/25 01:41:19 [INFO] connection pool is initialized successfully
Welcome to NebulaGraph!

> ADD HOSTS 192.168.10.111:9779, 192.168.10.112:9779, 192.168.10.113:9779, 192.168.10.114:9779, 192.168.10.115:9779;
> SHOW HOSTS;
+------------------+------+----------+-------------+---------------------+-----------------------+----------------------+
```

```
| Host          | Port | Status   | Leader count | Leader distribution  | Partition distribution | Version     |
+-----------------+------+----------+--------------+----------------------+------------------------+----------------------+
| "192.168.10.111" | 9779 | "ONLINE" | 0            | "No valid partition" | "No valid partition"   | "master" |
| "192.168.10.112" | 9779 | "ONLINE" | 0            | "No valid partition" | "No valid partition"   | "master" |
| "192.168.10.113" | 9779 | "ONLINE" | 0            | "No valid partition" | "No valid partition"   | "master" |
| "192.168.10.114" | 9779 | "ONLINE" | 0            | "No valid partition" | "No valid partition"   | "master" |
| "192.168.10.115" | 9779 | "ONLINE" | 0            | "No valid partition" | "No valid partition"   | "master" |
+-----------------+------+----------+--------------+----------------------+------------------------+----------------------+
```

最后更新: 2025年8月4日

5.5 使用 Docker Compose 部署 NebulaGraph

使用 Docker Compose 可以基于准备好的配置文件快速部署 NebulaGraph 服务，仅建议在测试 NebulaGraph 功能时使用该方式。

5.5.1 前提条件

- 主机上安装如下应用程序。

应用程序	推荐版本	官方安装参考
Docker	最新版本	Install Docker Engine
Docker Compose	最新版本	Install Docker Compose
Git	最新版本	Download Git

- 如果使用非 root 用户部署 NebulaGraph，请授权该用户 Docker 相关的权限。详细信息，请参见 Manage Docker as a non-root user。
- 启动主机上的 Docker 服务。
- 如果已经通过 Docker Compose 在主机上部署了另一个版本的 NebulaGraph，为避免兼容性问题，需要删除目录 `nebula-docker-compose/data`。

5.5.2 部署 NebulaGraph

1. 通过 Git 克隆 `nebula-docker-compose` 仓库的 `3.6.0` 分支到主机。

> Danger
>
> `master` 分支包含最新的未测试代码。请不要在生产环境使用此版本。

```
$ git clone -b release-3.6 https://github.com/vesoft-inc/nebula-docker-compose.git
```

> Note
>
> Docker Compose 的 `x.y` 版本对齐内核的 `x.y` 版本，对于内核 `z` 版本，Docker Compose 不会发布对应的 `z` 版本，但是会拉取 `z` 版本的内核镜像。

2. 切换至目录 `nebula-docker-compose`。

```
$ cd nebula-docker-compose/
```

3. 执行如下命令启动 NebulaGraph 服务。

> Note
>
> - 如果长期未内核更新镜像，请先更新 NebulaGraph 镜像和 NebulaGraph Console 镜像。
> - 执行命令后的返回结果因安装目录不同而不同。

```
[nebula-docker-compose]$ docker-compose up -d
Creating nebula-docker-compose_metad0_1 ... done
Creating nebula-docker-compose_metad2_1 ... done
Creating nebula-docker-compose_metad1_1 ... done
Creating nebula-docker-compose_graphd2_1  ... done
Creating nebula-docker-compose_graphd_1  ... done
Creating nebula-docker-compose_graphd1_1  ... done
Creating nebula-docker-compose_storaged0_1 ... done
Creating nebula-docker-compose_storaged2_1 ... done
Creating nebula-docker-compose_storaged1_1 ... done
```

Compatibility

从 3.1 版本开始，Docker-compose 会自动启动 NebulaGraph Console 镜像的容器，并将 Storage 主机增加至集群中（即 `ADD HOSTS` 命令）。

Note

上述服务的更多信息，请参见架构总览。

5.5.3 连接 NebulaGraph

连接 NebulaGraph 有两种方式：

- 在容器外通过 Nebula Console 连接。因为容器的配置文件中将 Graph 服务的外部映射端口也固定为 9669，因此可以直接通过默认端口连接。详情参见连接 NebulaGraph 。
- 登录安装了 NebulaGraph Console 的容器，然后再连接 Graph 服务。本小节介绍这种方式。

1. 使用 `docker-compose ps` 命令查看 NebulaGraph Console 容器名称。

```
$ docker-compose ps
      Name              Command          State                    Ports
--------------------------------------------------------------------------------------------------------------------------------
nebula-docker-compose_console_1   sh -c for i in `seq 1 60`;...  Up
nebula-docker-compose_graphd1_1   /usr/local/nebula/bin/nebu ...  Up (healthy)  0.0.0.0:32847->15669/tcp,:::32847->15669/tcp, 19669/tcp,
                                  0.0.0.0:32846->19670/tcp,:::32846->19670/tcp,
                                  0.0.0.0:32849->5669/tcp,:::32849->5669/tcp, 9669/tcp
......
```

Note

`nebula-docker-compose_console_1` 和 `nebula-docker-compose_graphd1_1` 为容器的名称。

2. 进入 NebulaGraph Console 容器中。

```
$ docker exec -it nebula-docker-compose_console_1 /bin/sh
/ #
```

3. 通过 NebulaGraph Console 连接 NebulaGraph 。

```
/ # ./usr/local/bin/nebula-console -u <user_name> -p <password> --address=graphd --port=9669
```

Note

默认情况下，身份认证功能是关闭的，只能使用已存在的用户名（默认为 `root` ）和任意密码登录。如果想使用身份认证，请参见身份认证。

4. 查看集群状态。

```
nebula> SHOW HOSTS;
+-------------+------+----------+--------------+---------------------+----------------------+---------+
| Host        | Port | Status   | Leader count | Leader distribution | Partition distribution | Version |
+-------------+------+----------+--------------+---------------------+----------------------+---------+
| "storaged0" | 9779 | "ONLINE" | 0            | "No valid partition" | "No valid partition"   | "master" |
| "storaged1" | 9779 | "ONLINE" | 0            | "No valid partition" | "No valid partition"   | "master" |
| "storaged2" | 9779 | "ONLINE" | 0            | "No valid partition" | "No valid partition"   | "master" |
+-------------+------+----------+--------------+---------------------+----------------------+---------+
```

执行两次 `exit` 可以退出容器。

5.5.4 查看 NebulaGraph 服务的状态和端口

执行命令 `docker-compose ps` 可以列出 NebulaGraph 服务的状态和端口。

> Note
>
> NebulaGraph 默认使用 `9669` 端口为客户端提供服务，如果需要修改端口，请修改目录 `nebula-docker-compose` 内的文件 `docker-compose.yaml` ，然后重启 NebulaGraph 服务。

```
$ docker-compose ps
nebula-docker-compose_console_1   sh -c sleep 3 &&        Up
                nebula-co ...
nebula-docker-compose_graphd1_1   /usr/local/nebula/bin/nebu ...  Up   0.0.0.0:49174->19669/tcp,:::49174->19669/tcp, 0.0.0.0:49171->19670/tcp,:::49171->19670/tcp, 0.0.0.0:49177->9669/tcp,:::
49177->9669/tcp
nebula-docker-compose_graphd2_1   /usr/local/nebula/bin/nebu ...  Up   0.0.0.0:49175->19669/tcp,:::49175->19669/tcp, 0.0.0.0:49172->19670/tcp,:::49172->19670/tcp, 0.0.0.0:49178->9669/tcp,:::
49178->9669/tcp
nebula-docker-compose_graphd_1   /usr/local/nebula/bin/nebu ...  Up   0.0.0.0:49180->19669/tcp,:::49180->19669/tcp, 0.0.0.0:49179->19670/tcp,:::49179->19670/tcp, 0.0.0.0:9669->9669/tcp,:::
9669->9669/tcp
nebula-docker-compose_metad0_1   /usr/local/nebula/bin/nebu ...  Up   0.0.0.0:49157->19559/tcp,:::49157->19559/tcp, 0.0.0.0:49154->19560/tcp,:::49154->19560/tcp, 0.0.0.0:49160->9559/tcp,:::
49160->9559/tcp, 9560/tcp
nebula-docker-compose_metad1_1   /usr/local/nebula/bin/nebu ...  Up   0.0.0.0:49156->19559/tcp,:::49156->19559/tcp, 0.0.0.0:49153->19560/tcp,:::49153->19560/tcp, 0.0.0.0:49159->9559/tcp,:::
49159->9559/tcp, 9560/tcp
nebula-docker-compose_metad2_1   /usr/local/nebula/bin/nebu ...  Up   0.0.0.0:49158->19559/tcp,:::49158->19559/tcp, 0.0.0.0:49155->19560/tcp,:::49155->19560/tcp, 0.0.0.0:49161->9559/tcp,:::
49161->9559/tcp, 9560/tcp
nebula-docker-compose_storaged0_1  /usr/local/nebula/bin/nebu ...  Up   0.0.0.0:49166->19779/tcp,:::49166->19779/tcp, 0.0.0.0:49163->19780/tcp,:::49163->19780/tcp, 9777/tcp, 9778/tcp, 0.
0.0.0:49169->9779/tcp,:::49169->9779/tcp, 9780/tcp
nebula-docker-compose_storaged1_1  /usr/local/nebula/bin/nebu ...  Up   0.0.0.0:49165->19779/tcp,:::49165->19779/tcp, 0.0.0.0:49162->19780/tcp,:::49162->19780/tcp, 9777/tcp, 9778/tcp, 0.
0.0.0:49168->9779/tcp,:::49168->9779/tcp, 9780/tcp
nebula-docker-compose_storaged2_1  /usr/local/nebula/bin/nebu ...  Up   0.0.0.0:49167->19779/tcp,:::49167->19779/tcp, 0.0.0.0:49164->19780/tcp,:::49164->19780/tcp, 9777/tcp, 9778/tcp, 0.
0.0.0:49170->9779/tcp,:::49170->9779/tcp, 9780/tcp
```

如果服务有异常，用户可以先确认异常的容器名称（例如 `nebula-docker-compose_graphd2_1` ），然后登录容器排查问题：

```
$ docker exec -it nebula-docker-compose_graphd2_1 bash
```

5.5.5 查看 NebulaGraph 服务的数据和日志

NebulaGraph 的所有数据和日志都持久化存储在 `nebula-docker-compose/data` 和 `nebula-docker-compose/logs` 目录中。

目录的结构如下：

```
nebula-docker-compose/
  |-- docker-compose.yaml
  ├──── data
  │     ├──── meta0
  │     ├──── meta1
  │     ├──── meta2
  │     ├──── storage0
  │     ├──── storage1
  │     └──── storage2
  └──── logs
        ├──── graph
        ├──── graph1
        ├──── graph2
        ├──── meta0
        ├──── meta1
        ├──── meta2
        ├──── storage0
        ├──── storage1
        └──── storage2
```

5.5.6 修改配置

Docker Compose 配置文件位置为 `nebula-docker-compose/docker-compose.yaml` ，修改该文件内的配置并重启服务即可使新配置生效。

`docker-compose.yaml` 文件中的配置会覆盖服务容器内的配置文件（ `/usr/local/nebula/etc` ）的配置，因此也可以通过修改 `docker-compose.yaml` 文件设置服务的配置。

具体的配置说明请参见配置管理。

5.5.7 重启 NebulaGraph 服务

重启所有 NebulaGraph 服务：

```
$ docker-compose restart
Restarting nebula-docker-compose_console_1 ... done
Restarting nebula-docker-compose_graphd_1  ... done
Restarting nebula-docker-compose_graphd1_1  ... done
Restarting nebula-docker-compose_graphd2_1  ... done
Restarting nebula-docker-compose_storaged1_1 ... done
Restarting nebula-docker-compose-storaged0_1 ... done
Restarting nebula-docker-compose_storaged2_1 ... done
Restarting nebula-docker-compose_metad1_1  ... done
Restarting nebula-docker-compose_metad2_1  ... done
Restarting nebula-docker-compose_metad0_1  ... done
```

重启多个服务，例如重启 graphd 和 stoarged0 服务：

```
$ docker-compose restart graphd storaged0
Restarting nebula-docker-compose_graphd_1  ... done
Restarting nebula-docker-compose_storaged0_1 ... done
```

5.5.8 停止并删除 NebulaGraph 服务

用户可以执行如下命令停止并删除 Docker Compose启动的所有 NebulaGraph 服务：

> Danger
>
> 该命令会停止并删除所有 NebulaGraph 服务的容器，以及相关网络。如果用户在 `docker-compose.yaml` 中定义了卷（`volumes`），则会保留相关数据。
>
> 命令 `docker-compose down -v` 的参数 `-v` 将会删除所有本地的数据。如果使用的是 nightly 版本，并且有一些兼容性问题，请尝试这个命令。

```
$ docker-compose down
```

如果返回如下信息，表示已经成功停止服务。

```
Stopping nebula-docker-compose_console_1  ... done
Stopping nebula-docker-compose_graphd1_1  ... done
Stopping nebula-docker-compose_graphd_1   ... done
Stopping nebula-docker-compose_graphd2_1  ... done
Stopping nebula-docker-compose_storaged1_1 ... done
Stopping nebula-docker-compose_storaged0_1 ... done
Stopping nebula-docker-compose_storaged2_1 ... done
Stopping nebula-docker-compose_metad2_1   ... done
Stopping nebula-docker-compose_metad0_1   ... done
Stopping nebula-docker-compose_metad1_1   ... done
Removing nebula-docker-compose_console_1  ... done
Removing nebula-docker-compose_graphd1_1  ... done
Removing nebula-docker-compose_graphd_1   ... done
Removing nebula-docker-compose_graphd2_1  ... done
Removing nebula-docker-compose_storaged1_1 ... done
Removing nebula-docker-compose_storaged0_1 ... done
Removing nebula-docker-compose_storaged2_1 ... done
Removing nebula-docker-compose_metad2_1   ... done
Removing nebula-docker-compose_metad0_1   ... done
Removing nebula-docker-compose_metad1_1   ... done
Removing network nebula-docker-compose_nebula-net
```

5.5.9 常见问题

如何固定 **Docker** 映射到外部的端口？

在目录 `nebula-docker-compose` 内修改文件 `docker-compose.yaml`，将对应服务的 `ports` 设置为固定映射，例如：

```
graphd:
  image: vesoft/nebula-graphd:release-3.6
  ...
  ports:
   - 9669:9669
   - 19669
   - 19670
```

`9669:9669` 表示内部的 9669 映射到外部的端口也是 9669，下方的 `19669` 表示内部的 19669 映射到外部的端口是随机的。

如何升级/更新 NebulaGraph 服务的 Docker 镜像?

1. 在文件 `nebula-docker-compose/docker-compose.yaml` 中，找到所有服务的 `image` 并修改其值为相应的镜像版本。
2. 在目录 `nebula-docker-compose` 内执行命令 `docker-compose pull`，更新 Graph 服务、Storage 服务、Meta 服务和 NebulaGraph Console 的镜像。
3. 执行命令 `docker-compose up -d` 启动 NebulaGraph 服务。
4. 通过 NebulaGraph Console 连接 NebulaGraph 后，分别执行命令 `SHOW HOSTS GRAPH`、 `SHOW HOSTS STORAGE`、 `SHOW HOSTS META` 查看各服务版本。

执行命令 `docker-compose pull` 报错 `ERROR: toomanyrequests`

可能遇到如下错误：

`ERROR: toomanyrequests: You have reached your pull rate limit. You may increase the limit by authenticating and upgrading: https://www.docker.com/increase-rate-limit`

以上错误表示已达到 Docker Hub 的速率限制。解决方案请参见 Understanding Docker Hub Rate Limiting。

如何更新 NebulaGraph Console?

`docker-compose pull` 会同时更新 NebulaGraph 服务和 NebulaGraph Console。

Storaged 容器一直处于 `offline` 状态

在小概率的情况下，Docker Compose 中的 Storaged 激活脚本并没有在适当的时机得到执行。如果发现 Storaged 容器的健康状态一直是 `offline`，可以通过 NebulaGraph Console 或者 NebulaGraph Studio 连接 NebulaGraph，并手动执行 `ADD HOSTS` 命令对其进行激活。激活命令示例如下：

```
nebula> ADD HOSTS "storaged0":9779,"storaged1":9779,"storaged2":9779
```

最后更新: 2025年8月4日

5.6 使用 NebulaGraph Lite 部署 NebulaGraph

使用 NebulaGraph Lite 可以快速安装部署 NebulaGraph，仅需五分钟即可开始体验 NebulaGraph 图数据库，适用于临时开发、学习 NebulaGraph。

5.6.1 优势

- 通过 Python 包管理工具快速安装 NebulaGraph Lite。
- 支持非 Root 权限部署 NebulaGraph。
- 支持在容器或基于 Linux 系统的 Jupyter Notebook 平台上部署 NebulaGraph。

5.6.2 操作步骤

1. 执行如下命令安装 NebulaGraph Lite。

```
pip3 install nebulagraph-lite
```

2. 启动 NebulaGraph Lite。NebulaGraph Lite 会自动部署并启动单节点 NebulaGraph 服务，以及导入测试数据集。

- 从 Jupyter Notebook 启动

```
from nebulagraph_lite import nebulagraph_let as ng_let
n = ng_let()
n.start()
```

- 从命令行启动

```
nebulagraph start
```

返回如下结果表示启动并导入测试数据集成功。

```
Info: loading basketballplayer dataset...

 _   _      _           _        ____                 _
| \ | | ___| |__  _   _| | __ _ / ___|_ __ __ _ _ __ | |__
|  \| |/ _ \ '_ \| | | | |/ _` | |  _| '__/ _` | '_ \| '_ \
| |\  |  __/ |_) | |_| | | (_| | |_| | | | (_| | |_) | | | |
|_| \_|\___|_.__/ \__,_|_|\__,_|\____|_|  \__,_| .__/|_| |_|
                                               |_|
                                               lite version

[ OK ] nebulagraph_lite started successfully!
```

5.6.3 下一步

- 使用NebulaGraph Jupyter Extension连接 NebulaGraph。
- 使用NebulaGraph Console连接 NebulaGraph。

最后更新: 2025年8月4日

5.7 使用生态工具安装 NebulaGraph

用户可以使用以下生态工具安装 NebulaGraph：

- NebulaGraph Operator

5.7.1 安装详情

使用 NebulaGraph Operator 安装 NebulaGraph 的详情，参见创建 NebulaGraph 集群。

最后更新: 2025年8月4日

5.8 管理 NebulaGraph 服务

NebulaGraph 支持通过脚本管理服务。

5.8.1 使用脚本管理服务

使用脚本 `nebula.service` 管理服务，包括启动、停止、重启、中止和查看。

> Note
>
> `nebula.service` 的默认路径是 `/usr/local/nebula/scripts`，如果修改过安装路径，请使用实际路径。

语法

```
$ sudo /usr/local/nebula/scripts/nebula.service
[-v] [-c <config_file_path>]
<start | stop | restart | kill | status>
<metad | graphd | storaged | all>
```

参数	说明
`-v`	显示详细调试信息。
`-c`	指定配置文件路径，默认路径为 `/usr/local/nebula/etc/`。
`start`	启动服务。
`stop`	停止服务。
`restart`	重启服务。
`kill`	中止服务。
`status`	查看服务状态。
`metad`	管理 Meta 服务。
`graphd`	管理 Graph 服务。
`storaged`	管理 Storage 服务。
`all`	管理所有服务。

5.8.2 启动 NebulaGraph 服务

执行如下命令启动服务：

```
$ sudo /usr/local/nebula/scripts/nebula.service start all
[INFO] Starting nebula-metad...
[INFO] Done
[INFO] Starting nebula-graphd...
[INFO] Done
[INFO] Starting nebula-storaged...
[INFO] Done
```

5.8.3 停止 NebulaGraph 服务

> Danger
>
> 请勿使用 `kill -9` 命令强制终止进程，否则可能较小概率出现数据丢失。

执行如下命令停止 NebulaGraph 服务：

```
$ sudo /usr/local/nebula/scripts/nebula.service stop all
[INFO] Stopping nebula-metad...
[INFO] Done
[INFO] Stopping nebula-graphd...
[INFO] Done
[INFO] Stopping nebula-storaged...
[INFO] Done
```

5.8.4 查看 NebulaGraph 服务

执行如下命令查看 NebulaGraph 服务状态：

```
$ sudo /usr/local/nebula/scripts/nebula.service status all
```

- 如果返回如下结果，表示 NebulaGraph 服务正常运行。

```
[INFO] nebula-metad(33fd35e): Running as 29020, Listening on 9559
[INFO] nebula-graphd(33fd35e): Running as 29095, Listening on 9669
[WARN] nebula-storaged after v3.0.0 will not start service until it is added to cluster.
[WARN] See Manage Storage hosts:ADD HOSTS in https://docs.nebula-graph.io/
[INFO] nebula-storaged(33fd35e): Running as 29147, Listening on 9779
```

> Note
>
> 正常启动 NebulaGraph 后， `nebula-storaged` 进程的端口显示红色。这是因为 `nebula-storaged` 在启动流程中会等待 `nebula-metad` 添加当前 Storage 服务，当前 Storage 服务收到 Ready 信号后才会正式启动服务。从 3.0.0 版本开始，在配置文件中添加的 Storage 节点无法直接读写，配置文件的作用仅仅是将 Storage 节点注册至 Meta 服务中。必须使用 `ADD HOSTS` 命令后，才能正常读写 Storage 节点。更多信息，参见管理 Storage 主机。

- 如果返回类似如下结果，表示 NebulaGraph 服务异常，可以根据异常服务信息进一步排查，或者在 NebulaGraph 社区寻求帮助。

```
[INFO] nebula-metad: Running as 25600, Listening on 9559
[INFO] nebula-graphd: Exited
[INFO] nebula-storaged: Running as 25646, Listening on 9779
```

NebulaGraph 服务由 Meta 服务、Graph 服务和 Storage 服务共同提供，这三种服务的配置文件都保存在安装目录的 `etc` 目录内，默认路径为 `/usr/local/nebula/etc/` ，用户可以检查相应的配置文件排查问题。

5.8.5 下一步

- 连接 NebulaGraph

最后更新: 2025年8月4日

5.9 连接 NebulaGraph 服务

本文介绍如何使用原生命令行客户端 NebulaGraph Console 连接 NebulaGraph。

> Caution
>
> 首次连接到 NebulaGraph 后，必须先注册 Storage 服务，才能正常查询数据。

NebulaGraph 支持多种类型的客户端，包括命令行客户端、可视化界面客户端和流行编程语言客户端。详情参见客户端列表。

5.9.1 前提条件

- NebulaGraph 服务已启动。
- 运行 NebulaGraph Console 的机器和运行 NebulaGraph 的服务器网络互通。
- NebulaGraph Console 的版本兼容 NebulaGraph 的版本。

> Note
>
> 版本相同的 NebulaGraph Console 和 NebulaGraph 兼容程度最高，版本不同的 NebulaGraph Console 连接 NebulaGraph 时，可能会有兼容问题，或者无法连接并报错 `incompatible version between client and server`。

5.9.2 操作步骤

1. 在 NebulaGraph Console 下载页面，确认需要的版本，单击 Assets。

> Note
>
> 建议选择最新版本。

2. 在 Assets 区域找到机器运行所需的二进制文件，下载文件到机器上。
3. （可选）为方便使用，重命名文件为 `nebula-console`。

> Note
>
> 在 Windows 系统中，请重命名为 `nebula-console.exe`。

4. 在运行 NebulaGraph Console 的机器上执行如下命令，为用户授予 nebula-console 文件的执行权限。

> Note
>
> Windows 系统请跳过此步骤。

```
$ chmod 111 nebula-console
```

5. 在命令行界面中，切换工作目录至 nebula-console 文件所在目录。

6. 执行如下命令连接 NebulaGraph 。

- Linux 或 macOS

```
$ ./nebula-console -addr <ip> -port <port> -u <username> -p <password>
  [-t 120] [-e "nGQL_statement" | -f filename.nGQL]
```

- Windows

```
> nebula-console.exe -addr <ip> -port <port> -u <username> -p <password>
  [-t 120] [-e "nGQL_statement" | -f filename.nGQL]
```

参数说明如下。

参数	说明
`-h/-help`	显示帮助菜单。
`-addr/-address`	设置要连接的 Graph 服务的 IP 或主机名。默认地址为 127.0.0.1。
`-P/-port`	设置要连接的 Graph 服务的端口。默认端口为 9669。
`-u/-user`	设置 NebulaGraph 账号的用户名。未启用身份认证时，可以使用任意已存在的用户名（默认为 `root` ）。
`-p/-password`	设置用户名对应的密码。未启用身份认证时，密码可以填写任意字符。
`-t/-timeout`	设置整数类型的连接超时时间。单位为毫秒，默认值为 120。
`-e/-eval`	设置字符串类型的 nGQL 语句。连接成功后会执行一次该语句并返回结果，然后自动断开连接。
`-f/-file`	设置存储 nGQL 语句的文件的路径。连接成功后会执行该文件内的 nGQL 语句并返回结果，执行完毕后自动断开连接。
`-enable_ssl`	连接 NebulaGraph 时使用 SSL 加密。
`-ssl_root_ca_path`	指定 CA 证书的存储路径。
`-ssl_cert_path`	指定 CRT 证书的存储路径。
`-ssl_private_key_path`	指定私钥文件的存储路径。

更多参数参见项目仓库。

最后更新: 2025年8月4日

5.10 管理 Storage 主机

从 3.0.0 版本开始，在配置文件中添加的 Storage 主机无法直接读写，配置文件的作用仅仅是将 Storage 主机注册至 Meta 服务中。必须使用 `ADD HOSTS` 命令后，才能正常读写 Storage 主机。

5.10.1 前提条件

- 已经连接服务

5.10.2 增加 Storage 主机

向集群中增加 Storage 主机。

```
nebula> ADD HOSTS <ip>:<port> [,<ip>:<port> ...];
nebula> ADD HOSTS "<hostname>":<port> [,"<hostname>":<port> ...];
```

> Note
>
> - 增加 Storage 主机在下一个心跳周期之后才能生效，为确保数据同步，请等待 2 个心跳周期（20 秒），然后执行 `SHOW HOSTS` 查看是否在线。
> - IP地址和端口请和配置文件中的设置保持一致，例如单机部署的默认为 `127.0.0.1:9779`。
> - 使用域名时，需要用引号包裹，例如 `ADD HOSTS "foo-bar":9779`。
> - 确保新增的 Storage 主机没有被其他集群使用过，否则会导致添加 Storage 节点失败。

5.10.3 删除 Storage 主机

从集群中删除 Storage 主机。

> Note
>
> 无法直接删除正在使用的 Storage 主机，需要先删除关联的图空间，才能删除 Storage 主机。

```
nebula> DROP HOSTS <ip>:<port> [,<ip>:<port> ...];
nebula> DROP HOSTS "<hostname>":<port> [,"<hostname>":<port> ...];
```

5.10.4 查看 Storage 主机

查看集群中的 Storage 主机。

```
nebula> SHOW HOSTS STORAGE;
+-------------+------+----------+-----------+--------------+---------+
| Host        | Port | Status   | Role      | Git Info Sha | Version |
+-------------+------+----------+-----------+--------------+---------+
| "storaged0" | 9779 | "ONLINE" | "STORAGE" | "3ba41bd"    | "master" |
| "storaged1" | 9779 | "ONLINE" | "STORAGE" | "3ba41bd"    | "master" |
| "storaged2" | 9779 | "ONLINE" | "STORAGE" | "3ba41bd"    | "master" |
+-------------+------+----------+-----------+--------------+---------+
```

最后更新: 2025年8月4日

5.11 升级 NebulaGraph 至 master 版本

本文以 NebulaGraph 2.6.1 版本升级到 master 版本为例，介绍 NebulaGraph 2.x、3.x 版本升级到 master 版本的方法。

5.11.1 适用版本

本文适用于将 NebulaGraph 从 2.5.0 及之后的 2.x、3.x 版本升级到 master 版本。不适用于 2.5.0 之前的历史版本（含 1.x 版本）。如需升级历史版本，将其根据最新的 2.x 版本文档升级到 2.5 版本，然后根据本文的说明升级到 master 版本。

> Caution
>
> 如需从 2.0.0 之前的版本（含 1.x 版本）升级到 master，还需找到 master 版本文件中 `share/resources` 目录下的 `date_time_zonespec.csv` 文件，将其复制到 NebulaGraph 安装路径下的相同目录内。也可从 GitHub 下载该文件。

- 不支持轮转热升级，需完全停止整个集群服务。
- 未提供升级脚本，需手动在每台服务器上依次执行。
- 不支持基于 Docker 容器（包括 Docker Swarm、Docker Compose、K8s）的升级。
- 必须在原服务器上原地升级，不能修改原机器的 IP 地址、配置文件，不可更改集群拓扑。
- 已知会造成数据丢失的 4 种场景，和 alter schema 以及 default value 相关，参见 github known issues。
- 数据目录不要使用软连接切换，避免失效。
- 部分升级操作需要有 sudo 权限。

5.11.2 升级影响

- 客户端兼容

  升级后旧版本客户端将无法连接 NebulaGraph，需将所有客户端都升级到兼容 NebulaGraph master 的版本。

- 配置变化

  少数配置参数发生改变，详情参考版本发布说明和参数文档。

- 语法兼容

  nGQL 语法有部分不兼容：

- 禁用 `YIELD` 子句返回自定义变量。
- `FETCH`、 `GO`、 `LOOKUP`、 `FIND PATH`、 `GET SUBGRAPH` 语句中必须添加 `YIELD` 子句。
- MATCH 语句中获取点属性时，必须指定 Tag，例如从 `return v.name` 变为 `return v.player.name`。

- 全文索引

  在升级部署了全文索引的 NebulaGraph 前，需要手动删除 Elasticsearch (ES) 中的全文索引。在升级后需要重新使用 `SIGN IN` 语句登录 ES 并重新创建全文索引。用户可通过 cURL 命令手动删除 ES 中全文索引。命令为 `curl -XDELETE -u <es_username>:<es_password> '<es_access_ip>:<port>/<fullindex_name>'`，例如 `curl -XDELETE -u elastic:elastic 'http://192.168.8.223:9200/nebula_index_2534'`。如果 ES 没有设置用户名及密码，则无需指定 `-u` 选项。

> Caution
>
> 可能存在其它暂未发现的影响，建议升级前详细查看版本发布说明和产品手册，并密切关注论坛与 GitHub 的最新动态。

5.11.3 升级准备

- 根据操作系统和架构下载 NebulaGraph master 版本的包文件并解压，升级过程中需要其中的二进制文件。下载地址参见 Download 页面。

> Note
>
> 编译源码或者下载 RPM/DEB、TAR 包都可以获取新版二进制文件。

- 根据 Storage 和 Meta 服务配置中 `data_path` 参数的值找到数据文件的位置，并备份数据。默认路径为 `nebula/data/storage` 和 `nebula/data/meta` 。

> Danger
>
> 升级时不会自动备份原有数据。务必手动备份数据，防止丢失。

- 备份配置文件。

- 统计所有图空间升级前的数据量，供升级后比较。统计方法如下：

a. 运行 `SUBMIT JOB STATS` 。

b. 运行 `SHOW JOBS` 并记录返回结果。

5.11.4 升级步骤

1. 停止所有 NebulaGraph 服务。

```
<nebula_install_path>/scripts/nebula.service stop all
```

`nebula_install_path` 代表 NebulaGraph 的安装目录。

`storaged` 进程 flush 数据要等待约 1 分钟。运行命令后可继续运行 `nebula.service status all` 命令以确认所有服务都已停止。启动和停止服务的详细说明参见管理服务。

> Note
>
> 如果超过 20 分钟不能停止服务，放弃本次升级，在论坛或 GitHub 提问。

> Caution
>
> 从 3.0.0 开始，支持插入无 Tag 的点。如果用户需要保留无 Tag 的点，在集群内所有 Graph 服务的配置文件（ `nebula-graphd.conf` ）中新增 `--graph_use_vertex_key=true` ；在所有 Storage 服务的配置文件（ `nebula-storaged.conf` ）中新增 `--use_vertex_key=true` 。

2. 在升级准备中下载的包的目的路径下，用此处 `bin` 目录中的新版二进制文件替换 NebulaGraph 安装路径下 `bin` 目录中的旧版二进制文件。

> Note
>
> 每台部署了 NebulaGraph 服务的机器上都要更新相应服务的二进制文件。

3. 编辑所有 Graph 服务的配置文件，修改以下参数以适应新版本的取值范围。如参数值已在规定范围内，忽略该步骤。

- 为 `session_idle_timeout_secs` 参数设置一个在 [1,604800] 区间的值，推荐值为 28800。
- 为 `client_idle_timeout_secs` 参数设置一个在 [1,604800] 区间的值，推荐值为 28800。

这些参数在 2.x 版本中的默认值不在新版本的取值范围内，如不修改会升级失败。详细参数说明参见Graph 服务配置。

4. 启动所有 Meta 服务。

```
<nebula_install_path>/scripts/nebula-metad.service start
```

启动后，Meta 服务选举 leader。该过程耗时数秒。

启动后可以任意启动一个 Graph 服务节点，使用 NebulaGraph 连接该节点并运行 `SHOW HOSTS meta` 和 `SHOW META LEADER`，如果能够正常返回 Meta 节点的状态，则 Meta 服务启动成功。

> Note
>
> 如果启动异常，放弃本次升级，并在论坛或 GitHub 提问。

5. 启动所有 Graph 和 Storage 服务。

> Note
>
> 如果启动异常，放弃本次升级，并在论坛或 GitHub 提问。

6. 连接新版 NebulaGraph，验证服务是否可用、数据是否正常。连接方法参见连接服务。

目前尚无有效方式判断升级是否完全成功，可用于测试的参考命令如下：

```
nebula> SHOW HOSTS;
nebula> SHOW HOSTS storage;
nebula> SHOW SPACES;
nebula> USE <space_name>
nebula> SHOW PARTS;
nebula> SUBMIT JOB STATS;
nebula> SHOW STATS;
nebula> MATCH (v) RETURN v LIMIT 5;
```

也可根据 master 版本的新功能测试，新功能列表参见发布说明。

5.11.5 升级失败回滚

如果升级失败，停止新版本的所有服务，从备份中恢复配置文件和二进制文件，启动历史版本的服务。

所有周边客户端也切换为旧版。

5.11.6 FAQ

升级过程中是否可以通过客户端写入数据?

不可以。升级过程中需要停止所有服务。

升级过程中出现 **`Space 0 not found`** 。

当升级过程中出现 `Space 0 not found` 告警信息时，用户可以忽略这个信息。升级过程会从磁盘读取所有 Space ID，而 `0` （路径为 `<nebula_storagepath>/data/storage/nebula/0` ）并不会存在磁盘上。Space `0` 用来存储 Storage 服务的元信息，并不包含用户数据，因此不会影响升级。

如果某台机器只有 **Graph** 服务，没有 **Storage** 服务，如何升级?

只需要升级 Graph 服务对应的二进制文件和配置文件。

操作报错 **`Permission denied`** 。

部分命令需要有 sudo 权限。

是否有工具或者办法验证新旧版本数据是否一致?

没有。如果只是检查数据量，可以在升级完成后再次运行 `SUBMIT JOB STATS` 和 `SHOW STATS` 统计数据量，并与升级之前做对比。

Storage `OFFLINE` 并且 `Leader count` 是 `0` 怎么处理?

运行以下命令手动添加 Storage 主机：

```
ADD HOSTS <ip>:<port>[, <ip>:<port> ...];
```

例如：

```
ADD HOSTS 192.168.10.100:9779, 192.168.10.101:9779, 192.168.10.102:9779;
```

如果有多个 Meta 服务节点，手动 `ADD HOSTS` 之后，部分 Storage 节点需等待数个心跳（`heartbeat_interval_secs`）的时间才能正常连接到集群。

如果添加 Storage 主机后问题仍然存在，在论坛或 GitHub 提问。

为什么升级后用 `SHOW JOBS` 查询到的 Job 的 ID 与升级前一样，但 Job 名称等信息不同了?

NebulaGraph 2.5.0 版本调整了 Job 的定义，详情参见 Pull request。如果是从 2.5.0 之前的版本升级，会出现该问题。

有哪些语法不兼容 ?

A: 参见Release Note Incompatibility 部分。

最后更新: 2025年8月4日

5.12 卸载 NebulaGraph

本文介绍如何卸载 NebulaGraph 。

Caution

如果需要重新部署 NebulaGraph，请务必完全卸载后再重新部署，否则可能会出现问题，包括 Meta 不一致等。

5.12.1 前提条件

停止 NebulaGraph 服务。详情参见管理 NebulaGraph 服务。

5.12.2 步骤 1：删除数据和元数据文件

如果在配置文件内修改了数据文件的路径，可能会导致安装路径和数据文件保存路径不一致，因此需要查看配置文件，确认数据文件保存路径，然后手动删除数据文件目录。

Note

如果是集群架构，需要删除所有 Storage 和 Meta 服务节点的数据文件。

1. 检查 Storage 服务的 disk 配置。例如：

```
########## Disk ##########
# Root data path. Split by comma. e.g. --data_path=/disk1/path1/,/disk2/path2/
# One path per Rocksdb instance.
--data_path=/nebula/data/storage
```

2. 检查 metad 服务的配置文件，找到对应元数据目录。
3. 删除以上数据和元数据目录。

5.12.3 步骤 2：卸载安装目录

Note

删除整个安装目录，包括 `cluster.id` 文件。

安装路径为参数 `--prefix` 指定的路径。默认路径为 `/usr/local/nebula` 。

卸载编译安装的 **NebulaGraph**

找到 NebulaGraph 的安装目录，删除整个安装目录。

卸载 **RPM** 包安装的 **NebulaGraph**

1. 使用如下命令查看 NebulaGraph 版本。

```
$ rpm -qa | grep "nebula"
```

返回类似如下结果。

```
nebula-graph-master-1.x86_64
```

2. 使用如下命令卸载 NebulaGraph 。

```
sudo rpm -e <nebula_version>
```

例如：

```
sudo rpm -e nebula-graph-master-1.x86_64
```

3. 删除安装目录。

卸载 DEB 包安装的 NebulaGraph

1. 使用如下命令查看 NebulaGraph 版本。

```
$ dpkg -l | grep "nebula"
```

返回类似如下结果。

```
ii nebula-graph master amd64   NebulaGraph Package built using CMake
```

2. 使用如下命令卸载 NebulaGraph 。

```
sudo dpkg -r <nebula_version>
```

例如：

```
sudo dpkg -r nebula-graph
```

3. 删除安装目录。

卸载 Docker Compose 部署的 NebulaGraph

1. 在目录 `nebula-docker-compose` 内执行如下命令停止 NebulaGraph 服务。

```
docker-compose down -v
```

2. 删除目录 `nebula-docker-compose` 。

最后更新: 2025年8月4日

6. 配置与日志

6.1 配置

6.1.1 配置管理

NebulaGraph 基于 gflags 库打造了系统配置，多数配置项都是其中的 flags。 NebulaGraph 服务启动时，默认会从配置文件中获取配置信息。对于文件中没有的配置项，系统使用默认值。

Note

- 由于配置项多且可能随着 NebulaGraph 的开发发生变化，文档不会介绍所有配置项。按下文说明可在命令行获取配置项的详细说明。
- 不建议修改文档未介绍的配置项，除非已经熟悉源代码并完全了解配置项的作用。

历史版本兼容性

1.x 版本的文档提供了使用 CONFIGS 命令修改缓存中配置的方法，但在生产环境中使用该方法容易导致集群配置与本地配置文件不一致。因此，自 2.x 版本开始文档中将不再介绍 CONFIGS 命令的使用方法。

查看配置项列表与说明

使用以下命令获取二进制文件对应服务的所有配置项信息：

```
<binary> --help
```

例如：

```
# 获取 Meta 配置项的帮助信息
$ /usr/local/nebula/bin/nebula-metad --help

# 获取 Graph 配置项的帮助信息
$ /usr/local/nebula/bin/nebula-graphd --help

# 获取 Storage 配置项的帮助信息
$ /usr/local/nebula/bin/nebula-storaged --help
```

以上示例使用了二进制文件的默认存储路径 `/usr/local/nebula/bin/`。如果修改了 NebulaGraph 安装路径，使用实际路径查询配置项信息。

查看运行配置

使用 `curl` 命令获取运行中的配置项的值，即 NebulaGraph 的运行配置。

例如：

```
# 获取 Meta 服务的运行配置
curl 127.0.0.1:19559/flags

# 获取 Graph 服务的运行配置
curl 127.0.0.1:19669/flags

# 获取 Storage 服务的运行配置
curl 127.0.0.1:19779/flags
```

使用 `-s` 或者 `--silent` 参数可以隐藏进度条和错误信息。例如：

```
curl -s 127.0.0.1:19559/flags
```

> Note
>
> 实际环境中需使用真实的 IP（或主机名）取代以上示例中的 `127.0.0.1`。

配置文件简介

源码、RPM/DEB、TAR 包集群的配置文件

NebulaGraph 为每个服务都提供了两份初始配置文件 `<service_name>.conf.default` 和 `<service_name>.conf.production`，方便用户在不同场景中使用。使用源码和 RPM/DEB 安装集群的配置文件的默认路径为 `/usr/local/nebula/etc/`；使用 TAR 包安装集群的配置文件路径为 `<install_path>/<tar_package_directory>/etc` TAR 包的安装路径。

初始配置文件中的配置值仅供参考，使用时可根据实际需求调整。如需使用初始配置文件，从上述两个文件选择其一，删除后缀 `.default` 或 `.production` 使其生效。

> Note
>
> 为确保服务的可用性，同类服务的配置建议保持一致，本机 `local_ip` 除外。例如，一个 NebulaGraph 集群中部署了 3 个 Storage 服务器，3 者除 `local_ip` 外的其它配置建议都相同。

下表列出了各服务对应的初始配置文件。

NebulaGraph 服务	初始配置文件	配置说明
Meta	`nebula-metad.conf.default` 和 `nebula-metad.conf.production`	Meta 服务配置
Graph	`nebula-graphd.conf.default` 和 `nebula-graphd.conf.production`	Graph 服务配置
Storage	`nebula-storaged.conf.default` 和 `nebula-storaged.conf.production`	Storage 服务配置

所有服务的初始配置文件中都包含 `local_config` 参数，预设值为 `true`，表示 NebulaGraph 服务会从其配置文件获取配置并启动。

> Caution
>
> 不建议修改 `local_config` 的值为 `false`。修改后 NebulaGraph 服务启动后会先尝试从 Meta 服务获取缓存的配置信息，可能导致集群配置不一致，造成未知风险。

DOCKER COMPOSE 集群的配置文件

对于使用 Docker Compose 创建的集群，集群的配置文件的默认路径为 `<install_path>/nebula-docker-compose/docker-compose.yaml`。配置文件中的 `command` 下面的参数为各服务的启动参数。

NEBULAGRAPH OPERATOR 集群的配置文件

对于通过 NebulaGraph Operator 使用 Kubectl 方式创建的集群，集群的配置文件的路径为用户创建集群 YAML 文件的路径。用户可通过配置文件中的 `spec.{graphd|storaged|metad}.config` 参数修改集群各个服务的相关配置。

> Note
>
> 通过 Helm 部署的集群，暂不支持修改集群服务的相关配置。

关于 NebulaGraph 的配置相关的更多信息，参见 NebulaGraph Config。

修改配置

用户可以在配置文件中修改 NebulaGraph 的配置，或使用命令动态修改配置。

Caution

同时使用两种方式修改配置会导致配置信息失去统一的管理方式，可能造成配置混乱。建议仅使用配置文件管理配置，或在通过命令动态更新配置后对配置文件做相同的修改，以保证一致性。

在配置文件中修改配置

默认情况下，所有 NebulaGraph 服务从配置文件获取配置。用户可以按照以下步骤修改配置并使其生效。

- 针对使用源码、RPM/DEB、TAR 包安装的集群：

a. 使用文本编辑器修改目标服务的配置文件并保存。

b. 选择合适的时间重启所有 NebulaGraph 服务使修改生效。

- 针对使用 Docker Compose 安装的集群：

a. 在文件 `<install_path>/nebula-docker-compose/docker-compose.yaml` 中，修改服务配置。

b. 在目录 `nebula-docker-compose` 内执行命令 `docker-compose up -d` 重启涉及配置变化的服务。

- 针对使用 Kubectl 方式创建的集群：

  具体操作，参见自定义集群的配置参数。

使用命令动态修改配置

用户可以通过 `curl` 命令动态修改 NebulaGraph 服务的配置。例如，修改 Storage 服务的 `wal_ttl` 参数为 `600`，命令如下：

```
curl -X PUT -H "Content-Type: application/json" -d'{"wal_ttl":"600"}' -s "http://192.168.15.6:19779/flags"
```

其中，`{"wal_ttl":"600"}` 为待修改的配置参数及其值；`192.168.15.6:19779` 为 Storage 服务的 IP 地址和 HTTP 端口号。

Caution

- 动态修改配置功能仅适用于原型验证和测试环境，不建议在生产环境中使用。因为当 `local_config` 值设置为 `true` 时，动态修改的配置不会持久化，重启服务后配置会恢复为初始配置。
- 仅支持动态修改部分配置参数，具体支持的参数列表，参见各服务配置中是否支持运行时动态修改的描述。

最后更新: 2025年8月4日

6.1.2 Meta 服务配置

Meta 服务提供了两份初始配置文件 `nebula-metad.conf.default` 和 `nebula-metad.conf.production`，方便在不同场景中使用。文件的默认路径为 `/usr/local/nebula/etc/`。

> Caution
>
> - 不建议修改 `local_config` 的值为 `false`。修改后 NebulaGraph 服务启动后会先尝试从 Meta 服务获取缓存的配置信息，可能导致集群配置不一致，造成未知风险。
> - 不建议修改文档未介绍的配置项，除非已经熟悉源代码并清楚了解配置项作用。

配置文件使用方式

如需使用初始配置文件，从上述两个文件选择其一，删除后缀 `.default` 或 `.production`，Meta 服务才能将其识别为配置文件并从中获取配置信息。

配置文件参数值说明

配置文件内没有设置某个参数表示参数使用的是默认值。文件内只预设了部分参数的值，而且两份初始配置文件内的参数值也略有不同，本文的预设值以 `nebula-metad.conf.default` 为准。

> Caution
>
> 配置文件中的部分参数值可以在运行时动态修改。本文将这些参数标记为支持运行时动态修改，并当 `local_config` 值设置为 `true` 时重启服务后配置会恢复为初始配置。详情参见修改配置。

如需查看所有的参数及其当前值，参见配置管理。

basics 配置

名称	预设值	说明	是否支持运行时动态修改
`daemonize`	`true`	是否启动守护进程。	不支持
`pid_file`	`pids/nebula-metad.pid`	记录进程 ID 的文件。	不支持
`timezone_name`	-	指定 NebulaGraph 的时区。初始配置文件中未设置该参数，如需使用请手动添加。系统默认值为 `UTC+00:00:00`。格式请参见 Specifying the Time Zone with TZ。例如，东八区的设置方式为 `--timezone_name=UTC+08:00`。	不支持

> Note
>
> - 在插入时间类型的属性值时，NebulaGraph 会根据 `timezone_name` 设置的时区将该时间值（TIMESTAMP 类型例外）转换成相应的 UTC 时间，因此在查询中返回的时间类型属性值为 UTC 时间。
> - `timezone_name` 参数只用于转换 NebulaGraph 中存储的数据，NebulaGraph 进程中其它时区相关数据，例如日志打印的时间等，仍然使用主机系统默认的时区。

logging 配置

名称	预设值	说明	是否支持运行时动态修改
`log_dir`	`logs`	存放 Meta 服务日志的目录，建议和数据保存在不同硬盘。	不支持
`minloglevel`	`0`	最小日志级别，即记录此级别或更高级别的日志。可选值为 `0` （INFO）、 `1` （WARNING）、 `2` （ERROR）、 `3` （FATAL）。建议在调试时设置为 `0` ，生产环境中设置为 `1` 。如果设置为 `4` ，NebulaGraph 不会记录任何日志。	支持
`v`	`0`	VLOG 日志详细级别，即记录小于或等于此级别的所有 VLOG 消息。可选值为 `0` 、 `1` 、 `2` 、 `3` 、 `4` 、 `5` 。glog 提供的 VLOG 宏允许用户定义自己的数字日志记录级别，并用参数 `v` 控制记录哪些详细消息。详情参见 Verbose Logging。	支持
`logbufsecs`	`0`	缓冲日志的最大时间，超时后输出到日志文件。 `0` 表示实时输出。单位：秒。	不支持
`redirect_stdout`	`true`	是否将标准输出和标准错误重定向到单独的输出文件。	不支持
`stdout_log_file`	`metad-stdout.log`	标准输出日志文件名称。	不支持
`stderr_log_file`	`metad-stderr.log`	标准错误日志文件名称。	不支持
`stderrthreshold`	`3`	要复制到标准错误中的最小日志级别（ `minloglevel` ）。	不支持
`timestamp_in_logfile_name`	`true`	日志文件名称中是否包含时间戳。 `true` 表示包含， `false` 表示不包含。	不支持

networking 配置

名称	预设值	说明	是否支持运行时动态修改
`meta_server_addrs`	`127.0.0.1:9559`	全部 Meta 服务的 IP（或主机名）和端口。多个 Meta 服务用英文逗号（,）分隔。	不支持
`local_ip`	`127.0.0.1`	Meta 服务的本地 IP（或主机名）。本地 IP 用于识别 nebula-metad 进程，如果是分布式集群或需要远程访问，请修改为对应地址。	不支持
`port`	`9559`	Meta 服务的 RPC 守护进程监听端口。同时还会使用相邻的 `+1` （ `9560` ）端口用于 Meta 服务之间的 Raft 通信。	不支持
`ws_ip`	`0.0.0.0`	HTTP 服务的 IP 地址。	不支持
`ws_http_port`	`19559`	HTTP 服务的端口。	不支持
`ws_storage_http_port`	`19779`	HTTP 协议监听 Storage 服务的端口，需要和 Storage 服务配置文件中的 `ws_http_port` 保持一致。仅存算合并版 NebulaGraph 需要设置本参数。	不支持

> **Caution**
>
> 使用 IP 时建议使用真实的 IP。否则某些情况下 `127.0.0.1/0.0.0.0` 无法正确解析。

storage 配置

名称	预设值	说明	是否支持运行时动态修改
`data_path`	`data/meta`	meta 数据存储路径。	不支持

misc 配置

名称	预设值	说明	是否支持运行时动态修改
`default_parts_num`	`10`	创建图空间时的默认分片数量。	不支持
`default_replica_factor`	`1`	创建图空间时的默认副本数量。	不支持
`heartbeat_interval_secs`	`10`	默认心跳间隔。请确保所有服务的 `heartbeat_interval_secs` 取值相同，否则会导致系统无法正常工作。单位：秒。	支持
`agent_heartbeat_interval_secs`	`60`	Agent 服务发送心跳的时间间隔。该值影响系统确定 Agent 服务离线状态的时间。	不支持

rocksdb options 配置

名称	预设值	说明	是否支持运行时动态修改
`rocksdb_wal_sync`	`true`	是否同步 RocksDB 的 WAL 日志。	不支持

最后更新: 2025年8月4日

6.1.3 Graph 服务配置

Graph 服务提供了两份初始配置文件 `nebula-graphd.conf.default` 和 `nebula-graphd.conf.production`，方便在不同场景中使用。文件的默认路径为 `/usr/local/nebula/etc/`。

> Caution
>
> - 不建议修改 `local_config` 的值为 `false`。修改后 NebulaGraph 服务启动后会先尝试从 Meta 服务获取缓存的配置信息，可能导致集群配置不一致，造成未知风险。
> - 不建议修改文档未介绍的配置项，除非已经熟悉源代码并完全了解配置项的作用。

配置文件使用方式

如需使用初始配置文件，从上述两个文件选择其一，删除后缀 `.default` 或 `.production`，Graph 服务才能将其识别为配置文件并从中获取配置信息。

配置文件参数值说明

配置文件内没有设置某个参数表示参数使用的是默认值。文件内只预设了部分参数的值，而且两份初始配置文件内的参数值也略有不同，本文的预设值以 `nebula-graphd.conf.default` 为准。

> Caution
>
> 配置文件中的部分参数值可以在运行时动态修改。本文将这些参数标记为支持运行时动态修改，并当 `local_config` 值设置为 `true` 时重启服务后配置会恢复为初始配置。详情参见修改配置。

如需查看所有的参数及其当前值，参见配置管理。

basics 配置

名称	预设值	说明	是否支持运行时动态修改
`daemonize`	`true`	是否启动守护进程。	不支持
`pid_file`	`pids/nebula-graphd.pid`	记录进程 ID 的文件。	不支持
`enable_optimizer`	`true`	是否启用优化器。	不支持
`timezone_name`	-	指定 NebulaGraph 的时区。初始配置文件中未设置该参数，使用需手动添加。系统默认值为 `UTC+00:00:00`。格式请参见 Specifying the Time Zone with TZ。例如，东八区的设置方式为 `--timezone_name=UTC+08:00`。	不支持
`default_charset`	`utf8`	创建图空间时的默认字符集。	不支持
`default_collate`	`utf8_bin`	创建图空间时的默认排序规则。	不支持
`local_config`	`true`	是否从配置文件获取配置信息。	不支持

> Note
>
> - 在插入时间类型 的属性值时，NebulaGraph 会根据 `timezone_name` 设置的时区将该时间值（TIMESTAMP 类型例外）转换成相应的 UTC 时间，因此在查询中返回的时间类型属性值为 UTC 时间。
> - `timezone_name` 参数只用于转换 NebulaGraph 中存储的数据，NebulaGraph 进程中其它时区相关数据，例如日志打印的时间等，仍然使用主机系统默认的时区。

logging 配置

名称	预设值	说明	是否支持运行时动态修改
`log_dir`	`logs`	存放 Graph 服务日志的目录，建议和数据保存在不同硬盘。	不支持
`minloglevel`	`0`	最小日志级别，即记录此级别或更高级别的日志。可选值为 `0`（INFO）、`1`（WARNING）、`2`（ERROR）、`3`（FATAL）。建议在调试时设置为 `0`，生产环境中设置为 `1`。如果设置为 `4`，NebulaGraph 不会记录任何日志。	支持
`v`	`0`	VLOG 日志详细级别，即记录小于或等于此级别的所有 VLOG 消息。可选值为 `0`、`1`、`2`、`3`、`4`、`5`。glog 提供的 VLOG 宏允许用户定义自己的数字日志记录级别，并用参数 `v` 控制记录哪些详细消息。详情参见 Verbose Logging。	支持
`logbufsecs`	`0`	缓冲日志的最大时间，超时后输出到日志文件。`0` 表示实时输出。单位：秒。	不支持
`redirect_stdout`	`true`	是否将标准输出和标准错误重定向到单独的输出文件。	不支持
`stdout_log_file`	`graphd-stdout.log`	标准输出日志文件名称。	不支持
`stderr_log_file`	`graphd-stderr.log`	标准错误日志文件名称。	不支持
`stderrthreshold`	`3`	要复制到标准错误中的最小日志级别（`minloglevel`）。	不支持
`timestamp_in_logfile_name`	`true`	日志文件名称中是否包含时间戳。`true` 表示包含，`false` 表示不包含。	不支持

query 配置

名称	预设值	说明	是否支持运行时动态修改
`accept_partial_success`	`false`	是否将部分成功视为错误。此配置仅适用于只读请求，写请求总是将部分成功视为错误。查询部分成功时，会提示 `Got partial result`。	支持
`session_reclaim_interval_secs`	`60`	将 Session 信息发送给 Meta 服务的间隔。单位：秒。	支持
`max_allowed_query_size`	`4194304`	最大查询语句长度。单位：字节。默认为 `4194304`，即 4MB。	支持

networking 配置

名称	预设值	说明	是否支持运行时动态修改
`meta_server_addrs`	`127.0.0.1:9559`	全部 Meta 服务的 IP（或主机名）和端口。多个 Meta 服务用英文逗号（,）分隔。	不支持
`local_ip`	`127.0.0.1`	Graph 服务的本地 IP（或主机名）。本地 IP 用于识别 nebula-graphd 进程，如果是分布式集群或需要远程访问，请修改为对应地址。	不支持
`listen_netdev`	`any`	监听的网络设备。	不支持
`port`	`9669`	Graph 服务的 RPC 守护进程监听端口。	不支持
`reuse_port`	`false`	是否启用 SO_REUSEPORT。	不支持
`listen_backlog`	`1024`	socket 监听的连接队列最大长度，调整本参数需要同时调整 `net.core.somaxconn` 。	不支持
`client_idle_timeout_secs`	`28800`	空闲连接的超时时间。取值范围为 1~604800，单位：秒。默认 8 小时。	不支持
`session_idle_timeout_secs`	`28800`	空闲会话的超时时间。取值范围为 1~604800。默认 8 小时。单位：秒。	不支持
`num_accept_threads`	`1`	接受传入连接的线程数。	不支持
`num_netio_threads`	`0`	网络 IO 线程数。 `0` 表示 CPU 核数。	不支持
`num_max_connections`	`0`	所有网络线程的最大活动连接数， `0` 表示没有限制。 每个网络线程的最大连接数= `num_max_connections` / `num_netio_threads` 。	不支持
`num_worker_threads`	`0`	执行用户查询的线程数。 `0` 表示 CPU 核数。	不支持
`ws_ip`	`0.0.0.0`	HTTP 服务的 IP 地址。	不支持
`ws_http_port`	`19669`	HTTP 服务的端口。	不支持
`heartbeat_interval_secs`	`10`	默认心跳间隔。请确保所有服务的 `heartbeat_interval_secs` 取值相同，否则会导致系统无法正常工作。单位：秒。	支持
`storage_client_timeout_ms`	-	Graph 服务与 Storage 服务的 RPC 连接超时时间。初始配置文件中未设置该参数，使用需手动添加。默认值为 `60000` 毫秒。	不支持
`slow_query_threshold_us`	`200000`	定义超过多长时间的查询为慢查询。单位：微秒。 注意：DML 语句的执行时间即使超过该值，也不会被记录为慢查询。	不支持
`ws_meta_http_port`	`19559`	HTTP 协议监听 Meta 服务的端口，需要和 Meta 服务配置文件中的 `ws_http_port` 保持一致。	不支持

> Caution
>
> 使用 IP 时建议使用真实的 IP。否则某些情况下 `127.0.0.1/0.0.0.0` 无法正确解析。

authorization 配置

名称	预设值	说明	是否支持运行时动态修改
`enable_authorize`	`false`	用户登录时是否进行身份验证。身份验证详情请参见身份验证。	不支持
`auth_type`	`password`	用户登录的身份验证方式。取值为 `password` 、 `ldap` 、 `cloud` 。	不支持

memory 配置

名称	预设值	说明	是否支持运行时动态修改
`system_memory_high_watermark_ratio`	`0.8`	内存高水位报警机制的触发阈值。系统内存占用率高于该值会触发报警机制，NebulaGraph 会停止接受查询。	支持

metrics 配置

名称	预设值	说明	是否支持运行时动态修改
`enable_space_level_metrics`	`false`	开启后可打开图空间级别的监控，对应的监控指标名称中包含图空间的名称，例如 `query_latency_us{space=basketballplayer}.avg.3600` 。支持的监控指标可用 `curl` 命令查看，详细说明参见查询监控指标。	不支持

session 配置

名称	预设值	说明	是否支持运行时动态修改
`max_sessions_per_ip_per_user`	`300`	相同用户和 IP 地址可以创建的最大活跃会话数。	不支持

experimental 配置

名称	预设值	说明	是否支持运行时动态修改
`enable_experimental_feature`	`false`	实验性功能开关。可选值为 `true` 和 `false` 。	不支持
`enable_data_balance`	`true`	是否开启均衡分片功能。仅当 `enable_experimental_feature` 为 `true` 时生效。	不支持

memory tracker 配置

> Note
>
> Memory Tracker 是一个内存管理工具，用于监控和限制内存使用。对于大型查询，Memory Tracker 可以防止 OOM 问题。如果在容器环境中使用 Memory Tracker，需要在 Graph 服务的配置文件中添加相关配置。操作如下：
>
> 1. 创建目录 `/sys/fs/cgroup/graphd/`，添加并配置 `memory.max`。
> 2. 在 `etc/nebula-graphd.conf` 添加以下配置。
>
> ```
> --containerized=true
> --cgroup_v2_controllers=/sys/fs/cgroup/graphd/cgroup.controllers
> --cgroup_v2_memory_stat_path=/sys/fs/cgroup/graphd/memory.stat
> --cgroup_v2_memory_max_path=/sys/fs/cgroup/graphd/memory.max
> --cgroup_v2_memory_current_path=/sys/fs/cgroup/graphd/memory.current
> ```
>
> 详细信息，请参见容器/cgroup 模式。

名称	预设值	说明	是否支持运行时动态修改
`memory_tracker_limit_ratio`	`0.8`	取值可设置为：`(0, 1]`、`2`、`3`。 警惕：设置该参数时请确保 **`system_memory_high_watermark_ratio`** 的值不为 **`1`**，否则该参数的值不生效。 `(0, 1]`：可用内存的百分比。计算公式：可用内存的百分比 = 可用内存 / (总内存 - 保留内存)。 当正在进行的查询导致内存使用超过配置的内存限制时，该查询会失败，并在失败后释放内存。 注意：对于云上和本地节点混合部署的集群，需要根据实际情况调小该参数。例如，当预期 Graphd 只占用 50% 的内存时，该参数的值可设置为小于 `0.5`。 `2`：动态自适应模式（Dynamic Self Adaptive），Memory Tracker 会根据系统当前的可用内存，动态调整可用内存。 注意：此功能为实验性功能，由于动态自适应不能做到实时监控操作系统内存使用情况，在一些大内存分配的场景，还是会存在 OOM 可能。 `3`：关掉 Memory Tracker，Memory Tracker 将只记录内存使用情况，即使超过限额，也不会干预执行。	支持
`memory_tracker_untracked_reserved_memory_mb`	`50`	保留内存的大小，单位：MB。	支持
`memory_tracker_detail_log`	`false`	是否定期生成较详细的内存跟踪日志。当值为 `true` 时，会定期生成内存跟踪日志。	支持
`memory_tracker_detail_log_interval_ms`	`60000`	内存跟踪日志的生成时间间隔，单位：毫秒。仅当 `memory_tracker_detail_log` 为 `true` 时，该参数生效。	支持
`memory_purge_enabled`	`true`	是否定期开启内存清理功能。当值为 `true` 时，会定期清理内存。	支持
`memory_purge_interval_seconds`	`10`	内存清理的时间间隔，单位：秒。`memory_purge_enabled` 为 `true` 时，该参数生效。	支持

performance optimization 配置

名称	预设值	说明	是否支持运行时动态修改
`max_job_size`	`1`	最大作业并发数，即查询执行时在可以并发执行的阶段所采用的最大线程数。建议为物理 CPU 核数的一半。	支持
`min_batch_size`	`8192`	处理数据集的最小批处理大小。仅在 `max_job_size` 大于 1 时生效。	支持
`optimize_appendvertices`	`false`	启用后，执行 `MATCH` 语句时不过滤悬挂边。	支持
`path_batch_size`	`10000`	每个线程构建的路径数。	支持

最后更新: 2025年8月4日

6.1.4 Storage 服务配置

Storage 服务提供了两份初始配置文件 `nebula-storaged.conf.default` 和 `nebula-storaged.conf.production`，方便在不同场景中使用。文件的默认路径为 `/usr/local/nebula/etc/`。

> **Caution**
>
> - 不建议修改 `local_config` 的值为 `false`。修改配置并重启 Storage 服务，会先尝试从 Meta 服务获取缓存的配置信息，可能导致集群配置不一致，造成未知风险。
> - 不建议修改文档未介绍的配置项，除非已经熟悉源代码并完全了解配置项的作用。

配置文件使用方式

如需使用初始配置文件，从上述两个文件选择其一，删除后缀 `.default` 或 `.production`，Storage 服务才能将其识别为配置文件并从中获取配置信息。

配置文件参数值说明

配置文件内没有设置某个参数表示参数使用的是默认值。文件内只预设了部分参数的值，而且两份初始配置文件内的参数值也略有不同，本文的预设值以 `nebula-storaged.conf.default` 文件为准，其中没有的参数则以 `nebula-storaged.conf.production` 文件为准。

> **Caution**
>
> 配置文件中的部分参数值可以在运行时动态修改。本文将这些参数标记为支持运行时动态修改，并当 `local_config` 值设置为 `true` 时重启服务后配置会恢复为初始配置。详情参见修改配置。

> **Note**
>
> Raft Listener 的配置和 Storage 服务配置不同，详情请参见部署 Raft listener。

如需查看所有的参数及其当前值，参见配置管理。

basics 配置

名称	预设值	说明	是否支持运行时动态修改
`daemonize`	`true`	是否启动守护进程。	不支持
`pid_file`	`pids/nebula-storaged.pid`	记录进程 ID 的文件。	不支持
`timezone_name`	`UTC+00:00:00`	指定 NebulaGraph 的时区。初始配置文件中未设置该参数，如需使用请手动添加。格式请参见 Specifying the Time Zone with TZ。例如，东八区的设置方式为 `--timezone_name=UTC+08:00`。	不支持
`local_config`	`true`	是否从配置文件获取配置信息。	不支持

> Note
>
> - 在插入时间类型的属性值时，NebulaGraph 会根据 `timezone_name` 设置的时区将该时间值（TIMESTAMP 类型例外）转换成相应的 UTC 时间，因此在查询中返回的时间类型属性值为 UTC 时间。
> - `timezone_name` 参数只用于转换 NebulaGraph 中存储的数据，NebulaGraph 进程中其它时区相关数据，例如日志打印的时间等，仍然使用主机系统默认的时区。

logging 配置

名称	预设值	说明	是否支持运行时动态修改
`log_dir`	`logs`	存放 Storage 服务日志的目录，建议和数据保存在不同硬盘。	不支持
`minloglevel`	`0`	最小日志级别，即记录此级别或更高级别的日志。可选值：`0`（INFO）、`1`（WARNING）、`2`（ERROR）、`3`（FATAL）。建议在调试时设置为`0`，生产环境中设置为`1`。如果设置为`4`，NebulaGraph 不会记录任何日志。	支持
`v`	`0`	VLOG 日志详细级别，即记录小于或等于此级别的所有 VLOG 消息。可选值为`0`、`1`、`2`、`3`、`4`、`5`。glog 提供的 VLOG 宏允许用户定义自己的数字日志记录级别，并用参数`v`控制记录哪些详细消息。详情参见 Verbose Logging。	支持
`logbufsecs`	`0`	缓冲日志的最大时间，超时后输出到日志文件。`0`表示实时输出。单位：秒。	不支持
`redirect_stdout`	`true`	是否将标准输出和标准错误重定向到单独的输出文件。	不支持
`stdout_log_file`	`storaged-stdout.log`	标准输出日志文件名称。	不支持
`stderr_log_file`	`storaged-stderr.log`	标准错误日志文件名称。	不支持
`stderrthreshold`	`3`	要复制到标准错误中的最小日志级别（`minloglevel`）。	不支持
`timestamp_in_logfile_name`	`true`	日志文件名称中是否包含时间戳。`true`表示包含，`false`表示不包含。	不支持

networking 配置

名称	预设值	说明	是否支持运行时动态修改
`meta_server_addrs`	`127.0.0.1:9559`	全部 Meta 服务的 IP（或主机名）和端口。多个 Meta 服务用英文逗号（,）分隔。	不支持
`local_ip`	`127.0.0.1`	Storage 服务的本地 IP（或主机名）。本地 IP 地址用于识别 nebula-storaged 进程，如果是分布式集群或需要远程访问，请修改为对应地址。	不支持
`port`	`9779`	Storage 服务的 RPC 守护进程监听端口。同时还会使用相邻的 `-1`（`9778`）和 `+1`（`9780`）端口。 `9778`：Admin 服务（Storage 接收 Meta 命令的服务）占用的端口。 `9780`：Storage 服务之间的 Raft 通信端口。	不支持
`ws_ip`	`0.0.0.0`	HTTP 服务的 IP 地址。	不支持
`ws_http_port`	`19779`	HTTP 服务的端口。	不支持
`heartbeat_interval_secs`	`10`	默认心跳间隔。请确保所有服务的 `heartbeat_interval_secs` 取值相同，否则会导致系统无法正常工作。单位：秒。	支持

> Caution
>
> 使用 IP 时建议使用真实的 IP。否则某些情况下 `127.0.0.1/0.0.0.0` 无法正确解析。

raft 配置

名称	预设值	说明	是否支持运行时动态修改
`raft_heartbeat_interval_secs`	`30`	Raft 选举超时时间。单位：秒。	支持
`raft_rpc_timeout_ms`	`500`	Raft 客户端的远程过程调用（RPC）超时时间。单位：毫秒。	支持
`wal_ttl`	`14400`	Raft WAL 的有效时间。单位：秒。	支持

disk 配置

名称	预设值	说明	是否支持运行时动态修改
`data_path`	`data/storage`	数据存储路径，多个路径用英文逗号（,）分隔。一个 RocksDB 实例对应一个路径。	不支持
`minimum_reserved_bytes`	`268435456`	每个数据存储路径的剩余空间最小值，低于该值时，可能会导致集群数据写入失败。单位：字节。	不支持
`rocksdb_batch_size`	`4096`	批量操作的缓存大小。单位：字节。	不支持
`rocksdb_block_cache`	`4`	BlockBasedTable 的默认块缓存大小。单位：兆字节。	不支持
`disable_page_cache`	`false`	允许或禁止 NebulaGraph 使用操作系统的页缓存。默认值为 `false`，表示允许使用 page cache。当值为 `true` 时，禁止 NebulaGraph 使用 page cache，此时须设置充足的块缓存（block cache）空间。	不支持
`engine_type`	`rocksdb`	存储引擎类型。	不支持
`rocksdb_compression`	`lz4`	压缩算法，可选值：`no`、`snappy`、`lz4`、`lz4hc`、`zlib`、`bzip2`、`zstd`。 该参数会修改每一层的压缩算法，如需为不同层级设置不同的压缩算法，请使用 `rocksdb_compression_per_level` 参数。	不支持
`rocksdb_compression_per_level`	-	为不同层级设置不同的压缩算法。优先级高于 `rocksdb_compression`。例如 `no:no:lz4:lz4:snappy:zstd:snappy`。也可以不设置某个层级的压缩算法，例如 `no:no:lz4:lz4::zstd`，此时 L4、L6 层使用 `rocksdb_compression` 参数的压缩算法。	不支持
`enable_rocksdb_statistics`	`false`	是否启用 RocksDB 的数据统计。	不支持
`rocksdb_stats_level`	`kExceptHistogramOrTimers`	RocksDB 的数据统计级别。可选值：`kExceptHistogramOrTimers`（禁用计时器统计，跳过柱状图统计）、`kExceptTimers`（跳过计时器统计）、`kExceptDetailedTimers`（收集除互斥锁和压缩花费时间之外的所有统计数据）、`kExceptTimeForMutex` 收集除互斥锁花费时间之外的所有统计数据）、`kAll`（收集所有统计数据）。	不支持
`enable_rocksdb_prefix_filtering`	`true`	是否启用 prefix bloom filter，启用时可以提升图遍历速度，但是会增加内存消耗。	不支持

名称	预设值	说明	是否支持运行时动态修改
`enable_rocksdb_whole_key_filtering`	`false`	是否启用 whole key bloom filter。	不支持
`rocksdb_filtering_prefix_length`	`12`	每个 key 的 prefix 长度。可选值：`12`（分片 ID+点 ID）、`16`（分片 ID+点 ID+TagID/Edge typeID）。单位：字节。	不支持
`enable_partitioned_index_filter`	`false`	设置为 `true` 可以降低 bloom 过滤器占用的内存大小，但是在某些随机寻道（random-seek）的情况下，可能会降低读取性能。初始配置文件中未设置该参数，如需使用请手动添加。	不支持

rocksdb options 配置

名称	预设值	说明	是否支持运行时动态修改
`rocksdb_db_options`	`{}`	RocksDB database 选项。	不支持
`rocksdb_column_family_options`	`{"write_buffer_size":"67108864", "max_write_buffer_number":"4", "max_bytes_for_level_base":"268435456"}`	RocksDB column family 选项。	不支持
`rocksdb_block_based_table_options`	`{"block_size":"8192"}`	RocksDB block based table 选项。	不支持

rocksdb options 配置的格式为 `{"<option_name>":"<option_value>"}`，多个选项用英文逗号（,）隔开。

`rocksdb_db_options` 和 `rocksdb_column_family_options` 支持的选项如下:

- `rocksdb_db_options`

```
max_total_wal_size
delete_obsolete_files_period_micros
max_background_jobs
stats_dump_period_sec
compaction_readahead_size
writable_file_max_buffer_size
bytes_per_sync
wal_bytes_per_sync
delayed_write_rate
avoid_flush_during_shutdown
max_open_files
stats_persist_period_sec
stats_history_buffer_size
strict_bytes_per_sync
enable_rocksdb_prefix_filtering
enable_rocksdb_whole_key_filtering
rocksdb_filtering_prefix_length
num_compaction_threads
rate_limit
```

- `rocksdb_column_family_options`

```
write_buffer_size
max_write_buffer_number
level0_file_num_compaction_trigger
level0_slowdown_writes_trigger
level0_stop_writes_trigger
target_file_size_base
target_file_size_multiplier
max_bytes_for_level_base
max_bytes_for_level_multiplier
disable_auto_compactions
```

参数的详细说明请参见 RocksDB 官方文档。

misc 配置

> Caution
>
> 下表中的 `snapshot` 与 NebulaGraph 快照是不同的概念，这里的 `snapshot` 指 Raft 同步过程中 leader 上的存量数据。

名称	预设值	说明	是否支持运行时动态修改
`query_concurrently`	`true`	是否开启多线程查询。开启后可以提高单个查询的时延性能，但大压力下会降低整体的吞吐量。	支持
`auto_remove_invalid_space`	`true`	在执行 `DROP SPACE` 后，会删除指定图空间，该参数设置是否同时删除指定图空间内的所有数据。当值为 `true` 时，同时删除指定图空间内的所有数据。	支持
`num_io_threads`	`16`	网络 I/O 线程的数量，用于发送 RPC 请求和接收响应。	不支持
`num_max_connections`	`0`	所有网络线程的最大活动连接数，`0` 表示没有限制。 每个网络线程的最大连接数= `num_max_connections` / `num_netio_threads` 。	不支持
`num_worker_threads`	`32`	Storage 的 RPC 服务的工作线程数量。	不支持
`max_concurrent_subtasks`	`10`	TaskManager 并发执行子任务的最大个数。	不支持
`snapshot_part_rate_limit`	`10485760`	Raft leader 向 Raft group 中其它成员同步存量数据时的限速。单位：字节/秒。	支持
`snapshot_batch_size`	`1048576`	Raft leader 向 Raft group 中其它成员同步存量数据时每批发送的数据量。单位：字节。	支持
`rebuild_index_part_rate_limit`	`4194304`	重建索引过程中，Raft leader 向 Raft group 中其它成员同步索引数据时的限速。单位：字节/秒。	支持
`rebuild_index_batch_size`	`1048576`	重建索引过程中，Raft leader 向 Raft group 中其它成员同步索引数据时每批发送的数据量。单位：字节。	支持

memory tracker 配置

> Note
>
> Memory Tracker 是一个内存管理工具，用于监控和限制内存使用。对于大型查询，Memory Tracker 可以防止 OOM 问题。如果在容器环境中使用 Memory Tracker，需要在 Storage 服务的配置文件中添加相关配置。操作如下：
>
> 1. 创建目录 `/sys/fs/cgroup/storaged/`，添加并配置 `memory.max`。
> 2. 在 `etc/nebula-storaged.conf` 添加以下配置。
>
> ```
> --containerized=true
> --cgroup_v2_controllers=/sys/fs/cgroup/graphd/cgroup.controllers
> --cgroup_v2_memory_stat_path=/sys/fs/cgroup/graphd/memory.stat
> --cgroup_v2_memory_max_path=/sys/fs/cgroup/graphd/memory.max
> --cgroup_v2_memory_current_path=/sys/fs/cgroup/graphd/memory.current
> ```
>
> 详细信息，请参见图数据库 NebulaGraph 的内存管理实践之 Memory Tracker。

名称	预设值	说明	是否支持运行时动态修改
`memory_tracker_limit_ratio`	`0.8`	取值可设置为：`(0, 1]`、`2`、`3`。 `(0, 1]`：可用内存的百分比。计算公式：可用内存的百分比 = 可用内存 / (总内存 - 保留内存)。 当正在进行的查询导致内存使用超过配置的内存限制时，该查询会失败，并在失败后释放内存。 注意：对于云上和本地节点混合部署的集群，需要根据实际情况调小该参数。例如，当预期 Graphd 只占用 50% 的内存时，该参数的值可设置为小于 `0.5`。 `2`：动态自适应模式（Dynamic Self Adaptive），Memory Tracker 会根据系统当前的可用内存，动态调整可用内存。 注意：此功能为实验性功能，由于动态自适应不能做到实时监控操作系统内存使用情况，在一些大内存分配的场景，还是会存在 OOM 可能。 `3`：关掉 Memory Tracker，Memory Tracker 将只记录内存使用情况，即使超过限额，也不会干预执行。	支持
`memory_tracker_untracked_reserved_memory_mb`	`50`	保留内存的大小，单位：MB。	支持
`memory_tracker_detail_log`	`false`	是否定期生成较详细的内存跟踪日志。当值为 `true` 时，会定期生成内存跟踪日志。	支持
`memory_tracker_detail_log_interval_ms`	`60000`	内存跟踪日志的生成时间间隔，单位：毫秒。仅当 `memory_tracker_detail_log` 为 `true` 时，该参数生效。	支持
`memory_purge_enabled`	`true`	是否定期开启内存清理功能。当值为 `true` 时，会定期清理内存。	支持
`memory_purge_interval_seconds`	`10`	内存清理的时间间隔，单位：秒。`memory_purge_enabled` 为 `true` 时，该参数生效。	支持

超级节点处理（出入边数量极多的点）

在每个点出发的查询获取到边时，直接截断。目的是避免超级节点的邻边过多，单个查询占用过多的硬盘和内存。截取前 `max_edge_returned_per_vertex` 个边，多余的边不返回。该参数作用于全局，不用于单个 space。

属性名	默认值	说明	是否支持运行时动态修改
max_edge_returned_per_vertex	`2147483647`	每个稠密点，最多返回多少条边，多余的边截断不返回。初始配置文件中未设置该参数，如需使用请手动添加。	不支持

数据量大而内存不够时

> Warning
>
> 一个图空间至少占用大约 300 MB 的内存。

如果数据量很大但内存不够，则可以尝试把 storage 配置中的 `enable_partitioned_index_filter` 设置为 `true`，减少 `rocksdb_block_cache`。但由于缓存了较少的 RocksDB 索引，性能可能会受影响。此外，当通过语句拉取大量数据时，仍存在 OOM 的可能。

最后更新: 2025年8月4日

6.1.5 Linux 内核配置

本文介绍与 NebulaGraph 相关的 Linux 内核配置，并介绍如何修改配置。

资源控制

虽然可以用 `ulimit` 进行资源控制，但是所做的更改仅对当前会话或子进程生效。如果需要永久生效，请编辑文件 `/etc/security/limits.conf` 。配置如下：

```
# <domain>   <type>   <item>      <value>
*       soft     core      unlimited
*       hard     core      unlimited
*       soft     nofile     130000
*       hard     nofile     130000
```

> **Note**
>
> 配置修改后会对新的会话生效。

参数说明如下。

参数	说明
`domain`	控制域。可以是用户名称、用户组名称（以 `@` 开头），或者用 `*` 表示所有用户。
`type`	控制类型。可以是 `soft` 或 `hard` 。 `soft` 表示资源的软阈值（默认阈值）， `hard` 表示用户可以设置的最大值。可以使用 `ulimit` 命令提高 `soft` ，但不能超过 `hard` 。
`item`	资源类型。例如 `core` 限制核心转储文件的大小， `nofile` 限制一个进程可以打开的最大文件描述符数量。
`value`	资源限制值。可以是一个数字，或者 unlimited 表示没有限制。

可以执行 `man limits.conf` 查看更多帮助信息。

内存

VM.SWAPPINESS

`vm.swappiness` 是触发虚拟内存（swap）的空闲内存百分比。值越大，使用 swap 的可能性就越大，建议设置为 0，表示首先删除页缓存。需要注意的是，0 表示尽量不使用 swap。

VM.MIN_FREE_KBYTES

`vm.min_free_kbytes` 用于设置 Linux 内核保留的最小空闲千字节数。如果系统内存足够，建议设置较大值。例如物理内存为 128 GB，可以将 `vm.min_free_kbytes` 设置为 5 GB。如果值太小，会导致系统无法申请足够的连续物理内存。

VM.MAX_MAP_COUNT

`vm.max_map_count` 用于限制单个进程的 VMA（虚拟内存区域）数量。默认值为 `65530` ，对于绝大多数应用程序来说已经足够。如果应用程序因为内存消耗过大而报错，请增大本参数的值。

VM.DIRTY_*

`vm.dirty_*` 是一系列控制系统脏数据缓存的参数。对于写密集型场景，用户可以根据需要进行调整（吞吐量优先或延迟优先），建议使用系统默认值。

透明大页

透明大页（Transparent Huge Pages, THP）是一种 Linux 内核的内存管理特性，可以提高系统使用大页的能力。在多数数据库系统中，透明大页会降低性能，建议关闭。

操作如下：

1. 编辑 GRUB 配置文件 `/etc/default/grub`。

```
sudo vi /etc/default/grub
```

2. 在 `GRUB_CMDLINE_LINUX` 选项中添加 `transparent_hugepage=never`，然后保存并退出。

```
GRUB_CMDLINE_LINUX="... transparent_hugepage=never"
```

3. 更新 GRUB 配置。

- CentOS

```
sudo grub2-mkconfig -o /boot/grub2/grub.cfg
```

- Ubuntu

```
sudo update-grub
```

4. 重启操作系统。

```
sudo reboot
```

如果不重启可临时关闭透明大页，直到下次重启。

```
echo 'never' > /sys/kernel/mm/transparent_hugepage/enabled
echo 'never' > /sys/kernel/mm/transparent_hugepage/defrag
```

网络

NET.IPV4.TCP_SLOW_START_AFTER_IDLE

`net.ipv4.tcp_slow_start_after_idle` 默认值为 1，会导致闲置一段时间后拥塞窗口超时，建议设置为 `0`，尤其适合大带宽高延迟场景。

NET.CORE.SOMAXCONN

`net.core.somaxconn` 用于限制 socket 监听的连接队列数量。默认值为 `128`。对于有大量突发连接的场景，建议设置为不低于 `1024`。

NET.IPV4.TCP_MAX_SYN_BACKLOG

`net.ipv4.tcp_max_syn_backlog` 用于限制处于 SYN_RECV（半连接）状态的 TCP 连接数量。默认值为 `128`。对于有大量突发连接的场景，建议设置为不低于 `1024`。

NET.CORE.NETDEV_MAX_BACKLOG

`net.core.netdev_max_backlog` 用于限制队列中数据包的数量。默认值为 `1000`，建议设置为 `10000` 以上，尤其是万兆网卡。

NET.IPV4.TCP_KEEPALIVE_*

`net.ipv4.tcp_keepalive_*` 是一系列保持 TCP 连接存活的参数。对于使用四层透明负载均衡的应用程序，如果空闲连接异常断开，请增大 `tcp_keepalive_time` 和 `tcp_keepalive_intvl` 的值。

NET.IPV4.TCP_WMEM/RMEM

TCP 套接字发送/接收缓冲池的最小、最大、默认空间。对于大连接，建议设置为 `带宽（GB）*往返时延（ms）`。

SCHEDULER

对于 SSD 设备，建议将 `scheduler` 设置为 `noop` 或者 `none`，路径为 `/sys/block/DEV_NAME/queue/scheduler`。

其他参数

KERNEL.CORE_PATTERN

建议设置为 `core`，并且将 `kernel.core_uses_pid` 设置为 `1`。

修改参数

SYSCTL 命令

- `sysctl <conf_name>`

  查看当前参数值。

- `sysctl -w <conf_name>=<value>`

  临时修改参数值，立即生效，重启后恢复原值。

- `sysctl -p [<file_path>]`

  从指定配置文件里加载 Linux 系统参数，默认从 `/etc/sysctl.conf` 加载。

PRLIMIT

命令 `prlimit` 可以获取和设置进程资源的限制，结合 `sudo` 可以修改硬阈值，例如，`prlimit --nofile=140000 --pid=$$` 调整当前进程允许的打开文件的最大数量为 `140000`，立即生效，此命令仅支持 RedHat 7u 或更高版本。

最后更新: 2025年8月4日

6.2 日志

6.2.1 运行日志

运行日志通常提供给 DBA 或开发人员查看，当系统出现故障，DBA 或开发人员可以根据运行日志定位问题。

NebulaGraph 默认使用 glog 打印运行日志，使用 gflags 控制日志级别，并在运行时通过 HTTP 接口动态修改日志级别，方便跟踪问题。

运行日志目录

运行日志的默认目录为 `/usr/local/nebula/logs/` 。

如果在 NebulaGraph 运行过程中删除运行日志目录，日志不会继续打印，但是不会影响业务。重启服务可以恢复正常。

配置说明

- `minloglevel` ：最小日志级别，即不会记录低于这个级别的日志。可选值为 `0` （INFO）、 `1` （WARNING）、 `2` （ERROR）、 `3` （FATAL）。建议在调试时设置为 `0` ，生产环境中设置为 `1` 。如果设置为 `4` ，NebulaGraph 不会记录任何日志。
- `v` ：日志详细级别，值越大，日志记录越详细。可选值为 `0` 、 `1` 、 `2` 、 `3` 。

Meta 服务、Graph 服务和 Storage 服务的日志级别可以在各自的配置文件中查看，默认路径为 `/usr/local/nebula/etc/` 。

查看运行日志级别

使用如下命令查看当前所有的 gflags 参数（包括日志参数）：

```
$ curl <ws_ip>:<ws_port>/flags
```

参数	说明
`ws_ip`	HTTP 服务的 IP 地址，可以在配置文件中查看。默认值为 `127.0.0.1` 。
`ws_port`	HTTP 服务的端口，可以在配置文件中查看。默认值分别为 `19559` （Meta）、 `19669` （Graph） `19779` （Storage）。

示例如下：

- 查看 Meta 服务当前的最小日志级别：

```
$ curl 127.0.0.1:19559/flags | grep 'minloglevel'
```

- 查看 Storage 服务当前的日志详细级别：

```
$ curl 127.0.0.1:19779/flags | grep -w 'v'
```

修改运行日志级别

使用如下命令修改运行日志级别：

```
$ curl -X PUT -H "Content-Type: application/json" -d '{"<key>":<value>[,"<key>":<value>]}' "<ws_ip>:<ws_port>/flags"
```

参数	说明
`key`	待修改的运行日志类型，可选值请参见配置说明。
`value`	运行日志级别，可选值请参见配置说明。
`ws_ip`	HTTP 服务的 IP 地址，可以在配置文件中查看。默认值为 `127.0.0.1`。
`ws_port`	HTTP 服务的端口，可以在配置文件中查看。默认值分别为 `19559`（Meta）、`19669`（Graph）`19779`（Storage）。

示例如下：

```
$ curl -X PUT -H "Content-Type: application/json" -d '{"minloglevel":0,"v":3}' "127.0.0.1:19779/flags" # storaged
$ curl -X PUT -H "Content-Type: application/json" -d '{"minloglevel":0,"v":3}' "127.0.0.1:19669/flags" # graphd
$ curl -X PUT -H "Content-Type: application/json" -d '{"minloglevel":0,"v":3}' "127.0.0.1:19559/flags" # metad
```

如果在NebulaGraph运行时修改了运行日志级别，重启服务后会恢复为配置文件中设置的级别，如果需要永久修改，请修改配置文件并重启服务。

RocksDB 运行日志

RocksDB 的运行日志通常在 `/usr/local/nebula/data/storage/nebula/$id/data/LOG`，其中 `$id` 为实例号。该日志通常用于调试 RocksDB 参数。

回收日志

glog 本身不支持回收日志，如果需要回收日志，可以使用 Linux 系统中定时任务（Cron Jobs）来定期删除旧的日志文件。或者，使用日志管理工具 logrotate 来轮转日志以定期归档和删除日志。

使用定时任务回收日志

本文以回收 Graph 服务的运行日志为例，说明如何使用定时任务来定期删除旧的日志文件。操作步骤如下：

1. 在 Graph 服务配置文件中，进行如下配置，然后重启服务。

```
timestamp_in_logfile_name = true
max_log_size = 500
```

- 设置 `timestamp_in_logfile_name` 为 `true`，这样日志文件名中会包含时间戳，以定期删除旧的日志文件。
- 添加 `max_log_size` 参数，设置单个日志文件的最大大小，例如 `500`。超过这个大小后，会自动创建新的日志文件，单位 MB，默认值为 `1800`。

2. 在 Linux 系统中，使用如下命令编辑定时任务：

```
crontab -e
```

3. 在定时任务中添加命令，以定期删除旧的日志文件。

```
* * * * * find <log_path> -name "<YourProjectName>" -mtime +7 -delete
```

> Caution
>
> 以上命令中的 `find` 命令需要使用 root 用户或者具有 sudo 权限的用户来执行。

- `* * * * *`：定时任务的时间字段，五个星号表示这个任务每分钟都会执行。其他设置，参见Cron Expression。
- `<log_path>`：服务运行日志文件的路径，例如 `/usr/local/nebula/logs`。
- `<YourProjectName>`：日志文件名，例如 `nebula-graphd.*`。
- `-mtime +7`：表示删除更新时间超过 7 天的日志文件。也可以使用 `-mmin +n`，表示删除更新时间超过 n 分钟的日志文件。详情参见 find 命令。
- `-delete`：表示删除满足条件的文件。

例如，每天凌晨 3 点自动删除更新时间超过 7 天的 Graph 服务运行日志文件的命令：

```
0 3 * * * find /usr/local/nebula/logs -name nebula-graphd.* -mtime +7 -delete
```

4. 保存定时任务。

使用 LOGROTATE 回收日志

用户可以使用 logrotate 工具对指定的日志文件进行轮转，以达到归档和回收日志的目的。

> Note
>
> 需要使用 root 用户或者具有 sudo 权限的用户来安装 logrotate 或者运行 logrotate。

本文以回收 Graph 服务 `INFO` 级别的日志文件（ `/usr/local/nebula/logs/nebula-graphd.INFO.impl` ）为例说明如何使用 logrotate 工具。操作步骤如下：

1. 在 Graph 服务配置文件中，将 `timestamp_in_logfile_name` 设置为 `false`，以便 logrotate 工具可以识别日志文件名。然后重启服务。

```
timestamp_in_logfile_name = false
```

2. 安装 logrotate。

- Debian/Ubuntu：

```
sudo apt-get install logrotate
```

- CentOS/RHEL：

```
sudo yum install logrotate
```

3. 创建 logrotate 配置文件，添加日志轮转规则，然后保存配置文件。

在 `/etc/logrotate.d` 目录下，创建一个新的 logrotate 配置文件 `nebula-graphd.INFO`。

```
sudo vim /etc/logrotate.d/nebula-graphd.INFO
```

添加以下内容：

```
# 需配置日志文件的绝对路径
# 并且文件名不能为软链接文件，如不能为`nebula-graph.INFO`
/usr/local/nebula/logs/nebula-graphd.INFO.impl {
  daily
  rotate 2
  copytruncate
  nocompress
  missingok
  notifempty
  create 644 root root
  dateext
  dateformat .%Y-%m-%d-%s
  maxsize 1k
}
```

参数	说明
`daily`	每天轮转日志。可用的时间单位有：`hourly`、`daily`、`weekly`、`monthly`、`yearly`。
`rotate 2`	在删除前日志文件前，其被轮转的次数。即保留最近生成的 2 个日志文件。
`copytruncate`	将当前日志文件复制一份，然后清空当前日志文件。
`nocompress`	不压缩旧的日志文件。
`missingok`	如果日志文件丢失，不报告错误。
`notifempty`	如果日志文件为空，不进行轮转。
`create 644 root root`	创建新的日志文件，并设置适当的权限和所有者。
`dateext`	在日志文件名中添加日期后缀。 默认是当前日期。默认是 `-%Y%m%d` 的后缀。可用 `dateformat` 选项扩展配置。
`dateformat .%Y-%m-%d-%s`	必须配合 `dateext` 使用，紧跟在下一行出现，定义文件切割后的文件名。 在V3.9.0 之前，只支持 `%Y`、`%m`、`%d`、`%s` 参数。在 V3.9.0 及之后，支持 %H 参数。
`maxsize 1k`	当日志文件大小超过 `1` 千字节（`1024` 字节）或者超过设定的周期（如 `daily`）时，进行日志轮转。可用的大小单位有：`k`、`M`，默认单位为字节。

用户可以根据实际需求修改配置文件中的参数。更多关于参数的配置及解释，参见 logrotate。

4. 测试 logrotate 配置。

为了验证 logrotate 的配置是否正确，可以使用以下命令来进行测试：

```
sudo logrotate --debug /etc/logrotate.d/nebula-graphd.INFO
```

5. 运行 logrotate。

尽管 `logrotate` 通常由定时作业自动执行，但也可以手动执行以下命令，以立即进行日志轮转：

```
sudo logrotate -fv /etc/logrotate.d/nebula-graphd.INFO
```

`-fv`： `f` 表示强制执行， `v` 表示打印详细信息。

6. 查看日志轮转结果。

   日志轮转后，会在 `/usr/local/nebula/logs` 目录下看到新的日志文件，例如 `nebula-graphd.INFO.impl.2024-01-04-1704338204` 。原始日志内容会被清空，但文件会被保留，新日志继续写入。当日志文件数量超过 `rotate` 设置的值时，会删除最旧的日志文件。

   例如， `rotate 2` 表示保留最近生成的 2 个日志文件，当日志文件数量超过 2 个时，会删除最旧的日志文件。

```
[test@test logs]$ ll
-rw-r--r-- 1 root root    0 Jan  4 11:18 nebula-graphd.INFO.impl
-rw-r--r-- 1 root root 6894 Jan  4 11:16 nebula-graphd.INFO.impl.2024-01-04-1704338204 # 当新的日志文件生成时，此文件被删除
-rw-r--r-- 1 root root  222 Jan  4 11:18 nebula-graphd.INFO.impl.2024-01-04-1704338287
[test@test logs]$ ll
-rw-r--r-- 1 root root    0 Jan  4 11:18 nebula-graphd.INFO.impl
-rw-r--r-- 1 root root  222 Jan  4 11:18 nebula-graphd.INFO.impl.2024-01-04-1704338287
-rw-r--r-- 1 root root  222 Jan  4 11:18 nebula-graphd.INFO.impl.2024-01-04-1704338339 # 新生成的日志文件
```

如果用户需要对多个日志文件进行轮转，可以在 `/etc/logrotate.d` 目录下创建多个配置文件，每个配置文件对应一个日志文件。例如，用户需要对 Meta 服务的 `INFO` 级别日志文件和 `WARNING` 级别日志文件进行轮转，可以创建两个配置文件 `nebula-metad.INFO` 和 `nebula-metad.WARNING` ，并在其中分别添加日志轮转规则。

最后更新: 2025年8月4日

7. 监控

7.1 查询 NebulaGraph 监控指标

NebulaGraph 支持多种方式查询服务的监控指标，本文将介绍最基础的方式，即通过 HTTP 端口查询。

7.1.1 监控指标结构说明

NebulaGraph 的每个监控指标都由三个部分组成，中间用英文句号（.）隔开，例如 `num_queries.sum.600`。不同的 NebulaGraph 服务支持查询的监控指标也不同。指标结构的说明如下。

类别	示例	说明
指标名称	`num_queries`	简单描述指标的含义。
统计类型	`sum`	指标统计的方法。当前支持 SUM、AVG、RATE 和 P 分位数（P75、P95、P99、P999）。
统计时间	`600`	指标统计的时间范围，当前支持 5 秒、60 秒、600 秒和 3600 秒，分别表示最近 5 秒、最近 1 分钟、最近 10 分钟和最近 1 小时。

7.1.2 通过 HTTP 端口查询监控指标

语法

```
curl -G "http://<host>:<port>/stats?stats=<metric_name_list> [&format=json]"
```

选项	说明
`host`	服务器的 IP 或主机名，可以在安装目录内查看配置文件获取。
`port`	服务器的 HTTP 端口，可以在安装目录内查看配置文件获取。默认情况下，Meta 服务端口为 19559，Graph 服务端口为 19669，Storage 服务端口为 19779。
`metric_name_list`	监控指标名称，多个监控指标用英文逗号（,）隔开。
`&format=json`	将结果以 JSON 格式返回。

> Note
>
> 如果 NebulaGraph 服务部署在容器中，需要执行 `docker-compose ps` 命令查看映射到容器外部的端口，然后通过该端口查询。

查询单个监控指标

查询 Graph 服务中，最近 10 分钟的请求总数。

```
$ curl -G "http://192.168.8.40:19669/stats?stats=num_queries.sum.600"
num_queries.sum.600=400
```

查询多个监控指标

查询 Meta 服务中，最近 1 分钟的心跳平均延迟和最近 10 分钟 P99 心跳（1%最慢的心跳）的平均延迟。

```
$ curl -G "http://192.168.8.40:19559/stats?stats=heartbeat_latency_us.avg.60,heartbeat_latency_us.p99.600"
heartbeat_latency_us.avg.60=281
heartbeat_latency_us.p99.600=985
```

查询监控指标并以 JSON 格式返回

查询 Storage 服务中，最近 10 分钟新增的点数量，并以 JSON 格式返回结果。

```
$ curl -G "http://192.168.8.40:19779/stats?stats=num_add_vertices.sum.600&format=json"
[{"value":1,"name":"num_add_vertices.sum.600"}]
```

查询服务器的所有监控指标

不指定查询某个监控指标时，会返回该服务器上所有的监控指标。

```
$ curl -G "http://192.168.8.40:19559/stats"
heartbeat_latency_us.avg.5=304
heartbeat_latency_us.avg.60=308
heartbeat_latency_us.avg.600=299
heartbeat_latency_us.avg.3600=285
heartbeat_latency_us.p75.5=652
heartbeat_latency_us.p75.60=669
heartbeat_latency_us.p75.600=651
heartbeat_latency_us.p75.3600=642
heartbeat_latency_us.p95.5=930
heartbeat_latency_us.p95.60=963
heartbeat_latency_us.p95.600=933
heartbeat_latency_us.p95.3600=929
heartbeat_latency_us.p99.5=986
heartbeat_latency_us.p99.60=1409
heartbeat_latency_us.p99.600=989
heartbeat_latency_us.p99.3600=986
num_heartbeats.rate.5=0
num_heartbeats.rate.60=0
num_heartbeats.rate.600=0
num_heartbeats.rate.3600=0
num_heartbeats.sum.5=2
num_heartbeats.sum.60=40
num_heartbeats.sum.600=394
num_heartbeats.sum.3600=2364
...
```

查询图空间监控指标

Graph 服务支持一系列基于图空间的监控指标，对不同图空间的数据分别记录。

图空间指标只能通过查询所有监控指标的形式查询到，例如 `curl -G "http://192.168.8.40:19559/stats"`，返回结果中以 `{space=space_name}` 的形式包含图空间名称，例如 `num_active_queries{space=basketballplayer}.sum.5=0`。

> **Caution**
>
> 如需开启图空间监控指标，先在 Graph 服务的配置文件中将 `enable_space_level_metrics` 参数的值修改为 `true`，再启动 NebulaGraph。修改配置的详细方式参见配置管理。

7.1.3 监控指标说明

Graph

参数	说明
`num_active_queries`	活跃的查询语句数的变化数。 计算公式：时间范围内开始执行的语句数减去执行完毕的语句数。
`num_active_sessions`	活跃的会话数的变化数。 计算公式：时间范围内登录的会话数减去登出的会话数。 例如查询num_active_sessions.sum.5，过去 5 秒中登录了 10 个会话数，登出了 30 个会话数，那么该指标值就是 `-20`（10-30）。
`num_aggregate_executors`	聚合（Aggregate）算子执行时间。
`num_auth_failed_sessions_bad_username_password`	因用户名密码错误导验证失败的会话数量。
`num_auth_failed_sessions_out_of_max_allowed`	因为超过 `FLAG_OUT_OF_MAX_ALLOWED_CONNECTIONS` 参数导致的验证登录的失败的会话数量。
`num_auth_failed_sessions`	登录验证失败的会话数量。
`num_indexscan_executors`	索引扫描（IndexScan）算子执行时间。
`num_killed_queries`	被终止的查询数量。
`num_opened_sessions`	服务端建立过的会话数量。
`num_queries`	查询次数。
`num_query_errors_leader_changes`	因 Leader 变更而导致的查询错误的次数。
`num_query_errors`	查询错误次数。
`num_reclaimed_expired_sessions`	服务端主动回收的过期的会话数量。
`num_rpc_sent_to_metad_failed`	Graphd 服务发给 Metad 的 RPC 请求失败的数量。
`num_rpc_sent_to_metad`	Graphd 服务发给 Metad 服务的 RPC 请求数量。
`num_rpc_sent_to_storaged_failed`	Graphd 服务发给 Storaged 服务的 RPC 请求失败的数量。
`num_rpc_sent_to_storaged`	Graphd 服务发给 Storaged 服务的 RPC 请求数量。
`num_sentences`	Graphd 服务接收的语句数。
`num_slow_queries`	慢查询次数。
`num_sort_executors`	排序（Sort）算子执行次数。
`optimizer_latency_us`	优化器阶段延迟时间。
`query_latency_us`	查询延迟时间。
`slow_query_latency_us`	慢查询延迟时间。
`num_queries_hit_memory_watermark`	达到内存水位线的语句的数量。
`resp_part_completeness`	部分成功的完整性。需要在 Graph 配置中设置 `accept_partial_success` 为 `true` 。

Meta

参数	说明
`commit_log_latency_us`	Raft 协议中 Commit 日志的延迟时间。
`commit_snapshot_latency_us`	Raft 协议中 Commit 快照的延迟时间。
`heartbeat_latency_us`	心跳延迟时间。
`num_heartbeats`	心跳次数。
`num_raft_votes`	Raft 协议中投票的次数。
`transfer_leader_latency_us`	Raft 协议中转移 Leader 的延迟时间。
`num_agent_heartbeats`	AgentHBProcessor 心跳次数。
`agent_heartbeat_latency_us`	AgentHBProcessor 延迟时间。
`replicate_log_latency_us`	Raft 复制日志至大多数节点的延迟。
`num_send_snapshot`	Raft 发送快照至其他节点的次数。
`append_log_latency_us`	Raft 复制日志到单个节点的延迟。
`append_wal_latency_us`	Raft 写入单条 WAL 的延迟。
`num_grant_votes`	Raft 投票给其他节点的次数。
`num_start_elect`	Raft 发起投票的次数。

Storage

参数	说明
`add_edges_latency_us`	添加边的延迟时间。
`add_vertices_latency_us`	添加点的延迟时间。
`commit_log_latency_us`	Raft 协议中 Commit 日志的延迟时间。
`commit_snapshot_latency_us`	Raft 协议中 Commit 快照的延迟时间。
`delete_edges_latency_us`	删除边的延迟时间。
`delete_vertices_latency_us`	删除点的延迟时间。
`get_neighbors_latency_us`	查询邻居延迟时间。
`get_dst_by_src_latency_us`	通过起始点获取终点的延迟时间。
`num_get_prop`	GetPropProcessor 执行的次数。
`num_get_neighbors_errors`	GetNeighborsProcessor 执行出错的次数。
`num_get_dst_by_src_errors`	GetDstBySrcProcessor 执行出错的次数。
`get_prop_latency_us`	GetPropProcessor 执行的延迟时间。
`num_edges_deleted`	删除的边数量。
`num_edges_inserted`	插入的边数量。
`num_raft_votes`	Raft 协议中投票的次数。
`num_rpc_sent_to_metad_failed`	Storaged 服务发给 Metad 服务的 RPC 请求失败的数量。
`num_rpc_sent_to_metad`	Storaged 服务发给 Metad 服务的 RPC 请求数量。
`num_tags_deleted`	删除的 Tag 数量。
`num_vertices_deleted`	删除的点数量。
`num_vertices_inserted`	插入的点数量。
`transfer_leader_latency_us`	Raft 协议中转移 Leader 的延迟时间。
`lookup_latency_us`	LookupProcessor 执行的延迟时间。
`num_lookup_errors`	LookupProcessor 执行时出错的次数。
`num_scan_vertex`	ScanVertexProcessor 执行的次数。
`num_scan_vertex_errors`	ScanVertexProcessor 执行时出错的次数。
`update_edge_latency_us`	UpdateEdgeProcessor 执行的延迟时间。
`num_update_vertex`	UpdateVertexProcessor 执行的次数。
`num_update_vertex_errors`	UpdateVertexProcessor 执行时出错的次数。
`kv_get_latency_us`	Getprocessor 的延迟时间。
`kv_put_latency_us`	PutProcessor 的延迟时间。
`kv_remove_latency_us`	RemoveProcessor 的延迟时间。
`num_kv_get_errors`	GetProcessor 执行出错次数。
`num_kv_get`	GetProcessor 执行次数。
`num_kv_put_errors`	PutProcessor 执行出错次数。
`num_kv_put`	PutProcessor 执行次数。

参数	说明
`num_kv_remove_errors`	RemoveProcessor 执行出错次数。
`num_kv_remove`	RemoveProcessor 执行次数。
`forward_tranx_latency_us`	传输延迟时间。
`scan_edge_latency_us`	ScanEdgeProcessor 执行的延迟时间。
`num_scan_edge_errors`	ScanEdgeProcessor 执行时出错的次数。
`num_scan_edge`	ScanEdgeProcessor 执行的次数。
`scan_vertex_latency_us`	ScanVertexProcessor 执行的延迟时间。
`num_add_edges`	添加边的次数。
`num_add_edges_errors`	添加边时出错的次数。
`num_add_vertices`	添加点的次数。
`num_start_elect`	Raft 发起投票的次数
`num_add_vertices_errors`	添加点时出错的次数。
`num_delete_vertices_errors`	删除点时出错的次数。
`append_log_latency_us`	Raft 复制日志到单个节点的延迟。
`num_grant_votes`	Raft 投票给其他节点的次数。
`replicate_log_latency_us`	Raft 复制日志到大多数节点的延迟。
`num_delete_tags`	删除 Tag 的次数。
`num_delete_tags_errors`	删除 Tag 时出错的次数。
`num_delete_edges`	删除边的次数。
`num_delete_edges_errors`	删除边时出错的次数。
`num_send_snapshot`	发送快照的次数。
`update_vertex_latency_us`	UpdateVertexProcessor 执行的延迟时间。
`append_wal_latency_us`	Raft 写入单条 WAL 的延迟。
`num_update_edge`	UpdateEdgeProcessor 执行的次数。
`delete_tags_latency_us`	删除 Tag 的延迟时间。
`num_update_edge_errors`	UpdateEdgeProcessor 执行时出错的次数。
`num_get_neighbors`	GetNeighborsProcessor 执行的次数。
`num_get_dst_by_src`	GetDstBySrcProcessor 执行的次数。
`num_get_prop_errors`	GetPropProcessor 执行时出错的次数。
`num_delete_vertices`	删除点的次数。
`num_lookup`	LookupProcessor 执行的次数。
`num_sync_data`	Storage 同步 Drainer 数据的次数。
`num_sync_data_errors`	Storage 同步 Drainer 数据出错的次数。
`sync_data_latency_us`	Storage 同步 Drainer 数据的延迟时间。

图空间级别监控指标

> Note
>
> 图空间级别监控指标是动态创建的，只有当图空间内触发该行为时，对应的指标才会创建，用户才能查询到。

参数	说明
`num_active_queries`	当前正在执行的查询数。
`num_queries`	查询次数。
`num_sentences`	Graphd 服务接收的语句数。
`optimizer_latency_us`	优化器阶段延迟时间。
`query_latency_us`	查询延迟时间。
`num_slow_queries`	慢查询次数。
`num_query_errors`	查询报错语句数量。
`num_query_errors_leader_changes`	因 Leader 变更而导致的查询错误的次数。
`num_killed_queries`	被终止的查询数量。
`num_aggregate_executors`	聚合（Aggregate）算子执行时间。
`num_sort_executors`	排序（Sort）算子执行次数。
`num_indexscan_executors`	索引扫描（IndexScan）算子执行时间。
`num_auth_failed_sessions_bad_username_password`	因用户名密码错误导验证失败的会话数量。
`num_auth_failed_sessions`	登录验证失败的会话数量。
`num_opened_sessions`	服务端建立过的会话数量。
`num_queries_hit_memory_watermark`	达到内存水位线的语句的数量。
`num_reclaimed_expired_sessions`	服务端主动回收的过期的会话数量。
`num_rpc_sent_to_metad_failed`	Graphd 服务发给 Metad 的 RPC 请求失败的数量。
`num_rpc_sent_to_metad`	Graphd 服务发给 Metad 服务的 RPC 请求数量。
`num_rpc_sent_to_storaged_failed`	Graphd 服务发给 Storaged 服务的 RPC 请求失败的数量。
`num_rpc_sent_to_storaged`	Graphd 服务发给 Storaged 服务的 RPC 请求数量。
`slow_query_latency_us`	慢查询延迟时间。

最后更新: 2025年8月4日

7.2 RocksDB 统计数据

NebulaGraph 使用 RocksDB 作为底层存储，本文介绍如何收集和展示 NebulaGraph 的 RocksDB 统计信息。

7.2.1 启用 RocksDB

RocksDB 统计功能默认关闭，启动 RocksDB 统计功能，你需要：

1. 修改 `nebula-storaged.conf` 文件中 `--enable_rocksdb_statistics` 参数为 `true`。配置默认文件目录为 `/use/local/nebula/etc`。
2. 重启服务使修改生效。

7.2.2 获取 RocksDB 统计信息

用户可以使用存储服务中的内置 HTTP 服务来获取以下类型的统计信息，且支持返回 JSON 格式的结果：

- 所有统计信息。
- 指定条目的信息。

7.2.3 示例

使用以下命令获取所有 RocksDB 统计信息：

```
curl -L "http://${storage_ip}:${port}/rocksdb_stats"
```

例如：

```
curl -L "http://172.28.2.1:19779/rocksdb_stats"

rocksdb.blobdb.blob.file.bytes.read=0
rocksdb.blobdb.blob.file.bytes.written=0
rocksdb.blobdb.blob.file.bytes.synced=0
...
```

使用以下命令获取部分 RocksDB 统计信息：

```
curl -L "http://${storage_ip}:${port}/rocksdb_stats?stats=${stats_name}"
```

例如使用以下语句获取 `rocksdb.bytes.read` 和 `rocksdb.block.cache.add` 的信息。

```
curl -L "http://172.28.2.1:19779/rocksdb_stats?stats=rocksdb.bytes.read,rocksdb.block.cache.add"

rocksdb.block.cache.add=14
rocksdb.bytes.read=1632
```

使用以下命令获取部分 JSON 格式的 RocksDB 统计信息：

```
curl -L "http://${storage_ip}:${port}/rocksdb_stats?stats=${stats_name}&format=json"
```

例如使用以下语句获取 `rocksdb.bytes.read` 和 `rocksdb.block.cache.add` 的统计信息并返回 JSON 的格式数据。

```
curl -L "http://172.28.2.1:19779/rocksdb_stats?stats=rocksdb.bytes.read,rocksdb.block.cache.add&format=json"

[
  {
    "rocksdb.block.cache.add": 1
  },
  {
    "rocksdb.bytes.read": 160
  }
]
```

最后更新: 2025年8月4日

8. 数据安全

8.1 验证和授权

8.1.1 身份验证

身份验证用于将会话映射到特定用户，从而实现访问控制。

当客户端连接到 NebulaGraph 时，NebulaGraph 会创建一个会话，会话中存储连接的各种信息，如果开启了身份验证，就会将会话映射到对应的用户。

本地身份验证

本地身份验证是指在服务器本地存储用户名、加密密码，当用户尝试访问 NebulaGraph 时，将进行身份验证。

启用本地身份验证

1. 编辑配置文件 `nebula-graphd.conf` （默认目录为 `/usr/local/nebula/etc/` ），设置如下参数：

- `--enable_authorize` ：是否启用身份验证，可选值： `true` 、 `false` 。

> Note
>
> - 默认情况下，身份验证功能是关闭的，用户可以使用 `root` 用户名和任意密码连接到 NebulaGraph 。
> - 开启身份验证后，默认的 God 角色账号为 `root` ，密码为 `nebula` 。角色详情请参见内置角色权限。

- `--failed_login_attempts` ：可选项，需要手动添加该参数。单个 Graph 节点允许连续输入错误密码的次数。超过该次数时，账户会被锁定。如果有多个 Graph 节点，允许的次数为 `节点数 * 次数` 。
- `--password_lock_time_in_secs` ：可选项，需要手动添加该参数。多次输入错误密码后，账户被锁定的时间。单位：秒。

2. 重启 NebulaGraph 服务。

最后更新: 2025年8月4日

8.1.2 用户管理

用户管理是 NebulaGraph 访问控制中不可或缺的组成部分，本文将介绍用户管理的相关语法。

开启身份验证后，用户需要使用已创建的用户才能连接 NebulaGraph，而且连接后可以进行的操作也取决于该用户拥有的角色权限。

> Note
>
> - 默认情况下，身份验证功能是关闭的，用户可以使用 root 用户名和任意密码连接到 NebulaGraph 。
> - 修改权限后，对应的用户需要重新登录才能生效。

创建用户（`CREATE USER`）

执行 `CREATE USER` 语句可以创建新的 NebulaGraph 用户。当前仅 God 角色用户（即 `root` 用户）能够执行 `CREATE USER` 语句。

- 语法

```
CREATE USER [IF NOT EXISTS] <user_name> [WITH PASSWORD '<password>'];
```

- `IF NOT EXISTS`：检测待创建的用户名是否存在，只有不存在时，才会创建新用户。
- `user_name`：待创建的用户名。长度最多为 16 个字符。
- `password`：用户名对应的密码。默认密码为空字符串（`''`）。长度最多为 24 个字符。

- 示例

```
nebula> CREATE USER user1 WITH PASSWORD 'nebula';
nebula> SHOW USERS;
+---------+
| Account |
+---------+
| "root"  |
| "user1" |
+---------+
```

授权用户（`GRANT ROLE`）

执行 `GRANT ROLE` 语句可以将指定图空间的内置角色权限授予用户。当前仅 God 角色用户和 Admin 角色用户能够执行 `GRANT ROLE` 语句。角色权限的说明，请参见内置角色权限。

- 语法

```
GRANT ROLE <role_type> ON <space_name> TO <user_name>;
```

- 示例

```
nebula> GRANT ROLE USER ON basketballplayer TO user1;
```

撤销用户权限（REVOKE ROLE）

执行 `REVOKE ROLE` 语句可以撤销用户的指定图空间的内置角色权限。当前仅 God 角色用户和 Admin 角色用户能够执行 `REVOKE ROLE` 语句。角色权限的说明，请参见内置角色权限。

- 语法

```
REVOKE ROLE <role_type> ON <space_name> FROM <user_name>;
```

- 示例

```
nebula> REVOKE ROLE USER ON basketballplayer FROM user1;
```

查看指定用户权限（DESCRIBE USER）

执行 `DESCRIBE USER` 语句可以查看指定用户的角色权限信息。

- 语法

```
DESCRIBE USER <user_name>;
DESC USER <user_name>;
```

- 示例

```
nebula> DESCRIBE USER user1;
+---------+--------------------+
| role    | space              |
+---------+--------------------+
| "ADMIN" | "basketballplayer" |
+---------+--------------------+
```

查看指定空间内用户权限（SHOW ROLES）

执行 `SHOW ROLES` 语句可以查看指定空间内的所有用户（除 `root` 以外）和对应角色权限的信息。

- 语法

```
SHOW ROLES IN <space_name>;
```

- 示例

```
nebula> SHOW ROLES IN basketballplayer;
+---------+-----------+
| Account | Role Type |
+---------+-----------+
| "user1" | "ADMIN"   |
+---------+-----------+
```

修改用户密码（CHANGE PASSWORD）

执行 `CHANGE PASSWORD` 语句可以修改用户密码，修改时需要提供旧密码和新密码。

- 语法

```
CHANGE PASSWORD <user_name> FROM '<old_password>' TO '<new_password>';
```

- 示例

```
nebula> CHANGE PASSWORD user1 FROM 'nebula' TO 'nebula123';
```

修改用户密码（`ALTER USER`）

执行 `ALTER USER` 语句可以修改用户密码，修改时不需要提供旧密码。当前仅 God 角色用户（即 `root` 用户）能够执行 `ALTER USER` 语句。

- 语法

```
ALTER USER <user_name> WITH PASSWORD '<password>';
```

- 示例

```
nebula> ALTER USER user2 WITH PASSWORD 'change_password';
nebula> SHOW USERS;
+---------+
| Account |
+---------+
| "root"  |
| "user1" |
| "user2" |
+---------+
```

删除用户（`DROP USER`）

执行 `DROP USER` 语句可以删除用户。当前仅 God 角色用户能够执行 `DROP USER` 语句。

> Note
>
> 删除用户不会自动断开该用户当前会话，而且权限仍在当前会话中生效。

- 语法

```
DROP USER [IF EXISTS] <user_name>;
```

- 示例

```
nebula> DROP USER user1;
```

查看用户列表（`SHOW USERS`）

执行 `SHOW USERS` 语句可以查看用户列表。当前仅 God 角色用户能够执行 `SHOW USERS` 语句。

- 语法

```
SHOW USERS;
```

- 示例

```
nebula> SHOW USERS;
+---------+-----------------+
| Account | IP Whitelist    |
+---------+-----------------+
| "root"  | ""              |
| "user1" | ""              |
| "user2" | "192.168.10.10" |
+---------+-----------------+
```

最后更新: 2025年8月4日

8.1.3 内置角色权限

所谓角色，就是一组相关权限的集合。用户可以把角色分配给创建的用户，从而实现访问控制。

内置角色

NebulaGraph 内置了多种角色，说明如下：

- God
- 初始最高权限角色，拥有所有操作的权限。类似于 Linux 中的 `root` 和 Windows 中的 `administrator`。
- Meta 服务初始化时，会自动创建 God 角色用户 `root`，密码为 `nebula`。

> Caution
>
> 请及时修改 `root` 用户的密码，保证数据安全。

- 在 `nebula-graphd.conf` 文件中（默认目录为 `/usr/local/nebula/etc/`），当 `--enable_authorize` 为 `true` 时：
- 一个集群只能有一个 God 角色用户，该用户可以管理集群内所有图空间。
- 不支持手动授权 God 角色，只能使用默认 God 角色用户 `root`。

- Admin
- 对权限内的图空间拥有 Schema 和 data 的读写权限。
- 可以将权限内的图空间授权给其他用户。

> Caution
>
> 只能授权低于 ADMIN 级别的角色给其他用户。

- DBA
- 对权限内的图空间拥有 Schema 和 data 的读写权限。
- 无法将权限内的图空间授权给其他用户。

- User
- 对权限内的图空间拥有 Schema 的只读权限。
- 对权限内的图空间拥有 data 的读写权限。

- Guest
- 对权限内的图空间拥有 Schema 和 data 的只读权限。

> Note
>
> - 不支持自行创建角色，只能使用默认的内置角色。
> - 一个用户在一个图空间内只能拥有一个角色权限。授权用户请参见用户管理。

角色权限

各角色的执行权限如下。

权限	God	Admin	DBA	User	Guest	相关语句
Read space	Y	Y	Y	Y	Y	`USE`、`DESCRIBE SPACE`
Read schema	Y	Y	Y	Y	Y	`DESCRIBE TAG`、`DESCRIBE EDGE`、`DESCRIBE TAG INDEX`、`DESCRIBE EDGE INDEX`
Write schema	Y	Y	Y			`CREATE TAG`、`ALTER TAG`、`CREATE EDGE`、`ALTER EDGE`、`DROP TAG`、`DELETE TAG`、`DROP EDGE`、`CREATE TAG INDEX`、`CREATE EDGE INDEX`、`DROP TAG INDEX`、`DROP EDGE INDEX`
Write user	Y					`CREATE USER`、`DROP USER`、`ALTER USER`
Write role	Y	Y				`GRANT`、`REVOKE`
Read data	Y	Y	Y	Y	Y	`GO`、`SET`、`PIPE`、`MATCH`、`ASSIGNMENT`、`LOOKUP`、`YIELD`、`ORDER BY`、`FETCH VERTICES`、`Find`、`FETCH EDGES`、`FIND PATH`、`LIMIT`、`GROUP BY`、`RETURN`
Write data	Y	Y	Y	Y		`INSERT VERTEX`、`UPDATE VERTEX`、`INSERT EDGE`、`UPDATE EDGE`、`DELETE VERTEX`、`DELETE EDGES`、`DELETE TAG`
Show operations	Y	Y	Y	Y	Y	`SHOW`、`CHANGE PASSWORD`
Job	Y	Y	Y	Y		`SUBMIT JOB COMPACT`、`SUBMIT JOB FLUSH`、`SUBMIT JOB STATS`、`STOP JOB`、`RECOVER JOB`、`BUILD TAG INDEX`、`BUILD EDGE INDEX`、`INGEST`、`DOWNLOAD`
Write space	Y					`CREATE SPACE`、`DROP SPACE`、`CREATE SNAPSHOT`、`DROP SNAPSHOT`、`BALANCE`、`CONFIG`

Caution

Show operations 为特殊操作，只会在自身权限内执行。例如 `SHOW SPACES`，每个角色都可以执行，但是只会返回自身权限内的图空间。 只有 God 角色可以执行 `SHOW USERS` 和 `SHOW SNAPSHOTS` 语句。

最后更新: 2025年8月4日

8.2 SSL 加密

NebulaGraph 支持在客户端、Graph 服务、Meta 服务和 Storage 服务之间进行 SSL 加密传输，本文介绍如何设置 SSL 加密。

8.2.1 注意事项

开启 SSL 加密会轻微影响性能。

8.2.2 参数介绍

参数	默认值	说明
`cert_path`	-	PEM 证书的路径。
`key_path`	-	密钥证书的路径。
`password_path`	-	密码文件证书的路径。
`ca_path`	-	受信任 CA 证书文件的路径。
`enable_ssl`	`false`	是否开启 SSL 加密。
`enable_graph_ssl`	`false`	是否仅在 Graph 服务上开启 SSL 加密。
`enable_meta_ssl`	`false`	是否仅在 Meta 服务上开启 SSL 加密。

8.2.3 证书模式

为了使用 SSL 加密，必须有 SSL 证书。 NebulaGraph 支持两种证书模式：

- 自签名证书模式

  需要自行制作签名证书。需要根据加密策略，在对应的配置文件内设置 `cert_path` 、 `key_path` 和 `password_path` 。

- CA 签名证书模式

  需要在认证机构（Certificate Authority）申请签名证书。需要根据加密策略，在对应的配置文件内设置 `cert_path` 、 `key_path` 和 `ca_path` 。

8.2.4 加密策略

NebulaGraph 支持三种加密策略。

- 对客户端、Graph 服务、Meta 服务和 Storage 服务之间的传输数据加密。

  需要修改 `nebula-graphd.conf` 、 `nebula-metad.conf` 和 `nebula-storaged.conf` 配置文件，设置 `enable_ssl = true` 。

- 对客户端和 Graph 服务之间的传输数据加密。

  适用于集群设置在同一个机房内，仅对外开放 Graph 服务的端口。因为其他服务可以通过内部网络通信，无需加密。需要修改 `nebula-graphd.conf` 配置文件，设置 `enable_graph_ssl = true` 。

- 对集群中 Meta 服务相关的传输数据加密。

  适用于向 meta 服务传输需保密的信息。需要修改 `nebula-graphd.conf` 、 `nebula-metad.conf` 和 `nebula-storaged.conf` 配置文件，设置 `enable_meta_ssl = true` 。

8.2.5 使用方式

1. 确认证书模式和加密策略。
2. 在对应的配置文件内增加证书配置和策略配置。

   例如使用自签名证书，并对客户端、Graph 服务、Meta 服务和 Storage 服务之间的数据传输进行加密。需要对三个配置文件都进行如下设置：

   ```
   --cert_path=xxxxxx
   --key_path=xxxxx
   --password_path=xxxxxx
   --enable_ssl=true
   ```

3. 客户端设置安全套接字并添加受信任的 CA。示例代码请参见 nebula-test-run.py。

最后更新: 2025年8月4日

9. 备份与恢复

9.1 NebulaGraph BR（社区版）

9.1.1 什么是 Backup&Restore

Backup&Restore（简称 BR）是一款命令行界面（CLI）工具，可以帮助备份 NebulaGraph 的图空间数据，或者通过备份文件恢复数据。

功能

- 一键操作备份和恢复数据。
- 支持基于以下备份文件恢复数据：
- 本地磁盘（SSD 或 HDD），建议仅在测试环境使用。
- 兼容亚马逊对象存储（Amazon S3）云存储服务接口，例如：阿里云对象存储（Alibaba Cloud OSS）、MinIO、Ceph RGW 等。
- 支持备份并恢复整个 NebulaGraph 集群。
- （实验性功能）支持备份指定图空间数据。

限制

- NebulaGraph 版本需要为 v3.x。
- 数据备份仅支持全量备份，不支持增量备份。
- Listener 暂时不支持备份，且全文索引也不支持备份。
- 如果备份数据到本地磁盘，备份的文件将会放置在每个服务器的本地路径中。也可以在本地挂载 NFS 文件系统，以便将备份数据还原到不同的主机上。
- 备份图空间只能恢复到原集群，不能跨集群恢复，并且集群的 Storage 主机数量及 IP 需一致。还原指定图空间时将清除集群中其余所有图空间。
- 数据备份过程中，所有图空间中的 DDL 和 DML 语句将会阻塞，我们建议在业务低峰期进行操作，例如凌晨 2 点至 5 点。
- 数据恢复期间有一段时间服务会被停止。
- 不支持在容器部署的 NebulaGraph 集群中使用 BR。

如何使用 **BR**

可以按照如下步骤使用 BR：

1. 安装 BR
2. 使用 BR 备份数据
3. 使用 BR 恢复数据

视频

- NebulaGraph 容灾备份工具 nebula-br 介绍（3 分 34 秒）

最后更新: 2025年8月4日

9.1.2 安装 BR

本文介绍裸机部署情况下的 BR 安装。

安装说明

使用 BR 工具备份和恢复 NebulaGraph 时，需要安装 NebulaGraph Agent 服务。Agent 是集群中每台机器的一个守护进程，用于启停 NebulaGraph 服务和上传、下载备份文件。BR 工具和 Agent 插件的安装方式如下文。

版本兼容性

NebulaGraph	BR 社区版	Agent
3.5.x ~ 3.6.0	3.6.0	3.6.x ~ 3.7.0
3.3.0 ~ 3.4.x	3.3.0	0.2.0 ~ 3.4.0
3.0.x ~ 3.2.x	0.6.1	0.1.0 ~ 0.2.0

安装 BR

使用二进制文件安装

1. 下载 BR。

```
wget https://github.com/vesoft-inc/nebula-br/releases/download/v3.6.0/br-3.6.0-linux-amd64
```

2. 修改文件名称为 `br` 。

```
sudo mv br-3.6.0-linux-amd64 br
```

3. 授予 BR 执行权限。

```
sudo chmod +x br
```

4. 执行 `./br version` 查看 BR 版本。

```
[nebula-br]$ ./br version
Nebula Backup And Restore Utility Tool,V-3.6.0
```

使用源码安装

使用源码安装 BR 前，准备工作如下：

- 安装 Go 1.14.x 或更新版本。
- 安装 make。

1. 克隆 `nebula-br` 库至机器。

```
git clone https://github.com/vesoft-inc/nebula-br.git
```

2. 进入 `br` 目录。

```
cd nebula-br
```

3. 编译 BR。

```
make
```

用户可以在命令行输入 `bin/br version` ，如果返回以下内容，则认为编译成功。

```
[nebula-br]$ bin/br version
NebulaGraph Backup And Restore Utility Tool,V-3.6.0
```

安装 Agent

NebulaGraph Agent 以二进制形式存在各个机器的安装目录中，通过 RPC 协议对 BR 工具提供服务。

在每台机器中，进行以下操作：

1. 下载 Agent。

```
wget https://github.com/vesoft-inc/nebula-agent/releases/download/v3.7.0/agent-3.7.0-linux-amd64
```

2. 修改 Agent 的名称为 `agent` 。

```
sudo mv agent-3.7.0-linux-amd64 agent
```

3. 授予 Agent 可执行权限。

```
sudo chmod +x agent
```

4. 执行以下命令启动 Agent。

> Note
>
> 启动 Agent 前，确保已经启动集群中的 Meta 服务，并且 Agent 有对应 NebulaGraph 服务目录和备份目录的读写权限。

```
sudo nohup ./agent --agent="<agent_node_ip>:8888" --meta="<metad_node_ip>:9559" > nebula_agent.log 2>&1 &
```

- `--agent`：Agent 所在机器的 IP 地址和访问端口。
- `--meta`：集群中任一 Meta 服务所在机器的 IP 地址和访问端口。
- `--ratelimit`：可选项，限制文件上传和下载的速度，防止带宽被占满导致其他服务不可用。单位：Bytes。

例如：

```
sudo nohup ./agent --agent="192.168.8.129:8888" --meta="192.168.8.129:9559" --ratelimit=1048576 > nebula_agent.log 2>&1 &
```

> Caution
>
> `--agent` 配置的 IP 地址需要和配置文件中 Meta 和 Storage 服务的地址格式保持一致，即都使用真实 IP 地址，否则 Agent 无法启动。

5. 连接服务并查看 Agent 的运行状态。

```
nebula> SHOW HOSTS AGENT;
+-----------------+------+----------+---------+-------------+---------+
| Host            | Port | Status   | Role    | Git Info Sha | Version |
+-----------------+------+----------+---------+-------------+---------+
| "192.168.8.129" | 8888 | "ONLINE" | "AGENT" | "96646b8"   |         |
+-----------------+------+----------+---------+-------------+---------+
```

常见问题

报错 `E_LIST_CLUSTER_NO_AGENT_FAILURE`

如果操作过程中遇见 `E_LIST_CLUSTER_NO_AGENT_FAILURE` 错误，可能是由于 Agent 服务没有启动或者 Agent 服务没有被注册至 Meta 服务。首先执行 `SHOW HOSTS AGENT` 查看集群上所有节点的 Agent 服务的状态，当时状态显示为 `OFFLINE` 时表示注册 Agent 失败， 然后检查启动 Agent 服务的命令中的 `--meta` 参数是否正确。

最后更新: 2025年8月4日

9.1.3 使用 BR 备份数据

成功安装 BR 工具后，可以备份整个图空间的数据，本文介绍如何使用 BR 备份数据。

准备工作

- 安装 BR 和 Agent 并在集群中的每个主机上运行 Agent。
- 确认 NebulaGraph 服务正在运行中。
- 如果在本地保存备份文件，需要在 Meta 服务器、Storage 服务器和 BR 机器上创建绝对路径相同的备份目录，并记录绝对路径，同时需要保证账号对该目录有写权限。

  > Warning
  >
  > 在生产环境中，我们建议用户将 NFS (Network File System）存储设备挂载到 Meta 服务器、Storage 服务器和 BR 机器上进行本地备份，或者使用 Alibaba Cloud OSS、Amazon S3 进行远程备份。否则当需要通过本地文件恢复数据时，必须手动将这些备份文件移动到指定目录，会导致数据冗余和某些问题。更多信息，请参见使用 BR 恢复数据。

操作步骤

在 BR 工具的安装路径下（编译安装的默认路径为 `./bin/br`）运行以下命令对整个集群进行全量备份操作。

> Note
>
> 确保备份文件的路径存在。

```
$ ./br backup full --meta <ip_address> --storage <storage_path>
```

例如：

- 运行以下命令对 meta 服务的地址为 `192.168.8.129:9559` 的整个集群进行全量备份操作，并将备份文件保存到本地的 `/home/nebula/backup/` 路径下。

  > Caution
  >
  > 如果有多个 metad 地址，可以使用其中任意一个。

  > Caution
  >
  > 备份至本地时，默认只备份 leader metad 的数据，因此当有多个 metad 节点时，需要手动将备份后的 leader metad 的目录（路径为 `<storage_path>/meta`）拷贝并覆盖其他 follower meatd 节点的对应目录。

  ```
  $ ./br backup full --meta "192.168.8.129:9559" --storage "local:///home/nebula/backup/"
  ```

- 运行以下命令对 meta 服务的地址为 `192.168.8.129:9559` 的整个集群进行全量备份操作，并将备份文件保存到兼容 s3 协议的对象存储服务 `br-test` 桶下的 `backup` 中。

  ```
  $ ./br backup full --meta "192.168.8.129:9559" --s3.endpoint "http://192.168.8.129:9000" --storage="s3://br-test/backup/" --s3.access_key=minioadmin --s3.secret_key=minioadmin --s3.region=default
  ```

以下列出命令的相关参数。

参数	数据类型	是否必需	默认值	说明
`-h`,`--help`	-	否	无	查看帮助。
`--debug`	-	否	无	查看更多日志信息。
`--log`	string	否	`"br.log"`	日志路径。
`--meta`	string	是	无	meta 服务的地址和端口号。
`--spaces`	stringArray	否	无	（实验性功能）指定要备份的图空间名字，未指定将备份所有图空间。可指定多个图空间，用法为 `--spaces nba_01 --spaces nba_02`。
`--storage`	string	是	无	BR 备份数据存储位置，格式为：`<Schema>://<PATH>` Schema：可选值为 `local` 和 `s3`。选择 s3 时，需要填写 `s3.access_key`、`s3.endpoint`、`s3.region` 和 `s3.secret_key`。 PATH：存储位置的路径。
`--s3.access_key`	string	否	无	用于标识用户。
`--s3.endpoint`	string	否	无	S3 对外服务的访问域名的 URL，指定 http 或 https。
`--s3.region`	string	否	无	数据中心所在物理位置。
`--s3.secret_key`	string	否	无	用户用于加密签名字符串和用来验证签名字符串的密钥，必须保密。

下一步

备份文件生成后，可以使用 BR 将备份文件的数据恢复到 NebulaGraph 中。具体操作，请参见使用 BR 恢复数据。

最后更新: 2025年8月4日

9.1.4 使用 BR 恢复数据

如果使用 BR 备份了 NebulaGraph 的数据，可以通过备份文件进行数据恢复。本文介绍如何通过备份文件恢复数据。

> Caution
>
> 恢复执行成功后，目标集群上已有的数据会被删除，然后替换为备份文件中的数据。建议提前备份目标集群上的数据。

> Caution
>
> 数据恢复需要离线进行。

准备工作

- 安装 BR 和 Agent 并在集群中的每个主机上运行 Agent。
- 确认没有应用程序连接到待恢复数据的 NebulaGraph 集群。
- 确认集群的拓扑结构一致，即原集群和目标集群的主机数量一致，且每个主机数据文件夹数量分布一致。

操作步骤

在 BR 工具的安装路径下（编译安装的默认路径为 `./bin/br`），完成以下操作。

1. 用户可以使用以下命令列出现有备份信息：

```
$ ./br show --storage <storage_path>
```

例如，可以使用以下命令列出在本地 `/home/nebula/backup` 路径中的备份的信息。

```
$ ./br show --storage "local:///home/nebula/backup"
+----------------------------+---------------------+------------------------+-------------+------------+
|            NAME            |     CREATE TIME     |         SPACES         | FULL BACKUP | ALL SPACES |
+----------------------------+---------------------+------------------------+-------------+------------+
| BACKUP_2022_02_10_07_40_41 | 2022-02-10 07:40:41 | basketballplayer       | true        | true       |
| BACKUP_2022_02_11_08_26_43 | 2022-02-11 08:26:47 | basketballplayer,foesa | true        | true       |
+----------------------------+---------------------+------------------------+-------------+------------+
```

或使用以下命令列出在兼容 s3 协议的对象存储服务 `br-test` 桶下的 `backup` 中的备份的信息。

```
$ ./br show --s3.endpoint "http://192.168.8.129:9000" --storage="s3://br-test/backup/" --s3.access_key=minioadmin --s3.secret_key=minioadmin --s3.region=default
```

以下列出命令的相关参数。

参数	数据类型	是否必需	默认值	说明
`-h,-help`	-	否	无	查看帮助。
`--debug`	-	否	无	查看更多日志信息。
`--log`	string	否	"br.log"	日志路径。
`--storage`	string	是	无	BR 备份数据存储位置，格式为：<Schema>://<PATH> Schema：可选值为 `local` 和 `s3`。选择 s3 时，需要填写 `s3.access_key`、`s3.endpoint`、`s3.region` 和 `s3.secret_key`。 PATH：存储位置的路径。
`--s3.access_key`	string	否	无	用于标识用户。
`--s3.endpoint`	string	否	无	S3 对外服务的访问域名的 URL，指定 http 或 https。
`--s3.region`	string	否	无	数据中心所在物理位置。
`--s3.secret_key`	string	否	无	用户用于加密签名字符串和用来验证签名字符串的密钥，必须保密。

2. 用户可以使用以下命令恢复数据：

```
$ ./br restore full --meta <ip_address> --storage <storage_path> --name <backup_name>
```

例如，可以使用以下命令，将本地 `/home/nebula/backup/` 路径中的数据恢复到为 meta 地址为 `192.168.8.129:9559` 集群中：

```
$ ./br restore full --meta "192.168.8.129:9559" --storage "local:///home/nebula/backup/" --name BACKUP_2021_12_08_18_38_08
```

或者使用以下命令，将兼容 s3 协议的对象存储服务 `br-test` 桶下的 `backup` 的备份，恢复到 meta 服务的地址为 `192.168.8.129:9559` 的集群中。

```
$ ./br restore full --meta "192.168.8.129:9559" --s3.endpoint "http://192.168.8.129:9000" --storage="s3://br-test/backup/" --s3.access_key=minioadmin --s3.secret_key=minioadmin --s3.region="default" --name BACKUP_2021_12_08_18_38_08
```

如果返回如下信息，表示数据已经恢复成功。

```
Restore succeed.
```

> **Caution**
>
> 如果用户新集群的IP和备份集群不同，在恢复集群后需要使用 `add host` 向新集群中添加 Storage 主机。

以下列出命令的相关参数。

参数	数据类型	是否必需	默认值	说明
`-h,-help`	-	否	-	查看帮助。
`--debug`	-	否	无	查看更多日志信息。
`--log`	string	否	"br.log"	日志路径。
`--meta`	string	是	无	meta 服务的地址和端口号。
`--name`	string	是	无	备份名字。
`--storage`	string	是	无	BR 备份数据存储位置，格式为：<Schema>://<PATH> Schema：可选值为 `local` 和 `s3`。选择 s3 时，需要填写 `s3.access_key`、`s3.endpoint`、`s3.region` 和 `s3.secret_key`。 PATH：存储位置的路径。
`--s3.access_key`	string	否	无	用于标识用户。
`--s3.endpoint`	string	否	无	S3 对外服务的访问域名的 URL，指定 http 或 https。
`--s3.region`	string	否	无	数据中心所在物理位置。
`--s3.secret_key`	string	否	无	用户用于加密签名字符串和用来验证签名字符串的密钥，必须保密。

3. 如果在备份期间发现任何错误，用户可以使用以下命令清理临时文件。该命令将清理集群和外部存储中的文件，同时用户也可以使用该命令清理外部存储中的旧的备份文件。

```
$ ./br cleanup --meta <ip_address> --storage <storage_path> --name <backup_name>
```

以下列出命令的相关参数。

参数	数据类型	是否必需	默认值	说明
`-h,-help`	-	否	-	查看帮助。
`--debug`	-	否	无	查看更多日志信息。
`--log`	string	否	"br.log"	日志路径。
`--meta`	string	是	无	meta 服务的地址和端口号。
`--name`	string	是	无	备份名字。
`--storage`	string	是	无	BR 备份数据存储位置，格式为：<Schema>://<PATH> Schema：可选值为 `local` 和 `s3`。选择 s3 时，需要填写 `s3.access_key`、`s3.endpoint`、`s3.region` 和 `s3.secret_key`。 PATH：存储位置的路径。
`--s3.access_key`	string	否	无	用于标识用户。
`--s3.endpoint`	string	否	无	S3 对外服务的访问域名的 URL，指定 http 或 https。
`--s3.region`	string	否	无	数据中心所在物理位置。
`--s3.secret_key`	string	否	无	用户用于加密签名字符串和用来验证签名字符串的密钥，必须保密。

最后更新: 2025年8月4日

9.2 管理快照

NebulaGraph 提供快照（snapshot）功能，用于保存集群当前时间点的数据状态，当出现数据丢失或误操作时，可以通过快照恢复数据。

9.2.1 前提条件

NebulaGraph 的身份认证功能默认是关闭的，此时任何用户都能使用快照功能。

如果身份认证开启，仅 God 角色用户可以使用快照功能。关于角色说明，请参见内置角色权限。

9.2.2 注意事项

- 系统结构发生变化后，建议立刻创建快照，例如在 `add host`、`drop host`、`create space`、`drop space`、`balance` 等操作之后。
- 不支持自动回收创建失败的快照垃圾文件，需要手动删除。
- 不支持修改快照保存路径。

9.2.3 创建快照

命令 `CREATE SNAPSHOT` 可以创建集群当前时间点的快照。只支持创建所有图空间的快照，不支持创建指定图空间的快照。

> Note
>
> 如果快照创建失败，请参考后文删除损坏的快照，然后重新创建快照。

```
nebula> CREATE SNAPSHOT;
```

9.2.4 查看快照

命令 `SHOW SNAPSHOTS` 可以查看集群中的所有快照。

```
nebula> SHOW SNAPSHOTS;
+--------------------------------+---------+------------------+
| Name                           | Status  | Hosts            |
+--------------------------------+---------+------------------+
| "SNAPSHOT_2021_03_09_08_43_12" | "VALID" | "127.0.0.1:9779" |
| "SNAPSHOT_2021_03_09_09_10_52" | "VALID" | "127.0.0.1:9779" |
+--------------------------------+---------+------------------+
```

参数说明如下：

参数	说明
`Name`	快照名称，前缀为 `SNAPSHOT`，表示该文件为快照文件，后缀为快照创建的时间点（UTC 时间）。
`Status`	快照状态。`VALID` 表示快照有效，`INVALID` 表示快照无效。
`Hosts`	创建快照时所有 Storage 服务器的 IP（或主机名）和端口。

快照路径

快照保存在 Meta 和 Storage 配置文件中 `data_path` 参数指定的路径中。创建快照时，在 leader Meta 服务和所有 Storage 服务的数据存储路径中会检查是否有 `checkpoints` 目录，如果没有会自动创建。新建的快照以子目录的形式储存在 `checkpoints` 目录内。例如 `SNAPSHOT_2021_03_09_08_43_12`，后缀 `2021_03_09_08_43_12` 根据创建时间（UTC）自动生成。

为了快速定位快照所在路径，可以在数据存储路径内使用 Linux 命令 `find`。例如：

```
$ cd /usr/local/yueshu-graph-master/data
$ find |grep 'SNAPSHOT_2021_03_09_08_43_12'
```

```
./data/meta2/nebula/0/checkpoints/SNAPSHOT_2021_03_09_08_43_12
./data/meta2/nebula/0/checkpoints/SNAPSHOT_2021_03_09_08_43_12/data
./data/meta2/nebula/0/checkpoints/SNAPSHOT_2021_03_09_08_43_12/data/000081.sst
...
```

9.2.5 删除快照

命令 `DROP SNAPSHOT` 可以删除指定的快照，语法为：

```
DROP SNAPSHOT <snapshot_name>;
```

示例如下：

```
nebula> DROP SNAPSHOT SNAPSHOT_2021_03_09_08_43_12;
nebula> SHOW SNAPSHOTS;
+--------------------------------+---------+------------------+
| Name                           | Status  | Hosts            |
+--------------------------------+---------+------------------+
| "SNAPSHOT_2021_03_09_09_10_52" | "VALID" | "127.0.0.1:9779" |
+--------------------------------+---------+------------------+
```

> Note
>
> 删除最后一个快照时，会将 `checkpoints` 目录一起删除。

9.2.6 恢复快照

> Warning
>
> 恢复快照数据时，如果快照中备份的图空间被删除（执行了 `DROP SPACE` 操作），那么该图空间数据将无法恢复。

当前未提供恢复快照命令，需要手动拷贝快照文件到对应的文件夹内，也可以通过 shell 脚本进行操作。实现逻辑如下：

1. 创建快照后，会在 leader Meta 服务和所有 Storage 服务的安装目录内生成 `checkpoints` 目录，保存创建的快照。以本文为例，当存在 2 个图空间时，创建的快照分别保存在 `/usr/local/nebula/data/meta/nebula/0/checkpoints` 、 `/usr/local/nebula/data/storage/nebula/3/checkpoints` 和 `/usr/local/nebula/data/storage/nebula/4/checkpoints` 中。

```
$ ls /usr/local/nebula/data/meta/nebula/0/checkpoints/
SNAPSHOT_2021_03_09_09_10_52
$ ls /usr/local/nebula/data/storage/nebula/3/checkpoints/
SNAPSHOT_2021_03_09_09_10_52
$ ls /usr/local/nebula/data/storage/nebula/4/checkpoints/
SNAPSHOT_2021_03_09_09_10_52
```

2. 当数据丢失需要通过快照恢复时，用户可以找到合适的时间点快照，将内部的文件夹 `data` 和 `wal` 分别拷贝到各自的上级目录（和 `checkpoints` 平级），覆盖之前的 `data` 和 `wal` ，然后重启集群即可。

> Warning
>
> 需要同时覆盖所有 Meta 服务的 data 和 wal 目录，因为存在重启集群后发生 Meta 服务重新选举 leader 的情况，如果不覆盖所有 Meta 服务，新的 leader 使用的还是最新的 Meta 服务数据，导致恢复失败。

最后更新: 2025年8月4日

10. 同步与迁移

10.1 负载均衡

我们可以提交任务让 NebulaGraph 的 Storage 服务实现负载均衡。详细示例请参见 Storage 负载均衡。

> Note
>
> 其他作业管理命令请参见作业管理。

负载均衡相关的语法说明如下。

语法	说明
`SUBMIT JOB BALANCE LEADER`	启动任务均衡分布所有图空间中的 leader。该命令会返回任务 ID（`job_id`）。

查看、停止、重启任务，请参见作业管理。

最后更新: 2025年8月4日

11. 导入与导出

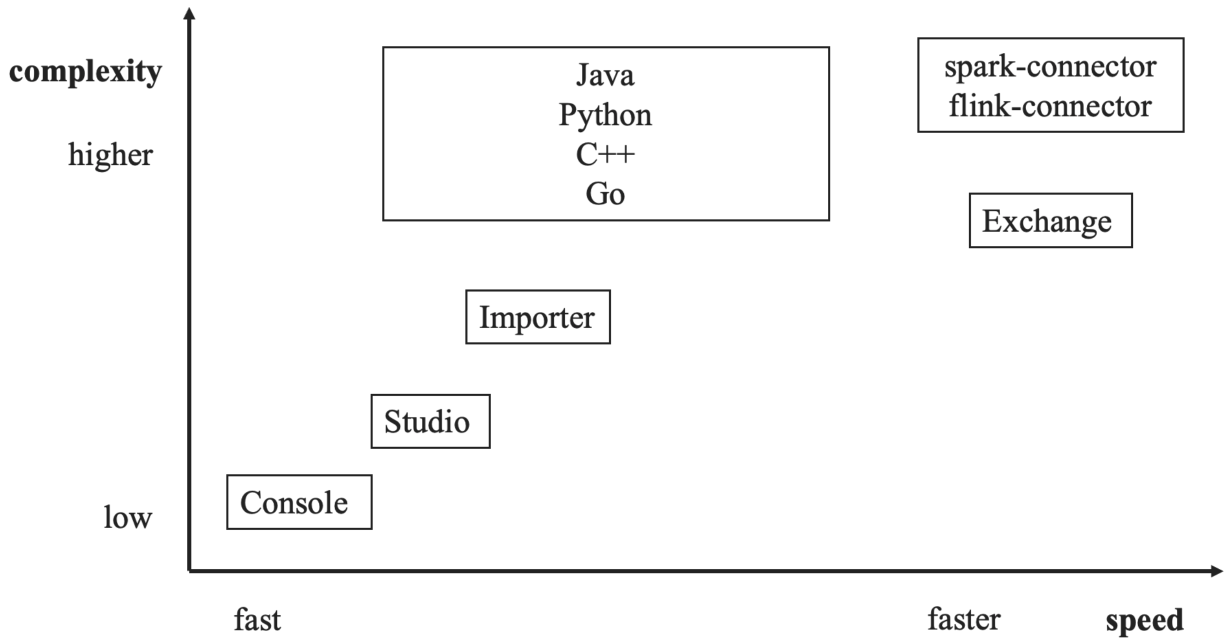
11.1 导入导出工具概述

11.1.1 导入工具

有多种方式可以将数据写入NebulaGraph master：

- 使用命令行 -f 的方式导入：可以导入少量准备好的 nGQL 文件，适合少量手工测试数据准备。
- 使用 Studio 导入：可以用过浏览器导入本机多个 CSV 文件，格式有限制。
- 使用 Importer 导入：导入单机多个 CSV 文件，大小没有限制，格式灵活。适合十亿条数据以内的场景。
- 使用 Exchange 导入：从 Neo4j、Hive、MySQL 等多种源分布式导入，需要有 Spark 集群。适合十亿条数据以上的场景。
- 使用 Spark-connector/Flink-connector 读写 API：这种方式需要编写少量代码来使用 Spark/Flink 连接器提供的 API。
- 使用 C++/GO/Java/Python SDK：编写程序的方式导入，需要有一定编程和调优能力。

下图给出了几种方式的定位：



11.1.2 导出工具

- 使用 Spark-connector/Flink-connector 读写 API：这种方式需要编写少量代码来使用 Spark/Flink 连接器提供的 API。
- 使用 Exchange 导出功能将数据导出至 CSV 文件或另一个图空间（支持不同 NebulaGraph 集群）中。

> Enterpriseonly
>
> 仅企业版 Exchange 提供导出功能。如需企业版，请联系我们。

最后更新: 2025年8月4日

11.2 NebulaGraph Importer

NebulaGraph Importer（简称 Importer）是一款 NebulaGraph 的 CSV 文件单机导入工具，可以读取并批量导入多种数据源的 CSV 文件数据，还支持批量更新和删除操作。

11.2.1 功能

- 支持多种数据源，包括本地、S3、OSS、HDFS、FTP、SFTP、GCS。
- 支持导入 CSV 格式文件的数据。单个文件内可以包含多种 Tag、多种 Edge type 或者二者混合的数据。
- 支持过滤数据源数据。
- 支持批量操作，包括导入、更新、删除。
- 支持同时连接多个 Graph 服务进行导入并且动态负载均衡。
- 支持失败后重连、重试。
- 支持多维度显示统计信息，包括导入时间、导入百分比等。统计信息支持打印在 Console 或日志中。
- 支持 SSL 加密。

11.2.2 优势

- 轻量快捷：不需要复杂环境即可使用，快速导入数据。
- 灵活筛选：通过配置文件可以实现对 CSV 文件数据的灵活筛选。

11.2.3 版本兼容性

NebulaGraph Importer 版本和 NebulaGraph 内核的版本对应关系如下。

NebulaGraph 版本	NebulaGraph Importer 版本
3.x.x	3.x.x、4.x.x
2.x.x	2.x.x、3.x.x

> Note
>
> Importer 4.0.0 对 Importer 进行了重做，性能得到了提高，但配置文件不兼容旧版本。建议使用新版 Importer。

11.2.4 更新说明

Release notes

11.2.5 前提条件

在使用 NebulaGraph Importer 之前，请确保：

- 已部署 NebulaGraph 服务。部署方式如下：
- RPM/DEB 包安装
- Docker Compose 部署
- 源码编译安装

- NebulaGraph 中已创建 Schema，包括图空间、Tag 和 Edge type，或者通过参数 `manager.hooks.before.statements` 设置。

11.2.6 操作步骤

创建 **CSV** 文件

准备好待导入的 CSV 文件并配置 YAML 文件，即可使用本工具向 NebulaGraph 批量导入数据。

> Note
>
> YAML 配置文件说明请参见下文的配置文件说明。

下载二进制包运行

1. 在 Release 页面下载和安装二进制包，并添加执行权限。

> Note
>
> 使用 RPM/DEB 包安装的文件路径为 `/usr/bin/nebula-importer` 。

2. 在 `nebula-importer` 的安装目录下，执行以下命令导入数据。

```
$ ./<binary_file_name> --config <yaml_config_file_path>
```

源码编译运行

编译源码需要部署 Golang 环境。详情请参见 Golang 环境搭建。

1. 克隆仓库。

```
$ git clone -b release-4.1 https://github.com/vesoft-inc/nebula-importer.git
```

> Note
>
> 请使用正确的分支。不同分支的 rpc 协议不同。

2. 进入目录 `nebula-importer` 。

```
$ cd nebula-importer
```

3. 编译源码。

```
$ make build
```

4. 开始导入数据。

```
$ ./bin/nebula-importer --config <yaml_config_file_path>
```

Docker 方式运行

使用 Docker 可以不必在本地安装 Go 语言环境，只需要拉取 NebulaGraph Importer 的镜像，并将本地配置文件和 CSV 数据文件挂载到容器中。命令如下：

```
$ docker pull vesoft/nebula-importer:<version>
$ docker run --rm -ti \
    --network=host \
    -v <config_file>:<config_file> \
    -v <data_dir>:<data_dir> \
```

```
    vesoft/nebula-importer:<version> \
    --config <config_file>
```

- `<config_file>`：填写 YAML 配置文件的绝对路径。
- `<data_dir>`：填写 CSV 数据文件的绝对路径。如果文件不在本地，请忽略该参数。
- `<version>`：填写 Importer 的版本号，请填写 `v4`。

> Note
>
> 建议使用相对路径。如果使用本地绝对路径，请检查路径映射到 Docker 中的路径。

例如：

```
$ docker pull vesoft/nebula-importer:v4
$ docker run --rm -ti \
    --network=host \
    -v /home/user/config.yaml:/home/user/config.yaml \
    -v /home/user/data:/home/user/data \
    vesoft/nebula-importer:v4 \
    --config /home/user/config.yaml
```

11.2.7 配置文件说明

NebulaGraph Importer 的 Github 内提供多种示例配置文件。配置文件用来描述待导入文件信息、 NebulaGraph 服务器信息等。下文将分类介绍配置文件内的字段。

> Note
>
> 如果用户下载的是二进制包，请手动创建配置文件。

Client 配置

Client 配置存储客户端连接 NebulaGraph 相关的配置。

示例配置如下：

```yaml
client:
  version: v3
  address: "192.168.1.100:9669,192.168.1.101:9669"
  user: root
  password: nebula
  ssl:
    enable: true
    certPath: "/home/xxx/cert/importer.crt"
    keyPath: "/home/xxx/cert/importer.key"
    caPath: "/home/xxx/cert/root.crt"
    insecureSkipVerify: false
  concurrencyPerAddress: 10
  reconnectInitialInterval: 1s
```

```
retry: 3
retryInitialInterval: 1s
```

参数	默认值	是否必须	说明
`client.version`	`v3`	是	指定连接的 NebulaGraph 的大版本。当前仅支持 `v3` 。
`client.address`	`"127.0.0.1:9669"`	是	指定连接的 NebulaGraph 地址。多个地址用英文逗号（,）分隔。
`client.user`	`root`	否	NebulaGraph 的用户名。
`client.password`	`nebula`	否	NebulaGraph 用户名对应的密码。
`client.ssl.enable`	`false`	否	指定是否开启 SSL 认证。
`client.ssl.certPath`	-	否	指定 SSL 公钥证书的存储路径。开启 SSL 认证后该参数必填。
`client.ssl.keyPath`	-	否	指定 SSL 密钥的存储路径。开启 SSL 认证后该参数必填。
`client.ssl.caPath`	-	否	指定 CA 根证书的存储路径。开启 SSL 认证后该参数必填。
`client.ssl.insecureSkipVerify`	`false`	否	指定是否跳过验证服务端的证书链和主机名。如果设置为 `true` ，则接受服务端提供的任何证书链和主机名。
`client.concurrencyPerAddress`	`10`	否	单个 Graph 服务的客户端并发连接数。
`client.retryInitialInterval`	`1s`	否	重连间隔时间。
`client.retry`	`3`	否	nGQL 语句执行失败的重试次数。
`client.retryInitialInterval`	`1s`	否	重试间隔时间。

Manager 配置

Manager 配置是连接数据库后的人为控制配置。

示例配置如下：

```
manager:
  spaceName: basic_string_examples
  batch: 128
  readerConcurrency: 50
  importerConcurrency: 512
  statsInterval: 10s
  hooks:
    before:
      - statements:
        - |
          DROP SPACE IF EXISTS basic_string_examples;
          CREATE SPACE IF NOT EXISTS basic_string_examples(partition_num=5, replica_factor=1, vid_type=int);
          USE basic_string_examples;
        wait: 10s
    after:
      - statements:
```

```
  - |
    SHOW SPACES;
```

参数	默认值	是否必须	说明
`manager.spaceName`	-	是	指定数据要导入的图空间。不支持同时导入多个图空间。
`manager.batch`	`128`	否	执行语句的批处理量（全局配置）。对某个数据源单独设置批处理量可以使用下文的 `sources.batch`。
`manager.readerConcurrency`	`50`	否	读取器读取数据源的并发数。
`manager.importerConcurrency`	`512`	否	生成待执行的 nGQL 语句的并发数，然后会调用客户端执行这些语句。
`manager.statsInterval`	`10s`	否	打印统计信息的时间间隔。
`manager.hooks.before.[].statements`	-	否	导入前在图空间内执行的命令。
`manager.hooks.before.[].wait`	-	否	执行 `statements` 语句后的等待时间。
`manager.hooks.after.[].statements`	-	否	导入后在图空间内执行的命令。
`manager.hooks.after.[].wait`	-	否	执行 `statements` 语句后的等待时间。

Log 配置

Log 配置是设置日志相关配置。

示例配置如下：

```
log:
  level: INFO
  console: true
  files:
    - logs/nebula-importer.log
```

参数	默认值	是否必须	说明
`log.level`	`INFO`	否	日志级别。可选值为 `DEBUG`、`INFO`、`WARN`、`ERROR`、`PANIC`、`FATAL`。
`log.console`	`true`	否	存储日志时是否将日志同步打印到 Console。
`log.files`	-	否	日志文件路径。需手动创建日志文件目录。

Source 配置

Source 配置中需要配置数据源信息、数据处理方式和 Schema 映射。

示例配置如下：

```
sources:
  - path: ./person.csv  # 指定存储数据文件的路径。如果使用相对路径，则路径和当前配置文件目录拼接。也支持通配符文件名，例如：./follower-*.csv，请确保所有匹配的文件具有相同的架构。
#  - s3: # AWS S3
#      endpoint: endpoint    # 可选。S3 服务端点，如果使用 AWS S3 可以省略。
#      region: us-east-1     # 必填。S3 服务的区域。
#      bucket: gdelt-open-data    # 必填。S3 服务中的 bucket。
#      key: events/20190918.export.csv     # 必填。S3 服务中文件的 key。
#      accessKeyID: ""    # 可选。S3 服务的访问密钥。如果是公共数据，则无需配置。
#      accessKeySecret: ""     # 可选。S3 服务的密钥。如果是公共数据，则无需配置。
#  - oss:
#      endpoint: https://oss-cn-hangzhou.aliyuncs.com    # 必填。OSS 服务端点。
#      bucket: bucketName    # 必填。OSS 服务中的 bucket。
#      key: objectKey    # 必填。OSS 服务中文件的 object key。
#      accessKeyID: accessKey    # 必填。OSS 服务的访问密钥。
#      accessKeySecret: secretKey    # 必填。OSS 服务的秘钥。
#  - ftp:
#      host: 192.168.0.10    # 必填。FTP 服务的主机。
#      port: 21    # 必填。FTP 服务的端口。
#      user: user    # 必填。FTP 服务的用户名。
#      password: password    # 必填。FTP 服务的密码。
#      path: "/events/20190918.export.csv"    # FTP 服务中文件的路径。
#  - sftp:
```

```yaml
#     host: 192.168.0.10   # 必填。SFTP 服务的主机。
#     port: 22   # 必填。SFTP 服务的端口。
#     user: user   # 必填。SFTP 服务的用户名。
#     password: password   # 可选。SFTP 服务的密码。
#     keyFile: keyFile   # 可选。SFTP 服务的 SSH 密钥文件路径。
#     keyData: keyData   $ 可选。SFTP 服务的 SSH 密钥文件内容。
#     passphrase: passphrase   # 可选。SFTP 服务的 SSH 密钥密码。
#     path: "/events/20190918.export.csv"   # 必填。SFTP 服务中文件的路径。
#  - hdfs:
#     address: "127.0.0.1:8020"   # 必填。HDFS 服务的地址。
#     user: "hdfs"   # 可选。HDFS 服务的用户名。
#     servicePrincipalName: <Kerberos Service Principal Name> # 可选。启用 Kerberos 认证时，HDFS 服务的 Kerberos 服务实例名称。
#     krb5ConfigFile: <Kerberos config file> # 可选。启用 Kerberos 认证时，HDFS 服务的 Kerberos 配置文件路径，默认为`/etc/krb5.conf`。
#     ccacheFile: <Kerberos ccache file> # 可选。启用 Kerberos 认证时，HDFS 服务的 Kerberos ccache 文件路径。
#     keyTabFile: <Kerberos keytab file> # 可选。启用 Kerberos 认证时，HDFS 服务的 Kerberos keytab 文件路径。
#     password: <Kerberos password> # 可选。启用 Kerberos 认证时，HDFS 服务的 Kerberos 密码。
#     dataTransferProtection: <Kerberos Data Transfer Protection> # 可选。启用 Kerberos 认证时的传输加密类型。可选值为`authentication`、`integrity`、`privacy`。
#     disablePAFXFAST: false # 可选。是否禁止客户端使用预身份验证（PA_FX_FAST）。
#     path: "/events/20190918.export.csv"   # 必填。HDFS 服务中文件的路径。也支持通配符文件名，例如：/events/*.export.csv，请确保所有匹配的文件具有相同的架构。
#  - gcs: # Google Cloud Storage
#     bucket: chicago-crime-sample  # 必填。GCS 服务中的 bucket 名称。
#     key: stats/000000000000.csv  # 必填。GCS 服务中文件的路径。
#     withoutAuthentication: false  # 可选。是否匿名访问。默认为 false，即使用凭证访问。
#     # 使用凭证访问时，credentialsFile 和 credentialsJSON 参数二选一即可。
#     credentialsFile: "/path/to/your/credentials/file"  # 可选。GCS 服务的凭证文件路径。
#     credentialsJSON: '{  # 可选。GCS 服务的凭证 JSON 内容。
#       "type": "service_account",
#       "project_id": "your-project-id",
#       "private_key_id": "key-id",
#       "private_key": "-----BEGIN PRIVATE KEY-----\nxxxxx\n-----END PRIVATE KEY-----\n",
#       "client_email": "your-client@your-project-id.iam.gserviceaccount.com",
#       "client_id": "client-id",
#       "auth_uri": "https://accounts.google.com/o/oauth2/auth",
#       "token_uri": "https://oauth2.googleapis.com/token",
#       "auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs",
#       "client_x509_cert_url": "https://www.googleapis.com/robot/v1/metadata/x509/your-client%40your-project-id.iam.gserviceaccount.com",
#       "universe_domain": "googleapis.com"
#     }'

    batch: 256
    csv:
      delimiter: "|"
      withHeader: false
      lazyQuotes: false
    tags:
    - name: Person
#     mode: INSERT
#     filter:
#       expr: Record[1] == "XXX"
      id:
        type: "STRING"
        function: "hash"
#       index: 0
        concatItems:
          - person_
          - 0
          - _id
      props:
        - name: "firstName"
          type: "STRING"
          index: 1
        - name: "lastName"
          type: "STRING"
          index: 2
        - name: "gender"
          type: "STRING"
          index: 3
          nullable: true
          defaultValue: female
        - name: "birthday"
          type: "DATE"
          index: 4
          nullable: true
          nullValue: _NULL_
        - name: "creationDate"
          type: "DATETIME"
          index: 5
        - name: "locationIP"
          type: "STRING"
          index: 6
        - name: "browserUsed"
          type: "STRING"
          index: 7
  - path: ./knows.csv
    batch: 256
    edges:
    - name: KNOWS # person_knows_person
#     mode: INSERT
#     filter:
#       expr: Record[1] == "XXX"
      src:
        id:
          type: "STRING"
          concatItems:
```

```
      - person_
      - 0
      - _id
  dst:
    id:
      type: "STRING"
      concatItems:
        - person_
        - 1
        - _id
  props:
    - name: "creationDate"
      type: "DATETIME"
      index: 2
      nullable: true
      nullValue: _NULL_
      defaultValue: 0000-00-00T00:00:00
```

配置主要包括以下几个部分：

- 指定数据源信息。
- 指定执行语句的批处理量。
- 指定 CSV 文件格式信息。
- 指定 Tag 的模式映射。
- 指定 Edge type 的模式映射。

参数	默认值	是否必须	说明
`sources.path` `sources.s3` `sources.oss` `sources.ftp` `sources.sftp` `sources.hdfs`	-	否	指定数据源信息，例如本地文件、HDFS、S3 等。一个 `sour` 见示例内的注释。
`sources.batch`	`256`	否	导入该数据源时执行语句的批处理量。优先级高于 `manager.l`
`sources.csv.delimiter`	`,`	否	CSV 文件的分隔符。仅支持 1 个字符的字符串分隔符。使用 为： `"\x03"` 或 `"\u0003"` 。关于 yaml 格式特殊字符转义的细
`sources.csv.withHeader`	`false`	否	是否忽略 CSV 文件中的第一条记录。
`sources.csv.lazyQuotes`	`false`	否	是否允许惰性解析引号。如果值为 `true` ，引号可以出现在非
`sources.tags.name`	-	是	Tag 名称。
`sources.tags.mode`	`INSERT`	否	批量操作类型，包括导入、更新和删除。可选值为 `INSERT` 、
`sources.tags.filter.expr`	-	否	过滤数据，满足过滤条件的才会导入。支持的比较符为 `==` 、 `(Record[0] == "Mahinda" or Record[0] == "Michael") and Recor`
`sources.tags.id.type`	`STRING`	否	VID 的类型。
`sources.tags.id.function`	-	否	生成 VID 的函数。目前仅支持 `hash` 。
`sources.tags.id.index`	-	否	VID 对应的数据文件中的列号。如果未配置 `sources.tags.id.`
`sources.tags.id.concatItems`	-	否	用于连接两个或多个数组，连接项可以是 `string` 、 `int` 或者 效。
`sources.tags.ignoreExistedIndex`	`true`	否	是否启用 `IGNORE_EXISTED_INDEX` ，即插入点后不更新索引。
`sources.tags.props.name`	-	是	VID 上的属性名称，必须与数据库中的属性相同。
`sources.tags.props.type`	`STRING`	否	VID 上属性的数据类型。目前支持 `BOOL` 、 `INT` 、 `FLOAT` 、 `DOU` `GEOGRAPHY(LINESTRING)` 、 `GEOGRAPHY(POLYGON)` 。
`sources.tags.props.index`	-	是	属性值对应的数据文件中的列号。
`sources.tags.props.nullable`	`false`	否	属性是否可以为 `NULL` ，可选 `true` 或者 `false` 。
`sources.tags.props.nullValue`	-	否	`nullable` 设置为 `true` 时，属性的值与 `nullValue` 相等则将该属
`sources.tags.props.alternativeIndices`	-	否	当 `nullable` 为 `false` 时忽略。该属性根据索引顺序从文件中获
`sources.tags.props.defaultValue`	-	否	当 `nullable` 为 `false` 时忽略。根据 `index` 和 `alternativeIndices`
`sources.edges.name`	-	是	Edge type 名称。
`sources.edges.mode`	`INSERT`	否	批量操作类型，包括导入、更新和删除。可选值为 `INSERT` 、
`sources.edges.filter.expr`	-	否	过滤数据，满足过滤条件的才会导入。支持的比较符为 `==` 、 `(Record[0] == "Mahinda" or Record[0] == "Michael") and Recor`
`sources.edges.src.id.type`	`STRING`	否	边上起点 VID 的数据类型。
`sources.edges.src.id.index`	-	是	边上起点 VID 对应的数据文件中的列号。
`sources.edges.dst.id.type`	`STRING`	否	边上终点 VID 的数据类型。
`sources.edges.dst.id.index`	-	是	边上终点 VID 对应的数据文件中的列号。
`sources.edges.rank.index`	-	否	边上 RANK 对应的数据文件中的列号。
`sources.edges.ignoreExistedIndex`	`true`	否	是否启用 `IGNORE_EXISTED_INDEX` ，即插入点后不更新索引。
`sources.edges.props.name`	-	否	边上属性的名称，必须与数据库中的属性相同。

参数	默认值	是否必须	说明
`sources.edges.props.type`	`STRING`	否	边上属性的数据类型。目前支持 `BOOL`、`INT`、`FLOAT`、`DOUBL` `GEOGRAPHY(LINESTRING)`、`GEOGRAPHY(POLYGON)`。
`sources.edges.props.index`	-	否	属性值对应的数据文件中的列号。
`sources.edges.props.nullable`	-	否	属性是否可以为 `NULL`，可选 `true` 或者 `false`。
`sources.edges.props.nullValue`	-	否	`nullable` 设置为 `true` 时，属性的值与 `nullValue` 相等则将该
`sources.edges.props.defaultValue`	-	否	当 `nullable` 为 `false` 时忽略。根据 `index` 和 `alternativeIndice`

> Note
>
> CSV 文件中列的序号从 0 开始，即第一列的序号为 0，第二列的序号为 1。

11.2.8 社区用户实践

- NebulaGraph Importer 数据导入实践和总结
- 基于 Nebula-Importer 批量导入工具性能验证方案总结
- 详解 nebula-importer 性能测试和数据导入调优

> Note
>
> 阅读他人实践后倘若想按原文实践，请留意原文所用的内核和周边工具版本号，请确保你的软件环境和原文兼容。

11.2.9 常见问题

日志输出的字段含义是什么?

例如以下日志内容：

```
“msg”: “44m20s 2h7m10s 25.85%(129 GiB/498 GiB) Records{Finished: 302016726, Failed: 0, Rate: 113538.13/s}, Requests{Finished: 181786, Failed: 0, Latency: 4.046496736s/4.06694393s, Rate:
68.34/s}, Processed{Finished: 908575178, Failed: 0, Rate: 341563.62/s}”
```

字段说明如下：

- `44m20s 2h7m10s 25.85%(129 GiB/498 GiB)` 对应基本信息，没有字段名称。
- 已用时间。
- 预计剩余时间。
- 已处理百分比。
- 已处理的数据大小。
- 总数据大小。
- `Records` 对应 CSV 文件的记录。
- `Finished`：完成的记录数。
- `Failed`：失败的记录数。
- `Rate`：每秒导入的记录数。
- `Requests` 对应请求。
- `Finished`：完成的请求数。
- `Failed`：失败的请求数。
- `Latency`：服务端请求耗时/客户端请求耗时。
- `Rate`：每秒处理的请求数。
- `Processed` 对应点边。
- `Finished`：处理完成的点边数量。
- `Failed`：处理失败的点边数量。
- `Rate`：每秒处理的点边数量。

最后更新: 2025年8月4日

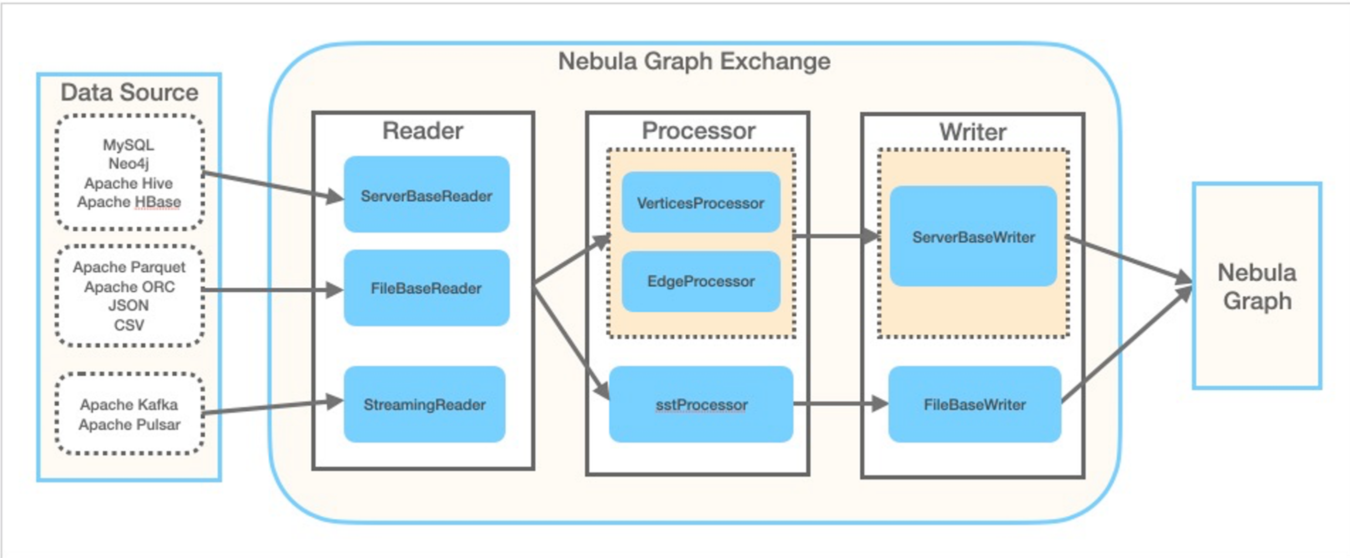
11.3 NebulaGraph Exchange

11.3.1 认识 NebulaGraph Exchange

什么是 NebulaGraph Exchange

NebulaGraph Exchange（简称 Exchange）是一款 Apache Spark™ 应用，用于在分布式环境中将集群中的数据批量迁移到 NebulaGraph 中，能支持多种不同格式的批式数据和流式数据的迁移。

Exchange 由 Reader、Processor 和 Writer 三部分组成。Reader 读取不同来源的数据返回 DataFrame 后，Processor 遍历 DataFrame 的每一行，根据配置文件中 `fields` 的映射关系，按列名获取对应的值。在遍历指定批处理的行数后，Writer 会将获取的数据一次性写入到 NebulaGraph 中。下图描述了 Exchange 完成数据转换和迁移的过程。



版本系列

Exchange 有社区版和企业版两个系列，二者功能不同。社区版在 GitHub 开源开发，企业版属于 NebulaGraph 企业套餐。

适用场景

Exchange 适用于以下场景：

- 需要将来自 Kafka、Pulsar 平台的流式数据，如日志文件、网购数据、游戏内玩家活动、社交网站信息、金融交易或地理空间服务，以及来自数据中心内所连接设备或仪器的遥测数据等转化为属性图的点或边数据，并导入 NebulaGraph 。
- 需要从关系型数据库（如 MySQL）或者分布式文件系统（如 HDFS）中读取批式数据，如某个时间段内的数据，将它们转化为属性图的点或边数据，并导入 NebulaGraph 。
- 需要将大批量数据生成 NebulaGraph 能识别的 SST 文件，再导入 NebulaGraph 。
- 需要导出 NebulaGraph 中保存的数据。

> Enterpriseonly
>
> 仅企业版 Exchange 支持从 NebulaGraph 中导出数据。

产品优点

Exchange 具有以下优点：

- 适应性强：支持将多种不同格式或不同来源的数据导入 NebulaGraph，便于迁移数据。
- 支持导入 SST：支持将不同来源的数据转换为 SST 文件，用于数据导入。
- 支持 SSL 加密：支持在 Exchange 与 NebulaGraph 之间建立 SSL 加密传输通道，保障数据安全。
- 支持断点续传：导入数据时支持断点续传，有助于节省时间，提高数据导入效率。

> Note
>
> 目前仅迁移 Neo4j 数据时支持断点续传。

- 异步操作：会在源数据中生成一条插入语句，发送给 Graph 服务，最后再执行插入操作。
- 灵活性强：支持同时导入多个 Tag 和 Edge type，不同 Tag 和 Edge type 可以是不同的数据来源或格式。
- 统计功能：使用 Apache Spark™ 中的累加器统计插入操作的成功和失败次数。
- 易于使用：采用 HOCON（Human-Optimized Config Object Notation）配置文件格式，具有面向对象风格，便于理解和操作。

版本兼容性

Exchange 支持 Spark 版本 2.2.x、2.4.x 和 3.x.x，针对不同 Spark 版本命名为：`nebula-exchange_spark_2.2`、`nebula-exchange_spark_2.4` 和 `nebula-exchange_spark_3.0`。

NebulaGraph Exchange 版本（即 JAR 包版本）、 NebulaGraph 内核版本和 Spark 版本对应关系如下。

Exchange 版本	NebulaGraph 版本	Spark 版本
nebula-exchange_spark_3.0-3.0-SNAPSHOT.jar	nightly	3.3.x、3.2.x、3.1.x、3.0.x
nebula-exchange_spark_2.4-3.0-SNAPSHOT.jar	nightly	2.4.x
nebula-exchange_spark_2.2-3.0-SNAPSHOT.jar	nightly	2.2.x
nebula-exchange_spark_3.0-3.4.0.jar	3.x.x	3.3.x、3.2.x、3.1.x、3.0.x
nebula-exchange_spark_2.4-3.4.0.jar	3.x.x	2.4.x
nebula-exchange_spark_2.2-3.4.0.jar	3.x.x	2.2.x
nebula-exchange_spark_3.0-3.3.0.jar	3.x.x	3.3.x、3.2.x、3.1.x、3.0.x
nebula-exchange_spark_2.4-3.3.0.jar	3.x.x	2.4.x
nebula-exchange_spark_2.2-3.3.0.jar	3.x.x	2.2.x
nebula-exchange_spark_3.0-3.0.0.jar	3.x.x	3.3.x、3.2.x、3.1.x、3.0.x
nebula-exchange_spark_2.4-3.0.0.jar	3.x.x	2.4.x
nebula-exchange_spark_2.2-3.0.0.jar	3.x.x	2.2.x
nebula-exchange-2.6.3.jar	2.6.1、2.6.0	2.4.x
nebula-exchange-2.6.2.jar	2.6.1、2.6.0	2.4.x
nebula-exchange-2.6.1.jar	2.6.1、2.6.0	2.4.x
nebula-exchange-2.6.0.jar	2.6.1、2.6.0	2.4.x
nebula-exchange-2.5.2.jar	2.5.1、2.5.0	2.4.x
nebula-exchange-2.5.1.jar	2.5.1、2.5.0	2.4.x
nebula-exchange-2.5.0.jar	2.5.1、2.5.0	2.4.x
nebula-exchange-2.1.0.jar	2.0.1、2.0.0	2.4.x
nebula-exchange-2.0.1.jar	2.0.1、2.0.0	2.4.x
nebula-exchange-2.0.0.jar	2.0.1、2.0.0	2.4.x

JAR 包有两种获取方式：自行编译或者从 maven 仓库下载。

数据源

Exchange 3.8.0 支持将以下格式或来源的数据转换为 NebulaGraph 能识别的点和边数据，然后通过 nGQL 语句的形式导入 NebulaGraph：

- 存储在 HDFS 或本地的数据：
- Apache Parquet
- Apache ORC
- JSON
- CSV

- Apache HBase™

- 数据仓库：
- Hive
- MaxCompute

- 图数据库：Neo4j（Client 版本 2.4.5-M1）

- 关系型数据库：
- MySQL
- PostgreSQL
- Oracle

- 列式数据库：ClickHouse

- 流处理软件平台：Apache Kafka®

- 发布/订阅消息平台：Apache Pulsar 2.4.5

- JDBC 数据源

除了用 nGQL 语句的形式导入数据，Exchange 还支持将数据源的数据生成 SST 文件，然后通过 Console 导入 SST 文件。

更新说明

Release

视频

- NebulaGraph 数据导入工具——Exchange（3 分 08 秒）

最后更新: 2025年8月4日

使用限制

本文描述 Exchange 3.8.0 的一些使用限制。

使用环境

Exchange 3.x 支持以下操作系统：

- CentOS 7
- macOS

软件依赖

为保证 Exchange 正常工作，请确认机器上已经安装如下软件：

- Java 1.8 版本
- Scala 2.10.7、2.11.12 或 2.12.10 版本
- Apache Spark。使用 Exchange 从不同数据源导出数据对 Spark 版本的要求如下：

> Note
>
> 使用 Exchange 时，需根据 Spark 版本选择相应的 JAR 文件。例如，当 Spark 版本为 2.4 时，选择 nebula-exchange_spark_2.4-3.8.0.jar。

数据源	Spark 2.2	Spark 2.4	Spark 3
CSV 文件	支持	支持	支持
JSON 文件	支持	支持	支持
ORC 文件	支持	支持	支持
Parquet 文件	支持	支持	支持
HBase	支持	支持	支持
MySQL	支持	支持	支持
PostgreSQL	支持	支持	支持
Oracle	支持	支持	支持
ClickHouse	支持	支持	支持
Neo4j	不支持	支持	不支持
Hive	支持	支持	支持
MaxCompute	不支持	支持	不支持
Pulsar	不支持	支持	未测试
Kafka	不支持	支持	未测试
NebulaGraph	不支持	支持	不支持

在以下使用场景，还需要部署 Hadoop Distributed File System (HDFS)：

- 迁移 HDFS 的数据
- 生成 SST 文件

最后更新: 2025年8月4日

11.3.2 获取 NebulaGraph Exchange

本文介绍如何获取 NebulaGraph Exchange 的 JAR 文件。

直接下载 JAR 文件

社区版 Exchange 的 JAR 文件可以直接下载。

要下载企业版 Exchange，需先联系我们。

编译源代码获取 JAR 文件

社区版 Exchange 的 JAR 文件还可以通过编译源代码获取。下文介绍如何编译 Exchange 源代码。

> Enterpriseonly
>
> 企业版 Exchange 仅能在 NebulaGraph 企业版套餐中获取。

前提条件

- 安装 Maven。
- 根据数据源安装需要的 Spark 版本，从各数据源导出数据支持的 Spark 版本参见软件依赖。

操作步骤

1. 在根目录克隆仓库 `nebula-exchange`。

```
git clone -b release-3.8 https://github.com/vesoft-inc/nebula-exchange.git
```

2. 切换到目录 `nebula-exchange`。

```
cd nebula-exchange
```

3. 根据 Exchange 使用环境中的 Spark 版本打包 Exchange。

- Spark 2.2：

```
mvn clean package -Dmaven.test.skip=true -Dgpg.skip -Dmaven.javadoc.skip=true \
-pl nebula-exchange_spark_2.2 -am -Pscala-2.11 -Pspark-2.2
```

- Spark 2.4：

```
mvn clean package -Dmaven.test.skip=true -Dgpg.skip -Dmaven.javadoc.skip=true \
-pl nebula-exchange_spark_2.4 -am -Pscala-2.11 -Pspark-2.4
```

- Spark 3.0：

```
mvn clean package -Dmaven.test.skip=true -Dgpg.skip -Dmaven.javadoc.skip=true \
-pl nebula-exchange_spark_3.0 -am -Pscala-2.12 -Pspark-3.0
```

编译成功后，可以在 `nebula-exchange_spark_x.x/target/` 目录里找到 `nebula-exchange_spark_x.x-release-3.8.jar` 文件。 `x.x` 代表 Spark 版本，例如 `2.4` 。

> Note
>
> JAR 文件版本号会因 NebulaGraph Java Client 的发布版本而变化。用户可以在 Releases 页面查看最新版本。

迁移数据时，用户可以参考配置文件 `target/classes/application.conf` 。

下载依赖包失败

如果编译时下载依赖包失败：

- 检查网络设置，确认网络正常。

- 修改 Maven 安装目录下 `libexec/conf/settings.xml` 文件的 `mirror` 部分：

```
<mirror>
 <id>alimaven</id>
 <mirrorOf>central</mirrorOf>
 <name>aliyun maven</name>
 <url>http://maven.aliyun.com/nexus/content/repositories/central/</url>
</mirror>
```

最后更新: 2025年8月4日

11.3.3 参数说明

导入命令参数

完成配置文件修改后，可以运行以下命令将指定来源的数据导入 NebulaGraph 数据库。

导入数据

```
<spark_install_path>/bin/spark-submit --master "spark://HOST:PORT" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-2.x.y.jar_path> -c <application.conf_path>
```

> Note
>
> 如果数据的属性值包含中文字符，可能出现乱码。请在提交 Spark 任务时加上以下选项：
>
> ```
> --conf spark.driver.extraJavaOptions=-Dfile.encoding=utf-8
> --conf spark.executor.extraJavaOptions=-Dfile.encoding=utf-8
> ```

参数说明如下。

参数	是否必需	默认值	说明
`--class`	是	无	指定驱动的主类。
`--master`	是	无	指定 Spark 集群的 master URL。详情请参见 master-urls。可选值为： `local`： 本地模式，使用单个线程运行 Spark 应用程序。适合在测试环境进行小数据量导入。 `yarn`： 在 YARN 集群上运行 Spark 应用程序。适合在线上环境进行大数据量导入。 `spark://HOST:PORT`： 连接到指定的 Spark standalone 集群。 `mesos://HOST:PORT`： 连接到指定的 Mesos 集群。 `k8s://HOST:PORT`： 连接到指定的 Kubernetes 集群。
`-c` / `--config`	是	无	指定配置文件的路径。
`-h` / `--hive`	否	`false`	添加这个参数表示支持从 Hive 中导入数据。
`-D` / `--dry`	否	`false`	指定是否检查配置文件的格式。该参数仅用于检查配置文件的格式，不检查 `tags` 和 `edges` 配置项的有效性，也不会导入数据。需要导入数据时不要添加这个参数。
`-r` / `--reload`	否	无	指定需要重新加载的 reload 文件路径。

更多 Spark 的参数配置说明请参见 Spark Configuration。

> Note
>
> - JAR 文件版本号以实际编译得到的 JAR 文件名称为准。
> - 如果使用 yarn 模式提交任务，请参考如下示例，尤其是示例中的两个 `--conf`。
>
> ```
> $SPARK_HOME/bin/spark-submit  --master yarn \
> --class com.vesoft.nebula.exchange.Exchange \
> --files application.conf \
> --conf spark.driver.extraClassPath=./ \
> --conf spark.executor.extraClassPath=./ \
> nebula-exchange-3.8.0.jar \
> -c application.conf
> ```

导入 RELOAD 文件

如果导入数据时有一些数据导入失败，会将导入失败的数据存入 reload 文件，可以用参数 `-r` 尝试导入 reload 文件中的数据。

```
<spark_install_path>/bin/spark-submit --master "spark://HOST:PORT" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange-2.x.y.jar_path> -c <application.conf_path> -r
"<reload_file_path>"
```

如果仍然导入失败，请到论坛寻求帮助。

最后更新: 2025年8月4日

配置说明

本文介绍使用 NebulaGraph Exchange 时如何自动生成示例配置文件，以及介绍配置文件 `application.conf`。

自动生成示例配置文件

通过如下命令，指定要导入的数据源，即可获得该数据源所对应的配置文件示例。

```
java -cp <exchange_jar_package> com.vesoft.exchange.common.GenerateConfigTemplate -s <source_type> -p
<config_file_save_path>
```

例如：

```
java -cp nebula-exchange_spark_2.4-3.0-SNAPSHOT.jar com.vesoft.exchange.common.GenerateConfigTemplate -s csv -p /home/nebula/csv_application.conf
```

使用加密密码

用户在配置文件里设置连接 NebulaGraph 的密码时，可以使用明文密码，也可以使用 RSA 加密后的密码。

如果需要使用 RSA 加密后的密码，需要在配置文件内进行如下设置：

- `nebula.pswd` 配置为 RSA 加密后的密码。
- `nebula.privateKey` 配置为 RSA 加密的密钥。
- `nebula.enableRSA` 配置为 `true`。

用户可以使用自己的工具加密，也可以使用 Exchange 的 JAR 包内提供的加密工具，例如：

```
spark-submit --master local --class com.vesoft.exchange.common.PasswordEncryption nebula-exchange_spark_2.4-3.0-SNAPSHOT.jar -p nebula
```

返回结果如下：

```
=================== public key begin ===================
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCLl7LaNSEXlZo2hYiJqzxgyFBQdkxbQXYU/
xQthsBJwjOPhkiY37nokzKnjNlp6mv5ZUomqxLsoNQHEJ6BZD4VPiaiElFAkTD+gyul1v8f3A446Fr2rnVLogWHnz8ECPt7X8jwmpiKOXkOPIhqU5EOCua+KkOnnVosbos/VShfiQIDAQAB
=================== public key end ===================


=================== private key begin ===================
MIICeAIBADANBgkqhkiG9w0BAQEFAASCAmIwggJeAgEAAoGBAIuXsto1IReVmjaFiImrPGDIUFB2TFtBdhT/FC2GwEnCM4+GSJjfueiTMqeM2Wnqa/llSiarEuyg1AcQnoFkPhU+JqISUUCRMP6DK6XW/x/
cDjjoWvaudUuiBYefPwQI+3tfyPCamIo5eQ48iGpTkTQK5r4qTSedWixuiz9VKF+JAgMBAAECgYADWbfEPwQ1UbTq3Bej3kVLuWMcGOrH4fFYnaq5UQOqgYvFRR7W9H+80lOj6+CIBOViLgkylmaU4WNVbBOx3VsUFFWSqIIIviKubg8m8ey7KAd9X2wMEcUH
WSacaXYS5LOmMevvuaOwLEVOQmyM+nNGRIjUdzCLR1935QJBAM+IF8YD5GnoAPPjGIDS1Ljhu/u/Gj6/YBCQKSHQ5+HxHEKjQ/YxQZ/otchmMZanYelf1y+byuJX3NZ04/
KSGT8CQQCsMaoFO2rF5M84HpAXPi6yH2chbtzOVTKZworwUnpmMVbNUojf4VwzAyOhT1U5oOPpFbpi+NqQhC63VUN5k0O3AkEArI8vnVGNMlZbvG7e5/bmM9hWs2viSbxdBOinOtv2g1M1OV+B2gp405ru0/
PNVcRVOHQFfCuhVfTSxmspQoAihwJBAJW6EZa/FZbB4JVxreUoAr6Lo8dkeOhT9M3SZbGWZivaFxot/Cp/
8QXCYwbuzrJxjqlsZUeOD6694Uk08JkURnOCQQC8V6aRa8ylMhLJFkGkMDHLqHcQCmY53Kd73mUu4+mjMJLZh14zQD9ydFtcOlbLXTeBAMWV3uEdeLhRvdAo3OwV
=================== private key end ===================


=================== encrypted password begin ===================
Io+3y3mLOMnZJJNUPHZ8pKb4VfTvg6wUh6jSu5xdmLAoX/59tK1HTwoN40aOOWJwa1a5io7S4JqcX/jEcAorw7pelITr+F4oBOAMCt71d+gJuu3/lw9bjUEl9tF4Raj82y2Dg39wYbagN84fZMgCD63TPiDIevSr6+MFKASpGrY=
=================== encrypted password end ===================
check: the real password decrypted by private key and encrypted password is: nebula
```

配置说明

修改配置文件之前，建议根据数据源复制并修改文件名称，便于区分。例如数据源为 CSV 文件，可以复制为 `csv_application.conf`。

配置文件的内容主要分为如下几类：

- Spark 相关配置
- Hive 配置（可选）
- NebulaGraph 相关配置
- 点配置
- 边配置

Spark 相关配置

本文只列出部分 Spark 参数，更多参数请参见官方文档。

参数	数据类型	默认值	是否必须	说明
`spark.app.name`	string	-	否	Spark 驱动程序名称。
`spark.driver.cores`	int	`1`	否	驱动程序使用的 CPU 核数，仅适用于集群模式。
`spark.driver.maxResultSize`	string	`1G`	否	单个 Spark 操作（例如 collect）时，所有分区的序列化结果的总大小限制（字节为单位）。最小值为 1M，0 表示无限制。
`spark.executor.memory`	string	`1G`	否	Spark 驱动程序使用的内存量，可以指定单位，例如 512M、1G。
`spark.cores.max`	int	`16`	否	当驱动程序以“粗粒度”共享模式在独立部署集群或 Mesos 集群上运行时，跨集群（而非从每台计算机）请求应用程序的最大 CPU 核数。如果未设置，则值为 Spark 的独立集群管理器上的 `spark.deploy.defaultCores` 或 Mesos 上的 infinite（所有可用的内核）。

Hive 配置（可选）

如果 Spark 和 Hive 部署在不同集群，才需要配置连接 Hive 的参数，否则请忽略这些配置。

参数	数据类型	默认值	是否必须	说明
`hive.warehouse`	string	-	是	HDFS 中的 warehouse 路径。用双引号括起路径，以 `hdfs://` 开头。
`hive.connectionURL`	string	-	是	JDBC 连接的 URL。例如 `"jdbc:mysql://127.0.0.1:3306/hive_spark?characterEncoding=UTF-8"`。
`hive.connectionDriverName`	string	`"com.mysql.jdbc.Driver"`	是	驱动名称。
`hive.connectionUserName`	list[string]	-	是	连接的用户名。
`hive.connectionPassword`	list[string]	-	是	用户名对应的密码。

NebulaGraph相关配置

参数	数据类型	默认值	是否必须	说明
`nebula.address.graph`	list[string]	`["127.0.0.1:9669"]`	是	所有 Graph 服务的地址，包括 IP 和端口，多个地址用英文逗号（,）分隔。格式为 `["ip1:port1","ip2:port2","ip3:port3"]` 。
`nebula.address.meta`	list[string]	`["127.0.0.1:9559"]`	是	所有 Meta 服务的地址，包括 IP 和端口，多个地址用英文逗号（,）分隔。格式为 `["ip1:port1","ip2:port2","ip3:port3"]` 。
`nebula.user`	string	-	是	拥有 NebulaGraph 写权限的用户名。
`nebula.pswd`	string	-	是	用户名对应的密码。密码支持明文或者 RSA 加密后的密码。如果使用 RSA 加密后的密码，需要设置 `enableRSA` 和 `privateKey` 。如何加密密码，参见上文使用加密密码。
`nebula.enableRSA`	bool	`false`	否	是否使用 RSA 加密的密码。
`nebula.privateKey`	string	-	否	RSA 加密的密钥。
`nebula.space`	string	-	是	需要导入数据的的图空间名称。
`nebula.ssl.enable.graph`	bool	`false`	是	开启 Exchange 与 Graph 服务之间的 SSL 加密传输。当值为 `true` 时开启，下方的 SSL 相关参数生效。如果 Exchange 运行在多机集群上，在设置以下 SSL 相关路径时，需要在每台机器的相同路径都存储相应的文件。
`nebula.ssl.enable.meta`	bool	`false`	是	开启 Exchange 与 Meta 服务之间的 SSL 加密传输。当值为 `true` 时开启，下方的 SSL 相关参数生效。如果 Exchange 运行在多机集群上，在设置以下 SSL 相关路径时，需要在每台机器的相同路径都存储相应的文件。
`nebula.ssl.sign`	string	`ca`	是	签名方式，可选值： `ca` （CA 签名）或 `self` （自签名）。
`nebula.ssl.ca.param.caCrtFilePath`	string	`"/path/caCrtFilePath"`	是	`nebula.ssl.sign` 的值为 `ca` 时生效，用于指定 CA 证书的存储路径。
`nebula.ssl.ca.param.crtFilePath`	string	`"/path/crtFilePath"`	是	`nebula.ssl.sign` 的值为 `ca` 时生效，用于指定 CRT 证书的存储路径。
`nebula.ssl.ca.param.keyFilePath`	string	`"/path/keyFilePath"`	是	`nebula.ssl.sign` 的值为 `ca` 时生效，用于指定私钥文件的存储路径。
`nebula.ssl.self.param.crtFilePath`	string	`"/path/crtFilePath"`	是	`nebula.ssl.sign` 的值为 `self` 时生效，用于指定 CRT 证书的存储路径。
`nebula.ssl.self.param.keyFilePath`	string	`"/path/keyFilePath"`	是	`nebula.ssl.sign` 的值为 `self` 时生效，用于指定私钥文件的存储路径。
`nebula.ssl.self.param.password`	string	`"nebula"`	是	`nebula.ssl.sign` 的值为 `self` 时生效，用于指定密码文件的存储路径。
`nebula.path.local`	string	`"/tmp"`	否	导入 SST 文件时需要设置本地 SST 文件路径。
`nebula.path.remote`	string	`"/sst"`	否	导入 SST 文件时需要设置远端 SST 文件路径。
`nebula.path.hdfs.namenode`	string	`"hdfs://name_node:9000"`	否	导入 SST 文件时需要设置 HDFS 的 namenode。

参数	数据类型	默认值	是否必须	说明
`nebula.connection.timeout`	int	`3000`	否	Thrift 连接的超时时间，单位为 ms。
`nebula.connection.retry`	int	`3`	否	Thrift 连接重试次数。
`nebula.execution.retry`	int	`3`	否	nGQL 语句执行重试次数。
`nebula.error.max`	int	`32`	否	导入过程中的最大失败次数。当失败次数达到最大值时，提交的 Spark 作业将自动停止。
`nebula.error.output`	string	`/tmp/errors`	否	输出错误日志的路径。错误日志保存执行失败的 nGQL 语句。
`nebula.rate.limit`	int	`1024`	否	导入数据时令牌桶的令牌数量限制。
`nebula.rate.timeout`	int	`1000`	否	令牌桶中拿取令牌的超时时间，单位：毫秒。

Note

NebulaGraph 默认不支持无 Tag 的点。如果需要导入无 Tag 的点，需要先在集群内开启支持无 Tag 点，然后在 Exchange 的配置文件内新增 `nebula.enableTagless` 参数，值为 `true`。示例如下：

```
nebula: {
  address:{
   graph:["127.0.0.1:9669"]
   meta:["127.0.0.1:9559"]
  }
  user: root
  pswd: nebula
  space: test
  enableTagless: true
  ......

}
```

点配置

对于不同的数据源，点的配置也有所不同，有很多通用参数，也有部分特有参数，配置时需要配置通用参数和不同数据源的特有参数。

通用参数

参数	数据类型	默认值	是否必须	说明
`tags.name`	string	-	是	NebulaGraph 中定义的 Tag 名称。
`tags.type.source`	string	-	是	指定数据源。例如 `csv`。
`tags.type.sink`	string	`client`	是	指定导入方式，可选值为 `client` 和 `SST`。
`tags.writeMode`	string	`INSERT`	否	对数据的批量操作类型，包括批量导入、更新和删除。可选值为 `INSERT`、`UPDATE`、`DELETE`。
`tags.deleteEdge`	string	`false`	否	进行批量删除操作时是否删除该点关联的出边和入边。`tags.writeMode` 为 `DELETE` 时该参数生效。
`tags.fields`	list[string]	-	是	属性对应的列的表头或列名。如果有表头或列名，请直接使用该名称。如果 CSV 文件没有表头，用 `[_c0, _c1, _c2]` 的形式表示第一列、第二列、第三列，以此类推。
`tags.nebula.fields`	list[string]	-	是	NebulaGraph 中定义的属性名称，顺序必须和 `tags.fields` 一一对应。例如 `[_c1, _c2]` 对应 `[name, age]`，表示第二列为属性 name 的值，第三列为属性 age 的值。
`tags.vertex.field`	string	-	是	点 ID 的列。例如 CSV 文件没有表头时，可以用 `_c0` 表示第一列的值作为点 ID。
`tags.vertex.udf.separator`	string	-	否	通过自定义规则合并多列，该参数指定连接符。
`tags.vertex.udf.oldColNames`	list	-	否	通过自定义规则合并多列，该参数指定待合并的列名。多个列用英文逗号（,）分隔。
`tags.vertex.udf.newColName`	string	-	否	通过自定义规则合并多列，该参数指定新列的列名。
`tags.vertex.prefix`	string	-	否	为 VID 增加指定的前缀。例如 VID 为 `12345`，增加前缀 `tag1` 后为 `tag1_12345`。下划线无法修改。
`tags.vertex.policy`	string	-	否	仅支持取值 `hash`。对 string 类型的 VID 进行哈希化操作。
`tags.batch`	int	`256`	是	单批次写入NebulaGraph的最大点数量。
`tags.partition`	int	`32`	是	数据写入NebulaGraph时需要创建的分区数。如果 `tags.partition ≤ 1`，在NebulaGraph中创建的分区数和数据源的分区数相同。
`tags.filter`	string	-	否	过滤规则。符合过滤规则的数据会被导入NebulaGraph。格式请参见Dataset。

Parquet/JSON/ORC 源特有参数

参数	数据类型	默认值	是否必须	说明
`tags.path`	string	-	是	HDFS 中点数据文件的路径。用双引号括起路径，以 `hdfs://` 开头。

CSV 源特有参数

参数	数据类型	默认值	是否必须	说明
`tags.path`	string	-	是	HDFS 中点数据文件的路径。用双引号括起路径，以 `hdfs://` 开头。
`tags.separator`	string	`,`	是	分隔符。默认值为英文逗号（,）。对于特殊字符，如控制符 `^A`,可以用 ASCII 八进制 `\001` 或 UNICODE 编码十六进制 `\u0001` 表示，控制符 `^B`，用 ASCII 八进制 `\002` 或 UNICODE 编码十六进制 `\u0002` 表示，控制符 `^C`，用 ASCII 八进制 `\003` 或 UNICODE 编码十六进制 `\u0003` 表示。
`tags.header`	bool	`true`	是	文件是否有表头。

Hive 源特有参数

参数	数据类型	默认值	是否必须	说明
`tags.exec`	string	-	是	查询数据源的语句。例如 `select name,age from mooc.users`。

MaxCompute 源特有参数

参数	数据类型	默认值	是否必须	说明
`tags.table`	string	-	是	MaxCompute 的表名。
`tags.project`	string	-	是	MaxCompute 的项目名。
`tags.odpsUrl`	string	-	是	MaxCompute 服务的 odpsUrl。地址可根据阿里云文档查看。
`tags.tunnelUrl`	string	-	是	MaxCompute 服务的 tunnelUrl。地址可根据阿里云文档查看。
`tags.accessKeyId`	string	-	是	MaxCompute 服务的 accessKeyId。
`tags.accessKeySecret`	string	-	是	MaxCompute 服务的 accessKeySecret。
`tags.partitionSpec`	string	-	否	MaxCompute 表的分区描述。
`tags.numPartitions`	int	`1`	否	MaxCompute 的 Spark 连接器在读取 MaxCompute 数据时使用的分区数。
`tags.sentence`	string	-	否	查询数据源的语句。SQL 语句中的表名和上方 table 的值相同。

Neo4j 源特有参数

参数	数据类型	默认值	是否必须	说明
`tags.exec`	string	-	是	查询数据源的语句。例如 `match (n:label) return n.neo4j-field-0`。
`tags.server`	string	`"bolt://127.0.0.1:7687"`	是	Neo4j 服务器地址。
`tags.user`	string	-	是	拥有读取权限的 Neo4j 用户名。
`tags.password`	string	-	是	用户名对应密码。
`tags.database`	string	-	是	Neo4j 中保存源数据的数据库名。
`tags.check_point_path`	string	`/tmp/test`	否	设置保存导入进度信息的目录，用于断点续传。如果未设置，表示不启用断点续传。

MySQL/PostgreSQL 源特有参数

参数	数据类型	默认值	是否必须	说明
`tags.host`	string	-	是	MySQL/PostgreSQL 服务器地址。
`tags.port`	string	-	是	MySQL/PostgreSQL 服务器端口。
`tags.database`	string	-	是	数据库名称。
`tags.table`	string	-	是	需要作为数据源的表名称。
`tags.user`	string	-	是	拥有读取权限的 MySQL/PostgreSQL 用户名。
`tags.password`	string	-	是	用户名对应密码。
`tags.sentence`	string	-	是	查询数据源的语句。例如 `"select teamid, name from team order by teamid"`。

Oracle 源特有参数

参数	数据类型	默认值	是否必须	说明
`tags.url`	string	-	是	Oracle 数据库连接地址。
`tags.driver`	string	-	是	Oracle 驱动地址。
`tags.user`	string	-	是	拥有读取权限的 Oracle 用户名。
`tags.password`	string	-	是	用户名对应密码。
`tags.table`	string	-	是	需要作为数据源的表名称。
`tags.sentence`	string	-	是	查询数据源的语句。例如 `"select playerid, name, age from player"`。

ClickHouse 源特有参数

参数	数据类型	默认值	是否必须	说明
`tags.url`	string	-	是	ClickHouse 的 JDBC URL。
`tags.user`	string	-	是	有读取权限的 ClickHouse 用户名。
`tags.password`	string	-	是	用户名对应密码。
`tags.numPartition`	string	-	是	ClickHouse 分区数。
`tags.sentence`	string	-	是	查询数据源的语句。

Hbase 源特有参数

参数	数据类型	默认值	是否必须	说明
`tags.host`	string	`127.0.0.1`	是	Hbase 服务器地址。
`tags.port`	string	`2181`	是	Hbase 服务器端口。
`tags.table`	string	-	是	需要作为数据源的表名称。
`tags.columnFamily`	string	-	是	表所属的列族（column family）。

Pulsar 源特有参数

参数	数据类型	默认值	是否必须	说明
`tags.service`	string	`"pulsar://localhost:6650"`	是	Pulsar 服务器地址。
`tags.admin`	string	`"http://localhost:8081"`	是	连接 pulsar 的 admin.url。
`tags.options.<topic|topics|topicsPattern>`	string	-	是	Pulsar 的选项，可以从 `topic`、`topics` 和 `topicsPattern` 选择一个进行配置。
`tags.interval.seconds`	int	`10`	是	读取消息的间隔。单位：秒。

Kafka 源特有参数

参数	数据类型	默认值	是否必须	说明
`tags.service`	string	-	是	Kafka 服务器地址。
`tags.topic`	string	-	是	消息类别。
`tags.interval.seconds`	int	`10`	是	读取消息的间隔。单位：秒。
`tags.securityProtocol`	string	-	否	Kafka 安全协议。
`tags.mechanism`	string	-	否	Kafka SASL 安全认证机制。
`tags.kerberos`	bool	`false`	否	是否启用 Kerberos 进行安全认证。如果 `tags.mechanism` 为 `kerberos` ，该参数必须设置为 `true` 。
`tags.kerberosServiceName`	string	-	否	Kerberos 服务名称。如果 `tags.kerberos` 为 `true` ，该参数必填。

生成 SST 时的特有参数

参数	数据类型	默认值	是否必须	说明
`tags.path`	string	-	是	指定需要生成 SST 文件的源文件的路径。
`tags.repartitionWithNebula`	bool	`true`	否	生成 SST 文件时是否要基于 NebulaGraph 中图空间的 partition 进行数据重分区。开启该功能可减少 DOWNLOAD 和 INGEST SST 文件需要的时间。

边配置

对于不同的数据源，边的配置也有所不同，有很多通用参数，也有部分特有参数，配置时需要配置通用参数和不同数据源的特有参数。

边配置的不同数据源特有参数请参见上方点配置内的特有参数介绍，注意区分 tags 和 edges 即可。

通用参数

参数	数据类型	默认值	是否必须	说明
`edges.name`	string	-	是	NebulaGraph 中定义的 Edge type 名称。
`edges.type.source`	string	-	是	指定数据源。例如 `csv` 。
`edges.type.sink`	string	`client`	是	指定导入方式，可选值为 `client` 和 `SST` 。
`edges.writeMode`	string	`INSERT`	否	对数据的批量操作类型，包括批量导入、更新和删除。可选值为 `INSERT` 、 `UPDATE` 、 `DELETE` 。
`edges.fields`	list[string]	-	是	属性对应的列的表头或列名。如果有表头或列名，请直接使用该名称。如果 CSV 文件没有表头，用 `[_c0, _c1, _c2]` 的形式表示第一列、第二列、第三列，以此类推。
`edges.nebula.fields`	list[string]	-	是	NebulaGraph 中定义的属性名称，顺序必须和 `edges.fields` 一一对应。例如 `[_c2, _c3]` 对应 `[start_year, end_year]` ，表示第三列为开始年份的值，第四列为结束年份的值。
`edges.source.field`	string	-	是	边的起始点的列。例如 `_c0` 表示第一列的值作为边的起始点。
`edges.source.prefix`	string	-	否	为 VID 增加指定的前缀。例如 VID 为 `12345` ，增加前缀 `tag1` 后为 `tag1_12345` 。下划线无法修改。
`edges.source.policy`	string	-	否	仅支持取值 `hash` 。对 string 类型的 VID 进行哈希化操作。
`edges.target.field`	string	-	是	边的目的点的列。例如 `_c1` 表示第二列的值作为边的目的点。
`edges.target.prefix`	string	-	否	为 VID 增加指定的前缀。例如 VID 为 `12345` ，增加前缀 `tag1` 后为 `tag1_12345` 。下划线无法修改。
`edges.target.policy`	string	-	否	仅支持取值 `hash` 。对 string 类型的 VID 进行哈希化操作。
`edges.ranking`	int	-	否	rank 值的列。没有指定时，默认所有 rank 值为 `0` 。
`edges.batch`	int	`256`	是	单批次写入NebulaGraph的最大边数量。
`edges.partition`	int	`32`	是	数据写入NebulaGraph时需要创建的分区数。如果 `edges.partition ≤ 1` ，在NebulaGraph中创建的分区数和数据源的分区数相同。
`edges.filter`	string	-	否	过滤规则。符合过滤规则的数据会被导入NebulaGraph。格式请参见Dataset。

生成 SST 时的特有参数

参数	数据类型	默认值	是否必须	说明
`edges.path`	string	-	是	指定需要生成 SST 文件的源文件的路径。
`edges.repartitionWithNebula`	bool	`true`	否	生成 SST 文件时是否要基于 NebulaGraph 中图空间的 partition 进行数据重分区。开启该功能可减少 DOWNLOAD 和 INGEST SST 文件需要的时间。

最后更新: 2025年8月4日

11.3.4 使用 NebulaGraph Exchange

导入 CSV 文件数据

本文以一个示例说明如何使用 Exchange 将存储在 HDFS 或本地的 CSV 文件数据导入 NebulaGraph 。

数据集

本文以 basketballplayer 数据集为例。

环境配置

本文示例在 MacOS 下完成，以下是相关的环境配置信息：

- 硬件规格：
- CPU：1.7 GHz Quad-Core Intel Core i7
- 内存：16 GB

- Spark：2.4.7 单机版

- Hadoop：2.9.2 伪分布式部署

- NebulaGraph ： master。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署 NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的的 IP 地址和端口。
- 拥有 NebulaGraph 写权限的用户名和密码。

- 已经编译 Exchange。详情请参见编译 Exchange。本示例中使用 Exchange 3.8.0。

- 已经安装 Spark。

- 了解 NebulaGraph 中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。

- 如果文件存储在 HDFS 上，需要确认 Hadoop 服务运行正常。

- 如果文件存储在本地且 NebulaGraph 是集群架构，需要在集群每台机器本地相同目录下放置文件。

操作步骤

步骤 1：在 **NebulaGraph** 中创建 **Schema**

分析 CSV 文件中的数据，按以下步骤在 NebulaGraph 中创建 Schema：

1. 确认 Schema 要素。 NebulaGraph 中的 Schema 要素如下表所示。

要素	名称	属性
Tag	`player`	`name string, age int`
Tag	`team`	`name string`
Edge Type	`follow`	`degree int`
Edge Type	`serve`	`start_year int, end_year int`

2. 使用 NebulaGraph Console 创建一个图空间 basketballplayer，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
        (partition_num = 10, \
        replica_factor = 1, \
        vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见快速开始。

步骤 2：处理 **CSV** 文件

确认以下信息：

1. 处理 CSV 文件以满足 Schema 的要求。

> Note
>
> Exchange 支持上传有表头或者无表头的 CSV 文件。

2. 获取 CSV 文件存储路径。

步骤 3：修改配置文件

编译 Exchange 后，复制 `target/classes/application.conf` 文件设置 CSV 数据源相关的配置。在本示例中，复制的文件名为 `csv_application.conf`。各个配置项的详细说明请参见配置说明。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.8.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    executor: {
        memory:1G
    }

    cores: {
```

```
      max: 16
    }
  }

  # NebulaGraph 相关配置
  nebula: {
    address:{
      # 指定 Graph 服务和所有 Meta 服务的 IP 地址和端口。
      # 如果有多台服务器，地址之间用英文逗号（,）分隔。
      # 格式："ip1:port","ip2:port","ip3:port"
      graph:["127.0.0.1:9669"]
      #任意一个 Meta 服务的地址。
      #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
      meta:["127.0.0.1:9559"]
    }

    # 指定拥有 NebulaGraph 写权限的用户名和密码。
    user: root
    pswd: nebula
    # 是否使用 RSA 加密的密码。
    # enableRSA: true
    # 使用 RSA 加密密码时的密钥。
    # privateKey: ""
    # 指定图空间名称。
    space: basketballplayer
    connection: {
      timeout: 3000
      retry: 3
    }
    execution: {
      retry: 3
    }
    error: {
      max: 32
      output: /tmp/errors
    }
    rate: {
      limit: 1024
      timeout: 1000
    }
  }

  # 处理点
  tags: [
    # 设置 Tag player 相关信息。
    {
      # 指定 NebulaGraph 中定义的 Tag 名称。
      name: player
      type: {
        # 指定数据源，使用 CSV。
        source: csv

        # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
        sink: client
      }

      # 指定 CSV 文件的路径。
      # 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
      # 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.csv"。
      path: "hdfs://192.168.*.*:9000/data/vertex_player.csv"

      # 如果 CSV 文件没有表头，使用 [_c0, _c1, _c2, ..., _cn] 表示其表头，并将列指示为属性值的源。
      # 如果 CSV 文件有表头，则使用实际的列名。
      fields: [_c1, _c2]

      # 指定 NebulaGraph 中定义的属性名称。
      # fields 与 nebula.fields 的顺序必须一一对应。
      nebula.fields: [age, name]

      # 指定一个列作为 VID 的源。
      # vertex 的值必须与上述 fields 或者 csv.fields 中的列名保持一致。
      # 目前，NebulaGraph  master仅支持字符串或整数类型的 VID。
      vertex: {
        field:_c0
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      # 指定的分隔符。默认值为英文逗号（,）。
      separator: ","

      # 如果 CSV 文件有表头，请将 header 设置为 true。
      # 如果 CSV 文件没有表头，请将 header 设置为 false。默认值为 false。
      header: false

      # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
      # filter: "name='Tom'"
```

```
      # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
      #writeMode: INSERT

      # 批量删除时是否删除该点关联的出边和入边。`writeMode`为`DELETE`时该参数生效。
      #deleteEdge: false

      # 指定单批次写入 NebulaGraph 的最大点数量。
      batch: 256

      # 数据写入 NebulaGraph 时需要创建的分区数。
      partition: 32
    }

    # 设置 Tag team 相关信息。
    {
      name: team
      type: {
        source: csv
        sink: client
      }
      path: "hdfs://192.168.*.*:9000/data/vertex_team.csv"
      fields: [_c1]
      nebula.fields: [name]
      vertex: {
        field:_c0
      }
      separator: ","
      header: false
      batch: 256
      partition: 32
    }
    # 如果需要添加更多点，请参考前面的配置进行添加。
  ]
  # 处理边
  edges: [
    # 设置 Edge type follow 相关信息。
    {
      # 指定 NebulaGraph 中定义的 Edge type 名称。
      name: follow
      type: {
        # 指定数据源，使用 CSV。
        source: csv

        # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
        sink: client
      }

      # 指定 CSV 文件的路径。
      # 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
      # 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.csv"。
      path: "hdfs://192.168.*.*:9000/data/edge_follow.csv"

      # 如果 CSV 文件没有表头，使用 [_c0, _c1, _c2, ..., _cn] 表示其表头，并将列指示为属性值的源。
      # 如果 CSV 文件有表头，则使用实际的列名。
      fields: [_c2]

      # 指定 NebulaGraph 中定义的属性名称。
      # fields 与 nebula.fields 的顺序必须一一对应。
      nebula.fields: [degree]

      # 指定一个列作为起始点和目的点的源。
      # vertex 的值必须与上述 fields 或者 csv.fields 中的列名保持一致。
      # 目前，NebulaGraph  master仅支持字符串或整数类型的 VID。
      source: {
        field: _c0
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }
      target: {
        field: _c1
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      # 指定的分隔符。默认值为英文逗号（,）。
      separator: ","

      # 指定一个列作为 rank 的源（可选）。
```

```
      #ranking: rank

      # 如果 CSV 文件有表头，请将 header 设置为 true。
      # 如果 CSV 文件没有表头，请将 header 设置为 false。默认值为 false。
      header: false

      # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
      # filter: "name='Tom'"

      # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
      #writeMode: INSERT

      # 指定单批次写入 NebulaGraph 的最大边数量。
      batch: 256

      # 数据写入 NebulaGraph 时需要创建的分区数。
      partition: 32
    }

    # 设置 Edge type serve 相关信息。
    {
      name: serve
      type: {
        source: csv
        sink: client
      }
      path: "hdfs://192.168.*.*:9000/data/edge_serve.csv"
      fields: [_c2,_c3]
      nebula.fields: [start_year, end_year]
      source: {
        field: _c0
      }
      target: {
        field: _c1
      }
      separator: ","
      header: false
      batch: 256
      partition: 32
    }

  ]
  # 如果需要添加更多边，请参考前面的配置进行添加。
}
```

步骤 4：向 **NebulaGraph** 导入数据

运行如下命令将 CSV 文件数据导入到 NebulaGraph 中。关于参数的说明，请参见导入命令参数。

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange.jar_path> -c <csv_application.conf_path>
```

> Note
>
> JAR 包有两种获取方式：自行编译或者从 maven 仓库下载。

示例：

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange_spark_2.4-3.8.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/csv_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`，确认成功的数量。例如 `batchSuccess.follow: 300`。

访问 Kerberos 认证的 HDFS

使用 Kerberos 进行安全认证时，需使用以下两种方式之一访问 Kerberos 认证的 HDFS。

- 在命令中设置 Kerberos 配置文件

  在命令中配置 `--conf` 和 `--files` ，例如：

```
${SPARK_HOME}/bin/spark-submit --master xxx --num-executors 2 --executor-cores 2 --executor-memory 1g \
--conf "spark.driver.extraJavaOptions=-Djava.security.krb5.conf=./krb5.conf" \
--conf "spark.executor.extraJavaOptions=-Djava.security.krb5.conf=./krb5.conf" \
--files /local/path/to/xxx.keytab,/local/path/to/krb5.conf \
--class com.vesoft.nebula.exchange.Exchange \
exchange.jar -c xx.conf
```

  `--conf` 中的文件路径有如下两种配置方式：

- 配置文件的绝对路径。要求所有 YARN 或者 Spark 机器相同路径下都有对应文件。
- （YARN 模式下推荐）配置文件的相对路径（例如 `./krb5.conf` ）。通过 `--files` 上传的资源文件就在 Java 虚拟机或者 JAR 的工作目录下。

  `--files` 中的文件必须存储在执行 `spark-submit` 命令的机器上。

- 不使用命令

  将 Spark 和 Kerberos 认证的 Hadoop 部署在相同集群内，共用 HDFS 和 YARN，然后在 Spark 的 `spark-env.sh` 中增加配置 `export HADOOP_HOME=<hadoop_home_path>` 。

步骤 **5**：（可选）验证数据

用户可以在 NebulaGraph 客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 **6**：（如有）在 **NebulaGraph** 中重建索引

导入数据后，用户可以在 NebulaGraph 中重新创建并重建索引。详情请参见索引介绍。

最后更新: 2025年8月4日

导入 JSON 文件数据

本文以一个示例说明如何使用 Exchange 将存储在 HDFS 或本地的 JSON 文件数据导入 NebulaGraph 。

数据集

本文以 basketballplayer 数据集为例。部分示例数据如下：

- player

```
{"id":"player100","age":42,"name":"Tim Duncan"}
{"id":"player101","age":36,"name":"Tony Parker"}
{"id":"player102","age":33,"name":"LaMarcus Aldridge"}
{"id":"player103","age":32,"name":"Rudy Gay"}
...
```

- team

```
{"id":"team200","name":"Warriors"}
{"id":"team201","name":"Nuggets"}
...
```

- follow

```
{"src":"player100","dst":"player101","degree":95}
{"src":"player101","dst":"player102","degree":90}
...
```

- serve

```
{"src":"player100","dst":"team204","start_year":"1997","end_year":"2016"}
{"src":"player101","dst":"team204","start_year":"1999","end_year":"2018"}
...
```

环境配置

本文示例在 MacOS 下完成，以下是相关的环境配置信息：

- 硬件规格：
- CPU：1.7 GHz Quad-Core Intel Core i7
- 内存：16 GB

- Spark：2.3.0，单机版

- Hadoop：2.9.2，伪分布式部署

- NebulaGraph ： master。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署 NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的的 IP 地址和端口。
- 拥有 NebulaGraph 写权限的用户名和密码。

- 已经编译 Exchange。详情请参见编译 Exchange。本示例中使用 Exchange 3.8.0。

- 已经安装 Spark。

- 了解 NebulaGraph 中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。

- 如果文件存储在 HDFS 上，需要确认 Hadoop 服务运行正常。

- 如果文件存储在本地且 NebulaGraph 是集群架构，需要在集群每台机器本地相同目录下放置文件。

操作步骤

步骤 1：在 NebulaGraph 中创建 Schema

分析文件中的数据，按以下步骤在 NebulaGraph 中创建 Schema：

1. 确认 Schema 要素。 NebulaGraph 中的 Schema 要素如下表所示。

要素	名称	属性
Tag	`player`	`name string, age int`
Tag	`team`	`name string`
Edge Type	`follow`	`degree int`
Edge Type	`serve`	`start_year int, end_year int`

2. 使用 NebulaGraph Console 创建一个图空间 basketballplayer，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
        (partition_num = 10, \
        replica_factor = 1, \
        vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见快速开始。

步骤 2：处理 JSON 文件

确认以下信息：

1. 处理 JSON 文件以满足 Schema 的要求。
2. 获取 JSON 文件存储路径。

步骤 **3.** 修改配置文件

编译 Exchange 后，复制 `target/classes/application.conf` 文件设置 JSON 数据源相关的配置。在本示例中，复制的文件名为 `json_application.conf` 。各个配置项的详细说明请参见配置说明。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.8.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    executor: {
        memory:1G
    }

    cores: {
      max: 16
    }
  }

  # NebulaGraph 相关配置
  nebula: {
    address:{
      # 指定 Graph 服务和所有 Meta 服务的 IP 地址和端口。
      # 如果有多台服务器，地址之间用英文逗号（,）分隔。
      # 格式："ip1:port","ip2:port","ip3:port"
      graph:["127.0.0.1:9669"]
      #任意一个 Meta 服务的地址。
      #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
      meta:["127.0.0.1:9559"]
    }

    # 指定拥有 NebulaGraph 写权限的用户名和密码。
    user: root
    pswd: nebula
    # 是否使用 RSA 加密的密码。
    # enableRSA: true
    # 使用 RSA 加密密码时的密钥。
    # privateKey: ""
    # 指定图空间名称。
    space: basketballplayer
    connection: {
      timeout: 3000
      retry: 3
    }
    execution: {
      retry: 3
    }
    error: {
      max: 32
      output: /tmp/errors
    }
    rate: {
      limit: 1024
      timeout: 1000
    }
  }

  # 处理点
  tags: [
    # 设置 Tag player 相关信息。
    {
      # 指定 NebulaGraph 中定义的 Tag 名称。
      name: player
      type: {
        # 指定数据源，使用 JSON。
        source: json

        # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
        sink: client
      }

      # 指定 JSON 文件的路径。
      # 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
      # 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.json"。
      path: "hdfs://192.168.*.*:9000/data/vertex_player.json"

      # 在 fields 里指定 JSON 文件中 key 名称，其对应的 value 会作为 NebulaGraph 中指定属性的数据源。
      # 如果需要指定多个值，用英文逗号（,）隔开。
      fields: [age,name]

      # 指定 NebulaGraph 中定义的属性名称。
      # fields 与 nebula.fields 的顺序必须一一对应。
      nebula.fields: [age, name]

      # 指定一个列作为 VID 的源。
      # vertex 的值必须与 JSON 文件中的字段保持一致。
      # 目前，NebulaGraph  master仅支持字符串或整数类型的 VID。
```

```
      vertex: {
        field:id
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      # 指定单批次写入 NebulaGraph 的最大点数量。
      batch: 256

      # 数据写入 NebulaGraph 时需要创建的分区数。
      partition: 32
    }

    # 设置 Tag team 相关信息。
    {
      name: team
      type: {
        source: json
        sink: client
      }
      path: "hdfs://192.168.*.*:9000/data/vertex_team.json"
      fields: [name]
      nebula.fields: [name]
      vertex: {
        field:id
      }
      batch: 256
      partition: 32
    }

    # 如果需要添加更多点，请参考前面的配置进行添加。
  ]
  # 处理边
  edges: [
    # 设置 Edge type follow 相关信息。
    {
      # 指定 NebulaGraph 中定义的 Edge type 名称。
      name: follow
      type: {
        # 指定数据源，使用 JSON。
        source: json

        # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
        sink: client
      }

      # 指定 JSON 文件的路径。
      # 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
      # 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.json"。
      path: "hdfs://192.168.*.*:9000/data/edge_follow.json"

      # 在 fields 里指定 JSON 文件中 key 名称，其对应的 value 会作为 NebulaGraph 中指定属性的数据源。
      # 如果需要指定多个值，用英文逗号（,）隔开。
      fields: [degree]

      # 指定 NebulaGraph 中定义的属性名称。
      # fields 与 nebula.fields 的顺序必须一一对应。
      nebula.fields: [degree]

      # 指定一个列作为起始点和目的点的源。
      # vertex 的值必须与 JSON 文件中的字段保持一致。
      # 目前，NebulaGraph  master仅支持字符串或整数类型的 VID。
      source: {
        field: src
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }
      target: {
        field: dst
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }
```

```
      # 指定一个列作为 rank 的源（可选）。
      #ranking: rank

      # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
      # filter: "name='Tom'"

      # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
      #writeMode: INSERT

      # 指定单批次写入 NebulaGraph 的最大边数量。
      batch: 256

      # 数据写入 NebulaGraph 时需要创建的分区数。
      partition: 32
    }

    # 设置 Edge type serve 相关信息。
    {
      name: serve
      type: {
        source: json
        sink: client
      }
      path: "hdfs://192.168.*.*:9000/data/edge_serve.json"
      fields: [start_year,end_year]
      nebula.fields: [start_year, end_year]
      source: {
        field: src
      }
      target: {
        field: dst
      }
      batch: 256
      partition: 32
    }

  ]
  # 如果需要添加更多边，请参考前面的配置进行添加。
}
```

步骤 **4**：向 **NebulaGraph** 导入数据

运行如下命令将 JSON 文件数据导入到 NebulaGraph 中。关于参数的说明，请参见导入命令参数。

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange.jar_path> -c <json_application.conf_path>
```

> **Note**
>
> JAR 包有两种获取方式：自行编译或者从 maven 仓库下载。

示例：

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-echange/nebula-exchange/target/nebula-exchange_spark_2.4-3.8.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/json_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`，确认成功的数量。例如 `batchSuccess.follow: 300`。

访问 Kerberos 认证的 HDFS

使用 Kerberos 进行安全认证时，需使用以下两种方式之一访问 Kerberos 认证的 HDFS。

- 在命令中设置 Kerberos 配置文件

  在命令中配置 `--conf` 和 `--files`，例如：

```
${SPARK_HOME}/bin/spark-submit --master xxx --num-executors 2 --executor-cores 2 --executor-memory 1g \
--conf "spark.driver.extraJavaOptions=-Djava.security.krb5.conf=./krb5.conf" \
--conf "spark.executor.extraJavaOptions=-Djava.security.krb5.conf=./krb5.conf" \
--files /local/path/to/xxx.keytab,/local/path/to/krb5.conf \
--class com.vesoft.nebula.exchange.Exchange \
exchange.jar -c xx.conf
```

  `--conf` 中的文件路径有如下两种配置方式：

- 配置文件的绝对路径。要求所有 YARN 或者 Spark 机器相同路径下都有对应文件。
- （YARN 模式下推荐）配置文件的相对路径（例如 `./krb5.conf`）。通过 `--files` 上传的资源文件就在 Java 虚拟机或者 JAR 的工作目录下。

  `--files` 中的文件必须存储在执行 `spark-submit` 命令的机器上。

- 不使用命令

  将 Spark 和 Kerberos 认证的 Hadoop 部署在相同集群内，共用 HDFS 和 YARN，然后在 Spark 的 `spark-env.sh` 中增加配置 `export HADOOP_HOME=<hadoop_home_path>`。

步骤 **5**：（可选）验证数据

用户可以在 NebulaGraph 客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 **6**：（如有）在 **NebulaGraph** 中重建索引

导入数据后，用户可以在 NebulaGraph 中重新创建并重建索引。详情请参见索引介绍。

最后更新: 2025年8月4日

导入 **ORC** 文件数据

本文以一个示例说明如何使用 Exchange 将存储在 HDFS 或本地的 ORC 文件数据导入 NebulaGraph。

数据集

本文以 basketballplayer 数据集为例。

环境配置

本文示例在 MacOS 下完成，以下是相关的环境配置信息：

- 硬件规格：
- CPU：1.7 GHz Quad-Core Intel Core i7
- 内存：16 GB

- Spark：2.4.7 单机版

- Hadoop：2.9.2 伪分布式部署

- NebulaGraph： master。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署 NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的的 IP 地址和端口。
- 拥有 NebulaGraph 写权限的用户名和密码。

- 已经编译 Exchange。详情请参见编译 Exchange。本示例中使用 Exchange 3.8.0。

- 已经安装 Spark。

- 了解 NebulaGraph 中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。

- 如果文件存储在 HDFS 上，需要确认 Hadoop 服务运行正常。

- 如果文件存储在本地且 NebulaGraph 是集群架构，需要在集群每台机器本地相同目录下放置文件。

操作步骤

步骤 **1**：在 **NebulaGraph** 中创建 **Schema**

分析 ORC 文件中的数据，按以下步骤在 NebulaGraph 中创建 Schema：

1. 确认 Schema 要素。 NebulaGraph 中的 Schema 要素如下表所示。

要素	名称	属性
Tag	`player`	`name string, age int`
Tag	`team`	`name string`
Edge Type	`follow`	`degree int`
Edge Type	`serve`	`start_year int, end_year int`

2. 使用 NebulaGraph Console 创建一个图空间 basketballplayer，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
        (partition_num = 10, \
        replica_factor = 1, \
        vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见快速开始。

步骤 **2**：处理 **ORC** 文件

确认以下信息：

1. 处理 ORC 文件以满足 Schema 的要求。
2. 获取 ORC 文件存储路径。

步骤 **3**：修改配置文件

编译 Exchange 后，复制 `target/classes/application.conf` 文件设置 ORC 数据源相关的配置。在本示例中，复制的文件名为 `orc_application.conf`。各个配置项的详细说明请参见配置说明。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.8.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    executor: {
        memory:1G
    }

    cores: {
      max: 16
    }
  }

  #  NebulaGraph 相关配置
  nebula: {
    address:{
      # 指定 Graph 服务和所有 Meta 服务的 IP 地址和端口。
      # 如果有多台服务器，地址之间用英文逗号（,）分隔。
```

```
      # 格式："ip1:port","ip2:port","ip3:port"
      graph:["127.0.0.1:9669"]
      #任意一个 Meta 服务的地址。
      #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
      meta:["127.0.0.1:9559"]
    }

    # 指定拥有 NebulaGraph 写权限的用户名和密码。
    user: root
    pswd: nebula
    # 是否使用 RSA 加密的密码。
    # enableRSA: true
    # 使用 RSA 加密密码时的密钥。
    # privateKey: ""

    # 指定图空间名称。
    space: basketballplayer
    connection: {
      timeout: 3000
      retry: 3
    }
    execution: {
      retry: 3
    }
    error: {
      max: 32
      output: /tmp/errors
    }
    rate: {
      limit: 1024
      timeout: 1000
    }
  }

  # 处理点
  tags: [
    # 设置 Tag player 相关信息。
    {
      # 指定 NebulaGraph 中定义的 Tag 名称。
      name: player
      type: {
        # 指定数据源，使用 ORC。
        source: orc

        # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
        sink: client
      }

      # 指定 ORC 文件的路径。
      # 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
      # 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.orc"。
      path: "hdfs://192.168.*.*:9000/data/vertex_player.orc"

      # 在 fields 里指定 ORC 文件中 key 名称，其对应的 value 会作为 NebulaGraph 中指定属性的数据源。
      # 如果需要指定多个值，用英文逗号（,）隔开。
      fields: [age,name]

      # 指定 NebulaGraph 中定义的属性名称。
      # fields 与 nebula.fields 的顺序必须一一对应。
      nebula.fields: [age, name]

      # 指定一个列作为 VID 的源。
      # vertex 的值必须与 ORC 文件中的字段保持一致。
      # 目前，NebulaGraph  master仅支持字符串或整数类型的 VID。
      vertex: {
        field:id
      # udf:{
      #           separator:"_"
      #           oldColNames:[field-0,field-1,field-2]
      #           newColName:new-field
      #       }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
      # filter: "name='Tom'"

      # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
      #writeMode: INSERT

      # 批量删除时是否删除该点关联的出边和入边。`writeMode`为`DELETE`时该参数生效。
      #deleteEdge: false

      # 指定单批次写入 NebulaGraph 的最大点数量。
      batch: 256

      # 数据写入 NebulaGraph 时需要创建的分区数。
      partition: 32
    }

    # 设置 Tag team 相关信息。
```

```
  {
    name: team
    type: {
      source: orc
      sink: client
    }
    path: "hdfs://192.168.*.*:9000/data/vertex_team.orc"
    fields: [name]
    nebula.fields: [name]
    vertex: {
      field:id
    }
    batch: 256
    partition: 32
  }

  # 如果需要添加更多点，请参考前面的配置进行添加。
]
# 处理边
edges: [
  # 设置 Edge type follow 相关信息。
  {
    # 指定 NebulaGraph 中定义的 Edge type 名称。
    name: follow
    type: {
      # 指定数据源，使用 ORC。
      source: orc

      # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
      sink: client
    }

    # 指定 ORC 文件的路径。
    # 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
    # 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.orc"。
    path: "hdfs://192.168.*.*:9000/data/edge_follow.orc"

    # 在 fields 里指定 ORC 文件中 key 名称，其对应的 value 会作为 NebulaGraph 中指定属性的数据源。
    # 如果需要指定多个值，用英文逗号（,）隔开。
    fields: [degree]

    # 指定 NebulaGraph 中定义的属性名称。
    # fields 与 nebula.fields 的顺序必须一一对应。
    nebula.fields: [degree]

    # 指定一个列作为起始点和目的点的源。
    # vertex 的值必须与 ORC 文件中的字段保持一致。
    # 目前，NebulaGraph  master仅支持字符串或整数类型的 VID。
    source: {
      field: src
    # udf:{
    #            separator:"_"
    #            oldColNames:[field-0,field-1,field-2]
    #            newColName:new-field
    #        }
    # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
    # prefix:"tag1"
    # 对 string 类型的 VID 进行哈希化操作。
    # policy:hash
    }
    target: {
      field: dst
    # udf:{
    #            separator:"_"
    #            oldColNames:[field-0,field-1,field-2]
    #            newColName:new-field
    #        }
    # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
    # prefix:"tag1"
    # 对 string 类型的 VID 进行哈希化操作。
    # policy:hash
    }

    # 指定一个列作为 rank 的源（可选）。
    #ranking: rank

    # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
    # filter: "name='Tom'"

    # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
    #writeMode: INSERT

    # 指定单批次写入 NebulaGraph 的最大边数量。
    batch: 256

    # 数据写入 NebulaGraph 时需要创建的分区数。
    partition: 32
  }

  # 设置 Edge type serve 相关信息。
  {
    name: serve
    type: {
      source: orc
```

```
        sink: client
      }
      path: "hdfs://192.168.*.*:9000/data/edge_serve.orc"
      fields: [start_year,end_year]
      nebula.fields: [start_year, end_year]
      source: {
        field: src
      }
      target: {
        field: dst
      }
      batch: 256
      partition: 32
    }

  ]
  # 如果需要添加更多边，请参考前面的配置进行添加。
}
```

步骤 **4**: 向 **NebulaGraph** 导入数据

运行如下命令将 ORC 文件数据导入到 NebulaGraph 中。关于参数的说明，请参见导入命令参数。

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange.jar_path> -c <orc_application.conf_path>
```

> Note
>
> JAR 包有两种获取方式：自行编译或者从 maven 仓库下载。

示例：

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange_spark_2.4-3.8.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/orc_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`，确认成功的数量。例如 `batchSuccess.follow: 300`。

访问 Kerberos 认证的 HDFS

使用 Kerberos 进行安全认证时，需使用以下两种方式之一访问 Kerberos 认证的 HDFS。

- 在命令中设置 Kerberos 配置文件

  在命令中配置 `--conf` 和 `--files`，例如：

  ```
  ${SPARK_HOME}/bin/spark-submit --master xxx --num-executors 2 --executor-cores 2 --executor-memory 1g \
  --conf "spark.driver.extraJavaOptions=-Djava.security.krb5.conf=./krb5.conf" \
  --conf "spark.executor.extraJavaOptions=-Djava.security.krb5.conf=./krb5.conf" \
  --files /local/path/to/xxx.keytab,/local/path/to/krb5.conf \
  --class com.vesoft.nebula.exchange.Exchange \
  exchange.jar -c xx.conf
  ```

  `--conf` 中的文件路径有如下两种配置方式：

- 配置文件的绝对路径。要求所有 YARN 或者 Spark 机器相同路径下都有对应文件。
- （YARN 模式下推荐）配置文件的相对路径（例如 `./krb5.conf`）。通过 `--files` 上传的资源文件就在 Java 虚拟机或者 JAR 的工作目录下。

  `--files` 中的文件必须存储在执行 `spark-submit` 命令的机器上。

- 不使用命令

  将 Spark 和 Kerberos 认证的 Hadoop 部署在相同集群内，共用 HDFS 和 YARN，然后在 Spark 的 `spark-env.sh` 中增加配置 `export HADOOP_HOME=<hadoop_home_path>`。

步骤 **5**: （可选）验证数据

用户可以在 NebulaGraph 客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 **6**：（如有）在 **NebulaGraph** 中重建索引

导入数据后，用户可以在 NebulaGraph 中重新创建并重建索引。详情请参见索引介绍。

最后更新: 2025年8月4日

导入 **Parquet** 文件数据

本文以一个示例说明如何使用 Exchange 将存储在 HDFS 或本地的 Parquet 文件数据导入 NebulaGraph 。

数据集

本文以 basketballplayer 数据集为例。

环境配置

本文示例在 MacOS 下完成，以下是相关的环境配置信息：

- 硬件规格：
- CPU：1.7 GHz Quad-Core Intel Core i7
- 内存：16 GB

- Spark：2.4.7 单机版

- Hadoop：2.9.2 伪分布式部署

- NebulaGraph ： master。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署 NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的的 IP 地址和端口。
- 拥有 NebulaGraph 写权限的用户名和密码。

- 已经编译 Exchange。详情请参见编译 Exchange。本示例中使用 Exchange 3.8.0。

- 已经安装 Spark。

- 了解 NebulaGraph 中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。

- 如果文件存储在 HDFS 上，需要确认 Hadoop 服务运行正常。

- 如果文件存储在本地且 NebulaGraph 是集群架构，需要在集群每台机器本地相同目录下放置文件。

操作步骤

步骤 **1**：在 **NebulaGraph** 中创建 **Schema**

分析 Parquet 文件中的数据，按以下步骤在 NebulaGraph 中创建 Schema：

1. 确认 Schema 要素。 NebulaGraph 中的 Schema 要素如下表所示。

要素	名称	属性
Tag	`player`	`name string, age int`
Tag	`team`	`name string`
Edge Type	`follow`	`degree int`
Edge Type	`serve`	`start_year int, end_year int`

2. 使用 NebulaGraph Console 创建一个图空间 basketballplayer，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
        (partition_num = 10, \
        replica_factor = 1, \
        vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见快速开始。

步骤 **2**：处理 **Parquet** 文件

确认以下信息：

1. 处理 Parquet 文件以满足 Schema 的要求。
2. 获取 Parquet 文件存储路径。

步骤 **3**：修改配置文件

编译 Exchange 后，复制 `target/classes/application.conf` 文件设置 Parquet 数据源相关的配置。在本示例中，复制的文件名为 `parquet_application.conf`。各个配置项的详细说明请参见配置说明。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.8.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    executor: {
        memory:1G
    }

    cores: {
      max: 16
    }
  }

  # NebulaGraph 相关配置
  nebula: {
    address:{
      # 指定 Graph 服务和所有 Meta 服务的 IP 地址和端口。
      # 如果有多台服务器，地址之间用英文逗号（,）分隔。
```

```
      # 格式："ip1:port","ip2:port","ip3:port"
      graph:["127.0.0.1:9669"]
      #任意一个 Meta 服务的地址。
      #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
      meta:["127.0.0.1:9559"]
    }

    # 指定拥有 NebulaGraph 写权限的用户名和密码。
    user: root
    pswd: nebula
    # 是否使用 RSA 加密的密码。
    # enableRSA: true
    # 使用 RSA 加密密码时的密钥。
    # privateKey: ""

    # 指定图空间名称。
    space: basketballplayer
    connection: {
      timeout: 3000
      retry: 3
    }
    execution: {
      retry: 3
    }
    error: {
      max: 32
      output: /tmp/errors
    }
    rate: {
      limit: 1024
      timeout: 1000
    }
  }

  # 处理点
  tags: [
    # 设置 Tag player 相关信息。
    {
      # 指定 NebulaGraph 中定义的 Tag 名称。
      name: player
      type: {
        # 指定数据源，使用 Parquet。
        source: parquet

        # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
        sink: client
      }

      # 指定 Parquet 文件的路径。
      # 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
      # 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.csv"。
      path: "hdfs://192.168.11.139000/data/vertex_player.parquet"

      # 在 fields 里指定 Parquet 文件中 key 名称，其对应的 value 会作为 NebulaGraph 中指定属性的数据源。
      # 如果需要指定多个值，用英文逗号（,）隔开。
      fields: [age,name]

      # 指定 NebulaGraph 中定义的属性名称。
      # fields 与 nebula.fields 的顺序必须一一对应。
      nebula.fields: [age, name]

      # 指定一个列作为 VID 的源。
      # vertex 的值必须与 Parquet 文件中的字段保持一致。
      # 目前，NebulaGraph  master仅支持字符串或整数类型的 VID。
      vertex: {
        field:id
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
      # filter: "name='Tom'"

      # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
      #writeMode: INSERT

      # 批量删除时是否删除该点关联的出边和入边。`writeMode`为`DELETE`时该参数生效。
      #deleteEdge: false

      # 指定单批次写入 NebulaGraph 的最大点数量。
      batch: 256

      # 数据写入 NebulaGraph 时需要创建的分区数。
      partition: 32
    }

    # 设置 Tag team 相关信息。
```

```
  {
    name: team
    type: {
      source: parquet
      sink: client
    }
    path: "hdfs://192.168.11.13:9000/data/vertex_team.parquet"
    fields: [name]
    nebula.fields: [name]
    vertex: {
      field:id
    }
    batch: 256
    partition: 32
  }

  # 如果需要添加更多点，请参考前面的配置进行添加。
]
# 处理边
edges: [
  # 设置 Edge type follow 相关信息。
  {
    # 指定 NebulaGraph 中定义的 Edge type 名称。
    name: follow
    type: {
      # 指定数据源，使用 Parquet。
      source: parquet

      # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
      sink: client
    }

    # 指定 Parquet 文件的路径。
    # 如果文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx"。
    # 如果文件存储在本地，用双引号括起路径，以 file://开头，例如"file:///tmp/xx.csv"。
    path: "hdfs://192.168.11.13:9000/data/edge_follow.parquet"

    # 在 fields 里指定 Parquet 文件中 key 名称，其对应的 value 会作为 NebulaGraph 中指定属性的数据源。
    # 如果需要指定多个值，用英文逗号（,）隔开。
    fields: [degree]

    # 指定 NebulaGraph 中定义的属性名称。
    # fields 与 nebula.fields 的顺序必须一一对应。
    nebula.fields: [degree]

    # 指定一个列作为起始点和目的点的源。
    # vertex 的值必须与 Parquet 文件中的字段保持一致。
    # 目前，NebulaGraph  master仅支持字符串或整数类型的 VID。
    source: {
      field: src
    # udf:{
    #            separator:"_"
    #            oldColNames:[field-0,field-1,field-2]
    #            newColName:new-field
    #        }
    # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
    # prefix:"tag1"
    # 对 string 类型的 VID 进行哈希化操作。
    # policy:hash
    }
    target: {
      field: dst
    # udf:{
    #            separator:"_"
    #            oldColNames:[field-0,field-1,field-2]
    #            newColName:new-field
    #        }
    # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
    # prefix:"tag1"
    # 对 string 类型的 VID 进行哈希化操作。
    # policy:hash
    }

    # 指定一个列作为 rank 的源（可选）。
    #ranking: rank

    # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
    # filter: "name='Tom'"

    # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
    #writeMode: INSERT

    # 指定单批次写入 NebulaGraph 的最大边数量。
    batch: 256

    # 数据写入 NebulaGraph 时需要创建的分区数。
    partition: 32
  }

  # 设置 Edge type serve 相关信息。
  {
    name: serve
    type: {
      source: parquet
```

```
      sink: client
    }
    path: "hdfs://192.168.11.13:9000/data/edge_serve.parquet"
    fields: [start_year,end_year]
    nebula.fields: [start_year, end_year]
    source: {
      field: src
    }
    target: {
      field: dst
    }
    batch: 256
    partition: 32
  }

 ]
 # 如果需要添加更多边，请参考前面的配置进行添加。
}
```

步骤 **4**：向 **NebulaGraph** 导入数据

运行如下命令将 Parquet 文件数据导入到 NebulaGraph 中。关于参数的说明，请参见导入命令参数。

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange.jar_path> -c <parquet_application.conf_path>
```

> Note
>
> JAR 包有两种获取方式：自行编译或者从 maven 仓库下载。

示例：

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange_spark_2.4-3.8.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/parquet_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`，确认成功的数量。例如 `batchSuccess.follow: 300`。

访问 Kerberos 认证的 HDFS

使用 Kerberos 进行安全认证时，需使用以下两种方式之一访问 Kerberos 认证的 HDFS。

- 在命令中设置 Kerberos 配置文件

  在命令中配置 `--conf` 和 `--files`，例如：

```
${SPARK_HOME}/bin/spark-submit --master xxx --num-executors 2 --executor-cores 2 --executor-memory 1g \
--conf "spark.driver.extraJavaOptions=-Djava.security.krb5.conf=./krb5.conf" \
--conf "spark.executor.extraJavaOptions=-Djava.security.krb5.conf=./krb5.conf" \
--files /local/path/to/xxx.keytab,/local/path/to/krb5.conf \
--class com.vesoft.nebula.exchange.Exchange \
exchange.jar -c xx.conf
```

  `--conf` 中的文件路径有如下两种配置方式：

- 配置文件的绝对路径。要求所有 YARN 或者 Spark 机器相同路径下都有对应文件。
- （YARN 模式下推荐）配置文件的相对路径（例如 `./krb5.conf`）。通过 `--files` 上传的资源文件就在 Java 虚拟机或者 JAR 的工作目录下。

  `--files` 中的文件必须存储在执行 `spark-submit` 命令的机器上。

- 不使用命令

  将 Spark 和 Kerberos 认证的 Hadoop 部署在相同集群内，共用 HDFS 和 YARN，然后在 Spark 的 `spark-env.sh` 中增加配置 `export HADOOP_HOME=<hadoop_home_path>`。

步骤 **5**：（可选）验证数据

用户可以在 NebulaGraph 客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 **6**：（如有）在 **NebulaGraph** 中重建索引

导入数据后，用户可以在 NebulaGraph 中重新创建并重建索引。详情请参见索引介绍。

最后更新: 2025年8月4日

导入 HBase 数据

本文以一个示例说明如何使用 Exchange 将存储在 HBase 上的数据导入 NebulaGraph 。

数据集

本文以 basketballplayer 数据集为例。

在本示例中，该数据集已经存入 HBase 中，以 `player` 、 `team` 、 `follow` 和 `serve` 四个表存储了所有点和边的信息。以下为各个表的部分数据。

```
hbase(main):002:0> scan "player"
ROW                  COLUMN+CELL
 player100              column=cf:age, timestamp=1618881347530, value=42
 player100              column=cf:name, timestamp=1618881354604, value=Tim Duncan
 player101              column=cf:age, timestamp=1618881369124, value=36
 player101              column=cf:name, timestamp=1618881379102, value=Tony Parker
 player102              column=cf:age, timestamp=1618881386987, value=33
 player102              column=cf:name, timestamp=1618881393370, value=LaMarcus Aldridge
 player103              column=cf:age, timestamp=1618881402002, value=32
 player103              column=cf:name, timestamp=1618881407882, value=Rudy Gay
 ...

hbase(main):003:0> scan "team"
ROW                  COLUMN+CELL
 team200               column=cf:name, timestamp=1618881445563, value=Warriors
 team201               column=cf:name, timestamp=1618881453636, value=Nuggets
 ...

hbase(main):004:0> scan "follow"
ROW                  COLUMN+CELL
 player100              column=cf:degree, timestamp=1618881804853, value=95
 player100              column=cf:dst_player, timestamp=1618881791522, value=player101
 player101              column=cf:degree, timestamp=1618881824685, value=90
 player101              column=cf:dst_player, timestamp=1618881816042, value=player102
 ...

hbase(main):005:0> scan "serve"
ROW                  COLUMN+CELL
 player100              column=cf:end_year, timestamp=1618881899333, value=2016
 player100              column=cf:start_year, timestamp=1618881890117, value=1997
 player100              column=cf:teamid, timestamp=1618881875739, value=team204
 ...
```

环境配置

本文示例在 MacOS 下完成，以下是相关的环境配置信息：

- 硬件规格：
- CPU：1.7 GHz Quad-Core Intel Core i7
- 内存：16 GB
- Spark：2.4.7，单机版
- HBase：2.2.7
- NebulaGraph ： master。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署 NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的的 IP 地址和端口。
- 拥有 NebulaGraph 写权限的用户名和密码。
- 已经编译 Exchange。详情请参见编译 Exchange。本示例中使用 Exchange 3.8.0。
- 已经安装 Spark。
- 了解 NebulaGraph 中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。

操作步骤

步骤 **1**：在 **NebulaGraph** 中创建 **Schema**

分析数据，按以下步骤在 NebulaGraph 中创建 Schema：

1. 确认 Schema 要素。 NebulaGraph 中的 Schema 要素如下表所示。

要素	名称	属性
Tag	`player`	`name string, age int`
Tag	`team`	`name string`
Edge Type	`follow`	`degree int`
Edge Type	`serve`	`start_year int, end_year int`

2. 在 NebulaGraph 中创建一个图空间 basketballplayer，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
        (partition_num = 10, \
        replica_factor = 1, \
        vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见快速开始。

步骤 **2**：修改配置文件

编译 Exchange 后，复制 `target/classes/application.conf` 文件设置 HBase 数据源相关的配置。在本示例中，复制的文件名为 `hbase_application.conf` 。各个配置项的详细说明请参见配置说明。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.8.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    cores: {
      max: 16
    }
  }

  # NebulaGraph 相关配置
  nebula: {
    address:{
      # 以下为 NebulaGraph 的 Graph 服务和 Meta 服务所在机器的 IP 地址及端口。
      # 如果有多个地址，格式为 "ip1:port","ip2:port","ip3:port"。
      # 不同地址之间以英文逗号 (,) 隔开。
      graph:["127.0.0.1:9669"]
      #任意一个 Meta 服务的地址。
      #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
      meta:["127.0.0.1:9559"]
    }
    # 填写的账号必须拥有 NebulaGraph 相应图空间的写数据权限。
    user: root
    pswd: nebula
    # 是否使用 RSA 加密的密码。
    # enableRSA: true
    # 使用 RSA 加密密码时的密钥。
    # privateKey: ""
    # 填写 NebulaGraph 中需要写入数据的图空间名称。
    space: basketballplayer
```

```
  connection: {
    timeout: 3000
    retry: 3
  }
  execution: {
    retry: 3
  }
  error: {
    max: 32
    output: /tmp/errors
  }
  rate: {
    limit: 1024
    timeout: 1000
  }
}
# 处理点
tags: [
  # 设置 Tag player 相关信息。
  # 如果需要将 rowkey 设置为数据源，请填写“rowkey”，列族内的列请填写实际列名。
  {
    # NebulaGraph 中对应的 Tag 名称。
    name: player
    type: {
      # 指定数据源文件格式，设置为 HBase。
      source: hbase
      # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
      sink: client
    }
    host:192.168.*.*
    port:2181
    table:"player"
    columnFamily:"cf"

    # 在 fields 里指定 player 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
    # fields 和 nebula.fields 里的配置必须一一对应。
    # 如果需要指定多个列名称，用英文逗号（,）隔开。
    fields: [age,name]
    nebula.fields: [age,name]

    # 指定表中某一列数据为 NebulaGraph 中点 VID 的来源。
    # 例如 rowkey 作为 VID 的来源，请填写“rowkey”。
    vertex:{
        field:rowkey
    # udf:{
    #            separator:"_"
    #            oldColNames:[field-0,field-1,field-2]
    #            newColName:new-field
    #        }
    # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
    # prefix:"tag1"
    # 对 string 类型的 VID 进行哈希化操作。
    # policy:hash
    }

    # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
    # filter: "name='Tom'"

    # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
    #writeMode: INSERT

    # 批量删除时是否删除该点关联的出边和入边。`writeMode`为`DELETE`时该参数生效。
    #deleteEdge: false

    # 单批次写入 NebulaGraph 的数据条数。
    batch: 256

    # 数据写入 NebulaGraph 时需要创建的分区数。
    partition: 32
  }
  # 设置 Tag team 相关信息。
  {
    name: team
    type: {
      source: hbase
      sink: client
    }
    host:192.168.*.*
    port:2181
    table:"team"
    columnFamily:"cf"
    fields: [name]
    nebula.fields: [name]
    vertex:{
        field:rowkey
    }
    batch: 256
    partition: 32
  }

]

# 处理边数据
edges: [
```

```
# 设置 Edge type follow 相关信息
{
  # NebulaGraph 中对应的 Edge type 名称。
  name: follow

  type: {
    # 指定数据源文件格式，设置为 HBase。
    source: hbase

    # 指定边数据导入 NebulaGraph 的方式，
    # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
    sink: client
  }

  host:192.168.*.*
  port:2181
  table:"follow"
  columnFamily:"cf"

  # 在 fields 里指定 follow 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
  # fields 和 nebula.fields 里的配置必须一一对应。
  # 如果需要指定多个列名称，用英文逗号（,）隔开。
  fields: [degree]
  nebula.fields: [degree]

  # 在 source 里，将 follow 表中某一列作为边的起始点数据源。示例使用 rowkey。
  # 在 target 里，将 follow 表中某一列作为边的目的点数据源。示例使用列 dst_player。
  source:{
      field:rowkey
  # udf:{
  #            separator:"_"
  #            oldColNames:[field-0,field-1,field-2]
  #            newColName:new-field
  #        }
  # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
  # prefix:"tag1"
  # 对 string 类型的 VID 进行哈希化操作。
  # policy:hash
  }

  target:{
      field:dst_player
  # udf:{
  #            separator:"_"
  #            oldColNames:[field-0,field-1,field-2]
  #            newColName:new-field
  #        }
  # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
  # prefix:"tag1"
  # 对 string 类型的 VID 进行哈希化操作。
  # policy:hash
  }

  # 指定一个列作为 rank 的源（可选）。
  #ranking: rank

  # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
  # filter: "name='Tom'"

  # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
  #writeMode: INSERT

  # 单批次写入 NebulaGraph 的数据条数。
  batch: 256

  # 数据写入 NebulaGraph 时需要创建的分区数。
  partition: 32
}

# 设置 Edge type serve 相关信息
{
  name: serve
  type: {
    source: hbase
    sink: client
  }
  host:192.168.*.*
  port:2181
  table:"serve"
  columnFamily:"cf"

  fields: [start_year,end_year]
  nebula.fields: [start_year,end_year]
  source:{
      field:rowkey
  }

  target:{
      field:teamid
  }

  # 指定一个列作为 rank 的源（可选）。
  #ranking: rank
```

```
      batch: 256
      partition: 32
    }
  ]
}
```

步骤 3：向 NebulaGraph 导入数据

运行如下命令将 HBase 数据导入到 NebulaGraph 中。关于参数的说明，请参见导入命令参数。

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange.jar_path> -c <hbase_application.conf_path>
```

> **Note**
>
> JAR 包有两种获取方式：自行编译或者从 maven 仓库下载。

示例：

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange_spark_2.4-3.8.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/hbase_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>` ，确认成功的数量。例如 `batchSuccess.follow: 300` 。

步骤 4：（可选）验证数据

用户可以在 NebulaGraph 客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 5：（如有）在 NebulaGraph 中重建索引

导入数据后，用户可以在 NebulaGraph 中重新创建并重建索引。详情请参见索引介绍。

最后更新: 2025年8月4日

导入 MySQL/PostgreSQL 数据

本文以一个示例说明如何使用 Exchange 将存储在 MySQL 上的数据导入 NebulaGraph，也适用于从 PostgreSQL 导出数据到 NebulaGraph 。

数据集

本文以 basketballplayer 数据集为例。

在本示例中，该数据集已经存入 MySQL 中名为 `basketball` 的数据库中，以 `player` 、 `team` 、 `follow` 和 `serve` 四个表存储了所有点和边的信息。以下为各个表的结构。

```
mysql> desc player;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| playerid | varchar(30) | YES  |     | NULL    |       |
| age      | int         | YES  |     | NULL    |       |
| name     | varchar(30) | YES  |     | NULL    |       |
+----------+-------------+------+-----+---------+-------+

mysql> desc team;
+--------+-------------+------+-----+---------+-------+
| Field  | Type        | Null | Key | Default | Extra |
+--------+-------------+------+-----+---------+-------+
| teamid | varchar(30) | YES  |     | NULL    |       |
| name   | varchar(30) | YES  |     | NULL    |       |
+--------+-------------+------+-----+---------+-------+

mysql> desc follow;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| src_player | varchar(30) | YES  |     | NULL    |       |
| dst_player | varchar(30) | YES  |     | NULL    |       |
| degree     | int         | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+

mysql> desc serve;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| playerid   | varchar(30) | YES  |     | NULL    |       |
| teamid     | varchar(30) | YES  |     | NULL    |       |
| start_year | int         | YES  |     | NULL    |       |
| end_year   | int         | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
```

环境配置

本文示例在 MacOS 下完成，以下是相关的环境配置信息：

- 硬件规格：
- CPU：1.7 GHz Quad-Core Intel Core i7
- 内存：16 GB

- Spark：2.4.7，单机版

- MySQL： 8.0.23

- NebulaGraph ： master。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署 NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的的 IP 地址和端口。
- 拥有 NebulaGraph 写权限的用户名和密码。

- 已经编译 Exchange。详情请参见编译 Exchange。本示例中使用 Exchange 3.8.0。
- 已经安装 Spark。
- mysql-connector-java-xxx.jar 已经下载并放置在 Spark 的 `SPARK_HOME/jars` 目录下。
- 了解 NebulaGraph 中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。
- 已经安装并开启 Hadoop 服务。

注意事项

nebula-exchange_spark_2.2 仅支持单表查询，不支持多表查询。

操作步骤

步骤 1：在 NebulaGraph 中创建 Schema

分析数据，按以下步骤在 NebulaGraph 中创建 Schema：

1. 确认 Schema 要素。 NebulaGraph 中的 Schema 要素如下表所示。

要素	名称	属性
Tag	`player`	`name string, age int`
Tag	`team`	`name string`
Edge Type	`follow`	`degree int`
Edge Type	`serve`	`start_year int, end_year int`

2. 在 NebulaGraph 中创建一个图空间 basketballplayer，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
        (partition_num = 10, \
        replica_factor = 1, \
        vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见快速开始。

步骤 2：修改配置文件

编译 Exchange 后，复制 `target/classes/application.conf` 文件设置 MySQL 数据源相关的配置。在本示例中，复制的文件名为 `mysql_application.conf`。各个配置项的详细说明请参见配置说明。

```
{
  # Spark 相关配置
  spark: {
```

```
    app: {
      name: NebulaGraph Exchange 3.8.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    cores: {
      max: 16
    }
  }

# NebulaGraph 相关配置
  nebula: {
    address:{
      # 以下为 NebulaGraph 的 Graph 服务和 Meta 服务所在机器的 IP 地址及端口。
      # 如果有多个地址，格式为 "ip1:port","ip2:port","ip3:port"。
      # 不同地址之间以英文逗号 (,) 隔开。
      graph:["127.0.0.1:9669"]
      #任意一个 Meta 服务的地址。
      #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
      meta:["127.0.0.1:9559"]
    }
    # 填写的账号必须拥有 NebulaGraph 相应图空间的写数据权限。
    user: root
    pswd: nebula
    # 是否使用 RSA 加密的密码。
    # enableRSA: true
    # 使用 RSA 加密密码时的密钥。
    # privateKey: ""
    # 填写 NebulaGraph 中需要写入数据的图空间名称。
    space: basketballplayer
    connection: {
      timeout: 3000
      retry: 3
    }
    execution: {
      retry: 3
    }
    error: {
      max: 32
      output: /tmp/errors
    }
    rate: {
      limit: 1024
      timeout: 1000
    }
  }
  # 处理点
  tags: [
    # 设置 Tag player 相关信息。
    {
      # NebulaGraph 中对应的 Tag 名称。
      name: player
      type: {
        # 指定数据源文件格式，设置为 MySQL。
        source: mysql
        # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
        sink: client
      }

      host:192.168.*.*
      port:3306
      user:"test"
      password:"123456"
      database:"basketball"

      # 扫描单个表读取数据。
      # nebula-exchange_spark_2.2 必须配置该参数。不支持配置 sentence。
      # nebula-exchange_spark_2.4 和 nebula-exchange_spark_3.0 可以配置该参数，但是不能和 sentence 同时配置。
      table:"basketball.player"

      # 通过查询语句读取数据。
      # nebula-exchange_spark_2.2 不支持该参数。
      # nebula-exchange_spark_2.4 和 nebula-exchange_spark_3.0 可以配置该参数，但是不能和 table 同时配置。支持多表查询。
      # sentence: "select * from  people, player, team"

      # 在 fields 里指定 player 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
      # fields 和 nebula.fields 里的配置必须一一对应。
      # 如果需要指定多个列名称，用英文逗号 (,) 隔开。
      fields: [age,name]
      nebula.fields: [age,name]

      # 指定表中某一列数据为 NebulaGraph 中点 VID 的来源。
      vertex: {
        field:playerid
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
```

```
      # policy:hash
      }

      # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
      # filter: "name='Tom'"

      # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
      #writeMode: INSERT

      # 批量删除时是否删除该点关联的出边和入边。`writeMode`为`DELETE`时该参数生效。
      #deleteEdge: false

      # 单批次写入 NebulaGraph 的数据条数。
      batch: 256

      # 数据写入 NebulaGraph 时需要创建的分区数。
      partition: 32
    }
    # 设置 Tag team 相关信息。
    {
      name: team
      type: {
        source: mysql
        sink: client
      }

      host:192.168.*.*
      port:3306
      database:"basketball"
      table:"team"
      user:"test"
      password:"123456"
      sentence:"select teamid, name from team order by teamid"

      fields: [name]
      nebula.fields: [name]
      vertex: {
        field: teamid
      }
      batch: 256
      partition: 32
    }

  ]

  # 处理边数据
  edges: [
    # 设置 Edge type follow 相关信息
    {
      # NebulaGraph 中对应的 Edge type 名称。
      name: follow

      type: {
        # 指定数据源文件格式，设置为 MySQL。
        source: mysql

        # 指定边数据导入 NebulaGraph 的方式，
        # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
        sink: client
      }

      host:192.168.*.*
      port:3306
      user:"test"
      password:"123456"
      database:"basketball"

      # 扫描单个表读取数据。
      # nebula-exchange_spark_2.2 必须配置该参数。不支持配置 sentence。
      # nebula-exchange_spark_2.4 和 nebula-exchange_spark_3.0 可以配置该参数，但是不能和 sentence 同时配置。
      table:"basketball.follow"

      # 通过查询语句读取数据。
      # nebula-exchange_spark_2.2 不支持该参数。
      # nebula-exchange_spark_2.4 和 nebula-exchange_spark_3.0 可以配置该参数，但是不能和 table 同时配置。支持多表查询。
      # sentence: "select * from  follow, serve"

      # 在 fields 里指定 follow 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
      # fields 和 nebula.fields 里的配置必须一一对应。
      # 如果需要指定多个列名称，用英文逗号（,）隔开。
      fields: [degree]
      nebula.fields: [degree]

      # 在 source 里，将 follow 表中某一列作为边的起始点数据源。
      # 在 target 里，将 follow 表中某一列作为边的目的点数据源。
      source: {
        field: src_player
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
```

```
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      target: {
        field: dst_player
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      # 指定一个列作为 rank 的源（可选）。
      #ranking: rank

      # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
      # filter: "name='Tom'"

      # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
      #writeMode: INSERT

      # 单批次写入 NebulaGraph 的数据条数。
      batch: 256

      # 数据写入 NebulaGraph 时需要创建的分区数。
      partition: 32
    }

    # 设置 Edge type serve 相关信息
    {
      name: serve
      type: {
        source: mysql
        sink: client
      }

      host:192.168.*.*
      port:3306
      database:"basketball"
      table:"serve"
      user:"test"
      password:"123456"
      sentence:"select playerid,teamid,start_year,end_year from serve order by playerid"
      fields: [start_year,end_year]
      nebula.fields: [start_year,end_year]
      source: {
        field: playerid
      }
      target: {
        field: teamid
      }

      # 指定一个列作为 rank 的源（可选）。
      #ranking: rank

      batch: 256
      partition: 32
    }
  ]
}
```

步骤 3：向 **NebulaGraph** 导入数据

运行如下命令将 MySQL 数据导入到 NebulaGraph 中。关于参数的说明，请参见导入命令参数。

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange.jar_path> -c <mysql_application.conf_path>
```

Note

JAR 包有两种获取方式：自行编译或者从 maven 仓库下载。

示例：

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange_spark_2.4-3.8.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/mysql_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>` ，确认成功的数量。例如 `batchSuccess.follow: 300` 。

步骤 **4**：（可选）验证数据

用户可以在 NebulaGraph 客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 **5**：（如有）在 **NebulaGraph** 中重建索引

导入数据后，用户可以在 NebulaGraph 中重新创建并重建索引。详情请参见索引介绍。

最后更新: 2025年8月4日

导入 Oracle 数据

本文以一个示例说明如何使用 Exchange 将存储在 Oracle 上的数据导入 NebulaGraph 。

数据集

本文以 basketballplayer 数据集为例。

在本示例中，该数据集已经存入 Oracle 中名为 `basketball` 的数据库中，以 `player` 、 `team` 、 `follow` 和 `serve` 四个表存储了所有点和边的信息。以下为各个表的结构。

```
oracle> desc player;
+-----------+-------+---------------+
| Column    | Null  | Type          |
+-----------+-------+---------------+
| PLAYERID  | -     | VARCHAR2(30)  |
| NAME      | -     | VARCHAR2(30)  |
| AGE       | -     | NUMBER        |
+-----------+-------+---------------+

oracle> desc team;
+-----------+-------+---------------+
| Column    | Null  | Type          |
+-----------+-------+---------------+
| TEAMID    | -     | VARCHAR2(30)  |
| NAME      | -     | VARCHAR2(30)  |
+-----------+-------+---------------+

oracle> desc follow;
+-------------+-------+---------------+
| Column      | Null  | Type          |
+-------------+-------+---------------+
| SRC_PLAYER  | -     | VARCHAR2(30)  |
| DST_PLAYER  | -     | VARCHAR2(30)  |
| DEGREE      | -     | NUMBER        |
+-------------+-------+---------------+

oracle> desc serve;
+------------+-------+---------------+
| Column     | Null  | Type          |
+------------+-------+---------------+
| PLAYERID   | -     | VARCHAR2(30)  |
| TEAMID     | -     | VARCHAR2(30)  |
| START_YEAR | -     | NUMBER        |
| END_YEAR   | -     | NUMBER        |
+------------+-------+---------------+
```

环境配置

本文示例在 MacOS 下完成，以下是相关的环境配置信息：

- 硬件规格：
- CPU：1.7 GHz Quad-Core Intel Core i7
- 内存：16 GB

- Spark：2.4.7，单机版

- NebulaGraph ： master。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署 NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的的 IP 地址和端口。
- 拥有 NebulaGraph 写权限的用户名和密码。

- 已经编译 Exchange。详情请参见编译 Exchange。本示例中使用 Exchange 3.8.0。

- 已经安装 Spark。

- 了解 NebulaGraph 中创建的 Schema 信息，包括 Tag 和 Edge type 的名称、属性等。

- 已经安装并开启 Hadoop 服务。

注意事项

nebula-exchange_spark_2.2 仅支持单表查询，不支持多表查询。

操作步骤

步骤 **1**：在 **NebulaGraph** 中创建 **Schema**

分析数据，按以下步骤在 NebulaGraph 中创建 Schema：

1. 确认 Schema 要素。 NebulaGraph 中的 Schema 要素如下表所示。

要素	名称	属性
Tag	`player`	`name string, age int`
Tag	`team`	`name string`
Edge Type	`follow`	`degree int`
Edge Type	`serve`	`start_year int, end_year int`

2. 在 NebulaGraph 中创建一个图空间 basketballplayer，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
        (partition_num = 10, \
        replica_factor = 1, \
        vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见快速开始。

步骤 **2**：修改配置文件

编译 Exchange 后，复制 `target/classes/application.conf` 文件设置 Oracle 数据源相关的配置。在本示例中，复制的文件名为 `oracle_application.conf` 。各个配置项的详细说明请参见配置说明。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.8.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    cores: {
      max: 16
    }
  }

# NebulaGraph 相关配置
  nebula: {
    address:{
      # 以下为 NebulaGraph 的 Graph 服务和 Meta 服务所在机器的 IP 地址及端口。
      # 如果有多个地址，格式为 "ip1:port","ip2:port","ip3:port"。
      # 不同地址之间以英文逗号 (,) 隔开。
      graph:["127.0.0.1:9669"]
      #任意一个 Meta 服务的地址。
      #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
      meta:["127.0.0.1:9559"]
    }
    # 填写的账号必须拥有 NebulaGraph 相应图空间的写数据权限。
    user: root
    pswd: nebula
    # 是否使用 RSA 加密的密码。
```

```
    # enableRSA: true
    # 使用 RSA 加密密码时的密钥。
    # privateKey: ""
    # 填写 NebulaGraph 中需要写入数据的图空间名称。
    space: basketballplayer
    connection: {
      timeout: 3000
      retry: 3
    }
    execution: {
      retry: 3
    }
    error: {
      max: 32
      output: /tmp/errors
    }
    rate: {
      limit: 1024
      timeout: 1000
    }
  }
  # 处理点
  tags: [
    {
      # NebulaGraph 中对应的 Tag 名称。
      name: player
      type: {
        # 指定数据源文件格式，设置为 Oracle。
        source: oracle
        # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
        sink: client
      }


      url:"jdbc:oracle:thin:@host:1521:basketball"
      driver: "oracle.jdbc.driver.OracleDriver"
      user: "root"
      password: "123456"

      # 扫描单个表读取数据。
      # nebula-exchange_spark_2.2 必须配置该参数。不支持配置 sentence。
      # nebula-exchange_spark_2.4 和 nebula-exchange_spark_3.0 可以配置该参数，但是不能和 sentence 同时配置。
      table:"basketball.player"

      # 通过查询语句读取数据。
      # nebula-exchange_spark_2.2 不支持该参数。
      # nebula-exchange_spark_2.4 和 nebula-exchange_spark_3.0 可以配置该参数，但是不能和 table 同时配置。支持多表查询。
      # sentence: "select * from  people, player, team"

      # 在 fields 里指定 player 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
      # fields 和 nebula.fields 里的配置必须一一对应。
      # 如果需要指定多个列名称，用英文逗号（,）隔开。
      fields: [age,name]
      nebula.fields: [age,name]

      # 指定表中某一列数据为 NebulaGraph 中点 VID 的来源。
      vertex: {
        field:playerid
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
      # filter: "name='Tom'"

      # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
      #writeMode: INSERT

      # 批量删除时是否删除该点关联的出边和入边。`writeMode`为`DELETE`时该参数生效。
      #deleteEdge: false

      # 单批次写入 NebulaGraph 的数据条数。
      batch: 256

      # 数据写入 NebulaGraph 时需要创建的分区数。
      partition: 32
    }
    # 设置 Tag team 相关信息。
    {
      name: team
      type: {
        source: oracle
        sink: client
      }

      url:"jdbc:oracle:thin:@host:1521:db"
      driver: "oracle.jdbc.driver.OracleDriver"
```

```
    user: "root"
    password: "123456"
    table: "basketball.team"
    sentence: "select teamid, name from team"

    fields: [name]
    nebula.fields: [name]
    vertex: {
      field: teamid
    }
    batch: 256
    partition: 32
  }

]

# 处理边数据
edges: [
  # 设置 Edge type follow 相关信息
  {
    #  NebulaGraph 中对应的 Edge type 名称。
    name: follow

    type: {
      # 指定数据源文件格式，设置为 Oracle。
      source: oracle

      # 指定边数据导入 NebulaGraph 的方式，
      # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
      sink: client
    }

    url:"jdbc:oracle:thin:@host:1521:db"
    driver: "oracle.jdbc.driver.OracleDriver"
    user: "root"
    password: "123456"

    # 扫描单个表读取数据。
    # nebula-exchange_spark_2.2 必须配置该参数。不支持配置 sentence。
    # nebula-exchange_spark_2.4 和 nebula-exchange_spark_3.0 可以配置该参数，但是不能和 sentence 同时配置。
    table:"basketball.follow"

    # 通过查询语句读取数据。
    # nebula-exchange_spark_2.2 不支持该参数。
    # nebula-exchange_spark_2.4 和 nebula-exchange_spark_3.0 可以配置该参数，但是不能和 table 同时配置。支持多表查询。
    # sentence: "select * from  follow, serve"

    # 在 fields 里指定 follow 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
    # fields 和 nebula.fields 里的配置必须一一对应。
    # 如果需要指定多个列名称，用英文逗号（,）隔开。
    fields: [degree]
    nebula.fields: [degree]

    # 在 source 里，将 follow 表中某一列作为边的起始点数据源。
    # 在 target 里，将 follow 表中某一列作为边的目的点数据源。
    source: {
      field: src_player
    # udf:{
    #            separator:"_"
    #            oldColNames:[field-0,field-1,field-2]
    #            newColName:new-field
    #        }
    # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
    # prefix:"tag1"
    # 对 string 类型的 VID 进行哈希化操作。
    # policy:hash
    }

    target: {
      field: dst_player
    # udf:{
    #            separator:"_"
    #            oldColNames:[field-0,field-1,field-2]
    #            newColName:new-field
    #        }
    # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
    # prefix:"tag1"
    # 对 string 类型的 VID 进行哈希化操作。
    # policy:hash
    }

    # 指定一个列作为 rank 的源（可选）。
    #ranking: rank

    # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
    # filter: "name='Tom'"

    # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
    #writeMode: INSERT

    # 单批次写入 NebulaGraph 的数据条数。
    batch: 256

    # 数据写入 NebulaGraph 时需要创建的分区数。
```

```
      partition: 32
    }

    # 设置 Edge type serve 相关信息
    {
      name: serve
      type: {
        source: oracle
        sink: client
      }

      url:"jdbc:oracle:thin:@host:1521:db"
      driver: "oracle.jdbc.driver.OracleDriver"
      user: "root"
      password: "123456"
      table: "basketball.serve"
      sentence: "select playerid, teamid, start_year, end_year from serve"

      fields: [start_year,end_year]
      nebula.fields: [start_year,end_year]
      source: {
        field: playerid
      }
      target: {
        field: teamid
      }

      # 指定一个列作为 rank 的源（可选）。
      #ranking: rank

      batch: 256
      partition: 32
    }
  ]
}
```

步骤 **3**：向 **NebulaGraph** 导入数据

运行如下命令将 Oracle 数据导入到 NebulaGraph 中。关于参数的说明，请参见导入命令参数。

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange.jar_path> -c <oracle_application.conf_path>
```

> **Note**
>
> JAR 包有两种获取方式：自行编译或者从 maven 仓库下载。

示例：

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange_spark_2.4-3.8.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/oracle_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`，确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 **4**：（可选）验证数据

用户可以在 NebulaGraph 客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 **5**：（如有）在 **NebulaGraph** 中重建索引

导入数据后，用户可以在 NebulaGraph 中重新创建并重建索引。详情请参见索引介绍。

最后更新: 2025年8月4日

导入 **ClickHouse** 数据

本文以一个示例说明如何使用 Exchange 将存储在 ClickHouse 上的数据导入 NebulaGraph 。

数据集

本文以 basketballplayer 数据集为例。

环境配置

本文示例在 MacOS 下完成，以下是相关的环境配置信息：

- 硬件规格：
- CPU：1.7 GHz Quad-Core Intel Core i7
- 内存：16 GB

- Spark：2.4.7，单机版

- ClickHouse：docker 部署 yandex/clickhouse-server tag: latest(2021.07.01)

- NebulaGraph： master。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署 NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的的 IP 地址和端口。
- 拥有 NebulaGraph 写权限的用户名和密码。

- 已经编译 Exchange。详情请参见编译 Exchange。本示例中使用 Exchange 3.8.0。

- 已经安装 Spark。

- 了解 NebulaGraph 中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。

操作步骤

步骤 **1**：在 **NebulaGraph** 中创建 **Schema**

分析数据，按以下步骤在 NebulaGraph 中创建 Schema：

1. 确认 Schema 要素。 NebulaGraph 中的 Schema 要素如下表所示。

要素	名称	属性
Tag	`player`	`name string, age int`
Tag	`team`	`name string`
Edge Type	`follow`	`degree int`
Edge Type	`serve`	`start_year int, end_year int`

2. 在 NebulaGraph 中创建一个图空间 basketballplayer，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
        (partition_num = 10, \
        replica_factor = 1, \
        vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见快速开始。

步骤 **2**：修改配置文件

编译 Exchange 后，复制 `target/classes/application.conf` 文件设置 ClickHouse 数据源相关的配置。在本示例中，复制的文件名为 `clickhouse_application.conf`。各个配置项的详细说明请参见配置说明。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: Nebula Exchange 3.8.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    cores: {
      max: 16
    }
  }

#  NebulaGraph 相关配置
  nebula: {
    address:{
      # 以下为 NebulaGraph 的 Graph 服务和 Meta 服务所在机器的 IP 地址及端口。
      # 如果有多个地址，格式为 "ip1:port","ip2:port","ip3:port"。
      # 不同地址之间以英文逗号 (,) 隔开。
      graph:["127.0.0.1:9669"]

      #任意一个 Meta 服务的地址。
      #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
      meta:["127.0.0.1:9559"]
    }
    # 填写的账号必须拥有 NebulaGraph 相应图空间的写数据权限。
    user: root
    pswd: nebula
    # 是否使用 RSA 加密的密码。
    # enableRSA: true
    # 使用 RSA 加密密码时的密钥。
    # privateKey: ""
    # 填写 NebulaGraph 中需要写入数据的图空间名称。
```

```
    space: basketballplayer
    connection: {
      timeout: 3000
      retry: 3
    }
    execution: {
      retry: 3
    }
    error: {
      max: 32
      output: /tmp/errors
    }
    rate: {
      limit: 1024
      timeout: 1000
    }
  }
  # 处理点
  tags: [
    # 设置 Tag player 相关信息。
    {
      name: player
      type: {
        # 指定数据源文件格式，设置为 ClickHouse。
        source: clickhouse
        # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
        sink: client
      }

      # ClickHouse 的 JDBC URL
      url:"jdbc:clickhouse://192.168.*.*:8123/basketballplayer"

      user:"user"
      password:"123456"

      # ClickHouse 分区数
      numPartition:"5"

      table:"player"
      sentence:"select * from player"

      # 在 fields 里指定 player 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
      # fields 和 nebula.fields 里的配置必须一一对应。
      # 如果需要指定多个列名称，用英文逗号（,）隔开。
      fields: [name,age]
      nebula.fields: [name,age]

      # 指定表中某一列数据为 NebulaGraph 中点 VID 的来源。
      vertex: {
        field:playerid
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
      # filter: "name='Tom'"

      # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
      #writeMode: INSERT

      # 批量删除时是否删除该点关联的出边和入边。`writeMode`为`DELETE`时该参数生效。
      #deleteEdge: false

      # 单批次写入 NebulaGraph 的数据条数。
      batch: 256

      # 数据写入 NebulaGraph 时需要创建的分区数。
      partition: 32
    }

    # 设置 Tag team 相关信息。
    {
      name: team
      type: {
        source: clickhouse
        sink: client
      }
      url:"jdbc:clickhouse://192.168.*.*:8123/basketballplayer"
      user:"user"
      password:"123456"
      numPartition:"5"
      table:"team"
      sentence:"select * from team"
      fields: [name]
      nebula.fields: [name]
      vertex: {
        field:teamid
```

```
      }

      batch: 256
      partition: 32
    }
  ]

  # 处理边数据
  edges: [
    # 设置 Edge type follow 相关信息
    {
      #  NebulaGraph 中对应的 Edge type 名称。
      name: follow

      type: {
        # 指定数据源文件格式，设置为 ClickHouse。
        source: clickhouse

        # 指定边数据导入 NebulaGraph 的方式，
        # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
        sink: client
      }

      # ClickHouse 的 JDBC URL
      url:"jdbc:clickhouse://192.168.*.*:8123/basketballplayer"

      user:"user"
      password:"123456"

      # ClickHouse 分区数
      numPartition:"5"

      table:"follow"
      sentence:"select * from follow"

      # 在 fields 里指定 follow 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
      # fields 和 nebula.fields 里的配置必须一一对应。
      # 如果需要指定多个列名称，用英文逗号（,）隔开。
      fields: [degree]
      nebula.fields: [degree]

      # 在 source 里，将 follow 表中某一列作为边的起始点数据源。
      source: {
        field:src_player
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      # 在 target 里，将 follow 表中某一列作为边的目的点数据源。
      target: {
        field:dst_player
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      # 指定一个列作为 rank 的源（可选）。
      #ranking: rank

      # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
      # filter: "name='Tom'"

      # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
      #writeMode: INSERT

      # 单批次写入 NebulaGraph 的数据条数。
      batch: 256

      # 数据写入 NebulaGraph 时需要创建的分区数。
      partition: 32
    }

    # 设置 Edge type serve 相关信息
    {
      name: serve
      type: {
        source: clickhouse
        sink: client
      }
      url:"jdbc:clickhouse://192.168.*.*:8123/basketballplayer"
      user:"user"
```

```
      password:"123456"
      numPartition:"5"
      sentence:"select * from serve"
      fields: [start_year,end_year]
      nebula.fields: [start_year,end_year]
      source: {
        field:playerid
      }
      target: {
        field:teamid
      }

      # 指定一个列作为 rank 的源（可选）。
      #ranking: rank

      batch: 256
      partition: 32
    }
  ]
}
```

步骤 3：向 **NebulaGraph** 导入数据

运行如下命令将 ClickHouse 数据导入到 NebulaGraph 中。关于参数的说明，请参见导入命令参数。

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange.jar_path> -c <clickhouse_application.conf_path>
```

> Note
>
> JAR 包有两种获取方式：自行编译或者从 maven 仓库下载。

示例：

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange_spark_2.4-3.8.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/clickhouse_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`，确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 4：（可选）验证数据

用户可以在 NebulaGraph 客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 5：（如有）在 **NebulaGraph** 中重建索引

导入数据后，用户可以在 NebulaGraph 中重新创建并重建索引。详情请参见索引介绍。

最后更新: 2025年8月4日

Macro Rendering Error

File: `import-export/nebula-exchange/use-exchange/ex-ug-import-from-neo4j.md`

UndefinedError: 'exchange_ent' is undefined

```
Traceback (most recent call last):
  File "/opt/hostedtoolcache/Python/3.8.18/x64/lib/python3.8/site-packages/mkdocs_macros/plugin.py", line 688, in render
    return md_template.render(**page_variables)
  File "/opt/hostedtoolcache/Python/3.8.18/x64/lib/python3.8/site-packages/jinja2/environment.py", line 1295, in render
    self.environment.handle_exception()
  File "/opt/hostedtoolcache/Python/3.8.18/x64/lib/python3.8/site-packages/jinja2/environment.py", line 942, in handle_exception
    raise rewrite_traceback_stack(source=source)
  File "<template>", line 117, in top-level template code
  File "/opt/hostedtoolcache/Python/3.8.18/x64/lib/python3.8/site-packages/jinja2/environment.py", line 490, in getattr
    return getattr(obj, attribute)
jinja2.exceptions.UndefinedError: 'exchange_ent' is undefined
```

最后更新: 2025年8月4日

导入 Hive 数据

本文以一个示例说明如何使用 Exchange 将存储在 Hive 上的数据导入 NebulaGraph 。

数据集

本文以 basketballplayer 数据集为例。

在本示例中，该数据集已经存入 Hive 中名为 `basketball` 的数据库中，以 `player` 、 `team` 、 `follow` 和 `serve` 四个表存储了所有点和边的信息。以下为各个表的结构。

```
scala> spark.sql("describe basketball.player").show
+--------+---------+-------+
|col_name|data_type|comment|
+--------+---------+-------+
|playerid|   string|   null|
|     age|   bigint|   null|
|    name|   string|   null|
+--------+---------+-------+

scala> spark.sql("describe basketball.team").show
+----------+---------+-------+
|  col_name|data_type|comment|
+----------+---------+-------+
|    teamid|   string|   null|
|      name|   string|   null|
+----------+---------+-------+

scala> spark.sql("describe basketball.follow").show
+----------+---------+-------+
|  col_name|data_type|comment|
+----------+---------+-------+
|src_player|   string|   null|
|dst_player|   string|   null|
|    degree|   bigint|   null|
+----------+---------+-------+

scala> spark.sql("describe basketball.serve").show
+----------+---------+-------+
|  col_name|data_type|comment|
+----------+---------+-------+
|  playerid|   string|   null|
|    teamid|   string|   null|
|start_year|   bigint|   null|
|  end_year|   bigint|   null|
+----------+---------+-------+
```

说明：Hive 的数据类型 `bigint` 与 NebulaGraph 的 `int` 对应。

环境配置

本文示例在 MacOS 下完成，以下是相关的环境配置信息：

- 硬件规格：
- CPU：1.7 GHz Quad-Core Intel Core i7
- 内存：16 GB
- Spark：2.4.7，单机版
- Hive：2.3.7，Hive Metastore 数据库为 MySQL 8.0.22
- NebulaGraph： master。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署 NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的的 IP 地址和端口。
- 拥有 NebulaGraph 写权限的用户名和密码。
- 已经编译 Exchange。详情请参见编译 Exchange。本示例中使用 Exchange 3.8.0。
- 已经安装 Spark。
- 了解 NebulaGraph 中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。
- 已经启动 Hive Metastore 数据库（本示例中为 MySQL）。

操作步骤

步骤 1：在 **NebulaGraph** 中创建 **Schema**

分析数据，按以下步骤在 NebulaGraph 中创建 Schema：

1. 确认 Schema 要素。 NebulaGraph 中的 Schema 要素如下表所示。

要素	名称	属性
Tag	`player`	`name string, age int`
Tag	`team`	`name string`
Edge Type	`follow`	`degree int`
Edge Type	`serve`	`start_year int, end_year int`

2. 在 NebulaGraph 中创建一个图空间 basketballplayer，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
        (partition_num = 10, \
        replica_factor = 1, \
        vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见快速开始。

步骤 2：使用 **Spark SQL** 确认 **Hive SQL** 语句

启动 spark-shell 环境后，依次运行以下语句，确认 Spark 能读取 Hive 中的数据。

```
scala> sql("select playerid, age, name from basketball.player").show
scala> sql("select teamid, name from basketball.team").show
scala> sql("select src_player, dst_player, degree from basketball.follow").show
scala> sql("select playerid, teamid, start_year, end_year from basketball.serve").show
```

以下为表 `basketball.player` 中读出的结果。

```
+---------+----+-----------------+
| playerid| age|             name|
+---------+----+-----------------+
|player100|  42|       Tim Duncan|
```

```
|player101| 36|     Tony Parker|
|player102| 33|LaMarcus Aldridge|
|player103| 32|       Rudy Gay|
|player104| 32| Marco Belinelli|
+---------+----+-----------------+
...
```

步骤 **3**：修改配置文件

编译 Exchange 后，复制 `target/classes/application.conf` 文件设置 Hive 数据源相关的配置。在本示例中，复制的文件名为 `hive_application.conf`。各个配置项的详细说明请参见配置说明。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.8.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    cores: {
      max: 16
    }
  }

  # 如果 Spark 和 Hive 部署在不同集群，才需要配置连接 Hive 的参数，否则请忽略这些配置。
  #hive: {
  #  waredir: "hdfs://NAMENODE_IP:9000/apps/svr/hive-xxx/warehouse/"
  #  connectionURL: "jdbc:mysql://your_ip:3306/hive_spark?characterEncoding=UTF-8"
  #  connectionDriverName: "com.mysql.jdbc.Driver"
  #  connectionUserName: "user"
  #  connectionPassword: "password"
  #}

  # NebulaGraph 相关配置
  nebula: {
    address:{
      # 以下为 NebulaGraph 的 Graph 服务和所有 Meta 服务所在机器的 IP 地址及端口。
      # 如果有多个地址，格式为 "ip1:port","ip2:port","ip3:port"。
      # 不同地址之间以英文逗号 (,) 隔开。
      graph:["127.0.0.1:9669"]
      #任意一个 Meta 服务的地址。
      #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
      meta:["127.0.0.1:9559"]
    }
    # 填写的账号必须拥有 NebulaGraph 相应图空间的写数据权限。
    user: root
    pswd: nebula
    # 是否使用 RSA 加密的密码。
    # enableRSA: true
    # 使用 RSA 加密密码时的密钥。
    # privateKey: ""
    # 填写 NebulaGraph 中需要写入数据的图空间名称。
    space: basketballplayer
    connection: {
      timeout: 3000
      retry: 3
    }
    execution: {
      retry: 3
    }
    error: {
      max: 32
      output: /tmp/errors
    }
    rate: {
      limit: 1024
      timeout: 1000
    }
  }
  # 处理点
  tags: [
    # 设置 Tag player 相关信息。
    {
      # NebulaGraph 中对应的 Tag 名称。
      name: player
      type: {
        # 指定数据源文件格式，设置为 hive。
        source: hive
        # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
        sink: client
      }

      # 设置读取数据库 basketball 中 player 表数据的 SQL 语句
      exec: "select playerid, age, name from basketball.player"

      # 在 fields 里指定 player 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
      # fields 和 nebula.fields 里的配置必须一一对应。
      # 如果需要指定多个列名称，用英文逗号 (,) 隔开。
```

```
      fields: [age,name]
      nebula.fields: [age,name]

      # 指定表中某一列数据为 NebulaGraph 中点 VID 的来源。
      vertex:{
        field:playerid
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
      # filter: "name='Tom'"

      # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
      #writeMode: INSERT

      # 批量删除时是否删除该点关联的出边和入边。`writeMode`为`DELETE`时该参数生效。
      #deleteEdge: false

      # 单批次写入 NebulaGraph 的最大数据条数。
      batch: 256

      # 数据写入 NebulaGraph 时需要创建的分区数。
      partition: 32
    }
    # 设置 Tag team 相关信息。
    {
      name: team
      type: {
        source: hive
        sink: client
      }
      exec: "select teamid, name from basketball.team"
      fields: [name]
      nebula.fields: [name]
      vertex: {
        field: teamid
      }
      batch: 256
      partition: 32
    }

  ]

  # 处理边数据
  edges: [
    # 设置 Edge type follow 相关信息
    {
      # NebulaGraph 中对应的 Edge type 名称。
      name: follow

      type: {
        # 指定数据源文件格式，设置为 hive。
        source: hive

        # 指定边数据导入 NebulaGraph 的方式，
        # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
        sink: client
      }

      # 设置读取数据库 basketball 中 follow 表数据的 SQL 语句。
      exec: "select src_player, dst_player, degree from basketball.follow"

      # 在 fields 里指定 follow 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
      # fields 和 nebula.fields 里的配置必须一一对应。
      # 如果需要指定多个列名称，用英文逗号（,）隔开。
      fields: [degree]
      nebula.fields: [degree]

      # 在 source 里，将 follow 表中某一列作为边的起始点数据源。
      # 在 target 里，将 follow 表中某一列作为边的目的点数据源。
      source: {
        field: src_player
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      target: {
        field: dst_player
      # udf:{
```

```
        #           separator:"_"
        #           oldColNames:[field-0,field-1,field-2]
        #           newColName:new-field
        #       }
        # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
        # prefix:"tag1"
        # 对 string 类型的 VID 进行哈希化操作。
        # policy:hash
        }

        # 指定一个列作为 rank 的源（可选）。
        #ranking: rank

        # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
        # filter: "name='Tom'"

        # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
        #writeMode: INSERT

        # 单批次写入 NebulaGraph 的最大数据条数。
        batch: 256

        # 数据写入 NebulaGraph 时需要创建的分区数。
        partition: 32
      }

      # 设置 Edge type serve 相关信息
      {
        name: serve
        type: {
          source: hive
          sink: client
        }
        exec: "select playerid, teamid, start_year, end_year from basketball.serve"
        fields: [start_year,end_year]
        nebula.fields: [start_year,end_year]
        source: {
          field: playerid
        }
        target: {
          field: teamid
        }

        # 指定一个列作为 rank 的源（可选）。
        #ranking: rank

        batch: 256
        partition: 32
      }
    ]
  }
```

步骤 4：向 **NebulaGraph** 导入数据

运行如下命令将 Hive 数据导入到 NebulaGraph 中。关于参数的说明，请参见导入命令参数。

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange.jar_path> -c <hive_application.conf_path> -h
```

Note

JAR 包有两种获取方式：自行编译或者从 maven 仓库下载。

示例：

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange_spark_2.4-3.8.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/hive_application.conf -h
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`，确认成功的数量。例如 `batchSuccess.follow: 300`。

访问 Kerberos 认证的 HDFS

使用 Kerberos 进行安全认证时，需使用以下两种方式之一访问 Kerberos 认证的 HDFS。

- 在命令中设置 Kerberos 配置文件

  在命令中配置 `--conf` 和 `--files`，例如：

```
${SPARK_HOME}/bin/spark-submit --master xxx --num-executors 2 --executor-cores 2 --executor-memory 1g \
--conf "spark.driver.extraJavaOptions=-Djava.security.krb5.conf=./krb5.conf" \
--conf "spark.executor.extraJavaOptions=-Djava.security.krb5.conf=./krb5.conf" \
--files /local/path/to/xxx.keytab,/local/path/to/krb5.conf \
--class com.vesoft.nebula.exchange.Exchange \
exchange.jar -c xx.conf
```

  `--conf` 中的文件路径有如下两种配置方式：

- 配置文件的绝对路径。要求所有 YARN 或者 Spark 机器相同路径下都有对应文件。
- （YARN 模式下推荐）配置文件的相对路径（例如 `./krb5.conf`）。通过 `--files` 上传的资源文件就在 Java 虚拟机或者 JAR 的工作目录下。

  `--files` 中的文件必须存储在执行 `spark-submit` 命令的机器上。

- 不使用命令

  将 Spark 和 Kerberos 认证的 Hadoop 部署在相同集群内，共用 HDFS 和 YARN，然后在 Spark 的 `spark-env.sh` 中增加配置 `export HADOOP_HOME=<hadoop_home_path>`。

步骤 **5**：（可选）验证数据

用户可以在 NebulaGraph 客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 **6**：（如有）在 **NebulaGraph** 中重建索引

导入数据后，用户可以在 NebulaGraph 中重新创建并重建索引。详情请参见索引介绍。

最后更新: 2025年8月4日

导入 **MaxCompute** 数据

本文以一个示例说明如何使用 Exchange 将存储在 MaxCompute 上的数据导入 NebulaGraph 。

数据集

本文以 basketballplayer 数据集为例。

环境配置

本文示例在 MacOS 下完成，以下是相关的环境配置信息：

- 硬件规格：
- CPU：1.7 GHz Quad-Core Intel Core i7
- 内存：16 GB

- Spark：2.4.7，单机版

- MaxCompute：阿里云官方版本

- NebulaGraph： master。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署 NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的的 IP 地址和端口。
- 拥有 NebulaGraph 写权限的用户名和密码。

- 已经编译 Exchange。详情请参见编译 Exchange。本示例中使用 Exchange 3.8.0。

- 已经安装 Spark。

- 了解 NebulaGraph 中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。

操作步骤

步骤 **1**：在 **NebulaGraph** 中创建 **Schema**

分析数据，按以下步骤在 NebulaGraph 中创建 Schema：

1. 确认 Schema 要素。 NebulaGraph 中的 Schema 要素如下表所示。

要素	名称	属性
Tag	`player`	`name string, age int`
Tag	`team`	`name string`
Edge Type	`follow`	`degree int`
Edge Type	`serve`	`start_year int, end_year int`

2. 在 NebulaGraph 中创建一个图空间 basketballplayer，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
        (partition_num = 10, \
        replica_factor = 1, \
        vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见快速开始。

步骤 **2**：修改配置文件

编译 Exchange 后，复制 `target/classes/application.conf` 文件设置 MaxCompute 数据源相关的配置。在本示例中，复制的文件名为 `maxcompute_application.conf`。各个配置项的详细说明请参见配置说明。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.8.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    cores: {
      max: 16
    }
  }

# NebulaGraph 相关配置
  nebula: {
    address:{
      # 以下为 NebulaGraph 的 Graph 服务和 Meta 服务所在机器的 IP 地址及端口。
      # 如果有多个地址，格式为 "ip1:port","ip2:port","ip3:port"。
      # 不同地址之间以英文逗号 (,) 隔开。
      graph:["127.0.0.1:9669"]
      #任意一个 Meta 服务的地址。
      #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
      meta:["127.0.0.1:9559"]
    }
    # 填写的账号必须拥有 NebulaGraph 相应图空间的写数据权限。
    user: root
    pswd: nebula
    # 是否使用 RSA 加密的密码。
    # enableRSA: true
    # 使用 RSA 加密密码时的密钥。
    # privateKey: ""
    # 填写 NebulaGraph 中需要写入数据的图空间名称。
    space: basketballplayer
```

```
  connection: {
    timeout: 3000
    retry: 3
  }
  execution: {
    retry: 3
  }
  error: {
    max: 32
    output: /tmp/errors
  }
  rate: {
    limit: 1024
    timeout: 1000
  }
}
# 处理点
tags: [
  # 设置 Tag player 相关信息
  {
    name: player
    type: {
      # 指定数据源文件格式，设置为 MaxCompute。
      source: maxcompute
      # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
      sink: client
    }

    # MaxCompute 的表名
    table:player

    # MaxCompute 的项目名
    project:project

    # MaxCompute 服务的 odpsUrl 和 tunnelUrl，
    # 地址可在 https://help.aliyun.com/document_detail/34951.html 查看。
    odpsUrl:"http://service.cn-hangzhou.maxcompute.aliyun.com/api"
    tunnelUrl:"http://dt.cn-hangzhou.maxcompute.aliyun.com"

    # MaxCompute 服务的 accessKeyId 和 accessKeySecret。
    accessKeyId:xxx
    accessKeySecret:xxx

    # MaxCompute 表的分区描述，该配置可选。
    partitionSpec:"dt='partition1'"

    # MaxCompute 的 Spark 连接器在读取 MaxCompute 数据时使用的分区数。默认为1，该配置可选。
    numPartitions:100

    # 请确保 SQL 语句中的表名和上方 table 的值相同，该配置可选。
    sentence:"select id, name, age, playerid from player where id < 10"

    # 在 fields 里指定 player 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
    # fields 和 nebula.fields 里的配置必须一一对应。
    # 如果需要指定多个列名称，用英文逗号（,）隔开。
    fields:[name, age]
    nebula.fields:[name, age]

    # 指定表中某一列数据为 NebulaGraph 中点 VID 的来源。
    vertex:{
      field: playerid
    # udf:{
    #            separator:"_"
    #            oldColNames:[field-0,field-1,field-2]
    #            newColName:new-field
    #        }
    # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
    # prefix:"tag1"
    # 对 string 类型的 VID 进行哈希化操作。
    # policy:hash
    }

    # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
    # filter: "name='Tom'"

    # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
    #writeMode: INSERT

    # 批量删除时是否删除该点关联的出边和入边。`writeMode`为`DELETE`时该参数生效。
    #deleteEdge: false

    # 单批次写入 NebulaGraph 的数据条数。
    batch: 256

    # 数据写入 NebulaGraph 时需要创建的分区数。
    partition: 32
  }

  # 设置 Tag team 相关信息。
  {
    name: team
    type: {
      source: maxcompute
      sink: client
```

```
      }
      table:team
      project:project
      odpsUrl:"http://service.cn-hangzhou.maxcompute.aliyun.com/api"
      tunnelUrl:"http://dt.cn-hangzhou.maxcompute.aliyun.com"
      accessKeyId:xxx
      accessKeySecret:xxx
      partitionSpec:"dt='partition1'"
      sentence:"select id, name, teamid from team where id < 10"
      fields:[name]
      nebula.fields:[name]
      vertex:{
        field: teamid
      }
      batch: 256
      partition: 32
    }
  ]

  # 处理边数据
  edges: [
    # 设置 Edge type follow 相关信息
    {
      # NebulaGraph 中对应的 Edge type 名称。
      name: follow

      type:{
        # 指定数据源文件格式，设置为 MaxCompute。
        source:maxcompute

        # 指定边数据导入 NebulaGraph 的方式，
        # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
        sink:client
      }

      # MaxCompute 的表名
      table:follow

      # MaxCompute 的项目名
      project:project

      # MaxCompute 服务的 odpsUrl 和 tunnelUrl，
      # 地址可在 https://help.aliyun.com/document_detail/34951.html 查看。
      odpsUrl:"http://service.cn-hangzhou.maxcompute.aliyun.com/api"
      tunnelUrl:"http://dt.cn-hangzhou.maxcompute.aliyun.com"

      # MaxCompute 服务的 accessKeyId 和 accessKeySecret。
      accessKeyId:xxx
      accessKeySecret:xxx

      # MaxCompute 表的分区描述，该配置可选。
      partitionSpec:"dt='partition1'"

      # 请确保 SQL 语句中的表名和上方 table 的值相同，该配置可选。
      sentence:"select * from follow"

      # 在 fields 里指定 follow 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
      # fields 和 nebula.fields 里的配置必须一一对应。
      # 如果需要指定多个列名称，用英文逗号（,）隔开。
      fields:[degree]
      nebula.fields:[degree]

      # 在 source 里，将 follow 表中某一列作为边的起始点数据源。
      source:{
        field: src_player
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      # 在 target 里，将 follow 表中某一列作为边的目的点数据源。
      target:{
        field: dst_player
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      # 指定一个列作为 rank 的源（可选）。
      #ranking: rank

      # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
```

```
      # filter: "name='Tom'"

      # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
      #writeMode: INSERT

      # 数据写入 NebulaGraph 时需要创建的分区数。
      partition:10

      # 单批次写入 NebulaGraph 的数据条数。
      batch:10
    }

    # 设置 Edge type serve 相关信息
    {
      name: serve
      type:{
        source:maxcompute
        sink:client
      }
      table:serve
      project:project
      odpsUrl:"http://service.cn-hangzhou.maxcompute.aliyun.com/api"
      tunnelUrl:"http://dt.cn-hangzhou.maxcompute.aliyun.com"
      accessKeyId:xxx
      accessKeySecret:xxx
      partitionSpec:"dt='partition1'"
      sentence:"select * from serve"
      fields:[start_year,end_year]
      nebula.fields:[start_year,end_year]
      source:{
        field: playerid
      }
      target:{
        field: teamid
      }

      # 指定一个列作为 rank 的源（可选）。
      #ranking: rank

      partition:10
      batch:10
    }
  ]
}
```

步骤 3：向 **NebulaGraph** 导入数据

运行如下命令将 MaxCompute 数据导入到 NebulaGraph 中。关于参数的说明，请参见导入命令参数。

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange.jar_path> -c <maxcompute_application.conf_path>
```

> Note
>
> JAR 包有两种获取方式：自行编译或者从 maven 仓库下载。

示例：

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange_spark_2.4-3.8.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/maxcompute_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`，确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 4：（可选）验证数据

用户可以在 NebulaGraph 客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 5：（如有）在 **NebulaGraph** 中重建索引

导入数据后，用户可以在 NebulaGraph 中重新创建并重建索引。详情请参见索引介绍。

最后更新: 2025年8月4日

导入 **Pulsar** 数据

本文简单说明如何使用 Exchange 将存储在 Pulsar 上的数据导入 NebulaGraph 。

环境配置

本文示例在 MacOS 下完成，以下是相关的环境配置信息：

- 硬件规格：
- CPU：1.7 GHz Quad-Core Intel Core i7
- 内存：16 GB

- Spark：2.4.7，单机版

- NebulaGraph ： master。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署 NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的的 IP 地址和端口。
- 拥有 NebulaGraph 写权限的用户名和密码。

- 已经编译 Exchange。详情请参见编译 Exchange。本示例中使用 Exchange 3.8.0。

- 已经安装 Spark。

- 了解 NebulaGraph 中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。

- 已经安装并开启 Pulsar 服务。

注意事项

- 导入 Pulsar 数据时只支持 Client 模式，即参数 `tags.type.sink` 和 `edges.type.sink` 的值为 `client` 。

- 导入 Pulsar 数据时请勿使用 Exchange 3.4.0 版本，该版本增加了对导入数据的缓存，不支持流式数据导入。请使用 Exchange 3.0.0、3.3.0、3.5.0 版本。

操作步骤

步骤 **1**：在 **NebulaGraph** 中创建 **Schema**

分析数据，按以下步骤在 NebulaGraph 中创建 Schema：

1. 确认 Schema 要素。 NebulaGraph 中的 Schema 要素如下表所示。

要素	名称	属性
Tag	`player`	`name string, age int`
Tag	`team`	`name string`
Edge Type	`follow`	`degree int`
Edge Type	`serve`	`start_year int, end_year int`

2. 在 NebulaGraph 中创建一个图空间 basketballplayer，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
        (partition_num = 10, \
        replica_factor = 1, \
        vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见快速开始。

步骤 **2**：修改配置文件

编译 Exchange 后，复制 `target/classes/application.conf` 文件设置 Pulsar 数据源相关的配置。在本示例中，复制的文件名为 `pulsar_application.conf`。各个配置项的详细说明请参见配置说明。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.8.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    cores: {
      max: 16
    }
  }

  # NebulaGraph 相关配置
  nebula: {
    address:{
      # 以下为 NebulaGraph 的 Graph 服务和 Meta 服务所在机器的 IP 地址及端口。
      # 如果有多个地址，格式为 "ip1:port","ip2:port","ip3:port"。
      # 不同地址之间以英文逗号 (,) 隔开。
      graph:["127.0.0.1:9669"]
      #任意一个 Meta 服务的地址。
      #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
      meta:["127.0.0.1:9559"]
    }
    # 填写的账号必须拥有 NebulaGraph 相应图空间的写数据权限。
    user: root
    pswd: nebula
    # 是否使用 RSA 加密的密码。
    # enableRSA: true
    # 使用 RSA 加密密码时的密钥。
    # privateKey: ""

    # 填写NebulaGraph中需要写入数据的图空间名称。
```

```
    space: basketballplayer
    connection: {
      timeout: 3000
      retry: 3
    }
    execution: {
      retry: 3
    }
    error: {
      max: 32
      output: /tmp/errors
    }
    rate: {
      limit: 1024
      timeout: 1000
    }
  }
  # 处理点
  tags: [
    # 设置 Tag player 相关信息。
    {
      # NebulaGraph 中对应的 Tag 名称。
      name: player
      type: {
        # 指定数据源文件格式，设置为 Pulsar。
        source: pulsar
        # 指定如何将数据导入 NebulaGraph 。只支持 Client。
        sink: client
      }
      # Pulsar 服务器地址。
      service: "pulsar://127.0.0.1:6650"
      # 连接 pulsar 的 admin.url。
      admin: "http://127.0.0.1:8081"
      # Pulsar 的选项，可以从 topic、topics 和 topicsPattern 选择一个进行配置。
      options: {
        topics: "topic1,topic2"
      }

      # 在 fields 里指定 player 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
      # fields 和 nebula.fields 里的配置必须一一对应。
      # 如果需要指定多个列名称，用英文逗号（,）隔开。
      fields: [age,name]
      nebula.fields: [age,name]

      # 指定表中某一列数据为 NebulaGraph 中点 VID 的来源。
      vertex:{
          field:playerid
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
      # filter: "name='Tom'"

      # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
      #writeMode: INSERT

      # 批量删除时是否删除该点关联的出边和入边。`writeMode`为`DELETE`时该参数生效。
      #deleteEdge: false

      # 单批次写入 NebulaGraph 的数据条数。
      batch: 10

      # 数据写入 NebulaGraph 时需要创建的分区数。
      partition: 10
      # 读取消息的间隔。单位：秒。
      interval.seconds: 10
    }
    # 设置 Tag team 相关信息。
    {
      name: team
      type: {
        source: pulsar
        sink: client
      }
      service: "pulsar://127.0.0.1:6650"
      admin: "http://127.0.0.1:8081"
      options: {
        topics: "topic1,topic2"
      }
      fields: [name]
      nebula.fields: [name]
      vertex:{
          field:teamid
      }
      batch: 10
      partition: 10
```

```
      interval.seconds: 10
    }

  ]

  # 处理边数据
  edges: [
    # 设置 Edge type follow 相关信息
    {
      # NebulaGraph 中对应的 Edge type 名称。
      name: follow

      type: {
        # 指定数据源文件格式，设置为 Pulsar。
        source: pulsar

        # 指定边数据导入 NebulaGraph 的方式，
        # 指定如何将数据导入 NebulaGraph 。只支持 Client。
        sink: client
      }

      # Pulsar 服务器地址。
      service: "pulsar://127.0.0.1:6650"
      # 连接 pulsar 的 admin.url。
      admin: "http://127.0.0.1:8081"
      # Pulsar 的选项，可以从 topic、topics 和 topicsPattern 选择一个进行配置。
      options: {
        topics: "topic1,topic2"
      }

      # 在 fields 里指定 follow 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
      # fields 和 nebula.fields 里的配置必须一一对应。
      # 如果需要指定多个列名称，用英文逗号（,）隔开。
      fields: [degree]
      nebula.fields: [degree]

      # 在 source 里，将 follow 表中某一列作为边的起始点数据源。
      # 在 target 里，将 follow 表中某一列作为边的目的点数据源。
      source:{
          field:src_player
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      target:{
          field:dst_player
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      # 指定一个列作为 rank 的源（可选）。
      #ranking: rank

      # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
      # filter: "name='Tom'"

      # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
      #writeMode: INSERT

      # 单批次写入 NebulaGraph 的数据条数。
      batch: 10

      # 数据写入 NebulaGraph 时需要创建的分区数。
      partition: 10

      # 读取消息的间隔。单位：秒。
      interval.seconds: 10
    }

    # 设置 Edge type serve 相关信息
    {
      name: serve
      type: {
        source: Pulsar
        sink: client
      }
      service: "pulsar://127.0.0.1:6650"
      admin: "http://127.0.0.1:8081"
      options: {
        topics: "topic1,topic2"
```

```
        }

        fields: [start_year,end_year]
        nebula.fields: [start_year,end_year]
        source:{
            field:playerid
        }

        target:{
            field:teamid
        }

        # 指定一个列作为 rank 的源（可选）。
        #ranking: rank

        batch: 10
        partition: 10
        interval.seconds: 10
      }
    ]
  }
```

步骤 **3**：向 **NebulaGraph** 导入数据

运行如下命令将 Pulsar 数据导入到 NebulaGraph 中。关于参数的说明，请参见导入命令参数。

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange.jar_path> -c <pulsar_application.conf_path>
```

> **Note**
>
> JAR 包有两种获取方式：自行编译或者从 maven 仓库下载。

示例：

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-exchange_spark_2.4-3.8.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/pulsar_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`，确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 **4**：（可选）验证数据

用户可以在 NebulaGraph 客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 **5**：（如有）在 **NebulaGraph** 中重建索引

导入数据后，用户可以在 NebulaGraph 中重新创建并重建索引。详情请参见索引介绍。

最后更新: 2025年8月4日

导入 **Kafka** 数据

本文简单说明如何使用 Exchange 将存储在 Kafka 上的数据导入 NebulaGraph 。

> **compatibility**
>
> 导入 Kafka 数据时请使用 Exchange 3.5.0/3.3.0/3.0.0 版本。3.4.0 版本增加了对导入数据的缓存，不支持流式数据导入。

环境配置

本文示例在 MacOS 下完成，以下是相关的环境配置信息：

- 硬件规格：
- CPU：1.7 GHz Quad-Core Intel Core i7
- 内存：16 GB

- Spark：2.4.7，单机版

- NebulaGraph： master。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署 NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的的 IP 地址和端口。
- 拥有 NebulaGraph 写权限的用户名和密码。

- 已经编译 Exchange。详情请参见编译 Exchange。本示例中使用 Exchange 3.8.0。

- 已经安装 Spark。

- 以下 JAR 包已经下载并放置在 Spark 的 `SPARK_HOME/jars` 目录下：
- spark-streaming-kafka_xxx.jar
- spark-sql-kafka-0-10_xxx.jar
- kafka-clients-xxx.jar

- 了解 NebulaGraph 中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。

- 已经安装并开启 Kafka 服务。

注意事项

- 导入 Kafka 数据时只支持 Client 模式，即参数 `tags.type.sink` 和 `edges.type.sink` 的值为 `client` 。

- 导入 Kafka 数据时请勿使用 Exchange 3.4.0 版本，该版本增加了对导入数据的缓存，不支持流式数据导入。请使用 Exchange 3.0.0、3.3.0、3.5.0 版本。

操作步骤

步骤 1：在 **NebulaGraph** 中创建 **Schema**

分析数据，按以下步骤在 NebulaGraph 中创建 Schema：

1. 确认 Schema 要素。 NebulaGraph 中的 Schema 要素如下表所示。

要素	名称	属性
Tag	`player`	`name string, age int`
Tag	`team`	`name string`
Edge Type	`follow`	`degree int`
Edge Type	`serve`	`start_year int, end_year int`

2. 在 NebulaGraph 中创建一个图空间 basketballplayer，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
        (partition_num = 10, \
        replica_factor = 1, \
        vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见快速开始。

步骤 2：修改配置文件

编译 Exchange 后，复制 `target/classes/application.conf` 文件设置 Kafka 数据源相关的配置。在本示例中，复制的文件名为 `kafka_application.conf`。各个配置项的详细说明请参见配置说明。

> Note
>
> 导入 Kafka 数据时，一个配置文件只能处理一个 Tag 或 Edge type。如果有多个 Tag 或 Edge type，需要创建多个配置文件。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.8.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    cores: {
      max: 16
    }
  }

  # NebulaGraph 相关配置
  nebula: {
    address:{
      # 以下为 NebulaGraph 的 Graph 服务和 Meta 服务所在机器的 IP 地址及端口。
      # 如果有多个地址，格式为 "ip1:port","ip2:port","ip3:port"。
      # 不同地址之间以英文逗号 (,) 隔开。
      graph:["127.0.0.1:9669"]
      #任意一个 Meta 服务的地址。
      #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
      meta:["127.0.0.1:9559"]
    }
```

```
    # 填写的账号必须拥有 NebulaGraph 相应图空间的写数据权限。
    user: root
    pswd: nebula
    # 是否使用 RSA 加密的密码。
    # enableRSA: true
    # 使用 RSA 加密密码时的密钥。
    # privateKey: ""
    # 填写 NebulaGraph 中需要写入数据的图空间名称。
    space: basketballplayer
    connection: {
      timeout: 3000
      retry: 3
    }
    execution: {
      retry: 3
    }
    error: {
      max: 32
      output: /tmp/errors
    }
    rate: {
      limit: 1024
      timeout: 1000
    }
  }
  # 处理点
  tags: [
    # 设置 Tag player 相关信息。
    {
      # NebulaGraph 中对应的 Tag 名称。
      name: player
      type: {
        # 指定数据源文件格式，设置为 Kafka。
        source: kafka
        # 指定如何将数据导入 NebulaGraph 。只支持 Client。
        sink: client
      }
      # Kafka 服务器地址。
      service: "127.0.0.1:9092"
      # 消息类别。
      topic: "topic_name1"

      # 如果 Kafka 使用 Kerberos 进行安全认证，需要设置以下参数。如果 Kafka 使用 SASL 或 SASL_PLAINTEXT 进行安全认证，无需设置 kerberos 和 kerberosServiceName。
      #securityProtocol: SASL_PLAINTEXT
      #mechanism: GASSAPI
      #kerberos: true
      #kerberosServiceName: kafka

      # 在 fields 里指定 Kafka value 中的字段名称，多个字段用英文逗号（,）隔开。Spark Structured Streaming 读取 Kafka 数据后会将其以 JSON 格式存储于 value 字段中，而这里的 fields 要配置
JSON 的 key 名。示例如下：
      fields: [personName, personAge]
      # 设置与 fields 中的 key 对应的 NebulaGraph 属性名，key 的 value 将保存为相应的属性值。下方设置会将 personName 的 value 保存到 NebulaGraph 中的 name 属性，personAge 的 value 则保存到
age 属性。
      nebula.fields: [name, age]

      # 指定表中某一列数据为 NebulaGraph 中点 VID 的来源。
      # 这里的值 key 和上面的 key 重复，表示 key 既作为 VID，也作为属性 name。
      vertex:{
          field:personId
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
      # filter: "name='Tom'"

      # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
      #writeMode: INSERT

      # 单批次写入 NebulaGraph 的数据条数。
      batch: 10

      # 数据写入 NebulaGraph 时需要创建的分区数。
      partition: 10
      # 读取消息的间隔。单位：秒。
      interval.seconds: 10
      # 消费起点，默认值 latest。 可选 latest、earliest
      startingOffsets: latest
      # 流控，对每个触发区间处理的最大偏移量的速率限制，可不配置。
      # maxOffsetsPerTrigger:10000
    }
  ]
  # 处理边数据
  #edges: [
  #  # 设置 Edge type follow 相关信息
  #  {
  #    # NebulaGraph 中对应的 Edge type 名称。
```

```
  #    name: follow

  #    type: {
  #      # 指定数据源文件格式，设置为 Kafka。
  #      source: kafka

  #      # 指定边数据导入 NebulaGraph 的方式，
  #      # 指定如何将数据导入 NebulaGraph 。只支持 Client。
  #      sink: client
  #    }

  #    # Kafka 服务器地址。
  #    service: "127.0.0.1:9092"
  #    # 消息类别。
  #    topic: "topic_name3"

  #    # 如果 Kafka 启用 Kerberos 进行安全认证，需要设置以下参数。如果 Kafka 启用 SASL 或 SASL_PLAINTEXT 进行安全认证，无需设置 kerberos 和 kerberosServiceName。
  #    #securityProtocol: SASL_PLAINTEXT
  #    #mechanism: GASSAPI
  #    #kerberos: true
  #    #kerberosServiceName: kafka

  #    # 在 fields 里指定 Kafka value 中的字段名称，多个字段用英文逗号（,）隔开。Spark Structured Streaming 读取 Kafka 数据后会将其以 JSON 格式存储于 value 字段中，而这里的 fields 要配置
JSON 的 key 名。示例如下：
  #    fields: [degree]
  #    # 设置与 fields 中的 key 对应的 NebulaGraph 属性名，key 的 value 将保存为相应的属性值。下方设置会将 degree 的 value 保存到 NebulaGraph 中的 degree 属性。
  #    nebula.fields: [degree]

  #    # 在 source 里，将 topic 中某一列作为边的起始点数据源。
  #    # 在 target 里，将 topic 中某一列作为边的目的点数据源。
  #    source:{
  #        field:srcPersonId
  #    # udf:{
  #    #            separator:"_"
  #    #            oldColNames:[field-0,field-1,field-2]
  #    #            newColName:new-field
  #    #        }
  #    # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
  #    # prefix:"tag1"
  #    # 对 string 类型的 VID 进行哈希化操作。
  #    # policy:hash
  #    }

  #    target:{
  #        field:dstPersonId
  #    # udf:{
  #    #            separator:"_"
  #    #            oldColNames:[field-0,field-1,field-2]
  #    #            newColName:new-field
  #    #        }
  #    # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
  #    # prefix:"tag1"
  #    # 对 string 类型的 VID 进行哈希化操作。
  #    # policy:hash
  #    }

  #    # 指定一个列作为 rank 的源（可选）。
  #    #ranking: rank

  #    # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
  #    #writeMode: INSERT

  #    # 单批次写入 NebulaGraph 的数据条数。
  #    batch: 10

  #    # 数据写入 NebulaGraph 时需要创建的分区数。
  #    partition: 10

  #    # 读取消息的间隔。单位：秒。
  #    interval.seconds: 10
  #    # 消费起点，默认值 latest。 可选 latest、earliest
  #    startingOffsets: latest
  #    # 流控，对每个触发区间处理的最大偏移量的速率限制，可不配置。
  #    # maxOffsetsPerTrigger:10000
  #  }
  #]
}
```

步骤 3：向 NebulaGraph 导入数据

运行如下命令将 Kafka 数据导入到 NebulaGraph 中。关于参数的说明，请参见导入命令参数。

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange.jar_path> -c <kafka_application.conf_path>
```

Note

- JAR 包需要从NebulaGraph套餐中获取。
- 如果 Kafka 启用了安全认证，导入数据时需要配置驱动程序和执行器，参见下方示例。

示例：

- 无安全认证

```
${SPARK_HOME}/bin/spark-submit --master "local" \
--class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/target/nebula-exchange_spark_2.4-3.8.0.jar \
-c /root/nebula-exchange/target/classes/kafka_application.conf
```

- 启用 Kerberos 安全认证

```
${SPARK_HOME}/bin/spark-submit --master "local" \
--conf "spark.driver.extraJavaOptions=-Djava.security.auth.login.config=/path/kafka_client_jaas.conf -Djava.security.krb5.conf=/path/krb5.conf" \
--conf "spark.executor.extraJavaOptions=-Djava.security.auth.login.config=/path/kafka_client_jaas.conf -Djava.security.krb5.conf=/path/krb5.conf" \
--files /local/path/kafka_client_jaas.conf,/local/path/kafka.keytab,/local/path/krb5.conf \
--class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/target/nebula-exchange_spark_2.4-3.8.0.jar \
-c /root/nebula-exchange/target/classes/kafka_application.conf
```

- 启用 SASL/SASL_PLAINTEXT 安全认证

```
${SPARK_HOME}/bin/spark-submit --master "local" \
--conf "spark.driver.extraJavaOptions=-Djava.security.auth.login.config=/path/kafka_client_jaas.conf" \
--conf "spark.executor.extraJavaOptions=-Djava.security.auth.login.config=/path/kafka_client_jaas.conf" \
--files /local/path/kafka_client_jaas.conf \
--class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/target/nebula-exchange_spark_2.4-3.8.0.jar \
-c /root/nebula-exchange/target/classes/kafka_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`，确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 4：（可选）验证数据

用户可以在 NebulaGraph 客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 5：（如有）在 **NebulaGraph** 中重建索引

导入数据后，用户可以在 NebulaGraph 中重新创建并重建索引。详情请参见索引介绍。

最后更新: 2025年8月4日

导入通用 JDBC 数据

JDBC 数据是指用 JDBC 接口访问的各类数据库的数据的统称。本文以 MySQL 数据库为例说明如何使用 Exchange 将 JDBC 数据导入 NebulaGraph 。

数据集

本文以 basketballplayer 数据集为例。

在本示例中，该数据集已经存入 MySQL 中名为 `basketball` 的数据库中，以 `player` 、 `team` 、 `follow` 和 `serve` 四个表存储了所有点和边的信息。以下为各个表的结构。

```
mysql> desc player;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| playerid | int         | YES  |     | NULL    |       |
| age      | int         | YES  |     | NULL    |       |
| name     | varchar(30) | YES  |     | NULL    |       |
+----------+-------------+------+-----+---------+-------+

mysql> desc team;
+--------+-------------+------+-----+---------+-------+
| Field  | Type        | Null | Key | Default | Extra |
+--------+-------------+------+-----+---------+-------+
| teamid | int         | YES  |     | NULL    |       |
| name   | varchar(30) | YES  |     | NULL    |       |
+--------+-------------+------+-----+---------+-------+

mysql> desc follow;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| src_player | int         | YES  |     | NULL    |       |
| dst_player | int         | YES  |     | NULL    |       |
| degree     | int         | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+

mysql> desc serve;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| playerid   | int         | YES  |     | NULL    |       |
| teamid     | int         | YES  |     | NULL    |       |
| start_year | int         | YES  |     | NULL    |       |
| end_year   | int         | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
```

环境配置

本文示例在 MacOS 下完成，以下是相关的环境配置信息：

- 硬件规格：
- CPU：1.7 GHz Quad-Core Intel Core i7
- 内存：16 GB

- Spark：2.3.0，单机版

- NebulaGraph： master。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署 NebulaGraph 并获取如下信息：
- Graph 服务和 Meta 服务的的 IP 地址和端口。
- 拥有 NebulaGraph 写权限的用户名和密码。

- 已经编译 Exchange。详情请参见编译 Exchange。本示例中使用 Exchange 3.8.0。

- 已经安装 Spark。

- 了解 NebulaGraph 中创建 Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。

- 如果文件存储在本地且 NebulaGraph 是集群架构，需要在集群每台机器本地相同目录下放置文件。

注意事项

nebula-exchange_spark_2.2 仅支持单表查询，不支持多表查询。

操作步骤

步骤 **1**：在 **NebulaGraph** 中创建 **Schema**

分析文件中的数据，按以下步骤在 NebulaGraph 中创建 Schema：

1. 确认 Schema 要素。 NebulaGraph 中的 Schema 要素如下表所示。

要素	名称	属性
Tag	`player`	`name string, age int`
Tag	`team`	`name string`
Edge Type	`follow`	`degree int`
Edge Type	`serve`	`start_year int, end_year int`

2. 使用 NebulaGraph Console 创建一个图空间 basketballplayer，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
        (partition_num = 10, \
        replica_factor = 1, \
        vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见快速开始。

步骤 **2.** 修改配置文件

编译 Exchange 后，复制 `target/classes/application.conf` 文件设置 JDBC 数据源相关的配置。在本示例中，复制的文件名为 `jdbc_application.conf`。各个配置项的详细说明请参见配置说明。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.8.0
    }
    driver: {
      cores: 1
      maxResultSize: 1G
    }
    executor: {
        memory:1G
    }

    cores: {
      max: 16
    }
  }

  # NebulaGraph 相关配置
  nebula: {
    address:{
      # 指定 Graph 服务和所有 Meta 服务的 IP 地址和端口。
      # 如果有多台服务器，地址之间用英文逗号（,）分隔。
      # 格式："ip1:port","ip2:port","ip3:port"
      graph:["127.0.0.1:9669"]
      #任意一个 Meta 服务的地址。
      #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
      meta:["127.0.0.1:9559"]
    }
```

```
    # 指定拥有 NebulaGraph 写权限的用户名和密码。
    user: root
    pswd: nebula
    # 是否使用 RSA 加密的密码。
    # enableRSA: true
    # 使用 RSA 加密密码时的密钥。
    # privateKey: ""
    # 指定图空间名称。
    space: basketballplayer
    connection: {
      timeout: 3000
      retry: 3
    }
    execution: {
      retry: 3
    }
    error: {
      max: 32
      output: /tmp/errors
    }
    rate: {
      limit: 1024
      timeout: 1000
    }
  }

  # 处理点
  tags: [
    # 设置 Tag player 相关信息。
    {
      # 指定 NebulaGraph 中定义的 Tag 名称。
      name: player
      type: {
        # 指定数据源，使用 JDBC。
        source: jdbc

        # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
        sink: client
      }

      # JDBC 数据源的 URL。示例为 MySql 数据库。
      url:"jdbc:mysql://127.0.0.1:3306/basketball?useUnicode=true&characterEncoding=utf-8"

      # JDBC 驱动。
      driver:"com.mysql.cj.jdbc.Driver"

      # 数据库用户名和密码。
      user:"root"
      password:"12345"

      # 扫描单个表读取数据。
      # nebula-exchange_spark_2.2 必须配置该参数，还可以额外配置 sentence。
      # nebula-exchange_spark_2.4 和 nebula-exchange_spark_3.0 可以配置该参数，但是不能和 sentence 同时配置。
      table:"basketball.player"

      # 通过查询语句读取数据。
      # nebula-exchange_spark_2.2 可以配置该参数。不支持多表查询。在 from 后只需要写表名，不支持`库名.表名`。
      # nebula-exchange_spark_2.4 和 nebula-exchange_spark_3.0 可以配置该参数，但是不能和 table 同时配置。支持多表查询。
      # sentence:"select playerid, age, name from player, team order by playerid"

      # （可选）多连接读取参数 参见 https://spark.apache.org/docs/latest/sql-data-sources-jdbc.html
      partitionColumn:playerid    # 可选。数值类型必须为数字、日期或时间戳。
      lowerBound:1                # 可选
      upperBound:5                # 可选
      numPartitions:5             # 可选


      fetchSize:2           # 每次请求数据库要读取的行数。

      # 在 fields 里指定 player 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
      # fields 和 nebula.fields 里的配置必须一一对应。
      # 如果需要指定多个列名称，用英文逗号（,）隔开。
      fields: [age,name]
      nebula.fields: [age,name]

      # 指定表中某一列数据为 NebulaGraph 中点 VID 的来源。
      vertex: {
        field:playerid
      # udf:{
      #            separator:"_"
      #            oldColNames:[field-0,field-1,field-2]
      #            newColName:new-field
      #        }
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
      # filter: "name='Tom'"

      # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
```

```
    #writeMode: INSERT

    # 批量删除时是否删除该点关联的出边和入边。`writeMode`为`DELETE`时该参数生效。
    #deleteEdge: false

    # 单批次写入 NebulaGraph 的数据条数。
    batch: 256

    # 数据写入 NebulaGraph 时需要创建的分区数。
    partition: 32
  }
  # 设置 Tag team 相关信息。
  {
    name: team
    type: {
      source: jdbc
      sink: client
    }


    url:"jdbc:mysql://127.0.0.1:3306/basketball?useUnicode=true&characterEncoding=utf-8"
    driver:"com.mysql.cj.jdbc.Driver"
    user:root
    password:"12345"
    table:team
    sentence:"select teamid, name from team order by teamid"
    partitionColumn:teamid
    lowerBound:1
    upperBound:5
    numPartitions:5
    fetchSize:2

    fields: [name]
    nebula.fields: [name]
    vertex: {
      field: teamid
    }
    batch: 256
    partition: 32
  }

]

# 处理边数据
edges: [
  # 设置 Edge type follow 相关信息
  {
    # NebulaGraph 中对应的 Edge type 名称。
    name: follow

    type: {
      # 指定数据源文件格式，设置为 JDBC。
      source: jdbc

      # 指定边数据导入 NebulaGraph 的方式，
      # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
      sink: client
    }

    url:"jdbc:mysql://127.0.0.1:3306/basketball?useUnicode=true&characterEncoding=utf-8"
    driver:"com.mysql.cj.jdbc.Driver"
    user:root
    password:"12345"

    # 扫描单个表读取数据。
    # nebula-exchange_spark_2.2 必须配置该参数，还可以额外配置 sentence。
    # nebula-exchange_spark_2.4 和 nebula-exchange_spark_3.0 可以配置该参数，但是不能和 sentence 同时配置。
    table:"basketball.follow"

    # 通过查询语句读取数据。
    # nebula-exchange_spark_2.2 可以配置该参数。不支持多表查询。在 from 后只需要写表名，不支持`库名.表名`。
    # nebula-exchange_spark_2.4 和 nebula-exchange_spark_3.0 可以配置该参数，但是不能和 table 同时配置。支持多表查询。
    # sentence:"select src_player,dst_player,degree from follow order by src_player"

    partitionColumn:src_player
    lowerBound:1
    upperBound:5
    numPartitions:5
    fetchSize:2

    # 在 fields 里指定 follow 表中的列名称，其对应的 value 会作为 NebulaGraph 中指定属性。
    # fields 和 nebula.fields 里的配置必须一一对应。
    # 如果需要指定多个列名称，用英文逗号（,）隔开。
    fields: [degree]
    nebula.fields: [degree]

    # 在 source 里，将 follow 表中某一列作为边的起始点数据源。
    # 在 target 里，将 follow 表中某一列作为边的目的点数据源。
    source: {
      field: src_player
    # udf:{
    #            separator:"_"
    #            oldColNames:[field-0,field-1,field-2]
    #            newColName:new-field
```

```
#        }
# 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
# prefix:"tag1"
# 对 string 类型的 VID 进行哈希化操作。
# policy:hash
}

target: {
  field: dst_player
# udf:{
#            separator:"_"
#            oldColNames:[field-0,field-1,field-2]
#            newColName:new-field
#        }
# 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
# prefix:"tag1"
# 对 string 类型的 VID 进行哈希化操作。
# policy:hash
}

# 指定一个列作为 rank 的源（可选）。
#ranking: rank

# 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
# filter: "name='Tom'"

# 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
#writeMode: INSERT

# 单批次写入 NebulaGraph 的数据条数。
batch: 256

# 数据写入 NebulaGraph 时需要创建的分区数。
partition: 32
}

# 设置 Edge type serve 相关信息
{
  name: serve
  type: {
    source: jdbc
    sink: client
  }

  url:"jdbc:mysql://127.0.0.1:3306/basketball?useUnicode=true&characterEncoding=utf-8"
  driver:"com.mysql.cj.jdbc.Driver"
  user:root
  password:"12345"
  table:serve
  sentence:"select playerid,teamid,start_year,end_year from serve order by playerid"
  partitionColumn:playerid
  lowerBound:1
  upperBound:5
  numPartitions:5
  fetchSize:2

  fields: [start_year,end_year]
  nebula.fields: [start_year,end_year]
  source: {
    field: playerid
  }
  target: {
    field: teamid
  }

  # 指定一个列作为 rank 的源（可选）。
  #ranking: rank

  batch: 256
  partition: 32
}
]
}
```

步骤 4：向 **NebulaGraph** 导入数据

运行如下命令将 JDBC 数据导入到 NebulaGraph 中。关于参数的说明，请参见导入命令参数。

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange <nebula-exchange.jar_path> -c <jdbc_application.conf_path>
```

> Note
>
> JAR 包有两种获取方式：自行编译或者从 maven 仓库下载。

示例：

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.exchange.Exchange /root/nebula-echange/nebula-exchange/target/nebula-exchange_spark_2.4-3.8.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/jdbc_application.conf
```

用户可以在返回信息中搜索 `batchSuccess.<tag_name/edge_name>`，确认成功的数量。例如 `batchSuccess.follow: 300`。

步骤 **5**：（可选）验证数据

用户可以在 NebulaGraph 客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 **6**：（如有）在 **NebulaGraph** 中重建索引

导入数据后，用户可以在 NebulaGraph 中重新创建并重建索引。详情请参见索引介绍。

最后更新: 2025年8月4日

导入 SST 文件数据

本文以一个示例说明如何将数据源的数据生成 SST（Sorted String Table）文件并保存在 HDFS 上，然后导入 NebulaGraph，示例数据源是 CSV 文件。

注意事项

- 仅 Linux 系统支持导入 SST 文件。
- 不支持属性的 Default 值。

背景信息

Exchange 支持两种数据导入模式：

- 直接将数据源的数据通过 nGQL 语句的形式导入 NebulaGraph 。
- 将数据源的数据生成 SST 文件，然后借助 Console 将 SST 文件导入 NebulaGraph 。

下文将介绍生成 SST 文件并用其导入数据的适用场景、实现方法、前提条件、操作步骤等内容。

适用场景

- 适合在线业务，因为生成时几乎不会影响业务（只是读取 Schema），导入速度快。

> **Caution**
>
> 导入期间（大约 10 秒）会阻塞对应空间的写操作，并且之后数小时内可能有历史数据整理，建议在业务低峰期进行导入。

- 适合数据源数据量较大的场景，导入速度快。

实现方法

NebulaGraph 底层使用 RocksDB 作为键值型存储引擎。RocksDB 是基于硬盘的存储引擎，提供了一系列 API 用于创建及导入 SST 格式的文件，有助于快速导入海量数据。

SST 文件是一个内部包含了任意长度的有序键值对集合的文件，用于高效地存储大量键值型数据。生成 SST 文件的整个过程主要由 Exchange 的 Reader、sstProcessor 和 sstWriter 完成。整个数据处理过程如下：

1. Reader 从数据源中读取数据。
2. sstProcessor 根据 NebulaGraph 的 Schema 信息生成 SST 文件，然后上传至 HDFS。SST 文件的格式请参见数据存储格式。
3. sstWriter 打开一个文件并插入数据。生成 SST 文件时，Key 必须按照顺序写入。
4. 生成 SST 文件之后，RocksDB 通过 `IngestExternalFile()` 方法将 SST 文件导入到 NebulaGraph 中。例如：

```
IngestExternalFileOptions ifo;
# 导入两个 SST 文件
Status s = db_->IngestExternalFile({"/home/usr/file1.sst", "/home/usr/file2.sst"}, ifo);
if (!s.ok()) {
  printf("Error while adding file %s and %s, Error %s\n",
         file_path1.c_str(), file_path2.c_str(), s.ToString().c_str());
  return 1;
}
```

调用 `IngestExternalFile()` 方法时，RocksDB 默认会将文件拷贝到数据目录，并且阻塞 RocksDB 写入操作。如果 SST 文件中的键范围覆盖了 Memtable 键的范围，则将 Memtable 落盘（flush）到硬盘。将 SST 文件放置在 LSM 树最优位置后，为文件分配一个全局序列号，并打开写操作。

数据集

本文以 basketballplayer 数据集为例。

环境配置

本文示例在 MacOS 下完成，以下是相关的环境配置信息：

- 硬件规格：
- CPU：1.7 GHz Quad-Core Intel Core i7
- 内存：16 GB
- Spark：2.4.7 单机版
- Hadoop：2.9.2 伪分布式部署
- NebulaGraph： master。

前提条件

开始导入数据之前，用户需要确认以下信息：

- 已经安装部署 NebulaGraph master 并获取如下信息：
- Graph 服务和 Meta 服务的的 IP 地址和端口。
- 拥有 NebulaGraph 写权限的用户名和密码。
- Meta 服务配置文件中的 `--ws_storage_http_port` 和 Storage 服务配置文件中的 `--ws_http_port` 一致。例如都为 `19779` 。
- Graph 服务配置文件中的 `--ws_meta_http_port` 和 Meta 服务配置文件中的 `--ws_http_port` 一致。例如都为 `19559` 。
- Schema 的信息，包括 Tag 和 Edge type 的名称、属性等。
- 已经编译 Exchange，或者直接下载编译完成的 jar 文件。本示例中使用 Exchange 3.8.0。
- 已经安装 Spark。
- 已经安装 JDK 1.8 或以上版本，并配置环境变量 JAVA_HOME。
- 确认 Hadoop 服务在所有部署 Storage 服务的机器上运行正常。

> Note
>
> - 如果需要生成其他数据源的 SST 文件，请参见相应数据源的文档，查看前提条件部分。
> - 如果只需要生成 SST 文件，不需要在部署 Storage 服务的机器上安装 Hadoop 服务。
> - 如需在 INGEST（数据导入）结束后自动移除 SST 文件，在 Storage 服务配置文件中增加 `--move_files=true`，该配置会让 NebulaGraph 在 INGEST 后将 SST 文件移动（ `mv` ）到 `data` 目录下。 `--move_files` 的默认值为 `false`，此时 NebulaGraph 会复制（ `cp` ）SST 文件而不是移动。

操作步骤

步骤 1：在 **NebulaGraph** 中创建 **Schema**

分析 CSV 文件中的数据，按以下步骤在 NebulaGraph 中创建 Schema：

1. 确认 Schema 要素。 NebulaGraph 中的 Schema 要素如下表所示。

要素	名称	属性
Tag	`player`	`name string, age int`
Tag	`team`	`name string`
Edge Type	`follow`	`degree int`
Edge Type	`serve`	`start_year int, end_year int`

2. 使用 NebulaGraph Console 创建一个图空间 basketballplayer，并创建一个 Schema，如下所示。

```
## 创建图空间
nebula> CREATE SPACE basketballplayer \
        (partition_num = 10, \
        replica_factor = 1, \
        vid_type = FIXED_STRING(30));

## 选择图空间 basketballplayer
nebula> USE basketballplayer;

## 创建 Tag player
nebula> CREATE TAG player(name string, age int);

## 创建 Tag team
nebula> CREATE TAG team(name string);

## 创建 Edge type follow
nebula> CREATE EDGE follow(degree int);

## 创建 Edge type serve
nebula> CREATE EDGE serve(start_year int, end_year int);
```

更多信息，请参见快速开始。

步骤 2：处理 **CSV** 文件

确认以下信息：

1. 处理 CSV 文件以满足 Schema 的要求。

> Note
>
> 可以使用有表头或者无表头的 CSV 文件。

2. 获取 CSV 文件存储路径。

步骤 3：修改配置文件

编译 Exchange 后，复制 `target/classes/application.conf` 文件设置相关配置。在本示例中，复制的文件名为 `sst_application.conf` 。各个配置项的详细说明请参见配置说明。

```
{
  # Spark 相关配置
  spark: {
    app: {
      name: NebulaGraph Exchange 3.8.0
    }

    master:local

    driver: {
      cores: 1
      maxResultSize: 1G
    }

    executor: {
```

```
      memory:1G
  }

  cores:{
    max: 16
  }
}

# NebulaGraph 相关配置
nebula: {
  address:{
    graph:["192.168.8.XXX:9669"]
    #任意一个 Meta 服务的地址。
    #如果您的 NebulaGraph 在虚拟网络中，如k8s，请配置 Leader Meta的地址。
    meta:["192.168.8.XXX:9559"]
  }
  user: root
  pswd: nebula
  # 是否使用 RSA 加密的密码。
  # enableRSA: true
  # 使用 RSA 加密密码时的密钥。
  # privateKey: ""

  space: basketballplayer

  # SST 文件相关配置
  path:{
      # 本地临时存放生成的 SST 文件的目录
      local:"/tmp"

      # SST 文件在 HDFS 的存储路径
      remote:"/sst"

      # HDFS 的 NameNode 地址，例如 "hdfs://<ip/hostname>:<port>"。
      hdfs.namenode: "hdfs://*.*.*.*:9000"
  }

  # 客户端连接参数
  connection: {
    # socket 连接、执行的超时时间，单位：毫秒。
    timeout: 30000
  }

  error: {
    # 最大失败数，超过后会退出应用程序。
    max: 32
    # 失败的导入作业将记录在输出路径中。
    output: /tmp/errors
  }

  # 使用Google Guava RateLimiter 来限制发送到 NebulaGraph 的请求。
  rate: {
    # RateLimiter 的稳定吞吐量。
    limit: 1024

    # 从 RateLimiter 获取允许的超时时间，单位：毫秒
    timeout: 1000
  }
}

# 处理点
tags: [
  # 设置 Tag player 相关信息。
  {
    # 指定 NebulaGraph 中定义的 Tag 名称。
    name: player
    type: {
      # 指定数据源，使用 CSV。
      source: csv

      # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
      sink: sst
    }

    # 指定 CSV 文件的路径。
    # 文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://ip:port/xx/xx.csv"。
    path: "hdfs://*.*.*.*:9000/dataset/vertex_player.csv"

    # 如果 CSV 文件没有表头，使用 [_c0, _c1, _c2, ..., _cn] 表示其表头，并将列指示为属性值的源。
    # 如果 CSV 文件有表头，则使用实际的列名。
    fields: [_c1, _c2]

    # 指定 NebulaGraph 中定义的属性名称。
    # fields 与 nebula.fields 的顺序必须一一对应。
    nebula.fields: [age, name]

    # 指定一个列作为 VID 的源。
    # vertex 的值必须与上述 fields 或者 csv.fields 中的列名保持一致。
    # 目前，NebulaGraph  master仅支持字符串或整数类型的 VID。
    vertex: {
      field:_c0
    # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
    # prefix:"tag1"
    # 对 string 类型的 VID 进行哈希化操作。
```

```
      # policy:hash
      }

      # 指定的分隔符。默认值为英文逗号（,）。
      separator: ","

      # 如果 CSV 文件有表头，请将 header 设置为 true。
      # 如果 CSV 文件没有表头，请将 header 设置为 false。默认值为 false。
      header: false

      # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
      # filter: "name='Tom'"

      # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
      #writeMode: INSERT

      # 指定单批次写入 NebulaGraph 的最大点数量。
      batch: 256

      # 数据写入 NebulaGraph 时需要创建的分区数。
      partition: 32

      # 生成 SST 文件时是否要基于 NebulaGraph 中图空间的 partition 进行数据重分区。
      repartitionWithNebula: false
    }

    # 设置 Tag team 相关信息。
    {
      name: team
      type: {
        source: csv
        sink: sst
      }
      path: "hdfs://*.*.*.*:9000/dataset/vertex_team.csv"
      fields: [_c1]
      nebula.fields: [name]
      vertex: {
        field:_c0
      }
      separator: ","
      header: false
      batch: 256
      partition: 32
      repartitionWithNebula: false
    }

    # 如果需要添加更多点，请参考前面的配置进行添加。
  ]
  # 处理边
  edges: [
    # 设置 Edge type follow 相关信息。
    {
      # 指定 NebulaGraph 中定义的 Edge type 名称。
      name: follow
      type: {
        # 指定数据源，使用 CSV。
        source: csv

        # 指定如何将点数据导入 NebulaGraph ：Client 或 SST。
        sink: sst
      }

      # 指定 CSV 文件的路径。
      # 文件存储在 HDFS 上，用双引号括起路径，以 hdfs://开头，例如"hdfs://<ip/hostname>:port/xx/xx.csv"。
      path: "hdfs://*.*.*.*:9000/dataset/edge_follow.csv"

      # 如果 CSV 文件没有表头，使用 [_c0, _c1, _c2, ..., _cn] 表示其表头，并将列指示为属性值的源。
      # 如果 CSV 文件有表头，则使用实际的列名。
      fields: [_c2]

      # 指定 NebulaGraph 中定义的属性名称。
      # fields 与 nebula.fields 的顺序必须一一对应。
      nebula.fields: [degree]

      # 指定一个列作为起始点和目的点的源。
      # vertex 的值必须与上述 fields 或者 csv.fields 中的列名保持一致。
      # 目前，NebulaGraph  master仅支持字符串或整数类型的 VID。
      source: {
        field: _c0
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }
      target: {
        field: _c1
      # 为 VID 增加指定的前缀。例如 VID 为 12345，增加前缀 tag1 后为 tag1_12345。下划线无法修改。
      # prefix:"tag1"
      # 对 string 类型的 VID 进行哈希化操作。
      # policy:hash
      }

      # 指定的分隔符。默认值为英文逗号（,）。
      separator: ","
```

```
      # 指定一个列作为 rank 的源（可选）。

      #ranking: rank

      # 如果 CSV 文件有表头，请将 header 设置为 true。
      # 如果 CSV 文件没有表头，请将 header 设置为 false。默认值为 false。
      header: false

      # 过滤规则。符合过滤规则的数据会被导入NebulaGraph。
      # filter: "name='Tom'"

      # 批量操作类型，包括 INSERT、UPDATE 和 DELETE。默认为 INSERT。
      #writeMode: INSERT

      # 指定单批次写入 NebulaGraph 的最大边数量。
      batch: 256

      # 数据写入 NebulaGraph 时需要创建的分区数。
      partition: 32

      # 生成 SST 文件时是否要基于 NebulaGraph 中图空间的 partition 进行数据重分区。
      repartitionWithNebula: false
    }

    # 设置 Edge type serve 相关信息。
    {
      name: serve
      type: {
        source: csv
        sink: sst
      }
      path: "hdfs://*.*.*.*:9000/dataset/edge_serve.csv"
      fields: [_c2,_c3]
      nebula.fields: [start_year, end_year]
      source: {
        field: _c0
      }
      target: {
        field: _c1
      }
      separator: ","
      header: false
      batch: 256
      partition: 32
      repartitionWithNebula: false
    }

  ]
  # 如果需要添加更多边，请参考前面的配置进行添加。
}
```

步骤 **4**：生成 **SST** 文件

运行如下命令将 CSV 源文件生成为 SST 文件。关于参数的说明，请参见命令参数。

```
${SPARK_HOME}/bin/spark-submit --master "local" --conf spark.sql.shuffle.partitions=<shuffle_concurrency> --class com.vesoft.nebula.exchange.Exchange <nebula-exchange.jar_path> -c
<sst_application.conf_path>
```

> Note
>
> 生成 SST 文件时，会涉及到 Spark 的 shuffle 操作，请注意在提交命令中增加 `spark.sql.shuffle.partitions` 的配置。

> Note
>
> JAR 包有两种获取方式：自行编译或者从 maven 仓库下载。

示例：

```
${SPARK_HOME}/bin/spark-submit --master "local" --conf spark.sql.shuffle.partitions=200 --class com.vesoft.nebula.exchange.Exchange /root/nebula-exchange/nebula-exchange/target/nebula-
exchange_spark_2.4-3.8.0.jar -c /root/nebula-exchange/nebula-exchange/target/classes/sst_application.conf
```

任务执行完成后，可以在 HDFS 上的 `/sst` 目录（`nebula.path.remote` 参数指定）内查看到生成的 SST 文件。

> Note
>
> 如果对 Schema 有修改操作，例如重建图空间、修改 Tag、修改 Edge type 等，需要重新生成 SST 文件，因为 SST 文件会验证 Space ID、Tag ID、Edge ID 等信息。

步骤 5：导入 **SST** 文件

> Note
>
> 导入前请确认以下信息：
>
> - 确认所有部署 Storage 服务的机器上都已部署 Hadoop 服务，并配置 HADOOP_HOME 和 JAVA_HOME。
> - Meta 服务配置文件中的 `--ws_storage_http_port`（如果没有，请手动添加）和 Storage 服务配置文件中的 `--ws_http_port` 一致。例如都为 `19779`。
> - Graph 服务配置文件中的 `--ws_meta_http_port`（如果没有，请手动添加）和 Meta 服务配置文件中的 `--ws_http_port` 一致。例如都为 `19559`。

使用客户端工具连接 NebulaGraph，按如下操作导入 SST 文件：

1. 执行命令选择之前创建的图空间。

```
nebula> USE basketballplayer;
```

2. 执行命令下载 SST 文件：

```
nebula> SUBMIT JOB DOWNLOAD HDFS "hdfs://<hadoop_address>:<hadoop_port>/<sst_file_path>";
```

示例：

```
nebula> SUBMIT JOB DOWNLOAD HDFS "hdfs://*.*.*.*:9000/sst";
```

3. 执行命令导入 SST 文件：

```
nebula> SUBMIT JOB INGEST;
```

> Note
>
> - 如果需要重新下载，请在 NebulaGraph 安装路径内的 `data/storage/nebula` 目录内，将对应 Space ID 目录内的 `download` 文件夹删除，然后重新下载 SST 文件。如果图空间是多副本，保存副本的所有机器都需要删除 `download` 文件夹。
> - 如果导入时出现问题需要重新导入，重新执行 `SUBMIT JOB INGEST;` 即可。

步骤 6：（可选）验证数据

用户可以在 NebulaGraph 客户端（例如 NebulaGraph Studio）中执行查询语句，确认数据是否已导入。例如：

```
LOOKUP ON player YIELD id(vertex);
```

用户也可以使用命令 `SHOW STATS` 查看统计数据。

步骤 7：（如有）在 **NebulaGraph** 中重建索引

导入数据后，用户可以在 NebulaGraph 中重新创建并重建索引。详情请参见索引介绍。

最后更新: 2025年8月4日

11.3.5 Exchange 常见问题

编译问题

Q：部分非 CENTRAL 仓库的包下载失败，报错 `COULD NOT RESOLVE DEPENDENCIES FOR PROJECT XXX`

请检查 Maven 安装目录下 `libexec/conf/settings.xml` 文件的 `mirror` 部分：

```
<mirror>
    <id>alimaven</id>
    <mirrorOf>central</mirrorOf>
    <name>aliyun maven</name>
    <url>http://maven.aliyun.com/nexus/content/repositories/central/</url>
</mirror>
```

检查 `mirrorOf` 的值是否配置为 `*`，如果为 `*`，请修改为 `central` 或 `*,!SparkPackagesRepo,!bintray-streamnative-maven`。

原因：Exchange 的 `pom.xml` 中有两个依赖包不在 Maven 的 central 仓库中，`pom.xml` 配置了这两个依赖所在的仓库地址。如果 maven 中配置的镜像地址对应的 `mirrorOf` 值为 `*`，那么所有依赖都会在 central 仓库下载，导致下载失败。

Q：编译 EXCHANGE 时无法下载 SNAPSHOT 包

现象：编译时提示 `Could not find artifact com.vesoft:client:jar:xxx-SNAPSHOT`。

原因：本地 maven 没有配置用于下载 SNAPSHOT 的仓库。maven 中默认的 central 仓库用于存放正式发布版本，而不是开发版本（SNAPSHOT）。

解决方案：在 maven 的 setting.xml文件的 `profiles` 作用域内中增加以下配置：

```
  <profile>
     <activation>
        <activeByDefault>true</activeByDefault>
     </activation>
     <repositories>
        <repository>
            <id>snapshots</id>
            <url>https://oss.sonatype.org/content/repositories/snapshots/</url>
            <snapshots>
               <enabled>true</enabled>
            </snapshots>
      </repository>
     </repositories>
  </profile>
```

执行问题

Q：报错 `JAVA.LANG.CLASSNOTFOUNDEXCEPTION: COM.VESOFT.NEBULA.EXCHANGE.EXCHANGE`

在 Yarn-Cluster 模式下提交任务，请参考如下命令，尤其是示例中的两个 `--conf`：

```
$SPARK_HOME/bin/spark-submit --class com.vesoft.nebula.exchange.Exchange \
--master yarn-cluster \
--files application.conf \
--conf spark.driver.extraClassPath=./ \
--conf spark.executor.extraClassPath=./ \
nebula-exchange-3.0.0.jar \
-c application.conf
```

Q：报错 `METHOD NAME XXX NOT FOUND`

一般是端口配置错误，需检查 Meta 服务、Graph 服务、Storage 服务的端口配置。

Q：报 NOSUCHMETHOD、METHODNOTFOUND 错误（`EXCEPTION IN THREAD "MAIN" JAVA.LANG.NOSUCHMETHODERROR` 等）

绝大多数是因为 JAR 包冲突和版本冲突导致的报错，请检查报错服务的版本，与 Exchange 中使用的版本进行对比，检查是否一致，尤其是 Spark 版本、Scala 版本、Hive 版本。

Q：EXCHANGE 导入 HIVE 数据时报错 `EXCEPTION IN THREAD "MAIN" ORG.APACHE.SPARK.SQL.ANALYSISEXCEPTION: TABLE OR VIEW NOT FOUND`

检查提交 exchange 任务的命令中是否遗漏参数 `-h`，检查 table 和 database 是否正确，在 spark-sql 中执行用户配置的 exec 语句，验证 exec 语句的正确性。

Q：运行时报错 `COM.FACEBOOK.THRIFT.PROTOCOL.TPROTOCOLEXCEPTION: EXPECTED PROTOCOL ID XXX`

请检查 NebulaGraph 服务端口配置是否正确。

- 如果是源码、RPM 或 DEB 安装，请配置各个服务的配置文件中 `--port` 对应的端口号。
- 如果是 docker 安装，请配置 docker 映射出来的端口号，查看方式如下：

  在 `nebula-docker-compose` 目录下执行 `docker-compose ps`，例如：

```
$ docker-compose ps
      Name             Command         State                          Ports
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
nebula-docker-compose_graphd_1   /usr/local/nebula/bin/nebu ...  Up (healthy)  0.0.0.0:33205->19669/tcp, 0.0.0.0:33204->19670/tcp, 0.0.0.0:9669->9669/tcp
nebula-docker-compose_metad0_1   ./bin/nebula-metad --flagf ...  Up (healthy)  0.0.0.0:33165->19559/tcp, 0.0.0.0:33162->19560/tcp, 0.0.0.0:33167->9559/tcp, 9560/tcp
nebula-docker-compose_metad1_1   ./bin/nebula-metad --flagf ...  Up (healthy)  0.0.0.0:33166->19559/tcp, 0.0.0.0:33163->19560/tcp, 0.0.0.0:33168->9559/tcp, 9560/tcp
nebula-docker-compose_metad2_1   ./bin/nebula-metad --flagf ...  Up (healthy)  0.0.0.0:33161->19559/tcp, 0.0.0.0:33160->19560/tcp, 0.0.0.0:33164->9559/tcp, 9560/tcp
nebula-docker-compose_storaged0_1  ./bin/nebula-storaged --fl ...  Up (healthy)  0.0.0.0:33180->19779/tcp, 0.0.0.0:33178->19780/tcp, 9777/tcp, 9778/tcp, 0.0.0.0:33183->9779/tcp, 9780/tcp
nebula-docker-compose_storaged1_1  ./bin/nebula-storaged --fl ...  Up (healthy)  0.0.0.0:33175->19779/tcp, 0.0.0.0:33172->19780/tcp, 9777/tcp, 9778/tcp, 0.0.0.0:33177->9779/tcp, 9780/tcp
nebula-docker-compose_storaged2_1  ./bin/nebula-storaged --fl ...  Up (healthy)  0.0.0.0:33184->19779/tcp, 0.0.0.0:33181->19780/tcp, 9777/tcp, 9778/tcp, 0.0.0.0:33185->9779/tcp, 9780/tcp
```

  查看 `Ports` 列，查找 docker 映射的端口号，例如：

  - Graph 服务可用的端口号是 9669。

  - Meta 服务可用的端口号有 33167、33168、33164。

  - Storage 服务可用的端口号有 33183、33177、33185。

Q：运行时报错 `EXCEPTION IN THREAD "MAIN" COM.FACEBOOK.THRIFT.PROTOCOL.TPROTOCOLEXCEPTION: THE FIELD 'CODE' HAS BEEN ASSIGNED THE INVALID VALUE -4`

检查 Exchange 版本与 NebulaGraph 版本是否匹配，详细信息可参考使用限制。

Q：SPARK 环境中导入数据时出现乱码如何解决?

如果数据的属性值包含中文字符，可能出现乱码。解决方案是在导入命令中的 JAR 包路径前加上以下选项：

```
--conf spark.driver.extraJavaOptions=-Dfile.encoding=utf-8
--conf spark.executor.extraJavaOptions=-Dfile.encoding=utf-8
```

即：

```
<spark_install_path>/bin/spark-submit --master "local" \
--conf spark.driver.extraJavaOptions=-Dfile.encoding=utf-8 \
--conf spark.executor.extraJavaOptions=-Dfile.encoding=utf-8 \
--class com.vesoft.nebula.exchange.Exchange \
<nebula-exchange-3.x.y.jar_path> -c <application.conf_path>
```

如果是在 YARN 中，则用以下命令：

```
<spark_install_path>/bin/spark-submit \
--class com.vesoft.nebula.exchange.Exchange \
--master yarn-cluster \
--files <application.conf_path> \
--conf spark.driver.extraClassPath=./ \
--conf spark.executor.extraClassPath=./ \
--conf spark.driver.extraJavaOptions=-Dfile.encoding=utf-8 \
--conf spark.executor.extraJavaOptions=-Dfile.encoding=utf-8 \
<nebula-exchange-3.x.y.jar_path> \
-c application.conf
```

Q：HIVE 数据导入时提示 SCHEMA 版本不一致

Spark 日志提示 `Hive Schema version 1.2.0 does not match metastore's schema version 2.1.0 Metastore is not upgraded or corrupt` 的原因是 Hive 环境中配置的 metastore schema 版本和 Spark 使用的 metastore 版本不一致。

解决方法：

1. 将 Hive 环境中存储 Hive metastore 信息的 MySQL version 信息更新为 Spark 中使用的 metastore 版本。

   假设 Hive 在 MySQL 中存储 metastore 的数据库是 `hive`，需要按如下方式修改 `hive.VERSION` 表中的 `version` 字段：

```
update hive.VERSION set SCHEMA_VERSION="2.1.0" where VER_ID=1
```

2. 在 Hive 环境的 `hive-site.xml` 文件中增加如下配置：

```
<property>
<name>hive.metastore.schema.verification</name>
<value>false</value>
</property>
```

3. 重启 Hive。

Q: 生成 SST 时提示 ORG.ROCKSDB.ROCKSDBEXCEPTION: WHILE OPEN A FILE FOR APPENDING: /PATH/SST/1-XXX.SST: NO SUCH FILE OR DIRECTORY

排查方法：

1. 检查 `/path` 是否存在，如没有或者路径设置错误，创建或修正路径。
2. 检查 Spark 在每台机器上的当前用户对 `/path` 是否有操作权限，如没有，添加权限。

配置问题

Q：哪些配置项影响导入性能?

- batch：每次发送给 NebulaGraph 服务的 nGQL 语句中包含的数据条数。
- partition：数据写入 NebulaGraph 时需要创建的分区数，表示数据导入的并发数。
- nebula.rate：向 NebulaGraph 发送请求前先去令牌桶获取令牌。

  - limit：表示令牌桶的大小。

  - timeout：表示获取令牌的超时时间。

根据机器性能可适当调整这四项参数的值。如果在导入过程中，Storage 服务的 leader 变更，可以适当调小这四项参数的值，降低导入速度。

其他问题

Q：EXCHANGE 支持哪些版本的 NEBULAGRAPH？

请参见 Exchange 的使用限制。

Q：EXCHANGE 与 SPARK WRITER 有什么关系?

Exchange 是在 Spark Writer 基础上开发的 Spark 应用程序，二者均适用于在分布式环境中将集群的数据批量迁移到 NebulaGraph 中，但是后期的维护工作将集中在 Exchange 上。与 Spark Writer 相比，Exchange 有以下改进：

- 支持更丰富的数据源，如 MySQL、Neo4j、Hive、HBase、Kafka、Pulsar 等。
- 修复了 Spark Writer 的部分问题。例如 Spark 读取 HDFS 里的数据时，默认读取到的源数据均为 String 类型，可能与 NebulaGraph 定义的 Schema 不同，所以 Exchange 增加了数据类型的自动匹配和类型转换，当 NebulaGraph 定义的 Schema 中数据类型为非 String 类型（如 double）时，Exchange 会将 String 类型的源数据转换为对应的类型（如 double）。

Q：EXCHANGE 传输数据的性能如何?

Exchange 的性能测试数据和测试方法参见 NebulaGraph Exchange test result。

最后更新: 2025年8月4日

12. 连接器

12.1 NebulaGraph Spark Connector

NebulaGraph Spark Connector 是一个 Spark 连接器，提供通过 Spark 标准形式读写 NebulaGraph 数据的能力。NebulaGraph Spark Connector 由 Reader 和 Writer 两部分组成。

- Reader

  提供一个 Spark SQL 接口，用户可以使用该接口编程读取 NebulaGraph 图数据，单次读取一个点或 Edge type 的数据，并将读取的结果组装成 Spark 的 DataFrame。

- Writer

  提供一个 Spark SQL 接口，用户可以使用该接口编程将 DataFrame 格式的数据逐条或批量写入 NebulaGraph 。

更多使用说明请参见 NebulaGraph Spark Connector。

12.1.1 版本兼容性

NebulaGraph Spark Connector、 NebulaGraph 内核版本和 Spark 版本对应关系如下。

Spark Connector 版本	NebulaGraph 版本	Spark 版本
nebula-spark-connector_3.0-3.0-SNAPSHOT.jar	nightly	3.x
nebula-spark-connector_2.2-3.0-SNAPSHOT.jar	nightly	2.2.x
nebula-spark-connector-3.0-SNAPSHOT.jar	nightly	2.4.x
nebula-spark-connector_3.0-3.8.0.jar	3.x	3.x
nebula-spark-connector_2.2-3.8.0.jar	3.x	2.2.x
nebula-spark-connector-3.8.0.jar	3.x	2.4.x
nebula-spark-connector_3.0-3.6.0.jar	3.x	3.x
nebula-spark-connector_2.2-3.6.0.jar	3.x	2.2.x
nebula-spark-connector-3.6.0.jar	3.x	2.4.x
nebula-spark-connector_2.2-3.4.0.jar	3.x	2.2.x
nebula-spark-connector-3.4.0.jar	3.x	2.4.x
nebula-spark-connector_2.2-3.3.0.jar	3.x	2.2.x
nebula-spark-connector-3.3.0.jar	3.x	2.4.x
nebula-spark-connector-3.0.0.jar	3.x	2.4.x
nebula-spark-connector-2.6.1.jar	2.6.0, 2.6.1	2.4.x
nebula-spark-connector-2.6.0.jar	2.6.0, 2.6.1	2.4.x
nebula-spark-connector-2.5.1.jar	2.5.0, 2.5.1	2.4.x
nebula-spark-connector-2.5.0.jar	2.5.0, 2.5.1	2.4.x
nebula-spark-connector-2.1.0.jar	2.0.0, 2.0.1	2.4.x
nebula-spark-connector-2.0.1.jar	2.0.0, 2.0.1	2.4.x
nebula-spark-connector-2.0.0.jar	2.0.0, 2.0.1	2.4.x

12.1.2 适用场景

NebulaGraph Spark Connector 适用于以下场景：

- 读取NebulaGraph数据进行分析计算。
- 分析计算完的数据写入NebulaGraph。
- 迁移NebulaGraph数据。
- 结合 NebulaGraph Algorithm 进行图计算。

12.1.3 特性

NebulaGraph Spark Connector 3.8.0版本特性如下：

- 提供多种连接配置项，如超时时间、连接重试次数、执行重试次数等。
- 提供多种数据配置项，如写入数据时设置对应列为点 ID、起始点 ID、目的点 ID 或属性。
- Reader 支持无属性读取和全属性读取。
- Reader 支持将 NebulaGraph 数据读取成 Graphx 的 VertexRDD 和 EdgeRDD，支持非 Long 型点 ID。
- 统一了 SparkSQL 的扩展数据源，统一采用 DataSourceV2 进行 NebulaGraph 数据扩展。
- 支持 `insert`、`update` 和 `delete` 三种写入模式。`insert` 模式会插入（覆盖）数据，`update` 模式仅会更新已存在的数据，`delete` 模式只删除数据。
- 支持与 NebulaGraph 之间的 SSL 加密连接。

12.1.4 更新说明

Release notes

12.1.5 获取 NebulaGraph Spark Connector

编译打包

1. 克隆仓库 `nebula-spark-connector`。

```
$ git clone -b release-3.8 https://github.com/vesoft-inc/nebula-spark-connector.git
```

2. 进入目录 `nebula-spark-connector`。
3. 编译打包。不同版本的 Spark 命令略有不同。

> Note
>
> 需已安装对应版本 Spark。

- Spark 2.4

```
$ mvn clean package -Dmaven.test.skip=true -Dgpg.skip -Dmaven.javadoc.skip=true -pl nebula-spark-connector -am -Pscala-2.11 -Pspark-2.4
```

- Spark 2.2

```
$ mvn clean package -Dmaven.test.skip=true -Dgpg.skip -Dmaven.javadoc.skip=true -pl nebula-spark-connector_2.2 -am -Pscala-2.11 -Pspark-2.2
```

- Spark 3.x

```
$ mvn clean package -Dmaven.test.skip=true -Dgpg.skip -Dmaven.javadoc.skip=true -pl nebula-spark-connector_3.0 -am -Pscala-2.12 -Pspark-3.0
```

编译完成后，在目录的文件夹 `target` 下生成类似文件 `nebula-spark-connector-3.8.0-SHANPSHOT.jar`。

Maven 远程仓库下载

下载地址

12.1.6 使用方法

使用 NebulaGraph Spark Connector 读写 NebulaGraph 时，只需要编写以下代码即可实现。

```
# 从 NebulaGraph 读取点边数据。
spark.read.nebula().loadVerticesToDF()
spark.read.nebula().loadEdgesToDF()

# 将 dataframe 数据作为点和边写入 NebulaGraph 中。
dataframe.write.nebula().writeVertices()
dataframe.write.nebula().writeEdges()
```

`nebula()` 接收两个配置参数，包括连接配置和读写配置。

> **Note**
>
> 如果数据的属性值包含中文字符，可能出现乱码。请在提交 Spark 任务时加上以下选项：
>
> ```
> --conf spark.driver.extraJavaOptions=-Dfile.encoding=utf-8
> --conf spark.executor.extraJavaOptions=-Dfile.encoding=utf-8
> ```

从 **NebulaGraph** 读取数据

```
val config = NebulaConnectionConfig
  .builder()
  .withMetaAddress("127.0.0.1:9559")
  .withConenctionRetry(2)
  .withExecuteRetry(2)
  .withTimeout(6000)
  .build()

val nebulaReadVertexConfig: ReadNebulaConfig = ReadNebulaConfig
  .builder()
  .withUser("root")
  .withPasswd("nebula")
  .withSpace("test")
  .withLabel("person")
  .withNoColumn(false)
  .withReturnCols(List("birthday"))
  .withLimit(10)
  .withPartitionNum(10)
  .build()
val vertex = spark.read.nebula(config, nebulaReadVertexConfig).loadVerticesToDF()

val nebulaReadEdgeConfig: ReadNebulaConfig = ReadNebulaConfig
  .builder()
  .withUser("root")
  .withPasswd("nebula")
  .withSpace("test")
  .withLabel("knows")
  .withNoColumn(false)
  .withReturnCols(List("degree"))
  .withLimit(10)
  .withPartitionNum(10)
```

```
 .build()
val edge = spark.read.nebula(config, nebulaReadEdgeConfig).loadEdgesToDF()
```

- `NebulaConnectionConfig` 是连接 NebulaGraph 的配置，说明如下。

参数	是否必须	说明
`withMetaAddress`	是	所有 Meta 服务的地址，多个地址用英文逗号（,）隔开，格式为 `ip1:port1,ip2:port2,...`。读取数据不需要配置 `withGraphAddress`。
`withConnectionRetry`	否	NebulaGraph Java Client 连接 NebulaGraph 的重试次数。默认值为 `1`。
`withExecuteRetry`	否	NebulaGraph Java Client 执行查询语句的重试次数。默认值为 `1`。
`withTimeout`	否	NebulaGraph Java Client 请求响应的超时时间。默认值为 `6000`，单位：毫秒（ms）。

- `ReadNebulaConfig` 是读取 NebulaGraph 数据的配置，说明如下。

参数	是否必须	说明
`withUser`	否	NebulaGraph用户名。Storage 服务要求身份认证时，需要填写该参数。仅 NebulaGraph 企业版支持该参数。
`withPasswd`	否	NebulaGraph用户名对应的密码。Storage 服务要求身份认证时，需要填写该参数。仅 NebulaGraph 企业版支持该参数。
`withSpace`	是	NebulaGraph 图空间名称。
`withLabel`	是	NebulaGraph 图空间内的 Tag 或 Edge type 名称。
`withNoColumn`	否	是否不读取属性。默认值为 `false`，表示读取属性。取值为 `true` 时，表示不读取属性，此时 `withReturnCols` 配置无效。
`withReturnCols`	否	配置要读取的点或边的属性集。格式为 `List(property1,property2,...)`，默认值为 `List()`，表示读取全部属性。
`withLimit`	否	配置 NebulaGraph Java Storage Client 一次从服务端读取的数据行数。默认值为 1000。
`withPartitionNum`	否	配置读取 NebulaGraph 数据时 Spark 的分区数。默认值为 100。该值的配置最好不超过图空间的的分片数量（partition_num）。

向 **NebulaGraph** 写入数据

> Note
>
> DataFrame 中的列会自动作为属性写入 NebulaGraph。

```
val config = NebulaConnectionConfig
 .builder()
 .withMetaAddress("127.0.0.1:9559")
 .withGraphAddress("127.0.0.1:9669")
 .withConenctionRetry(2)
 .build()

val nebulaWriteVertexConfig: WriteNebulaVertexConfig = WriteNebulaVertexConfig
 .builder()
 .withSpace("test")
 .withTag("person")
 .withVidField("id")
 .withVidPolicy("hash")
 .withVidAsProp(true)
 .withUser("root")
 .withPasswd("nebula")
 .withBatch(512)
 .build()
df.write.nebula(config, nebulaWriteVertexConfig).writeVertices()

val nebulaWriteEdgeConfig: WriteNebulaEdgeConfig = WriteNebulaEdgeConfig
 .builder()
 .withSpace("test")
 .withEdge("friend")
```

```
  .withSrcIdField("src")
  .withSrcPolicy(null)
  .withDstIdField("dst")
  .withDstPolicy(null)
  .withRankField("degree")
  .withSrcAsProperty(true)
  .withDstAsProperty(true)
  .withRankAsProperty(true)
  .withUser("root")
  .withPasswd("nebula")
  .withBatch(512)
  .build()
df.write.nebula(config, nebulaWriteEdgeConfig).writeEdges()
```

默认写入模式为 `insert`，可以通过 `withWriteMode` 配置修改为 `update` 或 `delete`：

```
val config = NebulaConnectionConfig
  .builder()
  .withMetaAddress("127.0.0.1:9559")
  .withGraphAddress("127.0.0.1:9669")
  .build()
val nebulaWriteVertexConfig = WriteNebulaVertexConfig
  .builder()
  .withSpace("test")
  .withTag("person")
  .withVidField("id")
  .withVidAsProp(true)
  .withBatch(512)
  .withWriteMode(WriteMode.UPDATE)
```

```
  .build()
df.write.nebula(config, nebulaWriteVertexConfig).writeVertices()
```

- `NebulaConnectionConfig` 是连接 NebulaGraph 的配置，说明如下。

参数	是否必须	说明
`withMetaAddress`	是	所有 Meta 服务的地址，多个地址用英文逗号（,）隔开，格式为 `ip1:port1,ip2:port2,...`。
`withGraphAddress`	是	Graph 服务的地址，多个地址用英文逗号（,）隔开，格式为 `ip1:port1,ip2:port2,...`。
`withConnectionRetry`	否	NebulaGraph Java Client 连接 NebulaGraph 的重试次数。默认值为 `1`。

- `WriteNebulaVertexConfig` 是写入点的配置，说明如下。

参数	是否必须	说明
`withSpace`	是	NebulaGraph 图空间名称。
`withTag`	是	写入点时需要关联的 Tag 名称。
`withVidField`	是	DataFrame 中作为点 ID 的列。
`withVidPolicy`	否	写入点 ID 时，采用的映射函数，NebulaGraph 仅支持 HASH。默认不做映射。
`withVidAsProp`	否	DataFrame 中作为点 ID 的列是否也作为属性写入。默认值为 `false`。如果配置为 `true`，请确保 Tag 中有和 `VidField` 相同的属性名。
`withUser`	否	NebulaGraph 用户名。若未开启身份验证，无需配置用户名和密码。
`withPasswd`	否	NebulaGraph 用户名对应的密码。
`withBatch`	是	一次写入的数据行数，默认值为 `512`。当 `withWriteMode` 为 `update` 时，该参数的最大值为 `512`。
`withWriteMode`	否	写入模式。可选值为 `insert`、`update` 和 `delete`。默认为 `insert`。
`withDeleteEdge`	否	删除点时是否删除该点关联的边。默认为 `false`。当 `withWriteMode` 为 `delete` 时生效。

- `WriteNebulaEdgeConfig` 是写入边的配置，说明如下。

参数	是否必须	说明
`withSpace`	是	NebulaGraph 图空间名称。
`withEdge`	是	写入边时需要关联的 Edge type 名称。
`withSrcIdField`	是	DataFrame 中作为起始点的列。
`withSrcPolicy`	否	写入起始点时，采用的映射函数，NebulaGraph 仅支持 HASH。默认不做映射。
`withDstIdField`	是	DataFrame 中作为目的点的列。
`withDstPolicy`	否	写入目的点时，采用的映射函数，NebulaGraph 仅支持 HASH。默认不做映射。
`withRankField`	否	DataFrame 中作为 rank 的列。默认不写入 rank。
`withSrcAsProperty`	否	DataFrame 中作为起始点的列是否也作为属性写入。默认值为 `false`。如果配置为 `true`，请确保 Edge type 中有和 `SrcIdField` 相同的属性名。
`withDstAsProperty`	否	DataFrame 中作为目的点的列是否也作为属性写入。默认值为 `false`。如果配置为 `true`，请确保 Edge type 中有和 `DstIdField` 相同的属性名。
`withRankAsProperty`	否	DataFrame 中作为 rank 的列是否也作为属性写入。默认值为 `false`。如果配置为 `true`，请确保 Edge type 中有和 `RankField` 相同的属性名。
`withUser`	否	NebulaGraph 用户名。若未开启身份验证，无需配置用户名和密码。
`withPasswd`	否	NebulaGraph 用户名对应的密码。
`withBatch`	是	一次写入的数据行数，默认值为 `512`。当 `withWriteMode` 为 `update` 时，该参数的最大值为 `512`。
`withWriteMode`	否	写入模式。可选值为 `insert`、`update` 和 `delete`。默认为 `insert`。

示例代码

详细的使用方式参见 示例代码。

最后更新: 2025年8月4日

12.2 NebulaGraph Flink Connector

NebulaGraph Flink Connector 是一款帮助 Flink 用户快速访问NebulaGraph的连接器，支持从NebulaGraph图数据库中读取数据，或者将其他外部数据源读取的数据写入NebulaGraph图数据库。

12.2.1 适用场景

NebulaGraph Flink Connector 适用于以下场景：

- 读取NebulaGraph数据进行分析计算。
- 分析计算完的数据写入NebulaGraph。
- 迁移数据。

12.2.2 更新说明

Release notes

12.2.3 版本兼容性

NebulaGraph Flink Connector 和NebulaGraph内核版本对应关系如下。

Flink Connector 版本	NebulaGraph版本
3.0-SNAPSHOT	nightly
3.8.0	3.x.x
3.5.0	3.x.x
3.3.0	3.x.x
3.0.0	3.x.x
2.6.1	2.6.0、2.6.1
2.6.0	2.6.0、2.6.1
2.5.0	2.5.0、2.5.1
2.0.0	2.0.0、2.0.1

12.2.4 前提条件

- 已安装 Java 8 或更高版本。
- 已安装 Flink 1.11.x。

12.2.5 获取 NebulaGraph Flink Connector

设置 **Maven** 依赖

在 Maven 的配置文件 `pom.xml` 里添加以下依赖自动获取 Flink Connector.

```
<dependency>
  <groupId>com.vesoft</groupId>
  <artifactId>nebula-flink-connector</artifactId>
  <version>3.8.0</version>
</dependency>
```

编译打包

按照以下步骤自行编译打包 Flink Connector。

1. 克隆仓库 `nebula-flink-connector`。

```
$ git clone -b release-3.8 https://github.com/vesoft-inc/nebula-flink-connector.git
```

2. 进入目录 `nebula-flink-connector`。
3. 编译打包。

```
$ mvn clean package -Dmaven.test.skip=true
```

编译完成后，在目录的文件夹 `connector/target` 下生成类似文件 `nebula-flink-connector-3.8.0.jar`。

12.2.6 使用方法

向**NebulaGraph**写入数据

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
NebulaClientOptions nebulaClientOptions = new NebulaClientOptions.NebulaClientOptionsBuilder()
            .setGraphAddress("127.0.0.1:9669")
            .setMetaAddress("127.0.0.1:9559")
            .build();
NebulaGraphConnectionProvider graphConnectionProvider = new NebulaGraphConnectionProvider(nebulaClientOptions);
NebulaMetaConnectionProvider metaConnectionProvider = new NebulaMetaConnectionProvider(nebulaClientOptions);

VertexExecutionOptions executionOptions = new VertexExecutionOptions.ExecutionOptionBuilder()
            .setGraphSpace("flinkSink")
            .setTag("player")
            .setIdIndex(0)
            .setFields(Arrays.asList("name", "age"))
            .setPositions(Arrays.asList(1, 2))
            .setBatchSize(2)
            .build();

NebulaVertexBatchOutputFormat outputFormat = new NebulaVertexBatchOutputFormat(
            graphConnectionProvider, metaConnectionProvider, executionOptions);
NebulaSinkFunction<Row> nebulaSinkFunction = new NebulaSinkFunction<>(outputFormat);
DataStream<Row> dataStream = playerSource.map(row -> {
         Row record = new org.apache.flink.types.Row(row.size());
         for (int i = 0; i < row.size(); i++) {
            record.setField(i, row.get(i));
         }
         return record;
      });
dataStream.addSink(nebulaSinkFunction);
env.execute("write nebula")
```

从**NebulaGraph**读取数据

```
NebulaClientOptions nebulaClientOptions = new NebulaClientOptions.NebulaClientOptionsBuilder()
      .setMetaAddress("127.0.0.1:9559")
      .build();
storageConnectionProvider = new NebulaStorageConnectionProvider(nebulaClientOptions);
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.setParallelism(1);

VertexExecutionOptions vertexExecutionOptions = new VertexExecutionOptions.ExecutionOptionBuilder()
      .setGraphSpace("flinkSource")
      .setTag("person")
      .setNoColumn(false)
      .setFields(Arrays.asList())
      .setLimit(100)
      .build();

NebulaSourceFunction sourceFunction = new NebulaSourceFunction(storageConnectionProvider)
      .setExecutionOptions(vertexExecutionOptions);
DataStreamSource<BaseTableRow> dataStreamSource = env.addSource(sourceFunction);
dataStreamSource.map(row -> {
   List<ValueWrapper> values = row.getValues();
   Row record = new Row(15);
   record.setField(0, values.get(0).asLong());
   record.setField(1, values.get(1).asString());
   record.setField(2, values.get(2).asString());
   record.setField(3, values.get(3).asLong());
   record.setField(4, values.get(4).asLong());
   record.setField(5, values.get(5).asLong());
   record.setField(6, values.get(6).asLong());
   record.setField(7, values.get(7).asDate());
```

```
    record.setField(8, values.get(8).asDateTime().getUTCDateTimeStr());
    record.setField(9, values.get(9).asLong());
    record.setField(10, values.get(10).asBoolean());
    record.setField(11, values.get(11).asDouble());
    record.setField(12, values.get(12).asDouble());
    record.setField(13, values.get(13).asTime().getUTCTimeStr());
    record.setField(14, values.get(14).asGeography());
    return record;
}).print();
env.execute("NebulaStreamSource");
```

参数说明

- `NebulaClientOptions` 是连接NebulaGraph的配置，说明如下。

参数	类型	是否必须	说明
`setGraphAddress`	String	是	NebulaGraph Graph 服务地址。
`setMetaAddress`	String	是	NebulaGraph Meta 服务地址。

- `VertexExecutionOptions` 是执行点读写的配置，说明如下。

参数	类型	是否必须	说明
`setGraphSpace`	String	是	图空间名称。
`setTag`	String	是	Tag 名称。
`setIdIndex`	Int	是	向NebulaGraph写入数据时作为 VID 的流数据字段下标。
`setFields`	List	是	Tag 的属性名集合。用于向NebulaGraph写入数据或从NebulaGraph读取数据。读取时需要确保 `setNoColumn` 为 `false`，否则配置无效。读取时本参数为空，表示读取所有属性。
`setPositions`	List	是	流数据字段下标的集合。表示将对应的字段值作为属性值写入NebulaGraph。需要和 `setFields` 一一对应。
`setBatchSize`	String	否	每次写入NebulaGraph的最大数据记录条数。默认值为 `2000`。
`setNoColumn`	String	否	读取数据时设置为 `true` 则不会读取属性。默认值为 `false`。
`setLimit`	String	否	读取数据时每次拉取的最大数据记录条数。默认值为 `2000`。

- `EdgeExecutionOptions` 是执行边读写的配置，说明如下。

参数	类型	是否必须	说明
`setGraphSpace`	String	是	图空间名称。
`setEdge`	String	是	Edge type 名称。
`setSrcIndex`	Int	是	向NebulaGraph写入数据时作为起始点 VID 的流数据字段下标。
`setDstIndex`	Int	是	向NebulaGraph写入数据时作为目的点 VID 的流数据字段下标。
`setRankIndex`	Int	是	向NebulaGraph写入数据时作为边的 Rank 的流数据字段下标。
`setFields`	List	是	Edge type 属性名集合。用于向NebulaGraph写入数据或从NebulaGraph读取数据。读取时需要确保 `setNoColumn` 为 `false`，否则配置无效。读取时本参数为空，表示读取所有属性。
`setPositions`	List	是	流数据字段下标的集合。表示将对应的字段值作为属性值写入NebulaGraph。需要和 `setFields` 一一对应。
`setBatchSize`	String	否	每次写入NebulaGraph的最大数据记录条数。默认值为 `2000`。
`setNoColumn`	String	否	读取数据时设置为 `true` 则不会读取属性。默认值为 `false`。
`setLimit`	String	否	读取数据时每次拉取的最大数据记录条数。默认值为 `2000`。

12.2.7 示例

1. 创建图空间。

```
NebulaCatalog nebulaCatalog = NebulaCatalogUtils.createNebulaCatalog(
    "NebulaCatalog",
    "default",
    "root",
    "nebula",
    "127.0.0.1:9559",
    "127.0.0.1:9669");

EnvironmentSettings settings = EnvironmentSettings.newInstance()
    .inStreamingMode()
    .build();
TableEnvironment tableEnv = TableEnvironment.create(settings);

tableEnv.registerCatalog(CATALOG_NAME, nebulaCatalog);
tableEnv.useCatalog(CATALOG_NAME);

String createDataBase = "CREATE DATABASE IF NOT EXISTS `db1` "
    + " COMMENT 'space 1'"
    + " WITH ("
    + " 'partition_num' = '100',"
    + " 'replica_factor' = '3',"
    + " 'vid_type' = 'FIXED_STRING(10)'"
    + ")";
tableEnv.executeSql(createDataBase);
```

2. 创建 Tag。

```
tableEnvironment.executeSql("CREATE TABLE `person` ("
    + " vid BIGINT,"
    + " col1 STRING,"
    + " col2 STRING,"
    + " col3 BIGINT,"
    + " col4 BIGINT,"
    + " col5 BIGINT,"
    + " col6 BIGINT,"
    + " col7 DATE,"
    + " col8 TIMESTAMP,"
    + " col9 BIGINT,"
    + " col10 BOOLEAN,"
    + " col11 DOUBLE,"
    + " col12 DOUBLE,"
    + " col13 TIME,"
    + " col14 STRING"
    + ") WITH ("
    + " 'connector' = 'nebula',"
    + " 'meta-address' = '127.0.0.1:9559',"
    + " 'graph-address' = '127.0.0.1:9669',"
    + " 'username' = 'root',"
    + " 'password' = 'nebula',"
    + " 'data-type' = 'vertex',"
    + " 'graph-space' = 'flink_test',"
    + " 'label-name' = 'person'"
    + ")"
);
```

3. 创建 Edge type。

```
tableEnvironment.executeSql("CREATE TABLE `friend` ("
    + " sid BIGINT,"
    + " did BIGINT,"
    + " rid BIGINT,"
    + " col1 STRING,"
    + " col2 STRING,"
    + " col3 BIGINT,"
    + " col4 BIGINT,"
    + " col5 BIGINT,"
    + " col6 BIGINT,"
    + " col7 DATE,"
    + " col8 TIMESTAMP,"
    + " col9 BIGINT,"
    + " col10 BOOLEAN,"
    + " col11 DOUBLE,"
    + " col12 DOUBLE,"
    + " col13 TIME,"
    + " col14 STRING"
    + ") WITH ("
    + " 'connector' = 'nebula',"
    + " 'meta-address' = '127.0.0.1:9559',"
    + " 'graph-address' = '127.0.0.1:9669',"
    + " 'username' = 'root',"
    + " 'password' = 'nebula',"
    + " 'graph-space' = 'flink_test',"
    + " 'label-name' = 'friend',"
    + " 'data-type'='edge',"
    + " 'src-id-index'='0',"
    + " 'dst-id-index'='1',"
    + " 'rank-id-index'='2'"
    + ")"
);
```

4. 查询边数据并插入到另一个边类型中。

```
Table table = tableEnvironment.sqlQuery("SELECT * FROM `friend`");
table.executeInsert("`friend_sink`").await();
```

最后更新: 2025年8月4日

13. 最佳实践

13.1 Compaction

本文介绍 Compaction 的相关信息。

NebulaGraph 中，`Compaction` 是最重要的后台操作，对性能有极其重要的影响。

`Compaction` 操作会读取硬盘上的数据，然后重组数据结构和索引，然后再写回硬盘，可以成倍提升读取性能。将大量数据写入 NebulaGraph 后，为了提高读取性能，需要手动触发 `Compaction` 操作（全量 `Compaction`）。

> **Note**
>
> `Compaction` 操作会长时间占用硬盘的 IO，建议在业务低峰期（例如凌晨）执行该操作。

NebulaGraph 有两种类型的 `Compaction` 操作：自动 `Compaction` 和全量 `Compaction`。

13.1.1 自动 `Compaction`

自动 `Compaction` 是在系统读取数据、写入数据或系统重启时自动触发 `Compaction` 操作，提升短时间内的读取性能。默认情况下，自动 `Compaction` 是开启状态，可能在业务高峰期触发，导致意外抢占 IO 影响业务。

13.1.2 全量 `Compaction`

全量 `Compaction` 可以对图空间进行大规模后台操作，例如合并文件、删除 TTL 过期数据等，该操作需要手动发起。使用如下语句执行全量 `Compaction` 操作：

> **Note**
>
> 建议在业务低峰期（例如凌晨）执行该操作，避免大量占用硬盘 IO 影响业务。

```
nebula> USE <your_graph_space>;
nebula> SUBMIT JOB COMPACT;
```

上述命令会返回作业的 ID，用户可以使用如下命令查看 `Compaction` 状态：

```
nebula> SHOW JOB <job_id>;
```

13.1.3 操作建议

为保证 NebulaGraph 的性能，请参考如下操作建议：

- 数据导入完成后，请执行 `SUBMIT JOB COMPACT`。
- 业务低峰期（例如凌晨）执行 `SUBMIT JOB COMPACT`。
- 为控制 `Compaction` 的写入速率，请在配置文件 `nebula-storaged.conf` 中设置如下参数（注：此参数限制全部写入，包括正常写入和 Compaction）：

```
# 写入速度限制为 20MB/S。
--rocksdb_rate_limit=20 (in MB/s)
```

13.1.4 FAQ

Compaction 相关的日志在哪?

默认情况下，`/usr/local/nebula/data/storage/nebula/{1}/data/` 目录下的文件名为 `LOG` 文件，或者类似 `LOG.old.1625797988509303`，找到如下的部分。

```
** Compaction Stats [default] **
Level    Files   Size     Score Read(GB)  Rn(GB) Rnp1(GB) Write(GB) Wnew(GB) Moved(GB) W-Amp Rd(MB/s) Wr(MB/s) Comp(sec) CompMergeCPU(sec) Comp(cnt) Avg(sec) KeyIn KeyDrop
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  L0      2/0    2.46 KB   0.5      0.0     0.0      0.0       0.0      0.0       0.0   1.0      0.0      0.0      0.53              0.51         2    0.264       0      0
 Sum      2/0    2.46 KB   0.0      0.0     0.0      0.0       0.0      0.0       0.0   1.0      0.0      0.0      0.53              0.51         2    0.264       0      0
 Int      0/0    0.00 KB   0.0      0.0     0.0      0.0       0.0      0.0       0.0   0.0      0.0      0.0      0.00              0.00         0    0.000       0      0
```

如果当前的 `L0` 文件数量较多，对读性能影响较大，可以触发 compaction。

可以同时在多个图空间执行全量 `Compaction` 操作吗?

可以，但是此时的硬盘 IO 会很高，可能会影响效率。

全量 `Compaction` 操作会耗费多长时间?

如果已经设置读写速率限制，例如 `rocksdb_rate_limit` 限制为 20MB/S 时，用户可以通过 `硬盘使用量/rocksdb_rate_limit` 预估需要耗费的时间。 如果没有设置读写速率限制，根据经验，速率大约为 50MB/S。

可以动态调整 `rocksdb_rate_limit` 吗?

不可以。

全量 `Compaction` 操作开始后可以停止吗?

不可以停止，必须等待操作完成。这是 RocksDB 的限制。

最后更新: 2025年8月4日

13.2 Storage 负载均衡

用户可以使用 `SUBMIT JOB BALANCE` 语句平衡 Raft leader 的分布。详情请参见 SUBMIT JOB BALANCE。

> **Danger**
>
> `SUBMIT JOB BALANCE` 命令通过创建和执行一组子任务来迁移数据和均衡分片分布，禁止停止集群中的任何机器或改变机器的 IP 地址，直到所有子任务完成，否则后续子任务会失败。

13.2.1 均衡 leader 分布

用户可以使用命令 `SUBMIT JOB BALANCE LEADER` 均衡分布所有图空间中的 Leader 分片副本。

示例

```
nebula> SUBMIT JOB BALANCE LEADER;
```

用户可以执行 `SHOW HOSTS` 检查结果。

```
nebula> SHOW HOSTS;
+------------------+------+----------+--------------+-----------------------------------+------------------------+---------+
| Host             | Port | Status   | Leader count | Leader distribution               | Partition distribution | Version |
+------------------+------+----------+--------------+-----------------------------------+------------------------+---------+
| "192.168.10.100" | 9779 | "ONLINE" | 4            | "basketballplayer:3"              | "basketballplayer:8"   | "master" |
| "192.168.10.101" | 9779 | "ONLINE" | 8            | "basketballplayer:3"              | "basketballplayer:8"   | "master" |
| "192.168.10.102" | 9779 | "ONLINE" | 3            | "basketballplayer:3"              | "basketballplayer:8"   | "master" |
| "192.168.10.103" | 9779 | "ONLINE" | 0            | "basketballplayer:2"              | "basketballplayer:7"   | "master" |
| "192.168.10.104" | 9779 | "ONLINE" | 0            | "basketballplayer:2"              | "basketballplayer:7"   | "master" |
| "192.168.10.105" | 9779 | "ONLINE" | 0            | "basketballplayer:2"              | "basketballplayer:7"   | "master" |
+------------------+------+----------+--------------+-----------------------------------+------------------------+---------+
```

> **Caution**
>
> 在 NebulaGraph master 中，Leader 分片副本切换期间，Leader 分片副本会被暂时禁止写入直到切换完成。如果在 Leader 分片副本切换期间，有大量的写入请求，将会导致请求错误（Storage Error `E_RPC_FAILURE`），错误处理方法见 FAQ。
>
> 用户可以在 Storage 配置文件中设置 `raft_heartbeat_interval_secs` 的值来控制 Leader 副本切换的超时时间。有关配置文件的详细信息，请参见 Storage 配置文件。

最后更新: 2025年8月4日

13.3 图建模设计

本文介绍在 NebulaGraph 项目中成功应用的一些图建模和系统设计的通用建议。

> Note
>
> 本文建议是通用的，在特定领域有例外，请结合实际业务情况进行图建模。

13.3.1 以性能为第一目标进行建模

目前 NebulaGraph 没有完美的建模方法，如何建模取决于想从数据中挖掘的内容。分析数据并根据业务模型创建方便直观的数据模型，测试模型并优化，逐渐适应业务。为了更好的性能，用户可以多次更改或重新设计模型。

设计和评估最重要的查询语句

在测试环节中，通常会验证各种各样的查询语句，以全面评估系统能力。但在大多数生产场景下，每个集群被频繁调用的查询语句的类型并不会太多；根据 20-80 原则，针对重要的查询语句进行建模优化。

避免全量扫描

通过属性索引或者 VID 来先定位到某个（些）点或者边，然后开始图遍历；对于有些查询，它们只给定了一个子图或者路径的（正则）模式，但无法通过属性索引或者 VID 定位到遍历起始的点边，而期望找到库中全部满足该模式的子图，这样的查询是通过全量扫描实现的，这样的性能会很差。NebulaGraph 没有实现对于子图或者路径的图结构的索引。

Tag 与 **Edge type** 之间没有绑定关系

任何 Tag 可以与任何 Edge type 相关联，完全交由应用程序控制。不需要在 NebulaGraph 中预先定义，也没有命令获取哪些 Tag 与哪些 Edge type 相关联。

Tag/Edge type 预先定义了一组属性

建立 Tag（或者 Edge type）时，需要指定对应的属性。通常称为 Schema。

区分"经常改变的部分"和"不经常改变的部分"

改变指的是业务模型和数据模型上的改变（元信息），不是数据自身的改变。

一些图数据库产品是 schema-free 的设计，所以在数据模型上，不论是图拓扑结构还是属性，都可以非常自由。属性可以建模转变为图拓扑，反之亦然。这类系统通常对于图拓扑的访问有特别的优化。

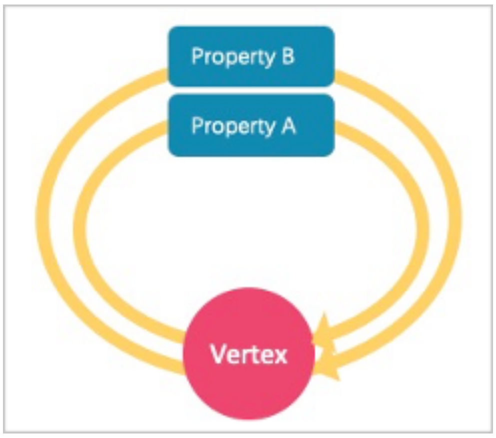
而 NebulaGraph master 是强 Schema 的（行存型）系统，这意味着业务数据模型中的部分是“不应该经常改变的”，例如属性 Schema 应该避免改变。类似于 MySQL 中 ALTER TABLE 是应该尽量避免的操作。

而点及邻边可以非常低成本的增删，因此可以将业务模型中“经常改变的部分”建模成点或边（关系），而不是属性 Schema。

例如，在一个业务模型中，人的属性是相对固定的，例如“年龄”，“性别”，“姓名”。而“通信好友”，“出入场所”，“交易账号”，“登录设备”等是相对容易改变的。前者适合建模为属性，后者适合建模为点或边。

自环

NebulaGraph 是强 Schema 类型系统，使用 `ALTER TAG` 的开销很大，而且也不支持List类型属性，当用户需要为点添加一些临时属性或者List类型的属性时，可以先创建包含所需属性的边类型，然后为点插入一条或多条指向自身的边。查询时只需要查询指向自己的边属性。如下图所示。



示例：

如果要检索点的临时属性，请从自环边中获取。例如：

```
//创建边类型并插入自环属性。
nebula> CREATE EDGE IF NOT EXISTS temp(tmp int);
nebula> INSERT EDGE temp(tmp) VALUES "player100"->"player100"@1:(1);
nebula> INSERT EDGE temp(tmp) VALUES "player100"->"player100"@2:(2);
nebula> INSERT EDGE temp(tmp) VALUES "player100"->"player100"@3:(3);

//插入数据后，可以通过查询语句查询，例如：
nebula> GO FROM "player100" OVER temp YIELD properties(edge).tmp;
+----------------------+
| properties(EDGE).tmp |
+----------------------+
| 1                    |
| 2                    |
| 3                    |
+----------------------+

//如果需要返回结果为List，可以通过函数实现，例如：
nebula> MATCH (v1:player)-[e:temp]->() return collect(e.tmp);
+----------------+
| collect(e.tmp) |
+----------------+
| [1, 2, 3]      |
+----------------+
```

对于自环的操作没有封装任何的语法糖，操作方式与普通的边无异。

- NebulaGraph 自环小科普（2 分 54 秒）

悬挂边

悬挂边 (Dangling edge) 是指一条起点或者终点不存在于数据库中的边。

在 NebulaGraph master中，有两种情况可能会出现悬挂边。

第一种情况：在起点和终点插入之前，用 INSERT EDGE 语句插入一条边。

第二种情况：使用 DELETE VERTEX 语句删除点的时候，没有使用 `WITH EDGE` 选项。此时系统默认不删除该点关联的出边和入边，这些边将变成悬挂边。

NebulaGraph master 的数据模型中，由于设计允许图中存在“悬挂边”；没有 openCypher 中的 MERGE 语句。对于悬挂边的保证完全依赖应用层面。用户可以使用 GO 和 LOOKUP 语句查询到悬挂边，但无法使用 MATCH 语句查询到悬挂边。

示例：

```
// 插入起点为“11”，终点为“13”并且都不存在于数据库中的悬挂边
nebula> CREATE EDGE IF NOT EXISTS e1 (name string, age int);
nebula> INSERT EDGE e1 (name, age) VALUES "11"->"13":("n1", 1);

// 使用 GO 语句查询
nebula> GO FROM "11" over e1 YIELD properties(edge);
+----------------------+
| properties(EDGE)     |
```

```
+---------------------+
| {age: 1, name: "n1"} |
+---------------------+

// 使用 LOOKUP 语句查询
nebula> LOOKUP ON e1 YIELD EDGE AS r;
+-------------------------------------------------------+
| r                          |
+-------------------------------------------------------+
| [:e2 "11"->"13" @0 {age: 1, name: "n1"}]       |
+-------------------------------------------------------+

// 使用 MATCH 查询，不能查询到悬挂边
nebula> MATCH ()-[e:e1]->() RETURN e;
+---+
| e |
+---+
+---+
Empty set (time spent 3153/3573 us)
```

- NebulaGraph 的悬挂边小科普（2 分 28 秒）

广度优先大于深度优先

- NebulaGraph 基于图拓扑结构进行深度图遍历的性能较低，广度优先遍历以及获取属性的性能较好。例如，模型 a 包括姓名、年龄、眼睛颜色三种属性，建议创建一个 Tag `person`，然后为它添加姓名、年龄、眼睛颜色的属性。如果创建一个包含眼睛颜色的 Tag 和一个 Edge type `has`，然后创建一个边用来表示人拥有的眼睛颜色，这种建模方法会降低遍历性能。
- “通过边属性获取边”的性能与“通过点属性获取点”的性能是接近的。在一些数据库中，会建议将边上的属性重新建模为中间节点的属性：例如 `(src)-[edge {P1, P2}]->(dst)`，`edge` 上有属性 `P1, P2`，会建议建模为 `(src)-[edge1]->(i_node {P1, P2})-[edge2]->(dst)`。在 NebulaGraph master 中可以直接使用 `(src)-[edge {P1, P2}]->(dst)`，减少遍历深度有助于性能。

边的方向

查询时，如果需要使用边的逆向查询，可以用如下语法：

`(dst)<-[edge]-(src)` 或者 `GO FROM dst REVERSELY`；

如果不关心边的方向，可以使用如下语法：

`(src)-[edge]-(dst)` 或者 `GO FROM src BIDIRECT`；

因此，通常同一条边没有必要反向再冗余插入一次。

合理设置 **Tag** 属性

在图建模中，请将一组类似的平级属性放入同一个 Tag，即按不同概念进行分组。

正确使用索引

使用属性索引可以通过属性查找到 VID。但是索引会导致写性能大幅下降，只有在根据点或边的属性定位点或边时才使用索引。

合理设计 **VID**

参考点 VID 一节。

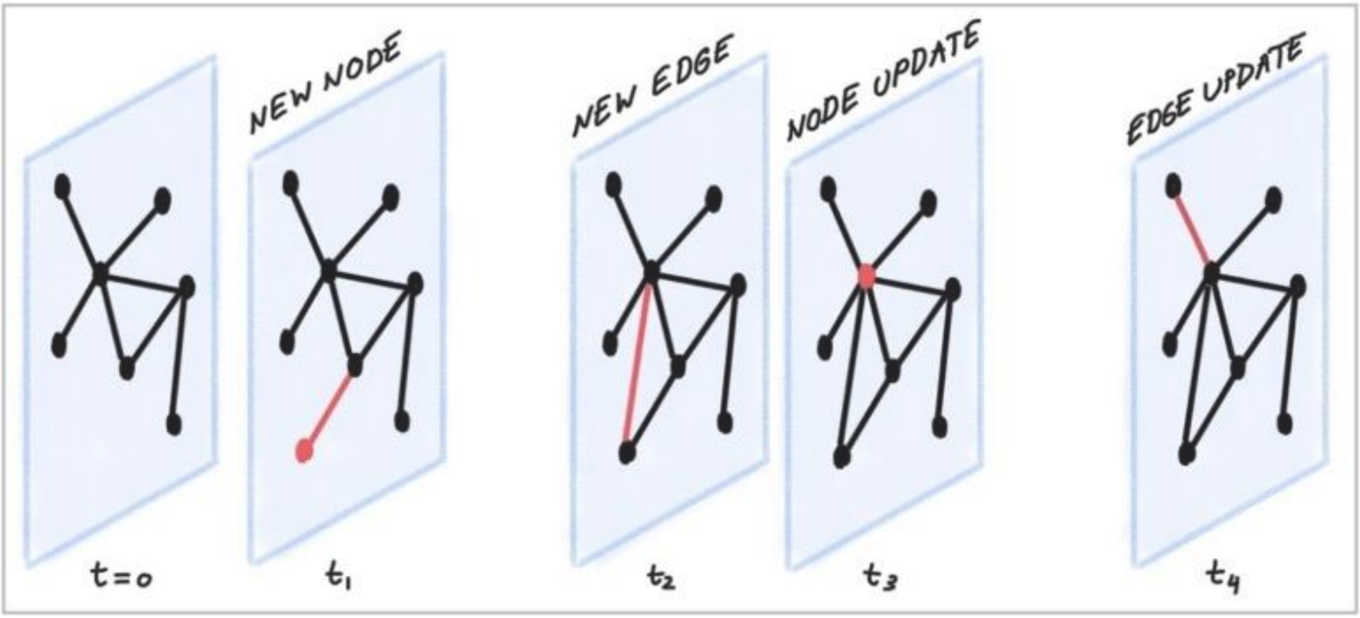
长文本

为边创建属性时请勿使用长文本：这些属性会被存储 2 份，导致写入放大问题（write amplification）。此时建议将长文本放在 HBase/ES 中，将其地址存放在 NebulaGraph 中。

13.3.2 关于支持动态图（时序图）

在某些场景下，图需要同时带有时序信息，以描述整个图的结构随着时间变化的情况 [1]。

NebulaGraph master 的边可以使用 Rank 字段存放时间信息 (int64)，但是点上没有字段可以存放时间信息（存放在属性会被新写入覆盖）。一个折中的办法是在点上设计自己指向自己的自环，并将时间信息放置在自环的 Rank 上。



13.3.3 一些免费的建模工具

arrows.app

---

1. https://blog.twitter.com/engineering/en_us/topics/insights/2021/temporal-graph-networks ↩

最后更新: 2025年8月4日

13.4 系统设计建议

13.4.1 选择 QPS 优先或时延优先

- NebulaGraph master 更擅长处理（互联网式的）有大量并发的小请求。也即：虽然全图很大（万亿点边），但是每个请求要访问到的子图本身并不大（几百万个点边）——单个请求时延不大；但这类请求的并发数量特别多——QPS 大。

- 但对于一些交互分析型的场景，并发请求的数量不多，而每个请求要访问的子图本身特别大（亿以上）。为降低时延，可以在应用程序中将一个大的请求，拆分为多个小请求，并发发送给多个 graphd。这样可以降低单个大请求的时延，降低单个 graphd 的内存占用。另外，也可以使用图计算功能 NebulaGraph Algorithm。

13.4.2 数据传输与优化

- 读写平衡。 NebulaGraph 适合读写平衡性的在线场景，也即 OLTP 型的的“并发的发生写入与读取”；而非数仓 OLAP 型的“一次写入多次读取”。
- 选择不同的写入方式。大批量的数据写入可以使用 sst 加载的方式；小批量的写入使用 `INSERT` 语句。
- 选择合适的时间运行 COMPACTION 和 BALANCE，来分别优化数据格式和存储分布。
- NebulaGraph master 不支持关系型数据库意义上的事务和隔离性，更接近 NoSQL。

13.4.3 查询预热与数据预热

应用端进行预热：

- Graphd 不支持预编译查询及相应生成查询计划，也不支持缓存之前的查询结果；
- Storaged 不支持预热数据，只有 RocksDB 自身的 LSM-tree 和 BloomFilter 会启动时加载到内存中。
- 点和边被访问过后，会各自缓存在 Storaged 的两种 (LRU) Cache 中。

最后更新: 2025年8月4日

13.5 执行计划

NebulaGraph master 实现了基于规则的执行计划。用户无法改变执行计划，无法进行查询的预编译（及相应的计划缓存），无法通过指定索引来加速查询。

要查看执行计划及执行概要，请参考 EXPLAIN 和 PROFILE。

最后更新: 2025年8月4日

13.6 超级顶点（稠密点）处理

13.6.1 原理介绍

在图论中，超级顶点（稠密点）是指一个点有着极多的相邻边。相邻边可以是出边（我指向谁）或者是入边（谁指向我）。

由于幂律分布的特点，超级顶点现象非常普遍。 例如社交网络中的影响力领袖（网红大 V）、证券市场中的热门股票、银行系统中的四大行、交通网络中的枢纽站、互联网中的高流量网站等、电商网络中的爆款产品。

在 NebulaGraph master 中，一个 `点` 和其 `属性` 是一个 `Key-Value`（以该点的 `VID` 以及其他元信息作为 `Key`），其 `Out-Edge Key-Value` 和 `In-Edge Key-Value` 都存储在同一个 partition 中（具体原理详见存储架构，并且以 LSM-tree 的形式组织存放在硬盘（和缓存）中。

因此不论是 `从该点出发的有向遍历`，或者 `以该点为终点的有向遍历`，都会涉及到大量的 `顺序 IO 扫描`（最理想情况，当完成 Compact 操作之后），或者大量的 `随机 IO`（有关于 `该点` 和其 `出入边` 频繁的写入）。

经验上说，当一个点的出入度超过 1 万时，就可以视为是稠密点。需要考虑一些特殊的设计和处理。

> Note
>
> NebulaGraph 中没有专用的字段来记录每个点的出度和入度，也没有内置任务来进行统计，因此无法预知哪些点会是超级节点。一个折中的办法是使用 Spark 周期性地计算和统计。

重复属性索引

在属性图中，除了网络拓扑结构中的超级顶点，还有一类情况类似于超级顶点———某属性有极高重复率，也即"相同的 `点类型 Tag`，不同的 `顶点 VID`，同一属性字段，拥有相同属性值"。

NebulaGraph master 属性索引的设计复用了存储模块 RocksDB 的功能，这种情况下的索引会被建模为 `前缀相同的 Key`。对于该属性的查找，（如果未能命中缓存，）会对应为硬盘上的“一次随机查找 + 一次前缀顺序扫描”，以找到对应的 `点 VID`（此后，通常会从该顶点开始图遍历，这样又会发生该点对应 Key-Value 的一次随机读+顺序扫描）。当重复率越高，扫描范围就越大。

关于属性索引的原理详细介绍在博客《分布式图数据库 NebulaGraph 的 Index 实践》。

经验上说，当重复属性值超过 1 万时，也需要特殊的设计和处理。

建议的办法

数据库端的常见办法

1. 截断: 只访问一定阈值的边，超过该阈值的其他边则不返回。
2. Compact：重新组织 RocksDB 中数据的排列方式，减少随机读，增加顺序读。

应用端的常见办法

根据业务意义，将一些超级顶点拆分：

- 删除多条边，合并为一条

  例如，一个转账场景： `(账户 A)-[转账]->(账户 B)`。每次 `转账` 建模为 `一条 AB 之间的边`，那么 `(账户 A)` 和 `(账户 B)` 之间会有着数万十次转账的场景。

  按日、周、或者月为粒度，合并陈旧的转账明细。也就是批量删除陈旧的边，改为少量的边“月总额”和“次数。而保留最近月的转账明细。

- 拆分相同类型的边，变为多种不同类型的边

  例如， `(机场)<-[depart]-(航班)` 场景，每个架次航班的离港，都建模为一条航班和机场之间的边。那么大型机场的离港航班会极多。

  根据不同的 `航空公司` 将 `depart` 这个 Edge type 拆分更细的 Edge type，如 `depart_ceair`，`depart_csair` 等。在查询（图遍历）时，指定离港的航空公司。

- 切分顶点本身

  例如，对于 `(人)-[借款]->(银行)` 的借款网络，某大型银行 A 的借款次数和借款人会非常的多。

  可以将该大行节点 A 拆分为多个相关联的子节点 A1、A2、A3,

  ```
  (人 1)-[借款]->(银行 A1), (人 2)-[借款]->(银行 A2), (人 2)-[借款]->(银行 A3);
  (银行 A1)-[属于]->(银行 A), (银行 A2)-[属于]->(银行 A), (银行 A3)-[属于]->(银行 A).
  ```

  这里的 A1、A2、A3 既可以是 A 真实的三个分行（例如北京、上海、浙江），也可以是三个按某种规则设立的虚拟分行，例如按借款金额划分 `A1: 1-1000, A2: 1001-10000, A3: 10000+`。这样，查询时对于 A 的任何操作，都转变为为对于 A1、A2、A3 的三次单独操作。

最后更新: 2025年8月4日

13.7 启用 AutoFDO

AutoFDO 可以对优化过的程序进行性能分析，并使用性能信息来指导编译器再次优化程序。本文将帮助您为 NebulaGraph 启用AutoFDO。

关于 AutoFDO 的更多信息，请参见 AutoFDO Wiki。

13.7.1 准备资源

安装依赖

- 安装 perf

```
sudo apt-get update
sudo apt-get install -y linux-tools-common \
linux-tools-generic \
linux-tools-`uname -r`
```

- 安装 autofdo tool

```
sudo apt-get update
sudo apt-get install -y autofdo
```

  或者你可以从源代码编译 autofdo tool。

编译 **NebulaGraph** 二进制文件

关于如何从源码编译 NebulaGraph，请参考使用源码安装 NebulaGraph 。

在配置步骤中，将 `CMAKE_BUILD_TYPE=Release` 替换为 `CMAKE_BUILD_TYPE=RelWithDebInfo` ：

```
$ cmake -DCMAKE_INSTALL_PREFIX=/usr/local/nebula -DENABLE_TESTING=OFF -DCMAKE_BUILD_TYPE=RelWithDebInfo ..
```

13.7.2 准备测试数据

在测试环境中，我们使用 NebulaGraph Bench 来准备测试数据，并通过运行 FindShortestPath、Go1Step、Go2Step、Go3Step、InsertPersonScenario 这5个场景脚本来收集性能数据。

> Note
>
> 可以在生产环境中使用 TopN 查询来收集性能数据，在你的环境中可以提高更多性能。

13.7.3 准备性能数据

收集 **AutoFDO** 工具的性能数据

1. 测试数据准备完成后，收集不同场景的性能数据。首先获取 `storaged` 、 `graphd` 、 `metad` 的 `pid` 。

```
$ nebula.service status all
[INFO] nebula-metad: Running as 305422, Listening on 9559
[INFO] nebula-graphd: Running as 305516, Listening on 9669
[INFO] nebula-storaged: Running as 305707, Listening on 9779
```

2. 为 `nebula-graphd` 和 `nebula-storaged` 启动 `perf record` 。

```
perf record -p 305516,305707 -b -e br_inst_retired.near_taken:pp -o ~/FindShortestPath.data
```

> Note
>
> 因为与 `nebula-graphd` 和 `nebula-storaged` 相比， `nebula-metad` 的贡献率很小。为了减少工作量，我们没有收集 `nebula-metad` 的性能数据。

3. 启动 FindShortestPath 场景的基准测试。

```
cd NebulaGraph-Bench
python3 run.py stress run -s benchmark -scenario find_path.FindShortestPath -a localhost:9669 --args='-u 100 -i 100000'
```

4. 测试完成后，按 Ctrl + C 结束 `perf record`。
5. 重复上述步骤为 Go1Step、Go2Step、Go3Step、InsertPersonScenario 这4个场景收集性能数据。

创建 **Gcov** 文件

```
create_gcov --binary=$NEBULA_HOME/bin/nebula-storaged \
--profile=~/FindShortestPath.data \
--gcov=~/FindShortestPath-storaged.gcov \
-gcov_version=1

create_gcov --binary=$NEBULA_HOME/bin/nebula-graphd \
--profile=~/FindShortestPath.data \
--gcov=~/FindShortestPath-graphd.gcov \
-gcov_version=1
```

按照上面 FindShortestPath 的例子，为 Go1Step、Go2Step、Go3Step、InsertPersonScenario 这4个场景也创建 Gcov 文件。

合并性能数据

```
profile_merger ~/FindShortestPath-graphd.gcov \
~/FindShortestPath-storaged.gcov \
~/go1step-storaged.gcov \
~/go1step-graphd.gcov \
~/go2step-storaged.gcov \
~/go2step-graphd.gcov \
~/go3step-storaged.gcov \
~/go3step-master-graphd.gcov \
~/InsertPersonScenario-storaged.gcov \
~/InsertPersonScenario-graphd.gcov
```

合并后的配置文件名称为 `fbdata.afdo`。

13.7.4 使用合并的性能数据文件重新编译 NebulaGraph 二进制文件

使用编译选项 `-fauto-profile` 重新编译 NebulaGraph 二进制文件。

```
diff --git a/cmake/nebula/GeneralCompilerConfig.cmake b/cmake/nebula/GeneralCompilerConfig.cmake
@@ -20,6 +20,8 @@ add_compile_options(-Wshadow)
 add_compile_options(-Wnon-virtual-dtor)
 add_compile_options(-Woverloaded-virtual)
 add_compile_options(-Wignored-qualifiers)
+add_compile_options(-fauto-profile=~/fbdata.afdo)
```

> Note
>
> 当你使用多个 `fbdata.afdo` 多次编译时，请在重新编译之前执行 `make clean` 操作，因为只是修改 `fbdata.afdo` 不会触发重新编译。

13.7.5 性能测试结果

软硬件环境

类型	环境
CPU Processor#	2
Sockets	2
NUMA	2
CPU Type	Intel(R) Xeon(R) Platinum 8380 CPU @ 2.30GHz
Cores per Processor	40C80T
Cache	L1 data: 48KB L1 i: 32KB L2: 1.25MB per physical core L3: shared 60MB per processor
Memory	Micron DDR4 3200MT/s 16GB16Micron DDR4 3200MT/s 16GB16
SSD Disk	INTEL SSDPE2KE016T8
SSD R/W Sequential	3200 MB/s (read) / 2100 MB/s(write)
Nebula Version	master with commit id 51d84a4ed7d2a032a337e3b996c927e3bc5d1415
Kernel	4.18.0-408.el8.x86_64

测试结果

场景	Average Latency(LiB)	Default Binary	Optimized Binary with AutoFDO	P95 Latency (LiB)	Default Binary	Optimized Binary with AutoFDO
FindShortestPath	1	8072.52	7260.10	1	22102.00	19108.00
	2	8034.32	7218.59	2	22060.85	19006.00
	3	8079.27	7257.24	3	22147.00	19053.00
	4	8087.66	7221.39	4	22143.00	19050.00
	5	8044.77	7239.85	5	22181.00	19055.00
	STDDEVP	20.57	17.34	STDDEVP	41.41	32.36
	Mean	8063.71	7239.43	Mean	22126.77	19054.40
	STDDEVP/ Mean	0.26%	0.24%	STDDEVP/ Mean	0.19%	0.17%
	Opt/Default	100.00%	10.22%	Opt/ Default	100.00%	13.89%
Go1Step	1	422.53	418.37	1	838.00	850.00
	2	432.37	402.44	2	866.00	815.00
	3	437.45	407.98	3	874.00	836.00
	4	429.16	408.38	4	858.00	838.00
	5	446.38	411.32	5	901.00	837.00
	STDDEVP	8.02	5.20	STDDEVP	20.63	11.30
	Mean	433.58	409.70	Mean	867.40	835.20
	STDDEVP/ Mean	1.85%	1.27%	STDDEVP/ Mean	2.38%	1.35%
	Opt/Default	100.00%	5.51%	Opt/ Default	100.00%	3.71%
Go2Step	1	2989.93	2824.29	1	10202.00	9656.95
	2	2957.22	2834.55	2	10129.00	9632.40
	3	2962.74	2818.62	3	10168.40	9624.70
	4	2992.39	2817.27	4	10285.10	9647.50
	5	2934.85	2834.91	5	10025.00	9699.65
	STDDEVP	21.53	7.57	STDDEVP	85.62	26.25
	Mean	2967.43	2825.93	Mean	10161.90	9652.24
	STDDEVP/ Mean	0.73%	0.27%	STDDEVP/ Mean	0.84%	0.27%
	Opt/Default	100.00%	4.77%	Opt/ Default	100.00%	5.02%
Go3Step	1	93551.97	89406.96	1	371359.55	345433.50
	2	92418.43	89977.25	2	368868.00	352375.20
	3	92587.67	90339.25	3	365390.15	356198.55

场景	Average Latency(LiB)	Default Binary	Optimized Binary with AutoFDO	P95 Latency (LiB)	Default Binary	Optimized Binary with AutoFDO
	4	93371.64	92458.95	4	373578.15	365177.75
	5	94046.05	89943.44	5	373392.25	352576.00
	STDDEVP	609.07	1059.54	STDDEVP	3077.38	6437.52
	Mean	93195.15	90425.17	Mean	370517.62	354352.20
	STDDEVP/ Mean	0.65%	1.17%	STDDEVP/ Mean	0.83%	1.82%
	Opt/Default	100.00%	2.97%	Opt/ Default	100.00%	4.36%
InsertPerson	1	2022.86	1937.36	1	2689.00	2633.45
	2	1966.05	1935.41	2	2620.45	2555.00
	3	1985.25	1953.58	3	2546.00	2593.00
	4	2026.73	1887.28	4	2564.00	2394.00
	5	2007.55	1964.41	5	2676.00	2581.00
	STDDEVP	23.02	26.42	STDDEVP	57.45	82.62
	Mean	2001.69	1935.61	Mean	2619.09	2551.29
	STDDEVP/ Mean	1.15%	1.37%	STDDEVP/ Mean	2.19%	3.24%
	Opt/Default	100.00%	3.30%	Opt/ Default	100.00%	2.59%

最后更新: 2025年8月4日

13.8 实践案例

NebulaGraph 在各行各业都有应用，本文介绍部分实践案例。更多实践分享内容请参见博客。

13.8.1 业务场景

- 案例
- 经验
- 三方评测

13.8.2 内核

- MATCH 中变长 Pattern 的实现
- 如何向 NebulaGraph 增加一个测试用例
- 基于 BDD 理论的 NebulaGraph 集成测试框架重构（上）
- 基于 BDD 理论的 NebulaGraph 集成测试框架重构（下）
- 解析 NebulaGraph 子图设计及实践
- 基于全文搜索引擎的文本搜索
- 实操 | LDBC 数据导入及 nGQL 实践

13.8.3 周边工具

- 基于 NebulaGraph Importer 批量导入工具性能验证方案总结
- 详解 NebulaGraph 3.0 性能报告
- NebulaGraph 支持 JDBC 协议
- Nebula・利器 | Norm 知乎开源的 ORM 工具
- 基于 NebulaGraph 的 Betweenness Centrality 算法
- 无依赖单机尝鲜 NebulaGraph Exchange 的 SST 导入
- logrotate 在 NebulaGraph 的日志滚动实践

视频

- Nebula 高性能图 schema 设计 by 青藤云安全（51 分 30 秒）

- 同花顺图数据库选型：消息面、基本面、技术面（21 分 53 秒）

- Nebula 在 Akulaku 智能风控的实践（40 分 03 秒）

- 从零到一：如何使用 NebulaGraph 构建一个企业股权图谱系统（09 分 34 秒）

- 美团图数据库平台建设及业务实践（上）（14 分 36 秒）

- 美团图数据库平台建设及业务实践（下）（21 分 33 秒）

- 信息图谱在携程酒店的应用（39 分 06 秒）

- OPPO 图平台建设（43 分 09 秒）

- NebulaGraph 在网易游戏业务中的实践（47 分 40 秒）

- BIGO 数据管理与应用实践（53 分 47 秒）

- NebulaGraph 保险反欺诈解决方案 Demo 分享（39 分 54 秒）

- NebulaGraph 人际关系查询之疫情防控场景 Demo 分享（1 时 01 分 55 秒）

- NebulaGraph 在携程金融风控的应用（29 分 06 秒）

最后更新: 2025年8月4日

14. 客户端

14.1 客户端介绍

NebulaGraph 提供多种类型客户端，便于用户连接、管理 NebulaGraph 图数据库。

- NebulaGraph Console：原生 CLI 客户端
- NebulaGraph CPP：C++ 客户端
- NebulaGraph Java：Java 客户端
- NebulaGraph Python：Python 客户端
- NebulaGraph Go：Go 客户端

> Note
>
> 仅以下类支持线程安全（thread-safe）：
>
> - NebulaGraph Java 客户端提供的 NebulaPool 和 SessionPool
> - NebulaGraph Go 客户端提供的 ConnectionPool 和 SessionPool

最后更新: 2025年8月4日

14.2 NebulaGraph Console

NebulaGraph Console 是 NebulaGraph 的原生命令行客户端，用于连接 NebulaGraph 集群并执行查询，同时支持管理参数、导出命令的执行结果、导入测试数据集等功能。

14.2.1 版本对照表

参见Github。

14.2.2 获取 NebulaGraph Console

NebulaGraph Console 的获取方式如下：

- 从 GitHub 发布页下载二进制文件。
- 编译源码获取二进制文件。编译方法参见 Install from source code。

14.2.3 功能说明

连接 **NebulaGraph**

运行二进制文件 `nebula-console` 连接 NebulaGraph 的命令语法如下：

```
<path_of_console> -addr <ip> -port <port> -u <username> -p <password>
```

- `path_of_console` 是 NebulaGraph Console 二进制文件的存储路径。
- 开启 SSL 加密后，需要双向认证时，连接时需要指定 SSL 相关参数。

示例如下：

- 直接连接 NebulaGraph

```
./nebula-console -addr 192.168.8.100 -port 9669 -u root -p nebula
```

- 开启 SSL 加密且需要双向认证

```
./nebula-console -addr 192.168.8.100 -port 9669 -u root -p nebula -enable_ssl -ssl_root_ca_path /home/xxx/cert/root.crt -ssl_cert_path /home/xxx/cert/client.crt -ssl_private_key_path /home/xxx/cert/client.key
```

常用参数的说明如下。

参数	说明
`-h/-help`	显示帮助菜单。
`-addr/-address`	设置要连接的 Graph 服务的 IP 或主机名。默认地址为 127.0.0.1。
`-P/-port`	设置要连接的 Graph 服务的端口。默认端口为 9669。
`-u/-user`	设置 NebulaGraph 账号的用户名。未启用身份认证时，可以使用任意已存在的用户名（默认为 `root`）。
`-p/-password`	设置用户名对应的密码。未启用身份认证时，密码可以填写任意字符。
`-t/-timeout`	设置整数类型的连接超时时间。单位为毫秒，默认值为 120。
`-e/-eval`	设置字符串类型的 nGQL 语句。连接成功后会执行一次该语句并返回结果，然后自动断开连接。
`-f/-file`	设置存储 nGQL 语句的文件的路径。连接成功后会执行该文件内的 nGQL 语句并返回结果，执行完毕后自动断开连接。
`-enable_ssl`	连接 NebulaGraph 时使用 SSL 加密双向认证。
`-ssl_root_ca_path`	指定 CA 根证书的存储路径。
`-ssl_cert_path`	指定 SSL 公钥证书的存储路径。
`-ssl_private_key_path`	指定 SSL 密钥的存储路径。
`-ssl_insecure_skip_verify`	指定客户端是否跳过验证服务端的证书链和主机名。默认为 `false`。如果设置为 `true`，则接受服务端提供的任何证书链和主机名。

更多参数参见项目仓库。

管理参数

NebulaGraph Console 可以保存参数，用于参数化查询。

Note

- VID 不支持参数化查询。
- `SAMPLE` 子句中不支持参数化查询。
- 会话释放后，参数不会保留。

- 保存参数命令如下：

```
nebula> :param <param_name> => <param_value>;
```

示例：

```
nebula> :param p1 => "Tim Duncan";
nebula> MATCH (v:player{name:$p1})-[:follow]->(n)  RETURN v,n;
+----------------------------------------------------+-------------------------------------------------------+
| v                                                  | n                                                     |
+----------------------------------------------------+-------------------------------------------------------+
| ("player100" :player{age: 42, name: "Tim Duncan"}) | ("player125" :player{age: 41, name: "Manu Ginobili"}) |
| ("player100" :player{age: 42, name: "Tim Duncan"}) | ("player101" :player{age: 36, name: "Tony Parker"})   |
+----------------------------------------------------+-------------------------------------------------------+

nebula> :param p2 => {"a":3,"b":false,"c":"Tim Duncan"};
nebula> RETURN $p2.b AS b;
+-------+
| b     |
+-------+
| false |
+-------+
```

- 查看当前保存的所有参数，命令如下：

```
nebula> :params;
```

- 查看指定参数，命令如下：

```
nebula> :params <param_name>;
```

- 删除指定参数，命令如下：

```
nebula> :param <param_name> =>;
```

导出执行结果

导出命令执行的返回结果，可以保存为 CSV 文件、 DOT 文件或者 Profile/Explain 结果。

Note

- 文件保存在当前工作目录中，即 Linux 命令 `pwd` 显示的目录。
- 命令只对下一条查询语句生效。
- DOT 文件的内容可以复制后在 GraphvizOnline 网页中粘贴，生成可视化的执行计划图。

- 导出 CSV 文件命令如下：

```
nebula> :CSV <file_name.csv>;
```

- 导出 DOT 文件命令如下：

```
nebula> :dot <file_name.dot>;
```

示例：

```
nebula> :dot a.dot;
nebula> PROFILE FORMAT="dot" GO FROM "player100" OVER follow;
```

- 导出 PROFILE/EXPLAIN 结果到文件命令如下：

```
nebula> :profile <file_name>;
```

或者

```
nebula> :explain <file_name>;
```

Note

相比于 Studio 中的截图、CSV 文件，因为保有更多信息量和拥有更好的可读性，经由此命令输出的文本文件内容是首推的在 GitHub issue、论坛中报告执行计划、图查询调优的方式。

示例：

```
nebula> :profile profile.log
nebula> PROFILE GO FROM "player102" OVER serve YIELD dst(edge);
nebula> :profile profile.dot
nebula> PROFILE FORMAT="dot" GO FROM "player102" OVER serve YIELD dst(edge);
nebula> :explain explain.log
nebula> EXPLAIN GO FROM "player102" OVER serve YIELD dst(edge);
```

加载测试数据集

测试数据集名称为 basketballplayer，详细 Schema 信息和数据信息请使用相关 `SHOW` 命令查看。

加载测试数据集命令如下：

```
nebula> :play basketballplayer;
```

重复执行语句

重复执行下一个命令 N 次，然后打印平均执行时间。命令如下：

```
nebula> :repeat N;
```

示例：

```
nebula> :repeat 3;
nebula> GO FROM "player100" OVER follow YIELD dst(edge);
+-------------+
| dst(EDGE)   |
+-------------+
| "player101" |
| "player125" |
+-------------+
Got 2 rows (time spent 2602/3214 us)

Fri, 20 Aug 2021 06:36:05 UTC

+-------------+
| dst(EDGE)   |
+-------------+
| "player101" |
| "player125" |
+-------------+
Got 2 rows (time spent 583/849 us)

Fri, 20 Aug 2021 06:36:05 UTC

+-------------+
| dst(EDGE)   |
+-------------+
| "player101" |
| "player125" |
+-------------+
Got 2 rows (time spent 496/671 us)

Fri, 20 Aug 2021 06:36:05 UTC

Executed 3 times, (total time spent 3681/4734 us), (average time spent 1227/1578 us)
```

睡眠

睡眠 N 秒。常用于修改 Schema 的操作中，因为修改 Schema 是异步实现的，需要在下一个心跳周期才同步数据。命令如下：

```
nebula> :sleep N;
```

断开连接

用户可以使用 `:EXIT` 或者 `:QUIT` 从 NebulaGraph 断开连接。为方便使用，NebulaGraph Console 支持使用不带冒号（:）的小写命令，例如 `quit`。

示例：

```
nebula> :QUIT;

Bye root!
```

最后更新: 2025年8月4日

14.3 NebulaGraph CPP

NebulaGraph CPP 是一款 C++ 语言的客户端，可以连接、管理 NebulaGraph 图数据库。

14.3.1 前提条件

请确保已安装 C++ 且 GCC 版本为 4.8 及以上。

14.3.2 版本对照表

参见Github。

14.3.3 安装 NebulaGraph CPP

本文介绍通过编译方式安装 NebulaGraph CPP。

前提条件

- 准备正确的编译环境，详情请参见软硬件要求和安装三方库依赖包。
- 确保已安装 C++ 且 GCC 版本为：{10.1.0 | 9.3.0 | 9.2.0 | 9.1.0 | 8.3.0 | 7.5.0 | 7.1.0}。详情请参见 gcc_preset_versions 参数。

安装步骤

1. 克隆 NebulaGraph CPP 源码到机器。

- （推荐）如果需要安装指定版本的 NebulaGraph CPP，请使用选项 `--branch` 指定分支。例如安装 v3.8.0发布版本，请执行如下命令：

```
$ git clone --branch release-3.8 https://github.com/vesoft-inc/nebula-cpp.git
```

- 如果需要安装日常开发版本，请执行如下命令下载 `master` 分支的源码：

```
$ git clone https://github.com/vesoft-inc/nebula-cpp.git
```

2. 进入目录 nebula-cpp。

```
$ cd nebula-cpp
```

3. 创建目录 build 并进入该目录。

```
$ mkdir build && cd build
```

4. 使用 CMake 生成 makefile 文件。

> Note
>
> 默认安装路径为 `/usr/local/nebula`，如果需要修改路径，请在下方命令内增加参数 `-DCMAKE_INSTALL_PREFIX=<installation_path>`。

```
$ cmake -DCMAKE_BUILD_TYPE=Release ..
```

> Note
>
> 如果 g++ 不支持 C++11，请添加选项 `-DDISABLE_CXX11_ABI=ON`。

5. 编译 NebulaGraph CPP。

为了适当地加快编译速度，可以使用选项 `-j` 并行编译。并行数量 `N` 建议为\(\min(\text{CPU}核数，\frac{内存 (GB)}{2})\)。

```
$ make -j{N}
```

6. 安装 NebulaGraph CPP。

```
$ sudo make install
```

7. 更新动态链接库。

```
$ sudo ldconfig
```

14.3.4 使用方法

将 CPP 文件编译为可执行文件即可。接下来以 `SessionExample.cpp` 为例，介绍如何操作。

1. 使用示例代码创建 `SessionExample.cpp` 文件。
2. 编译文件，命令如下：

```
$ LIBRARY_PATH=<library_folder_path>:$LIBRARY_PATH g++ -std=c++11 SessionExample.cpp -I<include_folder_path> -lnebula_graph_client -o session_example
```

- `library_folder_path`： NebulaGraph 动态库文件存储路径，默认为 `/usr/local/nebula/lib64`。
- `include_folder_path`： NebulaGraph 头文件存储路径，默认为 `/usr/local/nebula/include`。

示例：

```
$ LIBRARY_PATH=/usr/local/nebula/lib64:$LIBRARY_PATH g++ -std=c++11 SessionExample.cpp -I/usr/local/nebula/include -lnebula_graph_client -o session_example
```

14.3.5 API 文档

点击此处查看 CPP 客户端提供的各种类和方法。

14.3.6 核心代码

NebulaGraph CPP 客户端提供 Session Pool 和 Connection Pool 两种方式连接 NebulaGraph 。使用 Connection Pool 需要用户自行管理 Session 实例。

- Session Pool

  详细示例请参见 SessionPoolExample。

- Connection Pool

  详细示例请参见 SessionExample。

最后更新: 2025年8月4日

14.4 NebulaGraph Java

NebulaGraph Java 是一款 Java 语言的客户端，可以连接、管理 NebulaGraph 图数据库。

14.4.1 前提条件

已安装 Java，版本为 8.0 及以上。

14.4.2 版本对照表

参见Github。

14.4.3 下载 NebulaGraph Java

- （推荐）如果需要使用指定版本的 NebulaGraph Java，请使用选项 `--branch` 指定分支。例如使用 v3.8.0发布版本，请执行如下命令：

```
$ git clone --branch release-3.8 https://github.com/vesoft-inc/nebula-java.git
```

- 如果需要安装日常开发版本，请执行如下命令下载 `master` 分支的源码：

```
$ git clone https://github.com/vesoft-inc/nebula-java.git
```

14.4.4 使用方法

> Note
>
> 建议一个线程使用一个会话，如果多个线程使用同一个会话，会降低效率。

使用 IDEA 等工具导入 Maven 项目，请在 `pom.xml` 中添加如下依赖：

> Note
>
> `3.0.0-SNAPSHOT` 为日常研发版本，可能存在未知问题，建议使用 release 版本号替换 `3.0.0-SNAPSHOT` 。

```
<dependency>
 <groupId>com.vesoft</groupId>
 <artifactId>client</artifactId>
 <version>3.0.0-SNAPSHOT</version>
</dependency>
```

如果无法下载日常研发版本的依赖，请在 `pom.xml` 中添加如下内容（release 版本不需要添加）：

```
<repositories>
 <repository>
  <id>snapshots</id>
  <url>https://oss.sonatype.org/content/repositories/snapshots/</url>
 </repository>
</repositories>
```

如果没有 Maven 管理项目，请手动下载 JAR 包进行安装。

14.4.5 API 文档

点击此处查看 Java 客户端提供的各种类和方法。

14.4.6 核心代码

NebulaGraph Java 客户端提供 Connection Pool 和 Session Pool 两种使用方式，使用 Connection Pool 需要用户自行管理 Session 实例。

- Session Pool

  详细示例请参见 GraphSessionPoolExample。

- Connection Pool

  详细示例请参见 GraphClientExample。

14.4.7 Java 相关库

以下由非常酷的社区用户提供和维护，欢迎大家参与测试和贡献。

- java-jdbc
- java-orm
- java-springboot demo
- ngbatis

最后更新: 2025年8月4日

14.5 NebulaGraph Python

NebulaGraph Python 是一款 Python 语言的客户端，可以连接、管理 NebulaGraph 图数据库。

14.5.1 前提条件

已安装 Python，版本为 3.6 及以上。

14.5.2 版本对照表

参见Github。

14.5.3 安装 NebulaGraph Python

pip 安装

```
$ pip install nebula3-python==<version>
```

克隆源码安装

1. 克隆 NebulaGraph Python 源码到机器。

- （推荐）如果需要安装指定版本的 NebulaGraph Python，请使用选项 `--branch` 指定分支。例如安装 v3.8.0发布版本，请执行如下命令：

```
$ git clone --branch release-3.8 https://github.com/vesoft-inc/nebula-python.git
```

- 如果需要安装日常开发版本，请执行如下命令下载 `master` 分支的源码：

```
$ git clone https://github.com/vesoft-inc/nebula-python.git
```

2. 进入目录 nebula-python。

```
$ cd nebula-python
```

3. 执行如下命令安装。

```
$ pip install .
```

14.5.4 API 文档

点击此处查看 Python 客户端提供的各种类和方法。

14.5.5 核心代码

NebulaGraph Python 客户端提供 Connection Pool 和 Session Pool 两种使用方式，使用 Connection Pool 需要用户自行管理 Session 实例。

- Session Pool

  详细示例请参见 SessinPoolExample.py。

  使用限制请参见 Example of using session pool。

- Connection Pool

  详细示例请参见 Example。

最后更新: 2025年8月4日

14.6 NebulaGraph Go

NebulaGraph Go 是一款 Go 语言的客户端，可以连接、管理 NebulaGraph 图数据库。

14.6.1 前提条件

已安装 Go，版本为 1.13 及以上。

14.6.2 版本对照表

参见Github。

14.6.3 下载 NebulaGraph Go

- （推荐）如果需要下载指定版本的 NebulaGraph Go，请使用选项 `--branch` 指定分支。例如安装 v3.8.0发布版本，请执行如下命令：

```
$ git clone --branch release-3.8 https://github.com/vesoft-inc/nebula-go.git
```

- 如果需要安装日常开发版本，请执行如下命令下载 `master` 分支的源码：

```
$ git clone https://github.com/vesoft-inc/nebula-go.git
```

14.6.4 安装或更新

安装或更新的命令如下：

```
$ go get -u -v github.com/vesoft-inc/nebula-go@<tag>
```

`tag`：指定分支。例如 `master` 或 `release-3.8`。

14.6.5 API 文档

点击此处查看 Go 客户端提供的各种方法和类型。

14.6.6 核心代码

NebulaGraph Go 客户端提供 Connection Pool 和 Session Pool 两种使用方式，使用 Connection Pool 需要用户自行管理 Session 实例。

- Session Pool

  详细示例请参见 session_pool_example.go。

  使用限制请参见 Usage example。

- Connection Pool

  详细示例请参见 graph_client_basic_example 和 graph_client_goroutines_example。

最后更新: 2025年8月4日

14.7 社区贡献的客户端

你可以使用社区用户开发的以下客户端连接和管理 NebulaGraph 图数据库：

- NebulaGraph Rust
- NebulaGraph PHP
- NebulaGraph Node
- NebulaGraph .NET

最后更新: 2025年8月4日

15. NebulaGraph Studio

15.1 认识 NebulaGraph Studio

15.1.1 什么是 NebulaGraph Studio

NebulaGraph Studio（简称 Studio）是一款可以通过 Web 访问的开源图数据库可视化工具，搭配 NebulaGraph 内核使用，提供构图、数据导入、编写 nGQL 查询等一站式服务。用户可以在 NebulaGraph GitHub 仓库中查看最新源码，详情参见 nebula-studio。

> Note
>
> 用户可以在线试用 Studio 部分功能。企业版 NebulaGraph 提供功能更强大的可视化工具，点击免费试用即可在阿里云上体验。

发行版本

用户可以使用 RPM 包、DEB 包、tar 包和 Docker 服务安装部署 Studio，在 Kubernetes 集群里还支持使用 Helm 安装部署 Studio。详细信息参考 部署 Studio。

几种部署方式功能基本相同，在使用 Studio 时可能会受到限制。详细信息，参考使用限制。

产品功能

Studio 可以方便管理 NebulaGraph 数据，具备以下功能：

- 使用 Schema 管理功能，用户可以使用图形界面完成图空间、Tag（标签）、Edge Type（边类型）、索引的创建，查看图空间的统计数据，快速上手 NebulaGraph 。
- 使用导入功能，通过简单的配置，用户即能批量导入点和边数据，并能实时查看数据导入日志。
- 使用控制台功能，用户可以使用 nGQL 语句创建 Schema，并对数据执行增删改查操作。

适用场景

如果有以下任一需求，都可以使用 Studio：

- 已经安装部署了 NebulaGraph，想使用 GUI 工具创建 Schema、导入数据、执行 nGQL 语句查询。
- 刚开始学习 nGQL（NebulaGraph Query Language），但是不习惯用命令行工具，更希望使用 GUI 工具查看语句输出的结果。

身份验证

因为 NebulaGraph 默认不启用身份验证，所以用户可以使用 `root` 账号和任意密码登录 Studio。

当 NebulaGraph 启用了身份验证后，用户只能使用指定的账号和密码登录 Studio。关于 NebulaGraph 的身份验证功能，参考 身份验证。

版本兼容性

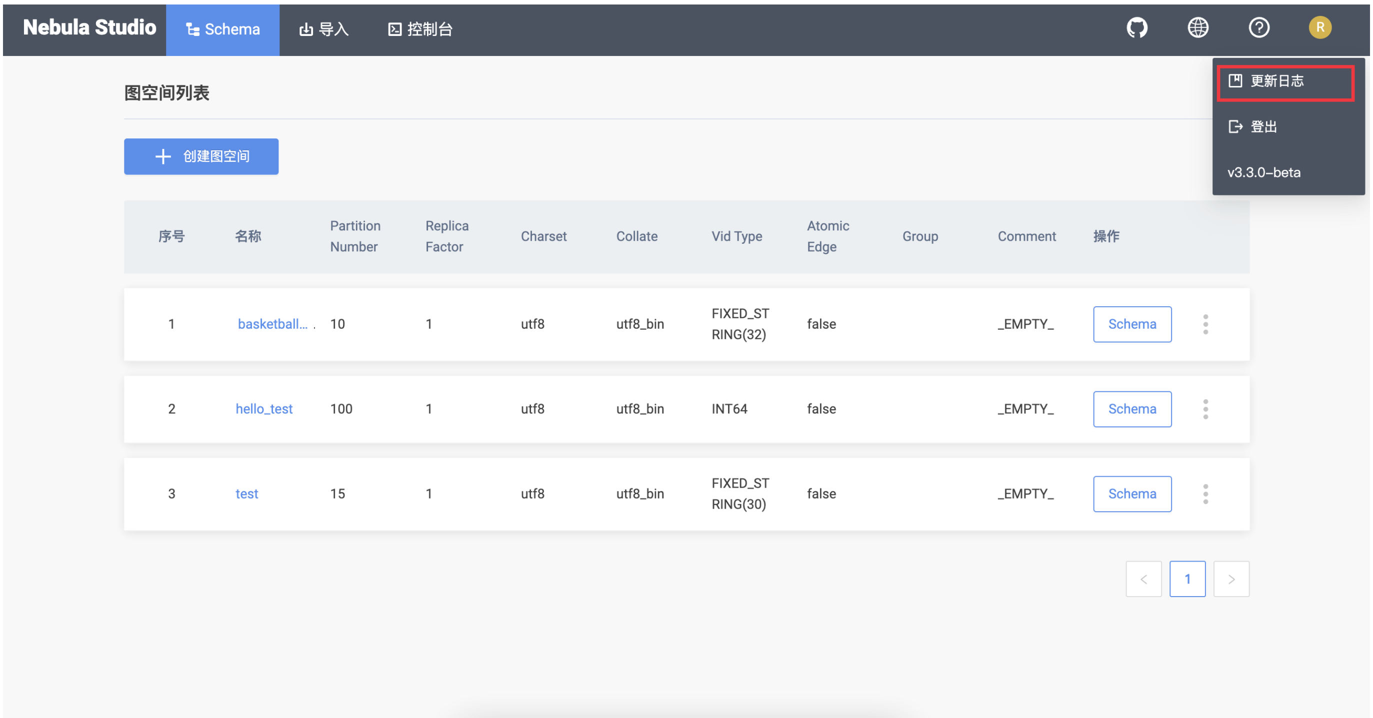
> Note
>
> Studio 版本发布节奏独立于 NebulaGraph 内核，其命名方式也不参照内核命名规则，两者兼容对应关系如下表。

NebulaGraph 版本	Studio 版本
3.8.0	3.10.0、3.9.x
3.6.0	3.8.0、3.7.0
3.5.x	3.7.0
3.4.0 ~ 3.4.1	3.7.0、3.6.0、3.5.1、3.5.0
3.3.0	3.5.1、3.5.0
3.0.0 ～ 3.2.x	3.4.1、3.4.0
3.1.0	3.3.2
3.0.0	3.2.x
2.6.x	3.1.x
2.6.x	3.1.x
2.0 & 2.0.1	2.x
1.x	1.x

版本更新

Studio 处于持续开发状态中。用户可以通过 Studio 版本更新说明查看最新发布的功能。

成功连接 Studio 后，用户可以在页面右上角点击用户头像，再点击 更新日志，查看 Studio 的版本更新说明。



最后更新: 2025年8月4日

15.1.2 使用限制

本文描述使用 Studio 的限制。

系统架构

Studio 目前仅支持 x86_64 架构。

数据上传

Studio 上传数据仅支持上传无表头的 CSV 文件，但是，单个文件大小及保存时间不受限制，而且数据总量以本地存储容量为准。

数据备份

目前仅支持在 控制台 上以 CSV 格式导出查询结果，不支持其他数据备份方式。

nGQL 支持

除以下内容外，用户可以在 控制台 上执行所有 nGQL 语句：

- `USE <space_name>`：只能在 Space 下拉列表中选择图空间，不能运行这个语句选择图空间。
- 控制台 上使用 nGQL 语句时，用户可以直接回车换行，不能使用换行符。

浏览器支持

建议使用最新版本的 Chrome 访问 Studio。否则会出现样式显示异常和交互异常等问题。

最后更新: 2025年8月4日

15.2 安装与登录

15.2.1 部署 Studio

本文介绍如何在本地通过 RPM、DEB、tar 包和 Docker 部署 Studio。

RPM 部署 Studio

前提条件

在部署 RPM 版 Studio 之前，用户需要确认以下信息：

- NebulaGraph 服务已经部署并启动。详细信息，参考 NebulaGraph 安装部署。
- 使用的 Linux 发行版为 CentOS ，已安装 lsof。
- 确保以下端口未被占用。

端口号	说明
7001	Studio 提供 web 服务使用。

安装

1. 根据需要选择并下载 RPM 包，建议选择最新版本。常用下载链接如下：

安装包	检验和	适用 NebulaGraph 版本
nebula-graph-studio-3.10.0.x86_64.rpm	nebula-graph-studio-3.10.0.x86_64.rpm.sha256	master

2. 使用 `sudo rpm -i <rpm_name>` 命令安装 RPM 包。

   例如，安装 Studio 3.10.0 版本需要运行以下命令，默认安装路径为 `/usr/local/nebula-graph-studio`：

```
$ sudo rpm -i nebula-graph-studio-3.10.0.x86_64.rpm
```

   也可以使用以下命令安装到指定路径：

```
$ sudo rpm -i nebula-graph-studio-3.10.0.x86_64.rpm --prefix=<path>
```

   当屏幕返回以下信息时，表示 PRM 版 Studio 已经成功启动。

```
Start installing NebulaGraph Studio now...
NebulaGraph Studio has been installed.
NebulaGraph Studio started automatically.
```

3. 启动成功后，在浏览器地址栏输入 `http://<ip address>:7001` 。

   如果在浏览器窗口中能看到以下登录界面，表示已经成功部署并启动 Studio。

卸载

用户可以使用以下的命令卸载 Studio。

```
$ sudo rpm -e nebula-graph-studio-3.10.0.x86_64
```

当屏幕返回以下信息时，表示 PRM 版 Studio 已经卸载。

```
NebulaGraph Studio removed, bye~
```

异常处理

如果在安装过程中自动启动失败或是需要手动启动或停止服务，请使用以下命令：

- 手动启动服务

```
$ bash /usr/local/nebula-graph-studio/scripts/rpm/start.sh
```

- 手动停止服务

```
$ bash /usr/local/nebula-graph-studio/scripts/rpm/stop.sh
```

如果启动服务时遇到报错 `ERROR: bind EADDRINUSE 0.0.0.0:7001`，用户可以通过以下命令查看端口 7001 是否被占用。

```
$ lsof -i:7001
```

如果端口被占用，且无法结束该端口上进程，用户可以修改 studio 配置内的启动端口，并重新启动服务。

```
//修改 studio 服务配置。配置文件默认路径为`/usr/local/nebula-graph-studio`。
$ vi etc/studio-api.yaml

//修改端口号，改成任意一个当前可用的即可。
Port: 7001

//重启服务
$ systemctl restart nebula-graph-studio.service
```

DEB 部署 Studio

前提条件

在通过 DEB 部署安装 Studio 之前，用户需要确认以下信息：

- NebulaGraph 服务已经部署并启动。详细信息，参考 NebulaGraph 安装部署。
- 使用的 Linux 发行版为 Ubuntu。
- 确保以下端口未被占用。

端口号	说明
7001	Studio 提供的 web 服务

- 确保系统中存在 `/usr/lib/systemd/system` 目录。如没有该目录，需手动创建。

安装

1. 根据需要选择并下载 DEB 包，建议选择最新版本。常用下载链接如下：

安装包	检验和	适用 NebulaGraph 版本
nebula-graph-studio-3.10.0.x86_64.deb	nebula-graph-studio-3.10.0.x86_64.deb.sha256	master

2. 使用 `sudo dpkg -i <deb_name>` 命令安装 DEB 包。

例如，安装 Studio 3.10.0 版本需要运行以下命令：

```
$ sudo dpkg -i nebula-graph-studio-3.10.0.x86_64.deb
```

3. 启动成功后，在浏览器地址栏输入 `http://<ip address>:7001` 。

如果在浏览器窗口中能看到以下登录界面，表示已经成功部署并启动 Studio。

卸载

用户可以使用以下的命令卸载 Studio。

```
$ sudo dpkg -r nebula-graph-studio
```

tar 包部署 Studio

前提条件

在部署 tar 包安装的 Studio 之前，用户需要确认以下信息：

- NebulaGraph 服务已经部署并启动。详细信息，参考 NebulaGraph 安装部署。

- 确保以下端口未被占用。

端口号	说明
7001	Studio 提供的 web 服务

安装部署

1. 根据需要下载 tar 包，建议选择最新版本。

安装包	Studio 版本	适用 NebulaGraph 版本
nebula-graph-studio-3.10.0.x86_64.tar.gz	3.10.0	master

2. 解压 tar 包。

```
tar -xvf nebula-graph-studio-3.10.0.x86_64.tar.gz
```

3. 部署 nebula-graph-studio 并启动。

```
$ cd nebula-graph-studio
$ ./server
```

4. 启动成功后，在浏览器地址栏输入 `http://<ip address>:7001`。

如果在浏览器窗口中能看到以下登录界面，表示已经成功部署并启动 Studio。

停止服务

用户可以采用 `kill <pid>` 的方式来关停服务：

```
$ kill $(lsof -t -i :7001) # stop nebula-graph-studio
```

Docker 部署 Studio

前提条件

在部署 Docker 版 Studio 之前，用户需要确认以下信息：

- NebulaGraph 服务已经部署并启动。详细信息，参考 NebulaGraph 安装部署。
- 在即将运行 Docker 版 Studio 的机器上安装并启动 Docker Compose。详细信息参考 Docker Compose 文档。
- 确保以下端口未被占用。

端口号	说明
7001	Studio 提供的 web 服务

- （可选）在中国大陆从 Docker Hub 拉取 Docker 镜像的速度可能比较慢，用户可以使用 `registry-mirrors` 参数配置加速镜像。例如，如果要使用 Docker 中国区官方镜像、网易镜像和中国科技大学的镜像，则按以下格式配置 `registry-mirrors` 参数：

```
{
"registry-mirrors": [
 "https://registry.docker-cn.com",
 "http://hub-mirror.c.163.com",
 "https://docker.mirrors.ustc.edu.cn"
 ]
}
```

配置文件的路径和方法因操作系统和/或 Docker Desktop 版本而异。详细信息参考 Docker Daemon 配置文档。

操作步骤

在命令行工具中按以下步骤依次运行命令，部署并启动 Docker 版 Studio，这里我们用 NebulaGraph 版本为 master 的进行演示：

1. 下载 Studio 的部署配置文件。

安装包	适用 NebulaGraph 版本
nebula-graph-studio-3.10.0.tar.gz	master

2. 创建 `nebula-graph-studio-3.10.0` 目录，并将安装包解压至目录中。

```
mkdir nebula-graph-studio-3.10.0 && tar -zxvf nebula-graph-studio-3.10.0.tar.gz -C nebula-graph-studio-3.10.0
```

3. 解压后进入 `nebula-graph-studio-3.10.0` 目录。

```
cd nebula-graph-studio-3.10.0
```

4. 拉取 Studio 的 Docker 镜像。

```
docker-compose pull
```

5. 构建并启动 Studio 服务。其中，`-d` 表示在后台运行服务容器。

```
docker-compose up -d
```

当屏幕返回以下信息时，表示 Docker 版 Studio 已经成功启动。

```
Creating docker_web_1   ... done
```

6. 启动成功后，在浏览器地址栏输入 `http://<ip address>:7001`。

> Note
>
> 在运行 Docker 版 Studio 的机器上，用户可以运行 `ifconfig` 或者 `ipconfig` 获取本机 IP 地址。如果使用这台机器访问 Studio，可以在浏览器地址栏里输入 `http://localhost:7001`。

如果在浏览器窗口中能看到以下登录界面，表示已经成功部署并启动 Studio。

Helm 部署 Studio

本小节介绍如何在 Kubernetes 中使用 Helm 部署并启动 Studio。

前提条件

安装 Studio 前，用户需要安装以下软件并确保安装版本的正确性：

软件	版本要求
Kubernetes	>= 1.14
Helm	>= 3.2.0

操作步骤

1. 克隆 Studio 的源代码到主机。

```
$ git clone https://github.com/vesoft-inc/nebula-studio.git
```

2. 进入 `nebula-studio` 目录。

```
$ cd nebula-studio
```

3. 更新并安装 Helm Chart，命名为 `my-studio` 。

```
$ helm upgrade --install my-studio --set service.type=NodePort --set service.port=30070 deployment/helm
```

Helm Chart 配置参数说明如下。

参数	默认值	描述
replicaCount	0	Deployment 的副本数。
image.nebulaStudio.name	vesoft/nebula-graph-studio	nebula-graph-studio 镜像的仓库地址。
image.nebulaStudio.version	v3.10.0	nebula-graph-studio 的版本。
service.type	ClusterIP	服务类型，必须为 `NodePort` ， `ClusterIP` 或 `LoadBalancer` 其中之一。
service.port	7001	nebula-graph-studio 中 web 服务的端口。
service.nodePort	32701	Kubernetes 集群外部访问 nebula-studio 的代理端口。
resources.nebulaStudio	{}	nebula-studio 的资源限制/请求。
persistent.storageClassName	""	storageClass 名称，如果不指定就使用默认值。
persistent.size	5Gi	存储盘大小。

4. 启动成功后，在浏览器地址栏输入 `http://<node_address>:30070` 。 如果在浏览器窗口中能看到以下登录界面，表示已经成功部署并启动 Studio。

卸载

```
$ helm uninstall my-studio
```

后续操作

进入 Studio 登录界面后，用户需要连接 NebulaGraph 。详细信息，参考连接数据库。

最后更新: 2025年8月4日

15.2.2 连接数据库

在成功启动 Studio 后，用户需要配置连接 NebulaGraph 。本文主要描述 Studio 如何连接 NebulaGraph 。

前提条件

在连接 NebulaGraph 数据库前，用户需要确认以下信息：

- Studio 已经启动。详细信息参考部署 Studio。
- NebulaGraph 的 Graph 服务本机 IP 地址以及服务所用端口。默认端口为 `9669` 。
- NebulaGraph 登录账号信息，包括用户名和密码。

操作步骤

按以下步骤连接 NebulaGraph：

1. 在浏览器地址栏输入 `http://<ip_address>:7001`。

   在浏览器窗口中看到以下登录界面表示已经成功部署并启动了 Studio。

2. 在 Studio 的 配置数据库 页面上，输入以下信息：

- Graphd IP 地址：填写 NebulaGraph 的 Graph 服务本机 IP 地址。例如 `192.168.10.100`。

> Note
>
> - 即使 NebulaGraph 与 Studio 部署在同一台机器上，用户也必须填写这台机器的本机 IP 地址，而不是 `127.0.0.1` 或者 `localhost`。
> - 在新的标签页连接另一个 NebulaGraph 时，会覆盖旧标签页的会话。如果需要同时登录多个 NebulaGraph，可以用不同的浏览器或者无痕模式。

- Port：Graphd 服务的端口。默认为 `9669`。
- 用户名 和 密码：根据 NebulaGraph 的身份验证设置填写登录账号和密码。
- 如果未启用身份验证，可以填写默认用户名 `root` 和任意密码。
- 如果已启用身份验证，但是未创建账号信息，用户只能以 GOD 角色登录，必须填写 `root` 及对应的密码 `nebula`。
- 如果已启用身份验证，同时又创建了不同的用户并分配了角色，不同角色的用户使用自己的账号和密码登录。

3. 完成设置后，点击 连接 按钮。

> Note
>
> 一次连接会话持续 30 分钟。如果超过 30 分钟没有操作，会话即断开，用户需要重新登录数据库。

首次登录会显示欢迎页，根据使用流程展示相关功能，并且支持自动下载并导入测试数据集。

想要再次访问欢迎页，单击 ?。

后续操作

成功连接 NebulaGraph 后，用户可以执行以下操作：

- 使用控制台或者 Schema 页面管理 Schema。
- 批量导入数据。
- 在 控制台 页面上执行 nGQL 语句查询数据。
- 在 Schema 草图页面图形化设计 Schema。

> **Note**
>
> 账号的权限决定了能执行哪些操作。详情参见内置角色权限。

登出

如果需要重新连接 NebulaGraph，可以登出后重新配置数据库。

在页面右上角单击用户头像，单击 登出。

最后更新: 2025年8月4日

15.3 快速开始

15.3.1 规划 Schema

在使用 Studio 之前，用户需要先根据 NebulaGraph 的要求规划 Schema（模式）。

Schema 至少要包含以下要素：

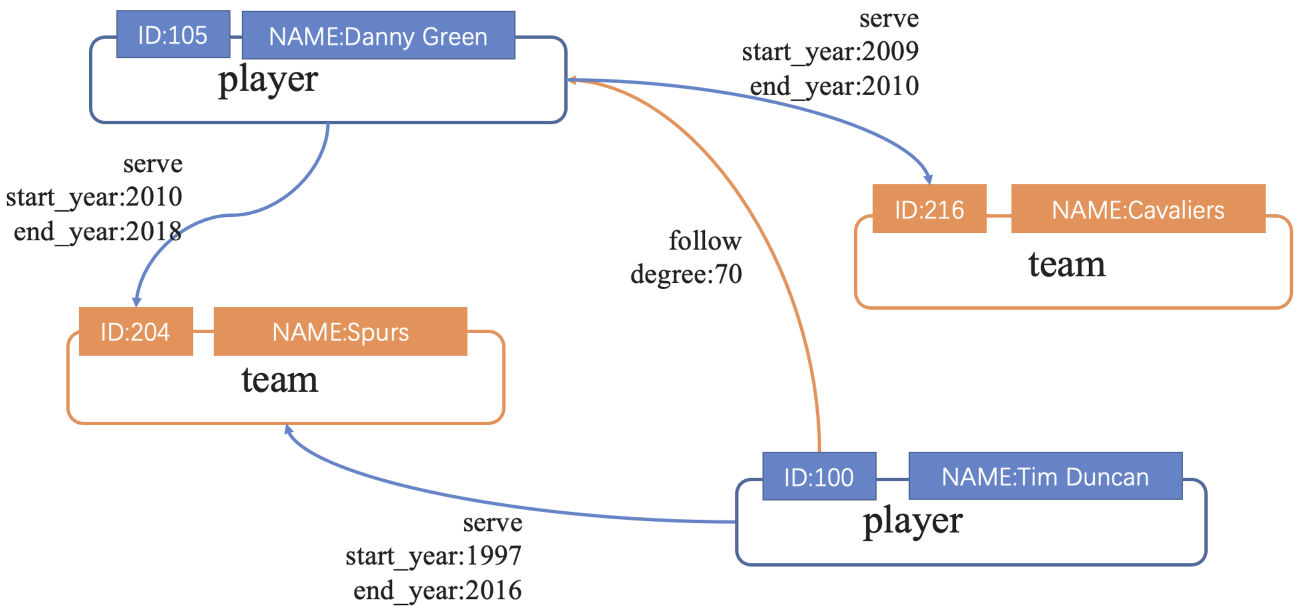
- Tag，以及每种 Tag 的属性。
- Edge type，以及每种 Edge type 的属性。

用户可以下载 NebulaGraph 示例数据集 basketballplayer，本文将通过该示例说明如何规划 Schema。

下表列出了 Schema 要素。

类型	名称	属性名（数据类型）	说明
Tag	player	- `name` （`string`） - `age` （`int`）	表示球员。
Tag	team	- `name` （`string`）	表示球队。
Edge type	serve	- `start_year` （`int`） - `end_year` （`int`）	表示球员的行为。 该行为将球员和球队联系起来，方向是从球员到球队。
Edge type	follow	- `degree` （`int`）	表示球员的行为。 该行为将两个球员联系起来，方向是从一个球员到另一个球员。

下图说明示例中 player 类点与 team 类点之间如何发生关系（serve/follow）。



最后更新: 2025年8月4日

15.3.2 创建 Schema

在 NebulaGraph 中，用户必须先有 Schema，才能向其中写入点数据和边数据。本文描述如何使用 NebulaGraph 的控制台或 Schema 功能创建 Schema。

> Note
>
> - 用户可以使用 nebula-console 创建 Schema。详情参见 NebulaGraph 使用手册和 NebulaGraph 快速开始。
> - 用户可以使用 Schema 草图功能图形化设计 Schema。详情参见 Schema 草图。

前提条件

在 Studio 上创建 Schema 之前，用户需要确认以下信息：

- Studio 已经连接到 NebulaGraph 数据库。
- 账号拥有 GOD、ADMIN 或 DBA 权限。详细信息，参考 NebulaGraph 内置角色。
- 已经规划 Schema 的要素。
- 已经创建图空间。

> Note
>
> 本示例已经创建图空间。如果账号拥有 GOD 权限，也可以在 控制台 或 Schema 上创建一个图空间。

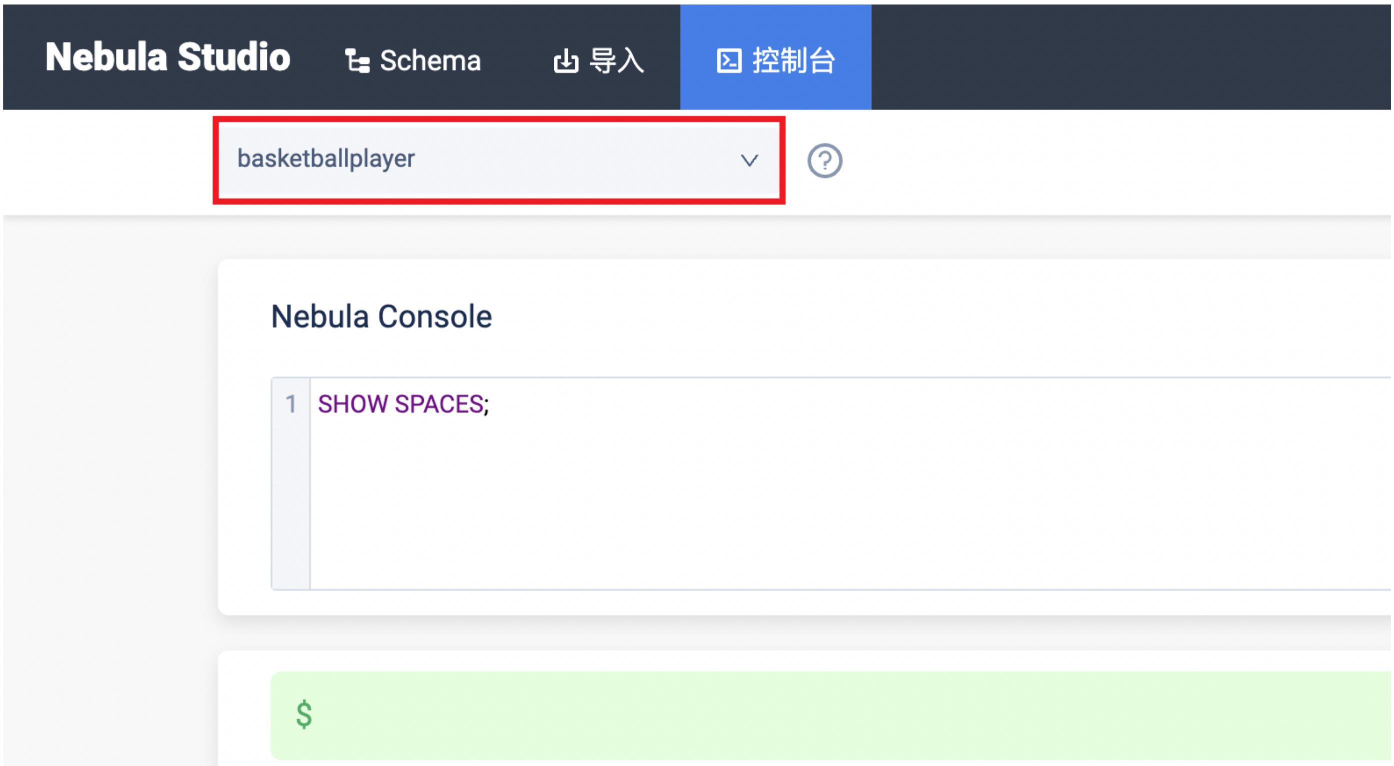
使用 **Schema** 管理功能创建 **Schema**

按以下步骤使用 Schema 管理功能创建 Schema：

1. 创建 Tag。详细信息，参考操作 Tag。
2. 创建 Edge type。详细信息，参考操作 Edge type。

使用控制台创建 **Schema**

1. 在顶部导航栏里，点击 控制台 页签。
2. 在 当前 Space 中选择一个图空间。在本示例中，选择 basketballplayer。



3. 在命令行中，依次输入以下语句，并点击右侧的运行按钮。

```
// 创建 Tag player，带有 2 个属性
CREATE TAG player(name string, age int);

// 创建 Tag team，带有 1 个属性
CREATE TAG team(name string);

// 创建 Edge type follow，带有 1 个属性
CREATE EDGE follow(degree int);

// 创建 Edge type serve，带有 2 个属性
CREATE EDGE serve(start_year int, end_year int);
```

至此，用户已经完成了 Schema 创建。用户可以运行以下语句查看 Tag 与 Edge type 的定义是否正确、完整。

```
// 列出当前图空间中所有 Tag
SHOW TAGS;

// 列出当前图空间中所有 Edge type
SHOW EDGES;

// 查看每种 Tag 和 Edge type 的结构是否正确
DESCRIBE TAG player;
DESCRIBE TAG team;
DESCRIBE EDGE follow;
DESCRIBE EDGE serve;
```

后续操作

创建 Schema 后，用户可以开始导入数据。

最后更新: 2025年8月4日

15.3.3 导入数据

Studio 支持界面化地将 CSV 格式数据导入至 NebulaGraph 中。

前提条件

导入数据之前，需要确认以下信息：

- NebulaGraph 里已经创建 Schema。
- CSV 文件符合 Schema 要求。
- 账号拥有 GOD、ADMIN 或 DBA 权限。详情参见 NebulaGraph 内置角色。

入口

在顶部导航栏里，单击导入。

操作步骤

导入数据主要分为 2 个部分，新建数据源和创建导入任务，接下来将详细介绍。

> Note
>
> 导入任务还支持 AI 导入，该功能为 Beta 功能，使用前需要在系统设置里设置和启用该功能。

新建数据源

在页面右上角单击新建数据源，设置数据来源及其相关设置。当前支持 3 种类型的数据源。

数据源类型	说明
云存储	添加云存储作为 CSV 文件来源，只支持兼容 Amazon S3接口的云服务。
SFTP	添加 SFTP 作为 CSV 文件来源。
本地文件	上传本地 CSV 文件。文件大小不能超过 200 MB，超过限制的文件请放入其他方式的数据源中。

> Note
>
> - 上传本地 CSV 文件时，一次可以选择多个 CSV 文件。
> - 数据源添加后，可以在页面上方单击数据源管理，切换页签即可查看不同类型的数据源详情，也可以编辑或删除数据源。

创建导入任务

1. 在页面左上角单击创建导入任务，完成如下设置：

> **Caution**
>
> 用户也可以单击导入模版，下载示例配置文件 `example.yaml`，配置后再上传配置文件。配置方式与 NebulaGraph Importer大致相同。

- 图空间：需要导入数据的图空间名称。
- 任务名称：默认自动生成，可以修改。
- 更多配置（可选）：可以自定义设置并发数、批处理量、重试次数、读取并发数和导入并发数。
- 关联标签：

a. 单击添加 Tag，然后在下方新增的标签内选择 Tag。

b. 单击添加导入文件，在文件源里选择数据源类型和文件路径，找到需要导入的文件，然后单击添加。

c. 在预览页面设置文件的分隔符和是否携带表头，然后单击确认。

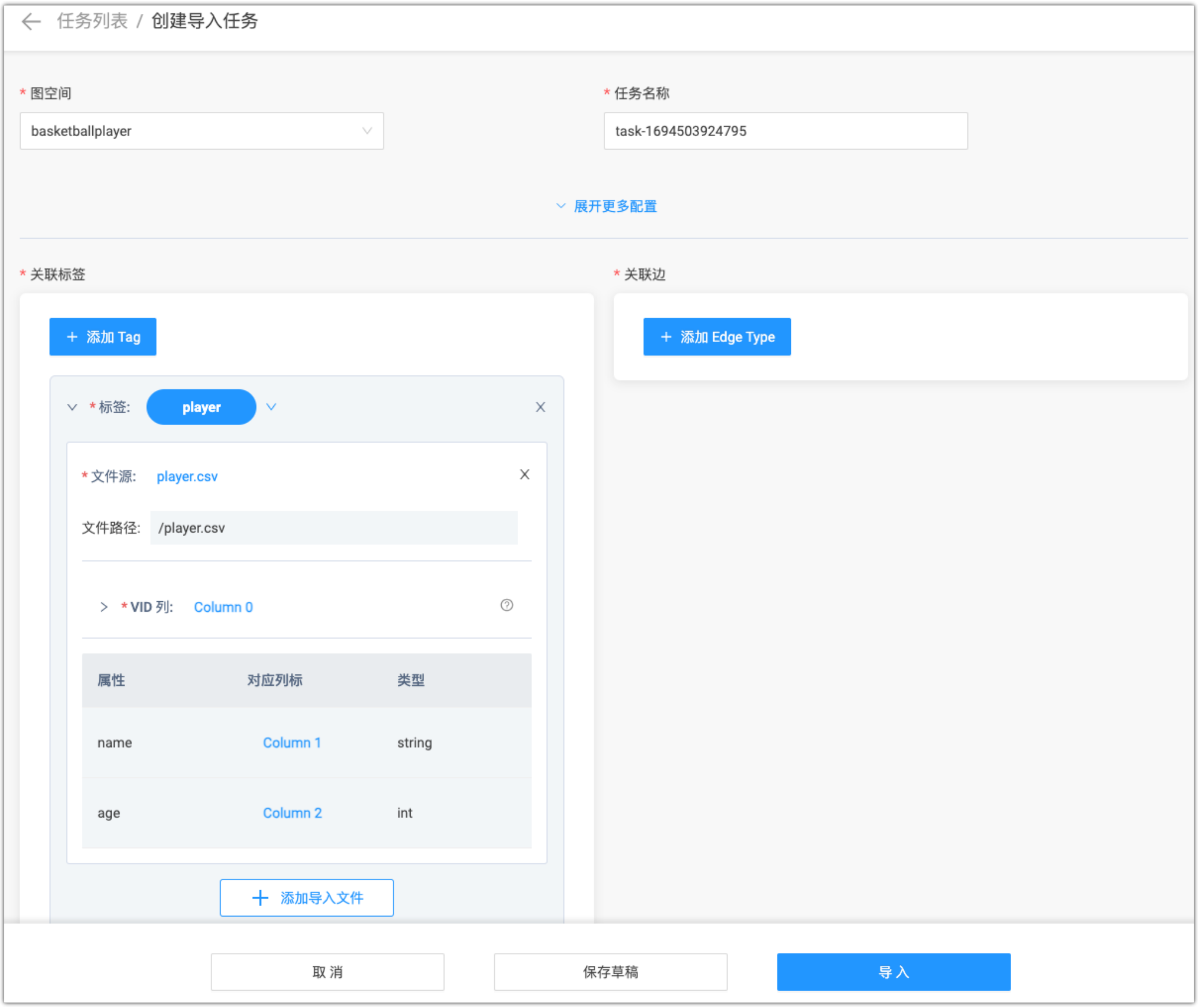
d. 在VID 列为 VID 选择对应的列。支持选择多个列合并为 VID，也可以为 VID 添加前缀或后缀。

e. 在属性框内为属性选择对应的列。对于可以为 `NULL` 或设置了 `DEFAULT` 的属性，可以不指定对应的列。

f. 重复 2 ~ 5 步骤将步骤 1 所选 Tag 的数据文件全部导入。

g. 重复 1 ~ 6 步骤将所有需要导入的 Tag 数据全部导入。

- 关联边：与关联标签的操作相同。



2. 完成设置后，单击导入,输入 NebulaGraph 账号的密码并确认。

   导入任务创建后，可以在导入数据页签内查看导入任务的进度，支持根据图空间筛选任务、编辑任务、查看日志、下载日志、重新导入、下载配置文件、删除任务等操作。

AI 导入

> **Note**
>
> 导入任务执行完成后，请手动检查数据是否正常导入，推荐结合导入页面的任务日志进行查看，排查超时、触发隐私政策、服务中断、编码等问题。

1. 在页面左上角单击AI 导入，完成如下设置：

- 文件：选择需要进行 AI 导入的文件。当前仅支持本地单个文件或本地目录。每次只能选择一个本地文件或目录。文件需要为纯文本文件。
- 导入图空间：待导入数据的图空间。图空间的 Schema 需要提前创建好。
- 导出 NGQL 文件路径：导出 ngql 文件的路径，通常无需修改。
- 附加提示：可以附加其他提示来指导数据导入。提示的精度直接影响导入结果的准确性。

  你可以在配置文件中查看和 AI 导入相关的 `LLM` 参数。

2. 完成设置后，单击下一步，确认待导入的文件和使用的 AI URL，然后单击开始。

   导入任务创建后，可以在导入数据页签内查看导入任务的进度，支持查看日志、下载日志、重新导入、删除任务等操作。

后续操作

完成数据导入后，用户可以进入控制台页面。

最后更新: 2025年8月4日

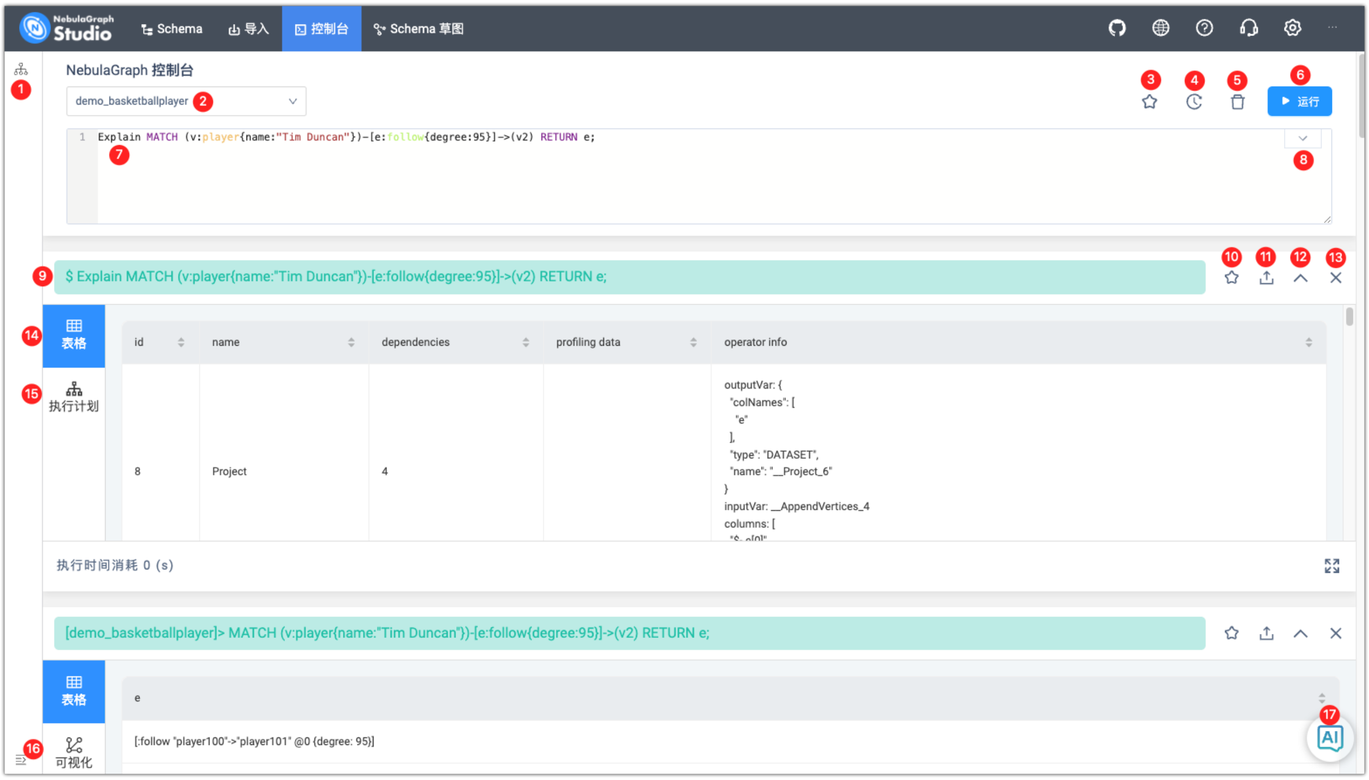
15.3.4 控制台界面

本文介绍 Studio 的控制台界面。

入口

在顶部导航栏里，单击控制台。

页面介绍

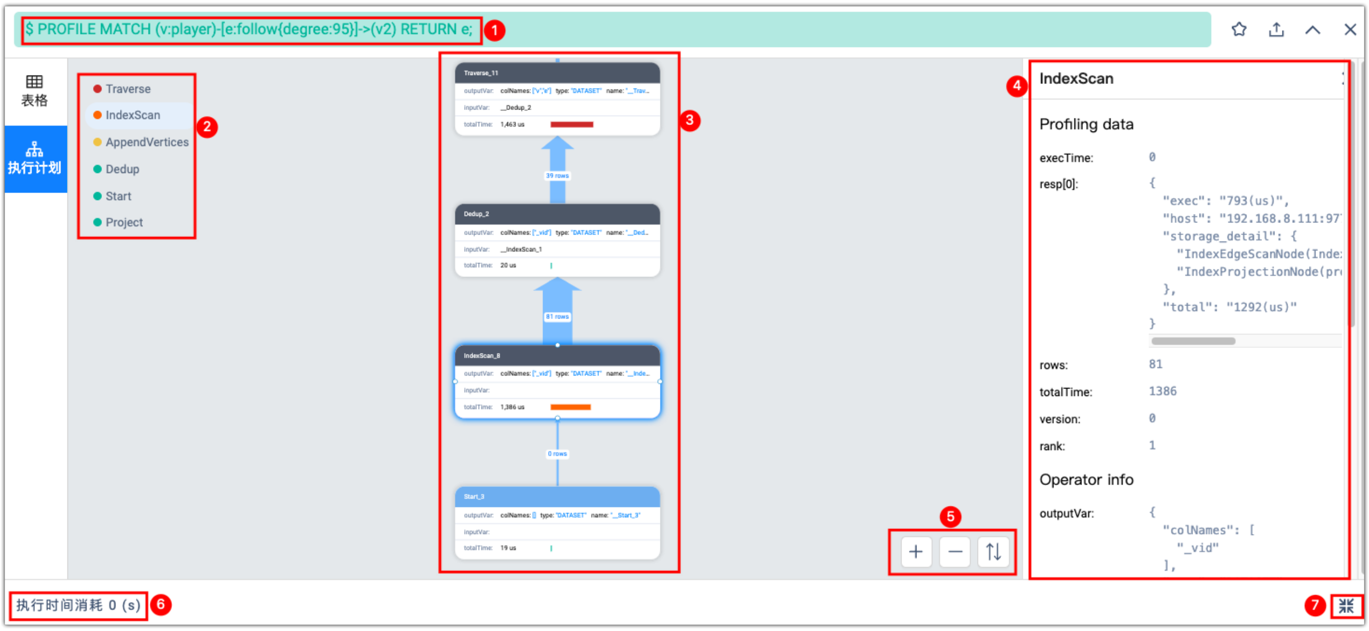


下表列出了控制台界面上的各种功能。

编号	功能	说明
1	查看 Schema	展示图空间的 Schema。
2	选择图空间	选择执行 nGQL 的图空间。不支持在控制台执行 `USE <space_name>` 语句切换图空间。
3	收藏夹	点击 按钮，展开收藏夹，点击其中一个语句，该语句会自动输入到输入框中。
4	历史清单	点击 按钮，在语句运行记录列表里，点击其中一个语句，输入框中即自动输入该语句。列表里提供最近 15 次语句运行记录。在输入框中输入 `/` 可以快速选择历史查询语句。
5	清空输入框	点击 按钮，清空输入框中已经输入的内容。
6	运行	在输入框中输入 nGQL 语句后，点击 按钮即开始运行语句。
7	输入框	输入 nGQL 语句的区域。语句会根据 Schema 或字符串展示不同颜色。支持代码补全，可以根据 Schema 快速输入 Tag 或者 Edge type。可以同时输入多个语句按顺序执行，语句之间以 `;` 分隔。可以用 `//` 添加注释。支持对选中的语句单击右键，然后执行剪切、复制、运行等操作。
8	自定义参数展示	点击 按钮可展开查看自定义参数，用于参数化查询。详情信息可见管理参数。
9	语句运行状态	运行 nGQL 语句后，这里显示语句运行状态。如果语句运行成功，语句以绿色显示。如果语句运行失败，语句以红色显示。
10	添加到收藏夹	点击 按钮，将语句存入收藏夹中，已收藏的语句该按钮以黄色展示。
11	导出 CSV 文件或 PNG 格式图片	运行 nGQL 语句返回结果后，返回结果为表格形式时，点击 按钮即能将结果以 CSV 文件的形式导出。切换到可视化窗口，点击 按钮即能将结果以 CSV 文件或 PNG 图片的形式导出。
12	展开/隐藏执行结果	点击 按钮，隐藏此条 nGQL 语句返回的结果。
13	关闭执行结果	点击 按钮，关闭此条 nGQL 语句返回的结果。
14	表格窗口	在表格中显示语句的运行结果。
15	执行计划窗口	显示执行计划。如果执行的是 `EXPLAIN` 或 `PROFILE` 语句，窗口会以可视化形式呈现执行计划。如何理解执行计划请参见下文说明。
16	可视化窗口	显示语句运行结果。如果结果包含完整的点边信息，窗口会以可视化形式呈现返回的结果。点击右方 按钮，展开数据概览面板。
17	AI 查询语句助手	用户可以和助手聊天，将自然语言转换为 nGQL 查询语句，并支持一键复制 nGQL 语句到输入框中。使用前需要在系统设置里设置和启用该功能。注意：和助手交互时，会将当前图空间的 Schema 信息发送给大规模语言模型，请注意信息安全。开关text2match用于切换普通问答和查询问答。查询问答可以输入自然语言转换为 nGQL 查询语句。

执行计划说明

Studio 可以展示语句的执行计划，界面说明如下。



序号	说明
1	`EXPLAIN` 或 `PROFILE` 语句。
2	执行计划涉及的算子并根据执行耗时排序。耗时前三的算子分别标记为为红色、橘红色和黄色。单击算子可以直接选中执行流程中的对应算子并展示算子信息。注意： `PROFILE` 语句会实际执行语句，所以能得到实际的耗时并排序。 `EXPLAIN` 不会执行语句，所有算子耗时视为相同并标记为红色。
3	算子执行流程。每个算子都会显示该算子的入参、出参和耗时。 `Select` 、 `Loop` 、 `PassThrough` 和 `Start` 算子有独立配色。箭头展示数据流向和行数，行数越多，箭头越粗。单击算子会在右侧展示算子的详细信息。
4	算子详细信息。分为 `Profiling data` 和 `Operator info` 两部分。 `Profiling data` 展示算子的性能数据，包括接收到的数据行数、执行耗时、总耗时等。 `Operator info` 展示算子的具体操作信息。
5	放大或缩小界面，还可以倒序显式算子执行流程。
6	语句耗时。
7	全屏展示或取消全屏。

关于算子和优化规则的介绍以及如何调优，可以参考论坛执行计划详解与调优。

最后更新: 2025年8月4日

15.3.5 管理Schema

操作图空间

Studio 连接到 NebulaGraph 后，用户可以创建或删除图空间。用户可以使用 控制台 或者 Schema 操作图空间。本文仅说明如何使用 Schema 操作图空间。

前提条件

操作图空间之前，用户需要确保以下信息：

- Studio 已经连接到 NebulaGraph 。
- 当前登录的账号拥有创建或删除图空间的权限，即：
- 如果 NebulaGraph 未开启身份验证，用户以默认用户名 `user` 账号和任意密码登录。
- 如果 NebulaGraph 已开启身份验证，用户以 `root` 账号及其密码登录。

创建图空间

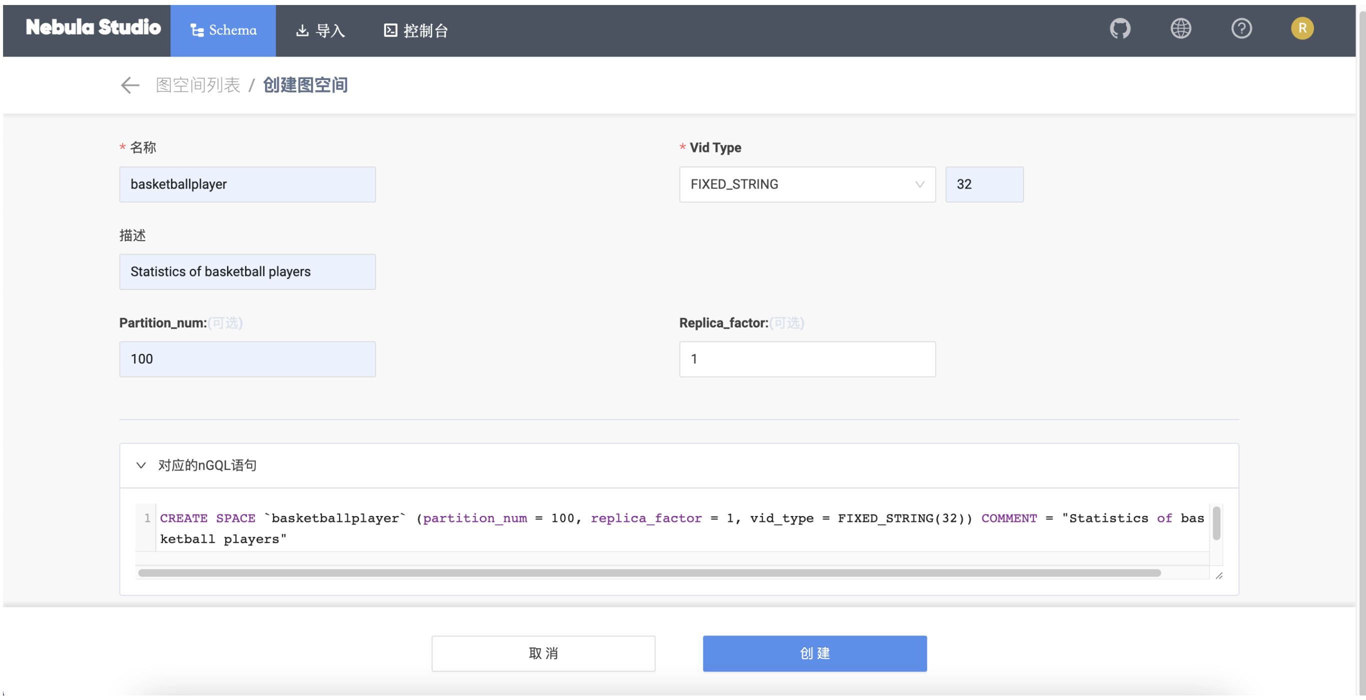
1. 在顶部导航栏里，点击 Schema 页签。
2. 在图空间列表上方，点击 + 创建图空间 按钮，完成以下配置：

- 名称：指定图空间名称，本示例中设置为 `basketballplayer` 。不可与已有的图空间名称重复。
- vid type：图空间中点 ID（VID）的数据类型，可选值为定长字符串 `FIXED_STRING(<N>)` 或 `INT64` ，一旦定义无法修改。本示例设置为 `FIXED_STRING(32)` 。详细信息，参考 VID。
- 描述：图空间的描述，最大为 256 字节。默认无描述。本示例设置为 `Statistics of basketball players` 。
- 可选参数：分别设置 `partition_num` 、 `replica_factor` 的值。在本示例中，两个参数分别设置为 `100` 、 `1` 。详细信息，参考 `CREATE SPACE` 语法。

在 对应的 nGQL 语句 面板上，用户能看到上述设置对应的 nGQL 语句。如下所示：

```
CREATE SPACE basketballplayer (partition_num = 100, replica_factor = 1, vid_type = FIXED_STRING(32)) COMMENT = "Statistics of basketball players"
```

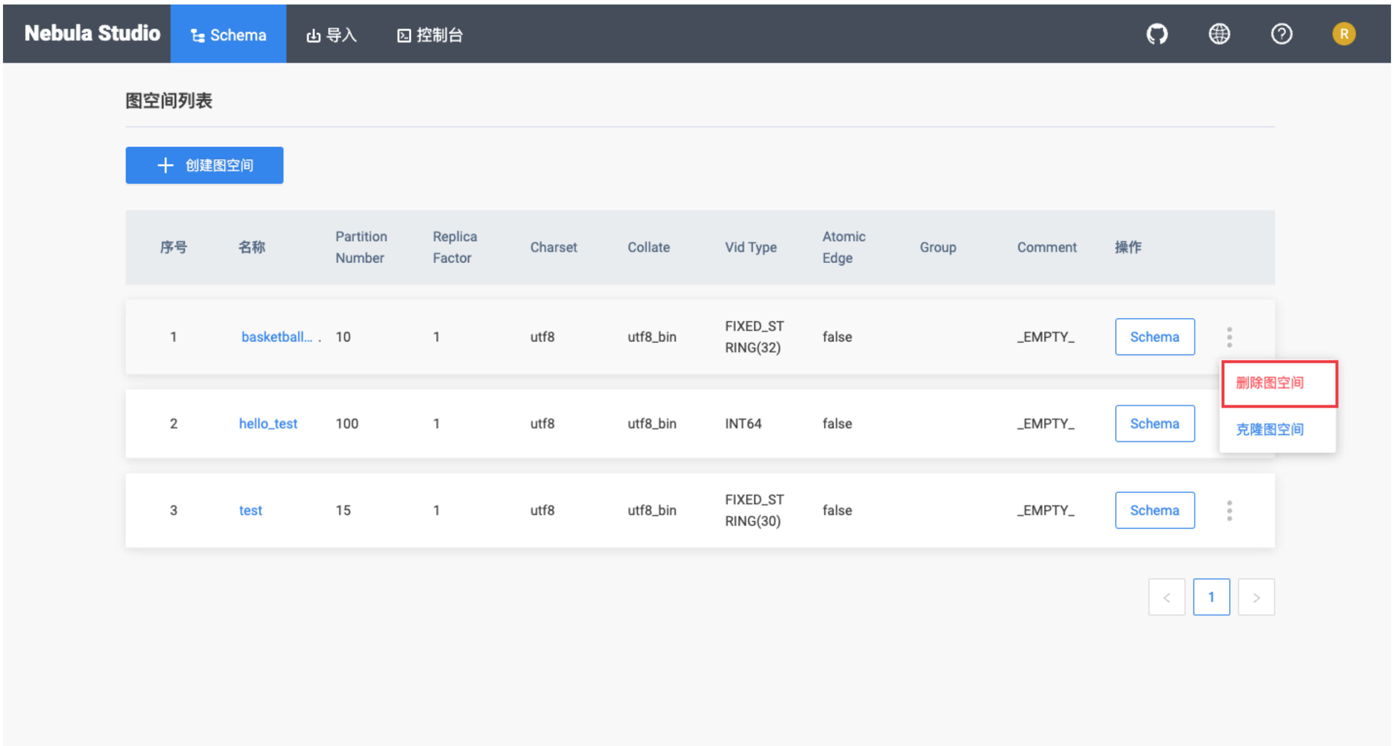
3. 配置确认无误后，点击 创建 按钮。 如果页面回到 图空间列表，而且列表中显示刚创建的图空间信息，表示图空间创建成功。



删除图空间

> Danger
>
> 删除图空间会删除其中的所有数据，已删除的数据如未备份无法恢复。

1. 在顶部导航栏里，点击 Schema 页签。
2. 在图空间列表里，找到需要删除的图空间，并在 操作 列中，选择 删除图空间。



3. 在弹出的对话框中点击 确认。

后续操作

图空间创建成功后，用户可以开始创建或修改 Schema，包括：

- 操作 Tag
- 操作 Edge type
- 操作索引

最后更新: 2025年8月4日

操作 **Tag**（点类型）

在 NebulaGraph 中创建图空间后，用户需要创建 Tag（点类型）。用户可以选择使用 控制台 或者 Schema 管理功能操作 Tag。本文仅说明如何使用 Schema 管理功能操作 Tag。

前提条件

在 Studio 上操作 Tag 之前，用户必须确认以下信息：

- Studio 已经连接到 NebulaGraph 。
- 图空间已经创建。
- 当前登录的账号拥有 GOD、ADMIN 或者 DBA 的权限。

创建 **TAG**

1. 在顶部导航栏中，点击 Schema 页签。
2. 在 图空间列表 中，找到图空间，点击图空间名称或者在 操作 列中点击 Schema。
3. 在 当前图空间 里确认图空间名称。用户也可以通过选择图空间名称切换图空间。
4. 点击 标签 页签，并点击 + 创建 按钮。
5. 在 创建标签 页面上，完成以下设置：

a. 名称：输入 Tag 名称。本示例中，输入 `player` 。

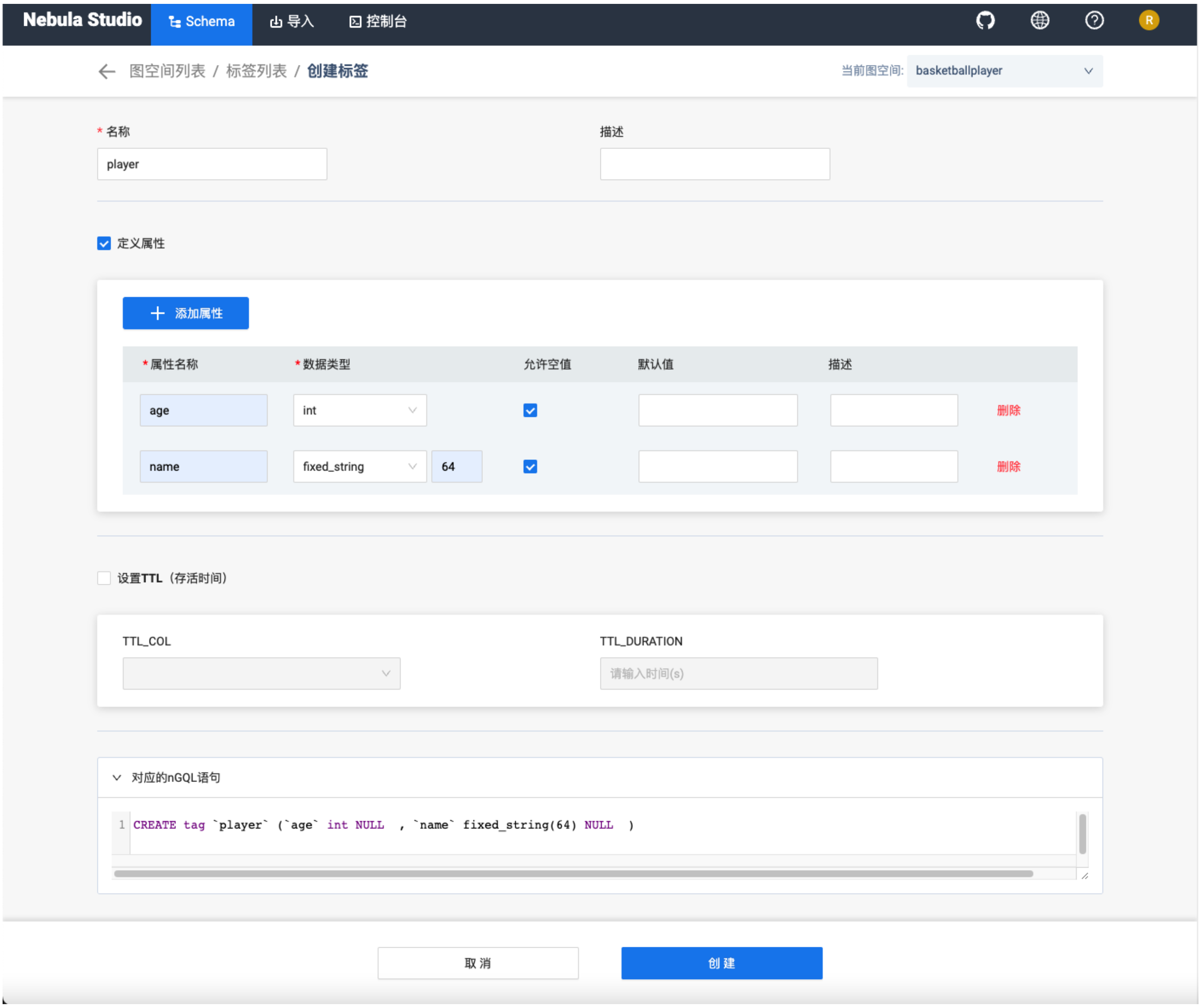
b. 描述（可选）：输入 Tag 的备注。

c. 定义属性（可选）：如果 Tag 需要属性，点击 + 添加属性，添加一个或多个属性：

- 输入属性名称。
- 选择数据类型。
- 选择是否允许空值。
- （可选）输入默认值。
- （可选）输入属性备注。

d. 设置TTL（存活时间）（可选）：Tag 未设置索引时，用户可以设置 TTL。勾选设置TTL（存活时间），设置 `TTL_COL` 和 `TTL_DURATION` （单位：秒）。详情参考 TTL 配置。

6. 完成设置后，在 对应的nGQL语句 面板，用户能看到与上述配置等价的 nGQL 语句。



7. 确认无误后，点击 创建 按钮。

   如果 Tag 创建成功，标签 面板会显示这个 Tag 的属性列表。

修改 TAG

1. 在顶部导航栏中，点击 Schema 页签。
2. 在 图空间列表 中，找到图空间，点击图空间名称或者在 操作 列中点击 Schema。
3. 在 当前图空间 里确认图空间名称。用户也可以通过选择图空间名称切换图空间。
4. 点击 标签 页签，找到需要修改的 Tag，并在 操作 列中，点击  图标。

5. 在 编辑 页面，用户可以选择以下操作：

- 如果要修改描述：在 描述 右侧点击 编辑，修改后点击 确认。
- 如果要修改属性：在 定义属性 区域，找到需要修改的属性，在右侧点击 编辑，修改后点击 确认。
- 如果要删除属性：在 定义属性 区域，找到需要删除的属性，在右侧点击 删除，然后点击 确认。
- 如果要添加属性：在 定义属性 区域，点击 + 添加属性，设置属性信息，点击 确认。详细说明参考创建 Tag 。
- 如果要修改 TTL 信息：在 设置 TTL 区域，点击 编辑，修改后点击 确认。详情参考 TTL 配置。
- 如果要删除已经配置的 TTL 信息：在 设置 TTL 区域，取消勾选 设置TTL（存活时间），然后点击 确定。
- 如果要配置 TTL 信息：在 设置 TTL 区域，勾选 设置TTL（存活时间），设置 `TTL_COL` 和 `TTL_DURATION` （单位：秒），点击 确认。详情参考 TTL 配置。

> Note
>
> TTL 与索引的共存问题，详情参考 TTL。

删除 TAG

> Danger
>
> 删除 Tag 前先确认影响，已删除的数据如未备份无法恢复。

1. 在顶部导航栏中，点击 Schema 页签。
2. 在 图空间列表 中，找到图空间，点击图空间名称或者在 操作 列中点击 Schema。
3. 在 当前图空间 里确认图空间名称。用户也可以通过选择图空间名称切换图空间。
4. 点击 标签 页签，找到需要删除的 Tag，并在 操作 列中，点击 图标。
5. 在弹出的对话框中点击 确认。

后续操作

Tag 创建成功后，用户可以在 控制台 上逐条插入点数据，或者使用 导入 功能批量插入点数据。

最后更新: 2025年8月4日

操作 **Edge type**

在 NebulaGraph 中创建图空间后，用户可能需要创建 Edge type（边类型）。用户可以选择使用 控制台 或者 Schema 操作 Edge type。本文仅说明如何使用 Schema 操作 Edge type。

前提条件

在 Studio 上操作 Edge type 之前，用户必须确认以下信息：

- Studio 已经连接到 NebulaGraph 。
- 图空间已经创建。
- 当前登录的账号拥有 GOD、ADMIN 或者 DBA 的权限。

创建边类型

1. 在顶部导航栏中，点击 Schema 页签。
2. 在 图空间列表 中，找到图空间，点击图空间名称或者在 操作 列中点击 Schema。
3. 在 当前图空间 里确认图空间名称。用户也可以通过选择图空间名称切换图空间。
4. 点击 边类型 页签，并点击 + 创建 按钮。
5. 在 创建边类型 页面上，完成以下设置：

a. 名称：输入 Edge type 名称。本示例中，输入 `serve` 。

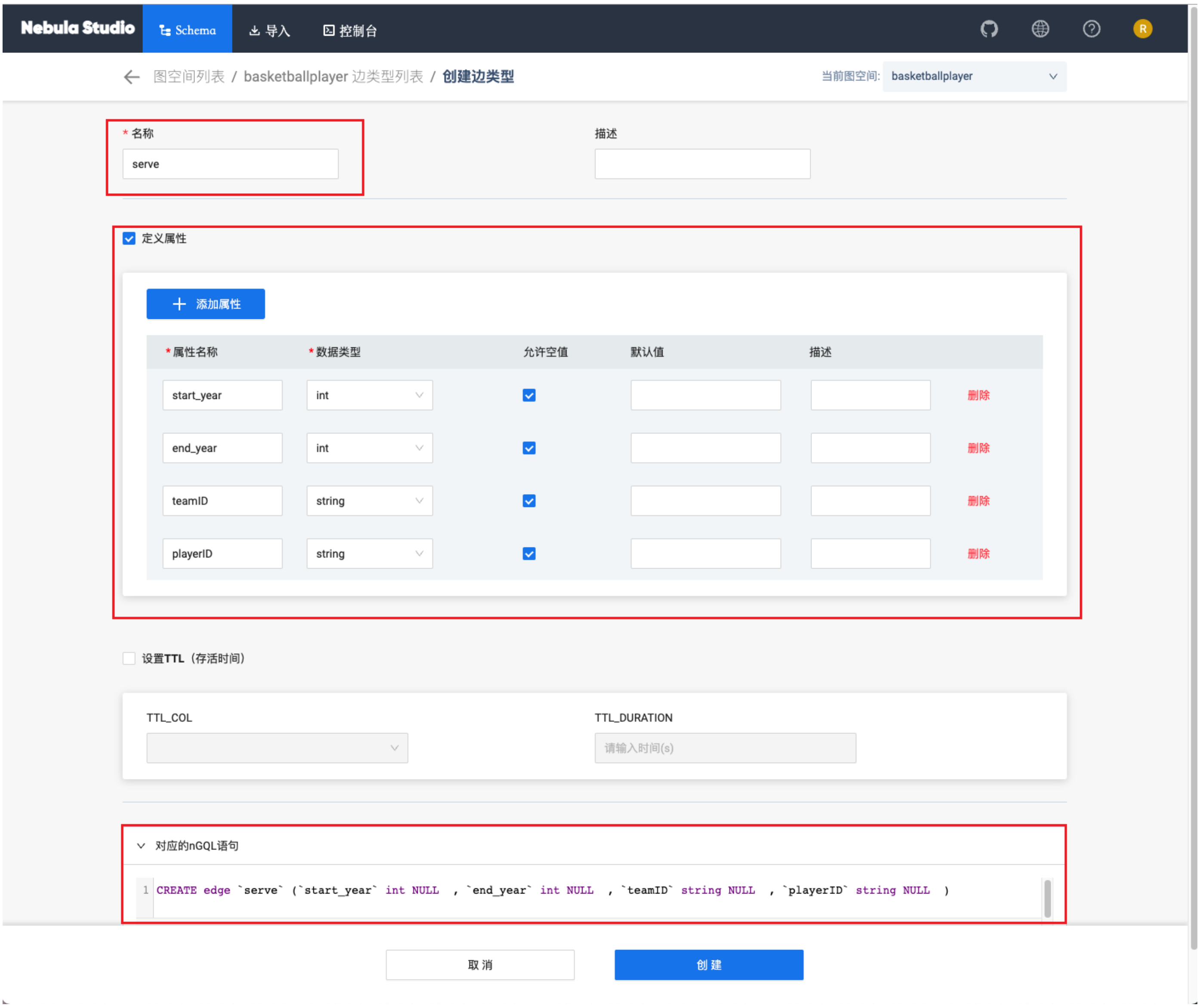
b. 描述（可选）：输入 Edge type 的备注。

c. 定义属性（可选）：如果 Edge type 需要属性，点击 + 添加属性，添加一个或多个属性：

- 输入属性名称。
- 选择数据类型。
- 选择是否允许空值。
- （可选）输入默认值。
- （可选）输入属性备注。

d. 设置TTL（存活时间）（可选）：Edge type 未设置索引时，用户可以设置 TTL。勾选设置TTL（存活时间），设置 `TTL_COL` 和 `TTL_DURATION` （单位：秒）。详情参考 TTL 配置。

6. 完成设置后，在 对应的 nGQL 语句 面板上，用户能看到与上述配置等价的 nGQL 语句。



7. 确认无误后，点击 创建 按钮。

如果 Edge Type 创建成功，边类型 面板会显示这个 Edge type 的属性列表。

修改 EDGE TYPE

按以下步骤使用 Schema 修改 Edge type：

1. 在顶部导航栏中，点击 Schema 页签。
2. 在 图空间列表 中，找到图空间，点击图空间名称或者在 操作 列中点击 Schema。
3. 在 当前图空间 里确认图空间名称。用户也可以通过选择图空间名称完成图空间切换。
4. 点击 边类型 页签，找到需要修改的 Edge Type，并在 操作 列中，点击 图标。
5. 在 编辑 页面，用户可以选择以下操作：

- 如果要修改描述：在 描述 右侧点击 编辑，修改后点击 确认。
- 如果要修改属性：在 定义属性 区域，找到需要修改的属性，在右侧点击 编辑，修改后点击 确认。
- 如果要删除属性：在 定义属性 区域，找到需要删除的属性，在右侧点击 删除，然后点击 确认。
- 如果要添加属性：在 定义属性 区域，点击 + 添加属性，设置属性信息，点击 确认。详细说明参考创建边类型 。
- 如果要修改 TTL 信息：在 设置 TTL 区域，点击 编辑，修改后点击 确认。详情参考 TTL 配置。
- 如果要删除已经配置的 TTL 信息：在 设置 TTL 区域，取消勾选 设置TTL（存活时间），然后点击 确定。
- 如果要配置 TTL 信息：在 设置 TTL 区域，勾选 设置TTL（存活时间），设置 `TTL_COL` 和 `TTL_DURATION` （单位：秒），点击 确认。详情参考 TTL 配置。

> Note
>
> TTL 与索引的共存问题，详情请见 TTL。

删除 EDGE TYPE

> Danger
>
> 删除 Edge type 前先确认影响，已删除的数据如未备份无法恢复。

1. 在顶部导航栏中，点击 Schema 页签。
2. 在 图空间列表 中，找到图空间，点击图空间名称或者在 操作 列中点击 Schema。
3. 在 当前图空间 里确认图空间名称。用户也可以通过选择图空间名称切换图空间。
4. 点击 边类型 页签，找到需要修改的 Edge type，并在 操作 列中，点击 图标。
5. 在弹出的对话框中点击 确认。

后续操作

Edge type 创建成功后，用户可以在 控制台 上逐条插入边数据，或者使用 导入 功能批量插入边数据。

最后更新: 2025年8月4日

操作索引

用户可以为 Tag 和 Edge type 创建索引，使得图查询时可以从拥有共同属性的同一类型的点或边开始遍历，使大型图的查询更为高效。用户可以选择使用 控制台 或者 Schema 操作索引。本文仅说明如何使用 Schema 操作索引。

> Note
>
> 一般在创建了 Tag 或者 Edge type 之后即可创建索引，但是，索引会影响写性能，所以，建议先导入数据，再批量重建索引。关于索引的详细信息，参考索引介绍。

前提条件

在 Studio 上操作索引之前，用户必须确认以下信息：

- Studio 已经连接到 NebulaGraph 。
- 图空间、Tag 和 Edge type 已经创建。
- 当前登录的账号拥有 GOD、ADMIN 或者 DBA 的权限。

创建索引

1. 在顶部导航栏中，点击 Schema 页签。
2. 在 图空间列表 中，找到图空间，点击图空间名称或者在 操作 列中点击 Schema。
3. 在 当前图空间 里确认图空间名称。用户也可以通过选择图空间名称切换图空间。
4. 点击 索引 页签，并点击 + 创建 按钮。
5. 在 创建索引 页面上，完成以下设置：

a. 索引类型：选择索引类型，即 Tag 或者 Edge type。本示例中选择 Edge type。

b. 关联名称：选择要关联的 Tag 或 Edge type 名称。本示例中选择 follow。

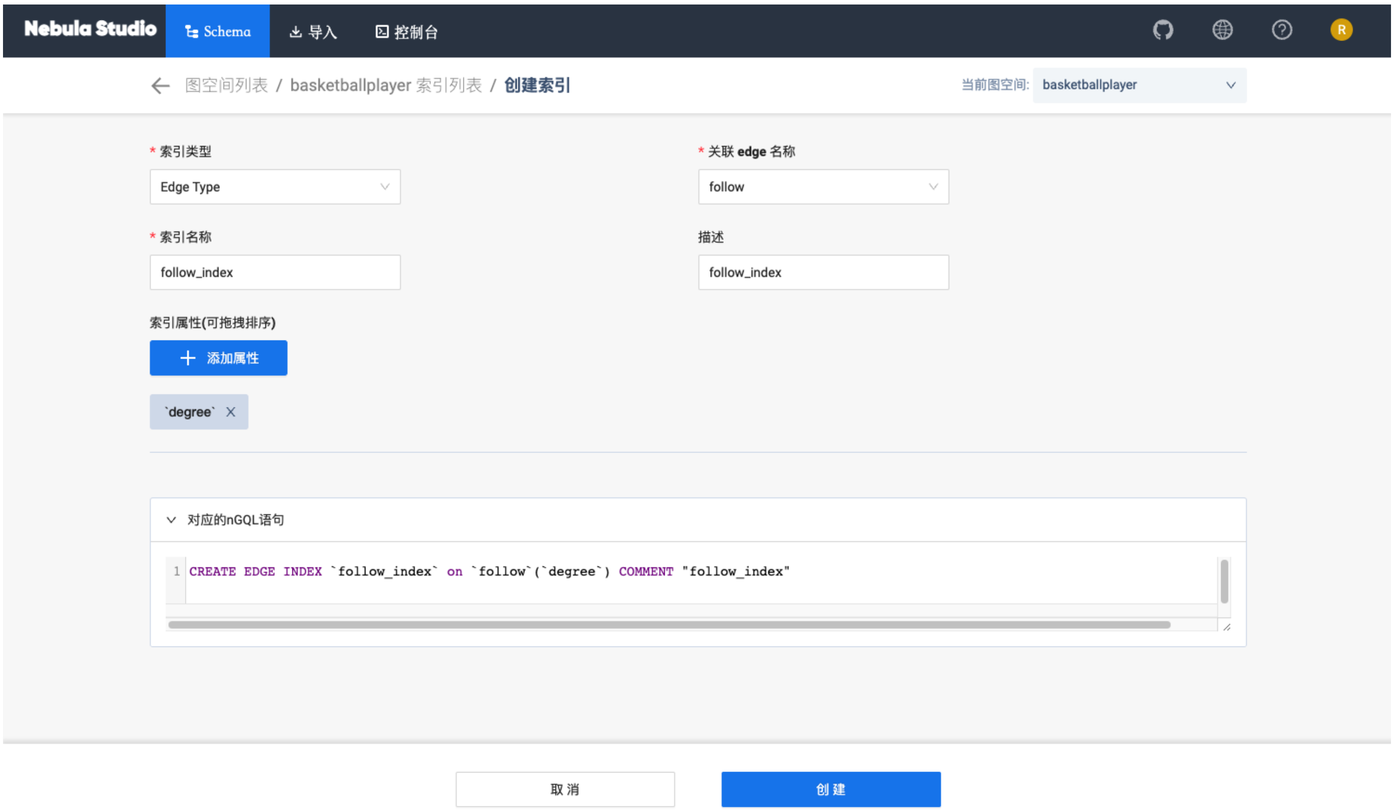
c. 索引名称：输入索引名称。本示例中，输入 follow_index。

d. 描述（可选）：输入索引的备注。

e. 索引属性（可选）：点击 + 添加属性，在弹出的对话框列表里选择需要关联的属性，点击 确认。如果需要关联多个属性，重复这一步骤。用户可以按界面提示重排索引属性的顺序。本示例中选择 `degree` 。

> Note
>
> 索引属性的顺序会影响 `LOOKUP` 语句的查询结果。详细信息，参考 LOOKUP。

6. 完成设置后，在 对应的 nGQL 面板，用户能看到与上述配置等价的 nGQL 语句。



7. 确认无误后，点击 创建 按钮。

   如果索引创建成功，索引 面板会显示这个索引的属性列表。

查看索引

1. 在顶部导航栏中，点击 Schema 页签。
2. 在 图空间列表 中，找到图空间，点击图空间名称或者在 操作 列中点击 Schema。
3. 在 当前图空间 里确认图空间名称。用户也可以通过选择图空间名称切换图空间。
4. 点击 索引 页签，在列表左上方，选择需要查看的索引类型。
5. 在列表中，找到需要查看的索引，点击索引所在行即可展开查看该索引相关的所有属性名称、数据类型。

重建索引

1. 在顶部导航栏中，点击 Schema 页签。
2. 在 图空间列表 中，找到图空间，点击图空间名称或者在 操作 列中点击 Schema。
3. 在 当前图空间 里确认图空间名称。用户也可以通过选择图空间名称切换图空间。
4. 点击 索引 页签，在列表左上方，选择索引类型。
5. 找到需要重建的索引，在 操作列 点击 重建索引。

> Note
>
> 关于重建索引，详情参见 REBUILD INDEX。

删除索引

1. 在顶部导航栏中，点击 Schema 页签。
2. 在 图空间列表 中，找到图空间，点击图空间名称或者在 操作 列中点击 Schema。

3. 在 当前图空间 里确认图空间名称。用户也可以通过选择图空间名称切换图空间。
4. 点击 索引 页签，在列表左上方，选择索引类型。
5. 找到需要删除的索引，并在 操作 列中，点击  图标。
6. 在弹出的对话框中点击 确认。

最后更新: 2025年8月4日

查看 Schema

用户可以在 Studio 上可视化地查看 Schema。

操作步骤

1. 在顶部导航栏里，点击 Schema 页签。
2. 在图空间列表中，找到图空间，单击图空间名称或者在操作列中单击 Schema。
3. 单击查看 Schema 页签，单击获取 Schema。

其他操作

在图空间列表中，找到图空间，在操作列可以执行如下操作：

- 查看 Schema DDL：显示该图空间的 Schema 创建语句，包括图空间、Tag、Edge type 和索引。
- 克隆图空间：克隆该图空间的 Schema 到新的图空间。
- 删除图空间：删除该图空间，包括 Schema 和所有点边数据。

最后更新: 2025年8月4日

15.3.6 Schema 草图

Studio 支持 Schema 草图功能。用户可以在画板上自行设计 Schema，可以直观展示点边关系，设计完成后可以将 Schema 应用到指定图空间。

功能说明

- 图形化设计 Schema。
- 应用 Schema 到指定图空间。
- 导出 Schema 为 PNG 格式图片。

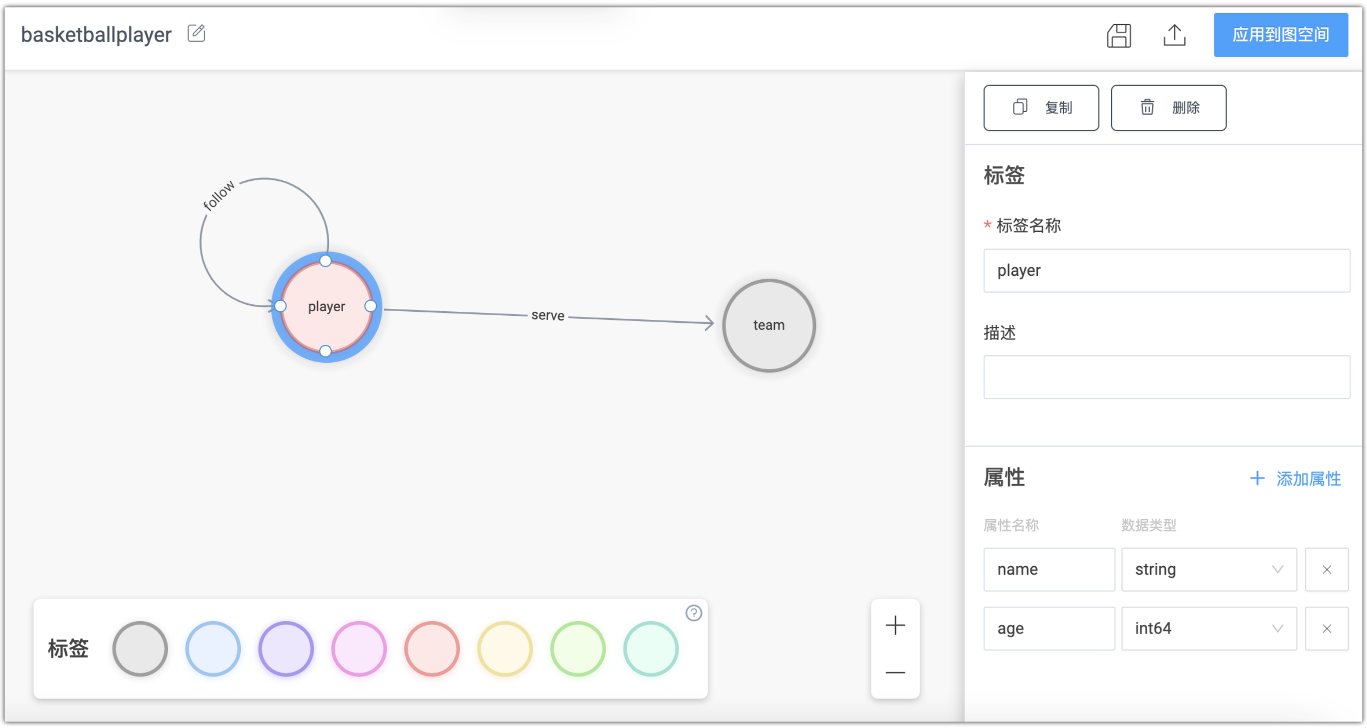
入口

在顶部导航栏里，点击 图标。

设计 **Schema**

以 `basketballplayer` 的 Schema 为例，说明如何设计 Schema。

1. 在页面左上角单击新建。
2. 在画布下方选择合适颜色的 Tag 标签，按住左键拖拽至画布中，创建一个 Tag。
3. 单击 Tag，在右侧填写标签名称 `player`、描述，并且添加属性 `name` 和 `age`。
4. 再次创建一个 Tag，标签名称为 `team`，属性为 `name`。
5. 从 Tag `player` 上的锚点连接至 Tag `team` 的锚点，单击生成的边，在右侧填写边类型名称 `serve`，并且添加属性 `start_year` 和 `end_year`。
6. 从 Tag `player` 上的锚点连接至自身另一个锚点，单击生成的边，在右侧填写边类型名称 `follow`，并且添加属性 `degree`。
7. 设计完成后，在页面上方单击 修改草图名称，然后在右上角单击 保存草图。



应用 **Schema**

1. 在页面左侧的草图列表内选择需要导入的 Schema 草图，然后在右上角单击应用到图空间。

2. 选择将 Schema 导入新建图空间或者已创建的图空间，单击确认。

Note

- 创建图空间的参数说明参见CREATE SPACE。
- 如果图空间中已有重名 Schema，导入操作会失败，并提示修改名称或更换图空间。

修改 **Schema**

在页面左侧的草图列表内选择需要修改的 Schema 草图，修改完成后在右上角单击 保存。

Note

已应用 Schema 的图空间不会同步修改。

删除 **Schema**

在页面左侧的草图列表内找到需要删除的 Schema 草图，在缩略图右上角单击X并确认即可删除。

导出 **Schema**

在页面右上角单击 可以导出 Schema 为 PNG 格式图片。

最后更新: 2025年8月4日

15.4 全局设置

本文介绍 Studio 的全局设置，包含语言切换和 Beta 功能。

- 语言：支持切换中文和英文。
- Beta 功能：Beta 功能开关。Beta 功能目前包括查看 Schema、文本转查询和 AI 导入。

  文本转查询和 AI 导入需要设置人工智能相关的配置。详细设置参见下文。

15.4.1 文本转查询和 AI 导入

文本转查询和 AI 导入是基于大规模语言模型（LLM）实现的人工智能，需要配置以下参数才能正常使用。

参数	说明
API type	API 接口类型。支持 `OpenAI` 和 `Aliyun`。
URL	API URL。请按照对应接口类型填写正确的 URL 格式。例如 `https://{your-resource-name}.openai.azure.com/openai/deployments/{deployment-id}/chat/completions?api-version={api-version}`。
key	用于验证 API 的 key。使用在线大语言模型时必填，使用离线大语言模型时根据实际设置选填。
model	大语言模型版本。使用在线大语言模型时必填，使用离线大语言模型时根据实际设置选填。
文本最大长度	接收或生成一段文本的最大长度限制。单位：字节。

最后更新: 2025年8月4日

15.5 故障排查

15.5.1 连接数据库错误

问题描述

按连接 Studio 文档操作，提示 配置失败。

可能的原因及解决方法

用户可以按以下步骤排查问题。

第 1 步。确认 HOST 字段的格式是否正确

必须填写 NebulaGraph 图数据库 Graph 服务的 IP 地址（`graph_server_ip`）和端口。如果未做修改，端口默认为 `9669`。即使 NebulaGraph 与 Studio 都部署在当前机器上，用户也必须使用本机 IP 地址，而不能使用 `127.0.0.1`、 `localhost` 或者 `0.0.0.0`。

第 2 步。确认 用户名 和 密码 是否正确

如果 NebulaGraph 没有开启身份认证，用户可以填写任意字符串登录。

如果已经开启身份认证，用户必须使用分配的账号登录。

第 3 步。确认 NEBULAGRAPH 服务是否正常

检查 NebulaGraph 服务状态。关于查看服务的操作：

- 如果在 Linux 服务器上通过编译部署的 NebulaGraph，参考查看 NebulaGraph 服务。
- 如果使用 Docker Compose 部署和 RPM 部署的 NebulaGraph，参考查看 NebulaGraph 服务状态和端口。

如果 NebulaGraph 服务正常，进入第 4 步继续排查问题。否则，请重启 NebulaGraph 服务。

> Note
>
> 如果之前使用 `docker-compose up -d` 启动 NebulaGraph，必须运行 `docker-compose down` 命令停止 NebulaGraph 。

第 4 步。确认 GRAPH 服务的网络连接是否正常

在 Studio 机器上运行命令（例如 `telnet <graph_server_ip> 9669`）确认 NebulaGraph 的 Graph 服务网络连接是否正常。

如果连接失败，则按以下要求检查：

- 如果 Studio 与 NebulaGraph 在同一台机器上，检查端口是否已暴露。
- 如果两者不在同一台机器上，检查 NebulaGraph 服务器的网络配置，例如，防火墙、网关以及端口。

如果按上述步骤排查后仍无法连接 NebulaGraph 服务，请前往 NebulaGraph 官方论坛咨询。

最后更新: 2025年8月4日

15.5.2 无法访问 Studio

问题描述

我按照文档描述启动 Studio 后访问 `127.0.0.1:7001` 或者 `0.0.0.0:7001`，但是打不开页面，为什么？

可能的原因及解决方法

用户可以按以下顺序排查问题。

1.确认系统架构

需要确认部署 Studio 服务的机器是否为 x86_64 架构。目前 Studio 仅支持 x86_64 系统架构。

2.检查 STUDIO 服务是否正常启动

- 使用 RPM 或 DEB 包部署的 Studio，使用 `systemctl status nebula-graph-studio` 查看运行状态。
- 使用 tar 包部署的 Studio，使用 `sudo lsof -i:7001` 查看端口状态。
- 使用 Docker-compose 部署的 Studio，使用 `docker-compose ps` 查看运行状态。

  如果服务正常，返回结果如下。其中， `State` 列应全部显示为 `Up` 。

```
  Name             Command        State        Ports
------------------------------------------------------------------------------------------------------
nebula-web-docker_client_1   ./nebula-go-api         Up   0.0.0.0:32782->8080/tcp
nebula-web-docker_importer_1  nebula-importer --port=569 ... Up   0.0.0.0:32783->5699/tcp
nebula-web-docker_nginx_1    /docker-entrypoint.sh ngin ... Up   0.0.0.0:7001->7001/tcp, 80/tcp
nebula-web-docker_web_1     docker-entrypoint.sh npm r ... Up   0.0.0.0:32784->7001/tcp
```

> Note
>
> 如果之前使用 `docker-compose up -d` 启动 Studio，必须运行 `docker-compose down` 命令停止 Studio。

如果服务没有正常运行，请重新启动 Studio。详细信息，参考部署 Studio。

3.确认访问地址

如果 Studio 与浏览器在同一台机器上，用户可以在浏览器里使用 `localhost:7001` 、 `127.0.0.1:7001` 或者 `0.0.0.0:7001` 访问 Studio。

如果两者不在同一台机器上，必须在浏览器里输入 `<studio_server_ip>:7001` 。其中， `studio_server_ip` 是指部署 Studio 服务的机器的 IP 地址。

4.确认网络连通性

运行 `curl <studio_server_ip>:7001 -I` 确认是否正常。如果返回 `HTTP/1.1 200 OK` ，表示网络连通正常。

如果连接被拒绝，则按以下要求检查：

- 如果浏览器与 Studio 在同一台机器上，检查端口是否已暴露。
- 如果两者不在同一台机器上，检查 Studio 所在机器的网络配置，例如，防火墙、网关以及端口。

如果按上述步骤排查后仍无法访问 Studio，请前往 NebulaGraph 官方论坛咨询。

最后更新: 2025年8月4日

15.5.3 常见问题

为什么我无法使用某个功能？

如果发现无法使用某个功能，建议按以下步骤排除问题：

1. 确认 NebulaGraph 是最新版本。如果使用 Docker Compose 部署 NebulaGraph 数据库，建议运行 `docker-compose pull && docker-compose up -d` 拉取最新的 Docker 镜像，并启动容器。
2. 确认 Studio 是最新版本。详细信息参考版本更新。
3. 搜索论坛或 GitHub 的 nebula 和 nebula-web-docker 项目，确认是否已经有类似的问题。
4. 如果上述操作均未解决问题，欢迎在论坛上提交问题。

最后更新: 2025年8月4日

16. NebulaGraph Dashboard（社区版）

16.1 什么是 NebulaGraph Dashboard（社区版）

NebulaGraph Dashboard（简称 Dashboard）是一款用于监控 NebulaGraph 集群中机器和服务状态的可视化工具。

> Enterpriseonly
>
> 企业版增加了可视化创建集群、批量导入集群、快速扩缩容等功能，点击定价查看更多。用户还可以在阿里云上免费试用。

16.1.1 产品功能

- 监控集群中所有机器的状态，包括 CPU、内存、负载、磁盘和流量。
- 监控集群中所有服务的信息，包括服务 IP 地址、版本和监控指标（例如查询数量、查询延迟、心跳延迟等）。
- 监控集群本身的信息，包括集群的服务信息、分区信息、配置和长时任务。
- 支持全局调整监控数据的页面更新频率。

16.1.2 适用场景

如果有以下任一需求，都可以使用 Dashboard：

- 需要方便快捷地监测关键指标，集中呈现业务的多个重点信息，保证业务正常运行。
- 需要多维度（例如时间段、聚合规则、指标）监控集群。
- 故障发生后，需要复盘问题，确认故障发生时间、异常现象。

16.1.3 注意事项

监控数据默认保留 14 天，即只能查询最近 14 天内任意时间段的监控数据。

> Note
>
> 监控服务由 prometheus 提供，更新频率和保留时间等都可以自行修改。详情请参见 prometheus 官方文档。

16.1.4 版本兼容性

NebulaGraph 的版本和 Dashboard 社区版的版本对应关系如下。

NebulaGraph 版本	Dashboard 版本
3.6.0	3.4.0
3.5.x	3.4.0
3.4.0 ~ 3.4.1	3.4.0、3.2.0
3.3.0	3.2.0
2.5.0 ~ 3.2.0	3.1.0
2.5.x ~ 3.1.0	1.1.1
2.0.1 ~ 2.5.1	1.0.2
2.0.1 ~ 2.5.1	1.0.1

16.1.5 更新说明

Release

最后更新: 2025年8月4日

16.2 部署 Dashboard 社区版

本文将介绍如何通过 TAR 包安装部署 NebulaGraph Dashboard。

下载和编译 Dashboard 的最新源码，参见 GitHub NebulaGraph dashboard 页面的说明。

16.2.1 前提条件

在部署 Dashboard 之前，用户需要确认以下信息：

- NebulaGraph 服务已经部署并启动。详细信息参考 NebulaGraph 安装部署。
- 确保以下端口未被使用：
- 9200
- 9100
- 9090
- 8090
- 7003
- 待监控的机器上已经安装 node-exporter。安装方法请参见 Prometheus 官方文档。

16.2.2 操作步骤

1. 下载 TAR 包 nebula-dashboard-3.4.0.x86_64.tar.gz。
2. 执行命令 `tar -xvf nebula-dashboard-3.4.0.x86_64.tar.gz` 解压缩。
3. 进入解压缩的 `nebula-dashboard` 文件夹，并修改配置文件 `config.yaml` 。

   配置文件内主要包含 4 种依赖服务的配置和集群的配置。依赖服务的说明如下。

服务名称	默认端口号	说明
nebula-http-gateway	8090	为集群服务提供 HTTP 接口，执行 nGQL 语句与 NebulaGraph 进行交互。
nebula-stats-exporter	9200	收集集群的性能指标，包括服务 IP 地址、版本和监控指标（例如查询数量、查询延迟、心跳延迟等）。
node-exporter	9100	收集集群中机器的资源信息，包括 CPU、内存、负载、磁盘和流量。
prometheus	9090	存储监控数据的时间序列数据库。

配置文件说明如下。

```yaml
port: 7003  # Web 服务端口。
gateway:
  ip: hostIP  # 部署 Dashboard 的机器 IP。
  port: 8090
  https: false  # 是否为 HTTPS 端口。
  runmode: dev  # 程序运行模式，包括 dev、test、prod。一般用于区分不同运行环境。
stats-exporter:
  ip: hostIP  # 部署 Dashboard 的机器 IP。
  nebulaPort: 9200
  https: false  # 是否为 HTTPS 端口。
node-exporter:
  - ip: nebulaHostIP_1 # 部署 NebulaGraph 的机器 IP。
    port: 9100
    https: false # 是否为 HTTPS 端口。
# - ip: nebulaHostIP_2
#   port: 9100
#   https: false
prometheus:
  ip: hostIP  # 部署 Dashboard 的机器 IP。
  prometheusPort: 9090
  https: false  # 是否为 HTTPS 端口。
```

```yaml
  scrape_interval: 5s # 收集监控数据的间隔时间。默认为 1 分钟。
  evaluation_interval: 5s # 告警规则扫描时间间隔。默认为 1 分钟。
# 集群节点信息
nebula-cluster:
  name: 'default' # 集群名称
  metad:
    - name: metad0
      endpointIP: nebulaMetadIP  # 部署 Meta 服务的机器 IP。
      port: 9559
      endpointPort: 19559
  # - name: metad1
  #   endpointIP: nebulaMetadIP
  #   port: 9559
  #   endpointPort: 19559
  graphd:
    - name: graphd0
      endpointIP: GraphdIP  # 部署 Graph 服务的机器 IP。
      port: 9669
      endpointPort: 19669
  # - name: graphd1
  #   endpointIP: GraphdIP
  #   port: 9669
  #   endpointPort: 19669
  storaged:
    - name: storaged0
      endpointIP: StoragedIP  # 部署 Storage 服务的机器 IP。
      port: 9779
      endpointPort: 19779
  # - name: storaged1
  #   endpointIP: StoragedIP
  #   port: 9779
  #   endpointPort: 19779
```

4. 执行 `./dashboard.service start all` 一键启动服务。

容器部署

如果使用容器部署 Dashboard，同样是修改配置文件 `config.yaml`，修改完成后，执行 `docker-compose up -d` 即可启动容器。

> Note
>
> 如果修改了 `config.yaml` 内的端口号，`docker-compose.yaml` 里的端口号也需要保持一致。

执行 `docker-compose stop` 命令停止容器部署的 Dashboard。

16.2.3 管理 Dashboard 服务

Dashboard 使用脚本 `dashboard.service` 管理服务，包括启动、重启、停止和查看状态。

```
sudo <dashboard_path>/dashboard.service
[-v] [-h]
<start|restart|stop|status> <prometheus|webserver|exporter|gateway|all>
```

参数	说明
`dashboard_path`	Dashboard 安装路径。
`-v`	显示详细调试信息。
`-h`	显示帮助信息。
`start`	启动服务。
`restart`	重启服务。
`stop`	停止服务。
`status`	查看服务状态。
`prometheus`	管理 prometheus 服务。
`webserver`	管理 webserver 服务。
`exporter`	管理 exporter 服务。
`gateway`	管理 gateway 服务。
`all`	管理所有服务。

Note

查看 Dashboard 版本可以使用命令 `./dashboard.service -version`。

16.2.4 后续操作

连接 Dashboard

最后更新: 2025年8月4日

16.3 连接 Dashboard

Dashboard 部署完成后，可以通过浏览器登录使用 Dashboard。

16.3.1 前提条件

- Dashboard 相关服务已经启动。详情请参见部署 Dashboard。
- 建议使用 Chrome 89 及以上的版本的 Chrome 浏览器，否则可能有兼容问题。

16.3.2 操作步骤

1. 确认 Dashboard 服务所在机器的 IP 地址，在浏览器中输入 `<IP>:7003` 打开登录页面。
2. 输入 NebulaGraph 数据库的账号和密码。

- 如果 NebulaGraph 已经启用身份验证，用户可以使用已创建的账号连接 Dashboard。
- 如果 NebulaGraph 未启用身份验证，用户只能使用默认用户 `root` 和任意密码连接 Dashboard。

有关如何启用身份验证请参见身份验证。

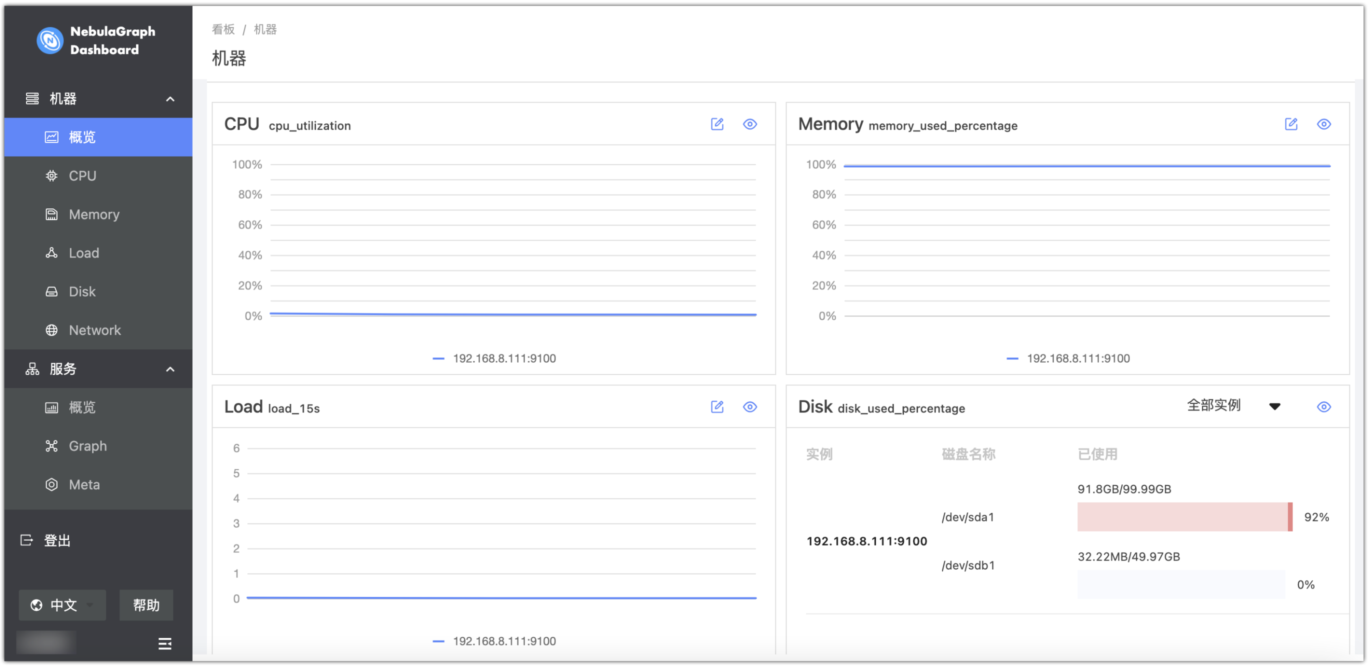
3. 单击登录。

最后更新: 2025年8月4日

16.4 Dashboard 页面介绍

Dashboard 页面主要分为机器、服务、管理三个部分，本文将详细介绍这些界面。

16.4.1 页面概览

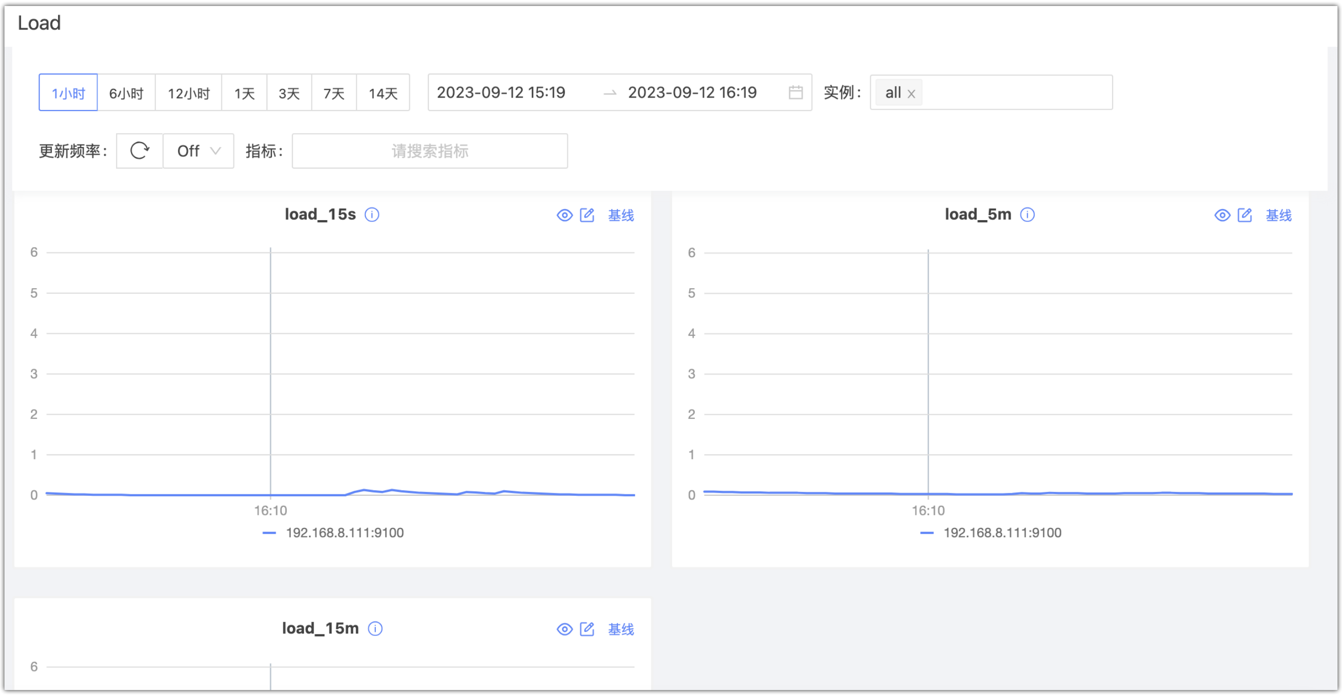


16.4.2 机器页面介绍

单击机器->概览进入机器概览页面。

用户可快速查看 CPU、Memory、Load、Disk 和 Network In/Out 变化情况。

- 默认最多可选择 14 天的监控数据进行查看，支持选择时间段，也可以快捷选择最近 1 小时、6 小时、12 小时、1 天、3 天、7 天和 14 天。
- 默认勾选显示所有实例的监控数据，可以在实例框内调整。
- 页面的监控数据默认不自动更新，可以调整更新频率让页面自动更新，也可以单击  按钮手动更新。
- 如果需要设置基线，作为参考标准线，可以单击模块右上角的  按钮。
- 如果需要查看某一项更详细的监控指标，可以单击模块右上角的  按钮，在示例中选择 `Load` 查看详情信息，如下图。



- 可以设置监控时间段、实例、更新频率和基线。
- 可以搜索和勾选指标。监控指标详情请参见监控指标说明。
- 可以暂时隐藏不需要查看的节点。
- 可以单击  按钮查看指标详情。

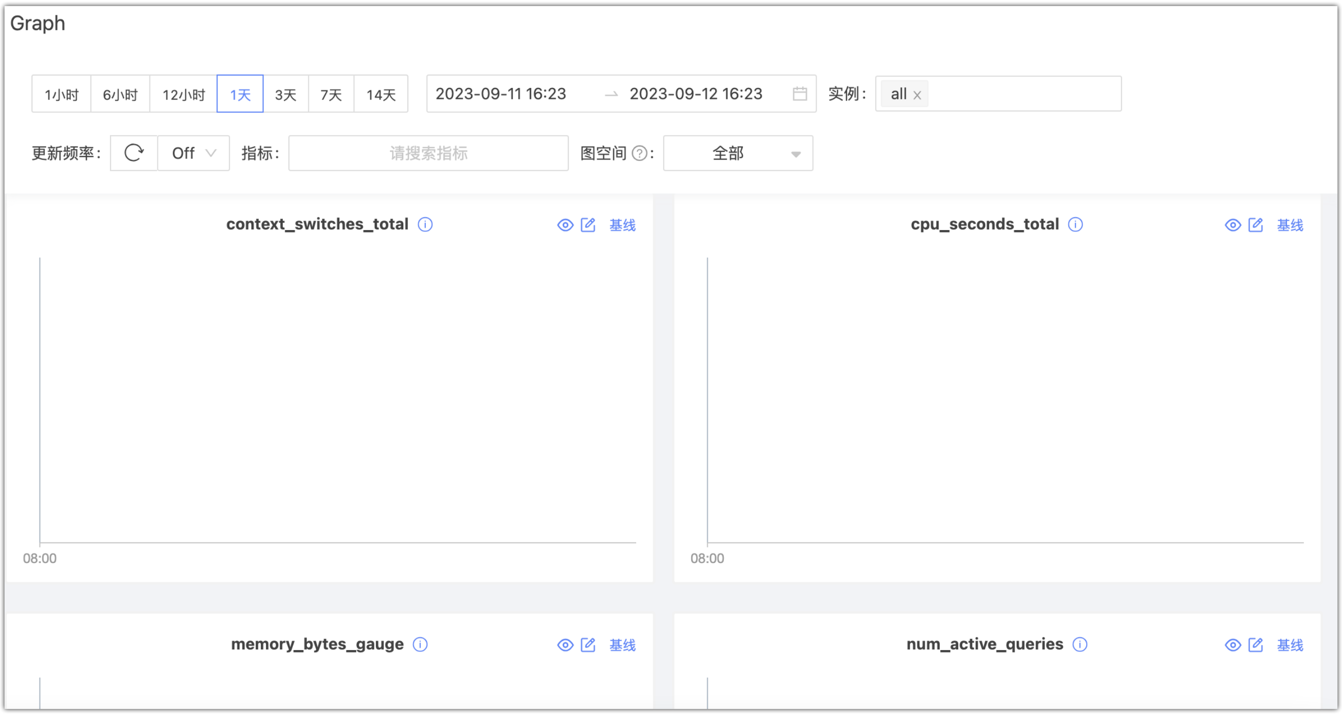
16.4.3 服务页面介绍

单击服务->概览进入服务概览页面。

用户可快速查看 Graph、Meta、Storage 服务的信息。

> Note
>
> 服务监控页仅支持每种服务设置两个监控指标，可以单击模块内的设置按钮进行调整。

- 默认最多可选择 14 天的监控数据进行查看，支持选择时间段，也可以快捷选择最近 1 小时、6 小时、12 小时、1 天、3 天、7 天和 14 天。
- 默认勾选显示所有实例的监控数据，可以在实例框内调整。
- 页面的监控数据默认不自动更新，可以调整更新频率让页面自动更新，也可以单击 按钮手动更新。
- 可以查看集群内所有服务的状态。
- 如果需要查看某一项更详细的监控指标，可以单击模块右上角的 按钮，在示例中选择 `Graph` 查看详情信息，如下图。



- 可以设置监控时间段、实例、更新频率、周期、聚合方式和基线。
- 可以搜索和勾选指标。监控指标详情请参见监控指标说明。
- 可以暂时隐藏不需要查看的节点。
- 可以单击 按钮查看指标详情。
- Graph 服务支持一系列基于图空间的监控指标。详情参见下文图空间监控。

图空间监控

> Caution
>
> 使用图空间指标前，用户需要在 Graph 服务中，设置 `enable_space_level_metrics` 为 `true`。具体操作，参见Graph 服务配置。

> **监控指标不兼容性**
>
> 如果图空间的名称包括特殊字符，可能会有图空间监控指标数据不显示的问题。

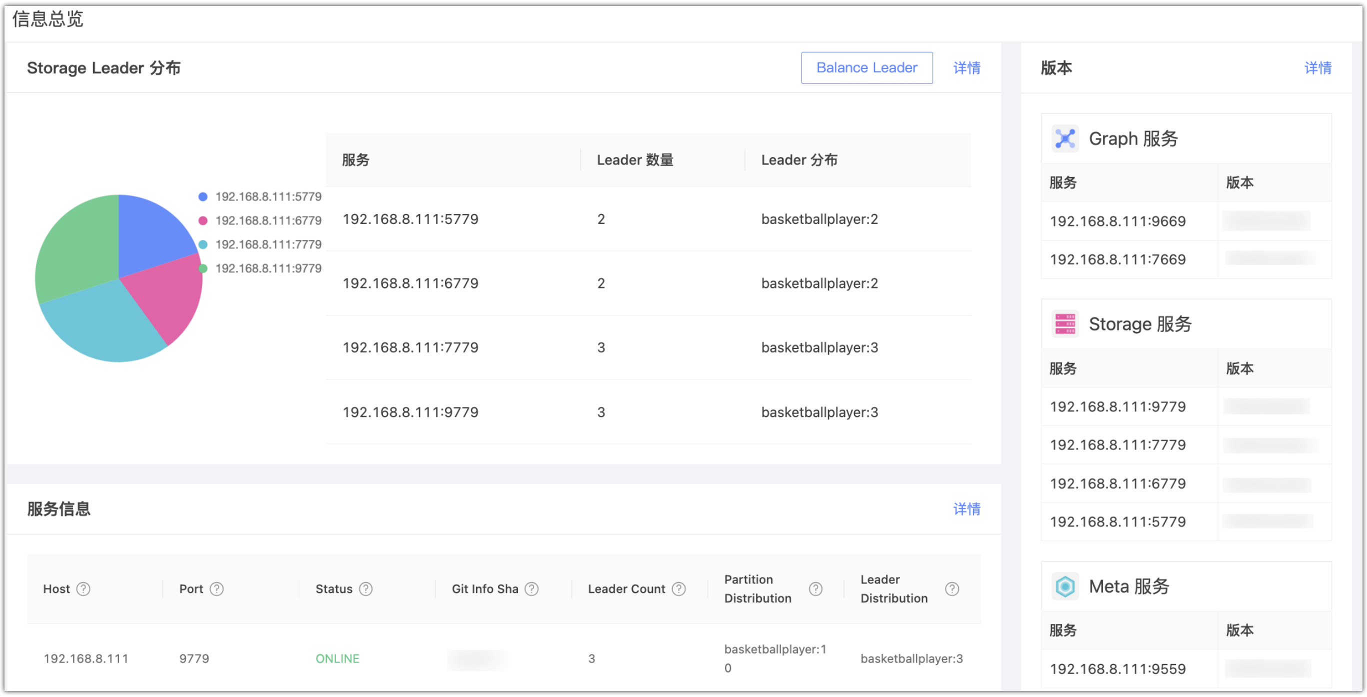
服务监控页面还可以监控图空间级别指标。只有当图空间指标的行为被触发后，用户才能指定图空间查看对应的图空间指标的信息。图空间的监控指标记录不同图空间的数据。目前，只有 Graph 服务支持基于图空间的监控指标。

Dashboard 支持的图空间指标，参见图空间级别监控指标。

16.4.4 管理页面介绍

信息总览

用户可以在信息总览页面查看 NebulaGraph 内核相关信息，包括 Storage 服务信息、Storage Leader 分布、 NebulaGraph 各个服务的版本信息及所在节点信息、分片的分布情况及详细信息。



STORAGE LEADER 分布

显示 Leader 数量及 Leader 的分布。

- 单击右上角的 Balance Leader 按钮可以快速在 NebulaGraph 集群中均衡分布 Leader。关于 Leader 的详细信息，参见 Storage 服务。
- 单击右上角的详情，查看 Leader 分布的详细信息。

版本

显示 NebulaGraph 所有服务版本及服务地址。单击右上角的详情，查看更多信息。

服务信息

展示 Storage 服务的基本信息。参数说明如下：

参数	说明
`Host`	主机地址
`Port`	主机端口号
`Status`	主机状态
`Git Info Sha`	版本 Commit ID
`Leader Count`	Leader 总数
`Partition Distribution`	分片分布
`Leader Distribution`	Leader 分布

单击右上角的详情，查看更多信息。

PARTITION 分布

左上方选择指定图空间，查看图空间的分片分布情况。显示所有 Storage 服务的 IP 地址、端口，及对应 Storage 服务中的分片数量。

单击右上角的详情，查看更多信息。

分片信息

显示分片信息。用户需要在左上角选择图空间，查看分片信息。参数说明如下：

参数	说明
`Partition ID`	分片序号。
`Leader`	分片的 leader 副本的 IP 地址和端口。
`Peers`	分片所有副本的 IP 地址和端口。
`Losts`	分片的故障副本的 IP 地址和端口。

单击右上角的详情，查看更多信息，通过右上角的输入框，输入分片 ID，筛选展示的数据。

配置

展示服务的配置信息。暂不支持在线修改配置。

16.4.5 其他

在页面左下角，还可以进行如下操作：

- 退出登录
- 切换中英文
- 查看当前 Dashboard 版本
- 查看帮助信息
- 折叠侧边栏

最后更新: 2025年8月4日

16.5 监控指标说明

本文介绍 Dashboard 中展示的 NebulaGraph 监控指标。

16.5.1 机器

Note

- 以下机器指标适用于 Linux 操作系统。
- 磁盘容量和流量的默认单位为字节（Byte），页面显示时单位会随着数据量级而变化，例如流量低于 1 KB/s 时单位为 Bytes/s。
- 对于高于 v1.0.2 版本的社区版 Dashboard，机器的 Buff 和 Cache 所占的内存没有被计算在内存使用率中。

CPU

参数	说明
`cpu_utilization`	CPU 已使用百分比
`cpu_idle`	CPU 空闲百分比
`cpu_wait`	等待 IO 操作的 CPU 百分比
`cpu_user`	用户空间（非 NebulaGraph 图空间）占用的 CPU 百分比
`cpu_system`	内核空间（非 NebulaGraph 内核空间）占用的 CPU 百分比

内存

参数	说明
`memory_utilization`	内存已使用百分比
`memory_used`	已使用内存（不包括缓存）。
`memory_free`	空闲内存

负载

参数	说明
`load_1m`	最近 1 分钟系统平均负载
`load_5m`	最近 5 分钟系统平均负载
`load_15m`	最近 15 分钟系统平均负载

磁盘

参数	说明
`disk_used_percentage`	磁盘使用率
`disk_used`	磁盘已使用存储空间
`disk_free`	磁盘剩余存储空间
`disk_readbytes`	磁盘每秒读取的字节数
`disk_writebytes`	磁盘每秒写入的字节数
`disk_readiops`	磁盘每秒的读请求数量
`disk_writeiops`	磁盘每秒的写请求数量
`inode_utilization`	inode 已使用百分比

流量

参数	说明
`network_in_rate`	网卡每秒接收的字节数
`network_out_rate`	网卡每秒发送的字节数
`network_in_errs`	网卡每秒接收错误的字节数
`network_out_errs`	网卡每秒发送错误的字节数
`network_in_packets`	网卡每秒接收的数据包数量
`network_out_packets`	网卡每秒发送的数据包数量

16.5.2 服务

周期

指标统计的时间范围，当前支持 5 秒、60 秒、600 秒和 3600 秒，分别表示最近 5 秒、最近 1 分钟、最近 10 分钟和最近 1 小时。

聚合方式

参数	说明
`rate`	周期内平均每秒操作的速率
`sum`	周期内操作的总和
`avg`	周期内响应平均耗时
`P75`	周期内响应耗时从小到大排列，顺序处于 75%位置的分位数
`P95`	周期内响应耗时从小到大排列，顺序处于 95%位置的分位数
`P99`	周期内响应耗时从小到大排列，顺序处于 99%位置的分位数
`P999`	周期内响应耗时从小到大排列，顺序处于 99.9%位置的分位数

> Note
>
> 以下为 Dashboard 获取内核的全量指标，但 Dashboard 仅展示重要的指标。

Graph

参数	说明
`num_active_queries`	活跃的查询语句数的变化数。 计算公式：时间范围内开始执行的语句数减去执行完毕的语句数。
`num_active_sessions`	活跃的会话数的变化数。 计算公式：时间范围内登录的会话数减去登出的会话数。 例如查询num_active_sessions.sum.5，过去 5 秒中登录了 10 个会话数，登出了 30 个会话数，那么该指标值就是 `-20`（10-30）。
`num_aggregate_executors`	聚合（Aggregate）算子执行时间。
`num_auth_failed_sessions_bad_username_password`	因用户名密码错误导验证失败的会话数量。
`num_auth_failed_sessions_out_of_max_allowed`	因为超过 `FLAG_OUT_OF_MAX_ALLOWED_CONNECTIONS` 参数导致的验证登录的失败的会话数量。
`num_auth_failed_sessions`	登录验证失败的会话数量。
`num_indexscan_executors`	索引扫描（IndexScan）算子执行时间。
`num_killed_queries`	被终止的查询数量。
`num_opened_sessions`	服务端建立过的会话数量。
`num_queries`	查询次数。
`num_query_errors_leader_changes`	因 Leader 变更而导致的查询错误的次数。
`num_query_errors`	查询错误次数。
`num_reclaimed_expired_sessions`	服务端主动回收的过期的会话数量。
`num_rpc_sent_to_metad_failed`	Graphd 服务发给 Metad 的 RPC 请求失败的数量。
`num_rpc_sent_to_metad`	Graphd 服务发给 Metad 服务的 RPC 请求数量。
`num_rpc_sent_to_storaged_failed`	Graphd 服务发给 Storaged 服务的 RPC 请求失败的数量。
`num_rpc_sent_to_storaged`	Graphd 服务发给 Storaged 服务的 RPC 请求数量。
`num_sentences`	Graphd 服务接收的语句数。
`num_slow_queries`	慢查询次数。
`num_sort_executors`	排序（Sort）算子执行次数。
`optimizer_latency_us`	优化器阶段延迟时间。
`query_latency_us`	查询延迟时间。
`slow_query_latency_us`	慢查询延迟时间。
`num_queries_hit_memory_watermark`	达到内存水位线的语句的数量。
`resp_part_completeness`	部分成功的完整性。需要在 Graph 配置中设置 `accept_partial_success` 为 `true`。

Meta

参数	说明
`commit_log_latency_us`	Raft 协议中 Commit 日志的延迟时间。
`commit_snapshot_latency_us`	Raft 协议中 Commit 快照的延迟时间。
`heartbeat_latency_us`	心跳延迟时间。
`num_heartbeats`	心跳次数。
`num_raft_votes`	Raft 协议中投票的次数。
`transfer_leader_latency_us`	Raft 协议中转移 Leader 的延迟时间。
`num_agent_heartbeats`	AgentHBProcessor 心跳次数。
`agent_heartbeat_latency_us`	AgentHBProcessor 延迟时间。
`replicate_log_latency_us`	Raft 复制日志至大多数节点的延迟。
`num_send_snapshot`	Raft 发送快照至其他节点的次数。
`append_log_latency_us`	Raft 复制日志到单个节点的延迟。
`append_wal_latency_us`	Raft 写入单条 WAL 的延迟。
`num_grant_votes`	Raft 投票给其他节点的次数。
`num_start_elect`	Raft 发起投票的次数。

Storage

参数	说明
`add_edges_latency_us`	添加边的延迟时间。
`add_vertices_latency_us`	添加点的延迟时间。
`commit_log_latency_us`	Raft 协议中 Commit 日志的延迟时间。
`commit_snapshot_latency_us`	Raft 协议中 Commit 快照的延迟时间。
`delete_edges_latency_us`	删除边的延迟时间。
`delete_vertices_latency_us`	删除点的延迟时间。
`get_neighbors_latency_us`	查询邻居延迟时间。
`get_dst_by_src_latency_us`	通过起始点获取终点的延迟时间。
`num_get_prop`	GetPropProcessor 执行的次数。
`num_get_neighbors_errors`	GetNeighborsProcessor 执行出错的次数。
`num_get_dst_by_src_errors`	GetDstBySrcProcessor 执行出错的次数。
`get_prop_latency_us`	GetPropProcessor 执行的延迟时间。
`num_edges_deleted`	删除的边数量。
`num_edges_inserted`	插入的边数量。
`num_raft_votes`	Raft 协议中投票的次数。
`num_rpc_sent_to_metad_failed`	Storaged 服务发给 Metad 服务的 RPC 请求失败的数量。
`num_rpc_sent_to_metad`	Storaged 服务发给 Metad 服务 RPC 请求数量。
`num_tags_deleted`	删除的 Tag 数量。
`num_vertices_deleted`	删除的点数量。
`num_vertices_inserted`	插入的点数量。
`transfer_leader_latency_us`	Raft 协议中转移 Leader 的延迟时间。
`lookup_latency_us`	LookupProcessor 执行的延迟时间。
`num_lookup_errors`	LookupProcessor 执行时出错的次数。
`num_scan_vertex`	ScanVertexProcessor 执行的次数。
`num_scan_vertex_errors`	ScanVertexProcessor 执行时出错的次数。
`update_edge_latency_us`	UpdateEdgeProcessor 执行的延迟时间。
`num_update_vertex`	UpdateVertexProcessor 执行的次数。
`num_update_vertex_errors`	UpdateVertexProcessor 执行时出错的次数。
`kv_get_latency_us`	Getprocessor 的延迟时间。
`kv_put_latency_us`	PutProcessor 的延迟时间。
`kv_remove_latency_us`	RemoveProcessor 的延迟时间。
`num_kv_get_errors`	GetProcessor 执行出错次数。
`num_kv_get`	GetProcessor 执行次数。
`num_kv_put_errors`	PutProcessor 执行出错次数。
`num_kv_put`	PutProcessor 执行次数。

参数	说明
`num_kv_remove_errors`	RemoveProcessor 执行出错次数。
`num_kv_remove`	RemoveProcessor 执行次数。
`forward_tranx_latency_us`	传输延迟时间。
`scan_edge_latency_us`	ScanEdgeProcessor 执行的延迟时间。
`num_scan_edge_errors`	ScanEdgeProcessor 执行时出错的次数。
`num_scan_edge`	ScanEdgeProcessor 执行的次数。
`scan_vertex_latency_us`	ScanVertexProcessor 执行的延迟时间。
`num_add_edges`	添加边的次数。
`num_add_edges_errors`	添加边时出错的次数。
`num_add_vertices`	添加点的次数。
`num_start_elect`	Raft 发起投票的次数
`num_add_vertices_errors`	添加点时出错的次数。
`num_delete_vertices_errors`	删除点时出错的次数。
`append_log_latency_us`	Raft 复制日志到单个节点的延迟。
`num_grant_votes`	Raft 投票给其他节点的次数。
`replicate_log_latency_us`	Raft 复制日志到大多数节点的延迟。
`num_delete_tags`	删除 Tag 的次数。
`num_delete_tags_errors`	删除 Tag 时出错的次数。
`num_delete_edges`	删除边的次数。
`num_delete_edges_errors`	删除边时出错的次数。
`num_send_snapshot`	发送快照的次数。
`update_vertex_latency_us`	UpdateVertexProcessor 执行的延迟时间。
`append_wal_latency_us`	Raft 写入单条 WAL 的延迟。
`num_update_edge`	UpdateEdgeProcessor 执行的次数。
`delete_tags_latency_us`	删除 Tag 的延迟时间。
`num_update_edge_errors`	UpdateEdgeProcessor 执行时出错的次数。
`num_get_neighbors`	GetNeighborsProcessor 执行的次数。
`num_get_dst_by_src`	GetDstBySrcProcessor 执行的次数。
`num_get_prop_errors`	GetPropProcessor 执行时出错的次数。
`num_delete_vertices`	删除点的次数。
`num_lookup`	LookupProcessor 执行的次数。
`num_sync_data`	Storage 同步 Drainer 数据的次数。
`num_sync_data_errors`	Storage 同步 Drainer 数据出错的次数。
`sync_data_latency_us`	Storage 同步 Drainer 数据的延迟时间。

图空间级别监控指标

> Note
>
> 图空间级别监控指标是动态创建的，只有当图空间内触发该行为时，对应的指标才会创建，用户才能查询到。

参数	说明
`num_active_queries`	当前正在执行的查询数。
`num_queries`	查询次数。
`num_sentences`	Graphd 服务接收的语句数。
`optimizer_latency_us`	优化器阶段延迟时间。
`query_latency_us`	查询延迟时间。
`num_slow_queries`	慢查询次数。
`num_query_errors`	查询报错语句数量。
`num_query_errors_leader_changes`	因 Leader 变更而导致的查询错误的次数。
`num_killed_queries`	被终止的查询数量。
`num_aggregate_executors`	聚合（Aggregate）算子执行时间。
`num_sort_executors`	排序（Sort）算子执行次数。
`num_indexscan_executors`	索引扫描（IndexScan）算子执行时间。
`num_auth_failed_sessions_bad_username_password`	因用户名密码错误导验证失败的会话数量。
`num_auth_failed_sessions`	登录验证失败的会话数量。
`num_opened_sessions`	服务端建立过的会话数量。
`num_queries_hit_memory_watermark`	达到内存水位线的语句的数量。
`num_reclaimed_expired_sessions`	服务端主动回收的过期的会话数量。
`num_rpc_sent_to_metad_failed`	Graphd 服务发给 Metad 的 RPC 请求失败的数量。
`num_rpc_sent_to_metad`	Graphd 服务发给 Metad 服务的 RPC 请求数量。
`num_rpc_sent_to_storaged_failed`	Graphd 服务发给 Storaged 服务的 RPC 请求失败的数量。
`num_rpc_sent_to_storaged`	Graphd 服务发给 Storaged 服务的 RPC 请求数量。
`slow_query_latency_us`	慢查询延迟时间。

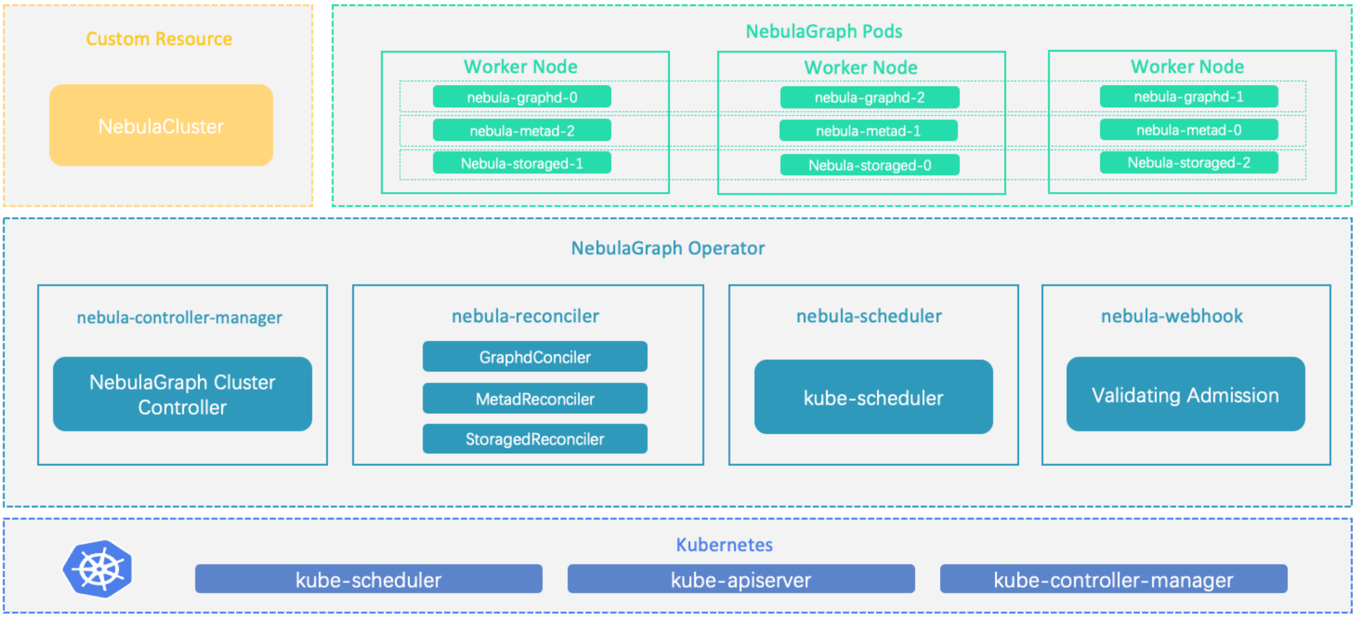
最后更新: 2025年8月4日

17. NebulaGraph Operator

17.1 什么是 NebulaGraph Operator

17.1.1 基本概念

NebulaGraph Operator 是用于在 Kubernetes 系统上自动化部署和运维 NebulaGraph 集群的工具。依托于 Kubernetes 扩展机制，NebulaGraph 将其运维领域的知识全面注入至 Kubernetes 系统中，让 NebulaGraph 成为真正的云原生图数据库。



17.1.2 工作原理

对于 Kubernetes 系统内不存在的资源类型，用户可以通过添加自定义 API 对象的方式注册，常见的方法是使用 CustomResourceDefinition（CRD）。

NebulaGraph Operator 将 NebulaGraph 集群的部署管理抽象为 CRD。通过结合多个内置的 API 对象，包括 StatefulSet、Service 和 ConfigMap，NebulaGraph 集群的日常管理和维护被编码为一个控制循环。在 Kubernetes 系统内，每一种内置资源对象，都运行着一个特定的控制循环，将它的实际状态通过事先规定好的编排动作，逐步调整为最终的期望状态。当一个 CR 实例被提交时，NebulaGraph Operator 会根据控制流程驱动数据库集群进入最终状态。

17.1.3 功能介绍

NebulaGraph Operator 已具备的功能如下：

- 集群创建和卸载：NebulaGraph Operator 简化了用户部署和卸载集群的过程。用户只需提供对应的 CR 文件，NebulaGraph Operator 即可快速创建或者删除一个对应的 NebulaGraph 集群。更多信息参见创建 NebulaGraph 集群。
- 集群升级：支持升级 3.5.0 版的 NebulaGraph 集群至 3.6.0 版。
- 故障自愈：NebulaGraph Operator 调用 NebulaGraph 集群提供的接口，动态地感知服务状态。一旦发现异常，NebulaGraph Operator 自动进行容错处理。更多信息参考故障自愈。
- 均衡调度：基于调度器扩展接口，NebulaGraph Operator 提供的调度器可以将应用 Pods 均匀地分布在 NebulaGraph 集群中。

17.1.4 使用限制

版本限制

NebulaGraph Operator 不支持 v1.x 版本的 NebulaGraph，其与 NebulaGraph 版本的对应关系如下：

NebulaGraph 版本	NebulaGraph Operator 版本
3.5.x ~ 3.6.0	1.5.0 ~ 1.8.x
3.0.0 ~ 3.4.1	1.3.0、1.4.0 ~ 1.4.2
3.0.0 ~ 3.3.x	1.0.0、1.1.0、1.2.0
2.5.x ~ 2.6.x	0.9.0
2.5.x	0.8.0

历史版本兼容性

- 1.x 版本的 NebulaGraph Operator 不兼容 3.x 以下版本的 NebulaGraph.
- 由于 0.9.0 版本的 NebulaGraph Operator 的日志盘和数据盘分开存储，因此用 0.9.0 版的 NebulaGraph Operator 管理通过 0.8.0 版本创建的 2.5.x 版本的 NebulaGraph 集群会导致兼容性问题。用户可以备份 2.5.x 版本的 NebulaGraph 集群，然后使用 0.9.0 版本的 Operator 创建 2.6.x 版本集群。

17.1.5 更新说明

Release

最后更新: 2025年8月4日

17.2 快速入门

17.2.1 安装 NebulaGraph Operator

用户可使用 Helm 工具部署 NebulaGraph Operator。

背景信息

NebulaGraph Operator 为用户管理 NebulaGraph 集群，使用户无需在生产环境中手动安装、扩展、升级和卸载 NebulaGraph，减轻用户管理不同应用版本的负担。

前提条件

安装 NebulaGraph Operator 前，用户需要安装以下软件并确保安装版本的正确性。

软件	版本要求
Kubernetes	>= 1.18
Helm	>= 3.2.0
CoreDNS	>= 1.6.0

Note

- 如果使用基于角色的访问控制的策略，用户需开启 RBAC（可选）。
- CoreDNS 是一个灵活的、可扩展的 DNS 服务器，被安装在集群内作为集群内 Pods 的 DNS 服务器。 NebulaGraph 集群中的每个组件通过 DNS 解析类似 `x.default.svc.cluster.local` 这样的域名相互通信。

操作步骤

1. 添加 NebulaGraph Operator Helm 仓库。

```
helm repo add nebula-operator https://vesoft-inc.github.io/nebula-operator/charts
```

2. 拉取最新的 Operator Helm 仓库。

```
helm repo update
```

参考 Helm 仓库获取更多 `helm repo` 相关信息。

3. 创建命名空间用于安装 NebulaGraph Operator。

```
kubectl create namespace <namespace_name>
```

例如，创建 `nebula-operator-system` 命名空间。

```
kubectl create namespace nebula-operator-system
```

nebula-operator chart 中的所有资源都会安装在该命名空间下。

4. 安装 NebulaGraph Operator。

```
helm install nebula-operator nebula-operator/nebula-operator --namespace=<namespace_name> --version=${chart_version}
```

例如，安装1.8.0版的 Operator 命令如下。

```
helm install nebula-operator nebula-operator/nebula-operator --namespace=nebula-operator-system --version=1.8.0
```

`1.8.0` 为 nebula-operator chart 的版本，不指定 `--version` 时默认使用最新版的 chart。

执行 `helm search repo -l nebula-operator` 查看 chart 版本。

您可在执行安装 NebulaGraph Operator chart 命令时自定义 Operator 的配置。更多信息，查看自定义 Operator 配置。

5. 查看默认创建的 CRD 信息。

```
kubectl get crd
```

返回示例：

```
NAME                        CREATED AT
nebulaautoscalers.autoscaling.nebula-graph.io    2023-11-01T04:16:51Z
nebulaclusters.apps.nebula-graph.io         2023-10-12T07:55:32Z
nebularestores.apps.nebula-graph.io         2023-02-04T23:01:00Z
```

后续操作

创建集群

最后更新: 2025年8月4日

17.2.2 快速创建集群

本文将介绍创建 NebulaGraph 集群的两种方式：

- 使用 Helm 创建 NebulaGraph 集群
- 使用 Kubectl 创建 NebulaGraph 集群

前提条件

- 安装 NebulaGraph Operator
- 已创建 StorageClass

使用 **Helm** 创建 **NebulaGraph** 集群

> 历史版本兼容性
>
> 1.x 版本的 NebulaGraph Operator 不兼容 3.x 以下版本的 NebulaGraph 。

1. 添加 NebulaGraph Operator Helm 仓库。

```
helm repo add nebula-operator https://vesoft-inc.github.io/nebula-operator/charts
```

2. 更新 Helm 仓库，拉取最新仓库资源。

```
helm repo update
```

3. 为安装集群所需的配置参数设置环境变量。

```
export NEBULA_CLUSTER_NAME=nebula       # NebulaGraph 集群的名字。
export NEBULA_CLUSTER_NAMESPACE=nebula   # NebulaGraph 集群所处的命名空间的名字。
export STORAGE_CLASS_NAME=fast-disks        # NebulaGraph 集群的 StorageClass。
```

4. 为 NebulaGraph 集群创建命名空间（如已创建，略过此步）。

```
kubectl create namespace "${NEBULA_CLUSTER_NAMESPACE}"
```

5. 创建 NebulaGraph 集群。

```
helm install "${NEBULA_CLUSTER_NAME}" nebula-operator/nebula-cluster \
  --set nameOverride="${NEBULA_CLUSTER_NAME}" \
  --set nebula.storageClassName="${STORAGE_CLASS_NAME}" \
  # 指定 NebulaGraph 集群的版本。
  --set nebula.version=vmaster \
  # 指定集群 chart 的版本，不指定则默认安装最新版本 chart。
  # 执行 helm search repo - l nebula-operator/nebula-cluster 命令可查看所有 chart 版本。
  --version=1.8.0 \
  --namespace="${NEBULA_CLUSTER_NAMESPACE}" \
```

使用 **Kubectl** 创建 **NebulaGraph** 集群

> 历史版本兼容性
>
> 1.x 版本的 NebulaGraph Operator 不兼容 3.x 以下版本的 NebulaGraph 。

下面以创建名为 `nebula` 的集群为例，说明如何部署 NebulaGraph 集群。

1. 创建命名空间，例如 `nebula`。如果不指定命名空间，默认使用 `default` 命名空间。

```
kubectl create namespace nebula
```

2. 创建集群配置文件，例如 `nebulacluster.yaml`。

展开查看集群的示例配置

```yaml
apiVersion: apps.nebula-graph.io/v1alpha1
kind: NebulaCluster
metadata:
  name: nebula
  namespace: default
spec:
  topologySpreadConstraints:
  - topologyKey: "kubernetes.io/hostname"
    whenUnsatisfiable: "ScheduleAnyway"
  graphd:
    # Graph 服务的容器镜像。
    image: vesoft/nebula-graphd
    logVolumeClaim:
      resources:
        requests:
          storage: 2Gi
      # 用于存储 Graph 服务的日志的存储类名称。
      storageClassName: local-sc
    replicas: 1
    resources:
      limits:
        cpu: "1"
        memory: 1Gi
      requests:
        cpu: 500m
        memory: 500Mi
    version: vmaster
  imagePullPolicy: Always
  metad:
    # Meta 服务的容器镜像。
    image: vesoft/nebula-metad
    logVolumeClaim:
      resources:
        requests:
          storage: 2Gi
      storageClassName: local-sc
    dataVolumeClaim:
      resources:
        requests:
          storage: 2Gi
      storageClassName: local-sc
    replicas: 1
    resources:
      limits:
        cpu: "1"
        memory: 1Gi
      requests:
        cpu: 500m
        memory: 500Mi
    version: vmaster
  reference:
    name: statefulsets.apps
    version: v1
  schedulerName: default-scheduler
  storaged:
    # Storage 服务的容器镜像。
    image: vesoft/nebula-storaged
    logVolumeClaim:
      resources:
        requests:
          storage: 2Gi
      storageClassName: local-sc
    dataVolumeClaims:
    - resources:
        requests:
          storage: 2Gi
      storageClassName: local-sc
    replicas: 1
    resources:
      limits:
        cpu: "1"
        memory: 1Gi
      requests:
        cpu: 500m
        memory: 500Mi
    version: vmaster
```

关于其它参数的详情，请参考创建 NebulaGraph 集群。

3. 创建 NebulaGraph 集群。

```
kubectl create -f nebulacluster.yaml
```

返回：

```
nebulacluster.apps.nebula-graph.io/nebula created
```

4. 查看 NebulaGraph 集群状态。

```
kubectl get nc nebula
```

返回：

```
NAME   READY  GRAPHD-DESIRED  GRAPHD-READY  METAD-DESIRED  METAD-READY  STORAGED-DESIRED  STORAGED-READY  AGE
nebula  True   1        1       1       1       1        1       86s
```

后续操作

连接集群

最后更新: 2025年8月4日

17.2.3 连接 NebulaGraph 集群

使用 NebulaGraph Operator 创建 NebulaGraph 集群后，用户可在 NebulaGraph 集群内部访问 NebulaGraph，也可在集群外访问 NebulaGraph 。如果用户在集群内部，可以通过访问集群内的虚拟 IP 地址（ClusterIP）访问数据库。而如果用户在集群外部，则需要使用集群节点的公共 IP 地址或者通过部署 Nginx Ingress 控制器来访问 NebulaGraph 。IP 地址类型可以在创建集群的配置文件中，通过 `spec.graphd.service` 指定。关于 Service 的更多信息，参考 Service。

前提条件

已使用 NebulaGraph Operator 创建 NebulaGraph 集群。具体步骤参考快速创建集群。

在 NebulaGraph 集群内连接 NebulaGraph

用户可以创建 `ClusterIP` 类型的 Service，为集群内的其他 Pod 提供访问 NebulaGraph 的入口。通过该 Service 的 IP 和数据库 Graph 服务的端口号（ `9669` ），可连接 NebulaGraph 。更多信息，请参考 ClusterIP。

1. 创建名为 `graphd-clusterip-service.yaml` 的文件。示例内容如下：

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/cluster: nebula
    app.kubernetes.io/component: graphd
    app.kubernetes.io/managed-by: nebula-operator
    app.kubernetes.io/name: nebula-graph
  name: nebula-graphd-svc
  namespace: default
spec:
  ports:
  - name: thrift
    port: 9669
    protocol: TCP
    targetPort: 9669
  - name: http
    port: 19669
    protocol: TCP
    targetPort: 19669
  selector:
    app.kubernetes.io/cluster: nebula
    app.kubernetes.io/component: graphd
    app.kubernetes.io/managed-by: nebula-operator
    app.kubernetes.io/name: nebula-graph
  type: ClusterIP  # 设置 Service 类型为 ClusterIP。
```

- NebulaGraph 默认使用 `9669` 端口为客户端提供服务。 `19669` 为 Graph 服务的 HTTP 端口号。
- `targetPort` 的值为映射至 Pod 的端口，可自定义。

2. 执行以下命令使 Service 服务在集群中生效。

```
kubectl create -f graphd-clusterip-service.yaml
```

3. 查看 Service，命令如下：

```
$ kubectl get service -l app.kubernetes.io/cluster=<nebula>  #<nebula>为变量值，请用实际集群名称替换。
NAME             TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)                       AGE
nebula-graphd-svc     ClusterIP   10.98.213.34   <none>        9669/TCP,19669/TCP,19670/TCP          23h
...
```

4. 使用上述 `<cluster-name>-graphd-svc` Service 的 IP 连接 NebulaGraph：

```
kubectl run -ti --image vesoft/nebula-console:v3.6.0 --restart=Never -- <nebula_console_name> -addr <cluster_ip>  -port <service_port> -u <username> -p <password>
```

示例：

```
kubectl run -ti --image vesoft/nebula-console:v3.6.0 --restart=Never -- nebula-console -addr 10.98.213.34  -port 9669 -u root -p vesoft
```

- `--image`： 为连接 NebulaGraph 的工具 NebulaGraph Console 的镜像。
- `<nebula-console>`： 自定义的 Pod 名称。
- `-addr`： 连接 Graphd 服务的 IP 地址，即 `ClusterIP` 类型的 Service IP 地址。
- `-port`： 连接 Graphd 服务的端口。默认端口为 `9669` 。
- `-u`： NebulaGraph 账号的用户名。未启用身份认证时，可以使用任意已存在的用户名（默认为 `root` ）。
- `-p`： 用户名对应的密码。未启用身份认证时，密码可以填写任意字符。

如果返回以下内容，说明成功连接数据库：

```
If you don't see a command prompt, try pressing enter.

(root@nebula) [(none)]>
```

用户还可以使用完全限定域名（FQDN）连接数据库，域名格式为 `<cluster-name>-graphd.<cluster-namespace>.svc.<CLUSTER_DOMAIN>` ， `CLUSTER_DOMAIN` 的默认值为 `cluster.local` 。

```
kubectl run -ti --image vesoft/nebula-console:v3.6.0 --restart=Never -- <nebula_console_name> -addr <cluster_name>-graphd-svc.default.svc.cluster.local -port <service_port> -u <username> -p <password>
```

- `<service_port>` 为 Graph 服务默认的端口 `9669`。

> Note
>
> 如果在集群配置文件中设置了 `spec.console` 字段，您还可以使用以下命令连接到 NebulaGraph 数据库：
>
> ```
> # 进入 nebula-console Pod。
> kubectl exec -it nebula-console -- /bin/sh
>
> # 连接到数据库。
> nebula-console -addr nebula-graphd-svc.default.svc.cluster.local -port 9669 -u <username> -p <password>
> ```
>
> 有关 `nebula-console` 容器的配置，请参见 nebula-console。

通过 `NodePort` 在 NebulaGraph 集群外部连接 NebulaGraph

用户可创建 `NodePort` 类型的 Service，通过集群任一节点 IP 和暴露的节点端口，从集群外部访问集群内部的服务。用户也可以使用云厂商（例如 Azure、AWS 等）提供的负载均衡服务，设置 Service 的类型为 `LoadBalancer`，通过云厂商提供的负载均衡器的公网 IP 和端口，从集群外部访问集群内部的服务。

`NodePort` 类型的 Service 通过标签选择器 `spec.selector` 将前端的请求转发到带有标签 `app.kubernetes.io/cluster: <cluster-name>`、`app.kubernetes.io/component: graphd` 的 Graphd pod 中。

当根据集群示例模板，其中 `spec.graphd.service.type=NodePort`，创建 NebulaGraph 集群后，NebulaGraph Operator 会自动在同一命名空间下，创建名为 `<cluster-name>-graphd-svc`、类型为 `NodePort` 的 Service。通过任一节点 IP 和暴露的节点端口，可直接连接 NebulaGraph 数据库（参见下文第 4 步）。用户也可以根据自己的需求，创建自定义的 Service。

操作步骤如下：

1. 创建名为 `graphd-nodeport-service.yaml` 的文件。示例内容如下：

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/cluster: nebula
    app.kubernetes.io/component: graphd
    app.kubernetes.io/managed-by: nebula-operator
    app.kubernetes.io/name: nebula-graph
  name: nebula-graphd-svc-nodeport
  namespace: default
spec:
  externalTrafficPolicy: Local
  ports:
  - name: thrift
    port: 9669
    protocol: TCP
    targetPort: 9669
  - name: http
    port: 19669
    protocol: TCP
    targetPort: 19669
  selector:
    app.kubernetes.io/cluster: nebula
    app.kubernetes.io/component: graphd
    app.kubernetes.io/managed-by: nebula-operator
    app.kubernetes.io/name: nebula-graph
  type: NodePort # 设置 Service 类型为 NodePort。
```

- NebulaGraph 默认使用 `9669` 端口为客户端提供服务。 `19669` 为 Graph 服务的 HTTP 端口号。
- `targetPort` 的值为映射至 Pod 的端口，可自定义。

2. 执行以下命令使 Service 服务在集群中生效。

```
kubectl create -f graphd-nodeport-service.yaml
```

3. 查看 Service 中 NebulaGraph 映射至集群节点的端口。

```
kubectl get services -l app.kubernetes.io/cluster=<nebula> #<nebula>为变量值，请用实际集群名称替换。
```

返回：

```
NAME             TYPE    CLUSTER-IP   EXTERNAL-IP  PORT(S)                   AGE
nebula-graphd-svc-nodeport   NodePort  10.107.153.129 <none>    9669:32236/TCP,19669:31674/TCP,19670:31057/TCP  24h
...
```

`NodePort` 类型的 Service 中，映射至集群节点的端口为 `32236` 。

4. 使用节点 IP 和上述映射的节点端口连接 NebulaGraph 。

```
kubectl run -ti --image vesoft/nebula-console:v3.6.0 --restart=Never -- <nebula_console_name> -addr <node_ip> -port <node_port> -u <username> -p <password>
```

示例如下：

```
kubectl run -ti --image vesoft/nebula-console:v3.6.0 --restart=Never -- nebula-console -addr 192.168.8.24 -port 32236 -u root -p vesoft
If you don't see a command prompt, try pressing enter.

(root@nebula) [(none)]>
```

- `--image`：为连接 NebulaGraph 的工具 Console 的镜像。
- `<nebula-console>`：自定义的 Pod 名称。本示例为 `nebula-console` 。
- `-addr`： NebulaGraph 集群中任一节点 IP 地址。本示例为 `192.168.8.24` 。
- `-port`： NebulaGraph 映射至节点的端口。本示例为 `32236` 。
- `-u`： NebulaGraph 账号的用户名。未启用身份认证时，可以使用任意已存在的用户名（默认为 root）。
- `-p`：用户名对应的密码。未启用身份认证时，密码可以填写任意字符。

Note

如果在集群配置文件中设置了 `spec.console` 字段，您还可以使用以下命令连接到 NebulaGraph 数据库：

```
# 进入nebula-console Pod。
kubectl exec -it nebula-console -- /bin/sh

# 连接到数据库。
nebula-console -addr <node_ip> -port <node_port> -u <username> -p <password>
```

有关 `nebula-console` 容器的配置，请参见 nebula-console。

通过 `Ingress` 在 NebulaGraph 集群外部连接 NebulaGraph

当集群中有多个 Pod 时，为每个 Pod 分别提供服务会变得非常困难和繁琐，而使用 Ingress 可以轻松解决这个问题。Ingress 可以将流量路由到集群内部的多个 Pod。

Nginx Ingress 是 Kubernetes 中的一个 Ingress 控制器（Controller），是对 Kubernetes Ingress 资源的一个实现，通过 Watch 机制感知 Kubernetes 集群中的 Ingress 资源。它将这些 Ingress 规则转换为 Nginx 配置并启动一个 Nginx 实例来处理流量。

用户可以通过 HostNetwork 和 DaemonSet 组合的模式使用 Nginx Ingress 从集群外部连接 NebulaGraph 集群。

由于使用 HostNetwork，Nginx Ingress 的 Pods 可能被调度在同一个节点上（当多个 Pod 尝试在同一个节点上监听相同的端口时，将会出现端口冲突）。为了避免这种情况，Nginx Ingress 以 DaemonSet 模式（确保集群中每个节点都运行一个 Pod 副本）部署在这些节点上，需先选择一些节点并为节点打上标签，专门用于部署 Nginx Ingress。

由于 Ingress 不支持 TCP 或 UDP 服务，为此 nginx-ingress-controller 使用 `--tcp-services-configmap` 和 `--udp-services-configmap` 参数指向一个 ConfigMap，该 ConfigMap 中的键指需要使用的外部端口，值指要公开的服务的格式，值的格式为 `<命名空间/服务名称>:<服务端口>`。

例如指向名为 `tcp-services` 的 ConfigMap 的配置如下：

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: tcp-services
  namespace: nginx-ingress
data:
  9769: "default/nebula-graphd-svc:9669"
```

操作步骤如下：

1. 创建名为 `nginx-ingress-daemonset-hostnetwork.yaml` 的文件。

   单击 nginx-ingress-daemonset-hostnetwork.yaml 查看完整的 YAML 示例内容。

   > Note
   >
   > 上述 YAML 中的资源对象均使用 `nginx-ingress` 命名空间。用户可执行 `kubectl create namesapce nginx-ingress` 创建命名空间，或者自定义其他命名空间。

2. 为任一节点（本示例使用的节点名为 `worker2`，IP 为 `192.168.8.160`）打上标签，以运行上述 YAML 文件中名为 `nginx-ingress-controller` 的 DaemonSet。

```
kubectl label node worker2 nginx-ingress=true
```

3. 执行以下命令使 Nginx Ingress 在集群中生效。

```
kubectl create -f nginx-ingress-daemonset-hostnetwork.yaml
```

   返回：

```
configmap/nginx-ingress-controller created
configmap/tcp-services created
serviceaccount/nginx-ingress created
serviceaccount/nginx-ingress-backend created
clusterrole.rbac.authorization.k8s.io/nginx-ingress created
clusterrolebinding.rbac.authorization.k8s.io/nginx-ingress created
role.rbac.authorization.k8s.io/nginx-ingress created
rolebinding.rbac.authorization.k8s.io/nginx-ingress created
service/nginx-ingress-controller-metrics created
service/nginx-ingress-default-backend created
service/nginx-ingress-proxy-tcp created
daemonset.apps/nginx-ingress-controller created
```

   成功部署 Nginx Ingress 后，由于 Nginx Ingress 中配置的网络类型为 `hostNetwork`，因此用户可通过部署了 Nginx Ingress 的节点的 IP（`192.168.8.160`）和外部端口（`9769`）访问 NebulaGraph 服务。

4. 执行以下命令部署连接 NebulaGraph 服务的 Console 并通过宿主机 IP（本示例为 `192.168.8.160`）和上述配置的外部端口访问 NebulaGraph 服务。

```
kubectl run -ti --image vesoft/nebula-console:v3.6.0 --restart=Never -- <nebula_console_name> -addr <host_ip> -port <external_port> -u <username> -p <password>
```

   示例：

```
kubectl run -ti --image vesoft/nebula-console:v3.6.0 --restart=Never -- nebula-console -addr 192.168.8.160 -port 9769 -u root -p vesoft
```

- `--image`：为连接 NebulaGraph 的工具 NebulaGraph Console 的镜像。
- `<nebula-console>`：自定义的 Pod 名称。本示例为 `nebula-console`。
- `-addr`：部署 Nginx Ingress 的节点 IP，本示例为 `192.168.8.160`。
- `-port`：外网访问使用的的端口。本示例设置为 `9769`。
- `-u`： NebulaGraph 账号的用户名。未启用身份认证时，可以使用任意已存在的用户名（默认为 root）。
- `-p`：用户名对应的密码。未启用身份认证时，密码可以填写任意字符。

   如果返回以下内容，说明成功连接数据库：

```
If you don't see a command prompt, try pressing enter.

(root@nebula) [(none)]>
```

> Note
>
> 如果在集群配置文件中设置了 `spec.console` 字段，您还可以使用以下命令连接到 NebulaGraph 数据库：
>
> ```
> # 进入nebula-console Pod。
> kubectl exec -it nebula-console -- /bin/sh
>
> # 连接到数据库。
> nebula-console -addr <ingress_host_ip> -port <external_port> -u <username> -p <password>
> ```
>
> 有关 `nebula-console` 容器的配置，请参见 nebula-console。

最后更新: 2025年8月4日

17.3 管理 NebulaGraph Operator

17.3.1 自定义安装配置

本文将介绍如何在安装 NebulaGraph Operator 时自定义默认配置项。

配置项

在执行 `helm install [NAME] [CHART] [flags]` 命令安装 Chart 时，可指定 Chart 配置。更多信息，请参考安装前自定义 Chart。

用户可以在 nebula-operator chart 配置文件中查看相关的配置选项。

或者执行命令 `helm show values nebula-operator/nebula-operator` 查看可配置的选项，返回结果如下。

```
[root@master ~]$ helm show values nebula-operator/nebula-operator
image:
  nebulaOperator:
    image: vesoft/nebula-operator:v1.8.0
    imagePullPolicy: Always

imagePullSecrets: [ ]
kubernetesClusterDomain: ""

controllerManager:
  create: true
  replicas: 2
  env: [ ]
  resources:
    limits:
      cpu: 200m
      memory: 200Mi
    requests:
      cpu: 100m
      memory: 100Mi
  verbosity: 0
  ## Additional InitContainers to initialize the pod
  # Example:
  #  extraInitContainers:
  #  - name: init-auth-sidecar
  #    command:
  #    - /bin/sh
  #    - -c
  #    args:
  #    - cp -R /certs/* /credentials/
  #    imagePullPolicy: Always
  #    image: reg.vesoft-inc.com/nebula-certs:latest
  #    volumeMounts:
  #    - name: credentials
  #      mountPath: /credentials
  extraInitContainers: []

  # sidecarContainers - add more containers to controller-manager
  # Key/Value where Key is the sidecar `- name: <Key>`
  # Example:
  #   sidecarContainers:
  #     webserver:
  #       image: nginx
  # OR for adding netshoot to controller manager
  #  sidecarContainers:
  #    netshoot:
  #      args:
  #      - -c
  #      - while true; do ping localhost; sleep 60;done
  #      command:
  #      - /bin/bash
  #      image: nicolaka/netshoot
  #      imagePullPolicy: Always
  #      name: netshoot
  #      resources: {}
  sidecarContainers: {}

  ## Additional controller-manager Volumes
  extraVolumes: []

  ## Additional controller-manager Volume mounts
  extraVolumeMounts: []

  securityContext: {}
  #  runAsNonRoot: true

admissionWebhook:
  create: false
```

```yaml
  # The TCP port the Webhook server binds to. (default 9443)
  webhookBindPort: 9443

scheduler:
  create: true
  schedulerName: nebula-scheduler
  replicas: 2
  env: [ ]
  resources:
    limits:
      cpu: 200m
      memory: 200Mi
    requests:
      cpu: 100m
      memory: 100Mi
  verbosity: 0
  plugins:
    enabled: ["NodeZone"]
    disabled: [] # only in-tree plugins need to be defined here
...
```

部分参数描述如下：

参数	默认值	描述
`image.nebulaOperator.image`	`vesoft/nebula-operator:v1.8.0`	NebulaGraph Operator 的镜像，版本为1.8.0。
`image.nebulaOperator.imagePullPolicy`	`IfNotPresent`	镜像拉取策略。
`imagePullSecrets`	`[]`	镜像拉取密钥，例如 `imagePullSecrets[0].name="vesoft"` 。
`kubernetesClusterDomain`	`cluster.local`	集群域名。
`controllerManager.create`	`true`	是否启用 controller-manager。
`controllerManager.replicas`	`2`	controller-manager 副本数。
`controllerManager.env`	`[]`	配置环境变量。
`controllerManager.extraInitContainers`	`[]`	配置初始化容器。
`controllerManager.sidecarContainers`	`{}`	配置 sidecar 容器。
`controllerManager.extraVolumes`	`[]`	配置存储卷。
`controllerManager.extraVolumeMounts`	`[]`	配置存储卷挂载路径。
`controllerManager.securityContext`	`{}`	配置访问和控制 NebulaGraph Operator 的操作。
`admissionWebhook.create`	`false`	是否启用 Admission Webhook。默认关闭，如需开启，需设置为 `true` 并且需要安装 cert-manager。详情参见开启准入控制。
`admissionWebhook.webhookBindPort`	`9443`	Webhook 服务器监听和接受传入请求的端口，默认 9443。
`scheduler.create`	`true`	是否启用 Scheduler。
`scheduler.schedulerName`	`nebula-scheduler`	NebulaGraph Operator 自定义的调度器名称。
`scheduler.replicas`	`2`	nebula-scheduler 副本数。

示例

以下示例为在安装 NebulaGraph Operator 时，指定 NebulaGraph Operator 的 AdmissionWebhook 机制为开启状态（默认关闭 AdmissionWebhook）：

```
helm install nebula-operator nebula-operator/nebula-operator --namespace=<nebula-operator-system> --set admissionWebhook.create=true
```

验证是否开启 AdmissionWebhook：

```
helm get values nebula-operator -n <nebula-operator-system>
```

示例输出：

```
USER-SUPPLIED VALUES:
admissionWebhook:
  create: true
```

关于 `helm install` 命令更多信息，参见 Helm Install。

最后更新: 2025年8月4日

17.3.2 更新 NebulaGraph Operator 配置

本文将介绍如何更新 NebulaGraph Operator 的配置。

操作步骤

1. 拉取最新的 Helm 仓库。

```
helm repo update
```

2. 查看 NebulaGraph Operator 的默认配置。

```
helm show values nebula-operator/nebula-operator
```

3. 通过 `--set` 传递配置参数，更新 NebulaGraph Operator。

- `--set`：通过命令行的方式新增或覆盖指定项。有关可以更新的配置项，请查看自定义配置。

例如，更新 NebulaGraph Operator 的 AdmissionWebhook 机制为开启状态。

```
helm upgrade nebula-operator nebula-operator/nebula-operator --namespace=nebula-operator-system --version=1.8.0 --set admissionWebhook.create=true
```

更多信息，参考 Helm 升级。

4. 查看 NebulaGraph Operator 的配置是否更新成功。

```
helm get values nebula-operator -n nebula-operator-system
```

示例输出：

```
USER-SUPPLIED VALUES:
admissionWebhook:
  create: true
```

最后更新: 2025年8月4日

17.3.3 管理指定NebulaGraph集群

在默认情况下，NebulaGraph Operator 会管理所有的NebulaGraph集群。但是，用户可以指定 NebulaGraph Operator 管理的NebulaGraph集群范围。本文介绍如何指定 NebulaGraph Operator 管理的集群范围。

应用场景

- NebulaGraph Operator 灰度发布：希望让新的 Nebula Operator 版本先在一部分集群上运行，以便可以在全面推出之前测试和验证其性能。
- 管理特定集群：希望 NebulaGraph Operator 只管理特定的NebulaGraph集群。

配置介绍

NebulaGraph Operator 支持通过 controller-manager 的启动参数来指定管理集群的范围。支持的参数如下：

- `watchNamespaces`：用于指定NebulaGraph集群所在的命名空间。多个命名空间之间使用英文逗号分隔。例如，`watchNamespaces=default,nebula`。如果不指定该参数，则 NebulaGraph Operator 会管理所有命名空间中的NebulaGraph集群。
- `nebulaObjectSelector`：允许设置具体的标签和值来选择要管理的NebulaGraph集群。支持 `=`、`==`、`!=` 三种标签运算操作符，其中，`=` 和 `==` 含义相同，表示标签的值等于指定的值；`!=` 表示标签的值不等于指定的值。多个标签之间使用英文逗号分隔且逗号需要使用 `\\` 转义。例如，`nebulaObjectSelector=key1=key1=value1\\,key2=value2`，这将只选择集群标签为 `key1=value1` 和 `key2=value2` 的NebulaGraph对象。如果不指定该参数，则 NebulaGraph Operator 会管理所有的NebulaGraph集群。

示例

通过命名空间指定管理集群范围

以下命令使 NebulaGraph Operator 仅管理 `default` 和 `nebula` 命名空间中的NebulaGraph集群。确保当前拉取的 Helm Chart 版本支持该参数。更多信息，参见更新配置。

```
helm upgrade nebula-operator nebula-operator/nebula-operator --set watchNamespaces=default,nebula
```

通过标签选择器指定管理集群范围

以下命令使 NebulaGraph Operator 仅管理标签为 `key1=value1` 和 `key2=value2` 的NebulaGraph集群。确保当前拉取的 Helm Chart 版本支持该参数。更多信息，参见更新配置。

```
helm upgrade nebula-operator nebula-operator/nebula-operator --set nebulaObjectSelector=key1=value1\\,key2=value2
```

常见问题

如何为NEBULAGRAPH集群设置标签?

执行以下命令为NebulaGraph集群设置标签：

```
kubectl label nc <cluster_name> -n <namespace> <key>=<value>
```

例如，在命名空间 `nebulaspace` 中，为名为 `nebula` 的NebulaGraph集群设置标签 `env=test`：

```
kubectl label nc nebula -n nebulaspace env=test
```

如何查看NEBULAGRAPH集群的标签?

执行以下命令查看NebulaGraph集群的标签：

```
kubectl get nc <cluster_name> -n <namespace> --show-labels
```

例如，在命名空间 `nebulaspace` 中，查看名为 `nebula` 的NebulaGraph集群的标签：

```
kubectl get nc nebula -n nebulaspace --show-labels
```

如何删除NEBULAGRAPH集群的标签?

执行以下命令删除NebulaGraph集群的标签：

```
kubectl label nc <cluster_name> -n <namespace> <key>-
```

例如，在命名空间 `nebulaspace` 中，删除名为 `nebula` 的NebulaGraph集群的标签 `env=test`：

```
kubectl label nc nebula -n nebulaspace env-
```

如何查看NEBULAGRAPH集群所在的命名空间?

执行以下命令列出所有NebulaGraph集群所在的命名空间：

```
kubectl get nc --all-namespaces
```

最后更新: 2025年8月4日

17.3.4 升级 NebulaGraph Operator

本文将介绍如何升级 NebulaGraph Operator 版本。

历史版本兼容性

- 不支持升级 0.9.0 及以下版本的 NebulaGraph Operator 至 1.x 版本。
- 1.x 版本的 NebulaGraph Operator 不兼容 3.x 以下版本的 NebulaGraph。

操作步骤

1. 查看当前 NebulaGraph Operator 的版本。

```
helm list --all-namespaces
```

返回示例：

```
NAME            NAMESPACE               REVISION    UPDATED                                 STATUS      CHART                   APP VERSION
nebula-operator nebula-operator-system  3           2023-11-06 12:06:24.742397418 +0800 CST deployed    nebula-operator-1.7.0   1.7.0
```

2. 拉取最新的 Helm 仓库。

```
helm repo update
```

3. 查看最新的 NebulaGraph Operator 版本。

```
helm search repo nebula-operator/nebula-operator
```

返回示例：

```
NAME                            CHART VERSION   APP VERSION DESCRIPTION
nebula-operator/nebula-operator 1.8.0           1.8.0       Nebula Operator Helm chart for Kubernetes
```

4. 升级 NebulaGraph Operator 至 1.8.0 版本。

```
helm upgrade nebula-operator nebula-operator/nebula-operator --namespace=<namespace_name> --version=1.8.0
```

示例：

```
helm upgrade nebula-operator nebula-operator/nebula-operator --namespace=nebula-operator-system --version=1.8.0
```

输出：

```
Release "nebula-operator" has been upgraded. Happy Helming!
NAME: nebula-operator
LAST DEPLOYED: Tue Nov 16 02:21:08 2021
NAMESPACE: nebula-operator-system
STATUS: deployed
REVISION: 3
TEST SUITE: None
NOTES:
NebulaGraph Operator installed!
```

5. 拉取最新的 CRD 配置文件。

> Note
>
> 升级 Operator 后，需要同时升级相应的 CRD 配置，否则 NebulaGraph 集群创建会失败。有关 CRD 的配置，参见 apps.nebula-graph.io_nebulaclusters.yaml。

a. 下载 NebulaGraph Operator chart 至本地。

```
helm pull nebula-operator/nebula-operator --version=1.8.0
```

- `--version`：升级版本号。如不指定，则默认为最新版本。

b. 执行 `tar -zxvf` 解压安装包。

例如：解压 1.8.0 chart 包至 `/tmp` 路径下。

```
tar -zxvf nebula-operator-1.8.0.tgz -C /tmp
```

- `-C /tmp`：如不指定，则默认解压至当前路径。

6. 在 `nebula-operator` 目录下应用最新的 CRD 配置文件。

```
kubectl apply -f crds/nebulaclusters.yaml
```

输出：

```
customresourcedefinition.apiextensions.k8s.io/nebulaclusters.apps.nebula-graph.io configured
```

最后更新: 2025年8月4日

17.3.5 卸载 NebulaGraph Operator

本文将介绍如何卸载 NebulaGraph Operator。

操作步骤

1. 卸载 NebulaGraph Operator chart。

```
helm uninstall nebula-operator --namespace=<nebula-operator-system>
```

2. 查看默认创建的 CRD 信息。

```
kubectl get crd
```

返回示例：

```
NAME                                        CREATED AT
nebulaautoscalers.autoscaling.nebula-graph.io    2023-11-01T04:16:51Z
nebulaclusters.apps.nebula-graph.io         2023-10-12T07:55:32Z
nebularestores.apps.nebula-graph.io         2023-02-04T23:01:00Z
```

3. 删除 CRD。

```
kubectl delete crd nebulaclusters.apps.nebula-graph.io nebularestores.apps.nebula-graph.io nebulaautoscalers.autoscaling.nebula-graph.io
```

最后更新: 2025年8月4日

17.4 管理集群

17.4.1 部署

使用 **NebulaGraph Operator** 安装 **NebulaGraph** 集群

采用 NebulaGraph Operator 安装 NebulaGraph 集群，能够实现集群管理的自动化，同时具备自动错误恢复的功能。本文介绍 `kubectl apply` 和 `helm` 两种方式来使用 NebulaGraph Operator 安装 NebulaGraph 集群。

> 历史版本兼容性
>
> 1.x 版本的 NebulaGraph Operator 不兼容 3.x 以下版本的 NebulaGraph 。

前提条件

- 安装 NebulaGraph Operator
- 已创建 StorageClass

使用 `KUBECTL APPLY`

1. 创建命名空间，用于存放 NebulaGraph 集群相关资源。例如，创建 `nebula` 命名空间。

```
kubectl create namespace nebula
```

2. 创建集群的 YAML 配置文件 `nebulacluster.yaml` 。例如，创建名为 `nebula` 的集群。

展开查看 `nebula` 集群的示例配置

```yaml
apiVersion: apps.nebula-graph.io/v1alpha1
kind: NebulaCluster
metadata:
  name: nebula
  namespace: default
spec:
  # 控制 Pod 的调度策略。
  topologySpreadConstraints:
  - topologyKey: "kubernetes.io/hostname"
    whenUnsatisfiable: "ScheduleAnyway"
  # 是否回收 PV。
  enablePVReclaim: false
  # 是否启用备份和恢复功能。
  exporter:
    image: vesoft/nebula-stats-exporter
    version: v3.3.0
    replicas: 1
    maxRequests: 20
  # 自定义 Agent 镜像，用于集群备份和恢复及日志清理。
  agent:
    image: vesoft/nebula-agent
    version: latest
    resources:
      requests:
        cpu: "100m"
        memory: "128Mi"
      limits:
        cpu: "200m"
        memory: "256Mi"
  # 配置镜像拉取策略。
  imagePullPolicy: Always
  # 选择 Pod 被调度的节点。
  nodeSelector:
    nebula: cloud
  # 依赖的控制器名称。
  reference:
    name: statefulsets.apps
    version: v1
  # 调度器名称。
  schedulerName: default-scheduler
  # 启动 NebulaGraph Console 服务，用于连接 Graph 服务。
  console:
    image: vesoft/nebula-console
    version: nightly
    username: "demo"
    password: "test"
  # Graph 服务的相关配置。
  graphd:
  # 用于检测 Graph 服务是否正常运行。
  # readinessProbe:
  #   failureThreshold: 3
  #   httpGet:
  #     path: /status
  #     port: 19669
  #     scheme: HTTP
  #   initialDelaySeconds: 40
  #   periodSeconds: 10
  #   successThreshold: 1
  #   timeoutSeconds: 10
    # Graph 服务的容器镜像。
    image: vesoft/nebula-graphd
    logVolumeClaim:
      resources:
        requests:
          storage: 2Gi
      # 用于存储 Graph 服务的日志的存储类名称。
      storageClassName: local-sc
    # Graph 服务的 Pod 副本数。
    replicas: 1
    # Graph 服务的资源配置。
    resources:
      limits:
        cpu: "1"
        memory: 1Gi
      requests:
        cpu: 500m
        memory: 500Mi
    # Graph 服务的版本。
    version: vmaster
    # 自定义 Graph 服务的 flags 配置项。
    config: {}
  # Meta 服务的相关配置。
  metad:
  # readinessProbe:
  #   failureThreshold: 3
  #   httpGet:
  #     path: /status
  #     port: 19559
  #     scheme: HTTP
  #   initialDelaySeconds: 5
  #   periodSeconds: 5
  #   successThreshold: 1
  #  timeoutSeconds: 5
    # Meta 服务的容器镜像。
    image: vesoft/nebula-metad
```

```yaml
    logVolumeClaim:
      resources:
        requests:
          storage: 2Gi
      storageClassName: local-sc
    dataVolumeClaim:
      resources:
        requests:
          storage: 2Gi
      storageClassName: local-sc
    replicas: 1
    resources:
      limits:
        cpu: "1"
        memory: 1Gi
      requests:
        cpu: 500m
        memory: 500Mi
    version: vmaster
    # 自定义 Meta 服务的 flags 配置项。
    config: {}
  # Storage 服务的相关配置。
  storaged:
  # readinessProbe:
  #   failureThreshold: 3
  #   httpGet:
  #     path: /status
  #     port: 19779
  #     scheme: HTTP
  #   initialDelaySeconds: 40
  #   periodSeconds: 10
  #   successThreshold: 1
  #   timeoutSeconds: 5
    # Storage 服务的容器镜像。
    image: vesoft/nebula-storaged
    logVolumeClaim:
      resources:
        requests:
          storage: 2Gi
      storageClassName: local-sc
    dataVolumeClaims:
    - resources:
        requests:
          storage: 2Gi
      storageClassName: local-sc
    replicas: 1
    resources:
      limits:
        cpu: "1"
        memory: 1Gi
      requests:
        cpu: 500m
        memory: 500Mi
    version: vmaster
    # 自定义 Storage 服务的 flags 配置项。
    config: {}
```

展开查看集群所有可配置的参数及描述

参数	默认值	描述
`metadata.name`	-	创建的NebulaGraph集群名称。
`spec.console`	-	启动 Console 容器用于连接 Graph 服务。配置详情，参见 nebula-console.
`spec.topologySpreadConstraints`	-	控制 Pod 的调度策略。详情参见 Topology Spread Constraints。当 `topologyKey` 的值为 `kubernetes.io/zone` 时，`whenUnsatisfiable` 的值需为 `DoNotSchedule` 并且 `spec.schedulerName` 的值为 `nebula-scheduler` 。
`spec.graphd.replicas`	`1`	Graphd 服务的副本数。
`spec.graphd.image`	`vesoft/nebula-graphd`	Graphd 服务的容器镜像。
`spec.graphd.version`	`master`	Graphd 服务的版本号。
`spec.graphd.service`		访问 Graphd 服务的 Service 配置。
`spec.graphd.logVolumeClaim.storageClassName`	-	Graphd 服务的日志盘存储卷的存储类名称。使用示例配置时需要将其替换为事先创建的存储类名称，参见 Storage Classes 查看创建存储类详情。
`spec.metad.replicas`	`1`	Metad 服务的副本数。
`spec.metad.image`	`vesoft/nebula-metad`	Metad 服务的容器镜像。
`spec.metad.version`	`master`	Metad 服务的版本号。
`spec.metad.dataVolumeClaim.storageClassName`	-	Metad 服务的数据盘存储配置。使用示例配置时需要将其替换为事先创建的存储类名称，参见 Storage Classes 查看创建存储类详情。
`spec.metad.logVolumeClaim.storageClassName`	-	Metad 服务的日志盘存储配置。使用示例配置时需要将其替换为事先创建的存储类名称，参见 Storage Classes 查看创建存储类详情。
`spec.storaged.replicas`	`3`	Storaged 服务的副本数。
`spec.storaged.image`	`vesoft/nebula-storaged`	Storaged 服务的容器镜像。
`spec.storaged.version`	`master`	Storaged 服务的版本号。
`spec.storaged.dataVolumeClaims.resources.requests.storage`	-	Storaged 服务的数据盘存储大小，可指定多块数据盘存储数据。当指定多块数据盘时，路径为：`/usr/local/nebula/data1`、`/usr/local/nebula/data2` 等。
`spec.storaged.dataVolumeClaims.storageClassName`	-	Storaged 服务的数据盘存储配置。使用示例配置时需要将其替换为事先创建的存储类名称，参见 Storage Classes 查看创建存储类详情。
`spec.storaged.logVolumeClaim.storageClassName`	-	Storaged 服务的日志盘存储配置。使用示例配置时需要将其替换为事先创建的存储类名称，参见 Storage Classes 查看创建存储类详情。
`spec.<metad|storaged|graphd>.securityContext`	`{}`	定义集群容器的权限和访问控制，以控制访问和执行容器的操作。详情参见 SecurityContext。
`spec.agent`	`{}`	Agent 服务的配置。用于备份和恢复及日志清理功能，如果不自定义该配置，将使用默认配置。

参数	默认值	描述
`spec.reference.name`	`{}`	依赖的控制器名称。
`spec.schedulerName`	`default-scheduler`	调度器名称。
`spec.imagePullPolicy`	`Always`	NebulaGraph镜像的拉取策略。关于拉取策略详情，请参考 Image pull policy。
`spec.logRotate`	`{}`	日志轮转配置。详情参见管理集群日志。
`spec.enablePVReclaim`	`false`	定义是否在删除集群后自动删除 PVC 以释放数据。详情参见回收 PV。
`spec.metad.licenseManagerURL`	-	配置指向 LM 的 URL，由 LM 的访问地址和端口（默认端口 `9119`）组成。例如，`192.168.8.xxx:9119`。仅适用于创建企业版 NebulaGraph集群。
`spec.storaged.enableAutoBalance`	`false`	是否启用自动均衡。详情参见均衡扩容后的 Storage 数据。
`spec.enableBR`	`false`	定义是否启用 BR 工具。详情参见备份与恢复。
`spec.imagePullSecrets`	`[]`	定义拉取私有仓库中镜像所需的 Secret。

3. 创建 NebulaGraph 集群。

```
kubectl create -f apps_v1alpha1_nebulacluster.yaml -n nebula
```

返回：

```
nebulacluster.apps.nebula-graph.io/nebula created
```

如果不通过 `-n` 指定命名空间，默认使用 `default` 命名空间。

4. 查看 NebulaGraph 集群状态。

```
kubectl get nebulaclusters nebula -n nebula
```

返回：

```
NAME   READY  GRAPHD-DESIRED  GRAPHD-READY  METAD-DESIRED  METAD-READY  STORAGED-DESIRED  STORAGED-READY  AGE
nebula  True  1        1        1       1       1        1        86s
```

使用 **HELM**

1. 添加 NebulaGraph Operator Helm 仓库（如果已添加，执行下步骤）。

```
helm repo add nebula-operator https://vesoft-inc.github.io/nebula-operator/charts
```

2. 更新 Helm 仓库，拉取最新仓库资源。

```
helm repo update nebula-operator
```

3. 为安装集群所需的配置参数设置环境变量。

```
export NEBULA_CLUSTER_NAME=nebula       # NebulaGraph 集群的名字。
export NEBULA_CLUSTER_NAMESPACE=nebula   # NebulaGraph 集群所处的命名空间的名字。
export STORAGE_CLASS_NAME=local-sc       # NebulaGraph 集群的 StorageClass。
```

4. 为 NebulaGraph 集群创建命名空间（如已创建，略过此步）。

```
kubectl create namespace "${NEBULA_CLUSTER_NAMESPACE}"
```

5. 查看创建集群时，`nebula-operator` 的 `nebula-cluster` chart 的可自定义的配置项。

- 单击 nebula-cluster/values.yaml 查看NebulaGraph集群的所有可配置参数。

- 执行以下命令查看所有可以配置的参数。

```
helm show values nebula-operator/nebula-cluster
```

展开查看返回结果

```yaml
nebula:
 version: master
 imagePullPolicy: Always
 storageClassName: ""
 enablePVReclaim: false
 enableBR: false
 enableForceUpdate: false
 schedulerName: default-scheduler
 topologySpreadConstraints:
 - topologyKey: "kubernetes.io/hostname"
  whenUnsatisfiable: "ScheduleAnyway"
 logRotate: {}
 reference:
  name: statefulsets.apps
  version: v1
 graphd:
  image: vesoft/nebula-graphd
  replicas: 2
  serviceType: NodePort
  env: []
  config: {}
  resources:
   requests:
    cpu: "500m"
    memory: "500Mi"
   limits:
    cpu: "1"
    memory: "500Mi"
  logVolume:
   enable: true
   storage: "500Mi"
  podLabels: {}
  podAnnotations: {}
  securityContext: {}
  nodeSelector: {}
  tolerations: []
  affinity: {}
  readinessProbe: {}
  livenessProbe: {}
  initContainers: []
  sidecarContainers: []
  volumes: []
  volumeMounts: []

 metad:
  image: vesoft/nebula-metad
  replicas: 3
  env: []
  config: {}
  resources:
   requests:
    cpu: "500m"
    memory: "500Mi"
   limits:
    cpu: "1"
    memory: "1Gi"
  logVolume:
   enable: true
   storage: "500Mi"
  dataVolume:
   storage: "2Gi"
  licenseManagerURL: ""
  license: {}
  podLabels: {}
  podAnnotations: {}
  securityContext: {}
  nodeSelector: {}
  tolerations: []
  affinity: {}
  readinessProbe: {}
  livenessProbe: {}
  initContainers: []
  sidecarContainers: []
  volumes: []
  volumeMounts: []

 storaged:
  image: vesoft/nebula-storaged
  replicas: 3
  env: []
  config: {}
  resources:
   requests:
    cpu: "500m"
    memory: "500Mi"
   limits:
    cpu: "1"
    memory: "1Gi"
  logVolume:
   enable: true
   storage: "500Mi"
  dataVolumes:
  - storage: "10Gi"
  enableAutoBalance: false
  podLabels: {}
```

```yaml
    podAnnotations: {}
    securityContext: {}
    nodeSelector: {}
    tolerations: []
    affinity: {}
    readinessProbe: {}
    livenessProbe: {}
    initContainers: []
    sidecarContainers: []
    volumes: []
    volumeMounts: []

  exporter:
    image: vesoft/nebula-stats-exporter
    version: v3.3.0
    replicas: 1
    env: []
    resources:
      requests:
        cpu: "100m"
        memory: "128Mi"
      limits:
        cpu: "200m"
        memory: "256Mi"
    podLabels: {}
    podAnnotations: {}
    securityContext: {}
    nodeSelector: {}
    tolerations: []
    affinity: {}
    readinessProbe: {}
    livenessProbe: {}
    initContainers: []
    sidecarContainers: []
    volumes: []
    volumeMounts: []
    maxRequests: 20

  agent:
    image: vesoft/nebula-agent
    version: latest
    resources:
      requests:
        cpu: "100m"
        memory: "128Mi"
      limits:
        cpu: "200m"
        memory: "256Mi"

  console:
    username: root
    password: nebula
    image: vesoft/nebula-console
    version: latest
    nodeSelector: {}

  alpineImage: ""

imagePullSecrets: []
nameOverride: ""
fullnameOverride: ""
```

展开查看可配置的参数描述

字段	默认值	描述
`nebula.version`	master	集群的版本。
`nebula.imagePullPolicy`	`Always`	容器镜像拉取策略。`Always` 表示总是尝试从远程拉取最新的镜像。
`nebula.storageClassName`	`""`	Kubernetes 存储类的名称，用于动态分配持久卷。
`nebula.enablePVReclaim`	`false`	是否启用持久卷回收功能。详情参见回收 PV。
`nebula.enableBR`	`false`	是否启用备份和恢复功能。详情参见使用 NebulaGraph Operator 备份和恢复数据。
`nebula.enableForceUpdate`	`false`	是否不迁移分片 Leader 副本而强制更新 Storage 服务。详情参见优化滚动更新中的 Leader 分布。
`nebula.schedulerName`	`default-scheduler`	Kubernetes 调度器的名称。使用 Zone 功能时，必须配置为 `nebula-scheduler`。
`nebula.topologySpreadConstraints`	`[]`	用于控制 Pod 在集群中的分布。
`nebula.logRotate`	`{}`	日志轮替配置。详情参见管理集群日志。
`nebula.reference`	`{"name": "statefulsets.apps", "version": "v1"}`	被NebulaGraph引用的工作负载。
`nebula.graphd.image`	`vesoft/nebula-graphd`	Graph 服务的容器镜像。
`nebula.graphd.replicas`	`2`	Graph 服务的副本数量。
`nebula.graphd.serviceType`	`NodePort`	Graph 服务的 Service 类型。用来定义访问 Graph 服务的方式。详情参见连接集群。
`nebula.graphd.env`	`[]`	Graph 服务的容器环境变量。
`nebula.graphd.config`	`{}`	Graph 服务的配置。详情参见更新集群的配置。
`nebula.graphd.resources`	`{"resources":{"requests":{"cpu":"500m","memory":"500Mi"},"limits":{"cpu":"1","memory":"500Mi"}}}`	Graph 服务的资源限制和请求。
`nebula.graphd.logVolume`	`{"logVolume": {"enable": true,"storage": "500Mi"}}`	Graph 服务的日志存储配置。当 `enable` 为 `false` 时，不启用日志卷。
`nebula.metad.image`	`vesoft/nebula-metad`	Meta 服务的容器镜像。
`nebula.metad.replicas`	`3`	Meta 服务的副本数量。
`nebula.metad.env`	`[]`	Meta 服务的容器环境变量。
`nebula.metad.config`	`{}`	Meta 服务的配置。详情参见更新集群的配置。
`nebula.metad.resources`	`{"resources":{"requests":{"cpu":"500m","memory":"500Mi"},"limits":{"cpu":"1","memory":"1Gi"}}}`	Meta 服务的资源限制和请求。
`nebula.metad.logVolume`	`{"logVolume": {"enable": true,"storage": "500Mi"}}`	Meta 服务的日志存储配置。当 `enable` 为 `false` 时，不启用日志卷。

字段	默认值	描述
`nebula.metad.dataVolume`	`{"dataVolume": {"storage": "2Gi"}}`	Meta 服务的数据存储配置。
`nebula.metad.licenseManagerURL`	`""`	LM 访问地址和端口号，用于获取 License 信息。仅适用于创建企业版 NebulaGraph集群。
`nebula.storaged.image`	`vesoft/nebula-storaged`	Storage 服务的容器镜像。
`nebula.storaged.replicas`	`3`	Storage 服务的副本数量。
`nebula.storaged.env`	`[]`	Storage 服务的容器环境变量。
`nebula.storaged.config`	`{}`	Storage 服务的配置。详情参见更新集群的配置。
`nebula.storaged.resources`	`{"resources":{"requests":{"cpu":"500m","memory":"500Mi"},"limits":{"cpu":"1","memory":"1Gi"}}}`	Storage 服务的资源限制和请求。
`nebula.storaged.logVolume`	`{"logVolume": {"enable": true,"storage": "500Mi"}}`	Storage 服务的日志存储配置。当 `enable` 为 `false` 时，不启用日志卷。
`nebula.storaged.dataVolumes`	`{"dataVolumes": [{"storage": "10Gi"}]}`	Storage 服务的数据存储配置。支持指定多个数据盘。
`nebula.storaged.enableAutoBalance`	`false`	是否启用自动均衡。详情参见均衡扩容后的 Storage 数据。
`nebula.exporter.image`	`vesoft/nebula-stats-exporter`	Exporter 服务的容器镜像。
`nebula.exporter.version`	`v3.3.0`	Exporter 服务的版本。
`nebula.exporter.replicas`	`1`	Exporter 服务的副本数量。
`nebula.exporter.env`	`[]`	Exporter 服务的环境变量。
`nebula.exporter.resources`	`{"resources":{"requests":{"cpu":"100m","memory":"128Mi"},"limits":{"cpu":"200m","memory":"256Mi"}}}`	Exporter 服务的资源限制和请求。
`nebula.agent.image`	`vesoft/nebula-agent`	Agent 服务的配置。用于备份和恢复及日志清理功能。
`nebula.agent.version`	`latest`	Agent 服务的版本。
`nebula.agent.resources`	`{"resources":{"requests":{"cpu":"100m","memory":"128Mi"},"limits":{"cpu":"200m","memory":"256Mi"}}}`	Agent 服务的资源限制和请求。
`nebula.console.username`	`root`	NebulaGraph命令行客户端的用户名。详情参见连接集群。
`nebula.console.password`	`nebula`	NebulaGraph命令行客户端的密码。
`nebula.console.image`	`vesoft/nebula-console`	NebulaGraph命令行客户端的容器镜像。
`nebula.console.version`	`latest`	NebulaGraph命令行客户端的版本。
`nebula.alpineImage`	`""`	Alpine Linux 容器镜像，用于获取节点所在 Zone 信息。
`imagePullSecrets`	`[]`	拉取私有镜像的 Secret 名称。
`nameOverride`	`""`	集群名称。
`fullnameOverride`	`""`	发布的实例名称。

<table>
<tr><th>字段</th><th>默认值</th><th>描述</th></tr>
<tr><td>nebula.<graphd|metad|storaged|exporter>.podLabels</td><td>{}</td><td>用于添加到 Pod 上的额外标签。</td></tr>
<tr><td>nebula.<graphd|metad|storaged|exporter>.podAnnotations</td><td>{}</td><td>用于添加到 Pod 上的额外注解。</td></tr>
<tr><td>nebula.<graphd|metad|storaged|exporter>.securityContext</td><td>{}</td><td>用于设置 Pod 的安全上下文。这可以包括运行 Pod 或容器的用户 ID、组 ID、Linux Capabilities 等。</td></tr>
<tr><td>nebula.<graphd|metad|storaged|exporter>.nodeSelector</td><td>{}</td><td>用于确定 Pod 应该在哪些节点上运行的标签选择器。</td></tr>
<tr><td>nebula.<graphd|metad|storaged|exporter>.tolerations</td><td>[]</td><td>用于允许 Pod 调度到具有特定 taints 的节点上。</td></tr>
<tr><td>nebula.<graphd|metad|storaged|exporter>.affinity</td><td>{}</td><td>用于设置 Pod 的亲和性规则，包括节点亲和性、Pod 亲和性和 Pod 反亲和性。</td></tr>
<tr><td>nebula.<graphd|metad|storaged|exporter>.readinessProbe</td><td>{}</td><td>用于检查容器是否已准备好处理服务请求。当探测器返回成功时，流量可以路由到该容器。</td></tr>
<tr><td>nebula.<graphd|metad|storaged|exporter>.livenessProbe</td><td>{}</td><td>用于检查容器是否仍在运行。如果探测器返回失败，Kubernetes 将杀死并重启容器。</td></tr>
<tr><td>nebula.<graphd|metad|storaged|exporter>.initContainers</td><td>[]</td><td>在应用容器启动之前运行的特殊容器，通常用于设置环境或初始化数据。</td></tr>
<tr><td>nebula.<graphd|metad|storaged|exporter>.sidecarContainers</td><td>[]</td><td>与主应用容器并行运行的容器，通常用于处理辅助任务，如日志处理、监控等。</td></tr>
<tr><td>nebula.<graphd|metad|storaged|exporter>.volumes</td><td>[]</td><td>定义需要附加到服务 Pod 的存储卷。</td></tr>
<tr><td>nebula.<graphd|metad|storaged|exporter>.volumeMounts</td><td>[]</td><td>指定容器内挂载存储卷的位置。</td></tr>
</table>

6. 创建 NebulaGraph 集群。

   通过 `--set` 参数自定义 NebulaGraph 集群配置项的默认值，例如，`--set nebula.storaged.replicas=3` 可设置 NebulaGraph 集群中 Storage 服务的副本数为 3。

```
helm install "${NEBULA_CLUSTER_NAME}" nebula-operator/nebula-cluster \

  # 指定集群 chart 的版本，不指定则默认安装最新版本 chart。
  # 执行 helm search repo -l nebula-operator/nebula-cluster 命令可查看所有 chart 版本。
  --version=1.8.0 \
  # 指定 NebulaGraph 集群所处的命名空间。
  --namespace="${NEBULA_CLUSTER_NAMESPACE}" \
  # 自定义集群名称。
  --set nameOverride="${NEBULA_CLUSTER_NAME}" \
  --set nebula.storageClassName="${STORAGE_CLASS_NAME}" \
  # 指定 NebulaGraph 集群的版本。
  --set nebula.version=vmaster
```

7. 查看 NebulaGraph 集群 Pod 的启动状态。

```
kubectl -n "${NEBULA_CLUSTER_NAMESPACE}" get pod -l "app.kubernetes.io/cluster=${NEBULA_CLUSTER_NAME}"
```

   返回：

```
NAME                           READY  STATUS   RESTARTS  AGE
nebula-exporter-854c76989c-mp725  1/1   Running  0        14h
```

```
nebula-graphd-0         1/1   Running 0     14h
nebula-graphd-1         1/1   Running 0     14h
nebula-metad-0          1/1   Running 0     14h
nebula-metad-1          1/1   Running 0     14h
nebula-metad-2          1/1   Running 0     14h
nebula-storaged-0       1/1   Running 0     14h
nebula-storaged-1       1/1   Running 0     14h
nebula-storaged-2       1/1   Running 0     14h
```

最后更新: 2025年8月4日

使用 **NebulaGraph Operator** 升级 **NebulaGraph** 集群版本

本文介绍如何使用 NebulaGraph Operator 在 Kubernetes 环境中升级 NebulaGraph 集群的版本。

> **历史版本兼容性**
>
> 1.x 版本的 NebulaGraph Operator 不兼容 3.x 以下版本的 NebulaGraph。

使用限制

- 只支持升级使用 NebulaGraph Operator 创建的 NebulaGraph 集群。
- 只支持升级 NebulaGraph 3.5.0 至 3.6.0 版本。
- 如需升级 NebulaGraph 企业版，联系我们。

前提条件

已创建 NebulaGraph 集群。具体步骤，参见创建 NebulaGraph 集群。

使用 KUBECTL 升级 NEBULAGRAPH 集群

本示例中待升级的 NebulaGraph 版本为 3.5.0。

1. 查看集群中服务的镜像版本。

```
kubectl get pods -l app.kubernetes.io/cluster=nebula -o jsonpath="{.items[*].spec.containers[*].image}" |tr -s '[[:space:]]' '\n' |sort |uniq -c
```

返回示例：

```
1 vesoft/nebula-graphd:v3.5.0
1 vesoft/nebula-metad:v3.5.0
3 vesoft/nebula-storaged:v3.5.0
```

2. 编辑 `nebula` 集群配置，将集群服务的 `version` 的值从 3.5.0 修改至 master。

a. 打开 `nebula` 集群的 YAML 文件。

```
kubectl edit nebulacluster nebula -n <namespace>
```

b. 更新 `version` 的值。

修改后的 YAML 文件内容如下：

```
apiVersion: apps.nebula-graph.io/v1alpha1
kind: NebulaCluster
metadata:
  name: nebula
spec:
  graphd:
    version: master //将值从 3.5.0 修改至 master。
    ...
  metad:
    version: master //将值从 3.5.0 修改至 master。
    ...
  storaged:
    version: master //将值从 3.5.0 修改至 master。
    ...
```

3. 应用配置。

保存 YAML 文件并退出后，Kubernetes 会自动更新集群的配置，并开始升级集群。

4. 等待约 2 分钟后，执行以下命令可查看到服务的镜像版本变更为 master。

```
kubectl get pods -l app.kubernetes.io/cluster=nebula -o jsonpath="{.items[*].spec.containers[*].image}" |tr -s '[[:space:]]' '\n' |sort |uniq -c
```

返回：

```
1 vesoft/nebula-graphd:master
1 vesoft/nebula-metad:master
3 vesoft/nebula-storaged:master
```

使用 HELM 升级 NEBULAGRAPH 集群

1. 更新 Helm 仓库，拉取最新的仓库资源。

```
helm repo update
```

2. 配置 Helm 的环境变量。

```
export NEBULA_CLUSTER_NAME=nebula      # NebulaGraph 集群的名字。
export NEBULA_CLUSTER_NAMESPACE=nebula  # NebulaGraph 集群所处的命名空间的名字。
```

3. 升级 NebulaGraph 集群。

例如升级至 master NebulaGraph 集群的命令如下。

```
helm upgrade "${NEBULA_CLUSTER_NAME}" nebula-operator/nebula-cluster \
  --namespace="${NEBULA_CLUSTER_NAMESPACE}" \
  --set nameOverride=${NEBULA_CLUSTER_NAME} \
  --set nebula.version=master
```

`--set nebula.version` 的值指需要升级集群的目标版本号。

4. 查看集群状态及集群版本。

   查看集群状态：

```
$ kubectl -n "${NEBULA_CLUSTER_NAMESPACE}" get pod -l "app.kubernetes.io/cluster=${NEBULA_CLUSTER_NAME}"
NAME        READY  STATUS   RESTARTS  AGE
nebula-graphd-0   1/1   Running  0     2m
nebula-graphd-1   1/1   Running  0     2m
nebula-metad-0   1/1   Running  0     2m
nebula-metad-1   1/1   Running  0     2m
nebula-metad-2   1/1   Running  0     2m
nebula-storaged-0  1/1   Running  0     2m
nebula-storaged-1  1/1   Running  0     2m
nebula-storaged-2  1/1   Running  0     2m
```

   查看集群版本：

```
$ kubectl get pods -l app.kubernetes.io/cluster=nebula  -o jsonpath="{.items[*].spec.containers[*].image}" |tr -s '[[:space:]]' '\n' |sort |uniq -c
   1 vesoft/nebula-graphd:master
   1 vesoft/nebula-metad:master
   3 vesoft/nebula-storaged:master
```

加速升级过程

集群的升级过程是一个滚动更新的过程也是耗时的过程，因为 Storage 服务的 Leader 分片副本会进行状态转移。用户可在集群实例的 YAML 文件中配置 `enableForceUpdate` 参数，来略过 Leader 分片副本转移操作，以达到加速升级的过程。详情参见配置滚动更新策略。

故障排查

升级过程中，如果遇到问题，查看集群服务 Pod 的日志。

```
kubectl logs <pod-name> -n <namespace>
```

同时，检查集群的状态和事件。

```
kubectl describe nebulaclusters <cluster-name> -n <namespace>
```

最后更新: 2025年8月4日

删除集群

本文介绍如何删除使用 NebulaGraph Operator 创建的 NebulaGraph 集群。

使用限制

- 只支持删除使用 NebulaGraph Operator 创建的NebulaGraph集群。
- 无法删除启用了删除保护的集群。更多信息，请参见配置删除保护。

使用 KUBECTL 删除 NEBULAGRAPH 集群

1. 查看所有创建的集群。

```
kubectl get nc --all-namespaces
```

返回示例：

```
NAMESPACE  NAME    READY  GRAPHD-DESIRED  GRAPHD-READY  METAD-DESIRED  METAD-READY  STORAGED-DESIRED  STORAGED-READY  AGE
default    nebula  True   2        2        3        3        3        3        38h
nebula     nebula2  True  1        1        1        1        1        1        2m7s
```

2. 删除集群。例如，删除名为 `nebula2` 的集群。

```
kubectl delete nc nebula2 -n nebula
```

返回示例：

```
nebulacluster.nebula-graph.io "nebula2" deleted
```

3. 确认删除。

```
kubectl get nc nebula2 -n nebula
```

返回示例：

```
No resources found in nebula namespace.
```

使用 HELM 删除 NEBULAGRAPH 集群

1. 查看所有的 Helm 发布实例。

```
helm list --all-namespaces
```

返回示例：

```
NAME            NAMESPACE               REVISION  UPDATED                                 STATUS    CHART                   APP VERSION
nebula          default                 1         2023-11-06 20:16:07.913136377 +0800 CST deployed  nebula-cluster-1.7.1    1.7.1
nebula-operator nebula-operator-system  3         2023-11-06 12:06:24.742397418 +0800 CST deployed  nebula-operator-1.7.1   1.7.1
```

2. 查看 Helm 发布实例的详细信息。例如，查看名为 `nebula` 的 Helm 发布实例的集群信息。

```
helm get values nebula -n default
```

返回示例：

```
USER-SUPPLIED VALUES:
imagePullSecrets:
- name: secret_for_pull_image
nameOverride: nebula # 集群名称
nebula:
  graphd:
    image: reg.vesoft-inc.com/xx
  metad:
    image: reg.vesoft-inc.com/xx
    licenseManagerURL: xxx:9119
  storageClassName: local-sc
  storaged:
    image: reg.vesoft-inc.com/xx
  version: v1.8.0 # 集群版本
```

3. 删除 Helm 发布实例。例如，删除名为 `nebula` 的 Helm 发布实例。

   ```
   helm uninstall nebula -n default
   ```

   返回示例：

   ```
   release "nebula" uninstalled
   ```

   当 Helm 发布实例被删除后，NebulaGraph Operator 会自动删除与该实例相关的所有 K8s 资源。

4. 确认集群资源被删除。

   ```
   kubectl get nc nebula -n default
   ```

   返回示例：

   ```
   No resources found in default namespace.
   ```

最后更新: 2025年8月4日

17.4.2 更新 NebulaGraph 集群的配置

NebulaGraph 集群中 Meta、Storage、Graph 服务都有各自的默认配置。NebulaGraph Operator 支持自定义集群服务的配置。本文介绍如何修改 NebulaGraph 集群的默认配置。

> Note
>
> 暂不支持通过 Helm 自定义 NebulaGraph 集群的配置参数。

前提条件

已使用 NebulaGraph Operator 创建一个集群。具体步骤，参见创建 NebulaGraph 集群。

配置方式

集群服务的配置在创建集群的 YAML 文件中，通过 `spec.<metad|graphd|storaged>.config` 参数指定。NebulaGraph Operator 会将 `config` 中的配置写入到对应服务的 ConfigMap 中，然后在服务启动时将 ConfigMap 挂载到服务的配置文件目录 `/usr/local/nebula/etc/` 下。

`config` 结构如下：

```
Config map[string]string `json:"config,omitempty"`
```

例如，修改 Graph 服务的 `enable_authorize` 参数的配置，可以在创建集群时或者之后，通过 `spec.graphd.config` 参数指定。

```
apiVersion: apps.nebula-graph.io/v1alpha1
kind: NebulaCluster
metadata:
  name: nebula
  namespace: default
spec:
  graphd:
    ...
    config: // 为 Graph 服务自定义参数。
      "enable_authorize": "true"  // 启用授权。默认值为 false。
...
```

如果需要为 Meta 服务和 Storage 服务配置 `config`，则在 `spec.metad.config` 和 `spec.storaged.config` 中添加对应的配置项。

可配置的参数

在 `config` 字段下可配置的参数详情，请分别参见：

- Meta 服务配置参数
- Storage 服务配置参数
- Graph 服务配置参数

参数更新与 **Pod** 重启规则

集群服务的配置参数分为两类：一类是必需重启服务才能更新的配置；另一类是可以在服务运行时动态更新的配置，在服务运行时更新的配置不会被持久化，重启服务后，配置会恢复到配置文件中的值。

关于服务的配置参数是否支持运行时动态更新，请查看上述服务配置参数详情页各个表格中是否支持运行时动态修改一列；或者查看 Dynamic runtime flags。

在更新集群服务的配置时，需要注意以下几点：

- 如果 `config` 下更新的参数均为支持运行时动态更新的参数，则不会触发服务 Pod 的重启，并且配置参数的更新不会被持久化。
- 如果 `config` 下更新的参数包含一个或多个不支持运行时动态更新的参数，则会触发服务 Pod 的重启，并且只有不支持运行时动态参数的更新才会被持久化。

> Note
>
> 若要在集群运行时动态修改参数配置且不触发 Pod 重启，请确保当前修改的参数全部支持运行时动态修改。

自定义端口配置

本示例演示如何自定义 Meta、Storage、Graph 服务的端口配置。

可以在 `config` 字段中添加 `port` 和 `ws_http_port` 参数，从而配置自定义的端口。这两个参数的详细信息，请参见Meta 服务配置参数、Storage 服务配置参数、Graph 服务配置参数的 networking 配置一节。

> Note
>
> - 自定义 `port` 和 `ws_http_port` 参数配置后，会触发 Pod 重启，并在重启后生效。
> - 在集群启动后，不建议修改 `port` 参数。

1. 修改集群配置文件。

a. 打开集群配置文件。

```
kubectl edit nc nebula
```

a. 修改配置文件，添加 `config` 字段，配置自定义端口。

```
apiVersion: apps.nebula-graph.io/v1alpha1
kind: NebulaCluster
metadata:
 name: nebula
 namespace: default
spec:
 graphd:
  config: # 为 Graph 服务自定义端口配置。
   port: "3669"
   ws_http_port: "8080"
  resources:
   requests:
    cpu: "200m"
    memory: "500Mi"
   limits:
    cpu: "1"
    memory: "1Gi"
  replicas: 1
  image: vesoft/nebula-graphd
  version: master
 metad:
  config: # 为 Meta 服务自定义端口配置。
   ws_http_port: 8081
  resources:
   requests:
    cpu: "300m"
    memory: "500Mi"
   limits:
    cpu: "1"
    memory: "1Gi"
  replicas: 1
  image: vesoft/nebula-metad
  version: master
  dataVolumeClaim:
   resources:
    requests:
     storage: 2Gi
   storageClassName: local-path
 storaged:
  config: # 为 Storage 服务自定义端口配置。
   ws_http_port: 8082
  resources:
   requests:
    cpu: "300m"
    memory: "500Mi"
   limits:
    cpu: "1"
    memory: "1Gi"
  replicas: 1
  image: vesoft/nebula-storaged
  version: master
  dataVolumeClaims:
  - resources:
    requests:
     storage: 2Gi
   storageClassName: local-path
  enableAutoBalance: true
 reference:
  name: statefulsets.apps
  version: v1
 schedulerName: default-scheduler
 imagePullPolicy: IfNotPresent
 imagePullSecrets:
 - name: nebula-image
 enablePVReclaim: true
 topologySpreadConstraints:
 - topologyKey: kubernetes.io/hostname
  whenUnsatisfiable: "ScheduleAnyway"
```

2. 保存配置文件。

   配置文件保存后，NebulaGraph Operator 会自动更新集群配置。

a. 按 `Esc` 键退出编辑模式。

b. 输入 `:wq` 保存配置文件并退出。

3. 验证配置已经生效。

```
kubectl get svc
```

返回示例：

```
NAME                       TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                       AGE
nebula-graphd-headless     ClusterIP   None             <none>        3669/TCP,8080/TCP             10m
nebula-graphd-svc          ClusterIP   10.102.13.115    <none>        3669/TCP,8080/TCP             10m
nebula-metad-headless      ClusterIP   None             <none>        9559/TCP,8081/TCP             11m
nebula-storaged-headless   ClusterIP   None             <none>        9779/TCP,8082/TCP,9778/TCP    11m
```

可以看到，Graph 服务的 RPC 守护进程监听端口已经变为 `3669`（默认 `9669`），HTTP 端口已经变为 `8080`（默认 `19669`）；Meta 服务的 HTTP 端口已经变为 `8081`（默认 `19559`）；Storage 服务的 HTTP 端口已经变为 `8082`（默认 `19779`）。

最后更新: 2025年8月4日

17.4.3 存储管理

动态在线扩容存储卷

在 K8s 环境中，NebulaGraph 的数据存储在持久化存储卷（PV）上。动态在线扩容存储卷指的是在不停机的情况下增加存储卷的容量，以满足 NebulaGraph 数据增长的需求。本文介绍如何在 K8s 环境中为 NebulaGraph 的服务动态在线扩容存储卷。

> Note
>
> - 集群创建后，不支持在集群运行时动态增加 PV 的数量。
> - 本文介绍的方法仅使用在线扩容存储卷，不支持在线缩容存储卷。

背景信息

K8s 中，StorageClass 是定义了一种存储类型的资源，它描述了一种存储的类，包括存储的提供者（provisioner）、参数和其他细节。当创建一个 PersistentVolumeClaim（PVC）并指定一个 StorageClass 时，K8s 会自动创建一个对应的 PV。动态扩容存储卷的原理是编辑 PVC 并增加存储卷的容量，然后 K8s 会根据 PVC 中指定的 `storageClassName` 自动扩容该 PVC 对应的 PV 的容量。在这个过程中，不会创建新的 PV，而是改变现有的 PV 的容量大小。只有动态存储卷才支持存储卷的动态扩容，即配置了 `storageClassName` 的 PVC。同时 StorageClass 的 `allowVolumeExpansion` 字段必须为 `true` 。详情参见 Expanding Persistent Volumes Claims。

在 Operator 中，不能直接编辑 PVC，因为 Operator 会根据 NebulaGraph 集群服务的配置 `spec.<metad|storaged>.dataVolumeClaim` 自动创建 PVC。因此，需要通过修改集群的配置来实现 PVC 的配置更新，然后自动触发 PV 的动态在线扩容。

前提条件

- K8s 的版本等于或高于 1.18。
- 已在 K8s 环境中创建 StorageClass。详情参见 Expanding Persistent Volumes Claims。
- 确保 StorageClass 配置了 `allowVolumeExpansion` 字段并且值为 `true` 。
- 确保 StorageClass 配置的 `provisioner` 支持动态扩容。
- 在 K8s 中创建一个 NebulaGraph 集群。具体步骤，参见创建 NebulaGraph 集群。
- NebulaGraph 集群 Pod 处于运行状态。

在线扩容存储卷示例

以下示例假设 StorageClass 的名称为 `ebs-sc`，NebulaGraph 集群的名称为 `nebula`，演示如何在线扩容 Storage 服务的存储卷。

1. 查看 Storage 服务 Pod 的状态。

```
kubectl get pod
```

示例输出：

```
nebula-storaged-0                    1/1   Running  0       43h
```

2. 查看 Storage 服务的 PVC 和 PV 信息。

```
# 查看 PVC 信息
kubectl get pvc
```

示例输出：

```
storaged-data-nebula-storaged-0  Bound  pvc-36ca3871-9265-460f-b812-7e73a718xxxx  5Gi    RWO      ebs-sc   43h
```

```
# 查看 PV 信息，确认 PV 的容量为 5Gi
kubectl get pv
```

示例输出：

```
pvc-36ca3871-9265-460f-b812-xxx  5Gi    RWO      Delete      Bound    default/storaged-data-nebula-storaged-0  ebs-sc       43h
```

3. 在符合前提条件的情况下，执行以下命令请求扩容 Storage 服务的存储卷至 10Gi。

```
kubectl patch nc nebula --type='merge' --patch '{"spec": {"storaged": {"dataVolumeClaims":[{"resources": {"requests": {"storage": "10Gi"}}, "storageClassName": "ebs-sc"}]}}}'
```

示例输出：

```
nebulacluster.apps.nebula-graph.io/nebula patched
```

4. 等待一分钟左右查看扩容后的 PVC 和 PV 信息。

```
kubectl get pvc
```

示例输出：

```
storaged-data-nebula-storaged-0  Bound  pvc-36ca3871-9265-460f-b812-7e73a718xxxx  10Gi    RWO      ebs-sc   43h
```

```
kubectl get pv
```

示例输出：

```
pvc-36ca3871-9265-460f-b812-xxx  10Gi    RWO      Delete      Bound    default/storaged-data-nebula-storaged-0  ebs-sc       43h
```

由上可见，PVC 和 PV 的容量都已扩容至 10Gi。

最后更新: 2025年8月4日

在 GKE 和 EKS 上使用本地持久化存储卷

在 K8s 中，本地持久化存储卷（Local Persistent Volumes，Local PV）直接使用节点的本地磁盘目录存储容器数据。相比网络存储，Local PV 提供更高的 IOPS 和更低的读写延迟，适合数据密集型应用。本文介绍如何在 GKE（Google Kubernetes Engine） 和 EKS（Amazon Elastic Kubernetes Service）中使用 Local PV 以及在使用 Local PV 过程中节点发生故障的处理方法。

虽然使用 Local PV 能提升性能，但是不同于网络存储，本地存储数据不会自动备份。如果节点因任何原因停止，本地存储上的所有数据可能会丢失。因此，使用 Local PV 时，需要在服务可用性、数据持久性和灵活性方面做出一定权衡取舍。

原理介绍

NebulaGraph Operator 实现了存储卷 Provisioner 接口。Provisioner 接口负责创建和删除 Persistent Volume 对象。通过存储卷 Provisioner，您可按需创建 Local PV。NebulaGraph Operator 根据集群配置文件中定义的 PVC 和 StorageClass，自动创建 PVC 并绑定到对应 Local PV 上。

当 Provisioner 接口创建 Local PV 时，Provisioner 控制器创建 `local` 类型的 PV，并且设置 `nodeAffinity` 字段，这样将使用 `local` 类型的 PV 的 Pod 被调度到特定节点上。当删除 Local PV 时，Provisioner 控制器移除 `local` 类型的 PV 对象，并清理节点上的存储资源。

前提条件

已安装 NebulaGraph Operator。详情参见安装 NebulaGraph Operator。

操作步骤

以下示例所创建的资源均处于默认命名空间 `default` 中。

GKE 上使用 Local PV　　EKS 上使用 Local PV

1. 创建具备本地 SSD 的节点池。

```
gcloud container node-pools create "pool-1" --cluster "gke-1" --region us-central1 --node-version "1.27.10-gke.1055000" --machine-type "n2-standard-2" --local-nvme-ssd-block count=2 --max-surge-upgrade 1 --max-unavailable-upgrade 0 --num-nodes 1 --enable-autoscaling --min-nodes 1 --max-nodes 2
```

关于创建具备本地 SSD 节点池的参数解释，请参见创建具备本地 SSD 的节点池。

2. 使用 DaemonSet 配置 RAID 并对磁盘进行格式化。

a. 下载 gke-daemonset-raid-disks.yaml 文件。

b. 部署 RAID 磁盘 DaemonSet。DaemonSet 会在所有本地 SSD 磁盘上设置 RAID 0 阵列，并将设备格式化为 ext4 文件系统。

```
kubectl apply -f gke-daemonset-raid-disks.yaml
```

3. 部署 Local PV Provisioner。

a. 下载 local-pv-provisioner.yaml 文件。

b. 部署 Local PV Provisioner。

```
kubectl apply -f local-pv-provisioner.yaml
```

4. 在集群配置文件中，通过 `spec.storaged.dataVolumeClaims` 或 `spec.metad.dataVolumeClaim` 定义的 PVC 和 StorageClass 来自动创建 Local PV。其中，StorageClass 需要配置成 `local-nvme` 。具体操作步骤，参见创建NebulaGraph集群。

NebulaGraph集群的部分配置

```
...
metad:
  dataVolumeClaim:
    resources:
      requests:
        storage: 2Gi
    storageClassName: local-nvme
storaged:
  dataVolumeClaims:
  - resources:
      requests:
        storage: 2Gi
    storageClassName: local-nvme
...
```

5. 查看 PV 列表。

```
kubectl get pv
```

返回：

```
NAME                                       CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS  CLAIM                                  STORAGECLASS  REASON  AGE
pvc-01be9b75-9c50-4532-8695-08e11b489718  5Gi       RWO           Delete          Bound   default/storaged-data-nebula-storaged-0  local-nvme            3m35s
pvc-09de8eb1-1225-4025-b91b-fbc0bcce670f  5Gi       RWO           Delete          Bound   default/storaged-data-nebula-storaged-1  local-nvme            3m35s
pvc-4b2a9ffb-9000-4998-a7bb-edb825c872cb  5Gi       RWO           Delete          Bound   default/storaged-data-nebula-storaged-2  local-nvme            3m35s
...
```

6. 查看 PV 的详细信息。

```
kubectl get pv pvc-01be9b75-9c50-4532-8695-08e11b489718 -o yaml
```

返回：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  annotations:
    local.pv.provisioner/selected-node: gke-snap-test-snap-test-591403a8-xdfc
    nebula-graph.io/pod-name: nebula-storaged-0
    pv.kubernetes.io/provisioned-by: nebula-cloud.io/local-pv
  creationTimestamp: "2024-03-05T06:12:32Z"
  finalizers:
  - kubernetes.io/pv-protection
  labels:
    app.kubernetes.io/cluster: nebula
    app.kubernetes.io/component: storaged
    app.kubernetes.io/managed-by: nebula-operator
    app.kubernetes.io/name: nebula-graph
  name: pvc-01be9b75-9c50-4532-8695-08e11b489718
  resourceVersion: "9999469"
  uid: ee28a4da-6026-49ac-819b-2075154b4724
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 5Gi
  claimRef:
    apiVersion: v1
```

云上环境中 **LOCAL PV** 的故障转移

使用网络存储（如 AWS EBS、Google Cloud Persistent Disk、Azure Disk Storage、Ceph、NFS 等）作为 PV，存储资源不依赖于任何特定节点，因此无论 Pod 被调度到哪个节点，都能挂载并使用此存储资源。然而，使用本地存储盘作为 PV 时，由于节点亲和性（NodeAffinity），存储资源只能被特定节点上的 Pod 使用。

NebulaGraph的 Storage 服务具备数据冗余能力，可以设置多个奇数分片副本。节点故障时，关联分片会自动迁移到健康节点。但是，使用 Local PV 的 Storage Pod 由于节点亲和性，不能在其他节点运行，必须等待节点恢复。若要在其他节点运行，需要解除 Pod 与 Local PV 的绑定。

针对使用 Local PV 过程中发生节点故障的情况，NebulaGraph Operator 支持结合云上环境的资源弹性伸缩能力进行自动故障转移操作。通过在集群的配置文件中设置 `spec.enableAutoFailover` 为 `true`，自动解除 Pod 与 Local PV 的绑定，从而使 Pod 能在其他节点上运行。

示例如下：

```
...
spec:
  # 开启自动故障转移
  enableAutoFailover: true
  # Storage 服务为`OFFLINE`状态后等待自动故障转移的时间。
  # 默认值 5 分钟。
  # 如果 Storage 服务在此期间内恢复正常状态，不会触发故障转移。
  failoverPeriod: "2m"
  ...
```

最后更新: 2025年8月4日

回收 PV

NebulaGraph Operator 使用持久化卷 PV（Persistent Volume）和持久化卷声明 PVC（Persistent Volume Claim）来存储持久化数据。如果用户不小心删除了一个 NebulaGraph 集群，默认 PV 和 PVC 对象及其数据仍可保留，以确保数据安全。

用户也可以在集群实例的配置文件中通过设置参数 `spec.enablePVReclaim` 为 `true` 来定义在删除集群后自动删除 PVC 以释放数据。至于在删除 PVC 后是否删除 PV，用户需要自定义 PV 的回收策略。参见 StorageClass 中设置 reclaimPolicy 和 PV Reclaiming 了解 PV 回收策略。

前提条件

在 K8s 中创建一个 NebulaGraph 集群。具体步骤，参见创建 NebulaGraph 集群。

操作步骤

以下示例使用名为 `nebula` 的集群、名为 `nebula_cluster.yaml` 的 YAML 配置文件，说明如何设置 `enablePVReclaim`：

1. 执行以下命令进入 `nebula` 集群的编辑页面。

```
kubectl edit nebulaclusters.apps.nebula-graph.io nebula
```

2. 在 YAML 文件的 `spec` 配置项中，添加 `enablePVReclaim` 并设置其值为 `true`。

```
apiVersion: apps.nebula-graph.io/v1alpha1
kind: NebulaCluster
metadata:
 name: nebula
spec:
 enablePVReclaim: true //设置其值为 true。
 graphd:
  image: vesoft/nebula-graphd
  logVolumeClaim:
   resources:
    requests:
     storage: 2Gi
   storageClassName: fast-disks
  replicas: 1
  resources:
   limits:
    cpu: "1"
    memory: 1Gi
   requests:
    cpu: 500m
    memory: 500Mi
  version: master
 imagePullPolicy: IfNotPresent
 metad:
  dataVolumeClaim:
   resources:
    requests:
     storage: 2Gi
   storageClassName: fast-disks
  image: vesoft/nebula-metad
  logVolumeClaim:
   resources:
    requests:
     storage: 2Gi
   storageClassName: fast-disks
  replicas: 1
  resources:
   limits:
    cpu: "1"
    memory: 1Gi
   requests:
    cpu: 500m
    memory: 500Mi
  version: master
 nodeSelector:
  nebula: cloud
 reference:
  name: statefulsets.apps
  version: v1
 schedulerName: default-scheduler
 storaged:
  dataVolumeClaims:
  - resources:
    requests:
     storage: 2Gi
   storageClassName: fast-disks
  - resources:
    requests:
     storage: 2Gi
   storageClassName: fast-disks
  image: vesoft/nebula-storaged
  logVolumeClaim:
   resources:
    requests:
     storage: 2Gi
   storageClassName: fast-disks
  replicas: 3
  resources:
   limits:
    cpu: "1"
    memory: 1Gi
   requests:
    cpu: 500m
    memory: 500Mi
  version: master
...
```

3. 执行 `kubectl apply -f nebula_cluster.yaml` 使上述更新生效。

在上述操作即 `enablePVReclaim` 设置为 `true` 后，当集群删除时，系统将会自动删除 PVC 以回收存储资源。是否删除 PV，取决于 PV 的回收策略。

最后更新: 2025年8月4日

17.4.4 管理集群日志

NebulaGraph 集群各服务（graphd、metad、storaged）在运行期间会生成运行日志，日志默认存放在各个服务容器的 `/usr/local/nebula/logs` 目录下。

查看运行日志

如果您需要查看 NebulaGraph 集群的运行日志，可以通过 `kubectl logs` 命令查看。

例如，查看 Storage 服务的运行日志：

```
// 查看 Storage 服务 Pod 的名称（nebula-storaged-0）。
$ kubectl get pods -l app.kubernetes.io/component=storaged
NAME                READY  STATUS   RESTARTS   AGE
nebula-storaged-0        1/1   Running  0       45h
...

// 进入 Storage 服务所在容器 storaged。
$ kubectl exec -it nebula-storaged-0 -c storaged -- /bin/bash

// 查看 Storage 服务的运行日志。
$ cd /usr/local/nebula/logs
```

清理日志

集群服务在运行期间生成的运行日志会占用磁盘空间，为避免占用过多磁盘空间，Operator 使用 sidecar 容器定期清理和归档日志。

为了方便日志的采集和管理，每个 NebulaGraph 服务都会部署一个 sidecar 容器，负责收集该服务容器产生的日志，并将其发送到指定的日志磁盘中。sidecar 容器使用 logrotate 工具自动清理和归档日志。

在集群实例的 YAML 配置文件中，通过 `spec.logRotate` 字段开启日志轮转功能，同时在 `spec.<graphd|metad|storaged>.config` 下设置 `timestamp_in_logfile_name` 值为 `false` 以自动对各服务日志进行清理和归档。默认情况下，日志轮转功能是关闭的。为 NebulaGraph 各个服务都开启日志轮转功能示例如下：

```
...
spec:
  graphd:
    config:
      # 是否在日志文件名中包含时间戳，需设置为 false 以实现日志轮转。默认为 true。
      "timestamp_in_logfile_name": "false"
  metad:
    config:
      "timestamp_in_logfile_name": "false"
  storaged:
    config:
      "timestamp_in_logfile_name": "false"
  logRotate: # 日志轮转配置
    # 日志文件在被删除前会被轮转的次数。默认值为 5，0 表示删除前不会被轮转。
    rotate: 5
    # 仅当日志文件增长超过定义的字节大小时才会轮转日志文件。默认值为 200M。
    size: "200M"
```

收集日志

如果不想挂载额外的日志磁盘备份日志文件，或者想通过诸如 fluent-bit 之类的服务收集日志并将其发送到日志中心，可以配置日志至标准错误输出。Operator 使用 glog 工具将日志记录到标准错误输出。

> **Note**
>
> 目前 Operator 仅收集标准错误日志。

在集群实例的 YAML 配置文件中，可以在各个服务下的 `config` 和 `env` 字段中配置日志记录到标准错误输出。

```
...
spec:
  graphd:
    config:
      # 是否将标准错误重定向到单独的输出文件。默认值为 false，表示不重定向。
```

```yaml
    redirect_stdout: "false"
    # 日志内容的严重程度级别：INFO、WARNING、ERROR 和 FATAL。取值分别为 0、1、2 和 3。
    stderrthreshold: "0"
  env:
  - name: GLOG_logtostderr # 日志写入标准错误而不是文件。
    value: "1" # 1 表示写入标准错误，0 表示写入文件中。
  image: vesoft/nebula-graphd
  replicas: 1
  resources:
    requests:
      cpu: 500m
      memory: 500Mi
  service:
    externalTrafficPolicy: Local
    type: NodePort
  version: vmaster
metad:
  config:
    redirect_stdout: "false"
    stderrthreshold: "0"
  dataVolumeClaim:
    resources:
      requests:
        storage: 1Gi
    storageClassName: ebs-sc
  env:
  - name: GLOG_logtostderr
    value: "1"
  image: vesoft/nebula-metad
...
```

最后更新: 2025年8月4日

17.4.5 安全

开启准入控制

K8s 的准入控制（Admission Control）是一种安全机制，并在运行时作为 Webhook 运行。通过准入 Webhook 对请求进行拦截和修改，从而保证集群的安全性。准入 Webhook 操作包括验证（Validating）和变更（Mutating）两类。NebulaGraph Operator 仅支持验证操作，并提供一些默认的准入控制规则。本文介绍 NebulaGraph Operator 的默认准入控制规则及如何开启准入控制。

前提条件

已使用 K8s 创建一个集群。具体步骤，参见创建 NebulaGraph 集群。

准入控制规则

K8s 的准入控制允许用户在 Kubernetes API Server 处理请求之前，插入自定义的逻辑或策略。这种机制可以用于实现一些安全策略，比如限制 Pod 的资源使用量，或者限制 Pod 的访问权限等。NebulaGraph Operator 仅支持验证操作，即对请求进行验证和拦截，不支持对请求进行变更操作。

开启准入控制后，NebulaGraph Operator 默认启用以下准入验证控制规则且不允许禁用：

- 禁止通过 `dataVolumeClaims` 为 Storage 服务追加额外的 PV。
- 禁止缩小所有服务的 PVC 的容量，但是可以扩容。
- 禁止在 Storage 服务扩缩容期间，进行任何二次操作。

开启准入控制后，NebulaGraph Operator 支持通过添加注解的方式自行配置以下准入验证控制规则：

- 添加了 `ha-mode` 注解的集群需满足高可用模式下的最小副本数：
- Graph 服务：至少需要 2 个副本。
- Meta 服务：至少需要 3 个副本。
- Storage 服务：至少需要 3 个副本。

> **Note**
>
> 高可用模式是指 NebulaGraph 集群服务的高可用。Storage 服务和 Meta 服务是有状态的服务，其副本数据通过 Raft 协议保持一致性且副本数量不能为偶数。因此，高可用模式下，至少需要 3 个 Storage 服务和 3 个 Meta 服务。Graph 服务为无状态的服务，因此其副本数量可以为偶数，但至少需要 2 个副本。

- 添加了 `delete-protection` 注解的集群将无法被删除。更多信息，请参见配置删除保护。

为准入 WEBHOOK 创建证书

为了确保通信的安全性和数据的完整性，K8s 的 API server 和准入 Webhook 之间的通信默认通过 HTTPS 协议进行，因此使用准入控制还需要为准入 Webhook 提供 TLS 证书。cert-manager 是一个 K8s 的证书管理控制器，可以自动化证书的签发和更新。NebulaGraph Operator 使用 cert-manager 来管理证书。

当 cert-manager 安装完成并且开启准入控制时，NebulaGraph Operator 会自动创建一个 Issuer，用于签发准入 Webhook 所需的证书，同时会创建一个 Certificate，用于存储签发的证书。签发的证书被存储在 `nebula-operator-webhook-secret` 的 Secret 中。

开启准入控制

1. 安装部署 cert-manager。

```
kubectl apply -f https://github.com/cert-manager/cert-manager/releases/download/v1.13.1/cert-manager.yaml
```

建议部署最新版本 cert-manager。详情参见 cert-manager 官方文档。

2. 修改 NebulaGraph Operator 的配置文件，开启准入控制。默认准入控制是关闭的，需要手动开启。

```
# 查看当前的配置
helm show values nebula-operator/nebula-operator
```

```
# 修改配置，将`enableAdmissionWebhook`设置为`true`。
helm upgrade nebula-operator nebula-operator/nebula-operator --set enableAdmissionWebhook=true
```

> Note
>
> `nebula-operator` 为 chart 所在仓库的名称，`nebula-operator/nebula-operator` 为 chart 的名称。如果没有指定 chart 的命名空间，默认为 `default`。

3. 查看准入 Webhook 的证书 Secret。

```
kubectl get secret nebula-operator-webhook-secret -o yaml
```

如果输出的结果中包含证书内容，则表示准入 Webhook 的证书已经创建成功。

4. 验证控制规则。

- 验证禁止通过 `dataVolumeClaims` 为 Storage 服务追加额外的 PV。

```
$ kubectl patch nc nebula --type='merge' --patch '{"spec": {"storaged": {"dataVolumeClaims":[{"resources": {"requests": {"storage": "2Gi"}}, "storageClassName": "local-path"},{"resources": {"requests": {"storage": "3Gi"}}, "storageClassName": "fask-disks"}]}}}'
Error from server: admission webhook "nebulaclustervalidating.nebula-graph.io" denied the request: spec.storaged.dataVolumeClaims: Forbidden: storaged dataVolumeClaims is immutable
```

- 验证禁止缩小 Storage 服务的 PVC 的容量。

```
$ kubectl patch nc nebula --type='merge' --patch '{"spec": {"storaged": {"dataVolumeClaims":[{"resources": {"requests": {"storage": "1Gi"}}, "storageClassName": "fast-disks"}]}}}'
Error from server: admission webhook "nebulaclustervalidating.nebula-graph.io" denied the request: spec.storaged.dataVolumeClaims: Invalid value: resource.Quantity{i:resource.int64Amount{value:1073741824, scale:0}, d:resource.infDecAmount{Dec:(*inf.Dec)(nil)}, s:"1Gi", Format:"BinarySI"}: data volume size can only be increased
```

- 验证禁止在 Storage 服务缩容期间，进行任何二次操作。

```
$ kubectl patch nc nebula --type='merge' --patch '{"spec": {"storaged": {"replicas": 5}}}'
nebulacluster.apps.nebula-graph.io/nebula patched
$ kubectl patch nc nebula --type='merge' --patch '{"spec": {"storaged": {"replicas": 3}}}'
Error from server: admission webhook "nebulaclustervalidating.nebula-graph.io" denied the request: [spec.storaged: Forbidden: field is immutable while in ScaleOut phase, spec.storaged.replicas: Invalid value: 3: field is immutable while not in Running phase]
```

- 验证高可用模式下的最小副本数。

```
# 标注集群为高可用模式
$ kubectl annotate nc nebula nebula-graph.io/ha-mode=true
# 验证 Graph 服务的最小副本数
$ kubectl patch nc nebula --type='merge' --patch '{"spec": {"graphd": {"replicas":1}}}'
Error from server: admission webhook "nebulaclustervalidating.nebula-graph.io" denied the request: spec.graphd.replicas: Invalid value: 1: should be at least 2 in HA mode
```

- 验证集群删除保护，具体请参见配置删除保护。

最后更新: 2025年8月4日

配置删除保护

NebulaGraph Operator 支持删除保护，以防止NebulaGraph集群被意外删除。本文介绍如何为NebulaGraph集群配置删除保护。

前提条件

- NebulaGraph集群已创建。更多信息，请参见创建NebulaGraph集群。
- NebulaGraph集群已启用准入控制。更多信息，请参见开启准入控制。

添加注解以开启删除保护

添加 `delete-protection` 注解到集群。

```
kubectl annotate nc nebula -n nebula-test nebula-graph.io/delete-protection=true
```

以上命令为 `nebula-test` 命名空间中的 `nebula` 集群启用了删除保护。

验证删除保护

To verify that deletion protection is enabled, run the following command:

运行以下命令验证删除保护是否已启用：

```
kubectl delete nc nebula -n nebula-test
```

以上命令尝试删除 `nebula-test` 命名空间中的 `nebula` 集群。

返回信息：

```
Error from server: admission webhook "nebulaclustervalidating.nebula-graph.io" denied the request: metadata.annotations[nebula-graph.io/delete-protection]: Forbidden: protected cluster cannot be deleted
```

删除注解以关闭删除保护

从集群中删除 `delete-protection` 注解：

```
kubectl annotate nc nebula -n nebula-test nebula-graph.io/delete-protection-
```

以上命令为 `nebula-test` 命名空间中的 `nebula` 集群关闭了删除保护。

最后更新: 2025年8月4日

17.4.6 高可用和负载均衡

故障自愈

NebulaGraph Operator 调用 NebulaGraph 集群提供的接口，动态地感知服务是否正常运行。当 NebulaGraph 集群中某一组件停止运行时，NebulaGraph Operator 会自动地进行容错处理。本文通过删除 NebulaGraph 集群中 1 个 Storage 服务 Pod，模拟集群故障为例，说明 Nebular Operator 如何进行故障自愈。

前提条件

安装 NebulaGraph Operator

操作步骤

1. 创建 NebulaGraph 集群。具体步骤参考创建 NebulaGraph 集群。
2. 待所有 Pods 都处于 `Running` 状态时，模拟故障，删除名为 `<cluster_name>-storaged-2` Pod。

```
kubectl delete pod <cluster-name>-storaged-2 --now
```

`<cluster_name>` 为 NebulaGraph 集群的名称。

3. NebulaGraph Operator 自动创建名为 `<cluster-name>-storaged-2` 的 Pod，以修复故障。

执行 `kubectl get pods` 查看 `<cluster-name>-storaged-2` Pod 的创建状态。

```
...
nebula-cluster-storaged-1    1/1  Running          0     5d23h
nebula-cluster-storaged-2    0/1  ContainerCreating 0     1s
...
```

```
...
nebula-cluster-storaged-1    1/1  Running   0     5d23h
nebula-cluster-storaged-2    1/1  Running   0     4m2s
...
```

当 `<cluster-name>-storaged-2` 的状态由 `ContainerCreating` 变为 `Running` 时，说明自愈成功。

最后更新: 2025年8月4日

17.4.7 高阶功能

优化滚动更新中的 **Leader** 分布

NebulaGraph 集群使用分布式架构将数据分成多个逻辑分片，这些分片通常均分在不同的节点上。分布式系统中，同一份数据通常会有多个副本。为了保证多个副本数据的一致性，NebulaGraph 集群使用 Raft 协议实现了多分片副本同步。Raft 协议中，每个分片都会选举出一个 Leader 副本，Leader 副本负责处理写请求，Follower 副本负责处理读请求。

通过 Operator 创建的 NebulaGraph 集群在滚动更新过程中，一个存储节点会暂时停止提供服务以进行更新。关于滚动更新的概述，参见执行滚动更新。如果 Leader 副本所在的节点停止提供服务，会导致该分片的读写不可用。为了避免这种情况，Operator 会在 NebulaGraph 集群滚动更新过程中，默认将 Leader 副本迁移到其他未受影响节点上。这样，当一个存储节点处于更新状态时，其他节点上的 Leader 副本能够继续处理客户端请求，以保证集群的读写可用性。

一个存储节点上的所有 Leader 副本全部迁移到其他节点的这个过程可能会持续较长时间。为了更好地控制滚动更新的时间，Operator 提供了一个名为 `enableForceUpdate` 参数。当确定没有外部访问流量时，可将该参数设置为 `true`，这样，Leader 副本将不会被迁移到其他节点上，从而加快滚动更新的速度。

滚动更新触发条件

Operator 会在以下情况下触发 NebulaGraph 集群的滚动更新：

- NebulaGraph 集群的版本发生变化。
- NebulaGraph 集群的配置发生变化。
- NebulaGraph 集群的服务执行重启操作。

配置滚动更新策略

在创建集群实例的 YAML 文件中，添加 `spec.storaged.enableForceUpdate` 参数，设置为 `true` 或 `false`，以控制滚动更新的速度。

当 `enableForceUpdate` 为 `true` 时，表示不迁移分片 Leader 副本，从而加快滚动更新的速度；反之，表示迁移分片 Leader 副本，以保证集群的读写可用性。默认值为 `false`。

> **Warning**
>
> 设置 `enableForceUpdate` 为 `true` 时，确保没有流量进入集群进行读写操作。因为该设置会强制重建集群 Pod，此过程会发生数据丢失或客户端请求失败的情况。

配置示例：

```
...
spec:
...
  storaged:
    enableForceUpdate: true // 设置为 true 时，表示不迁移分片 Leader 副本，而是直接重建集群 Pod。
    ...
```

最后更新: 2025年8月4日

重启 K8s 上的NebulaGraph集群服务

> Note
>
> 重启集群服务 Pod 功能为 Alpha 版本功能。

在日常维护时，出于各种原因需要重启NebulaGraph集群的某个服务 Pod，例如 Pod 状态异常或是执行强行重启逻辑。Pod 重启的本质是重启服务进程，为了确保服务的高可用性，NebulaGraph Operator 支持优雅滚动重启集群内所有 Graph，Meta，或 Storage 服务 Pod，也支持优雅重启单个 Storage 服务 Pod。

前提条件

已经在 K8s 环境中创建了一个NebulaGraph集群。具体步骤，参见创建NebulaGraph集群。

优雅滚动重启集群内某类服务的所有 POD

通过在不同服务 StatefulSet 控制器的配置中添加注解（annotation） `nebula-graph.io/restart-timestamp` 并将值设置为当前时间来实现优雅滚动重启集群同类服务 Pod。当 NebulaGraph Operator 检测到相应服务的 StatefulSet 控制器存在注解 `nebula-graph.io/restart-timestamp` 并且其值发生了变更，即会触发优雅滚动重启集群内某类服务的所有 Pod 的操作。

以下示例中，为所有 Graph 服务都设置注解，表示将逐个重启所有 Graph 服务 Pod。

假设所有集群名为 `nebula`，集群资源都放在 `default` 命名空间下，执行以下命令：

1. 查看 StatefulSet 控制器的名称。

```
kubectl get statefulset
```

示例输出：

```
NAME        READY  AGE
nebula-graphd   2/2   33s
nebula-metad    3/3   69s
nebula-storaged 3/3   69s
```

2. 获取当前时间戳。

```
date -u +%s
```

示例输出：

```
1700547115
```

3. 覆盖 StatefulSet 控制器时间戳注解以触发优雅滚动重启操作。

```
kubectl annotate statefulset nebula-graphd nebula-graph.io/restart-timestamp="1700547115" --overwrite
```

示例输出：

```
statefulset.apps/nebula-graphd annotate
```

4. 观察重启过程。

```
kubectl get pods -l app.kubernetes.io/cluster=nebula,app.kubernetes.io/component=graphd -w
```

示例输出：

```
NAME        READY  STATUS  RESTARTS  AGE
nebula-graphd-0 1/1   Running 0     9m37s
nebula-graphd-1 0/1   Running 0     17s
nebula-graphd-1 1/1   Running 0     20s
nebula-graphd-0 1/1   Terminating 0     9m40s
nebula-graphd-0 0/1   Terminating 0     9m41s
nebula-graphd-0 0/1   Terminating 0     9m42s
nebula-graphd-0 0/1   Terminating 0     9m42s
nebula-graphd-0 0/1   Terminating 0     9m42s
nebula-graphd-0 0/1   Pending   0     0s
nebula-graphd-0 0/1   Pending   0     0s
nebula-graphd-0 0/1   ContainerCreating 0     0s
nebula-graphd-0 0/1   Running     0     2s
```

上述输出显示所有 Graph 服务 Pod 重启过程。

5. 确认 StatefulSet 控制器注解更新。

```
kubectl get statefulset nebula-graphd -o yaml | grep "nebula-graph.io/restart-timestamp"
```

示例输出：

```
nebula-graph.io/last-applied-configuration: '{"persistentVolumeClaimRetentionPolicy":{"whenDeleted":"Retain","whenScaled":"Retain"},"podManagementPolicy":"Parallel","replicas":2,"revisionHistoryLimit":10,"selector":{"matchLabels":{"app.kubernetes.io/cluster":"nebula","app.kubernetes.io/component":"graphd","app.kubernetes.io/managed-by":"nebula-operator","app.kubernetes.io/name":"nebula-graph"}},"serviceName":"nebula-graphd-headless","template":{"metadata":{"annotations":{"nebula-graph.io/cm-hash":"7c55c0e5ac74e85f","nebula-graph.io/restart-timestamp":"1700547815"},"creationTimestamp":null,"labels":{"app.kubernetes.io/cluster":"nebula","app.kubernetes.io/component":"graphd","app.kubernetes.io/managed-by":"nebula-operator","app.kubernetes.io/name":"nebula-graph"}},"spec":{"containers":[{"command":["/bin/sh","-ecx","exec
nebula-graph.io/restart-timestamp: "1700547115"
  nebula-graph.io/restart-timestamp: "1700547815"
```

由上述输出可知，StatefulSet 控制器的注解已经更新并且 Graph 服务的所有 Pod 已经重启。

优雅滚动重启单个 STORAGE 服务 POD

通过在 Storage 服务的 StatefulSet 控制器的配置中添加注解（annotation） `nebula-graph.io/restart-ordinal` 并将值设置为 Storage 服务 Pod 的序号来实现优雅滚动重启单个 Storage 服务 Pod，即执行状态转移操作。在 Storage 服务 Pod 重启后，添加的注解会被删除。

以下示例为序号为 `1` 的 Storage Pod 添加注解，表示将优雅重启名为 `nebula-storaged-1` 的 Storage 服务 Pod。

假设所有集群名为 `nebula`，集群资源都放在 `default` 命名空间下，执行以下命令：

1. 查看 StatefulSet 控制器名称。

```
kubectl get statefulset
```

示例输出：

```
NAME            READY   AGE
nebula-graphd    2/2    33s
nebula-metad     3/3    69s
nebula-storaged  3/3    69s
```

2. 获取 Storage 服务 Pod 的序号。

```
kubectl get pods -l app.kubernetes.io/cluster=nebula,app.kubernetes.io/component=storaged
```

示例输出：

```
NAME              READY   STATUS    RESTARTS   AGE
nebula-storaged-0 1/1     Running   0          13h
nebula-storaged-1 1/1     Running   0          13h
nebula-storaged-2 1/1     Running   0          13h
nebula-storaged-3 1/1     Running   0          13h
nebula-storaged-4 1/1     Running   0          13h
nebula-storaged-5 1/1     Running   0          13h
nebula-storaged-6 1/1     Running   0          13h
nebula-storaged-7 1/1     Running   0          13h
nebula-storaged-8 1/1     Running   0          13h
```

3. 为 `nebula-storaged-1` Pod 添加注解以触发优雅滚动重启该 Pod 操作。

```
kubectl annotate statefulset nebula-storaged nebula-graph.io/restart-ordinal="1"
```

示例输出：

```
statefulset.apps/nebula-storaged annotate
```

4. 观察重启过程。

```
kubectl get pods -l app.kubernetes.io/cluster=nebula,app.kubernetes.io/component=storaged -w
```

示例输出：

```
NAME              READY   STATUS    RESTARTS   AGE
nebula-storaged-0 1/1     Running   0          13h
nebula-storaged-1 1/1     Running   0          13h
nebula-storaged-2 1/1     Running   0          13h
nebula-storaged-3 1/1     Running   0          13h
nebula-storaged-4 1/1     Running   0          13h
nebula-storaged-5 1/1     Running   0          12h
nebula-storaged-6 1/1     Running   0          12h
nebula-storaged-7 1/1     Running   0          12h
nebula-storaged-8 1/1     Running   0          12h


nebula-storaged-1 1/1     Running   0          13h
nebula-storaged-1 1/1     Terminating   0          13h
nebula-storaged-1 0/1     Terminating   0          13h
nebula-storaged-1 0/1     Terminating   0          13h
nebula-storaged-1 0/1     Terminating   0          13h
nebula-storaged-1 0/1     Terminating   0          13h
nebula-storaged-1 0/1     Pending   0          0s
nebula-storaged-1 0/1     Pending   0          0s
nebula-storaged-1 0/1     ContainerCreating   0          0s
nebula-storaged-1 0/1     Running             0          1s
nebula-storaged-1 1/1     Running             0          10s
```

由上述输出可知，`nebula-storaged-1` Storage 服务 Pod 已经重启。

在重启单个 Storage 服务 Pod 后，数据 Leader 的分布可能不均衡。可以执行 `BALANCE LEADER` 命令重新均衡数据 Leader 的分布。关于如何查看 Leader 分布情况，请参见 `SHOW HOSTS`。

最后更新: 2025年8月4日

17.5 常见问题

17.5.1 NebulaGraph Operator 支持 v1.x 版本的 NebulaGraph 吗?

不支持，因为 v1.x 版本的 NebulaGraph 不支持 DNS，而 NebulaGraph Operator 需要使用 DNS。

17.5.2 使用本地存储是否可以保证集群稳定性?

无法保证。使用本地存储意味着 Pod 被绑定到一个特定的节点，NebulaGraph Operator 目前不支持在绑定的节点发生故障时进行故障转移。

17.5.3 扩缩容集群时，如何确保稳定性?

建议提前备份数据，以便故障发生时回滚数据。

17.5.4 Operator 文档中的 replica 和 NebulaGraph 内核文档中的 replica 是一样的吗?

二者是不同的概念。Operator 文档中的 replica 是 K8s 中的 Pod 副本，而内核文档中的 replica 是 NebulaGraph 中的分片副本。

17.5.5 如何查看 NebulaGraph 集群中各服务的日志?

用户可以通过进入容器并在容器内查看日志文件的方式来获取 NebulaGraph 集群各服务日志。

操作如下：

```
# 查找要进入的容器所在的 Pod 名称。其中 <cluster-name> 为集群名称。
kubectl get pods -l app.kubernetes.io/cluster=<cluster-name>

# 进入 Pod 中的容器，例如进入 nebula-graphd-0 容器。
kubectl exec -it nebula-graphd-0 -- /bin/bash

# 进入 /usr/local/nebula/logs 目录，查看日志文件。
cd /usr/local/nebula/logs
```

17.5.6 `host not found:nebula-<metad|storaged|graphd>-0.nebula.<metad|storaged|graphd>-headless.default.svc.cluster.local` 错误如何解决?

这个错误一般是由于 DNS 解析失败导致的，需检查是否修改了集群域名。如果修改了集群域名，需要同步修改 Operator 配置文件中的 `kubernetesClusterDomain` 字段。同步修改 Operator 配置文件的操作如下：

1. 查看 Operator 配置文件。

```
[abby@master ~]$ helm show values nebula-operator/nebula-operator
image:
  nebulaOperator:
    image: vesoft/nebula-operator:v1.8.0
    imagePullPolicy: Always
  kubeRBACProxy:
    image: bitnami/kube-rbac-proxy:0.14.2
    imagePullPolicy: Always
  kubeScheduler:
    image: registry.k8s.io/kube-scheduler:v1.24.11
    imagePullPolicy: Always

imagePullSecrets: []
kubernetesClusterDomain: "" # 集群域名，默认为 cluster.local。
```

2. 修改 `kubernetesClusterDomain` 字段的值为集群域名。

```
helm upgrade nebula-operator nebula-operator/nebula-operator --namespace=<nebula-operator-system> --version=1.8.0 --set kubernetesClusterDomain=<cluster-domain>
```

为 Operator 所在的命名空间，为更新后的集群域名。

最后更新: 2025年8月4日

18. 图计算

18.1 NebulaGraph Algorithm

NebulaGraph Algorithm （简称 Algorithm）是一款基于 GraphX 的 Spark 应用程序，通过提交 Spark 任务的形式使用完整的算法工具对 NebulaGraph 数据库中的数据执行图计算，也可以通过编程形式调用 lib 库下的算法针对 DataFrame 执行图计算。

18.1.1 版本兼容性

NebulaGraph Algorithm 版本和 NebulaGraph 内核的版本对应关系如下。

NebulaGraph 版本	NebulaGraph Algorithm 版本
nightly	3.0-SNAPSHOT
3.0.0 ~ 3.6.x	3.x.0
2.6.x	2.6.x
2.5.0、2.5.1	2.5.0
2.0.0、2.0.1	2.1.0

18.1.2 前提条件

在使用 Algorithm 之前，用户需要确认以下信息：

- NebulaGraph 服务已经部署并启动。详细信息，参考 NebulaGraph 安装部署。
- Spark 版本为 2.4.x。
- Scala 版本为 2.11。
- （可选）如果用户需要在 Github 中克隆最新的 Algorithm，并自行编译打包，可以选择安装 Maven。

18.1.3 使用限制

图计算会输出点的数据集，算法结果会以DataFrame形式作为点的属性存储。用户可以根据业务需求，自行对算法结果做进一步操作，例如统计、筛选。

> Compatibility
>
> Algorithm v3.1.0 版本之前，直接提交算法包时，点 ID 的数据必须为整数，即点 ID 可以是 INT 类型，或者是 String 类型但数据本身为整数。

18.1.4 支持算法

NebulaGraph Algorithm 支持的图计算算法如下。

算法名	说明	应用场景	属性名称	属性数据类型
PageRank	页面排序	网页排序、重点节点挖掘	pagerank	double/string
Louvain	鲁汶	社团挖掘、层次化聚类	louvain	int/string
KCore	K 核	社区发现、金融风控	kcore	int/string
LabelPropagation	标签传播	资讯传播、广告推荐、社区发现	lpa	int/string
Hanp	标签传播进阶版	社区发现、推荐	hanp	int/string
ConnectedComponent	弱联通分量	社区发现、孤岛发现	cc	int/string
StronglyConnectedComponent	强联通分量	社区发现	scc	int/string
ShortestPath	最短路径	路径规划、网络规划	shortestpath	string
TriangleCount	三角形计数	网络结构分析	trianglecount	int/string
GraphTriangleCount	全图三角形计数	网络结构及紧密程度分析	count	int
BetweennessCentrality	中介中心性	关键节点挖掘，节点影响力计算	betweenness	double/string
ClosenessCentrality	紧密中心性	关键节点挖掘、节点影响力计算	closeness	double/string
DegreeStatic	度统计	图结构分析	degree,inDegree,outDegree	int/string
ClusteringCoefficient	聚集系数	推荐、电信诈骗分析	clustercoefficient	double/string
Jaccard	杰卡德相似度计算	相似度计算、推荐	jaccard	string
BFS	广度优先遍历	层序遍历、最短路径规划	bfs	string
DFS	深度优先遍历	层序遍历、最短路径规划	dfs	string
Node2Vec	-	图分类	node2vec	string

Note

如果需要将算法结果写入到 NebulaGraph 中，请确保对应图空间中的 Tag 有和上表对应的属性名称和数据类型。

18.1.5 实现方法

NebulaGraph Algorithm 实现图计算的流程如下：

1. 利用 NebulaGraph Spark Connector 从 NebulaGraph 中读取图数据为 DataFrame。
2. 将 DataFrame 转换为 GraphX 的图。
3. 调用 GraphX 提供的图算法（例如 PageRank）或者自行实现的算法（例如 Louvain 社区发现）。

详细的实现方法可以参见相关 Scala 文件。

18.1.6 获取 NebulaGraph Algorithm

编译打包

1. 克隆仓库 `nebula-algorithm`。

```
$ git clone -b v3.0.0 https://github.com/vesoft-inc/nebula-algorithm.git
```

2. 进入目录 `nebula-algorithm`。

```
$ cd nebula-algorithm
```

3. 编译打包。

```
$ mvn clean package -Dgpg.skip -Dmaven.javadoc.skip=true -Dmaven.test.skip=true
```

编译完成后，在目录 `nebula-algorithm/target` 下生成类似文件 `nebula-algorithm-3.x.x.jar`。

Maven 远程仓库下载

下载地址

18.1.7 使用方法

> **Note**
>
> 如果数据的属性值包含中文字符，可能出现乱码。请在提交 Spark 任务时加上以下选项：
>
> ```
> --conf spark.driver.extraJavaOptions=-Dfile.encoding=utf-8
> --conf spark.executor.extraJavaOptions=-Dfile.encoding=utf-8
> ```

调用算法接口（推荐）

`lib` 库中提供了 10 种常用图计算算法，用户可以通过编程调用的形式调用算法。

1. 在文件 `pom.xml` 中添加依赖。

```
<dependency>
  <groupId>com.vesoft</groupId>
  <artifactId>nebula-algorithm</artifactId>
  <version>3.0.0</version>
</dependency>
```

2. 传入参数调用算法（以 PageRank 为例）。更多调用示例请参见示例。

> **Note**
>
> 执行算法的 DataFrame 默认第一列是起始点，第二列是目的点，第三列是边权重（非 NebulaGraph 中的 Rank）。

```
val prConfig = new PRConfig(5, 1.0)
val prResult = PageRankAlgo.apply(spark, data, prConfig, false)
```

如果用户的节点 ID 是 String 类型，可以参考 PageRank 的示例。示例中进行了 ID 转换，将 String 类型编码为 Long 类型， 并在算法结果中将 Long 类型 ID 解码为原始的 String 类型。

直接提交算法包

1. 设置配置文件。

```
{
  # Spark 相关配置
  spark: {
  app: {
    name: LPA
    # Spark 分片数量
    partitionNum:100
  }
  master:local
  }

  data: {
  # 数据源，可选值为 nebula、csv、json。
  source: nebula
  # 数据落库，即图计算的结果写入的目标，可选值为 nebula、csv、json。
  sink: nebula
  # 算法是否需要权重。
  hasWeight: false
  }

  # NebulaGraph 相关配置
  nebula: {
  # 数据源。 NebulaGraph 作为图计算的数据源时，nebula.read 的配置才生效。
  read: {
    # 所有 Meta 服务的 IP 地址和端口，多个地址用英文逗号（,）分隔。格式："ip1:port1,ip2:port2"。
    # 使用 docker-compose 部署，端口需要填写 docker-compose 映射到外部的端口
    # 可以用`docker-compose ps`查看
    metaAddress: "192.168.*.10:9559"
    # NebulaGraph 图空间名称
    space: basketballplayer
    # NebulaGraph Edge type, 多个 labels 时，多个边的数据将合并。
    labels: ["serve"]
    # NebulaGraph 每个 Edge type 的属性名称，此属性将作为算法的权重列，请确保和 Edge type 对应。
    weightCols: ["start_year"]
  }

  # 数据落库。图计算结果落库到 NebulaGraph 时，nebula.write 的配置才生效。
  write:{
    # Graph 服务的 IP 地址和端口，多个地址用英文逗号（,）分隔。格式："ip1:port1,ip2:port2"。
    # 使用 docker-compose 部署，端口需要填写 docker-compose 映射到外部的端口
    # 可以用`docker-compose ps`查看
    graphAddress: "192.168.*.11:9669"
    # 所有 Meta 服务的 IP 地址和端口，多个地址用英文逗号（,）分隔。格式："ip1:port1,ip2:port2"。
    # 使用 docker-compose 部署，端口需要填写 docker-compose 映射到外部的端口
    # 可以用`docker-compose ps`查看
    metaAddress: "192.168.*.12:9559"
    user:root
    pswd:nebula
    # 在提交图计算任务之前需要自行创建图空间及 Tag
    # NebulaGraph 图空间名称
    space:nb
    # NebulaGraph Tag 名称，图计算结果会写入该 Tag。Tag 中的属性名称固定如下：
    # PageRank：pagerank
    # Louvain：louvain
    # ConnectedComponent：cc
    # StronglyConnectedComponent：scc
    # LabelPropagation：lpa
    # ShortestPath：shortestpath
    # DegreeStatic：degree、inDegree、outDegree
    # KCore：kcore
    # TriangleCount：tranglecpunt
    # BetweennessCentrality：betweennedss
    tag:pagerank
  }
  }

  local: {
  # 数据源。图计算的数据源为 csv 文件或 json 文件时，local.read 的配置才生效。
  read:{
    filePath: "hdfs://127.0.0.1:9000/edge/work_for.csv"
    # 如果 CSV 文件没有表头，使用 [_c0, _c1, _c2, ..., _cn] 表示其表头，有表头或者是 json 文件时，直接使用表头名称即可。
    # 起始点 ID 列的表头。
    srcId:"_c0"
    # 目的点 ID 列的表头。
    dstId:"_c1"
    # 权重列的表头
    weight: "_c2"
    # csv 文件是否有表头
    header: false
    # csv 文件的分隔符
    delimiter:","
  }

  # 数据落库。图计算结果落库到 csv 文件或 text 文件时，local.write 的配置才生效。
  write:{
    resultPath:/tmp/
  }
```

```
  algorithm: {
  # 需要执行的算法，可选值为：
  # pagerank、louvain、connectedcomponent、labelpropagation、shortestpaths、
  # degreestatic、kcore、stronglyconnectedcomponent、trianglecount、
  # betweenness、graphtriangleCount。
  executeAlgo: pagerank

  # PageRank 参数
  pagerank: {
    maxIter: 10
    resetProb: 0.15
    encodeId:false # 如果点 ID 是字符串类型，请配置为 true。
  }

  # Louvain 参数
  louvain: {
    maxIter: 20
    internalIter: 10
    tol: 0.5
    encodeId:false # 如果点 ID 是字符串类型，请配置为 true。
  }

  # ...
  }
  }
}
```

> Note
>
> 当配置为 `sink: nebula` 的时候，意味着算法运算结果将被写回 NebulaGraph 集群，这对写回到的 TAG 中的属性名有隐含的约定。详情参考本文支持算法部分。

2. 提交图计算任务。

```
${SPARK_HOME}/bin/spark-submit --master <mode> --class com.vesoft.nebula.algorithm.Main <nebula-algorithm-3.0.0.jar_path> -p <application.conf_path>
```

示例：

```
${SPARK_HOME}/bin/spark-submit --master "local" --class com.vesoft.nebula.algorithm.Main /root/nebula-algorithm/target/nebula-algorithm-3.0-SNAPSHOT.jar -p /root/nebula-algorithm/src/main/resources/application.conf
```

18.1.8 视频

- 图计算工具——NebulaGraph Algorithm 介绍（2 分 36 秒）

最后更新: 2025年8月4日

19. NebulaGraph Bench

NebulaGraph Bench 是一款利用 LDBC 数据集对 NebulaGraph 进行性能测试的工具。

19.1 适用场景

- 生成测试数据并导入 NebulaGraph 。
- 对 NebulaGraph 集群进行性能测试。

19.2 更新说明

Release

详细使用说明请参见 NebulaGraph Bench。

最后更新: 2025年8月4日

20. 常见问题 FAQ

本文列出了使用 NebulaGraph master 时可能遇到的常见问题，用户可以使用文档中心或者浏览器的搜索功能查找相应问题。

如果按照文中的建议无法解决问题，请到 NebulaGraph 论坛 提问或提交 GitHub issue。

20.1 关于本手册

20.1.1 为什么手册示例和系统行为不一致?

NebulaGraph 一直在持续开发，功能或操作的行为可能会有变化，如果发现不一致，请提交 issue 通知 NebulaGraph 团队。

> Note
>
> 如果发现本文档中的错误：
>
> 1. 用户可以点击页面顶部右上角的"铅笔"图标进入编辑页面。
> 2. 使用 Markdown 修改文档。完成后点击页面底部的 "Commit changes"，这会触发一个 GitHub pull request。
> 3. 完成 CLA 签署，并且至少 2 位 reviewer 审核通过即可合并。

20.2 关于历史兼容性

> Compatibility
>
> NebulaGraph master 与 历史版本 （包括 NebulaGraph 1.x 和 2.x) 的数据格式、客户端通信协议均双向不兼容。 使用老版本客户端连接新版本服务端，会导致服务进程退出。 数据格式升级参见升级 NebulaGraph 历史版本至当前版本。 客户端与工具均需要下载对应版本。

20.3 关于执行报错

20.3.1 如何处理错误信息 `-1005:GraphMemoryExceeded: (-2600)`

这个错误是由 Memory Tracker 发出的，它在观察到内存使用超过了设定阈值时发出警报。这个机制可以帮助避免服务进程被系统的 OOM（Out of Memory）killer 终止。解决步骤：

1. 检查内存使用情况：首先，你需要查看命令执行过程中的内存用量。如果内存使用量确实很高，那么这个错误可能是预期中的情况。
2. 检查 Memory Tracker 的配置：如果内存使用量并不高，检查 Memory Tracker 的相关配置。这包括 `memory_tracker_untracked_reserved_memory_mb` （未被跟踪的预留内存量，单位为 MB），`memory_tracker_limit_ratio` （内存限制比例），以及 `memory_purge_enabled` （是否启用内存清理）。关于 Memory Tracker 的配置，参见 memory tracker 配置。
3. 优化配置：根据实际情况，调整这些配置。例如，如果可用内存限制过低，可以调高 `memory_tracker_limit_ratio` 的值。

20.3.2 如何处理错误信息 `SemanticError: Missing yield clause`

从 NebulaGraph 3.0.0 开始，查询语句 `LOOKUP`、`GO`、`FETCH` 必须用 `YIELD` 子句指定输出结果。详情请参见YIELD。

20.3.3 如何处理错误信息 `Host not enough!`

从 3.0.0 版本开始，在配置文件中添加的 Storage 节点无法直接读写，配置文件的作用仅仅是将 Storage 节点注册至 Meta 服务中。必须使用 `ADD HOSTS` 命令后，才能正常读写 Storage 节点。详情参见管理 Storage 主机。

20.3.4 如何处理错误信息 `To get the property of the vertex in 'v.age', should use the format 'var.tag.prop'`

从 3.0.0 版本开始，`pattern` 支持同时匹配多个 Tag，所以返回属性时，需要额外指定 Tag 名称。即从 `RETURN 变量名.属性名` 改为 `RETURN 变量名.Tag名.属性名`。

20.3.5 如何处理错误信息 `Used memory hits the high watermark(0.800000) of total system memory.`

报错原因： NebulaGraph 的 `system_memory_high_watermark_ratio` 参数指定了内存高水位报警机制的触发阈值，默认为 `0.8`。系统内存占用率高于该值会触发报警机制，NebulaGraph 会停止接受查询。

解决方案:

- 清理系统内存，使其降低到阈值以下。
- 修改 Graph 配置。在所有 Graph 服务器的配置文件中增加 `system_memory_high_watermark_ratio` 参数，为其设置一个大于 `0.8` 的值，例如 `0.9`。

需要注意的是 `system_memory_high_watermark_ratio` 参数已被弃用。建议用户使用 Memory Tracker 功能来限制NebulaGraph的内存使用量。详情请参见 Graph 服务的 memory tracker 配置和 Storage 服务的 memory tracker 配置。

20.3.6 如何处理错误信息 `Storage Error E_RPC_FAILURE`

报错原因通常为 Graph 服务向 Storage 服务请求了过多的数据，导致 Storage 服务超时。请尝试以下解决方案:

- 修改配置文件：在 `nebula-graphd.conf` 文件中修改 `--storage_client_timeout_ms` 参数的值，以增加 Storage client 的连接超时时间。该值的单位为毫秒（ms）。例如，设置 `--storage_client_timeout_ms=60000`。如果 `nebula-graphd.conf` 文件中未配置该参数，请手动增加。提示：请在配置文件开头添加 `--local_config=true` 再重启服务。
- 优化查询语句：减少全库扫描型的查询，无论是否用 `LIMIT` 限制了返回结果的数量；用 GO 语句改写 MATCH 语句（前者有优化，后者无优化）。
- 检查 Storaged 是否发生过 OOM。(`dmesg |grep nebula`)。
- 为 Storage 服务器提供性能更好的 SSD 或者内存。
- 重试请求。

20.3.7 如何处理错误信息 `The leader has changed. Try again later`

已知问题，通常需要重试 1-N 次 (N==partition 数量）。原因为 meta client 更新 leader 缓存需要 1-2 个心跳或者通过错误触发强制更新。

如果登录 NebulaGraph 时，出现该报错，可以考虑使用 `df -h` 查看磁盘空间，检查本地磁盘空间是否已满。

20.3.8 Schema not exist: xxx

查询时提示 `Schema not exist`，请确认:

- Schema 中是否存在该 Tag 或 Edge type。
- Tag 或 Edge type 的名称是否为关键字，如果是关键字，请使用反引号（`）将它们括起来。详情请参见关键字。

20.3.9 编译 Exchange、Connectors、Algorithm 时无法下载 SNAPSHOT 包

现象：编译时提示 `Could not find artifact com.vesoft:client:jar:xxx-SNAPSHOT`。

原因：本地 maven 没有配置用于下载 SNAPSHOT 的仓库。maven 中默认的 central 仓库用于存放正式发布版本，而不是开发版本（SNAPSHOT）。

解决方案：在 maven 的 setting.xml文件的 `profiles` 作用域内中增加以下配置:

```
<profile>
   <activation>
      <activeByDefault>true</activeByDefault>
   </activation>
   <repositories>
      <repository>
          <id>snapshots</id>
          <url>https://oss.sonatype.org/content/repositories/snapshots/</url>
          <snapshots>
```

```
                <enabled>true</enabled>
              </snapshots>
          </repository>
        </repositories>
    </profile>
```

20.3.10 如何处理错误信息 `[ERROR (-1004)]: SyntaxError: syntax error near`？

大部分情况下，查询语句需要有 `YIELD` 或 `RETURN`，请检查查询语句是否包含。

20.3.11 如何处理错误信息 `can't solve the start vids from the sentence`

查询引擎需要知道从哪些 VID 开始图遍历。这些开始图遍历的 VID，或者通过用户指定，例如：

```
> GO FROM ${vids} ...
> MATCH (src) WHERE id(src) == ${vids}
# 开始图遍历的 VID 通过如上办法指定
```

或者通过一个属性索引来得到，例如：

```
# CREATE TAG INDEX IF NOT EXISTS i_player ON player(name(20));
# REBUILD TAG INDEX i_player;

> LOOKUP ON player WHERE player.name == "abc" | ... YIELD ...
> MATCH (src) WHERE src.name == "abc" ...
# 通过点属性 name 的索引，来得到 VID
```

否则，就会抛出这样一个异常 `can't solve the start vids from the sentence`。

20.3.12 如何处理错误信息 `Wrong vertex id type: 1001`

检查输入的 VID 类型是否是 `create space` 设置的 `INT64` 或 `FIXED_STRING(N)`。详情请参见 create space。

20.3.13 如何处理错误信息 `The VID must be a 64-bit integer or a string fitting space vertex id length limit.`

检查输入的 VID 是否超过限制长度。详情请参见 create space。

20.3.14 如何处理错误信息 `edge conflict` 或 `vertex conflict`

Storage 服务在毫秒级时间内多次收到插入或者更新同一点或边的请求时，可能返回该错误。请稍后重试。

20.3.15 如何处理错误信息 `RPC failure in MetaClient: Connection refused`

报错原因通常为 metad 服务状态异常，或是 metad 和 graphd 服务所在机器网络不通。请尝试以下解决方案：

- 在 metad 所在服务器查看下 metad 服务状态，如果服务状态异常，可以重新启动 metad 服务。
- 在报错服务器下使用 `telnet meta-ip:port` 查看网络状态。
- 检查配置文件中的端口配置，如果端口号与连接时使用的不同，改用配置文件中的端口或者修改配置。

20.3.16 如何处理 `nebula-graph.INFO` 中错误日志 `StorageClientBase.inl:214] Request to "x.x.x.x":9779 failed: N6apache6thrift9transport19TTransportExceptionE: Timed Out`

报错原因可能是查询的数据量比较大，storaged 处理超时。请尝试以下解决方法：

- 导入数据时，手动 compaction，加速读的速度。
- 增加 Graph 服务与 Storage 服务的 RPC 连接超时时间，在 `nebula-graphd.conf` 文件里面修改 `--storage_client_timeout_ms` 参数的值。该值的单位为毫秒（ms），默认值为 60000 毫秒。

20.3.17 如何处理 `nebula-storaged.INFO` 中错误日志 `MetaClient.cpp:65] Heartbeat failed, status:Wrong cluster!` 或者 `nebula-metad.INFO` 含有错误日志 `HBProcessor.cpp:54] Reject wrong cluster host "x.x.x.x":9771!`

报错的原因可能是用户修改了 metad 的 ip 或者端口信息，或者 storage 之前加入过其他集群。请尝试以下解决方法：

用户到 storage 部署的机器所在的安装目录（默认安装目录为 `/usr/local/nebula`）下面将 `cluster.id` 文件删除，然后重启 storaged 服务。

20.3.18 如何处理报错信息 `Storage Error: More than one request trying to add/update/delete one edge/vertex at he same time.`

上述报错表示当前版本不允许同时对一个点或者一条边进行并发操作。如出现此报错请重新执行操作命令。

20.3.19 如何处理错误信息 `The maximum number of statements allowed has been exceeded` 和 `the maximum number of statements for Nebula is 512`

这两个报错信息都是由于单条插入语句长度超过 `max_allowed_statements` 引起的，通过修改 `nebula-graphd.conf` 配置文件的 `etc` 目录下 `--max_allowed_statements` 参数项，能避免语句插入报错。

20.4 关于设计与功能

20.4.1 返回消息中 `time spent` 的含义是什么？

将命令 `SHOW SPACES` 返回的消息作为示例：

```
nebula> SHOW SPACES;
+--------------------+
| Name               |
+--------------------+
| "basketballplayer" |
+--------------------+
Got 1 rows (time spent 1235/1934 us)
```

- 第一个数字 `1235` 表示数据库本身执行该命令花费的时间，即查询引擎从客户端接收到一个查询，然后从存储服务器获取数据并执行一系列计算所花费的时间。
- 第二个数字 `1934` 表示从客户端角度看所花费的时间，即从客户端发送请求、接收结果，然后在屏幕上显示结果所花费的时间。

20.4.2 为什么在正常连接 NebulaGraph 后，`nebula-storaged` 进程的端口号一直显示红色？

`nebula-storaged` 进程的端口号的红色闪烁状态是因为 `nebula-storaged` 在启动流程中会等待 `nebula-metad` 添加当前 Storage 服务，当前 Storage 服务收到 Ready 信号后才会正式启动服务。从 3.0.0 版本开始，Meta 服务无法直接读写在配置文件中添加的 Storage 服务，配置文件的作用仅仅是将 Storage 服务注册至 Meta 服务中。用户必须使用 `ADD HOSTS` 命令后，才能使 Meta 服务正常读写 Storage 服务。更多信息，参见管理 Storage 主机。

20.4.3 为什么 NebulaGraph 的返回结果每行之间没有横线分隔了？

这是 NebulaGraph Console 2.6.0 版本的变动造成的，不是 NebulaGraph 内核的变更，不影响返回数据本身的内容。

20.4.4 关于悬挂边

悬挂边 (Dangling edge) 是指一条边的起点或者终点在数据库中不存在。

NebulaGraph master 的数据模型中，由于设计允许图中存在“悬挂边”； 没有 openCypher 中的 MERGE 语句。 对于悬挂边的保证完全依赖应用层面。 详见 INSERT VERTEX, DELETE VERTEX, INSERT EDGE, DELETE EDGE。

20.4.5 可以在 `CREATE SPACE` 时设置 `replica_factor` 为偶数（例如设置为 2）吗？

不要这样设置。

Storage 服务使用 Raft 协议（多数表决），为保证可用性，要求出故障的副本数量不能达到一半。

当机器数量为 1 时，`replica_factor` 只能设置为 `1`。

当机器数量足够时，如果 `replica_factor=2`，当其中一个副本故障时，就会导致系统无法正常工作；如果 `replica_factor=4`，只能有一个副本可以出现故障，这和 `replica_factor=3` 是一样。以此类推，所以 `replica_factor` 设置为奇数即可。

建议在生产环境中设置 `replica_factor=3`，测试环境中设置 `replica_factor=1`，不要使用偶数。

20.4.6 是否支持停止或者中断慢查询

支持。

详情请参见终止查询。

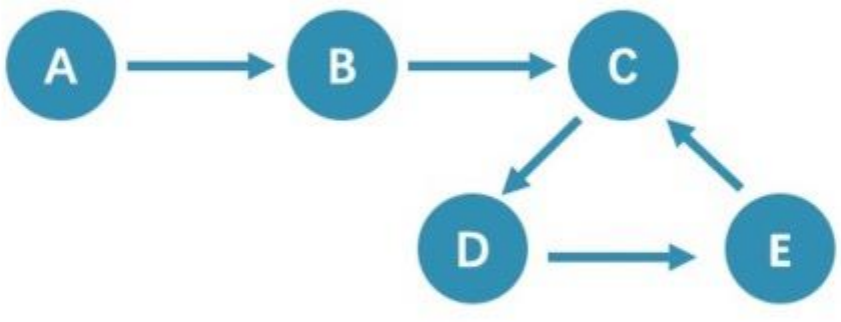
20.4.7 使用 `GO` 和 `MATCH` 执行相同语义的查询，查询结果为什么不同?

原因可能有以下几种:

- `GO` 查询到了悬挂边。
- `RETURN` 命令未指定排序方式。
- 触发了 Storage 服务中 `max_edge_returned_per_vertex` 定义的稠密点截断限制。

- 路径的类型不同，导致查询结果可能会不同。
- `GO` 语句采用的是 `walk` 类型，遍历时点和边可以重复。
- `MATCH` 语句兼容 openCypher，采用的是 `trail` 类型，遍历时只有点可以重复，边不可以重复。

因路径类型不同导致查询结果不同的示例图和说明如下。



从点 A 开始查询距离 5 跳的点，都会查询到点 C（`A->B->C->D->E->C`），查询 6 跳的点时，`GO` 语句会查询到点 D（`A->B->C->D->E->C->D`），因为边 `C->D` 可以重复查询，而 `MATCH` 语句查询为空，因为边不可以重复。

所以使用 `GO` 和 `MATCH` 执行相同语义的查询，可能会出现 `MATCH` 语句的查询结果比 `GO` 语句少。

关于路径的详细说明，请参见维基百科。

20.4.8 如何统计每种 Tag 有多少个点，每个 Edge type 有多少条边?

请参见 show-stats。

20.4.9 如何获取每种 Tag 的所有点，或者每种 Edge type 的所有边?

1. 建立并重建索引。

```
> CREATE TAG INDEX IF NOT EXISTS i_player ON player();
> REBUILD TAG INDEX i_player;
```

2. 使用 `LOOKUP` 或 `MATCH` 语句。例如：

```
> LOOKUP ON player;
> MATCH (n:player) RETURN n;
```

更多详情请参见 INDEX、LOOKUP 和 MATCH。

20.4.10 能不能用中文字符做标识符，比如图空间、Tag、Edge type、属性、索引的名称?

能，详情参见关键字和保留字。

20.4.11 获取指定点的出度（或者入度）？

一个点的“出度”是指从该点出发的“边”的条数。入度，是指指向该点的边的条数。

```
nebula > MATCH (s)-[e]->() WHERE id(s) == "given" RETURN count(e); #出度
nebula > MATCH (s)<-[e]-() WHERE id(s) == "given" RETURN count(e); #入度
```

因为没有对出度和入度的进行特殊处理（例如索引或者缓存），这是一个较慢的操作。特别当遇到超级节点时，可能会发生OOM。

20.4.12 是否有办法快速获取“所有”点的出度和入度?

没有直接命令。

可以使用 NebulaGraph Algorithm。

20.5 关于运维

20.5.1 运行日志文件过大时如何回收日志?

NebulaGraph使用 glog 打印日志，glog 没有日志回收的功能。用户可以通过定时任务或日志管理工具 logrotate 来管理运行日志。操作详情，参见回收日志。

20.5.2 如何查看 NebulaGraph 版本

服务运行时：`nebula-console` 中执行命令 `SHOW HOSTS META`，详见 SHOW HOSTS

服务未运行时：在安装路径的 `bin` 目录内，执行 `./<binary_name> --version` 命令，可以查看到 version 和 GitHub 上的 commit ID，例如：

```
$ ./nebula-graphd --version
```

- Docker Compose 部署

  查看 Docker Compose 部署的 NebulaGraph 版本，方式和编译安装类似，只是要先进入容器内部，示例命令如下：

  ```
  docker exec -it nebula-docker-compose_graphd_1 bash
  cd bin/
  ./nebula-graphd --version
  ```

- RPM/DEB 包安装

  执行 `rpm -qa |grep nebula` 即可查看版本。

20.5.3 修改 Host 名称后，旧的 Host 一直显示 `OFFLINE` 怎么办?

`OFFLINE` 状态的 Host 将在一天后自动删除。

20.5.4 如何查看 dmp 文件?

dmp 文件是错误报告文件，详细记录了进程退出的信息，可以用操作系统自带的 gdb 工具查看。Coredump 文件默认保存在启动二进制的当前目录下（默认为 `/usr/local/nebula` 目录），在 NebulaGraph 服务 crash 时，系统会自动生成。

1. 查看 Core 文件进程名字，pid 一般为数值。

```
$ file core.<pid>
```

2. 使用 gdb 调试。

```
$ gdb <process.name> core.<pid>
```

3. 查看文件内容。

```
$(gdb) bt
```

例如：

```
$ file core.1316027
core.1316027: ELF 64-bit LSB core file, x86-64, version 1 (SYSV), SVR4-style, from '/home/workspace/fork/nebula-debug/bin/nebula-metad --flagfile /home/k', real uid: 1008, effective uid: 1008, real gid: 1008, effective gid: 1008, execfn: '/home/workspace/fork/nebula-debug/bin/nebula-metad', platform: 'x86_64'

$ gdb /home/workspace/fork/nebula-debug/bin/nebula-metad core.1316027

$(gdb) bt
#0  0x00007f9de58fecf5 in __memcpy_ssse3_back () from /lib64/libc.so.6
#1  0x0000000000eb2299 in void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::_M_construct<char*>(char*, char*, std::forward_iterator_tag) ()
#2  0x0000000000ef71a7 in nebula::meta::cpp2::QueryDesc::QueryDesc(nebula::meta::cpp2::QueryDesc const&) ()
...
```

如果用户不清楚 dmp 打印出来的相关信息，可以将打印出来的内容，带上操作系统版本、硬件配置、Core文件产生前后的错误日志和可能导致错误的操作贴在 NebulaGraph 论坛上寻求帮助。

20.5.5 如何通过 systemctl 设置开机自动启动 NebulaGraph 服务?

1. 执行 `systemctl enable` 启动 metad、graphd 和 storaged 服务。

```
[root]# systemctl enable nebula-metad.service
Created symlink from /etc/systemd/system/multi-user.target.wants/nebula-metad.service to /usr/lib/systemd/system/nebula-metad.service.
[root]# systemctl enable nebula-graphd.service
Created symlink from /etc/systemd/system/multi-user.target.wants/nebula-graphd.service to /usr/lib/systemd/system/nebula-graphd.service.
[root]# systemctl enable nebula-storaged.service
Created symlink from /etc/systemd/system/multi-user.target.wants/nebula-storaged.service to /usr/lib/systemd/system/nebula-storaged.service.
```

2. 配置 metad、graphd 和 storaged 的 service 文件，设置服务自动拉起。

Caution

在配置service文件时需注意以下几点： - PIDFile、ExecStart、ExecReload、ExecStop 参数的路径需要与服务器上的一致。 - RestartSec 为重启前等待的时长(秒)，可根据实际情况修改。 - （可选）StartLimitInterval 为无限次重启，默认是 10 秒内如果重启超过 5 次则不再重启，设置为 0 表示不限次数重启。 - （可选）LimitNOFILE 为服务最大文件打开数，默认为1024，可根据实际情况修改。

配置 metad 服务的 service 文件：

```
$ vi /usr/lib/systemd/system/nebula-metad.service

[Unit]
Description=Nebula Graph Metad Service
After=network.target

[Service ]
Type=forking
Restart=always
RestartSec=15s
PIDFile=/usr/local/nebula/pids/nebula-metad.pid
ExecStart=/usr/local/nebula/scripts/nebula.service start metad
ExecReload=/usr/local/nebula/scripts/nebula.service restart metad
ExecStop=/usr/local/nebula/scripts/nebula.service stop metad
PrivateTmp=true
StartLimitInterval=0
LimitNOFILE=1024
```

```
[Install]
WantedBy=multi-user.target
```

配置 graphd 服务的 service 文件：

```
$ vi /usr/lib/systemd/system/nebula-graphd.service
[Unit]
Description=Nebula Graph Graphd Service
After=network.target

[Service]
Type=forking
Restart=always
RestartSec=15s
PIDFile=/usr/local/nebula/pids/nebula-graphd.pid
ExecStart=/usr/local/nebula/scripts/nebula.service start graphd
ExecReload=/usr/local/nebula/scripts/nebula.service restart graphd
ExecStop=/usr/local/nebula/scripts/nebula.service stop graphd
PrivateTmp=true
StartLimitInterval=0
LimitNOFILE=1024

[Install]
WantedBy=multi-user.target
```

配置 storaged 服务的 service 文件：

```
$ vi /usr/lib/systemd/system/nebula-storaged.service
[Unit]
Description=Nebula Graph Storaged Service
After=network.target

[Service]
Type=forking
Restart=always
RestartSec=15s
PIDFile=/usr/local/nebula/pids/nebula-storaged.pid
ExecStart=/usr/local/nebula/scripts/nebula.service start storaged
ExecReload=/usr/local/nebula/scripts/nebula.service restart storaged
ExecStop=/usr/local/nebula/scripts/nebula.service stop storaged
PrivateTmp=true
StartLimitInterval=0
LimitNOFILE=1024

[Install]
WantedBy=multi-user.target
```

3. reload 配置文件。

```
[root]# sudo systemctl daemon-reload
```

4. 重启服务。

```
$ systemctl restart nebula-metad.service
$ systemctl restart nebula-graphd.service
$ systemctl restart nebula-storaged.service
```

20.6 关于连接

20.6.1 防火墙中需要开放哪些端口？

如果没有修改过配置文件中预设的端口，请在防火墙中开放如下端口：

服务类型	端口
Meta	9559, 9560, 19559
Graph	9669, 19669
Storage	9777 ~ 9780, 19779

如果修改过配置文件中预设的端口，请找出实际使用的端口并在防火墙中开放它们。

更多端口信息，参见公司产品端口全集。

20.6.2 如何测试端口是否已开放?

用户可以使用如下 telnet 命令检查端口状态:

```
telnet <ip> <port>
```

> Note
>
> 如果无法使用 telnet 命令,请先检查主机中是否安装并启动了 telnet。

示例:

```
// 如果端口已开放:
$ telnet 192.168.1.10 9669
Trying 192.168.1.10...
Connected to 192.168.1.10.
Escape character is '^]'.

// 如果端口未开放:
$ telnet 192.168.1.10 9777
Trying 192.168.1.10...
telnet: connect to address 192.168.1.10: Connection refused
```

20.6.3 会话数量达到 `max_sessions_per_ip_per_user` 所设值上限时怎么办?

当会话数量达到上限时,即超过 `max_sessions_per_ip_per_user` 所设值,在连接 NebulaGraph 的时候,会出现以下报错:

```
Fail to create a new session from connection pool, fail to authenticate, error: Create Session failed: Too many sessions created from 127.0.0.1 by user root. the threshold is 2. You can change it by modifying 'max_sessions_per_ip_per_user' in nebula-graphd.conf
```

用户可以通过以下三种方式解决问题:

1. 执行 `KILL SESSION` 命令,关闭不需要的 Session。详情参见终止会话。
2. 在 Graph 服务的配置文件 `nebula_graphd.conf` 中增大 `max_sessions_per_ip_per_user` 的值,增加可以创建的会话总数。重启 Graph 服务使配置生效。
3. 在 Graph 服务的配置文件 `nebula_graphd.conf` 中减小 `session_idle_timeout_secs` 参数的值,降低空闲会话的超时时间。重启 Graph 服务使配置生效。

20.7 关于扩容、缩容

> Warning
>
> 集群扩缩容功能未正式在社区版中发布。以下涉及 `SUBMIT JOB BALANCE DATA REMOVE` 和 `SUBMIT JOB BALANCE DATA` 的操作在社区版中均为实验性功能,功能未稳定。如需使用,请先做好数据备份,并且在 Graph 配置文件中将 `enable_experimental_feature` 和 `enable_data_balance` 均设置为 `true`。

20.7.1 如何增加或减少 Meta、Graph、Storage 节点的数量

- NebulaGraph master 未提供运维命令以实现自动扩缩容，但可参考以下步骤实现手动扩缩容：
- Meta 的扩容和缩容： Meta 不支持自动扩缩容。

> Note
>
> 用户可以使用脚本工具 迁移 Meta 服务，但是需要自行修改 Graph 服务和 Storage 服务的配置文件中的 Meta 设置。

- Graph 的缩容：将该 Graph 的 IP 从 client 的代码中移除，关闭该 Graph 进程。
- Graph 的扩容：在新机器上准备 Graph 二进制文件和配置文件，在配置文件中修改或增加已在运行的 Meta 地址，启动 Graph 进程。
- Storage 的缩容：执行 `SUBMIT JOB BALANCE DATA REMOVE <ip:port>` 和 `DROP HOSTS <ip:port>` 。完成后关闭 Storage 进程。

> Caution
>
> - 执行该命令迁移指定 Storage 节点中的分片数据前，需要确保其他 Storage 节点数量足够以满足设置的副本数。例如，如果设置了副本数为 3，那么在执行该命令前，需要确保其他 Storage 节点数量大于等于 3。
> - 如果待迁移的 Storage 节点中有多个图空间的分片，需要分别在每个图空间内执行该命令以迁移该 Storage 节点中的所有分片。

- Storage 的扩容：在新机器上准备 Storage 二进制文件和配置文件，在配置文件中修改或增加已在运行的 Meta 地址，然后注册 Storage 到 Meta 并启动 Storage 进程。详情参见注册 Storage 服务。

  Storage 扩缩容之后，根据需要执行 `SUBMIT JOB BALANCE DATA` 将当前图空间的分片平均分配到所有 Storage 节点中和执行 `SUBMIT JOB BALANCE LEADER` 命令均衡分布所有图空间中的 leader。运行命令前，需要选择一个图空间。

20.7.2 如何在 Storage 节点中增加或减少磁盘

目前，Storage 不支持动态识别新的磁盘。可以通过以下步骤在分布式集群的 Storage 节点中增加或减少磁盘（不支持对单个 Storage 节点的集群增加或减少磁盘）：

1. 执行 `SUBMIT JOB BALANCE DATA REMOVE <ip:port>` 将待增加或减少磁盘的 Storage 节点迁到其他 Storage 节点中。

> Caution
>
> - 执行该命令迁移指定 Storage 节点中的分片数据前，需要确保其他 Storage 节点数量足够以满足设置的副本数。例如，如果设置了副本数为 3，那么在执行该命令前，需要确保其他 Storage 节点数量大于等于 3。
> - 如果待迁移的 Storage 节点中有多个图空间的分片，需要分别在每个图空间内执行该命令以迁移该 Storage 节点中的所有分片。

2. 执行 `DROP HOSTS <ip:port>` 移除待增加或减少磁盘的 Storage 节点。
3. 在所有 Storage 节点配置文件中通过 `--data_path` 配置新增或减少磁盘路径，详情参见 Storage 配置文件。
4. 执行 `ADD HOSTS <ip:port>` 重新添加待增加或减少磁盘的 Storage 节点。
5. 根据需要执行 `SUBMIT JOB BALANCE DATA` 将当前图空间的分片平均分配到所有 Storage 节点中和执行 `SUBMIT JOB BALANCE LEADER` 命令均衡分布所有图空间中的 leader。运行命令前，需要选择一个图空间。

最后更新: 2025年8月4日

21. 附录

21.1 更新说明

21.1.1 NebulaGraph 更新说明

v3.8.0

- 功能：
- 现已支持 SINGLE SHORTEST PATH 功能。 #5664
- 已实现 INNER JOIN 功能。 #5664
- ROUND() 函数现已支持舍入模式。#5680

- 增强：
- 性能：
- SHORTEST PATH 现已支持 LIMIT 下推，以提升性能。 #5657
- 优化了部分逻辑，以避免 follower 宕机后对写性能的影响。 #5673
- 优化了 meta service 对 session 的管理，以减少高并发场景下的延迟。 #5762

- 易用性：
- 优化了删除 graph space 的过程，减少了阻塞时间。 #5754

- 稳定性：
- 优化了 LEADER BALANCE 算法，以实现更均衡的负载分配。 #5670
- 增加了对最大语句条数的限制，以增强系统的保护机制。 #5790

- 缺陷修复：
- DQL：
- 修复了多次执行 LOOKUP 语句时，结果不一致的问题。 #5662
- 修复了 UNION ALL 报语法错误的问题。 #5674
- 修复了在 SHORTEST PATH、ALL PATH、NOLOOP PATH 场景下 LIMIT 结果错误、崩溃等问题。 #5679、#5699、#5787、#5789
- 修复了在设置 memory tracker 的情况下，多次重复执行 SHORTEST PATH 导致的崩溃问题。#5720
- 修复了 Filter 错误，避免 Graph Service 崩溃。 #5740
- 修复了多变量场景下执行失败的问题。 #5734
- 修复了 MATCH SHORTEST PATH 不支持自环检测的问题。 #5738
- 修复了在过滤条件永远不成立时，部分场景下的崩溃问题。 #5740
- 修复了 ROUND 函数的崩溃问题。 #5773
- 修复了 FIND PATH WITH PROP 在一跳查询时，结果错误的问题。 #5759
- 修复了在执行 USE SPACE + 查询时，性能变慢的问题。 #5793
- 修复了 FIND NOLOOP PATH 未排除自环的问题。 #5805

- 其他：
- 修复了在执行 CLONE SPACE 时的错误。 #3005、#5781
- 修复了在存在索引时，num_vertices_inserted 监控指标无数据的问题。 #5756
- 修复了在进行查询和 Schema 的变更同时进行时可能出现崩溃的问题。 #5855

最后更新: 2025年8月4日

21.1.2 NebulaGraph Studio 更新说明

v3.10.0（2024.5）

- 增强
- 调整 AI Assistant 默认最大上下文长度。
- 调整部分文案与样式。

- 缺陷修复
- 修复导入模板的示例文件中的错误。
- 修复创建 Tag 或 Edge type 时默认值为非 NULL 的问题。
- 修复在导入任务页面编辑大量属性时页面崩溃的问题。

v3.9.1（2024.2）

- 缺陷修复
- 修复 Schema 草图中属性值使用参数 `NOT NULL` 的问题，替换为默认参数 `NULL` 。

v3.9.0（2024.1）

- 功能
- 支持使用 AI 导入数据。
- 支持使用 AI 查询语句助手生成 nGQL 语句。

- 增强
- 优化控制台功能，包括自动补全 Tag/Edge type、快速选择历史语句、快速查看 Schema 等。
- 优化执行计划展示样式。

最后更新: 2025年8月4日

21.1.3 NebulaGraph Dashboard 社区版更新说明

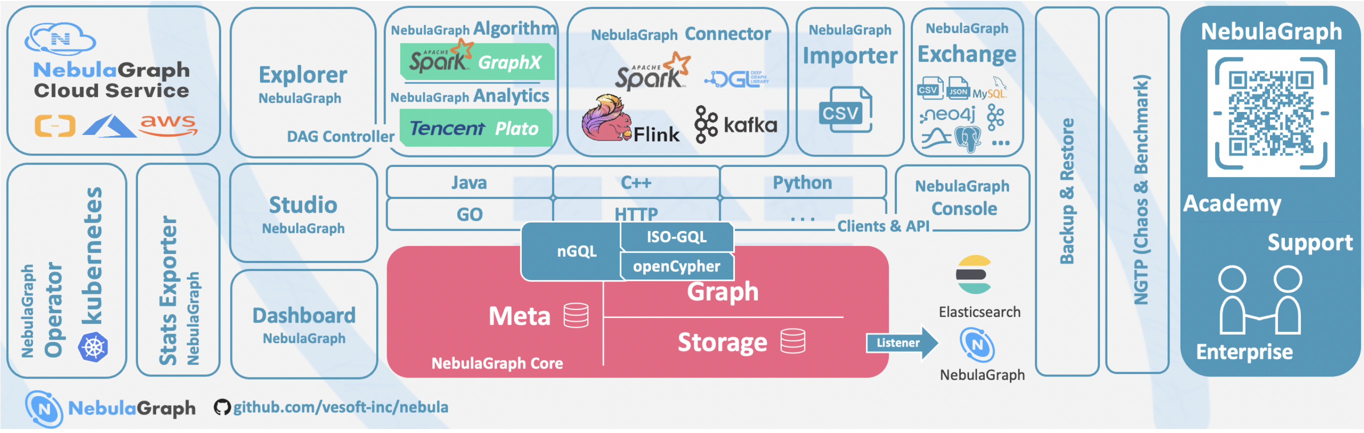
社区版 **v3.4.0**

- 功能
- 内置 dashboard.service 脚本，支持一键管理 Dashboard 服务和查看 Dashboard 版本。
- 支持查看 Meta 服务的配置。

- 优化
- 调整目录结构，简化部署步骤。
- 机器的概览页面显示监控指标名称。
- 优化 `num_queries` 等监控指标的计算方式，调整为时序聚合显示。

最后更新: 2025年8月4日

21.2 生态工具概览



21.2.1 NebulaGraph Studio

NebulaGraph Studio（简称 Studio）是一款可以通过 Web 访问的图数据库可视化工具，搭配 NebulaGraph 使用，提供构图、数据导入、编写 nGQL 查询、图探索等一站式服务。详情请参见什么是 NebulaGraph Studio。

> Note
>
> Studio 版本发布节奏独立于 NebulaGraph 内核，其命名方式也不参照内核命名规则，两者兼容对应关系如下表。

NebulaGraph 版本	Studio 版本
master	v3.10.0

21.2.2 NebulaGraph Dashboard（社区版）

NebulaGraph Dashboard（简称 Dashboard）是一款用于监控NebulaGraph集群中机器和服务状态的可视化工具。详情参见什么是 NebulaGraph Dashboard。

NebulaGraph版本	Dashboard 社区版本
master	v3.4.0

21.2.3 NebulaGraph Exchange

NebulaGraph Exchange（简称 Exchange）是一款 Apache Spark™ 应用，用于在分布式环境中将集群中的数据批量迁移到 NebulaGraph 中，能支持多种不同格式的批式数据和流式数据的迁移。详情请参见什么是 NebulaGraph Exchange。

NebulaGraph 版本	Exchange 版本
master	v3.8.0

21.2.4 NebulaGraph Operator

NebulaGraph Operator（简称 Operator）是用于在 Kubernetes 系统上自动化部署和运维 NebulaGraph 集群的工具。依托于 Kubernetes 扩展机制，NebulaGraph 将其运维领域的知识全面注入至 Kubernetes 系统中，让 NebulaGraph 成为真正的云原生图数据库。详情请参考什么是 NebulaGraph Operator。

NebulaGraph 版本	Operator 版本
master	v1.8.0

21.2.5 NebulaGraph Importer

NebulaGraph Importer（简称 Importer）是一款 NebulaGraph 的 CSV 文件导入工具。Importer 可以读取本地的 CSV 文件，然后导入数据至 NebulaGraph 中。详情请参见什么是 NebulaGraph Importer。

NebulaGraph 版本	Importer 版本
master	v4.1.0

21.2.6 NebulaGraph Spark Connector

NebulaGraph Spark Connector 是一个 Spark 连接器，提供通过 Spark 标准形式读写NebulaGraph数据的能力。NebulaGraph Spark Connector 由 Reader 和 Writer 两部分组成。详情请参见什么是 NebulaGraph Spark Connector。

NebulaGraph 版本	Spark Connector 版本
master	v3.8.0

21.2.7 NebulaGraph Flink Connector

NebulaGraph Flink Connector 是一款帮助 Flink 用户快速访问NebulaGraph的连接器，支持从NebulaGraph中读取数据，或者将其他外部数据源读取的数据写入NebulaGraph。详情请参见什么是 NebulaGraph Flink Connector。

NebulaGraph 版本	Flink Connector 版本
master	v3.8.0

21.2.8 NebulaGraph Algorithm

NebulaGraph Algorithm（简称 Algorithm）是一款基于 GraphX 的 Spark 应用程序，通过提交 Spark 任务的形式使用完整的算法工具对 NebulaGraph 数据库中的数据执行图计算，也可以通过编程形式调用 lib 库下的算法针对 DataFrame 执行图计算。详情请参见什么是 NebulaGraph Algorithm。

NebulaGraph 版本	Algorithm 版本
master	v3.0.0

21.2.9 NebulaGraph Console

NebulaGraph Console 是 NebulaGraph 的原生 CLI 客户端。如何使用请参见 NebulaGraph Console。

NebulaGraph 版本	Console 版本
master	v3.6.0

21.2.10 NebulaGraph Docker Compose

Docker Compose 可以快速部署 NebulaGraph 集群。如何使用请参见 Docker Compose 部署 NebulaGraph。

NebulaGraph 版本	Docker Compose 版本
master	v3.6.0

21.2.11 Backup & Restore

Backup&Restore（简称 BR）是一款命令行界面（CLI）工具，可以帮助备份 NebulaGraph 的图空间数据，或者通过备份文件恢复数据。

NebulaGraph 版本	BR 版本
master	v3.6.0

21.2.12 NebulaGraph Bench

NebulaGraph Bench 用于测试 NebulaGraph 的基线性能数据，使用 LDBC v0.3.3 的标准数据集。

NebulaGraph 版本	Bench 版本
master	v1.2.0

21.2.13 API、SDK

Compatibility

选择与内核版本相同 `X.Y.*` 的最新版本。

NebulaGraph 版本	语言（commit id）
master	C++
master	Go
master	Python
master	Java
master	HTTP

21.2.14 社区贡献的工具

这里罗列社区用户贡献的实用工具，各项目由其发起人及工具爱好者共同维护。

- ORM（Object Relational Mapping）框架
- NGBATIS：对接 Spring Boot 生态的数据库 ORM 框架
- graph-ocean：基于 nebula-java 客户端的 ORM，详细介绍参见文稿《隐藏在 graph-ocean 背后的星辰大海》
- nebula-jdbc：对接 JDBC 的 nebula-java 衍生品，详细介绍参见文稿《NebulaGraph 支持 JDBC 协议》
- nebula-carina：基于 nebula-python 客户端的 ORM，详细介绍参见文稿《NebulaGraph ORM 项目 Carina 简化 Web、AI 开发》
- norm：采用 Golang 编写的 ORM 项目，详细介绍参见文稿《Norm 知乎开源的 ORM 利器》

- 数据工具
- nebula-real-time-exchange：支持 MySQL 到 NebulaGraph 的数据实时同步功能
- nebula-datax-plugin：基于 DataX 实现的 NebulaGraph 的 Reader 和 Writer 插件，可方便用户离线实现数据同步

- 轻便部署
- nebulagraph-docker-ext：10s 拉起 Docker 图库服务
- nebulagraph-lite：一个运行在浏览器的 NebulaGraph 沙盒

- 测试服务
- testcontainers-nebula：轻量级的数据库 Java 测试库

- 客户端
- zio-nebula：Scala 客户端
- nebula-node：Node.js 客户端
- nebula-php：PHP 客户端
- nebula-net：.NET 客户端
- nebula-rust：Rust 客户端

- 终端工具
- nebula-console-intellij-plugin：JetBrains IDE 系列 nebula-console 插件，支持语法高亮、函数字段自动补全、数据表格分页显示、关系图展示等等功能

最后更新: 2025年8月4日

21.3 产品端口全集

以下是 NebulaGraph 内核及周边工具使用的默认端口信息：

序号	所属产品/服务	类型	默认端口	说明
1	NebulaGraph	TCP	9669	Graph 服务的 RPC 守护进程监听端口。通常用于客户端连接 Graph服务。
2	NebulaGraph	TCP	19669	Graph 服务的 HTTP 端口。
3	NebulaGraph	TCP	19670	Graph 服务的 HTTP/2 端口。（3.x 后已弃用该端口）
4	NebulaGraph	TCP	9559、9560	`9559` 是 Meta 服务的 RPC 守护进程监听端口。通常由 Graph 服务和 Storage 服务发起请求，用于获取和更新图数据库的元数据信息。 同时还会使用相邻的 `+1` （`9560`）端口用于 Meta 服务之间的 Raft 通信。
5	NebulaGraph	TCP	19559	Meta 服务的 HTTP 端口。
6	NebulaGraph	TCP	19560	Meta 服务的 HTTP/2 端口。（3.x 后已弃用该端口）
7	NebulaGraph	TCP	9779、9778、9780	`9779` 是 Storage 服务的 RPC 守护进程监听端口。通常由 Graph 服务发起请求，用于执行数据存储相关的操作，例如读取、写入或删除数据。 同时还会使用相邻的 `-1` （`9778`）和 `+1` （`9780`）端口。 `9778`：Admin 服务（Storage 接收 Meta 命令的服务）占用的端口。 `9780`：Storage 服务之间的 Raft 通信端口。
8	NebulaGraph	TCP	19779	Storage 服务的 HTTP 端口。
9	NebulaGraph	TCP	19780	Storage 服务的 HTTP/2 端口。（3.x 后已弃用该端口）
10	NebulaGraph	TCP	8888	备份和恢复功能的 Agent 服务端口。Agent 是集群中每台机器的一个守护进程，用于启停 NebulaGraph 服务和上传、下载备份文件。
11	NebulaGraph	TCP	9789、9788、9790	`9789` 是全文索引中 Raft Listener 的端口，从 Storage 服务读取数据，然后将它们写入 Elasticsearch 集群。 也是集群间数据同步中 Storage Listener 的端口。用于同步主集群的 Storage 数据。 同时还会使用相邻的 `-1` （`9788`）和 `+1` （`9790`）端口。 `9788`：内部端口。 `9790`：Raft 通信端口。
12	NebulaGraph	TCP	9200	NebulaGraph 使用该端口与 Elasticsearch 进行 HTTP 通信，以执行全文搜索查询和管理全文索引。
13	NebulaGraph	TCP	9569、9568、9570	`9569` 是集群间数据同步功能中 Meta Listener 的端口，用于同步主集群的 Meta 数据。 同时还会使用相邻的 `-1` （`9568`）和 `+1` （`9570`）端口。 `9568`：内部端口。 `9570`：Raft 通信端口。
14	NebulaGraph	TCP	9889、9888、9890	`9889` 是集群间数据同步功能中 Drainer 服务端口。用于同步 Storage、Meta 数据给从集群。 同时还会使用相邻的 `-1` （`9888`）和 `+1` （`9890`）端口。 `9888`：内部端口。 `9890`：Raft 通信端口。
15	NebulaGraph Studio	TCP	7001	Studio 提供 Web 服务占用端口。
16	NebulaGraph Dashboard	TCP	8090	Nebula HTTP Gateway 依赖服务端口。为集群服务提供 HTTP 接口，执行 nGQL 语句与 NebulaGraph 数据库进行交互。
17		TCP	9200	

序号	所属产品/服务	类型	默认端口	说明
	NebulaGraph Dashboard			Nebula Stats Exporter 依赖服务端口。收集集群的性能指标，包括服务 IP 地址、版本和监控指标（例如查询数量、查询延迟、心跳延迟 等）。
18	NebulaGraph Dashboard	TCP	9100	Node Exporter 依赖服务端口。收集集群中机器的资源信息，包括 CPU、内存、负载、磁盘和流量。
19	NebulaGraph Dashboard	TCP	9090	Prometheus 服务的端口。存储监控数据的时间序列数据库。
20	NebulaGraph Dashboard	TCP	7003	Dashboard 社区版 提供 Web 服务占用端口。

最后更新: 2025年8月4日

21.4 如何贡献代码和文档

21.4.1 开始之前

GitHub 或社区提交问题

欢迎为项目贡献任何代码或文档，但是建议先在 GitHub 或社区上提交一个问题，和大家共同讨论。

签署贡献者许可协议 CLA

1. 打开 CLA 服务登录页面。
2. 单击 Sign in with GitHub。
3. 阅读并同意协议 vesoft inc. Contributor License Agreement。

如果有任何问题，请提交 issue。

21.4.2 修改单篇文档

NebulaGraph 文档以 Markdown 语言编写。单击文档标题右侧的铅笔图标即可提交修改建议。

该方法仅适用于修改单篇文档。

21.4.3 批量修改或新增文件

该方法适用于贡献代码、批量修改多篇文档或者新增文档。

Step 1：通过 GitHub fork 仓库

NebulaGraph 项目有很多仓库，以 NebulaGraph 仓库为例：

1. 访问 github.com/vesoft-inc/nebula。
2. 在右上角单击按钮 `Fork`，然后单击用户名，即可 fork 出 NebulaGraph 仓库。

Step 2：将分支克隆到本地

1. 定义本地工作目录。

```
# 定义工作目录。
working_dir=$HOME/Workspace
```

2. 将 `user` 设置为 GitHub 的用户名。

```
user={GitHub 用户名}
```

3. 克隆代码。

```
mkdir -p $working_dir
cd $working_dir
git clone https://github.com/$user/nebula.git
# 或：git clone git@github.com:$user/nebula.git

cd $working_dir/nebula
git remote add upstream https://github.com/vesoft-inc/nebula.git
# 或：git remote add upstream git@github.com:vesoft-inc/nebula.git

# 由于没有写访问权限，请勿推送至上游主分支。
git remote set-url --push upstream no_push

# 确认远程分支有效。
# 正确的格式为：
# origin    git@github.com:$(user)/nebula.git (fetch)
# origin    git@github.com:$(user)/nebula.git (push)
# upstream  https://github.com/vesoft-inc/nebula (fetch)
```

```
# upstream  no_push (push)
git remote -v
```

4. （可选）定义 pre-commit hook。

   请将 NebulaGraph 的 pre-commit hook 连接到 `.git` 目录。

   hook 将检查 commit，包括格式、构建、文档生成等。

```
cd $working_dir/nebula/.git/hooks
ln -s $working_dir/nebula/.linters/cpp/hooks/pre-commit.sh .
```

   pre-commit hook 有时候可能无法正常执行，用户必须手动执行。

```
cd $working_dir/nebula/.git/hooks
chmod +x pre-commit
```

Step 3：分支

1. 更新本地主分支。

```
cd $working_dir/nebula
git fetch upstream
git checkout master
git rebase upstream/master
```

2. 从主分支创建并切换分支：

```
git checkout -b myfeature
```

> Note
>
> 由于一个 PR 通常包含多个 commits，最终合入 upstream/master 分支时，我们会将这些 commits 挤压（squash）成一个 commit 进行合并。因此强烈建议创建一个独立的分支进行更改，这样在合入时才容易被挤压。合并后，这个分支可以被丢弃。如果未创建单独的分支，而是直接将 commits 提交至 origin/master，在合入时，可能会出现问题。 若未创建单独的分支（或是 origin/master 合并了其他的分支等），导致 origin/master 和 upstream/master 不一致时，用户可以使用 hard reset 强制两者进行一致。例如：
>
> ```
> git fetch upstream
> git checkout master
> git reset --hard upstream/master
> git push --force origin master
> ```

Step 4：开发

- 代码风格

NebulaGraph 采用 `cpplint` 来确保代码符合 Google 的代码风格指南。检查器将在提交代码之前执行。

- 单元测试要求

  请为新功能或 Bug 修复添加单元测试。

- 构建代码时开启单元测试

  详情请参见使用源码安装 NebulaGraph 。

  > Note
  >
  > 请确保已设置 `-DENABLE_TESTING = ON` 启用构建单元测试。

- 运行所有单元测试

  在 `nebula` 根目录执行如下命令：

```
cd nebula/build
ctest -j$(nproc)
```

Step 5：保持分支同步

```
# 当处于 myfeature 分支时。
git fetch upstream
git rebase upstream/master
```

在其他贡献者将 PR 合并到基础分支之后，用户需要更新 head 分支。

Step 6：**Commit**

提交代码更改：

```
git commit -a
```

用户可以使用命令 `--amend` 重新编辑之前的代码。

Step 7：**Push**

需要审核或离线备份代码时，可以将本地仓库创建的分支 push 到 GitHub 的远程仓库。

```
git push origin myfeature
```

Step 8：创建 **pull request**

1. 访问 fork 出的仓库 `https://github.com/$user/nebula` （替换此处的用户名 `$user` ）。
2. 单击 `myfeature` 分支旁的按钮 `Compare & pull request` 。

Step 9：代码审查

pull request 创建后，至少需要两人审查。审查人员将进行彻底的代码审查，以确保变更满足存储库的贡献准则和其他质量标准。

21.4.4 添加测试用例

添加测试用例的方法参见 How to add test cases。

21.4.5 捐赠项目

Step 1：确认项目捐赠

通过邮件、微信、Slack 等方式联络 NebulaGraph 官方人员，确认捐赠项目一事。项目将被捐赠至 NebulaGraph Contrib 组织下。

- 邮件地址：info@vesoft.com
- 微信：NebulaGraphbot
- Slack：Join Slack

Step 2：获取项目接收人信息

由 NebulaGraph 官方人员给出 NebulaGraph Contrib 的项目接收者 ID。

Step 3：捐赠项目

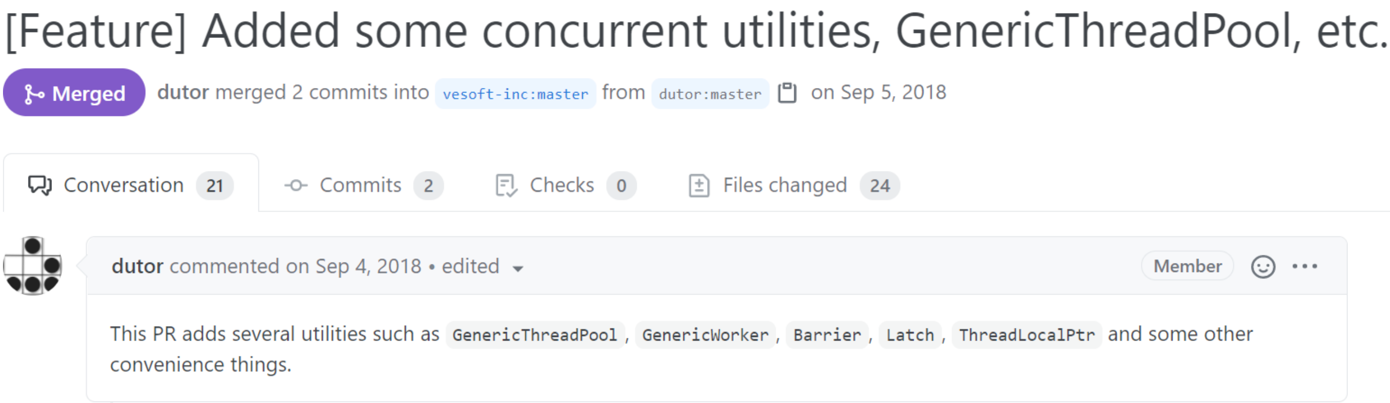
由您将项目转移至本次捐赠的项目接受人，并由项目接收者将该项目转移至 NebulaGraph Contrib 组织下。捐赠后，您将以 Maintain 角色继续主导社区项目的发展。

GitHub 上转移仓库的操作，请参见 Transferring a repository owned by your user account。

最后更新: 2025年8月4日

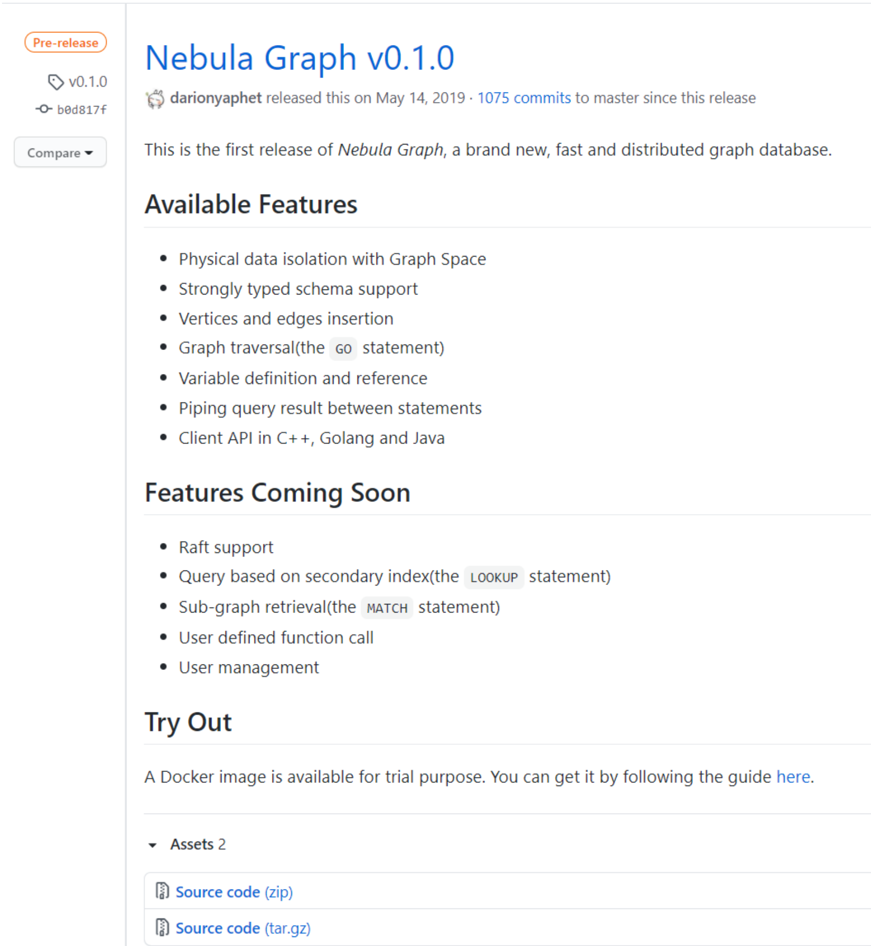
21.5 NebulaGraph 年表

1. 2018.9.5 由 @dutor 提交了第一行代码。



2. 2019.5 发布了 v0.1.0 alpha 版本, 并开源。

此后一年内陆续发布 v1.0.0-beta, v1.0.0-rc1, v1.0.0-rc2, v1.0.0-rc3, v1.0.0-rc4



Pre-release

v0.1.0

b0d817f

Compare

Nebula Graph v0.1.0

darionyaphet released this on May 14, 2019 · 1075 commits to master since this release

This is the first release of *Nebula Graph*, a brand new, fast and distributed graph database.

Available Features

- Physical data isolation with Graph Space
- Strongly typed schema support
- Vertices and edges insertion
- Graph traversal(the `GO` statement)
- Variable definition and reference
- Piping query result between statements
- Client API in C++, Golang and Java

Features Coming Soon

- Raft support
- Query based on secondary index(the `LOOKUP` statement)
- Sub-graph retrieval(the `MATCH` statement)
- User defined function call
- User management

Try Out

A Docker image is available for trial purpose. You can get it by following the guide here.

Assets 2

Source code (zip)

Source code (tar.gz)

3. 2019.7 在 HBaseCon 第一次公开亮相[1] @dangleptr

4. 2020.3 在 v1.0 开发的收尾阶段，启动了 v2.0 项目的研发
5. 2020.6 发布了第一个正式大版本 v1.0.0 GA

v1.0.0
06a5db4
Verified
Compare

V1.0.0 GA

jude-zhu released this on Jun 10, 2020 · 146 commits to master since this release

Basic Features

- Online DDL & DML. Support updating schemas and data without stopping or affecting your ongoing operations.
- Graph traversal. `GO` statement supports forward/reverse and bidirectional graph traversal. `GO minHops TO maxHops` is supported to get variable hops relationships.
- Aggregate. Support aggregation functions such as `GROUP BY`, `ORDER BY`, and `LIMIT`.
- Composite query. Support composite clauses: `UNION`, `UNION DISTINCT`, `INTERSECT`, and `MINUS`.
- PIPE statements. The result yielded from the previous statement could be piped to the next statement as input.
- Use defined variables. Support user-defined variables to pass the result of a query to another.
- Index. Both the single-property index and composite index are supported to make searches of related data more efficient. `LOOKUP ON` statement is to query on the index.

Advanced Features

- Privilege Management. Support user authentication and role-based access control. Nebula Graph can easily integrate with third-party authentication systems. There are five built-in roles in Nebula Graph: `GOD`, `ADMIN`, `DBA`, `USER`, and `GUEST`. Each role has its corresponding privileges.
- Support Reservoir Sampling, which will retrieve k elements randomly for the sampling of the supernode at the complexity of `O(n)`.
- Cluster snapshot. Support creating snapshots for the cluster as an online backup strategy.
- TTL. Support TTL to expire items after a certain amount of time automatically.
- Operation & Maintenance
  - Scale in/out. Support online scale in/out and load balance for storage
  - `HOSTS` clause to manage storage hosts
  - `CONFIGS` clause to manage configuration options
- Job Manager & Scheduler. A tool for job managing and scheduling. Currently, `COMPACT` and `FLUSH` jobs are supported.
- Graph Algorithms. Support finding the full path and the shortest path between vertices.
- Provide OLAP interfaces to integrate with third-party graph analytics platforms.
- Support multiple character sets and collations. The default `CHARSET` and `COLLATE` are `utf8` and `utf8_bin`.

Clients

- Java Client. Support source code building and downloading from the MVN repository, see Java Client for more details.
- Python Client. Support source code building and installation with pip, see Python Client for more details.
- Golang Client. Install the client with the command `go get -u -v github.com/vesoft-inc/nebula-go`, see Go Client for more details.

Nebula Graph Studio

A graphical user interface for working with Nebula Graph. Support querying, designing schema, data loading, and graph exploring. See Nebula Graph Studio for more details.

6. 2021.3 发布了第二个大版本 v2.0 GA

v2.0.0
91639db
Verified
Compare

Nebula Graph v2.0 GA

jude-zhu released this on Mar 23

New Features

- `vertexID` supports both `Integer` and `String`.
- New data types:
  - NULL: the property can be set to `NULL`. `NOT NULL` constraint is also supported
  - Composite types: LIST, SET, and MAP(Cannot be set as property types)
  - Temporal types: DATE and DATETIME.
  - FIXED_STRING: a fixed size `String`
- Full-text indexes are supported to do prefix, wildcard, regex, and fuzzy search on a string property.
- Explain & Profile outputs the execution plan of an nGQL statement and execution profile.
- Subgraph to retrieve vertices and edges reachable from the start vertices.
- Support to collect statistics of the graph space.
- OpenCypher compatibility
  - Partially support the `MATCH` clause
  - Support `RETURN`, `WITH`, `UNWIND`, `LIMIT` & `SKIP` clauses
- More built-in functions
  - Predicate functions
  - Scalar functions
  - List functions
  - Aggregating functions
  - Mathematical functions
  - String functions
  - Temporal functions

Improvements

- Optimize the performance of inserting, updating, and deleting data with indexes.
- `LOOKUP ON` filtering data supports `OR` and `AND` operators.
- `FIND PATH` supports finding paths with or without regard to direction, and also supports excluding cycles in paths.
- `SHOW HOSTS graph/meta/storage` supports to retrieve the basic information of graphd/metad/storaged hosts.

Changelog

- The data type of `vertexID` must be specified when creating a graph space.
- `FETCH PROP ON` returns a composite object if not specify the result set.
- Changed the default port numbers of `metad`, `graphd`, and `storaged`.
- Refactor metrics counters.

Nebula-graph Console

Supports local commands mode. `:set csv` outputs the query results to the console and the specified CSV file. For more information, please refer to https://github.com/vesoft-inc/nebula-console.

Clients

Support connection pool and load balance.

- cpp client https://github.com/vesoft-inc/nebula-cpp
- java client https://github.com/vesoft-inc/nebula-java
- python client https://github.com/vesoft-inc/nebula-python
- go client https://github.com/vesoft-inc/nebula-go

Nebula Graph Studio

With Studio, you can create a graph schema, load data, execute nGQL statements, and explore graphs in one stop. For more information, please refer to https://github.com/vesoft-inc/nebula-web-docker.

Known Issues

- #860

7. 2021.8 发布 v2.5.0
8. 2021.10 发布 v2.6.0
9. 2022.2 发布 v3.0.0
10. 2022.4 发布 v3.1.0
11. 2022.7 发布 v3.2.0
12. 2022.10 发布 v3.3.0
13. 2023.2 发布 v3.4.0
14. 2023.5 发布 v3.5.0
15. 2023.8 发布 v3.6.0

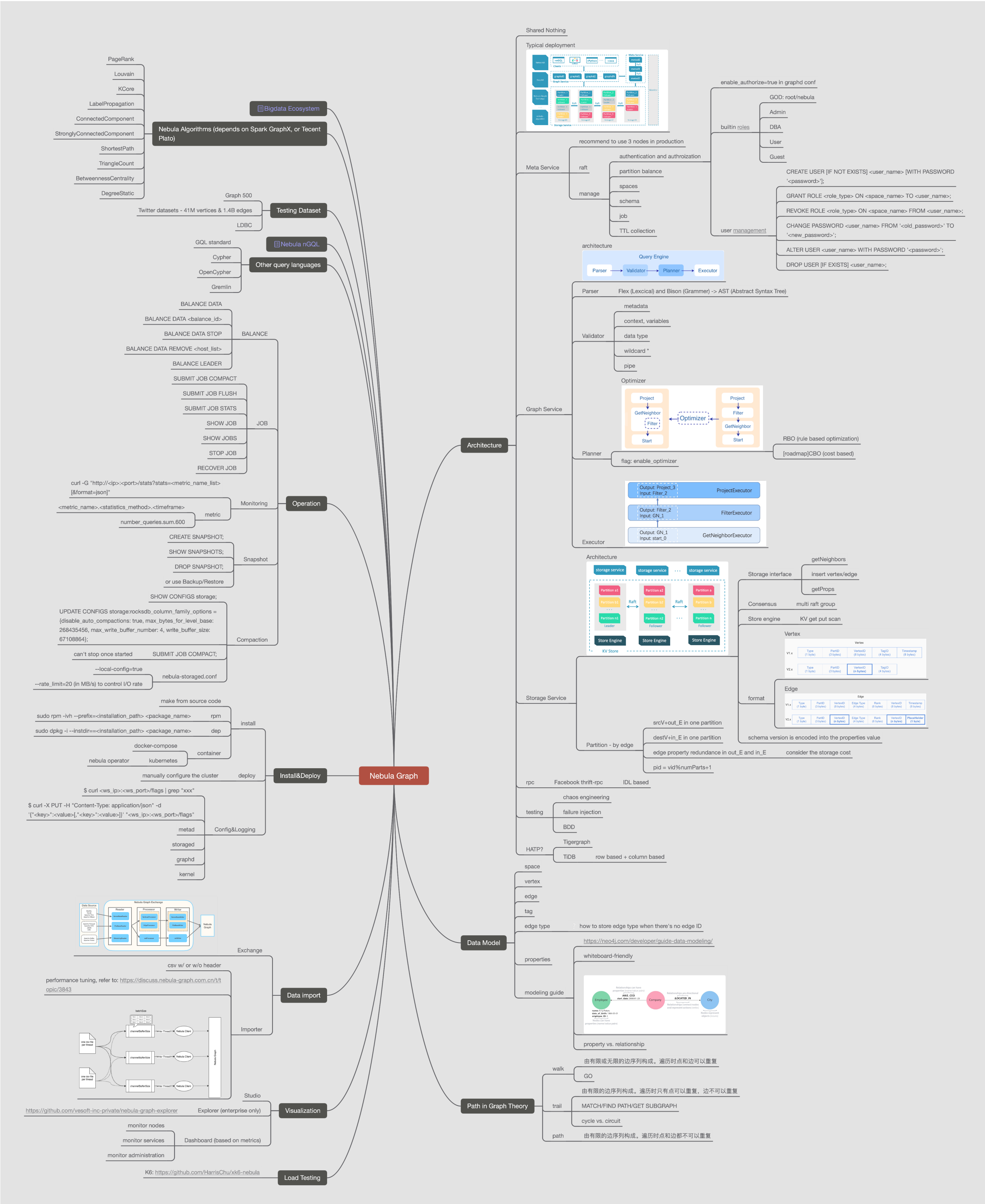
---

1. NebulaGraph 1.x 版本支持 RocksDB 和 HBase 两种主要的后端，但在 NebulaGraph 2.x 版本取消了默认对 HBase 的支持。↩

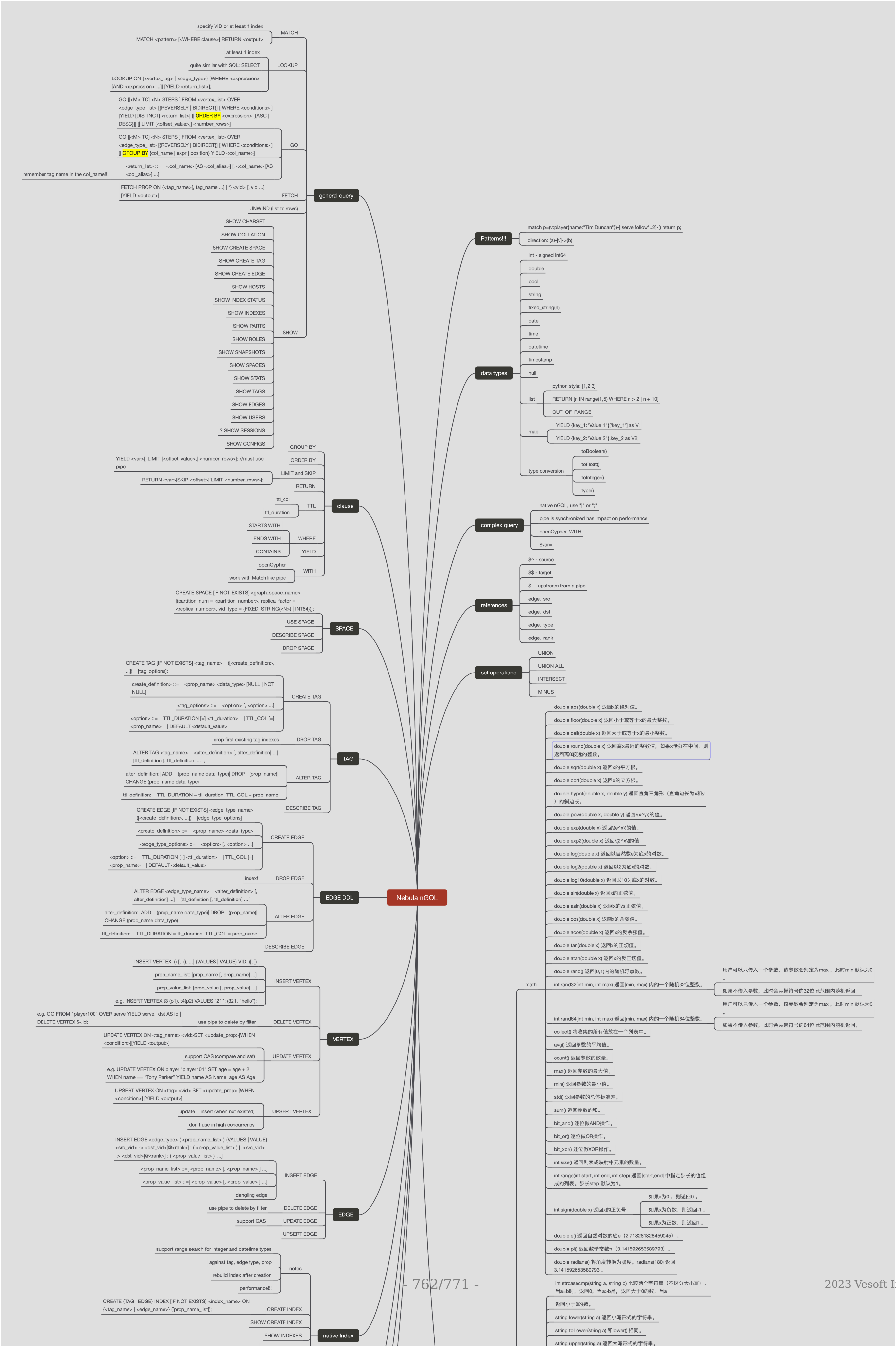
最后更新: 2025年8月4日

21.6 思维导图

以下给出 NebulaGraph 结构框架的思维导图，用户可以点击 并查看大图。



以下给出 nGQL 的思维导图，用户可以点击 并查看大图。


Nebula nGQL
general query
MATCH
specify VID or at least 1 index
MATCH <pattern> [<WHERE clause>] RETURN <output>
LOOKUP
at least 1 index
quite similar with SQL: SELECT
LOOKUP ON {<vertex_tag> | <edge_type>} [WHERE <expression> [AND <expression> ...]] [YIELD <return_list>];
GO
GO [[<M> TO] <N> STEPS] FROM <vertex_list> OVER <edge_type_list> [{REVERSELY | BIDIRECT}] [WHERE <conditions>] [YIELD [DISTINCT] <return_list>] [| ORDER BY <expression> [{ASC | DESC}]] [| LIMIT [<offset_value>,] <number_rows>]
GO [[<M> TO] <N> STEPS] FROM <vertex_list> OVER <edge_type_list> [{REVERSELY | BIDIRECT}] [WHERE <conditions>] [| GROUP BY {col_name | expr | position} YIELD <col_name>]
<return_list> ::= <col_name> [AS <col_alias>] [, <col_name> [AS <col_alias>] ...]
remember tag name in the col_name!!!
FETCH
FETCH PROP ON {<tag_name>[, tag_name ...] | *} <vid> [, vid ...] [YIELD <output>]
UNWIND (list to rows)
SHOW
SHOW CHARSET
SHOW COLLATION
SHOW CREATE SPACE
SHOW CREATE TAG
SHOW CREATE EDGE
SHOW HOSTS
SHOW INDEX STATUS
SHOW INDEXES
SHOW PARTS
SHOW ROLES
SHOW SNAPSHOTS
SHOW SPACES
SHOW STATS
SHOW TAGS
SHOW EDGES
SHOW USERS
? SHOW SESSIONS
SHOW CONFIGS
clause
GROUP BY
ORDER BY
LIMIT and SKIP
YIELD <var>[| LIMIT [<offset_value>,] <number_rows>]; //must use pipe
RETURN <var>[SKIP <offset>][LIMIT <number_rows>];
RETURN
TTL
ttl_col
ttl_duration
WHERE
STARTS WITH
ENDS WITH
CONTAINS
YIELD
WITH
openCypher
work with Match like pipe
SPACE
CREATE SPACE [IF NOT EXISTS] <graph_space_name> [(partition_num = <partition_number>, replica_factor = <replica_number>, vid_type = {FIXED_STRING(<N>) | INT64})];
USE SPACE
DESCRIBE SPACE
DROP SPACE
TAG
CREATE TAG
CREATE TAG [IF NOT EXISTS] <tag_name> ([<create_definition>, ...]) [tag_options];
create_definition> ::= <prop_name> <data_type> [NULL | NOT NULL]
<tag_options> ::= <option> [, <option> ...]
<option> ::= TTL_DURATION [=] <ttl_duration> | TTL_COL [=] <prop_name> | DEFAULT <default_value>
DROP TAG
drop first existing tag indexes
ALTER TAG
ALTER TAG <tag_name> <alter_definition> [, alter_definition] ...] [ttl_definition [, ttl_definition] ...];
alter_definition:| ADD (prop_name data_type)| DROP (prop_name)| CHANGE (prop_name data_type)
ttl_definition: TTL_DURATION = ttl_duration, TTL_COL = prop_name
DESCRIBE TAG
EDGE DDL
CREATE EDGE
CREATE EDGE [IF NOT EXISTS] <edge_type_name> ([<create_definition>, ...]) [edge_type_options]
<create_definition> ::= <prop_name> <data_type>
<edge_type_options> ::= <option> [, <option> ...]
<option> ::= TTL_DURATION [=] <ttl_duration> | TTL_COL [=] <prop_name> | DEFAULT <default_value>
DROP EDGE
index!
ALTER EDGE
ALTER EDGE <edge_type_name> <alter_definition> [, alter_definition] ...] [ttl_definition [, ttl_definition] ...]
alter_definition:| ADD (prop_name data_type)| DROP (prop_name)| CHANGE (prop_name data_type)
ttl_definition: TTL_DURATION = ttl_duration, TTL_COL = prop_name
DESCRIBE EDGE
VERTEX
INSERT VERTEX
INSERT VERTEX () [{, () ...}] {VALUES | VALUE} VID: ({, })
prop_name_list: [prop_name [, prop_name] ...]
prop_value_list: [prop_value [, prop_value] ...]
e.g. INSERT VERTEX t3 (p1), t4(p2) VALUES "21": (321, "hello");
DELETE VERTEX
e.g. GO FROM "player100" OVER serve YIELD serve._dst AS id | DELETE VERTEX $-.id;
use pipe to delete by filter
UPDATE VERTEX
UPDATE VERTEX ON <tag_name> <vid>SET <update_prop>[WHEN <condition>][YIELD <output>]
support CAS (compare and set)
e.g. UPDATE VERTEX ON player "player101" SET age = age + 2 WHEN name == "Tony Parker" YIELD name AS Name, age AS Age
UPSERT VERTEX
UPSERT VERTEX ON <tag> <vid> SET <update_prop> [WHEN <condition>] [YIELD <output>]
update + insert (when not existed)
don't use in high concurrency
EDGE
INSERT EDGE
INSERT EDGE <edge_type> (<prop_name_list>) {VALUES | VALUE} <src_vid> -> <dst_vid>[@<rank>] : (<prop_value_list>) [, <src_vid> -> <dst_vid>[@<rank>] : (<prop_value_list>), ...]
<prop_name_list> ::=[<prop_name> [, <prop_name>] ...]
<prop_value_list> ::=[<prop_value> [, <prop_value>] ...]
dangling edge
DELETE EDGE
use pipe to delete by filter
UPDATE EDGE
support CAS
UPSERT EDGE
native Index
notes
support range search for integer and datetime types
against tag, edge type, prop
rebuild index after creation
performance!!!
CREATE INDEX
CREATE {TAG | EDGE} INDEX [IF NOT EXISTS] <index_name> ON {<tag_name> | <edge_name>} ([prop_name_list]);
SHOW CREATE INDEX
SHOW INDEXES
Patterns!!!
match p=(v:player{name:"Tim Duncan"})-[:serve|follow*..2]-() return p;
direction: (a)-[v]->(b)
data types
int - signed int64
double
bool
string
fixed_string(n)
date
time
datetime
timestamp
null
list
python style: [1,2,3]
RETURN [n IN range(1,5) WHERE n > 2 | n + 10]
OUT_OF_RANGE
map
YIELD {key_1:"Value 1"}['key_1'] as V;
YIELD {key_2:"Value 2"}.key_2 as V2;
type conversion
toBoolean()
toFloat()
toInteger()
type()
complex query
native nGQL, use "|" or ";"
pipe is synchronized has impact on performance
openCypher, WITH
$var=
references
$^ - source
$$ - target
$- - upstream from a pipe
edge._src
edge._dst
edge._type
edge._rank
set operations
UNION
UNION ALL
INTERSECT
MINUS
math
double abs(double x) 返回x的绝对值。
double floor(double x) 返回小于或等于x的最大整数。
double ceil(double x) 返回大于或等于x的最小整数。
double round(double x) 返回离x最近的整数值，如果x恰好在中间，则返回离0较远的整数。
double sqrt(double x) 返回x的平方根。
double cbrt(double x) 返回x的立方根。
double hypot(double x, double y) 返回直角三角形（直角边长为x和y）的斜边长。
double pow(double x, double y) 返回\(x^y\)的值。
double exp(double x) 返回\(e^x\)的值。
double exp2(double x) 返回\(2^x\)的值。
double log(double x) 返回以自然数e为底x的对数。
double log2(double x) 返回以2为底x的对数。
double log10(double x) 返回以10为底x的对数。
double sin(double x) 返回x的正弦值。
double asin(double x) 返回x的反正弦值。
double cos(double x) 返回x的余弦值。
double acos(double x) 返回x的反余弦值。
double tan(double x) 返回x的正切值。
double atan(double x) 返回x的反正切值。
double rand() 返回[0,1)内的随机浮点数。
int rand32(int min, int max) 返回[min, max) 内的一个随机32位整数。
用户可以只传入一个参数，该参数会判定为max，此时min 默认为0。
如果不传入参数，此时会从带符号的32位int范围内随机返回。
int rand64(int min, int max) 返回[min, max) 内的一个随机64位整数。
用户可以只传入一个参数，该参数会判定为max，此时min 默认为0。
如果不传入参数，此时会从带符号的64位int范围内随机返回。
collect() 将收集的所有值放在一个列表中。
avg() 返回参数的平均值。
count() 返回参数的数量。
max() 返回参数的最大值。
min() 返回参数的最小值。
std() 返回参数的总体标准差。
sum() 返回参数的和。
bit_and() 逐位做AND操作。
bit_or() 逐位做OR操作。
bit_xor() 逐位做XOR操作。
int size() 返回列表或映射中元素的数量。
int range(int start, int end, int step) 返回[start,end] 中指定步长的值组成的列表。步长step 默认为1。
int sign(double x) 返回x的正负号。
如果x为0，则返回0。
如果x为负数，则返回-1。
如果x为正数，则返回1。
double e() 返回自然对数的底e（2.718281828459045）。
double pi() 返回数学常数π（3.141592653589793）。
double radians() 将角度转换为弧度。radians(180) 返回 3.141592653589793。
int strcasecmp(string a, string b) 比较两个字符串（不区分大小写）。当a=b时，返回0，当a>b是，返回大于0的数，当a
返回小于0的数。
string lower(string a) 返回小写形式的字符串。
string toLower(string a) 和lower() 相同。
string upper(string a) 返回大写形式的字符串。

最后更新: 2025年8月4日

21.7 错误码

NebulaGraph 运行出现问题时，会返回错误码。本文介绍错误码的详细信息。

Note

- 如果出现错误但没有返回错误码，或错误码描述不清，请在论坛或 GitHub 反馈。
- 返回 `0` 表示执行成功。

错误名称	错误码	说明
`E_DISCONNECTED`	`-1`	连接断开
`E_FAIL_TO_CONNECT`	`-2`	无法建立连接
`E_RPC_FAILURE`	`-3`	RPC 失败
`E_LEADER_CHANGED`	`-4`	Raft leader 变更
`E_SPACE_NOT_FOUND`	`-5`	图空间不存在
`E_TAG_NOT_FOUND`	`-6`	Tag 不存在
`E_EDGE_NOT_FOUND`	`-7`	Edge type不存在
`E_INDEX_NOT_FOUND`	`-8`	索引不存在
`E_EDGE_PROP_NOT_FOUND`	`-9`	边属性不存在
`E_TAG_PROP_NOT_FOUND`	`-10`	Tag 属性不存在
`E_ROLE_NOT_FOUND`	`-11`	当前角色不存在
`E_CONFIG_NOT_FOUND`	`-12`	当前配置不存在
`E_MACHINE_NOT_FOUND`	`-13`	当前主机不存在
`E_LISTENER_NOT_FOUND`	`-15`	listener 不存在
`E_PART_NOT_FOUND`	`-16`	当前分区不存在
`E_KEY_NOT_FOUND`	`-17`	key 不存在
`E_USER_NOT_FOUND`	`-18`	用户不存在
`E_STATS_NOT_FOUND`	`-19`	统计信息不存在
`E_SERVICE_NOT_FOUND`	`-20`	没有找到当前服务
`E_DRAINER_NOT_FOUND`	`-21`	drainer 不存在
`E_DRAINER_CLIENT_NOT_FOUND`	`-22`	drainer 客户端不存在
`E_PART_STOPPED`	`-23`	当前 partition 已经被停止
`E_BACKUP_FAILED`	`-24`	备份失败
`E_BACKUP_EMPTY_TABLE`	`-25`	备份的表为空
`E_BACKUP_TABLE_FAILED`	`-26`	备份表失败
`E_PARTIAL_RESULT`	`-27`	multiget 无法获得所有数据
`E_REBUILD_INDEX_FAILED`	`-28`	重建索引失败
`E_INVALID_PASSWORD`	`-29`	密码无效
`E_FAILED_GET_ABS_PATH`	`-30`	无法获得绝对路径
`E_BAD_USERNAME_PASSWORD`	`-1001`	身份验证失败
`E_SESSION_INVALID`	`-1002`	无效会话
`E_SESSION_TIMEOUT`	`-1003`	会话超时
`E_SYNTAX_ERROR`	`-1004`	语法错误
`E_EXECUTION_ERROR`	`-1005`	执行错误
`E_STATEMENT_EMPTY`	`-1006`	语句为空

错误名称	错误码	说明
`E_BAD_PERMISSION`	-1008	权限不足
`E_SEMANTIC_ERROR`	-1009	语义错误
`E_TOO_MANY_CONNECTIONS`	-1010	超出最大连接数
`E_PARTIAL_SUCCEEDED`	-1011	访问存储失败（仅有部分请求成功）
`E_NO_HOSTS`	-2001	主机不存在
`E_EXISTED`	-2002	主机已经存在
`E_INVALID_HOST`	-2003	无效主机
`E_UNSUPPORTED`	-2004	当前命令、语句、功能不支持
`E_NOT_DROP`	-2005	不允许删除
`E_CONFIG_IMMUTABLE`	-2007	配置项不能改变
`E_CONFLICT`	-2008	参数与 meta 数据冲突
`E_INVALID_PARM`	-2009	无效的参数
`E_WRONGCLUSTER`	-2010	错误的集群
`E_ZONE_NOT_ENOUGH`	-2011	listener 冲突
`E_ZONE_IS_EMPTY`	-2012	主机不存在
`E_SCHEMA_NAME_EXISTS`	-2013	Schema 名字已存在
`E_RELATED_INDEX_EXISTS`	-2014	与 Tag 或 Edge Type 相关的索引存在，不能被删除
`E_RELATED_SPACE_EXISTS`	-2015	仍有图空间在主机上，不能被删除
`E_STORE_FAILURE`	-2021	存储数据失败
`E_STORE_SEGMENT_ILLEGAL`	-2022	存储段非法
`E_BAD_BALANCE_PLAN`	-2023	无效的数据均衡计划
`E_BALANCED`	-2024	集群已经处于数据均衡状态
`E_NO_RUNNING_BALANCE_PLAN`	-2025	没有正在运行的数据均衡计划
`E_NO_VALID_HOST`	-2026	缺少有效的主机
`E_CORRUPTED_BALANCE_PLAN`	-2027	已经损坏的数据均衡计划
`E_IMPROPER_ROLE`	-2030	回收用户角色失败
`E_INVALID_PARTITION_NUM`	-2031	无效的分区数量
`E_INVALID_REPLICA_FACTOR`	-2032	无效的副本因子
`E_INVALID_CHARSET`	-2033	无效的字符集
`E_INVALID_COLLATE`	-2034	无效的字符排序规则
`E_CHARSET_COLLATE_NOT_MATCH`	-2035	字符集和字符排序规则不匹配
`E_SNAPSHOT_FAILURE`	-2040	生成快照失败
`E_BLOCK_WRITE_FAILURE`	-2041	写入块数据失败
`E_ADD_JOB_FAILURE`	-2044	增加新的任务失败
`E_STOP_JOB_FAILURE`	-2045	停止任务失败

错误名称	错误码	说明
`E_SAVE_JOB_FAILURE`	`-2046`	保存任务信息失败
`E_BALANCER_FAILURE`	`-2047`	数据均衡失败
`E_JOB_NOT_FINISHED`	`-2048`	当前任务还没有完成
`E_TASK_REPORT_OUT_DATE`	`-2049`	任务报表失效
`E_JOB_NOT_IN_SPACE`	`-2050`	当前任务不在图空间内
`E_JOB_NEED_RECOVER`	`-2051`	当前任务需要恢复
`E_JOB_ALREADY_FINISH`	`-2052`	任务已经失败或完成
`E_JOB_SUBMITTED`	`-2053`	任务默认状态
`E_JOB_NOT_STOPPABLE`	`-2054`	给定任务不支持停止
`E_JOB_HAS_NO_TARGET_STORAGE`	`-2055`	leader 分布未上报，因此无法将任务发送到存储
`E_INVALID_JOB`	`-2065`	无效的任务
`E_BACKUP_BUILDING_INDEX`	`-2066`	备份终止（正在创建索引）
`E_BACKUP_SPACE_NOT_FOUND`	`-2067`	备份时图空间不存在
`E_RESTORE_FAILURE`	`-2068`	备份恢复失败
`E_SESSION_NOT_FOUND`	`-2069`	会话不存在
`E_LIST_CLUSTER_FAILURE`	`-2070`	获取集群信息失败
`E_LIST_CLUSTER_GET_ABS_PATH_FAILURE`	`-2071`	获取集群信息时无法获取绝对路径
`E_LIST_CLUSTER_NO_AGENT_FAILURE`	`-2072`	获取集群信息时无法获得 agent
`E_QUERY_NOT_FOUND`	`-2073`	query 未找到
`E_AGENT_HB_FAILURE`	`-2074`	agent 没有汇报心跳
`E_HOST_CAN_NOT_BE_ADDED`	`-2082`	该主机不能被添加，因为它不是一个 Storage 主机
`E_ACCESS_ES_FAILURE`	`-2090`	访问 Elasticsearch 失败
`E_GRAPH_MEMORY_EXCEEDED`	`-2600`	Graph 内存超出
`E_CONSENSUS_ERROR`	`-3001`	选举时无法达成共识
`E_KEY_HAS_EXISTS`	`-3002`	key 已经存在
`E_DATA_TYPE_MISMATCH`	`-3003`	数据类型不匹配
`E_INVALID_FIELD_VALUE`	`-3004`	无效的字段值
`E_INVALID_OPERATION`	`-3005`	无效的操作
`E_NOT_NULLABLE`	`-3006`	当前值不允许为空
`E_FIELD_UNSET`	`-3007`	字段非空或者没有默认值时，字段值必须设置
`E_OUT_OF_RANGE`	`-3008`	取值超出了当前类型的范围
`E_DATA_CONFLICT_ERROR`	`-3010`	数据冲突
`E_WRITE_STALLED`	`-3011`	写入被延迟
`E_IMPROPER_DATA_TYPE`	`-3021`	不正确的数据类型
`E_INVALID_SPACEVIDLEN`	`-3022`	VID 长度无效

错误名称	错误码	说明
`E_INVALID_FILTER`	`-3031`	无效的过滤器
`E_INVALID_UPDATER`	`-3032`	无效的字段更新
`E_INVALID_STORE`	`-3033`	无效的 KV 存储
`E_INVALID_PEER`	`-3034`	peer 无效
`E_RETRY_EXHAUSTED`	`-3035`	重试次数耗光
`E_TRANSFER_LEADER_FAILED`	`-3036`	leader 转换失败
`E_INVALID_STAT_TYPE`	`-3037`	无效的统计类型
`E_INVALID_VID`	`-3038`	VID 无效
`E_LOAD_META_FAILED`	`-3040`	加载元信息失败
`E_FAILED_TO_CHECKPOINT`	`-3041`	生成 checkpoint 失败
`E_CHECKPOINT_BLOCKED`	`-3042`	生成 checkpoint 被阻塞
`E_FILTER_OUT`	`-3043`	数据被过滤
`E_INVALID_DATA`	`-3044`	无效的数据
`E_MUTATE_EDGE_CONFLICT`	`-3045`	并发写入同一条边发生冲突
`E_MUTATE_TAG_CONFLICT`	`-3046`	并发写入同一个点发生冲突
`E_OUTDATED_LOCK`	`-3047`	锁已经失效
`E_INVALID_TASK_PARA`	`-3051`	无效的任务参数
`E_USER_CANCEL`	`-3052`	用户取消了任务
`E_TASK_EXECUTION_FAILED`	`-3053`	任务执行失败
`E_PLAN_IS_KILLED`	`-3060`	执行计划被清除
`E_NO_TERM`	`-3070`	收到请求时心跳流程未完成
`E_OUTDATED_TERM`	`-3071`	收到旧 leader 的过时心跳（已选举出新的 leader）
`E_WRITE_WRITE_CONFLICT`	`-3073`	并发写入时与后到的请求发生冲突
`E_RAFT_UNKNOWN_PART`	`-3500`	未知的分区
`E_RAFT_LOG_GAP`	`-3501`	raft 日志落后
`E_RAFT_LOG_STALE`	`-3502`	raft 日志过期
`E_RAFT_TERM_OUT_OF_DATE`	`-3503`	心跳信息已经过期
`E_RAFT_UNKNOWN_APPEND_LOG`	`-3504`	未知的追加日志
`E_RAFT_WAITING_SNAPSHOT`	`-3511`	等待快照完成
`E_RAFT_SENDING_SNAPSHOT`	`-3512`	发送快照过程出错
`E_RAFT_INVALID_PEER`	`-3513`	无效的接收端
`E_RAFT_NOT_READY`	`-3514`	Raft 没有启动
`E_RAFT_STOPPED`	`-3515`	Raft 已经停止
`E_RAFT_BAD_ROLE`	`-3516`	错误的角色
`E_RAFT_WAL_FAIL`	`-3521`	写入 WAL 失败

错误名称	错误码	说明
`E_RAFT_HOST_STOPPED`	`-3522`	主机已经停止
`E_RAFT_TOO_MANY_REQUESTS`	`-3523`	请求数量过多
`E_RAFT_PERSIST_SNAPSHOT_FAILED`	`-3524`	持久化快照失败
`E_RAFT_RPC_EXCEPTION`	`-3525`	RPC 异常
`E_RAFT_NO_WAL_FOUND`	`-3526`	没有发现 WAL 日志
`E_RAFT_HOST_PAUSED`	`-3527`	主机暂停
`E_RAFT_WRITE_BLOCKED`	`-3528`	写入被堵塞
`E_RAFT_BUFFER_OVERFLOW`	`-3529`	缓存溢出
`E_RAFT_ATOMIC_OP_FAILED`	`-3530`	原子操作失败
`E_LEADER_LEASE_FAILED`	`-3531`	leader 租约过期
`E_RAFT_CAUGHT_UP`	`-3532`	Raft 已经同步数据
`E_STORAGE_MEMORY_EXCEEDED`	`-3600`	Storage 内存超出
`E_LOG_GAP`	`-4001`	drainer 日志落后
`E_LOG_STALE`	`-4002`	drainer 日志过期
`E_INVALID_DRAINER_STORE`	`-4003`	drainer 数据存储无效
`E_SPACE_MISMATCH`	`-4004`	图空间不匹配
`E_PART_MISMATCH`	`-4005`	分区不匹配
`E_DATA_CONFLICT`	`-4006`	数据冲突
`E_REQ_CONFLICT`	`-4007`	请求冲突
`E_DATA_ILLEGAL`	`-4008`	数据非法
`E_CACHE_CONFIG_ERROR`	`-5001`	缓存配置错误
`E_NOT_ENOUGH_SPACE`	`-5002`	空间不足
`E_CACHE_MISS`	`-5003`	没有命中缓存
`E_POOL_NOT_FOUND`	`-5005`	写缓存失败
`E_NODE_NUMBER_EXCEED_LIMIT`	`-7001`	机器节点数超出限制
`E_PARSING_LICENSE_FAILURE`	`-7002`	解析证书失败
`E_UNKNOWN`	`-8000`	未知错误

最后更新: 2025年8月4日

https://docs.nebula-graph.com.cn/master