CirroData Graph V1.2 使用手册

北京东方国信科技股份有限公司

2021 年 12 月

目　录

一、Server

1 依赖

1.1 安装 JDK-1.8

CirroData Graph 基于 jdk-1.8 开发，代码用到了较多 jdk-1.8 中的类和方法，请用户自行安装配置。

在往下阅读之前务必执行 `java -version` 命令查看 jdk 版本

1.2 安装 GCC-4.3.0（GLIBCXX_3.4.10）或更新版本

在往下阅读之前务必执行 `gcc --version` 命令查看 gcc 版本

2.部署

a. **单机版部署：**

单机版只需安装 CirroData Graph 一个组件即可, 安装部署参考《CirroData Graph Sever V1.2 安装配置手册》

b. **集群版部署：**

集群版包含 PD，CirroData Graph 以及 CirroDS 三个组件

PD：主要负责整个集群的管理调度，需首先进行安装，安装部署参考《CirroData Graph PD V1.2 安装配置手册》

CirroData Graph：安装部署参考《CirroData Graph Sever V1.2 安装配置手册》

CirroDS：图数据库的存储组件，增加 CirroDS 节点可对 CirroData Graph 进行动态的扩容，可根据实际情况来衡量是否部署 CirroDS。安装部署参考《CirroData Graph CirroDS V1.2 安装配置手册》

3.请求 server

CirroData Graph 的 RESTful API 包括多种类型的资源，典型的包括：graph、schema、gremlin、traverser 和 task

- `graph` 包含 `vertices、edges`

- `schema` 包含 `vertexlabels`、 `propertykeys`、 `edgelabels`、 `indexlabels`
- `gremlin` 包含各种 `Gremlin` 语句，如 `g.v(),` 可以同步或者异步执行
- `traverser` 包含各种高级查询，包括最短路径、交叉点、N 部可达邻居
- `task` 包含异步任务的查询和删除

二、loader

1.概述

Loader 是 CirroData Graph 的数据导入组件，能够将多种数据源的数据转化为图的顶点和边并批量导入到图数据库中。

目前支持的数据源包括：

- 本地磁盘文件或目录，支持 TEXT、CSV 和 JSON 格式的文件，支持压缩文件
- HDFS 文件或目录，支持压缩文件
- 主流关系型数据库，如 MySQL、PostgreSQL、Oracle、SQL Server

2.安装使用

参考《Loader 使用手册》

三、Graph Developer

1.概述

Graph Developer 是 CirroData Graph 的前端展示工具，是基于 Web 的图形化 IDE 环境。通过 Graph Developer，用户可以执行 Gremlin 语句，并及时获得图形化的展示结果。功能包括：

- 图数据的输入
- 图数据的展示

- 图数据的分析
- schema 的创建管理
- 数据的备份恢复
- 数据导入
- 应用算法库

2.安装使用

参考《CirroData Graph Developer 安装配置手册》

四、client

1.概述

Client 向 CirroData Graph 发出 HTTP 请求，获取并解析 Server 的执行结果。目前仅有 Java 版，用户可以使用 Client 编写 Java 代码操作 CirroData Graph，比如元数据和图数据的增删改查，或者执行 gremlin 语句。

2.环境要求

JDK1.8

3.示例

3.1 maven 依赖

```
<dependencies>

<dependency>

<groupId>com.baidu.hugegraph</groupId>

<artifactId>hugegraph-client</artifactId>
```

```
<version>1.8.0</version>
</dependency>
</dependencies>
```

3.2 Example

3.2.1 SingleExample

```java
import java.io.IOException;
import java.util.Iterator;
import java.util.List;
import com.baidu.hugegraph.driver.GraphManager;
import com.baidu.hugegraph.driver.GremlinManager;
import com.baidu.hugegraph.driver.HugeClient;
import com.baidu.hugegraph.driver.SchemaManager;
import com.baidu.hugegraph.structure.constant.T;
import com.baidu.hugegraph.structure.graph.Edge;
import com.baidu.hugegraph.structure.graph.Path;
import com.baidu.hugegraph.structure.graph.Vertex;
import com.baidu.hugegraph.structure.gremlin.Result;
import com.baidu.hugegraph.structure.gremlin.ResultSet;
public class SingleExample {
public static void main(String[] args) throws IOException {
// If connect failed will throw a exception.
HugeClient hugeClient = new HugeClient("http://localhost:8080", "hugegraph");
SchemaManager schema = hugeClient.schema();
schema.propertyKey("name").asText().ifNotExist().create();
schema.propertyKey("age").asInt().ifNotExist().create();
schema.propertyKey("city").asText().ifNotExist().create();
schema.propertyKey("weight").asDouble().ifNotExist().create();
schema.propertyKey("lang").asText().ifNotExist().create();
schema.propertyKey("date").asText().ifNotExist().create();
schema.propertyKey("price").asInt().ifNotExist().create();
schema.vertexLabel("person")
.properties("name", "age", "city")
.primaryKeys("name")
.ifNotExist()
.create();
schema.vertexLabel("software")
.properties("name", "lang", "price")
.primaryKeys("name")
.ifNotExist()
.create();
schema.indexLabel("personByCity")
.onV("person")
.by("city")
.secondary()
.ifNotExist()
.create();
schema.indexLabel("personByAgeAndCity")
.onV("person")
.by("age", "city")
.secondary()
.ifNotExist()
.create();
schema.indexLabel("softwareByPrice")
.onV("software")
.by("price")
.range()
```

```
.ifNotExist()
.create();
schema.edgeLabel("knows")
.sourceLabel("person")
.targetLabel("person")
.properties("date", "weight")
.ifNotExist()
.create();
schema.edgeLabel("created")
.sourceLabel("person").targetLabel("software")
.properties("date", "weight")
.ifNotExist()
.create();
schema.indexLabel("createdByDate")
.onE("created")
.by("date")
.secondary()
.ifNotExist()
.create();
schema.indexLabel("createdByWeight")
.onE("created")
.by("weight")
.range()
.ifNotExist()
.create();
schema.indexLabel("knowsByWeight")
.onE("knows")
.by("weight")
.range()
.ifNotExist()
```

```
.create();
GraphManager graph = hugeClient.graph();
Vertex marko = graph.addVertex(T.label, "person", "name", "marko",
"age", 29, "city", "Beijing");
Vertex vadas = graph.addVertex(T.label, "person", "name", "vadas",
"age", 27, "city", "Hongkong");
Vertex lop = graph.addVertex(T.label, "software", "name", "lop",
"lang", "java", "price", 328);
Vertex josh = graph.addVertex(T.label, "person", "name", "josh",
"age", 32, "city", "Beijing");
Vertex ripple = graph.addVertex(T.label, "software", "name", "ripple",
"lang", "java", "price", 199);
Vertex peter = graph.addVertex(T.label, "person", "name", "peter",
"age", 35, "city", "Shanghai");
marko.addEdge("knows", vadas, "date", "20160110", "weight", 0.5);
marko.addEdge("knows", josh, "date", "20130220", "weight", 1.0);
marko.addEdge("created", lop, "date", "20171210", "weight", 0.4);
josh.addEdge("created", lop, "date", "20091111", "weight", 0.4);
josh.addEdge("created", ripple, "date", "20171210", "weight", 1.0);
peter.addEdge("created", lop, "date", "20170324", "weight", 0.2);
GremlinManager gremlin = hugeClient.gremlin();
ResultSet resultSet = gremlin.gremlin("g.V().outE().path()").execute();
Iterator<Result> results = resultSet.iterator();
results.forEachRemaining(result -> {
System.out.println(result.getObject().getClass());
Object object = result.getObject();
if (object instanceof Vertex) {
System.out.println(((Vertex) object).id());
} else if (object instanceof Edge) {
System.out.println(((Edge) object).id());
} else if (object instanceof Path) {
List<Object> elements = ((Path) object).objects();
elements.forEach(element -> {
System.out.println(element.getClass());
System.out.println(element);
});
} else {
System.out.println(object);
}
});
}
}
```

3.2.2 BatchExample

```
import java.util.LinkedList;
import java.util.List;
import com.baidu.hugegraph.driver.GraphManager;
import com.baidu.hugegraph.driver.HugeClient;
import com.baidu.hugegraph.driver.SchemaManager;
import com.baidu.hugegraph.structure.graph.Edge;
import com.baidu.hugegraph.structure.graph.Vertex;
public class BatchExample {
public static void main(String[] args) {
// If connect failed will throw a exception.
HugeClient hugeClient = new HugeClient("http://localhost:8080", "hugegraph");
SchemaManager schema = hugeClient.schema();
schema.propertyKey("name").asText().ifNotExist().create();
schema.propertyKey("age").asInt().ifNotExist().create();
schema.propertyKey("lang").asText().ifNotExist().create();
schema.propertyKey("date").asText().ifNotExist().create();
schema.propertyKey("price").asInt().ifNotExist().create();
schema.vertexLabel("person")
.properties("name", "age")
.primaryKeys("name")
.ifNotExist()
.create();
schema.vertexLabel("person")
.properties("price")
.nullableKeys("price")
.append();
schema.vertexLabel("software")
.properties("name", "lang", "price")
.primaryKeys("name")
.ifNotExist()
.create();
schema.indexLabel("softwareByPrice")
.onV("software").by("price")
.range()
.ifNotExist()
.create();
schema.edgeLabel("knows")
.link("person", "person")
.properties("date")
.ifNotExist()
.create();
schema.edgeLabel("created")
.link("person", "software")
.properties("date")
.ifNotExist()
```

```
.create();
schema.indexLabel("createdByDate")
.onE("created").by("date")
.secondary()
.ifNotExist()
.create();
GraphManager graph = hugeClient.graph();
Vertex marko = new Vertex("person").property("name", "marko")
.property("age", 29);
Vertex vadas = new Vertex("person").property("name", "vadas")
.property("age", 27);
Vertex lop = new Vertex("software").property("name", "lop")
.property("lang", "java")
.property("price", 328);
Vertex josh = new Vertex("person").property("name", "josh")
.property("age", 32);
Vertex ripple = new Vertex("software").property("name", "ripple")
.property("lang", "java")
.property("price", 199);
Vertex peter = new Vertex("person").property("name", "peter")
.property("age", 35);
// Create a list to put vertex(Default max size is 500)
List<Vertex> vertices = new LinkedList<>();
vertices.add(marko);
vertices.add(vadas);
vertices.add(lop);
vertices.add(josh);
vertices.add(ripple);
vertices.add(peter);
// Post a vertex list to server
```

```
vertices = graph.addVertices(vertices);
vertices.forEach(vertex -> System.out.println(vertex));
Edge markoKnowsVadas = new Edge("knows").source(marko).target(vadas)
.property("date", "20160110");
Edge markoKnowsJosh = new Edge("knows").source(marko).target(josh)
.property("date", "20130220");
Edge markoCreateLop = new Edge("created").source(marko).target(lop)
.property("date", "20171210");
Edge joshCreateRipple = new Edge("created").source(josh).target(ripple)
.property("date", "20171210");
Edge joshCreateLop = new Edge("created").source(josh).target(lop)
.property("date", "20091111");
Edge peterCreateLop = new Edge("created").source(peter).target(lop)
.property("date", "20170324");
// Create a list to put edge(Default max size is 500)
List<Edge> edges = new LinkedList<>();
edges.add(markoKnowsVadas);
edges.add(markoKnowsJosh);
edges.add(markoCreateLop);
edges.add(joshCreateRipple);
edges.add(joshCreateLop);
edges.add(peterCreateLop);
// Post a edge list to server
edges = graph.addEdges(edges, false);
edges.forEach(edge -> System.out.println(edge));
}
}
```

五、ID 策略

1.VertexID 策略

CirroData Graph 的 Vertex 支持三种 ID 策略，在同一个数据库中不同的 VertexLabel 可以使用不同的 id 策略，目前支持的 id 策略分别是：

- 自动生成（AUTOMATIC）：使用 Snowflake 算法自动生成全局唯一 Id，Long 类型；
- 主键（PRIMARY_KEY）：通过 VertexLabel+PrimaryKeyValues 生成 Id，String 类型；
- 自定义（CUSTOMIZE_STRING|CUSTOMIZE_NUMBER）：用户自定义 Id，分为 String 和 Long 类型两种，需自己保证 Id 的唯一性；

默认的 Id 策略是 AUTOMATIC，如果用户调用 primaryKeys()方法并设置了正确的 PrimaryKeys，则自动启用 PRIMARY_KEY 策略。启用 PRIMARY_KEY 策略后可以根据 PrimaryKeys 实现数据去重

1) AUTOMATIC ID 策略

```
schema.vertexLabel("person")
.useAutomaticId()
.properties("name", "age", "city")
.create();
graph.addVertex(T.label, "person","name", "marko", "age", 18, "city", "Beijing");
```

2) PRIMARY_KEY 策略

```
schema.vertexLabel("person")
.usePrimaryKeyId()
.properties("name", "age", "city")
.primaryKeys("name", "age")
.create();
graph.addVertex(T.label, "person","name", "marko", "age", 18, "city", "Beijing");
```

3) CUSTOMIZE_STRING ID 策略

```
schema.vertexLabel("person")
.useCustomizeStringId()
.properties("name", "age", "city")
.create();
graph.addVertex(T.label, "person", T.id, "123456", "name", "marko","age", 18, "city",
"Beijing");
```

4) CUSTOMIZE_NUMBER ID 策略

```
schema.vertexLabel("person")
.useCustomizeNumberId()
.properties("name", "age", "city")
.create();
graph.addVertex(T.label, "person", T.id, 123456, "name", "marko","age", 18, "city",
"Beijing");
```

如果用户需要 Vertex 去重，有三种方案：

- 采用 PRIMARY_KEY 策略，自动覆盖，适合大数据量批量插入，用户无法知道是否发生了覆盖行为
- 采用 AUTOMATIC 策略，read-and-modify，适合小数据量插入，用户可以明确知道是否发生覆盖
- 采用 CUSTOMIZE_STRING 或 CUSTOMIZE_NUMBER 策略，用户自己保证唯一

2.EdgeId 策略

EdgeId 是由 srcVertexId+edgeLabel+sortKey+tgtVertexId 四部分组合而成。其中 sortKey 是一个重要概念。在 Edge 中加入 sortKey 作为 Edge 的唯一标识的原因有两个：

- 如果两个顶点之间存在多条相同 Label 的边可通过 sortKey 来区分

- 对于 SuperNode 的节点，可以通过 sortKey 来排序截断。

由于 EdgeId 是由 srcVertexId+edgeLabel+sortKey+tgtVertexId 四部分组合，多次插入相同的 Edge 时会自动覆盖以实现去重。需要注意的是如果批量插入模式下 Edge 的属性也将会覆盖。

另外由于 EdgeId 采用自动去重策略，对于 self-loop（一个顶点存在一条指向自身的边）的情况下会认为仅有一条边，对于采用 AUTOMATIC 策略的图数据库（例如 TitianDB）则会认为该图存在两条边。

六、Clients

1.Restful API

1.1 PropertyKey

1.1.1 创建一个 PropertyKey

Method & Url

```
POST http://localhost:8080/graphs/cirrograph/schema/propertykeys
```

Request Body

```
{
"name": "age",
"data_type": "INT",
"cardinality": "SINGLE"
}
```

Response Status

```
201
```

Response Body

```
{
"id": 2,
"name": "age",
"data_type": "INT",
"cardinality": "SINGLE",
"properties": [],
"user_data": {}
}
```

1.1.2 为已存在的 PropertyKey 添加或移除 userdata

Params

action：表示当前行为是添加还是移除，去取为 append（添加）和 eliminate（移除）

Method & Url

```
PUT
http://localhost:8080/graphs/cirrograph/schema/propertykeys/age?action=append
```

Request Body

```
{
"name": "age",
"user_data": {
"min": 0,
"max": 100
}
}
```

Response Status

```
201
```

Response Body

```
{
"id": 2,
"name": "age",
"data_type": "INT",
"cardinality": "SINGLE",
"properties": [],
"user_data": {
"min": 0,
"max": 100
}
}
```

1.1.3 获取所有的 PropertyKey

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/schema/propertykeys
```

Response Status

```
200
```

Response Body

```
{
"propertykeys": [
{
"id": 3,
"name": "city",
"data_type": "TEXT",
"cardinality": "SINGLE",
"properties": [],
"user_data": {}
},
{
"id": 2,
"name": "age",
"data_type": "INT",
"cardinality": "SINGLE",
"properties": [],
"user_data": {}
},
{
"id": 5,
"name": "lang",
"data_type": "TEXT",
"cardinality": "SINGLE",
"properties": [],
"user_data": {}
},
{
"id": 4,
"name": "weight",
"data_type": "DOUBLE",
"cardinality": "SINGLE",
"properties": [],
"user_data": {}
},
{
"id": 6,
"name": "date",
"data_type": "TEXT",
"cardinality": "SINGLE",
"properties": [],
"user_data": {}
},
{
"id": 1,
"name": "name",
"data_type": "TEXT",
"cardinality": "SINGLE",
"properties": [],
"user_data": {}
},
{
"id": 7,
"name": "price",
"data_type": "INT",
"cardinality": "SINGLE",
"properties": [],
"user_data": {}
}
]
}
```

1.1.4 根据 name 获取 PropertyKey

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/schema/propertykeys/age
```

其中 `age` 为要获取的 PropertyKey 的名字

Response Status

```
200
```

Response Body

```
{
"id": 2,
"name": "age",
"data_type": "INT",
"cardinality": "SINGLE",
"properties": [],
"user_data": {}
}
```

1.1.5 根据 name 删除 PropertyKey

Method & Url

```
DELETE http://localhost:8080/graphs/cirrograph/schema/propertykeys/age
```

其中 `age` 为要获取的 PropertyKey 的名字

Response Status

```
204
```

1.2 VertexLabel

假设已经创建好了 1.1.3 中列出来的 PropertyKeys

1.2.1 创建一个 VertexLabel

Method & Url

```
POST http://localhost:8080/graphs/cirrograph/schema/vertexlabels
```

Request Body

```
{
"name": "person",
"id_strategy": "DEFAULT",
"properties": [
"name",
"age"
],
"primary_keys": [
"name"
],
"nullable_keys": [],
"enable_label_index": true
}
```

Response Status

```
201
```

Response Body

```
{
"id": 1,
"primary_keys": [
"name"
],
"id_strategy": "PRIMARY_KEY",
"name": "person2",
"index_names": [
],
"properties": [
"name",
"age"
],
"nullable_keys": [
],
"enable_label_index": true,
"user_data": {}
}
```

1.2.2 为已存在的 VertexLabel 添加 properties 或者 userdata，或者移除 userdata

目前不支持移除 properties

Params

action：表示当前行为是添加还是移除，取值为 `append`（添加）和 `eliminate`（移除）

Method & Url

```
PUT
http://localhost:8080/graphs/cirrograph/schema/vertexlabels/person?action=append
```

Request Body

```
{
"name": "person",
"properties": [
"city"
],
"nullable_keys": ["city"],
"user_data": {
"super": "animal"
}
}
```

Response Status

```
200
```

Response Body

```
{
"id": 1,
"primary_keys": [
"name"
],
"id_strategy": "PRIMARY_KEY",
"name": "person",
"index_names": [
],
"properties": [
"city",
"name",
"age"
],
"nullable_keys": [
"city"
],
"enable_label_index": true,
"user_data": {
"super": "animal"
}
}
```

1.2.3 获取所有的 VertexLabel

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/schema/vertexlabels
```

Response Status

```
200
```

Response Body

```
{
"vertexlabels": [
{
"id": 1,
"primary_keys": [
"name"
],
"id_strategy": "PRIMARY_KEY",
"name": "person",
"index_names": [
],
"properties": [
"city",
"name",
"age"
],
"nullable_keys": [
"city"
],
"enable_label_index": true,
"user_data": {
"super": "animal"
}
},
{
"id": 2,
"primary_keys": [
"name"
],
"id_strategy": "PRIMARY_KEY",
"name": "software",
"index_names": [
],
"properties": [
"price",
"name",
"lang"
],
"nullable_keys": [
"price"
],
"enable_label_index": false,
"user_data": {}
}
]
}
```

1.2.4 根据 name 获取 VertexLabel

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/schema/vertexlabels/person
```

Response Status

```
200
```

Response Body

```
{
"id": 1,
"primary_keys": [
"name"
],
"id_strategy": "PRIMARY_KEY",
"name": "person",
"index_names": [
],
"properties": [
"city",
"name",
"age"
],
"nullable_keys": [
"city"
],
"enable_label_index": true,
"user_data": {
"super": "animal"
}
}
```

1.2.5 根据 name 删除 VertexLabel

删除 VertexLabel 会导致删除对应的顶点以及相关的索引数据，会产生一个异步任务

Method & Url

```
DELETE http://localhost:8080/graphs/cirrograph/schema/vertexlabels/person
```

Response Status

```
202
```

Response Bod

```
{
"task_id": 1
}
```

1.3 EdgeLabel

假设已经创建好了 1.1.3 中的 PropertyKeys 和 1.2.3 中的 VertexLabels

1.3.1 创建一个 EdgeLabel

Method & Url

```
POST http://localhost:8080/graphs/cirrograph/schema/edgelabels
```

Request Body

```
{
"name": "created",
"source_label": "person",
"target_label": "software",
"frequency": "SINGLE",
"properties": [
"date"
],
"sort_keys": [],
"nullable_keys": [],
"enable_label_index": true
}
```

Response Status

```
201
```

Response Body

```
{
"id": 1,
"sort_keys": [
],
"source_label": "person",
"name": "created",
"index_names": [
],
"properties": [
"date"
],
"target_label": "software",
"frequency": "SINGLE",
"nullable_keys": [
],
"enable_label_index": true,
"user_data": {}
}
```

1.3.2 为已存在的 EdgeLabel 添加 properties 或 userdata， 或者移除 userdata

目前不支持移除 properties

Params

action：表示当前行为是添加还是移除，取值为 `append`（添加）和 `eliminate`（移除）

Method & Url

```
PUT
http://localhost:8080/graphs/cirrograph/schema/edgelabels/created?action=append
```

Request Body

```
{
"name": "created",
"properties": [
"weight"
],
"nullable_keys": [
"weight"
]
}
```

Response Status

```
200
```

Response Body

```
{
```

```
"id": 2,
"sort_keys": [
],
"source_label": "person",
"name": "created",
"index_names": [
],
"properties": [
"date",
"weight"
],
"target_label": "software",
"frequency": "SINGLE",
"nullable_keys": [
"weight"
],
"enable_label_index": true,
"user_data": {}
}
```

1.3.3 获取所有的 EdgeLabel

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/schema/edgelabels
```

Response Status

```
200
```

Response Body

```
{
"edgelabels": [
{
"id": 1,
"sort_keys": [
],
"source_label": "person",
"name": "created",
"index_names": [
],
"properties": [
"date",
"weight"
],
"target_label": "software",
"frequency": "SINGLE",
"nullable_keys": [
"weight"
],
"enable_label_index": true,
"user_data": {}
},
{
"id": 2,
"sort_keys": [
],
"source_label": "person",
"name": "knows",
"index_names": [
],
"properties": [
"date",
"weight"
],
"target_label": "person",
"frequency": "SINGLE",
"nullable_keys": [
],
"enable_label_index": false,
"user_data": {}
}
]
}
```

1.3.4 根据 name 获取 EdgeLabel

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/schema/edgelabels/created
```

Response Status

```
200
```

Response Body

```
{
"id": 1,
"sort_keys": [
],
"source_label": "person",
"name": "created",
"index_names": [
],
"properties": [
"date",
"city",
"weight"
],
"target_label": "software",
"frequency": "SINGLE",
"nullable_keys": [
"city",
"weight"
],
"enable_label_index": true,
"user_data": {}
}
```

1.3.5 根据 name 删除 EdgeLabel

删除 EdgeLabel 会导致删除对应的边以及相关的索引数据，会产生一个异步任务

Method & Url

```
DELETE http://localhost:8080/graphs/cirrograph/schema/edgelabels/created
```

Response Status

```
202
```

Response Body

```
{
```

```
"task_id": 1
}
```

1.4 IndexLabel

假设已经创建好了 1.1.3 中的 PropertyKeys、1.2.3 中的 VertexLabel 已经 1.3.3 中的 EdgeLabel

1.4.1 创建一个 IndexLabel

Method & Url

```
POST http://localhost:8080/graphs/cirrograph/schema/indexlabels
```

Request Body

```
{
"name": "personByCity",
"base_type": "VERTEX_LABEL",
"base_value": "person",
"index_type": "SECONDARY",
"fields": [
"city"
]
}
```

Response Status

```
202
```

Response Body

```
{
"index_label": {
```

```
"id": 1,
"base_type": "VERTEX_LABEL",
"base_value": "person",
"name": "personByCity",
"fields": [
"city"
],
"index_type": "SECONDARY"
},
"task_id": 2
}
```

1.4.2 获取所有的 IndexLabel

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/schema/indexlabels
```

Response Status

```
200
```

Response Status

```
{
"indexlabels": [
{
"id": 3,
"base_type": "VERTEX_LABEL",
"base_value": "software",
"name": "softwareByPrice",
"fields": [
"price"
],
"index_type": "RANGE"
},
{
"id": 4,
"base_type": "EDGE_LABEL",
"base_value": "created",
"name": "createdByDate",
"fields": [
"date"
],
"index_type": "SECONDARY"
},
{
"id": 1,
"base_type": "VERTEX_LABEL",
"base_value": "person",
"name": "personByCity",
"fields": [
"city"
],
"index_type": "SECONDARY"
},
{
"id": 3,
"base_type": "VERTEX_LABEL",
"base_value": "person",
"name": "personByAgeAndCity",
"fields": [
"age",
"city"
],
"index_type": "SECONDARY"
}
]
}
```

1.4.3 根据 name 获取 IndexLabel

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/schema/indexlabels/personByCity
```

Response Status

```
200
```

Response Body

```
{
"id": 1,
"base_type": "VERTEX_LABEL",
"base_value": "person",
"name": "personByCity",
"fields": [
"city"
],
"index_type": "SECONDARY"
}
```

1.4.4 根据 name 删除 IndexLabel

删除 IndexLabel 会导致删除相关的索引数据，会产生一个异步任务

Method & Url

```
DELETE http://localhost:8080/graphs/cirrograph/schema/indexlabels/personByCity
```

Response Status

```
202
```

Response Body

```
{
"task_id": 1
}
```

1.5 Vertex

VertexLabel 中的 Id 策略决定了顶点的 Id 类型，其对应关系如下

Id_Strategy	id type
AUTOMATIC	number
PRIMARY_KEY	string
CUSTOMIZE_STRING	string
CUSTOMIZE_NUMBER	numbe

顶点的 GET/PUT/DELETE API 中 url 的 id 部分传入的应是带有类型信息的 id 值，这个类型信息用 json 串是否带引号表示，也就是说：

- 当 id 类型为 number 时，url 中的 id 不带引号，形如 xxx/vertices/123456
- 当 id 类型为 string 时，url 中的 id 带引号，形如 xxx/vertices/“123456”

假设已经创建好了前述的各种 schema 信息

1.5.1 创建一个顶点

Method & Url

```
POST http://localhost:8080/graphs/cirrograph/graph/vertices
```

Request Body

```
{
"label": "person",
"properties": {
"name": "marko",
"age": 29
}
}
```

Response Status

```
201
```

Response Body

```
{
"id": "1:marko",
"label": "person",
"type": "vertex",
"properties": {
```

```
"name": [
{
"id": "1:marko>name",
"value": "marko"
}
],
"age": [
{
"id": "1:marko>age",
"value": 29
}
]
}
}
```

1.5.2 创建多个顶点

Method & Url

```
POST http://localhost:8080/graphs/cirrograph/graph/vertices/batch
```

Request Body

```
[
{
"label": "person",
"properties": {
"name": "marko",
"age": 29
}
},
{
```

```
"label": "software",
"properties": {
"name": "ripple",
"lang": "java",
"price": 199
}
}
]
```

Response Status

```
201
```

Response Body

```
[
"1:marko",
"2:ripple"
]
```

1.5.3 更新顶点属性

Method & Url

```
PUT
http://127.0.0.1:8080/graphs/cirrograph/graph/vertices/"1:marko"?action=append
```

Request Body

```
{
"label": "person",
"properties": {
"age": 30,
"city": "Beijing"
}
}
```

Response Status

```
200
```

Response Body

```
{
"id": "1:marko",
"label": "person",
"type": "vertex",
"properties": {
"city": [
{
"id": "1:marko>city",
"value": "Beijing"
}
],
"name": [
{
"id": "1:marko>name",
"value": "marko"
}
],
"age": [
{
"id": "1:marko>age",
"value": 30
}
]
}
}
```

1.5.4 删除顶点属性

Method & Url

```
PUT
http://127.0.0.1:8080/graphs/cirrograph/graph/vertices/"1:marko"?action=eliminate
```

Request Body

```
{
"label": "person",
"properties": {
"city": "Beijing"
}
}
```

Response Status

```
200
```

Response Body

```
{
"id": "1:marko",
"label": "person",
"type": "vertex",
"properties": {
"name": [
{
"id": "1:marko>name",
"value": "marko"
}
],
"age": [
{
"id": "1:marko>age",
"value": 30
}
]
}
}
```

1.5.5 获取符合条件的顶点

Params

- label：顶点类型
- properties：属性键值对(根据属性查询的前提是建立了索引)
- limit：查询最大数目
- page：页号

以上参数都是可选的，如果提供 page 参数，必须提供 limit 参数，不允许带其他参数。label, properties 和 limit 可以任意组合。

a) 查询所有 age 为 20 且 label 为 person 的顶点

Method & Url

```
GET
http://localhost:8080/graphs/cirrograph/graph/vertices?label=person&properties={"age":29}&limit=1
```

Response Status

```
200
```

Response Body

```
{
"vertices": [
{
"id": "1:marko",
"label": "person",
"type": "vertex",
"properties": {
"city": [
{
"id": "1:marko>city",
"value": "Beijing"
}
],
"name": [
{
"id": "1:marko>name",
"value": "marko"
}
],
"age": [
{
"id": "1:marko>age",
"value": 29
}
]
}
}
]
}
```

b） 分页查询所有顶点，获取第一页（page 不带参数值），限定 3 条

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/graph/vertices?page&limit=3
```

Response Status

```
200
```

Response Body

```
{
"vertices": [{
"id": "2:ripple",
"label": "software",
"type": "vertex",
"properties": {
"price": [{
"id": "2:ripple>price",
"value": 199
}],
"name": [{
"id": "2:ripple>name",
"value": "ripple"
}],
"lang": [{
"id": "2:ripple>lang",
"value": "java"
```

```
}]
}
},
{
"id": "1:vadas",
"label": "person",
"type": "vertex",
"properties": {
"city": [{
"id": "1:vadas>city",
"value": "Hongkong"
}],
"name": [{
"id": "1:vadas>name",
"value": "vadas"
}],
"age": [{
"id": "1:vadas>age",
"value": 27
}]
}
},
{
"id": "1:peter",
"label": "person",
"type": "vertex",
"properties": {
"city": [{
"id": "1:peter>city",
"value": "Shanghai"
```

```
}],
"name": [{
"id": "1:peter>name",
"value": "peter"
}],
"age": [{
"id": "1:peter>age",
"value": 35
}]
}
}
],
"page":
"001000100853313a706574657200f07ffffffc00e797c6349be736fffc8699e8a502efe10004"
}
```

返回的 body 里面是带有下一页的页号信息的，`"page": "001000100853313a706574657200f07ffffffc00e797c6349be736fffc8699e8a502efe10004"`，在查询下一页的时候将该值赋给 page 参数。

c) 分页查询所有顶点，获取下一页（page 带上上一页返回的 page 值），限定 3 条

Method & Url

```
GET
http://localhost:8080/graphs/cirrograph/graph/vertices?page=001000100853313a706574657200f07ffffffc00e797c6349be736fffc8699e8a502efe10004&limit=3
```

Response Status

```
200
```

Response Body

```
{
"vertices": [{
"id": "1:josh",
"label": "person",
"type": "vertex",
"properties": {
"city": [{
"id": "1:josh>city",
"value": "Beijing"
}],
"name": [{
"id": "1:josh>name",
"value": "josh"
}],
"age": [{
"id": "1:josh>age",
"value": 32
}]
}
},
{
"id": "1:marko",
"label": "person",
"type": "vertex",
"properties": {
"city": [{
"id": "1:marko>city",
"value": "Beijing"
}],
"name": [{
"id": "1:marko>name",
"value": "marko"
}],
"age": [{
"id": "1:marko>age",
"value": 29
}]
}
},
{
"id": "2:lop",
"label": "software",
"type": "vertex",
"properties": {
"price": [{
"id": "2:lop>price",
"value": 328
}],
"name": [{
"id": "2:lop>name",
"value": "lop"
}],
"lang": [{
"id": "2:lop>lang",
"value": "java"
}]
}
}
```

```
],
"page": null
}
```

此时"page": null 表示已经没有下一页了。

1.5.6 根据 Id 获取顶点

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/graph/vertices/"1:marko"
```

Response Status

```
200
```

Response Body

```
{
"id": "1:marko",
"label": "person",
"type": "vertex",
"properties": {
"name": [
{
"id": "1:marko>name",
"value": "marko"
}
],
"age": [
{
"id": "1:marko>age",
"value": 29
}
]
}
}
```

1.5.7 根据 Id 删除顶点

Method & Url

```
DELETE http://localhost:8080/graphs/cirrograph/graph/vertices/"1:marko"
```

Response Status

```
204
```

1.6 Edge

顶点 id 格式的修改也影响到了边的 Id 以及源顶点和目标顶点 id 的格式。

EdgeId 是由 `src-vertex-id + direction + label + sort-values + tgt-vertex-id` 拼接而成，但是这里的顶点 id 类型不是通过引号区分的，而是根据前缀区分：

- 当 id 类型为 number 时，EdgeId 的顶点 id 前有一个前缀 L，形如 "L123456>1>>L987654"
- 当 id 类型为 string 时，EdgeId 的顶点 id 前有一个前缀 S，形如 "S1:peter>1>>S2:lop"

接下来的示例均假设已经创建好了前述的各种 schema 和 vertex 信息

1.6.1 创建一条边

Method & Url

```
POST http://localhost:8080/graphs/cirrograph/graph/edges
```

Request Body

```
{
"label": "created",
"outV": "1:peter",
"inV": "2:lop",
"outVLabel": "person",
"inVLabel": "software",
"properties": {
"date": "2017-5-18",
"weight": 0.2
}
}
```

Response Status

```
201
```

Response Body

```
{
"id": "S1:peter>1>>S2:lop",
"label": "created",
"type": "edge",
"inVLabel": "software",
"outVLabel": "person",
"inV": "2:lop",
"outV": "1:peter",
"properties": {
"date": "2017-5-18",
"weight": 0.2
}
}
```

1.6.2 创建多条边

Params

- check_vertex：是否检查顶点存在(true | false)，当设置为 true 而待插入边的源顶点或目标顶点不存在时会报错。

Method & Url

```
POST http://localhost:8080/graphs/cirrograph/graph/edges/batch
```

Request Body

```
[
{
"label": "created",
"outV": "1:peter",
"inV": "2:lop",
"outVLabel": "person",
"inVLabel": "software",
"properties": {
"date": "2017-5-18",
"weight": 0.2
}
},
{
"label": "knows",
"outV": "1:marko",
"inV": "1:vadas",
"outVLabel": "person",
"inVLabel": "person",
"properties": {
"date": "2016-01-10",
"weight": 0.5
}
}
]
```

Response Status

```
201
```

Response Body

```
[
"S1:peter>1>>S2:lop",
"S1:marko>2>>S1:vadas"
]
```

1.6.3 更新边属性

Method & Url

```
PUT
http://localhost:8080/graphs/cirrograph/graph/edges/S1:peter>1>>S2:lop?action=append
```

Request Body

```
{
"properties": {
"weight": 1.0
}
}
```

Response Status

```
200
```

Response Body

```
{
"id": "S1:peter>1>>S2:lop",
"label": "created",
"type": "edge",
"inVLabel": "software",
"outVLabel": "person",
"inV": "2:lop",
"outV": "1:peter",
"properties": {
"date": "2017-5-18",
"weight": 1
}
}
```

1.6.4 删除边属性

Method & Url

```
PUT
http://localhost:8080/graphs/cirrograph/graph/edges/S1:peter>1>>S2:lop?action=eliminate
```

Request Body

```
{
"properties": {
"weight": 1.0
}
}
```

Response Status

```
200
```

Response Body

```
{
"id": "S1:peter>1>>S2:lop",
"label": "created",
"type": "edge",
"inVLabel": "software",
"outVLabel": "person",
"inV": "2:lop",
"outV": "1:peter",
"properties": {
"date": "20170324"
}
}
```

1.6.5 获取符合条件的边

Params

- vertex_id：顶点 id
- direction：边的方向(OUT | IN | BOTH)
- label：边的标签
- properties：属性键值对(根据属性查询的前提是建立了索引)
- offset： 偏移，默认为 0
- limit：查询数目，默认为 100
- page：页号

支持的查询有以下几种：

- 提供 vertex_id 参数时，不可以使用参数 page，direction、label、properties 可选，offset 和 limit 可以限制结果范围
- 不提供 vertex_id 参数时，label 和 properties 可选
  - 如果使用 page 参数，则：offset 参数不可用（不填或者为 0），direction 不可用，properties 最多只能有一个
  - 如果不使用 page 参数，则：offset 和 limit 可以用来限制结果范围，direction 参数忽略

查询与顶点 person:josh(vertex_id=”1:josh”)相连且 label 为 created 的边

Method & Url

```
GET
http://127.0.0.1:8080/graphs/cirrograph/graph/edges?vertex_id="1:josh"&direction=BOTH
&label=created&properties={}
```

Response Status

```
200
```

Response Body

```
{
"edges": [
{
"id": "S1:josh>1>>S2:lop",
"label": "created",
"type": "edge",
"inVLabel": "software",
"outVLabel": "person",
"inV": "2:lop",
"outV": "1:josh",
"properties": {
"date": "20091111",
"weight": 0.4
}
},
{
"id": "S1:josh>1>>S2:ripple",
"label": "created",
"type": "edge",
"inVLabel": "software",
"outVLabel": "person",
"inV": "2:ripple",
"outV": "1:josh",
"properties": {
"date": "20171210",
"weight": 1
}
}
]
}
```

分页查询所有边，获取第一页（page 不带参数值），限定 3 条

Method & Url

```
GET http://127.0.0.1:8080/graphs/cirrograph/graph/edges?page&limit=3
```

Response Status

```
200
```

Response Body

```
{
"edges": [{
```

```
"id": "S1:peter>2>>S2:lop",
"label": "created",
"type": "edge",
"inVLabel": "software",
"outVLabel": "person",
"inV": "2:lop",
"outV": "1:peter",
"properties": {
"weight": 0.2,
"date": "20170324"
}
},
{
"id": "S1:josh>2>>S2:lop",
"label": "created",
"type": "edge",
"inVLabel": "software",
"outVLabel": "person",
"inV": "2:lop",
"outV": "1:josh",
"properties": {
"weight": 0.4,
"date": "20091111"
}
},
{
"id": "S1:josh>2>>S2:ripple",
"label": "created",
"type": "edge",
"inVLabel": "software",
"outVLabel": "person",
"inV": "2:ripple",
"outV": "1:josh",
"properties": {
"weight": 1,
"date": "20171210"
}
}
],
"page":
"002500100753313a6a6f73681210010004000000020953323a726970706c65f07ffffffcf07f
fffffd8460d63f4b398dd2721ed4fdb7716b420004"
}
```

返回的 body 里面是带有下一页的页号信息的，`"page": "002500100753313a6a6f73681210010004000000020953323a726970706c65f07ffffffcf07ffffffd8460d63f4b398dd2721ed4fdb7716b420004"`，在查询下一页的时候将该值赋给 page 参数。

分页查询所有边，获取下一页（page 带上上一页返回的 page 值），限定 3 条

Method & Url

```
GET
http://127.0.0.1:8080/graphs/cirrograph/graph/edges?page=002500100753313a6a6f736812
10010004000000020953323a726970706c65f07ffffffcf07ffffffd8460d63f4b398dd2721ed4f
db7716b420004&limit=3
```

Response Status

```
200
```

Response Body

```
{
"edges": [{
"id": "S1:marko>1>20130220>S1:josh",
"label": "knows",
"type": "edge",
"inVLabel": "person",
"outVLabel": "person",
"inV": "1:josh",
"outV": "1:marko",
"properties": {
"weight": 1,
"date": "20130220"
}
},
{
"id": "S1:marko>1>20160110>S1:vadas",
"label": "knows",
"type": "edge",
"inVLabel": "person",
"outVLabel": "person",
"inV": "1:vadas",
"outV": "1:marko",
"properties": {
"weight": 0.5,
"date": "20160110"
}
},
{
```

```
"id": "S1:marko>2>>S2:lop",
"label": "created",
"type": "edge",
"inVLabel": "software",
"outVLabel": "person",
"inV": "2:lop",
"outV": "1:marko",
"properties": {
"weight": 0.4,
"date": "20171210"
}
}
],
"page": null
}
```

此时`"page": null`表示已经没有下一页了。

1.6.6 根据 Id 获取边

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/graph/edges/S1:peter>1>>S2:lop
```

Response Status

```
200
```

Response Body

```
{
"id": "S1:peter>1>>S2:lop",
"label": "created",
"type": "edge",
"inVLabel": "software",
"outVLabel": "person",
"inV": "2:lop",
"outV": "1:peter",
"properties": {
"date": "2017-5-18",
"weight": 0.2
}
}
```

1.6.7 根据 Id 删除边

Method & Url

```
DELETE
http://localhost:8080/graphs/cirrograph/graph/edges/S1:peter>1>>S2:lop
```

Response Status

```
204
```

1.7 Traverser

1.7.1 概述

traverser API 实现了一些复杂的图算法，方便用户对图进行分析和挖掘，支持的 traverser API 包括：

- K-out API，根据起始顶点，查找恰好 N 步可达的邻居
- K-neighbor API，根据起始顶点，查找 N 步以内可达的所有邻居
- Shortest Path API，查找两个顶点之间的最短路径
- Paths API，查找两个顶点间的全部路径

- Customized Paths API，从一批顶点出发，按（一种）模式遍历经过的全部路径
- Crosspoints API，查找两个顶点的交点（共同祖先或者共同子孙）
- Customized Crosspoints API，从一批顶点出发，按多种模式遍历，最后一步到达的顶点的交点
- Rings API，从起始顶点出发，可到达的环路路径
- Rays API，从起始顶点出发，可到达边界的路径（即无环路径）
- Vertices API
  - 按 ID 批量查询顶点；
  - 获取顶点的分区；
  - 按分区查询顶点；
- Edges API
  - 按 ID 批量查询边；
  - 获取边的分区；
  - 按分区查询边；

1.7.2 K-out API

1.7.2.1 功能介绍

根据起始顶点、方向、边的类型（可选）和深度 depth，查找从起始顶点出发恰好 depth 步可达的顶点

Params

- source：起始顶点 id，必填项
- direction：起始顶点向外发散的方向（OUT,IN,BOTH），选填项，默认是 BOTH
- max_depth：步数，必填项
- label：边的类型，选填项，默认代表所有 edge label
- nearest：nearest 为 true 时，代表起始顶点到达结果顶点的最短路径长度为 depth，不存在更短的路径；nearest 为 false 时，代表起始顶点到结果顶点有一条长度为 depth 的路径（未必最短且可以有环），选填项，默认为 true

- max_degree：查询过程中，单个顶点最大边数目，选填项，默认为 10000
- capacity：遍历过程中最大的访问的顶点数目，选填项，默认为 10000000
- limit：返回的顶点的最大数目，选填项，默认为 10000000

1.7.2.2 使用方法

Method & Url

```
GET http://localhost:8080/graphs/{graph}/traversers/kout?source="1:marko"&max_depth=2
```

Response Status

```
200
```

Response Body

```
{
"vertices":[
"2:ripple",
"1:peter"
]
}
```

1.7.2.3 使用场景

查找恰好 N 步关系可达的顶点。两个例子：

- 家族关系中，查找一个人的所有孙子，person A 通过连续的两条“儿子”边到达的顶点集合。
- 社交关系中发现潜在好友，例如：与目标用户相隔两层朋友关系的用户，可以通过连续两条“朋友”边到达的顶点。

1.7.3 K-neighbor

1.7.3.1 功能介绍

根据起始顶点、方向、边的类型（可选）和深度 depth，查找包括起始顶点在内、depth 步之内可达的所有顶点

Params

- source：起始顶点 id，必填项
- direction：起始顶点向外发散的方向（OUT,IN,BOTH），选填项，默认是 BOTH
- max_depth：步数，必填项
- label：边的类型，选填项，默认代表所有 edge label
- max_degree：查询过程中，单个顶点最大边数目，选填项，默认为 10000
- limit：返回的顶点的最大数目，也即遍历过程中最大的访问的顶点数目，选填项，默认为 10000000

1.7.3.2 使用方法

Method & Url

```
GET
http://localhost:8080/graphs/{graph}/traversers/kneighbor?source=“1:marko”&max_depth=2
```

Response Status

```
200
```

Response Body

```
{
"vertices":[
    "2:ripple",
    "1:marko",
    "1:josh",
    "1:vadas",
    "1:peter",
    "2:lop"
  ]
}
```

1.7.3.3 使用场景

查找 N 步以内可达的所有顶点，例如：

- 家族关系中，查找一个人五服以内所有子孙，person A 通过连续的 5 条“亲子”边到达的顶点集合。
- 社交关系中发现好友圈子，例如目标用户通过 1 条、2 条、3 条“朋友”边可到达的用户可以组成目标用户的朋友圈子

1.7.4 Shortest Path

1.7.4.1 功能介绍

根据起始顶点、目的顶点、方向、边的类型（可选）和最大深度，查找一条最短路径

Params

- source：起始顶点 id，必填项
- target：目的顶点 id，必填项
- direction：起始顶点向外发散的方向（OUT,IN,BOTH），选填项，默认是 BOTH
- max_depth：最大步数，必填项
- label：边的类型，选填项，默认代表所有 edge label

- max_degree：查询过程中，单个顶点最大边数目，选填项，默认为 10000
- skip_degree：查询过程中需要跳过的顶点的最小的边数目，即当顶点的边数目大于 skip_degree 时，跳过该顶点，可用于规避超级点，选填项，默认为 0，表示不跳过任何点
- capacity：遍历过程中最大的访问的顶点数目，选填项，默认为 10000000

1.7.4.2 使用方法

Method & Url

```
GET http://localhost:8080/graphs/{graph}/traversers/shortestpath?source="1:marko"&target="2:ripple"&max_depth=3
```

Response Status

```
200
```

Response Body

```
{
"path":[
"1:marko",
"1:josh",
"2:ripple"
]
}
```

1.7.4.3 适用场景

查找两个顶点间的最短路径，例如：

- 社交关系网中，查找两个用户有关系的最短路径，即最近的朋友关系链
- 设备关联网络中，查找两个设备最短的关联关系

1.7.5 Paths

1.7.5.1 功能介绍

根据起始顶点、目的顶点、方向、边的类型（可选）和最大深度等条件查找所有路径

Params

- source：起始顶点 id，必填项
- target：目的顶点 id，必填项
- direction：起始顶点向外发散的方向（OUT,IN,BOTH），选填项，默认是 BOTH
- label：边的类型，选填项，默认代表所有 edge label
- max_depth：步数，必填项
- max_degree：查询过程中，单个顶点最大边数目，选填项，默认为 10000
- capacity：遍历过程中最大的访问的顶点数目，选填项，默认为 10000000
- limit：返回的路径的最大数目，选填项，默认为 10

1.7.5.2 使用方法

Method & Url

```
GET http://localhost:8080/graphs/{graph}/traversers/paths?source="1:marko"&target="1:josh"&max_depth=5
```

Response Status

```
200
```

Response Body

```
{
"paths":[
{
"objects":[
"1:marko",
"1:josh"
]
},
{
"objects":[
"1:marko",
"2:lop",
"1:josh"
]
}
]
}
```

1.7.5.3 使用场景

查找两个顶点间的所有路径，例如：

- 社交网络中，查找两个用户所有可能的关系路径
- 设备关联网络中，查找两个设备之间所有的关联路径

1.7.6 Crosspoints

1.7.6.1 功能介绍

根据起始顶点、目的顶点、方向、边的类型（可选）和最大深度等条件查找相交点

Params

- source：起始顶点 id，必填项
- target：目的顶点 id，必填项
- direction：起始顶点到目的顶点的方向，目的点到起始点是反方向，BOTH 时不考虑方向（OUT,IN,BOTH），选填项，默认是 BOTH
- label：边的类型，选填项，默认代表所有 edge label
- max_depth：步数，必填项
- max_degree：查询过程中，单个顶点最大边数目，选填项，默认为 10000
- capacity：遍历过程中最大的访问的顶点数目，选填项，默认为 10000000
- limit：返回的交点的最大数目，选填项，默认为 10

1.7.6.2 使用方法

Method & Url

```
GET http://localhost:8080/graphs/{graph}/traversers/crosspoints?source="2:lop"&target="2:ripple"&max_depth=5&direction=IN
```

Response Status

```
200
```

Response Body

```
{
"crosspoints":[
{
"crosspoint":"1:josh",
"objects":[
"2:lop",
"1:josh",
"2:ripple"
]
}
]
}
```

1.7.6.3 适用场景

查找两个顶点的交点及其路径，例如：

- 社交网络中，查找两个用户共同关注的话题或者大 V
- 家族关系中，查找共同的祖先

1.7.7 Rings

1.7.7.1 功能介绍

根据起始顶点、方向、边的类型（可选）和最大深度等条件查找可达的环路

例如：1 -> 25 -> 775 -> 14690 -> 25，其中环路为 25 -> 775 -> 14690 -> 25

Params

- source：起始顶点 id，必填项
- direction：起始顶点发出的边的方向（OUT,IN,BOTH），选填项，默认是 BOTH
- label：边的类型，选填项，默认代表所有 edge label
- max_depth：步数，必填项

- source_in_ring：环路是否包含起点，选填项，默认为 true
- max_degree：查询过程中，单个顶点最大边数目，选填项，默认为 10000
- capacity：遍历过程中最大的访问的顶点数目，选填项，默认为 10000000
- limit：返回的可达环路的最大数目，选填项，默认为 10

1.7.7.2 使用方法

Method & Url

```
GET http://localhost:8080/graphs/{graph}/traversers/rings?source="1:marko"&max_depth=2
```

Response Status

```
200
```

Response Body

```
{
"rings":[
{
"objects":[
"1:marko",
"1:josh",
"1:marko"
]
},
{
"objects":[
"1:marko",
"1:vadas",
"1:marko"
]
},
{
"objects":[
"1:marko",
"2:lop",
"1:marko"
]
}
]
}
```

1.7.7.3 适用场景

查询起始顶点可达的环路，例如：

- 风控项目中，查询一个用户可达的循环担保的人或者设备
- 设备关联网络中，发现一个设备周围的循环引用的设备

1.7.8 Rays

1.7.8.1 功能介绍

根据起始顶点、方向、边的类型（可选）和最大深度等条件查找发散到边界顶点的路径

例如：1 -> 25 -> 775 -> 14690 -> 2289 -> 18379，其中 18379 为边界顶点，即没有从 18379 发出的边

Params

- source：起始顶点 id，必填项

- direction：起始顶点发出的边的方向（OUT,IN,BOTH），选填项，默认是 BOTH
- label：边的类型，选填项，默认代表所有 edge label
- max_depth：步数，必填项
- max_degree：查询过程中，单个顶点最大边数目，选填项，默认为 10000
- capacity：遍历过程中最大的访问的顶点数目，选填项，默认为 10000000
- limit：返回的非环路的最大数目，选填项，默认为 10

1.7.8.2 使用方法

Method & Url

```
GET http://localhost:8080/graphs/{graph}/traversers/rays?source="1:marko"&max_depth=2&direction=OUT
```

Response Status

```
200
```

Response Body

```
{
"rays":[
{
"objects":[
"1:marko",
"1:vadas"
]
},
{
"objects":[
"1:marko",
"2:lop"
]
},
{
"objects":[
"1:marko",
"1:josh",
"2:ripple"
]
},
{
"objects":[
"1:marko",
"1:josh",
"2:lop"
]
}
]
}
```

1.7.8.3 适用场景

查找起始顶点到某种关系的边界顶点的路径，例如：

- 家族关系中，查找一个人到所有还没有孩子的子孙的路径
- 设备关联网络中，找到某个设备到终端设备的路径

1.7.9 Customized Paths

1.7.9.1 功能介绍

根据一批起始顶点、边规则（包括方向、边的类型和属性过滤）和最大深度等条件查找符合条件的所有的路径

Params

- sources：定义起始顶点，必填项，指定方式包括：
  - ids：通过顶点 id 列表提供起始顶点
  - labels 和 properties：如果没有指定 ids，则使用 label 和 properties 的联合条件查询起始顶点
    - labels：顶点的类型列表
    - properties：通过属性的值查询起始顶点

注意：properties 中的属性值可以是列表，表示只要 key 对应的 value 在列表中就可以

- steps：表示从起始顶点走过的路径规则，是一组 Step 的列表。必填项。每个 Step 的结构如下：
  - direction：表示边的方向（OUT,IN,BOTH），默认是 BOTH
  - labels：边的类型列表
  - properties：通过属性的值过滤边
  - weight_by：根据指定的属性计算边的权重，sort_by 不为 NONE 时有效，与 default_weight 互斥
  - default_weight：当边没有属性作为权重计算值时，采取的默认权重，sort_by 不为 NONE 时有效，与 weight_by 互斥
  - degree：查询过程中，单个顶点最大边数目，默认为 10000
  - sample：当需要对某个 step 的符合条件的边进行采样时设置，-1 表示不采样，默认为采样 100
- sort_by：根据路径的权重排序，选填项，默认为 NONE：
  - NONE 表示不排序，默认值
  - INCR 表示按照路径权重的升序排序

- DECR 表示按照路径权重的降序排序

- capacity：遍历过程中最大的访问的顶点数目，选填项，默认为 10000000
- limit：返回的路径的最大数目，选填项，默认为 10
- with_vertex：true 表示返回结果包含完整的顶点信息（路径中的全部顶点），false 时表示只返回顶点 id，选填项，默认为 false

1.7.9.2 使用方法

Method & Url

```
POST http://localhost:8080/graphs/{graph}/traversers/customizedpaths
```

Request Body

```
{
"sources":{
"ids":[
],
"label":"person",
"properties":{
"name":"marko"
}
},
"steps":[
{
"direction":"OUT",
"labels":[
"knows"
],
"weight_by":"weight",
"degree":-1
},
{
"direction":"OUT",
"labels":[
"created"
],
"default_weight":8,
"degree":-1,
"sample":1
}
],
"sort_by":"INCR",
"with_vertex":true,
"capacity":-1,
"limit":-1
}
```

Response Status

```
201
```

Response Body

```
{
"paths":[
{
"objects":[
"1:marko",
"1:josh",
"2:lop"
],
"weights":[
1,
8
]
}
],
"vertices":[
{
"id":"1:marko",
"label":"person",
"type":"vertex",
"properties":{
"city":[
{
"id":"1:marko>city",
"value":"Beijing"
}
],
"name":[
{
"id":"1:marko>name",
"value":"marko"
}
],
"age":[
{
"id":"1:marko>age",
"value":29
}
]
}
},
{
"id":"1:josh",
"label":"person",
"type":"vertex",
"properties":{
"city":[
{
"id":"1:josh>city",
"value":"Beijing"
}
],
"name":[
{
"id":"1:josh>name",
"value":"josh"
}
],
"age":[
{
"id":"1:josh>age",
"value":32
}
]
}
},
{
"id":"2:lop",
"label":"software",
```

适合查找各种复杂的路径集合，例如：

```
"type":"vertex",
"properties":{
"price":[
{
"id":"2:lop>price",
"value":328
}
],
"name":[
{
"id":"2:lop>name",
"value":"lop"
}
],
"lang":[
{
"id":"2:lop>lang",
"value":"java"
}
]
}
}
]
}
```

1.7.9.3 适用场景

适合查找各种复杂的路径集合，例如：

- 社交网络中，查找看过张艺谋所导演的电影的用户关注的大 V 的路径（张艺谋—->电影——>用户—->大 V）

- 风控网络中，查找多个高风险用户的直系亲属的朋友的路径（高风险用户—->直系亲属—->朋友）

1.7.10 Customized Crosspoints

1.7.10.1 功能介绍

根据一批起始顶点、多种边规则（包括方向、边的类型和属性过滤）和最大深度等条件查找符合条件的所有的路径终点的交集

Params

- sources：定义起始顶点，必填项，指定方式包括：
  - ids：通过顶点 id 列表提供起始顶点
  - labels 和 properties：如果没有指定 ids，则使用 label 和 properties 的联合条件查询起始顶点
    - labels：顶点的类型列表
    - properties：通过属性的值查询起始顶点

注意：properties 中的属性值可以是列表，表示只要 key 对应的 value 在列表中就可以

- path_patterns：表示从起始顶点走过的路径规则，是一组规则的列表。必填项。每个规则是一个 PathPattern
  - 每个 PathPattern 是一组 Step 列表，每个 Step 结构如下：
    - direction：表示边的方向（OUT,IN,BOTH），默认是 BOTH
    - labels：边的类型列表
    - properties：通过属性的值过滤边
    - degree：查询过程中，单个顶点最大边数目，默认为 10000
- capacity：遍历过程中最大的访问的顶点数目，选填项，默认为 10000000
- limit：返回的路径的最大数目，选填项，默认为 10
- with_path：true 表示返回交点所在的路径，false 表示不返回交点所在的路径，选填项，默认为 false
- with_vertex，选填项，默认为 false：

- true 表示返回结果包含完整的顶点信息（路径中的全部顶点）
  - with_path 为 true 时，返回所有路径中的顶点的完整信息
  - with_path 为 false 时，返回所有交点的完整信息
- false 时表示只返回顶点 id

1.7.10.2 使用方法

Method & Url

```
POST http://localhost:8080/graphs/{graph}/traversers/customizedcrosspoints
```

Request Body

```
{
"sources":{
"ids":[
"2:lop",
"2:ripple"
]
},
"path_patterns":[
{
"steps":[
{
"direction":"IN",
"labels":[
"created"
],
"degree":-1
}
]
}
],
"with_path":true,
"with_vertex":true,
"capacity":-1,
"limit":-1
}
```

Response Status

```
201
```

Response Body

```
{
"crosspoints":[
"1:josh"
],
"paths":[
{
"objects":[
"2:ripple",
"1:josh"
]
},
{
"objects":[
"2:lop",
"1:josh"
]
}
],
"vertices":[
{
"id":"2:ripple",
"label":"software",
"type":"vertex",
"properties":{
"price":[
{
"id":"2:ripple>price",
"value":199
}
],
"name":[
{
"id":"2:ripple>name",
"value":"ripple"
}
],
"lang":[
{
"id":"2:ripple>lang",
"value":"java"
}
]
}
},
{
"id":"1:josh",
"label":"person",
"type":"vertex",
"properties":{
"city":[
{
"id":"1:josh>city",
"value":"Beijing"
}
],
"name":[
{
"id":"1:josh>name",
"value":"josh"
}
],
"age":[
{
"id":"1:josh>age",
"value":32
}
]
}
},
{
"id":"2:lop",
"label":"software",
"type":"vertex",
"properties":{
"price":[
{
"id":"2:lop>price",
"value":328
}
],
"name":[
{
"id":"2:lop>name",
"value":"lop"
}
],
"lang":[
{
"id":"2:lop>lang",
"value":"java"
}
]
}
}
]
}
```

1.7.10.3 适用场景

查询一组顶点通过多种路径在终点有交集的情况。例如：

- 在商品图谱中，多款手机、学习机、游戏机通过不同的低级别的类目路径，最终都属于一级类目的电子设备

1.7.11 Vertices

1.7.11.1 根据顶点的 id 列表，批量查询顶点

Params

- ids：要查询的顶点 id 列表

Method & Url

```
GET
http://localhost:8080/graphs/cirrograph/traversers/vertices?ids="1:marko"&ids="2:lop"
```

Response Status

```
200
```

Response Body

```
{
"vertices":[
{
"id":"1:marko",
"label":"person",
"type":"vertex",
"properties":{
"city":[
{
"id":"1:marko>city",
"value":"Beijing"
}
],
"name":[
{
"id":"1:marko>name",
"value":"marko"
}
],
"age":[
{
```

```
"id":"1:marko>age",
"value":29
}
]
}
},
{
"id":"2:lop",
"label":"software",
"type":"vertex",
"properties":{
"price":[
{
"id":"2:lop>price",
"value":328
}
],
"name":[
{
"id":"2:lop>name",
"value":"lop"
}
],
"lang":[
{
"id":"2:lop>lang",
"value":"java"
}
]
}
```

```
}
]
}
```

1.7.11.2 获取顶点 Shard 信息

通过指定的分片大小 split_size，获取顶点分片信息

Params

- split_size：分片大小，必填项

Method & Url

```
GET
http://localhost:8080/graphs/cirrograph/traversers/vertices/shards?split_size=67108864
```

Response Status

```
200
```

Response Body

```
{
"shards":[
{
"start": "0",
"end": "2165893",
"length": 0
},
{
"start": "2165893",
"end": "4331786",
"length": 0
},
{
"start": "4331786",
"end": "6497679",
"length": 0
},
{
"start": "6497679",
"end": "8663572",
"length": 0
},
......
]
}
```

1.7.11.3 根据 Shard 信息批量获取顶点

通过指定的分片信息批量查询顶点。

Params

- start：分片起始位置，必填项
- end：分片结束位置，必填项
- page：分页位置，选填项，默认为 null，不分页；当 page 为“”时表示分页的第一页，从 start 指示的位置开始
- page_limit：分页获取顶点时，一页中顶点数目的上限，选填项，默认为 100000

Method & Url

```
GET
http://localhost:8080/graphs/cirrograph/traversers/vertices/scan?start=0&end=4294967295
```

Response Status

```
200
```

Response Body

```
{
"vertices":[
{
"id":"2:ripple",
"label":"software",
"type":"vertex",
"properties":{
"price":[
{
"id":"2:ripple>price",
"value":199
}
],
"name":[
{
"id":"2:ripple>name",
"value":"ripple"
}
],
"lang":[
{
"id":"2:ripple>lang",
"value":"java"
}
]
}
},
{
```

```
"id":"1:vadas",
"label":"person",
"type":"vertex",
"properties":{
"city":[
{
"id":"1:vadas>city",
"value":"Hongkong"
}
],
"name":[
{
"id":"1:vadas>name",
"value":"vadas"
}
],
"age":[
{
"id":"1:vadas>age",
"value":27
}
]
}
},
{
"id":"1:peter",
"label":"person",
"type":"vertex",
"properties":{
"city":[
{
"id":"1:peter>city",
"value":"Shanghai"
}
],
"name":[
{
"id":"1:peter>name",
"value":"peter"
}
],
"age":[
{
"id":"1:peter>age",
"value":35
}
]
}
},
{
"id":"1:josh",
"label":"person",
"type":"vertex",
"properties":{
"city":[
{
"id":"1:josh>city",
"value":"Beijing"
}
],
"name":[
{
"id":"1:josh>name",
"value":"josh"
}
],
"age":[
{
"id":"1:josh>age",
"value":32
}
]
}
},
{
"id":"1:marko",
"label":"person",
"type":"vertex",
"properties":{
"city":[
{
"id":"1:marko>city",
"value":"Beijing"
}
],
"name":[
{
"id":"1:marko>name",
"value":"marko"
}
```

```
],
"age":[
{
"id":"1:marko>age",
"value":29
}
]
}
},
{
"id":"2:lop",
"label":"software",
"type":"vertex",
"properties":{
"price":[
{
"id":"2:lop>price",
"value":328
}
],
"name":[
{
"id":"2:lop>name",
"value":"lop"
}
],
"lang":[
{
"id":"2:lop>lang",
"value":"java"
}
]
}
}
]
}
```

1.7.11.4 适用场景

- 按 id 列表查询顶点，可用于批量查询顶点，比如在 path 查询到多条路径之后，可以进一步查询某条路径的所有顶点属性。
- 获取分片和按分片查询顶点，可以用来遍历全部顶点

1.7.12 Edges

1.7.12.1 根据边的 id 列表，批量查询边

Params

- ids：要查询的边 id 列表

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/traversers/edges?ids="S1:josh>1>>S2:lop"&ids="S1:josh>1>>S2:ripple"
```

Response Status

```
200
```

Response Body

```
{
"edges": [
{
"id": "S1:josh>1>>S2:lop",
"label": "created",
"type": "edge",
"inVLabel": "software",
"outVLabel": "person",
"inV": "2:lop",
"outV": "1:josh",
"properties": {
"date": "20091111",
"weight": 0.4
}
},
{
"id": "S1:josh>1>>S2:ripple",
"label": "created",
"type": "edge",
"inVLabel": "software",
"outVLabel": "person",
"inV": "2:ripple",
"outV": "1:josh",
"properties": {
"date": "20171210",
"weight": 1
}
}
]
}
```

1.7.12.2 获取边 Shard 信息

通过指定的分片大小 split_size，获取边分片信息

Params

- split_size：分片大小，必填项

Method & Url

```
GET
http://localhost:8080/graphs/cirrograph/traversers/edges/shards?split_size=4294967295
```

Response Status

```
200
```

Response Body

```
{
"shards":[
{
"start": "0",
"end": "1073741823",
"length": 0
},
{
"start": "1073741823",
"end": "2147483646",
"length": 0
},
{
"start": "2147483646",
"end": "3221225469",
"length": 0
},
{
"start": "3221225469",
"end": "4294967292",
"length": 0
},
{
"start": "4294967292",
"end": "4294967295",
"length": 0
}
]
}
```

1.7.12.3 根据 Shard 信息批量获取边

通过指定的分片信息批量查询边

Params

- start：分片起始位置，必填项
- end：分片结束位置，必填项
- page：分页位置，选填项，默认为 null，不分页；当 page 为“”时表示分页的第一页，从 start 指示的位置开始
- page_limit：分页获取边时，一页中边数目的上限，选填项，默认为 100000

Method & Url

```
GET
http://localhost:8080/graphs/cirrograph/traversers/edges/scan?start=0&end=3221225469
```

Response Status

```
200
```

Response Body

```
{
"edges":[
{
"id":"S1:peter>2>>S2:lop",
"label":"created",
"type":"edge",
"inVLabel":"software",
"outVLabel":"person",
"inV":"2:lop",
"outV":"1:peter",
"properties":{
"weight":0.2,
"date":"20170324"
}
},
{
"id":"S1:josh>2>>S2:lop",
"label":"created",
"type":"edge",
"inVLabel":"software",
"outVLabel":"person",
"inV":"2:lop",
"outV":"1:josh",
"properties":{
"weight":0.4,
"date":"20091111"
}
},
{
"id":"S1:josh>2>>S2:ripple",
"label":"created",
"type":"edge",
"inVLabel":"software",
"outVLabel":"person",
"inV":"2:ripple",
"outV":"1:josh",
"properties":{
"weight":1,
"date":"20171210"
}
},
{
"id":"S1:marko>1>20130220>S1:josh",
"label":"knows",
"type":"edge",
"inVLabel":"person",
"outVLabel":"person",
"inV":"1:josh",
"outV":"1:marko",
"properties":{
"weight":1,
"date":"20130220"
}
},
{
"id":"S1:marko>1>20160110>S1:vadas",
"label":"knows",
"type":"edge",
"inVLabel":"person",
"outVLabel":"person",
"inV":"1:vadas",
"outV":"1:marko",
"properties":{
"weight":0.5,
"date":"20160110"
}
},
{
"id":"S1:marko>2>>S2:lop",
"label":"created",
"type":"edge",
"inVLabel":"software",
"outVLabel":"person",
"inV":"2:lop",
"outV":"1:marko",
"properties":{
"weight":0.4,
"date":"20171210"
}
}
]
}
```

1.7.12.4 适用场景

- 按 id 列表查询边，可用于批量查询边
- 获取分片和按分片查询边，可以用来遍历全部边

1.8 Rank

1.8.1 rank API 概述

`rank API` 可在图中为一个点推荐与其关系密切的其它点。

1.8.2 rank API 详解

1.8.2.1 Personal Rank API

1.8.2.1.1 功能介绍

适用于二分图，给出所有源顶点相关的其他顶点及其相关性组成的列表。

二分图：也称二部图，是图论里的一种特殊模型，也是一种特殊的网络流。其最大的特点在于，可以将图里的顶点分为两个集合，两个集合之间的点有边相连，但集合内的点之间没有直接关联。

假设有一个用户和物品的二分图，基于随机游走的 PersonalRank 算法步骤如下：

1. 选定一个起点用户 u，其初始权重为 1.0，从 Vu 开始游走（有 alpha 的概率走到邻居点，1 - alpha 的概率停留）；
2. 如果决定游走： 2.1. 那就从当前节点的邻居节点中按照均匀分布随机选择一个，并且按照均匀分布划分权重值； 2.2. 给源顶点补偿权重 1 - alpha； 2.3. 重复步骤 2；
3. 达到一定步数后收敛，得到推荐列表。

Params

- source：源顶点 id，必填项
- label：边的类型，必须是连接两类不同顶点的边，必填项
- alpha：每轮迭代时从某个点往外走的概率，与 PageRank 算法中的 alpha 类似，必填项，取值区间为 (0, 1]
- degree：查询过程中，单个顶点最大边数目，选填项，默认为 10000
- max_depth：迭代次数，必填项，取值区间为 (0, 50]

- with_label：筛选结果中保留哪些结果，选填项，可选值为 [SAME_LABEL, OTHER_LABEL, BOTH_LABEL]，默认为 BOTH_LABEL
    - SAME_LABEL：保留与源顶点相同类别的顶点
    - OTHER_LABEL：保留与源顶点不同类别（二分图的另一端）的顶点
    - BOTH_LABEL：保留与源顶点相同和相反类别的顶点
- limit：返回的顶点的最大数目，选填项，默认为 10000000
- sorted：返回的结果是否根据 rank 排序，为 true 时降序排列，反之不排序，选填项，默认为 true

1.8.2.1.2 使用方法

Method & Url

```
POST http://localhost:8080/graphs/cirrograph/traversers/personalrank
```

Request Body

```
{
"source": "1:1",
"label": "rating",
"alpha": 0.6,
"max_depth": 15,
"with_label": "OTHER_LABEL",
"sorted": true,
"limit": 10
}
```

Response Status

```
200
```

Response Body

```
{
"2:2858": 0.00050140260178169271,
"2:1196": 0.00043367083576536l7,
"2:1210": 0.0004128083140214213,
"2:593": 0.00038117341069881513,
"2:480": 0.00037005373269728036,
"2:1198": 0.000366641614652057,
"2:2396": 0.00036223624105388881,
"2:2571": 0.00035933124573009531,
"2:589": 0.00035922123055598566,
"2:110": 0.0003466135844390885
}
```

1.8.2.1.3 适用场景

两类不同顶点连接形成的二分图中，给某个点推荐相关性最高的其他顶点，例如：

- 商品推荐中，查找最应该给某人推荐的商品列表

1.8.2.2 Neighbor Rank API

1.8.2.2.1 功能介绍

在一般图结构中，找出每一层与给定起点相关性最高的前 N 个顶点及其相关度，用图的语义理解就是：从起点往外走，走到各层各个顶点的概率。

Params

- source：源顶点 id，必填项
- alpha：每轮迭代时从某个点往外走的概率，与 PageRank 算法中的 alpha 类似，必填项，取值区间为 (0, 1]
- steps：表示从起始顶点走过的路径规则，是一组 Step 的列表，每个 Step 对应结果中的一层，必填项。每个 Step 的结构如下：
  - direction：表示边的方向（OUT, IN, BOTH），默认是 BOTH

- labels：边的类型列表，多个边类型取并集
- degree：查询过程中，单个顶点最大边数目，默认为 10000
- top：在结果中每一层只保留权重最高的前 N 个结果，默认为 100，最大值为 1000

- capacity：遍历过程中最大的访问的顶点数目，选填项，默认为 10000000

1.8.2.2.2 使用方法

Method & Url

```
POST http://localhost:8080/graphs/cirrograph/traversers/neighborrank
```

Request Body

```
{
"source":"O",
"steps":[
{
"direction":"OUT",
"labels":[
"follow"
],
"degree":-1,
"top":100
},
{
"direction":"OUT",
"labels":[
"follow",
"like"
],
"degree":-1,
"top":100
},
{
"direction":"OUT",
"labels":[
"directedBy"
],
"degree":-1,
"top":100
}
],
"alpha":0.9,
"capacity":-1
}
```

Response Status

```
200
```

Response Body

```
{
"ranks": [
{
"O": 1
},
{
"B": 0.4305,
"A": 0.3,
"C": 0.3
},
{
"G": 0.17550000000000002,
"H": 0.17550000000000002,
"I": 0.135,
"J": 0.135,
"E": 0.09000000000000001,
"F": 0.09000000000000001
},
{
"M": 0.15795,
"K": 0.08100000000000002,
"L": 0.04050000000000001
}
]
}
```

1.8.2.2.3 适用场景

为给定的起点在不同的层中找到最应该推荐的顶点。

- 比如：在观众、朋友、电影、导演的四层图结构中，根据某个观众的朋友们喜欢的电影，为这个观众推荐电影；或者根据这些电影是谁拍的，为其推荐导演。

1.9 Variables

Variables 可以用来存储有关整个图的数据，数据按照键值对的方式存取

1.9.1 创建或者更新某个键值对

Method & Url

```
PUT http://localhost:8080/graphs/cirrograph/variables/name
```

Request Body

```
{
"data": "tom"
}
```

Response Status

```
200
```

Response Body

```
{
"name": "tom"
}
```

1.9.2 列出全部键值对

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/variables
```

Response Status

```
200
```

Response Body

```
{
"name": "tom"
}
```

1.9.3 列出某个键值对

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/variables/name
```

Response Status

```
200
```

Response Body

```
{
"name": "tom"
}
```

1.9.3 删除某个键值对

Method & Url

```
DELETE http://localhost:8080/graphs/cirrograph/variables/name
```

Response Status

```
204
```

1.10 Task

1.10.1 列出某个图中全部的异步任务

Params

- status：异步任务的状态
- limit：返回异步任务数目上限

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/tasks?status=success
```

Response Status

```
200
```

Response Body

```
{
"tasks": [{
"task_name": "cirrograph.traversal().V()",
"task_progress": 0,
"task_create": 1532943976585,
"task_status": "success",
"task_update": 1532943976736,
"task_result": "0",
"task_retries": 0,
"id": 2,
"task_type": "gremlin",
"task_callable": "com.baidu.cirrograph.api.job.GremlinAPI$GremlinJob",
"task_input":
"{\"gremlin\":\"cirrograph.traversal().V()\",\"bindings\":{},\"language\":\"gremlin-groovy\",\"
aliases\":{\"cirrograph\":\"graph\"}}"
}]
}
```

1.10.2 查看某个异步任务的信息

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/tasks/2
```

Response Status

```
200
```

Response Body

```
{
"task_name": "cirrograph.traversal().V()",
"task_progress": 0,
"task_create": 1532943976585,
"task_status": "success",
"task_update": 1532943976736,
"task_result": "0",
"task_retries": 0,
"id": 2,
"task_type": "gremlin",
"task_callable": "com.baidu.cirrograph.api.job.GremlinAPI$GremlinJob",
"task_input":
"{\"gremlin\":\"cirrograph.traversal().V()\",\"bindings\":{},\"language\":\"gremlin-groovy\",\"aliases\":{\"cirrograph\":\"graph\"}}"
}
```

1.11.1 重建 IndexLabel

Method & Url

```
PUT
http://localhost:8080/graphs/cirrograph/jobs/rebuild/indexlabels/personByCity
```

Response Status

```
202
```

Response Body

```
{
"task_id": 1
}
```

1.10.3 删除某个异步任务信息，不删除异步任务本身

Method & Url

```
DELETE http://localhost:8080/graphs/cirrograph/tasks/2
```

Response Status

```
204
```

1.10.4 取消某个异步任务，该异步任务必须具有处理中断的能力

假设已经通过 Gremlin API 创建了一个异步任务如下：

```
"for (int i = 0; i < 10; i++) {" +
"cirrograph.addVertex(T.label, 'man');" +
"cirrograph.tx().commit();" +
"try {" +
"sleep(1000);" +
"} catch (InterruptedException e) {" +
"break;" +
"}" +
"}"
```

Method & Url

```
PUT http://localhost:8080/graphs/cirrograph/tasks/2?action=cancel
```

Response Status

```
202
```

Response Body

```
{
```

```
"cancelled": true
}
```

1.11 Rebuild

1.11.2 VertexLabel 对应的全部索引重建

Method & Url

```
PUT http://localhost:8080/graphs/cirrograph/jobs/rebuild/vertexlabels/person
```

Response Status

```
202
```

Response Body

```
{
"task_id": 2
}
```

1.11.3 EdgeLabel 对应的全部索引重建

Method & Url

```
PUT http://localhost:8080/graphs/cirrograph/jobs/rebuild/edgelabels/created
```

Response Status

```
202
```

Response Body

```
{
"task_id": 3
}
```

1.12 Gremlin

1.12.1 向 Server 发送 gremlin 语句，同步执行

Params

- gremlin：要发送给 Server 执行的 gremlin 语句
- bindings：可以给 gremlin 语句中的变量绑定值
- language：发送语句的语言类型，默认为 gremlin-groovy
- aliases：为存在于图空间的已有变量添加别名

查询顶点

Method & Url

```
GET http://127.0.0.1:8080/gremlin?gremlin=cirrograph.traversal().V('1:marko')
```

Response Status

```
200
```

Response Body

```
{
"requestId": "c6ef47a8-b634-4b07-9d38-6b3b69a3a556",
"status": {
"message": "",
"code": 200,
"attributes": {}
},
"result": {
"data": [{
"id": "1:marko",
"label": "person",
"type": "vertex",
"properties": {
"city": [{
"id": "1:marko>city",
"value": "Beijing"
}],
"name": [{
"id": "1:marko>name",
"value": "marko"
}],
"age": [{
"id": "1:marko>age",
"value": 29
}]
}
}],
"meta": {}
}
}
```

1.12.2 向 Server 发送 gremlin 语句（POST），同步执行

Method & Url

```
POST http://localhost:8080/gremlin
```

查询顶点

```
{
"gremlin": "cirrograph.traversal().V('1:marko')",
"bindings": {},
"language": "gremlin-groovy",
"aliases": {}
}
```

Response Status

```
200
```

Response Body

```
{
"requestId": "c6ef47a8-b634-4b07-9d38-6b3b69a3a556",
"status": {
"message": "",
"code": 200,
"attributes": {}
},
"result": {
"data": [{
"id": "1:marko",
"label": "person",
"type": "vertex",
"properties": {
"city": [{
"id": "1:marko>city",
"value": "Beijing"
}],
"name": [{
"id": "1:marko>name",
"value": "marko"
}],
"age": [{
"id": "1:marko>age",
"value": 29
}]
}
}],
"meta": {}
}
}
```

查询边

Request Body

```
{
"gremlin": "g.E('S1:marko>2>>S2:lop')",
"bindings": {},
"language": "gremlin-groovy",
"aliases": {
"graph": "cirrograph",
"g": "__g_cirrograph"
}
}
```

Response Status

```
200
```

Response Body

```
{
"requestId": "3f117cd4-eedc-4e08-a106-ee01d7bb8249",
"status": {
```

```
"message": "",
"code": 200,
"attributes": {}
},
"result": {
"data": [{
"id": "S1:marko>2>>S2:lop",
"label": "created",
"type": "edge",
"inVLabel": "software",
"outVLabel": "person",
"inV": "2:lop",
"outV": "1:marko",
"properties": {
"weight": 0.4,
"date": "20171210"
}
}],
"meta": {}
}
}
```

1.12.3 向 Server 发送 gremlin 语句（POST），异步执行

Method & Url

```
POST http://localhost:8080/graphs/cirrograph/jobs/gremlin
```

查询顶点

Request Body

```
{
```

```
"gremlin": "g.V('1:marko')",
"bindings": {},
"language": "gremlin-groovy",
"aliases": {}
}
```

Response Status

```
200
```

Response Body

```
{
"task_id": 1
}
```

查询边

Request Body

```
{
"gremlin": "g.E('S1:marko>2>>S2:lop')",
"bindings": {},
"language": "gremlin-groovy",
"aliases": {}
}
```

Response Status

```
200
```

Response Body

```
{
"task_id": 2
}
```

1.13 节点信息查询

Method & Url

```
GET http://127.0.0.1:8080/graphs/cirrograph/monitor/store/detail
```

Response Status

```
200
```

Response Body

```
{
"状态":"正常",
"region 详情":[
{"region 所在节点":
["Peer{id=-1,storeId=-1,endpoint=127.0.0.1:8281}"],
"起始 Key":null,
"id":-1,
"regionEpoch":"RegionEpoch
{confVer=-1, version=-1}",
"结束 Key":null}
],
"region 个数":1
}
```

1.14 Authentication

用户认证与权限控制概述:

Cirrodata Graph 支持多用户认证、以及细粒度的权限访问控制，采用基于“用户-用户组-操作-资源”的 4 层设计，灵活控制用户角色与权限。 资源描述了图数据库中的数据，比如符合某一类条件的顶点，每一个资源包括 type、label、properties 三个要素，共有 18 种 type、 任意 label、任意 properties 的组合形成的资源，一个资源的内部条件是且关系，多个资源之间的条件是或关系。用户可以属于一个或多个用户组， 每个用户组可以拥有对任意个资源的操作权限，操作类型包括：读、写、删除、执行等种类。 Cirrodata Graph 支持动态创建用户、用户组、资源， 支持动态分配或取消权限。初始化数据库时超级管理员用户被创建，后续可通过超级管理员创建各类角色用户，新创建的用户如果被分配足够权限后，可以由其创建或管理更多的用户。

举例说明：

user(name=boss) -belong-> group(name=all) -access(read)->
target(graph=graph1, resource={label: person, city: Beijing})

描述：用户’boss’拥有对’graph1’图中北京人的读权限。

接口说明：

用户认证与权限控制接口包括 5 类：UserAPI、GroupAPI、TargetAPI、BelongAPI、AccessAPI。

1.14.1 用户（User）API

用户接口包括：创建用户，删除用户，修改用户，和查询用户相关信息接口。

1.14.1.1 创建用户

Params

- user_name：用户名称

- user_password：用户密码
- user_phone：用户手机号
- user_email：用户邮箱

其中 user_name 和 user_password 为必填。

Request Body

```
{
"user_name": "boss",
"user_password": "******",
"user_phone": "182****9088",
"user_email": "123@xx.com"
}
```

Method & Url

```
POST http://localhost:8080/graphs/cirrograph/auth/users
```

Response Status

```
201
```

Response Body

返回报文中，密码为加密后的密文

```
{
"user_password": "******",
"user_email": "123@xx.com",
"user_update": "2020-11-17 14:31:07.833",
"user_name": "boss",
"user_creator": "admin",
"user_phone": "182****9088",
"id": "-63:boss",
"user_create": "2020-11-17 14:31:07.833"
}
```

1.14.1.2 删除用户

Params

- id：需要删除的用户 Id

Method & Url

```
DELETE http://localhost:8080/graphs/cirrograph/auth/users/-63:test
```

Response Status

```
204
```

Response Body

```
1
```

1.14.1.3 修改用户

Params

- id：需要修改的用户 Id

Method & Url

```
PUT http://localhost:8080/graphs/cirrograph/auth/users/-63:test
```

Request Body

修改 user_name、user_password 和 user_phone

```
{
"user_name": "test",
"user_password": "******",
"user_phone": "183****9266"
}
```

Response Status

200

Response Body

返回结果是包含修改过的内容在内的整个用户组对象

```
{
"user_password": "******",
"user_update": "2020-11-12 10:29:30.455",
"user_name": "test",
"user_creator": "admin",
"user_phone": "183****9266",
"id": "-63:test",
"user_create": "2020-11-12 10:27:13.601"
}
```

1.14.1.4 查询用户列表

Params

- limit：返回结果条数的上限

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/auth/users
```

Response Status

```
200
```

Response Body

```
{
"users": [
{
"user_password": "******",
"user_update": "2020-11-11 11:41:12.254",
"user_name": "admin",
"user_creator": "system",
"id": "-63:admin",
"user_create": "2020-11-11 11:41:12.254"
}
]
}
```

1.14.1.5 查询某个用户

Params

- id：需要查询的用户 Id

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/auth/users/-63:admin
```

Response Status

```
200
```

Response Body

```
{
"users": [
{
"user_password": "******",
"user_update": "2020-11-11 11:41:12.254",
"user_name": "admin",
"user_creator": "system",
"id": "-63:admin",
"user_create": "2020-11-11 11:41:12.254"
}
]
}
```

1.14.1.6 查询某个用户的角色

Method & Url

GET http://localhost:8080/graphs/cirrograph/auth/users/-63:boss/role

Response Status

```
200
```

Response Body

```
{
"roles": {
"cirrograph": {
"READ": [
{
"type": "ALL",
"label": "*",
"properties": null
}
]
}
}
}
```

1.14.2 用户组（Group）API

用户组会赋予相应的资源权限，用户会被分配不同的用户组，即可拥有不同的资源权限。

用户组接口包括：创建用户组，删除用户组，修改用户组，和查询用户组相关信息接口。

1.14.2.1 创建用户组

Params

- group_name：用户组名称
- group_description：用户组描述

Request Body

```
{
 "group_name": "all",
 "group_description": "group can do anything"
}
```

Method & Url

```
POST http://localhost:8080/graphs/cirrograph/auth/groups
```

Response Status

```
201
```

Response Body

```
{
"group_creator": "admin",
"group_name": "all",
"group_create": "2020-11-11 15:46:08.791",
"group_update": "2020-11-11 15:46:08.791",
"id": "-69:all",
"group_description": "group can do anything"
}
```

1.14.2.2 删除用户组

Params

- id：需要删除的用户组 Id

Method & Url

```
DELETE http://localhost:8080/graphs/cirrograph/auth/groups/-69:grant
```

Response Status

```
204
```

Response Body

```
1
```

1.14.2.3 修改用户组

Params

- id：需要修改的用户组 Id

Method & Url

```
PUT http://localhost:8080/graphs/cirrograph/auth/groups/-69:grant
```

Request Body

修改 group_description

```
{
"group_name": "grant",
"group_description": "grant"
}
```

Response Status

```
200
```

Response Body

返回结果是包含修改过的内容在内的整个用户组对象

```
{
"group_creator": "admin",
"group_name": "grant",
"group_create": "2020-11-12 09:50:58.458",
"group_update": "2020-11-12 09:57:58.155",
"id": "-69:grant",
"group_description": "grant"
}
```

1.14.2.4 查询用户组列表

Params

- limit: 返回结果条数的上限

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/auth/groups
```

Response Status

```
200
```

Response Body

```
{
"groups": [
{
"group_creator": "admin",
"group_name": "all",
"group_create": "2020-11-11 15:46:08.791",
"group_update": "2020-11-11 15:46:08.791",
"id": "-69:all",
"group_description": "group can do anything"
}
]
}
```

系。

1.14.2.5 查询某个用户组

Params

- id：需要查询的用户组 Id

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/auth/groups/-69:all
```

Response Status

```
200
```

Response Body

```
{
"group_creator": "admin",
"group_name": "all",
"group_create": "2020-11-11 15:46:08.791",
"group_update": "2020-11-11 15:46:08.791",
"id": "-69:all",
"group_description": "group can do anything"
}
```

1.14.3 资源（Target）API

资源描述了图数据库中的数据，比如符合某一类条件的顶点，每一个资源包括 type、label、properties 三个要素，共有 18 种 type、 任意 label、任意 properties 的组合形成的资源，一个资源的内部条件是且关系，多个资源之间的条件是或关系。

资源接口包括：资源的创建、删除、修改和查询。

1.14.3.1 创建资源

Params

- target_name：资源名称
- target_graph：资源图
- target_url：资源地址
- target_resources：资源定义(列表)

target_resources 可以包括多个 target_resource，以列表的形式存储。
每个 target_resource 包含：

- type： 可选值 VERTEX, EDGE 等，可填 ALL，则表示可以是顶点或边；
- label： 可选值，一个顶点或边类型的名称，可填*，则表示任意类型；
- properties： map 类型，可包含多个属性的键值对，必须匹配所有属性值，属性值支持填条件范围（age: P.gte(18)），properties 如果为 null 表示任意属性均可，如果属性名和属性值均为‘*’也表示任意属性均可。

如精细资源：“target_resources”：
[{“type”：“VERTEX”，“label”：“person”，“properties”:{“city”：“Beijing”，“age”：“P.gte(20)”}}]**
资源定义含义：类型是’person’的顶点，且城市属性是’Beijing’，年龄属性大于等于 20。

Request Body

```
{
"target_name": "all",
"target_graph": "cirrograph",
"target_url": "127.0.0.1:8080",
"target_resources": [
{
"type": "ALL"
}
]
}
```

Method & Url

```
POST http://localhost:8080/graphs/cirrograph/auth/targets
```

Response Status

```
201
```

Response Body

```
{
"target_creator": "admin",
"target_name": "all",
"target_url": "127.0.0.1:8080",
"target_graph": "cirrograph",
"target_create": "2020-11-11 15:32:01.192",
"target_resources": [
{
"type": "ALL",
"label": "*",
"properties": null
}
],
"id": "-77:all",
"target_update": "2020-11-11 15:32:01.192"
}
```

1.14.3.2 删除资源

Params

- id：需要删除的资源 Id

Method & Url

```
DELETE http://localhost:8080/graphs/cirrograph/auth/targets/-77:gremlin
```

Response Status

```
204
```

Response Body

```
1
```

1.14.3.3 修改资源

Params

- id：需要修改的资源 Id

Method & Url

```
PUT http://localhost:8080/graphs/cirrograph/auth/targets/-77:gremlin
```

Request Body

修改资源定义中的 type

```
{
```

```
"target_name": "gremlin",
"target_graph": "cirrograph",
"target_url": "127.0.0.1:8080",
"target_resources": [
{
"type": "NONE"
}
]
}
```

Response Status

```
200
```

Response Body

返回结果是包含修改过的内容在内的整个用户组对象

```
{
"target_creator": "admin",
"target_name": "gremlin",
"target_url": "127.0.0.1:8080",
"target_graph": "cirrograph",
"target_create": "2020-11-12 09:34:13.848",
"target_resources": [
{
"type": "NONE",
"label": "*",
"properties": null
}
],
"id": "-77:gremlin",
"target_update": "2020-11-12 09:37:12.780"
}
```

1.14.3.4 查询资源列表

Params

- limit: 返回结果条数的上限

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/auth/targets
```

Response Status

```
200
```

Response Body

```
{
"targets": [
{
"target_creator": "admin",
"target_name": "all",
"target_url": "127.0.0.1:8080",
"target_graph": "cirrograph",
"target_create": "2020-11-11 15:32:01.192",
"target_resources": [
{
"type": "ALL",
"label": "*",
"properties": null
}
],
"id": "-77:all",
"target_update": "2020-11-11 15:32:01.192"
},
{
"target_creator": "admin",
"target_name": "grant",
"target_url": "127.0.0.1:8080",
"target_graph": "cirrograph",
"target_create": "2020-11-11 15:43:24.841",
"target_resources": [
{
"type": "GRANT",
"label": "*",
"properties": null
}
],
"id": "-77:grant",
"target_update": "2020-11-11 15:43:24.841"
```

```
}
]
}
```

1.14.3.5 查询某个资源

Params

- id：需要查询的资源 Id

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/auth/targets/-77:grant
```

Response Status

```
200
```

Response Body

```
{
"target_creator": "admin",
"target_name": "grant",
"target_url": "127.0.0.1:8080",
"target_graph": "cirrograph",
"target_create": "2020-11-11 15:43:24.841",
"target_resources": [
{
"type": "GRANT",
"label": "*",
"properties": null
}
],
"id": "-77:grant",
"target_update": "2020-11-11 15:43:24.841"
}
```

1.14.4 关联角色（Belong）API

关联用户和用户组的关系，一个用户可以关联一个或者多个用户组。用户组拥有相关资源的权限，不同用户组的资源权限可以理解为不同的角色。即给用户关联角色。

关联角色接口包括：用户关联角色的创建、删除、修改和查询。

1.14.4.1 创建用户的关联角色

Params

- user：用户 Id
- group：用户组 Id
- belong_description：描述

Request Body

```
{
"user": "-63:boss",
"group": "-69:all"
}
```

Method & Url

```
POST http://localhost:8080/graphs/cirrograph/auth/belongs
```

Response Status

```
201
```

Response Body

```
{
"belong_create": "2020-11-11 16:19:35.422",
"belong_creator": "admin",
"belong_update": "2020-11-11 16:19:35.422",
"id": "S-63:boss>-82>>S-69:all",
"user": "-63:boss",
"group": "-69:all"
}
```

1.14.4.2 删除关联角色

Params

- id：需要删除的关联角色 Id

Method & Url

```
DELETE
http://localhost:8080/graphs/cirrograph/auth/belongs/S-63:boss>-82>>S-69:grant
```

Response Status

```
204
```

Response Body

```
1
```

1.14.4.3 修改关联角色

关联角色只能修改描述，不能修改 user 和 group 属性，如果需要修改关联角色，需要删除原来关联关系，新增关联角色。

Params

- id：需要修改的关联角色 Id

Method & Url

```
PUT http://localhost:8080/graphs/cirrograph/auth/belongs/S-63:boss>-82>>S-69:grant
```

Request Body

修改 belong_description

```
{
"belong_description": "update test"
}
```

Response Status

```
200
```

Response Body

返回结果是包含修改过的内容在内的整个用户组对象

```
{
"belong_description": "update test",
"belong_create": "2020-11-12 10:40:21.720",
"belong_creator": "admin",
"belong_update": "2020-11-12 10:42:47.265",
"id": "S-63:boss>-82>>S-69:grant",
"user": "-63:boss",
"group": "-69:grant"
}
```

1.14.4.4 查询关联角色列表

Params

- limit: 返回结果条数的上限

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/auth/belongs
```

Response Status

```
200
```

Response Body

```
{
"belongs": [
{
"belong_create": "2020-11-11 16:19:35.422",
"belong_creator": "admin",
"belong_update": "2020-11-11 16:19:35.422",
"id": "S-63:boss>-82>>S-69:all",
"user": "-63:boss",
"group": "-69:all"
}
]
}
```

1.14.4.5 查看某个关联角色

Params

- id：需要查询的关联角色 Id

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/auth/belongs/S-63:boss>-82>>S-69:all
```

Response Status

```
200
```

Response Body

```
{
"belong_create": "2020-11-11 16:19:35.422",
"belong_creator": "admin",
"belong_update": "2020-11-11 16:19:35.422",
"id": "S-63:boss>-82>>S-69:all",
"user": "-63:boss",
"group": "-69:all"
}

```

1.14.5 赋权（Access）API

给用户组赋予资源的权限，主要包含：读操作(READ)、写操作(WRITE)、删除操作(DELETE)、执行操作(EXECUTE)等。

赋权接口包括：赋权的创建、删除、修改和查询。

1.14.5.1 创建赋权(用户组赋予资源的权限)

Params

- group：用户组 Id
- target：资源 Id
- access_permission：权限许可
- access_description：赋权描述

access_permission：

- READ：读操作，所有的查询，包括查询 Schema、查顶点/边，查询顶点和边的数量 VERTEX_AGGR/EDGE_AGGR，也包括读图的状态 STATUS、变量 VAR、任务 TASK 等；
- WRITE：写操作，所有的创建、更新操作，包括给 Schema 增加 property key，给顶点增加或更新属性等；
- DELETE：删除操作，包括删除元数据、删除顶点/边；
- EXECUTE：执行操作，包括执行Gremlin 语句、执行Task、执行metadata 函数；

Request Body

```
{
"group": "-69:all",
"target": "-77:all",
"access_permission": "READ"
}
```

Method & Url

```
POST http://localhost:8080/graphs/cirrograph/auth/accesses
```

Response Status

```
201
```

Response Body

```
{
"access_permission": "READ",
"access_create": "2020-11-11 15:54:54.008",
"id": "S-69:all>-88>11>S-77:all",
"access_update": "2020-11-11 15:54:54.008",
"access_creator": "admin",
"group": "-69:all",
"target": "-77:all"
}
```

1.14.5.2 删除赋权

Params

- id：需要删除的赋权 Id

Method & Url

```
DELETE http://localhost:8080/graphs/cirrograph/auth/accesses/S-69:all>-88>12>S-77:all
```

Response Status

```
204
```

Response Body

```
1
```

1.14.5.3 修改赋权

赋权只能修改描述，不能修改用户组、资源和权限许可，如果需要修改赋权的关系，可以删除原来的赋权关系，新增赋权。

Params

- id：需要修改的赋权 Id

Method & Url

```
PUT http://localhost:8080/graphs/cirrograph/auth/accesses/S-69:all>-88>12>S-77:all
```

Request Body

修改 access_description

```
{
"access_description": "test"
}
```

Response Status

```
200
```

Response Body

返回结果是包含修改过的内容在内的整个用户组对象

```
{
"access_description": "test",
"access_permission": "WRITE",
"access_create": "2020-11-12 10:12:03.074",
"id": "S-69:all>-88>12>S-77:all",
"access_update": "2020-11-12 10:16:18.637",
"access_creator": "admin",
"group": "-69:all",
"target": "-77:all"
}
```

1.14.5.4 查询赋权列表

Params

- limit：返回结果条数的上限

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/auth/accesses
```

Response Status

```
200
```

Response Body

```
{
"accesses": [
{
"access_permission": "READ",
"access_create": "2020-11-11 15:54:54.008",
"id": "S-69:all>-88>11>S-77:all",
"access_update": "2020-11-11 15:54:54.008",
"access_creator": "admin",
"group": "-69:all",
"target": "-77:all"
}
]
}
```

1.14.5.5 查询某个赋权

Params

- id：需要查询的赋权 Id

Method & Url

```
GET http://localhost:8080/graphs/cirrograph/auth/accesses/S-69:all>-88>11>S-77:all
```

Response Status

```
200
```

Response Body

```
{
"access_permission": "READ",
"access_create": "2020-11-11 15:54:54.008",
"id": "S-69:all>-88>11>S-77:all",
"access_update": "2020-11-11 15:54:54.008",
"access_creator": "admin",
"group": "-69:all",
"target": "-77:all"
}
```

2.Java Client

2.1 Client

Client 是操作 graph 的总入口，用户必须先创建出 Client 对象，与 Server 建立连接后，才能获取 schema、graph 以及 gremlin 的操作入口对象

```
// Server 地址："http://localhost:8080"
// 图的名称："cirrograph"
HugeClient hugeClient = new HugeClient("http://localhost:8080", "cirrograph");
```

上述创建 HugeClient 的过程如果失败会抛出异常，用户需要 try-catch。如果成功则继续获取 schema、graph 以及 gremlin 的 manager。

2.2 元数据

2.2.1 SchemaManager

SchemaManager 用于管理 Graph 中的四种元数据，分别是 PropertyKey（属性类型）、VertexLabel（顶点类型）、EdgeLabel（边类型）和 IndexLabel（索引标签）。在定义元数据信息之前必须先创建 SchemaManager 对象。

用户可使用如下方法获得 SchemaManager 对象：

```
SchemaManager schema = hugeClient.schema()
```

下面分别对三种元数据的定义过程进行介绍。

2.2.2 PropertyKey

2.2.2.1 接口以及参数介绍

PropertyKey 用来规范顶点和边的属性的约束，暂不支持定义属性的属性。

PropertyKey 允许定义的约束信息包括：name、datatype、cardinality、userdata，下面逐一介绍。

- name：属性的名字，用来区分不同的 PropertyKey，不允许有同名的属性；

interface	param	must set
propertyKey(String name)	name	y

- datatype：属性值类型，必须从下表中选择符合具体业务场景的一项显式设置；

interface	Java Class
asText()	String
asInt()	Integer
asDate()	Date
asUuid()	UUID
asBoolean()	Boolean
asByte()	Byte
asBlob()	Byte[]
asDouble()	Double
asFloat()	Float

interface	Java Class
asLong()	Long

- cardinality：属性值是单值还是多值，多值的情况下又分为允许有重复值和不允许有重复值，该项默认为 single，如有必要可从下表中选择一项设置；

interface	cardinality	description
valueSingle()	single	single value
valueList()	list	multi-values that allow duplicate value
valueSet()	set	multi-values that not allow duplicate value

- userdata：用户可以自己添加一些约束或额外信息，然后自行检查传入的属性是否满足约束，或者必要的时候提取出额外信息

interface	description
userdata(String key, Object value)	The same key, the latter will cover the former

2.2.2.2 创建 PropertyKey

```
schema.propertyKey("name").asText().valueSet().ifNotExist().create()
```

- ifNotExist()：为 create 添加判断机制，若当前 PropertyKey 已经存在则不再创建，否则创建该属性。若不添加判断，在 properkey 已存在的情况下会抛出异常信息，下同，不再赘述。

2.2.2.3 删除 PropertyKey

```
schema.propertyKey("name").remove()
```

2.2.2.4 查询 PropertyKey

```
// 获取 PropertyKey 对象
schema.getPropertyKey("name")
// 获取 PropertyKey 属性
schema.getPropertyKey("name").cardinality()
schema.getPropertyKey("name").dataType()
schema.getPropertyKey("name").name()
schema.getPropertyKey("name").userdata()
```

2.2.3 VertexLabel

2.2.3.1 接口及参数介绍

VertexLabel 用来定义顶点类型，描述顶点的约束信息：

VertexLabel 允许定义的约束信息包括：name、idStrategy、properties、primaryKeys 和 nullableKeys，下面逐一介绍。

- name：属性的名字，用来区分不同的 VertexLabel，不允许有同名的属性；

interface	param	must set
vertexLabel(String name)	name	y

- idStrategy：每一个 VertexLabel 都可以选择自己的 Id 策略，目前有三种策略供选择，即 Automatic（自动生成）、Customize（用户传入）和 PrimaryKey（主属性键）。其中 Automatic 使用 Snowflake 算法生成 Id，Customize 需要用户自行传入字符串或数字类型的 Id，PrimaryKey 则允许用户从 VertexLabel 的属性中选择若干主属性作为区分的依据，Cirrodata-graph

内部会根据主属性的值拼接生成 Id。idStrategy 默认使用 Automatic 的，但如果用户没有显式设置 idStrategy 又调用了 primaryKeys(…) 方法设置了主属性，则 idStrategy 将自动使用 PrimaryKey；

interface	idStrategy	description
useAutomaticId	AUTOMATIC	generate id automaticly by Snowflake algorithom
useCustomizeStringId	CUSTOMIZE_STRING	passed id by user, must be string type
useCustomizeNumberId	CUSTOMIZE_NUMBER	passed id by user, must be number type
usePrimaryKeyId	PRIMARY_KEY	choose some important prop as primary key to splice id

- properties：定义顶点的属性，传入的参数是 PropertyKey 的 name

interface	description
properties(String… properties)	allow to pass multi props

- primaryKeys：当用户选择了 PrimaryKey 的 Id 策略时，需要从 VertexLabel 的属性中选择若干主属性作为区分的依据；

interface	description
primaryKeys(String… keys)	allow to choose multi prop as primaryKeys

需要注意的是，Id 策略的选择与 primaryKeys 的设置有一些相互约束，不能随意调用，约束关系见下表：

	useAutomaticId	useCustomizeStringId	useCustomizeNumberId	usePrimaryKeyId
unset primaryKeys	AUTOMATIC	CUSTOMIZE_STRING	CUSTOMIZE_NUMBER	ERROR
set primaryKeys	ERROR	ERROR	ERROR	PRIMARY_KEY

- nullableKeys：对于通过 properties(…) 方法设置过的属性，默认全都是不可为空的，也就是在创建顶点时该属性必须赋值，这样可能对用户数据提出了太过严格的完整性要求。为避免这样的强约束，用户可以通过本方法设置若干属性为可空的，这样添加顶点时该属性可以不赋值。

interface	description
nullableKeys(String… properties)	allow to pass multi props

注意：primaryKeys 和 nullableKeys 不能有交集，因为一个属性不能既作为主属性，又是可空的。

- enableLabelIndex：用户可以指定是否需要为 label 创建索引。不创建则无法全局搜索指定 label 的顶点和边，创建则可以全局搜索，做类似于 g.V().hasLabel('person'), g.E().has('label', 'person')这样的查询，但是插入数据时性能上会更加慢，并且需要占用更多的存储空间。此项默认为 true。

interface	description
enableLabelIndex(boolean enable)	Whether to create a label index

- userdata：用户可以自己添加一些约束或额外信息，然后自行检查传入的属性是否满足约束，或者必要的时候提取出额外信息

interface	description
userdata(String key, Object value)	The same key, the latter will cover the former

2.2.3.2 创建 VertexLabel

```
// 使用 Automatic 的 Id 策略
schema.vertexLabel("person").properties("name", "age").ifNotExist().create();
schema.vertexLabel("person").useAutomaticId().properties("name",
"age").ifNotExist().create();
// 使用 Customize_String 的 Id 策略
schema.vertexLabel("person").useCustomizeStringId().properties("name",
"age").ifNotExist().create();
// 使用 Customize_Number 的 Id 策略
schema.vertexLabel("person").useCustomizeNumberId().properties("name",
"age").ifNotExist().create();
// 使用 PrimaryKey 的 Id 策略
schema.vertexLabel("person").properties("name",
"age").primaryKeys("name").ifNotExist().create();
schema.vertexLabel("person").usePrimaryKeyId().properties("name",
"age").primaryKeys("name").ifNotExist().create();
```

2.2.3.3 追加 VertexLabel

VertexLabel 是可以追加约束的，不过仅限 properties 和 nullableKeys，而且追加的属性也必须添加到 nullableKeys 集合中。

```
schema.vertexLabel("person").properties("price").nullableKeys("price").append()
;
```

2.2.3.4 删除 VertexLabel

```
schema.vertexLabel("person").remove();
```

2.2.3.5 查询 VertexLabel

```
// 获取 VertexLabel 对象
schema.getVertexLabel("name")
// 获取 property key 属性
schema.getVertexLabel("person").idStrategy()
schema.getVertexLabel("person").primaryKeys()
schema.getVertexLabel("person").name()
schema.getVertexLabel("person").properties()
schema.getVertexLabel("person").nullableKeys()
schema.getVertexLabel("person").userdata()
```

2.2.4 EdgeLabel

2.2.4.1 接口及参数介绍

EdgeLabel 用来定义边类型，描述边的约束信息。

EdgeLabel 允许定义的约束信息包括：name、sourceLabel、targetLabel、frequency、properties、sortKeys 和 nullableKeys，下面逐一介绍。

- name：属性的名字，用来区分不同的 EdgeLabel，不允许有同名的属性；

interface	param	must set
edgeLabel(String name)	name	y

- sourceLabel：边连接的源顶点类型名，只允许设置一个；
- targetLabel：边连接的目标顶点类型名，只允许设置一个；

interface	param	must set
sourceLabel(String label)	label	y
targetLabel(String label)	label	y

- frequency：字面意思是频率，表示在两个具体的顶点间某个关系出现的次数，可以是单次（single）或多次（frequency），默认为 single；

interface	frequency	description
singleTime()	single	a relationship can only occur once
multiTimes()	multiple	a relationship can occur many times

- properties：定义边的属性

interface	description
properties(String… properties)	allow to pass multi props

- sortKeys：当 EdgeLabel 的 frequency 为 multiple 时，需要某些属性来区分这多次的关系，故引入了 sortKeys（排序键）；

interface	description
sortKeys(String… keys)	allow to choose multi prop as sortKeys

- nullableKeys：与顶点中的 nullableKeys 概念一致，不再赘述

注意：sortKeys 和 nullableKeys 也不能有交集。

- enableLabelIndex：与顶点中的 enableLabelIndex 概念一致，不再赘述

- userdata：用户可以自己添加一些约束或额外信息，然后自行检查传入的属性是否满足约束，或者必要的时候提取出额外信息

interface	description
userdata(String key, Object value)	The same key, the latter will cover the former

2.2.4.2 创建 EdgeLabel

```
schema.edgeLabel("knows").link("person", "person").properties("date").ifNotExist().create();
schema.edgeLabel("created").multiTimes().link("person", "software").properties("date").sortKeys("date").ifNotExist().create();
```

2.2.4.3 追加 EdgeLabel

```
schema.edgeLabel("knows").properties("price").nullableKeys("price").append();
```

2.2.4.4 删除 EdgeLabel

```
schema.edgeLabel("knows").remove();
```

2.2.4.5 查询 EdgeLabel

```
// 获取 EdgeLabel 对象
schema.getEdgeLabel("knows")
// 获取 property key 属性
schema.getEdgeLabel("knows").frequency()
schema.getEdgeLabel("knows").sourceLabel()
schema.getEdgeLabel("knows").targetLabel()
schema.getEdgeLabel("knows").sortKeys()
schema.getEdgeLabel("knows").name()
schema.getEdgeLabel("knows").properties()
schema.getEdgeLabel("knows").nullableKeys()
schema.getEdgeLabel("knows").userdata()
```

2.2.5 IndexLabel

2.2.5.1 接口及参数介绍

IndexLabel 用来定义索引类型，描述索引的约束信息，主要是为了方便查询。

IndexLabel 允许定义的约束信息包括：name、baseType、baseValue、indexFeilds、indexType，下面逐一介绍。

- name：属性的名字，用来区分不同的 IndexLabel，不允许有同名的属性；

interface	param	must set
indexLabel(String name)	name	y

- baseType：表示要为 VertexLabel 还是 EdgeLabel 建立索引，与下面的 baseValue 配合使用；
- baseValue：指定要建立索引的 VertexLabel 或 EdgeLabel 的名称；

interface	param	description
onV(String baseValue)	baseValue	build index for VertexLabel: ‘baseValue’
onE(String baseValue)	baseValue	build index for EdgeLabel: ‘baseValue’

- indexFeilds：要在哪些属性上建立索引，可以是为多列建立联合索引；

interface	param	description
by(String… fields)	files	allow to build index for multi fields for secondary index

- indexType：建立的索引类型，目前支持五种，即 Secondary、Range、Search、Shard 和 Unique。
    - ◆ Secondary 支持精确匹配的二级索引，允许建立联合索引，联合索引支持索引前缀搜索
        - · 单个属性，支持相等查询，比如：person 顶点的 city 属性的二级索引，可以用 g.V().has("city", "北京")查询”city 属性值是北京”的全部顶点
        - · 联合索引，支持前缀查询和相等查询，比如：person 顶点的 city 和 street 属性的联合索引，可以用 g.V().has ("city", "北京").has('street', '中关村街道')查询”city 属性值是北京且 street 属性值是中关村”的全部顶点，或者 g.V() .has("city", "北京")查询”city 属性值是北京”的全部顶点
    - ◆ Range 支持数值类型的范围查询
        - · 必须是单个数字或者日期属性，比如：person 顶点的 age 属性的范围索引，可以用 g.V().has("age", P.gt(18))查询”age 属性值大于 18”的顶点。除了 P.gt()以外，还支持 P.gte(), P.lte(), P.lt(),P.eq(), P.between(), P.inside()和 P.outside()等
    - ◆ Search 支持全文检索的索引
        - · 必须是单个文本属性，比如：person 顶点的 address 属性的全文索引，可以用 g.V().has("address", Text .contains('大厦')查询”address 属性中包含大厦”的全部顶点
    - ◆ Shard 支持前缀匹配 + 数字范围查询的索引
        - · N 个属性的分片索引，支持前缀相等情况下的范围查询，比如：person 顶点的 city 和 age 属性的分片索引，可以用 g.V().has ("city", "北京").has("age", P.between(18, 30))查询”city 属性是北京且年龄大于等于 18 小于 30”的全部顶点
        - · shard index N 个属性全是文本属性时，等价于 secondary index
        - · shard index 只有单个数字或者日期属性时，等价于 range index

- Unique 支持属性值唯一性约束，即可以限定属性的值不重复，允许联合索引，但不支持查询
  - 单个或者多个属性的唯一性索引，不可用来查询，只可对属性的值进行限定，当出现重复值时将报错

interface	indexType	description
secondary()	Secondary	support prefix search
range()	Range	support range(numeric or date type) search
search()	Search	support full text search
shard()	Shard	support prefix + range(numeric or date type) search
unique()	Unique	support unique props value, not support search

2.2.5.2 创建 IndexLabel

```
schema.indexLabel("personByAge").onV("person").by("age").range().ifNotExist().create();
schema.indexLabel("createdByDate").onE("created").by("date").secondary().ifNotExist().create();
schema.indexLabel("personByLived").onE("person").by("lived").search().ifNotExist().create();
schema.indexLabel("personByCityAndAge").onV("person").by("city", "age").shard().ifNotExist().create();
schema.indexLabel("personById").onV("person").by("id").unique().ifNotExist().create();
```

2.2.5.3 删除 IndexLabel

```
schema.indexLabel("personByAge").remove()
```

2.2.5.4 查询 IndexLabel

```
// 获取 IndexLabel 对象
schema.getIndexLabel("personByAge")
// 获取 property key 属性
schema.getIndexLabel("personByAge").baseType()
schema.getIndexLabel("personByAge").baseValue()
schema.getIndexLabel("personByAge").indexFields()
schema.getIndexLabel("personByAge").indexType()
schema.getIndexLabel("personByAge").name()
```

2.3 图数据

2.3.1 Vertex

顶点是构成图的最基本元素，一个图中可以有非常多的顶点。下面给出一个添加顶点的例子：

```
Vertex marko = graph.addVertex(T.label, "person", "name", "marko", "age", 29);
Vertex lop = graph.addVertex(T.label, "software", "name", "lop", "lang", "java",
"price", 328);
```

- 添加顶点的关键是顶点属性，添加顶点函数的参数个数必须为偶数，且满足 key1 -> val1, key2 -> val2 ・・・的顺序排列，键值对之间的顺序是自由的。
- 参数中必须包含一对特殊的键值对，就是 T.label -> "val"，用来定义该顶点的类别，以便于程序从缓存或后端获取到该 VertexLabel 的 schema 定义，然后做后续的约束检查。例子中的 label 定义为 person。

- 如果顶点类型的 Id 策略为 AUTOMATIC，则不允许用户传入 id 键值对。
- 如果顶点类型的 Id 策略为 CUSTOMIZE_STRING，则用户需要自己传入 String 类型 id 的值，键值对形如："T.id", "123456"。
- 如果顶点类型的 Id 策略为 CUSTOMIZE_NUMBER，则用户需要自己传入 Number 类型 id 的值，键值对形如："T.id", 123456。
- 如果顶点类型的 Id 策略为 PRIMARY_KEY，参数还必须全部包含该 primaryKeys 对应属性的名和值，如果不设置会抛出异常。比如之前 person 的 primaryKeys 是 name，例子中就设置了 name 的值为 marko。
- 对于非 nullableKeys 的属性，必须要赋值。
- 剩下的参数就是顶点其他属性的设置，但并非必须。
- 调用 addVertex 方法后，顶点会立刻被插入到后端存储系统中。

2.3.2 Edge

有了点，还需要边才能构成完整的图。下面给出一个添加边的例子：

```
Edge knows1 = marko.addEdge("knows", vadas, "city", "Beijing");
```

- 由（源）顶点来调用添加边的函数，函数第一个参数为边的 label，第二个参数是目标顶点，这两个参数的位置和顺序是固定的。后续的参数就是 key1 -> val1, key2 -> val2 ・・・的顺序排列，设置边的属性，键值对顺序自由。
- 源顶点和目标顶点必须符合 EdgeLabel 中 sourcelabel 和 targetlabel 的定义，不能随意添加。
- 对于非 nullableKeys 的属性，必须要赋值。