ALIBABA CLOUD

阿里云

图数据库

SDK参考

文档版本：20220602

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.Gremlin SDK参考

1.1. Gremlin兼容性

图数据库GDB支持TinkerPop Gremlin查询语言。对于TinkerPop Gremlin查询语言，图数据库GDB Gremlin与TinkerPop Gremlin在实现细节上存在差异。本文主要介绍图数据库GDB对于TinkerPop Gremlin查询语言的兼容性。

版本兼容

- 图数据库GDB Gremlin兼容TinkerPop Gremlin 3.3.*x*以及3.4.*x*版本。
- 与图数据库GDB服务端进行交互时，使用GraphSON格式。

> **说明**
>
> GraphSON是Gremlin的标准格式，使用JSON格式表示点、边和属性。

DSL使用限制

图数据库GDB Gremlin在DSL的使用规则上与TinkerPop的差异如下：

- 所有的DSL必须以内置的变量g开头，该变量等同于TinkerPop中的 `Graph.traversal()` 。
- 不支持对于查询策略的控制，所有查询优化由图数据库GDB内置自动执行。
- 点和边的ID为字符串类型，可以由您指定。如果您没有指定，则自动生成UUID作为点或边的ID。
- 图数据库GDB的属性图模型中，目前支持Single、Set两种模式，即TinkerPop中的

```
org.apache.tinkerpop.gremlin.structure.VertexProperty.Cardinality.single;
org.apache.tinkerpop.gremlin.structure.VertexProperty.Cardinality.set
```

- 图数据库GDB属性值的类型当前仅支持简单类型（包括数字、字符串和布尔型，即Java编程语言中的byte、char、short、int、long、float、double、boolean和String）；不支持复杂类型（例如日期）。

Groovy限制

图数据库GDB Gremlin不支持Groovy语言的相关特性，包括：

- Groovy风格的Lambda表达式和内嵌函数调用，例如 `map.findAll{it.value>3}` 。
- 数学表达式，例如 `1+1` 。
- 系统调用，例如 `System.currentTimeMillis()` 。

事务的支持

图数据库GDB Gremlin默认支持事务，具体如下：

- 不支持ThreadedTransaction（即跨线程，由用户自己控制开始和提交的方式），而是采用内置的Sesionless方式的事务。一个DSL内部所有的操作认为在一个事务内部。

- 如果DSL内部含有更新类型的Step（Mutation类别），则认为该事务是一个读写事务，反之认为是一个只读事务。
- 一个DSL从开始的时候，自动开启一个事务，在结束的时候，按运行情况自动提交或回滚。
- 事务隔离的级别为Read-Committed。
- 数据导入拆分为按行的一组单条事务，事务（各数据行）之间无关联。

对TinkerPop Gremlin的Step接口支持情况

> **说明**
>
> - 主要不支持的接口主要为以下几类：
>   - 用到了GraphComputer类的OLAP接口。
>   - Explain和Profiling类型的接口。
>   - 其他非增、删、查、改类型的辅助接口。
> - 下表中✔代表支持，✘代表不支持。

Step接口	是否支持	备注
`connectedComponent()`	✘	不支持 `GraphComputer`
`io(String)`	✘	-
`pageRank()`	✘	不支持 `GraphComputer`
`pageRank(double)`	✘	不支持 `GraphComputer`
`peerPressure()`	✘	不支持 `GraphComputer`
`profile()`	✘	-
`profile(String)`	✘	-

Step接口	是否支持	备注
`program(VertexProgram<?>)`	❌	不支持 `GraphComputer`
`shortestPath()`	❌	不支持 `GraphComputer`
`subgraph(String)`	❌	-
`tx()`	❌	-
`withBulk(boolean)`	❌	-
`withComputer()`	❌	-
`withComputer(Class<? extends GraphComputer>)`	❌	-
`withComputer(Computer)`	❌	-
`withoutStrategies(Class<? extends TraversalStrategy>…)`	❌	-
`withPath()`	❌	-
`withRemote(Configuration)`	❌	-
`withRemote(RemoteConnection)`	❌	-
`withRemote(String)`	❌	-

Step接口	是否支持	备注
`write()`	✗	-
`addE(String)`	✓	-
`addE(Traversal<?, String>)`	✓	-
`addV()`	✓	-
`addV(String)`	✓	-
`addV(Traversal<?, String>)`	✓	-
`aggregate(String)`	✓	-
`and(Traversal<?, ?>…)`	✓	-
`as(String, String…)`	✓	-
`barrier()`	✓	-
`barrier(Consumer>)`	✓	-
`barrier(int)`	✓	-
`both(String…)`	✓	-

Step接口	是否支持	备注
bothE(String…)	✓	-
bothV()	✓	-
branch(Function,M>	✓	-
branch(Traversal<?, M>)	✓	-
by()	✓	-
by(Comparator)	✓	-
by(Function, Comparator)	✓	-
by(Function)	✓	-
by(Order)	✓	-
by(String)	✓	-
by(String, Comparator)	✓	-
by(T)	✓	-
by(Traversal<?, ?>)	✓	-

Step接口	是否支持	备注
`by(Traversal<?, ?>, Comparator)`	✓	-
`cap(String, String…)`	✓	-
`choose(Function)`	✓	-
`choose(Predicate, Traversal<?, E2>)`	✓	-
`choose(Predicate, Traversal<?, E2>, Traversal<?, E2>)`	✓	-
`choose(Traversal<?, ?>, Traversal<?, E2>)`	✓	-
`choose(Traversal<?, ?>, Traversal<?, E2>, Traversal<?, E2>)`	✓	-
`choose(Traversal<?, M>)`	✓	-
`coalesce(Traversal<?, E2>…)`	✓	-
`coin(double)`	✓	-
`constant(E2)`	✓	-
`count()`	✓	-

Step接口	是否支持	备注
`count(Scope)`	✓	-
`cyclicPath()`	✓	-
`dedup(Scope, String…)`	✓	-
`dedup(String…)`	✓	-
`drop()`	✓	-
`E(Object…)`	✓	-
`emit()`	✓	-
`emit(Predicate>)`	✓	-
`emit(Traversal<?, ?>)`	✓	-
`filter(Predicate>)`	✓	-
`filter(Traversal<?, ?>)`	✓	-
`flatMap(Function, Iterator>)`	✓	-
`flatMap(Traversal<?, E2>)`	✓	-

Step接口	是否支持	备注
`fold()`	✓	-
`fold(E2, BiFunction)`	✓	-
`from(String)`	✓	-
`from(Traversal<?, Vertex>)`	✓	-
`from(Vertex)`	✓	-
`group()`	✓	-
`group(String)`	✓	-
`groupCount()`	✓	-
`groupCount(String)`	✓	-
`has(String)`	✓	-
`has(String, Object)`	✓	-
`has(String, P<?>)`	✓	-
`has(String, String, Object)`	✓	-

Step接口	是否支持	备注
`has(String, String, P<?>)`	✓	-
`has(String, Traversal<?, ?>)`	✓	-
`has(T, Object)`	✓	-
`has(T, P<?>)`	✓	-
`has(T, Traversal<?, ?>)`	✓	-
`hasId(Object, Object…)`	✓	-
`hasId(P)`	✓	-
`hasKey(P)`	✓	-
`hasKey(String, String…)`	✓	-
`hasLabel(P)`	✓	-
`hasLabel(String, String…)`	✓	-
`hasNot(String)`	✓	-
`hasValue(Object, Object…)`	✓	-

Step接口	是否支持	备注
`hasValue(P)`	✓	-
`id()`	✓	-
`identity()`	✓	-
`in(String…)`	✓	-
`index()`	✓	-
`inE(String…)`	✓	-
`inject(E…)`	✓	-
`inV()`	✓	-
`is(Object)`	✓	-
`is(P)`	✓	-
`iterate()`	✓	-
`key()`	✓	-
`label()`	✓	-

Step接口	是否支持	备注
`limit(long)`	✓	-
`limit(Scope, long)`	✓	-
`local(Traversal<?, E2>)`	✓	-
`loops()`	✓	-
`loops(String)`	✓	-
`map(Function, E2>)`	✓	-
`map(Traversal<?, E2>)`	✓	-
`match(Traversal<?, ?>…)`	✓	-
`math(String)`	✓	-
`max()`	✓	-
`max(Scope)`	✓	-
`mean()`	✓	-
`mean(Scope)`	✓	-

Step接口	是否支持	备注
`min()`	✓	-
`min(Scope)`	✓	-
`not(Traversal<?, ?>)`	✓	-
`option(M, Traversal<?, E2>)`	✓	-
`option(Traversal<?, E2>)`	✓	-
`optional(Traversal<?, E2>)`	✓	-
`or(Traversal<?, ?>…)`	✓	-
`order()`	✓	-
`order(Scope)`	✓	-
`otherV()`	✓	-
`out(String…)`	✓	-
`outE(String…)`	✓	-
`outV()`	✓	-

Step接口	是否支持	备注
`path()`	✓	-
`project(String, String…)`	✓	-
`properties(String…)`	✓	-
`property(Object, Object, Object…)`	✓	-
`property(Cardinality, Object, Object, Object…)`	✓	仅支持 `Cardinality.single`
`propertyMap(String…)`	✓	-
`range(long, long)`	✓	-
`range(Scope, long, long)`	✓	-
`read()`	✓	-
`repeat(String, Traversal<?, E>)`	✓	-
`repeat(Traversal<?, E>)`	✓	-
`sack()`	✓	-
`sack(BiFunction)`	✓	-

Step接口	是否支持	备注
`sample(int)`	✓	-
`sample(Scope, int)`	✓	-
`select(Column)`	✓	-
`select(Pop, String)`	✓	-
`select(Pop, String, String, String…)`	✓	-
`select(Pop, Traversal)`	✓	-
`select(String)`	✓	-
`select(String, String, String…)`	✓	-
`select(Traversal)`	✓	-
`sideEffect(Consumer>)`	✓	-
`sideEffect(Traversal<?, ?>)`	✓	-
`simplePath()`	✓	-
`skip(long)`	✓	-

Step接口	是否支持	备注
`skip(Scope, long)`	✓	-
`store(String)`	✓	-
`sum()`	✓	-
`sum(Scope)`	✓	-
`tail()`	✓	-
`tail(long)`	✓	-
`tail(Scope)`	✓	-
`tail(Scope, long)`	✓	-
`timeLimit(long)`	✓	-
`times(int)`	✓	-
`to(Direction, String…)`	✓	-
`to(String)`	✓	-
`to(Traversal<?, Vertex>)`	✓	-

Step接口	是否支持	备注
to(Vertex)	✓	-
toE(Direction, String…)	✓	-
toV(Direction)	✓	-
tree()	✓	-
tree(String)	✓	-
unfold()	✓	-
union(Traversal<?, E2>…)	✓	-
until(Predicate>)	✓	-
until(Traversal<?, ?>)	✓	-
V(Object…)	✓	-
value()	✓	-
valueMap(boolean, String…)	✓	-
valueMap(String…)	✓	-

Step接口	是否支持	备注
values(String…)	✓	-
where(P)	✓	-
where(String, P)	✓	-
where(Traversal<?, ?>)	✓	-
with(String)	✓	-
with(String, Object)	✓	-
withSack(A)	✓	-
withSack(A, BinaryOperator)	✓	-
withSack(A, UnaryOperator)	✓	-
withSack(A, UnaryOperator<A>, BinaryOperator<A>)	✓	-
withSack(Supplier<A>)	✓	-
withSack(Supplier,BinaryOperator)	✓	-
withSack(Supplier,UnaryOperator)	✓	-

Step接口	是否支持	备注
`withSack(Supplier,UnaryOperator,BinaryOperator)`	✓	-
`withSideEffect(String, A)`	✓	-
`withSideEffect(String, A, BinaryOperator)`	✓	-
`withSideEffect(String,Supplier)`	✓	-
`withSideEffect(String,Supplier, BinaryOperator)`	✓	-

图数据库GDB Gremlin支持的特性

> **说明**
>
> 下表中特性的返回内容与TinkerPop的Graph.features()返回内容相同。

分类	特性	是否支持
Graph	ThreadedTransactions	✗
	Computer	✗
	Transactions	✓
	Persistence	✓
	ConcurrentAccess	✓
	SerializableValues	✗

分类	特性	是否支持
Variable	UniformListValues	✗
	BooleanArrayValues	✗
	DoubleArrayValues	✗
	IntegerArrayValues	✗
	StringArrayValues	✗
	MapValues	✗
	MixedListValues	✗
	ByteArrayValues	✗
	FloatArrayValues	✗
	LongArrayValues	✗
	Variables	✓
	BooleanValues	✓
	ByteValues	✓
	DoubleValues	✓
	FloatValues	✓
	IntegerValues	✓
	LongValues	✓

分类	特性	是否支持
	StringValues	✓
Vertex	MetaProperties	✗
	DuplicateMultiProperties	✗
	MultiProperties	✗
	NumericIds	✗
	UuidIds	✗
	CustomIds	✗
	AnyIds	✗
	AddVertices	✓
	RemoveVertices	✓
	UserSuppliedIds	✓
	AddProperty	✓
	RemoveProperty	✓
	StringIds	✓
	UserSuppliedIds	✗
	NumericIds	✗
	UuidIds	✗

分类	特性	是否支持
Vertex Property	CustomIds	✗
	AnyIds	✗
	SerializableValues	✗
	UniformListValues	✗
	BooleanArrayValues	✗
	DoubleArrayValues	✗
	IntegerArrayValues	✗
	StringArrayValues	✗
	MapValues	✗
	MixedListValues	✗
	ByteArrayValues	✗
	FloatArrayValues	✗
	LongArrayValues	✗
	AddProperty	✓
	RemoveProperty	✓
	StringIds	✓
	Properties	✓

分类	特性	是否支持
	BooleanValues	✓
	ByteValues	✓
	DoubleValues	✓
	FloatValues	✓
	IntegerValues	✓
	LongValues	✓
	StringValues	✓
Edge	NumericIds	✗
	UuidIds	✗
	CustomIds	✗
	AnyIds	✗
	AddEdges	✓
	RemoveEdges	✓
	UserSuppliedIds	✓
	AddProperty	✓
	RemoveProperty	✓

分类	特性	是否支持
	StringIds	✓
Edge Property	SerializableValues	✗
	UniformListValues	✗
	BooleanArrayValues	✗
	DoubleArrayValues	✗
	IntegerArrayValues	✗
	StringArrayValues	✗
	MapValues	✗
	MixedListValues	✗
	ByteArrayValues	✗
	FloatArrayValues	✗
	LongArrayValues	✗
	Properties	✓
	BooleanValues	✓
	ByteValues	✓
	DoubleValues	✓
	FloatValues	✓

分类	特性	是否支持
	IntegerValues	✓
	LongValues	✓
	StringValues	✓

1.2. HTTP REST

本文介绍如何使用HTTP REST方式连接和操作图数据库GDB。REST API兼容Gremlin3.3.x和3.4.0版本。

前提条件

请确保图数据库GDB的实例与您的ECS虚拟机处于同一个VPC网络环境。

操作步骤

以下示例中，需要将配置信息替换成您的图数据库实例的相关信息。

- 将${your-gdb-endpoint}改为您的图数据库GDB实例的域名。
- 将${username}改为您的图数据库GDB实例的用户名。
- 将${password}改为您的图数据库GDB实例的密码。

1. 图数据库GDB的HTTP接入点为 `${your_gdb_endpoint}:8182/gremlin` 。
2. 连接时可以使用GET请求，也可以使用POST请求。建议使用POST请求进行操作：

```
curl -u ${username}:${password} -X POST -d '{"gremlin":"g.V().limit(1)"}' http://${your_gdb_endpoint}:8182/gremlin
//此处注意需要对label值的双引号进行转义
curl -u ${username}:${password} -X POST -d '{"gremlin":"g.V().hasLabel(\"movie\").limit(10)"}' http://${your_gdb_endpoint}:8182/gremlin
curl -u ${username}:${password} "http://${your_gdb_endpoint}:8182/gremlin?gremlin=g.V().limit(1)"
```

3. 使用REST请求访问服务器的时候，可以以binding形式携带参数：

```
curl -u ${username}:${password} -X POST -d '{"gremlin":"g.V().limit(n)","bindings":{"n":1}}' http://${your_gdb_endpoint}:8182/gremlin
```

4. 返回结果示例如下：

```
{
    "requestId": "e640005a-f2fd-403e-812c-f61e7a3fb7cb",
    "result": {
        "data": {
            "@type": "g:List",
            "@value": [
                {
                    "@type": "g:Vertex",
                    "@value": {
                        "id": "3a63cc90-d957-4324-9ffc-16a8e4c1c1f4",
                        "label": "label",
                        "properties": {
                            "pint": [
                                {
                                    "@type": "g:VertexProperty",
                                    "@value": {
                                        "id": "3a63cc90-d957-4324-9ffc-16a8e4c1c1f4",
                                        "label": "label",
                                        "value": {
                                            "@type": "g:Int32",
                                            "@value": 3
                                        }
                                    }
                                }
                            ]
                        }
                    }
                }
            ]
        },
        "meta": {
            "@type": "g:Map",
            "@value": []
        }
    },
    "status": {
        "attributes": {
            "@type": "g:Map",
            "@value": []
        },
        "code": 200,
        "message": ""
    }
}
```


以上示例是使用 `g.V().limit(n)` 遍历返回GDB中的部分点。更多DSL范例，请参见文档通过Gremlin Console连接实例。

REST接口支持模板化调用方式，示例如下：

```
DSL硬编码
g.V().hasLabel('gdb_sample_person').drop()
            |
            |
            V
模板化调用
g.V().hasLabel(label).drop()
bindings:{"label":"gdb_sample_person"}
```

有关Gremlin REST接口的更多信息，请参阅Apache TinkerPop3文档中的Connecting via HTTP。

1.3. Java

本文介绍如何基于Java编程环境连接和操作图数据库GDB。这是以常驻服务形式操作图数据库GDB的常用形式。

前提条件

图数据库GDB实例需要与ECS虚拟机处于同一个VPC中。

安装Maven

1. 添加具有Maven程序包的存储库。

```
wget http://repos.fedorapeople.org/repos/dchen/apache-maven/epel-apache-maven.repo
-O /etc/yum.repos.d/epel-apache-maven.repo
```

2. 设置该存储库的版本号。

```
sudo sed -i s/\$releasever/6/g /etc/yum.repos.d/epel-apache-maven.repo
```

3. 下载并安装Maven。

```
sudo yum install -y apache-maven
```

安装Java

1. 安装JDK 8.0。

```
sudo yum install java-1.8.0-devel
```

2. 如果您的ECS实例上有多个Java版本，请将Java8设置为默认运行。

```
sudo /usr/sbin/alternatives --config java
共有 4 个提供"java"的程序。
选项    命令
-----------------------------------------------
*+ 1           java-1.8.0-openjdk.x86_64 (/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.191.b12
-1.el7_6.x86_64/jre/bin/java)
 2           java-1.8.0-openjdk.x86_64 (/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.191.b12-0
.el7_5.x86_64-debug/jre/bin/java)
 3           java-1.7.0-openjdk.x86_64 (/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.191-2.6.1
5.4.el7_5.x86_64/jre/bin/java)
 4           /usr/lib/jvm/jre-1.6.0-openjdk.x86_64/bin/java
```

编写Java客户端代码

1. 创建gdb-gremlin-test的目录。

```
mkdir gdb-gremlin-test;
cd gdb-gremlin-test
```

2. 创建pom.xml文件，并写入如下内容。

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XM
LSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v
4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.gdb.alibaba</groupId>
  <artifactId>GdbGremlinExample</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>GdbGremlinExample</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
       <groupId>org.apache.tinkerpop</groupId>
       <artifactId>gremlin-driver</artifactId>
       <version>3.4.0</version>
    </dependency>
  </dependencies>
  <build>
     <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>2.0.2</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>exec-maven-plugin</artifactId>
            <version>1.3</version>
            <configuration>
                <mainClass>com.gdb.alibaba.Test</mainClass>
                <complianceLevel>1.8</complianceLevel>
            </configuration>
        </plugin>
    </plugins>
  </build>
</project>
```

3. 创建目录并新建文件。

```
mkdir -p src/main/java/com/gdb/alibaba/;
touch src/main/java/com/gdb/alibaba/Test.java
```

4. 编写测试程序。

```
package com.gdb.alibaba;
import org.apache.tinkerpop.gremlin.driver.Cluster;
import org.apache.tinkerpop.gremlin.driver.Client;
import org.apache.tinkerpop.gremlin.driver.Result;
import org.apache.tinkerpop.gremlin.driver.ResultSet;
import java.util.List;
import java.util.Map;
import java.util.HashMap;
import java.io.File;
public class Test
{
    public static void main( String[] args )
    {
      try {
        if(args.length != 1) {
            System.out.println("gdb-remote.yaml path needed");
            return;
        }
        String yaml = args[0];
        // 1. 初始化客户端，客户端包含连接池，线程安全，支持多线程并发
        Cluster cluster = Cluster.build(new File(yaml)).create();
        Client client = cluster.connect().init();
        // 2. 发送gremlin请求到GDB服务端，根据业务逻辑定制
        String dsl = "g.addV(yourlabel).property(propertyKey, propertyValue)";
        Map<String,Object> parameters = new HashMap<>();
        parameters.put("yourlabel","area");
        parameters.put("propertyKey","wherence");
        parameters.put("propertyValue","shenzheng");
        ResultSet results = client.submit(dsl,parameters);
        List<Result> result = results.all().join();
        result.forEach(p -> System.out.println(p.getObject()));
        // 3. 关闭客户端，完成所有gremlin请求清理资源
        cluster.close();
      } catch (Exception e) {
        System.out.println(e.getMessage());
      }
  }
}
```

> **说明** SDK中client包含有连接池，支持多线程并发。业务上可以全局维护一个client，不需要每次请求都新建，等所有图数据库GDB请求处理完或者应用退出时再关闭。

5. 创建gdb-remote.yaml文件，该文件为Java客户端与GDB图数据库建立连接的配置文件。

```
hosts: [ ${gdbHost} ]
port: 8182
username: ${username}
password: ${password}
serializer: {
  className: org.apache.tinkerpop.gremlin.driver.ser.GraphBinaryMessageSerializerV1,
  config: { serializeResultToString: false }
}
```

其中，示例参数说明如下：

参数	说明
${gdbHost}	GDB的连接地址，例如 gds-bp******************.graphdb.rds.aliyuncs.com 。
${username}	GDB的数据库账号。
${password}	GDB数据库账号对应的密码。

6. 进入gdb-gremlin-test主目录，编译并执行Java程序。

```
mvn compile exec:java  -Dexec.args="/root/apache-tinkerpop-gmlin-console-3.4.0/conf/gdb-remote.yaml"
```

执行结果如下：

```
v[ba8f60b7-0786-4014-a4e2-451f09b79878]
```

显式配置客户端

如果您需要集中管理配置，而不是使用单独的GDB配置文件，可以通过Driver提供的API对客户端进行显式配置，示例代码如下：

```
// 1. 初始化客户端，客户端包含连接池，线程安全，支持多线程并发
// Cluster cluster = Cluster.build(new File(yaml)).create();
// Client client = cluster.connect().init();
// 显式配置客户端
Cluster.build(${gdbHost}).port(${gdbPort}).
    serializer(Serializers.GRAPHBINARY_V1D0).
    maxConnectionPoolSize(8).
    minConnectionPoolSize(8).
    maxContentLength(65536).
    credentials(${username}, ${password}).create();
client = cluster.connect().init();
```

其中，示例参数说明如下：

参数	说明
${gdbHost}	GDB的连接地址，例如 gds-bp******************.graphdb.rds.aliyuncs.com 。

参数	说明
`${gdbPort}`	GDB的端口号，例如 `8182` 。
`${username}`	GDB的数据库账号。
`${password}`	GDB数据库账号对应的密码。

更多客户端常用配置参数如下：

参数	默认值	说明
connectionPool.maxContentLength	65536	消息最大字节数。请求返回的数据量较大时需要增大该参数的值，否则可能出现返回结果无法处理的错误。
connectionPool.maxSize	8	连接池最大连接数。并发较大的情况下建议增加最大连接数。
connectionPool.minSize	2	连接池最小连接数。

更多dsl样例

本文中以上示例内容是适用参数化的方式，通过 `dsl g.addV(yourlabel).property(propertyKey, propertyValue)` 和参数 `map` 添加的点。以下内容将结合具体的图的点、边结构来进行更多的dsl示例。关于图的点、边结构的介绍，请参见ThinkerPop文档。

> **说明** 以下dsl示例需要改造成参数化的调用方式，您可以使用硬编码方式来进行范例讲解：

```
//DSL硬编码：
dsl = "user_defined_dsl";
//比如：g.addV('sand131_id_5_99').property(id,'sand131_id_5_99').property('name','sand13
1_name_5_99')
ResultSet results = client.submit(dsl);
                |
                |
                v
//参数化脚本：
String dsl ="g.addV(vertex).property(id,vertex).property('name',vertex)";
Map<String, Object> parameters = new HashMap<>();
parameters.put("vertex","sand131_id_5_99"); //填写dsl语句中的vertex参数
ResultSet results = client.submit(dsl, parameters,timeoutInMillis);
```

示例步骤如下：

1. 删除指定label的点、边。

```
g.E().hasLabel('gdb_sample_knows').drop()
g.E().hasLabel('gdb_sample_created').drop()
g.V().hasLabel('gdb_sample_person').drop()
g.V().hasLabel('gdb_sample_software').drop()
```

2. 添加顶点,为其设置id、property。

```
g.addV('gdb_sample_person').property(id, 'gdb_sample_marko').property('age', 28).proper
ty('name', 'marko')
g.addV('gdb_sample_person').property(id, 'gdb_sample_vadas').property('age', 27).proper
ty('name', 'vadas')
g.addV('gdb_sample_person').property(id, 'gdb_sample_josh').property('age', 32).propert
y('name', 'josh')
g.addV('gdb_sample_person').property(id, 'gdb_sample_peter').property('age', 35).proper
ty('name', 'peter')
g.addV('gdb_sample_software').property(id, 'gdb_sample_lop').property('lang', 'java').p
roperty('name', 'lop')
g.addV('gdb_sample_software').property(id, 'gdb_sample_ripple').property('lang', 'java'
).property('name', 'ripple')
```

3. 修改或新增age属性。

```
g.V('gdb_sample_marko').property('age', 29)
```

4. 建立关系，设置属性weight。

```
g.addE('gdb_sample_knows').from(V('gdb_sample_marko')).to(V('gdb_sample_vadas')).proper
ty('weight', 0.5f)
g.addE('gdb_sample_knows').from(V('gdb_sample_marko')).to(V('gdb_sample_josh')).propert
y('weight', 1.0f)
g.addE('gdb_sample_created').from(V('gdb_sample_marko')).to(V('gdb_sample_lop')).proper
ty('weight', 0.4f)
g.addE('gdb_sample_created').from(V('gdb_sample_josh')).to(V('gdb_sample_lop')).propert
y('weight', 0.4f)
g.addE('gdb_sample_created').from(V('gdb_sample_josh')).to(V('gdb_sample_ripple')).prop
erty('weight', 1.0f)
g.addE('gdb_sample_created').from(V('gdb_sample_peter')).to(V('gdb_sample_lop')).proper
ty('weight', 0.2f)
```

5. 查询所有点或指定label的点数量。

```
g.V().count()
g.V().hasLabel('gdb_sample_person').count()
```

6. 查询指定条件的顶点（>29岁的人，按name降序排列所有人）。

```
g.V().hasLabel('gdb_sample_person').has('age', gt(29))
g.V().hasLabel('gdb_sample_person').order().by('name', decr)
```

7. 关联查询（获取marko 认识的人， marko认识的人created的software）。

```
g.V('gdb_sample_marko').outE('gdb_sample_knows').inV().hasLabel('gdb_sample_person')
g.V('gdb_sample_marko').outE('gdb_sample_knows').inV().hasLabel('gdb_sample_person').ou
tE('gdb_sample_created').inV().hasLabel('gdb_sample_software')
```

8. 删除关系、顶点。

```
g.V('gdb_sample_marko').outE('gdb_sample_knows').where(inV().has(id, 'gdb_sample_josh')
).drop()
g.V('gdb_sample_marko').drop()
```

您也可以进行其他测试，详细的Gremlin查询语句，请参见TinkerPop的Gremlin文档。

1.4. Python

本文介绍如何基于Python编程环境连接和操作图数据库GDB。

前提条件

- 进行以下操作时，请确保图数据库GDB的实例与您的ECS虚拟机处于同一个VPC网络环境。
- 在使用Python连接GDB实例之前，需要准备以下环境：
  - Python 2.7+或者3.5+。
  - pip。
- 如果您的Python为2.7版本，则需要额外安装以下module：

```
pip install futures --user
```

使用Python连接到GDB实例

1. 输入以下命令以安装gremlinpython程序包：

```
pip install gremlinpython --user
```

2. 创建 `test.py` 文件，编辑并输入以下内容，并替换其中的配置信息：
   - 将 `${your-gdb-endpoint}` 改为您的图数据库GDB实例的域名
   - 将 `${username}` 改为您的图数据库GDB实例的用户名
   - 将 `${password}` 改为您的图数据库GDB实例的密码

```
from __future__ import print_function  # Python 2/3 compatibility
from gremlin_python.driver import client
client = client.Client('ws://${your-gdb-endpoint}:8182/gremlin', 'g', username="${usern
ame}", password="${password}")
callback = client.submit("g.V().limit(1)").all().result()
for result in callback:
    print(result)
client.close()
```

3. 执行以下命令运行示例。

```
python test.py
```

   回显如下：

```
[v[3a63cc90-d957-4324-9ffc-16a8e4c1c1f4]]
```

   以上回显实例是使用 `g.V().limit(1)` 遍历返回GDB中的一个点，要查询其他内容，需替换成对应的DSL。

4. （可选）常见异常处理。
   - Cannot run the event loop while another loop is running

     在jupyter等环境使用gremlin_python可能会报该错误，嵌套event loop问题，可以添加下面程序包解决

```
# 安装程序包
!pip install nest_asyncio
# 代码文件中导入并使用
import nest_asyncio
nest_asyncio.apply()
```

- Windows下gremlin driver报NotImplementError

  Python 3.8以后asyncio库修改了Windows系统下默认的event loop实现方式，gremlin driver用来处理网络通信的tornado依赖于asyncio，因此会抛出NotImplementError异常。解决方法是添加如下代码修改event loop策略

```
import sys
import asyncio
if sys.platform == 'win32':
    asyncio.set_event_loop_policy(asyncio.WindowsSelectorEventLoopPolicy())
```

其他DSL范例如下

1. 删除指定label的点和边。

```
g.E().hasLabel('gdb_sample_knows').drop()
g.E().hasLabel('gdb_sample_created').drop()
g.V().hasLabel('gdb_sample_person').drop()
g.V().hasLabel('gdb_sample_software').drop()
```

2. 添加顶点为其设置id、property。

```
g.addV('gdb_sample_person').property(id, 'gdb_sample_marko').property('age', 28).proper
ty('name', 'marko')
g.addV('gdb_sample_person').property(id, 'gdb_sample_vadas').property('age', 27).proper
ty('name', 'vadas')
g.addV('gdb_sample_person').property(id, 'gdb_sample_josh').property('age', 32).propert
y('name', 'josh')
g.addV('gdb_sample_person').property(id, 'gdb_sample_peter').property('age', 35).proper
ty('name', 'peter')
g.addV('gdb_sample_software').property(id, 'gdb_sample_lop').property('lang', 'java').p
roperty('name', 'lop')
g.addV('gdb_sample_software').property(id, 'gdb_sample_ripple').property('lang', 'java'
).property('name', 'ripple')
```

3. 修改或新增age属性。

```
g.V('gdb_sample_marko').property('age', 29)
```

4. 建立关系，设置属性 weight。

```
g.addE('gdb_sample_knows').from(V('gdb_sample_marko')).to(V('gdb_sample_vadas')).proper
ty('weight', 0.5f)
g.addE('gdb_sample_knows').from(V('gdb_sample_marko')).to(V('gdb_sample_josh')).propert
y('weight', 1.0f)
g.addE('gdb_sample_created').from(V('gdb_sample_marko')).to(V('gdb_sample_lop')).proper
ty('weight', 0.4f)
g.addE('gdb_sample_created').from(V('gdb_sample_josh')).to(V('gdb_sample_lop')).propert
y('weight', 0.4f)
g.addE('gdb_sample_created').from(V('gdb_sample_josh')).to(V('gdb_sample_ripple')).prop
erty('weight', 1.0f)
g.addE('gdb_sample_created').from(V('gdb_sample_peter')).to(V('gdb_sample_lop')).proper
ty('weight', 0.2f)
```

5. 查询所有点或指定label的点数量。

```
g.V().count()
g.V().hasLabel('gdb_sample_person').count()
```

6. 查询指定条件的顶点（>29岁的人，按name降序排列所有人）。

```
g.V().hasLabel('gdb_sample_person').has('age', gt(29))
g.V().hasLabel('gdb_sample_person').order().by('name', decr)
```

7. 关联查询（获取marko认识的人，marko认识的人created的software）。

```
g.V('gdb_sample_marko').outE('gdb_sample_knows').inV().hasLabel('gdb_sample_person')
g.V('gdb_sample_marko').outE('gdb_sample_knows').inV().hasLabel('gdb_sample_person').ou
tE('gdb_sample_created').inV().hasLabel('gdb_sample_software')
```

8. 删除关系、顶点。

```
g.V('gdb_sample_marko').outE('gdb_sample_knows').where(inV().has(id, 'gdb_sample_josh')
).drop()
g.V('gdb_sample_marko').drop()
```

相关资源

有关Gremlin Python接口的更多信息，请参阅Apache TinkerPop3文档中的Gremlin-Python。

1.5. .Net

本文介绍如何基于 `.NET` 编程环境连接和操作图数据库GDB。

前提条件

请确保您的图数据库GDB实例与您的ECS虚拟机处于同一个VPC网络环境。

准备工作

1. 安装版本在2.0以上的dotnet环境。如果使用windows开发环境，请使用Windows8以上的OS版本。
2. 执行.Net命令生成console工程gremlinTest。

```
dotnet new console -o gremlinTest
```

3. 进入该工程目录。

```
cd gremlinTest
```

4. 为您的程序工程来安装Gremlin的SDK依赖 `Gremlin.NET` 。

```
dotnet add package gremlin.net
```

5. 编辑Program.cs文件。

   将内容替换如下，需要将配置信息替换成您的图数据库实例的相关信息：

   - 将 `${your-gdb-endpoint}` 改为您的图数据库GDB实例的域名
   - 将 `${username}` 改为您的图数据库GDB实例的用户名
   - 将 `${password}` 改为您的图数据库GDB实例的密码

```
using System;
using System.Threading.Tasks;
using System.Collections.Generic;
using Gremlin.Net;
using Gremlin.Net.Driver;
namespace gremlinTest
{
    class Program
    {
        private static string endpoint = "${your_gdb_endpoint}";
        private static int port = 8182;
        private static string username = "${username}";
        private static string password = "${password}";
        static void Main(string[] args)
        {
            try
            {
                var gremlinServer = new GremlinServer(endpoint, port, username: usernam
e, password: password);
                var gremlinClient = new GremlinClient(gremlinServer);
                Program program = new Program();
                program.RunQueryAsync(gremlinClient).Wait();
            }
            catch (Exception e)
            {
                Console.WriteLine("{0}", e);
            }
        }
        private async Task RunQueryAsync(GremlinClient gremlinClient)
        {
                var vertices = await gremlinClient.SubmitWithSingleResultAsync<dynamic>
("g.V().limit(1)");
                Console.WriteLine("{0}", vertices);
        }
    }
}
```

连接并执行

- 运行命令执行程序，输出结果的范例如下：

```
v[3a63cc90-d957-4324-9ffc-16a8e4c1c1f4]
```

- 上面的例子是使用 `g.V().limit(1)` 遍历返回GDB中的一个点。要查询其他内容，可以替换成其他的DSL。
- 其他DSL的范例如下，图的点、边结构链接 http://tinkerpop.apache.org/docs/current/reference/#traversal。

1. 删除指定label的点、边。

```
g.E().hasLabel('gdb_sample_knows').drop()
g.E().hasLabel('gdb_sample_created').drop()
g.V().hasLabel('gdb_sample_person').drop()
g.V().hasLabel('gdb_sample_software').drop()
```

2. 添加顶点，为其设置id、property。

```
g.addV('gdb_sample_person').property(id, 'gdb_sample_marko').property('age', 28).proper
ty('name', 'marko')
g.addV('gdb_sample_person').property(id, 'gdb_sample_vadas').property('age', 27).proper
ty('name', 'vadas')
g.addV('gdb_sample_person').property(id, 'gdb_sample_josh').property('age', 32).propert
y('name', 'josh')
g.addV('gdb_sample_person').property(id, 'gdb_sample_peter').property('age', 35).proper
ty('name', 'peter')
g.addV('gdb_sample_software').property(id, 'gdb_sample_lop').property('lang', 'java').p
roperty('name', 'lop')
g.addV('gdb_sample_software').property(id, 'gdb_sample_ripple').property('lang', 'java'
).property('name', 'ripple')
```

3. 修改或新增age属性。

```
g.V('gdb_sample_marko').property('age', 29)
```

4. 建立关系，设置属性 weight。

```
g.addE('gdb_sample_knows').from(V('gdb_sample_marko')).to(V('gdb_sample_vadas')).proper
ty('weight', 0.5f)
g.addE('gdb_sample_knows').from(V('gdb_sample_marko')).to(V('gdb_sample_josh')).propert
y('weight', 1.0f)
g.addE('gdb_sample_created').from(V('gdb_sample_marko')).to(V('gdb_sample_lop')).proper
ty('weight', 0.4f)
g.addE('gdb_sample_created').from(V('gdb_sample_josh')).to(V('gdb_sample_lop')).propert
y('weight', 0.4f)
g.addE('gdb_sample_created').from(V('gdb_sample_josh')).to(V('gdb_sample_ripple')).prop
erty('weight', 1.0f)
g.addE('gdb_sample_created').from(V('gdb_sample_peter')).to(V('gdb_sample_lop')).proper
ty('weight', 0.2f)
```

5. 查询所有点/指定label的点数量。

```
g.V().count()
g.V().hasLabel('gdb_sample_person').count()
```

6. 查询指定条件的顶点 (>29岁的人，按name降序排列所有人）。

```
g.V().hasLabel('gdb_sample_person').has('age', gt(29))
g.V().hasLabel('gdb_sample_person').order().by('name', decr)
```

7. 关联查询（获取marko认识的人，marko认识的人created的software）。

```
g.V('gdb_sample_marko').outE('gdb_sample_knows').inV().hasLabel('gdb_sample_person')
g.V('gdb_sample_marko').outE('gdb_sample_knows').inV().hasLabel('gdb_sample_person').ou
tE('gdb_sample_created').inV().hasLabel('gdb_sample_software')
```

8. 删除关系、顶点。

```
g.V('gdb_sample_marko').outE('gdb_sample_knows').where(inV().has(id, 'gdb_sample_josh')
).drop()
g.V('gdb_sample_marko').drop()
```

有关Gremlin.Net使用方式的更多信息，请参阅Apache TinkerPop3文档中的Gremlin.Net。

1.6. Go

本文介绍如何基于Go语言编程环境连接和操作图数据库GDB，这也是以常驻服务形式操作图数据库GDB的常用形式。

> 注意
> - 以下示例的正确运行依赖您的运行环境网络与图数据库GDB实例联通
> - 运行环境需要安装有Go语言环境，推荐使用1.12版本，这也是本文示例使用的版本。

环境准备

1. 安装go语言环境，Linux环境可以直接安装二进制包。
2. 安装GDB SDK。

```
go get -u github.com/aliyun/alibabacloud-gdb-go-sdk/gdbclient
```

3. 获取图数据库GDB连接参数。

   在**实例管理>基本信息**页面，可以查看实例内网地址和内网端口，如果开通外网访问实例，也可以使用外网地址和外网端口。

```
# 内网环境
host = "${gdbID}.graphdb.rds.aliyuncs.com"
port = 8182
# 外网环境
host = "${gdbID}pub.graphdb.rds.aliyuncs.com"
port = 3734
```

> 注意 确保运行环境在实例白名单配置的可访问范围内。

   在**实例管理>账号管理**页面，可以查询实例账号信息，如果忘记密码，请点击**重置密码**按钮。

示例程序

上述GDB SDK中包含有演示程序，以下代码来自其中的演示程序。

1. 新建一个demo目录，并添加测试文件。

```
mkdir gdb_go_demo
cd gdb_go_demo
touch main.go
```

2. 编写测试代码，运行以下代码时需要图数据库GDB的连接参数，直接以参数提供。

```
package main
import (
    "flag"
    "log"
    goClient "github.com/aliyun/alibabacloud-gdb-go-sdk/gdbclient"
)
var (
    host, username, password string
    port                     int
)
func main() {
    flag.StringVar(&host, "host", "", "GDB Connection Host")
    flag.StringVar(&username, "username", "", "GDB username")
    flag.StringVar(&password, "password", "", "GDB password")
    flag.IntVar(&port, "port", 8182, "GDB Connection Port")
    flag.Parse()
    if host == "" || username == "" || password == "" {
        log.Fatal("No enough args provided. Please run:" +
            " go run main.go -host <gdb host> -username <username> -password <password>
-port <gdb port>")
        return
    }
    settings := &goClient.Settings{
        Host:     host,
        Port:     port,
        Username: username,
        Password: password,
    }
    // connect GDB with auth
    client := goClient.NewClient(settings)
    // send script dsl with bindings to GDB
    bindings := make(map[string]interface{})
    bindings["GDB___id"] = "gdb_vertex_test_id"
    bindings["GDB___label"] = "gdb_vertex_test_label"
    bindings["GDB___PK"] = "name"
    bindings["GDB___PV"] = "Jack"
    dsl := "g.addV(GDB___label).property(id, GDB___id).property(GDB___PK, GDB___PV)"
    results, err := client.SubmitScriptBound(dsl, bindings)
    if err != nil {
        log.Fatalf("Error while querying: %s\n", err.Error())
    }
    // get response, add vertex should return a Vertex
    for _, result := range results {
        v := result.GetVertex()
        log.Printf("get vertex[id:%s, label:%s, propLen %d", v.Id(), v.Label(), len(v.P
roperties()))
        // read vertex property
```

```
        for _, p := range v.VProperties() {
            log.Printf(" {PK: %s, PV: %s}", p.PKey(), p.PValue().(string))
        }
    }
    // drop all vertex
    _, err = client.SubmitScript("g.V().drop()")
    if err != nil {
        log.Fatalf("Error while querying: %s\n", err.Error())
    }
    client.Close()
}
```

上述代码实现创建连接GDB的client，并发送gremlin操作添加一个点，打印执行结果，最后发送清除所有数据的gremlin操作。针对client更多的参数配置请参考SDK包中的演示代码，包括连接池，session的支持等。

运行测试程序

将上一节中的GDB连接参数替换下面命令行参数，运行测试代码，正常会有下面输出。

```
go run main.go -host ${host} -port ${port} -username ${username} -password ${username}
2019/12/23 14:22:49 get vertex[id:gdb_vertex_test_id, label:gdb_vertex_test_label, propLen
1
2019/12/23 14:22:49  {PK: name, PV: Jack}
```

关于GDB GO SDK

- GDB的GO语言SDK是阿里GDB团队开发维护的Gremlin客户端，欢迎使用和提交MR。
- GO SDK目前不支持bytecode方式与GDB交互，用户可以使用script方式访问GDB，并将可变参数模版化，这也是GDB推荐的使用方式。
- GO SDK封装有session接口，支持在一次事务中包含多次对GDB更新的操作，保证所有操作都成功或失败，不会出现部分更新的情况。

1.7. NodeJs

本文介绍如何基于 `Node.js` 编程环境连接和操作图数据库GDB。

> **注意**
> - 进行以下操作时请确保图数据库GDB实例与您运行环境网络联通。
> - 确保运行环境已经安装有Node.js，且版本不低于8.11。

环境准备

1. 安装Node.js，版本8.11或更高版本，Linux环境可以直接安装二进制包。
2. 安装 `gremlin-javascript` 程序包。

```
npm install gremlin
```

3. 获取图数据库GDB连接参数。

在**实例管理>基本信息**页面，可以查看实例内网地址和内网端口，如果开通外网访问实例，也可以使用外网地址和外网端口。

```
# 内网环境
endpoint = "${gdbID}.graphdb.rds.aliyuncs.com:8182"
# 外网环境
endpoint = "${gdbID}pub.graphdb.rds.aliyuncs.com:3734"
```

> **注意** 确保运行环境在实例白名单配置的可访问范围内。

在**实例管理>账号管理**页面，可以查询实例账号信息，如果忘记密码，请点击**重置密码**。

```
cusername="${username}"
password="${password}"
```

示例代码

1. 新建一个demo目录，并添加测试文件。

```
mkdir gdb_demo
cd gdb_demo
touch demo.js
```

2. 编写测试代码，将代码中指定的 `your-name` 、 `your-password` 、 `your-endpoint` 替换为您实例的账号、密码、和域名。

```
"use strict";
const gremlin = require('gremlin');
const authenticator = new gremlin.driver.auth.PlainTextSaslAuthenticator('your-username
', 'your-password');
const client = new gremlin.driver.Client(
    'ws://your-endpoint/gremlin',
    {authenticator}
);
function countVertices()
{
    console.log('Running Vertex Count');
    return client.submit("g.V().count()", { })
        .then((result) => {
            console.log("Vertex Count Result: %s\n", JSON.stringify(result));
        });
}
function addVertex1()
{
    console.log('Running Add Vertex 1');
    return client.submit("g.addV(GDB___label).property(id, GDB___id).property(GDB___pk,
GDB___pv)",{
        GDB___id: "gdb_vertex_test_id_1",
        GDB___label: "gdb_vertex_test_label",
        GDB___pk: "name",
        GDB___pv: "Jack"
    }).then(data => {
        console.log("Add Vertex 1 Result: %s\n", JSON.stringify(data));
```

```
    });
}
function addVertex2()
{
    console.log('Running Add Vertex 2');
    return client.submit("g.addV(GDB___label).property(id, GDB___id).property(GDB___pk,
GDB___pv)",{
        GDB___id: "gdb_vertex_test_id_2",
        GDB___label: "gdb_vertex_test_label",
        GDB___pk: "name",
        GDB___pv: "Lucy"
    }).then(data => {
        console.log("Add Vertex 2 Result: %s\n", JSON.stringify(data));
    });
}
function addEdge()
{
    console.log('Running Add Edge');
    return client.submit("g.addE(GDB___label).from(V(GDB___from)).to(V(GDB___to)).prope
rty(id, GDB___id).property(GDB___pk, GDB___pv)",{
        GDB___id: "gdb_edge_test_id",
        GDB___label: "gdb_edge_test_label",
        GDB___from: "gdb_vertex_test_id_1",
        GDB___to: "gdb_vertex_test_id_2",
        GDB___pk: "relation",
        GDB___pv: "lover"
    }).then(data => {
        console.log("Add Edge Result: %s\n", JSON.stringify(data));
    });
}
function dropVertices()
{
    console.log('Running Drop');
    return client.submit('g.V().drop()', { })
        .then((result) => {
            console.log("Drop Result: %s\n", JSON.stringify(result));
        });
}
client.open()
    .then(countVertices)
    .then(addVertex1)
    .then(addVertex2)
    .then(addEdge)
    .then(dropVertices)
    .catch((error) => {
        console.error("Error running query...");
        console.error(error);
    }).then((res) => {
        client.close()
        console.log("Finished, Press any key to exit")
        process.stdin.resume();
        process.stdin.on('data', process.exit.bind(process, 0));
    }).catch((err) => {
        console.error("Fatal error:", err);
```

```
    });
```

上述代码使用图数据库GDB连接参数新建访问client，然后查询点数量，再添加两个点和一个边，接着删除所有点，最后等待用户键入退出。

> 注意 使用 `script` 方式交互时，最好将DSL中可变参数模版化，对性能提升有显著帮助。

3. 复制上述代码到demo.js，运行测试程序。

```
# 安装gremlin依赖到测试工程目录
npm install gremlin
# 运行测试程序
node demo.js
```

4. 连接新创建的图数据库GDB实例，以下是测试程序的输出。

```
Running Vertex Count
Vertex Count Result: {"_items":[0],"attributes":{},"length":1}
Running Add Vertex 1
Add Vertex 1 Result: {"_items":[{"id":"gdb_vertex_test_id_1","label":"gdb_vertex_test_l
abel","properties":{"name":[{"id":"gdb_vertex_test_id_1","label":"name","value":"Jack",
"key":"name"}]}}],"attributes":{},"length":1}
Running Add Vertex 2
Add Vertex 2 Result: {"_items":[{"id":"gdb_vertex_test_id_2","label":"gdb_vertex_test_l
abel","properties":{"name":[{"id":"gdb_vertex_test_id_2","label":"name","value":"Lucy",
"key":"name"}]}}],"attributes":{},"length":1}
Running Add Edge
Add Edge Result: {"_items":[{"id":"gdb_edge_test_id","label":"gdb_edge_test_label","out
V":"gdb_vertex_test_id_1","inV":"gdb_vertex_test_id_2","properties":{"relation":"lover"
}}],"attributes":{},"length":1}
Running Drop
Drop Result: {"_items":[],"attributes":{},"length":0}
Finished, Press any key to exit
```

使用bytecode方式访问图数据库GDB

除上述 `script` 方式与图数据库GDB交互外，您也可以使用 `bytecode` 方式与GDB交互，以下直接给出与上一节相同功能的代码，您可以直接用相同方法运行查看执行结果。

同样您也需要替换图数据库GDB实例对连接参数到下面代码中。

```
"use strict";
const gremlin = require('gremlin');
const DriverRemoteConnection = gremlin.driver.DriverRemoteConnection;
const Graph = gremlin.structure.Graph;
const __ = gremlin.process.statics;
const t = gremlin.process.t;
const authenticator = new gremlin.driver.auth.PlainTextSaslAuthenticator('your-username', '
your-password');
const graph = new Graph();
const g = graph.traversal().withRemote(
    new DriverRemoteConnection(
        'ws://your-endpoint/gremlin',
        {authenticator}
    ));
```

```
function vertexCount()
{
    console.log('Running Vertex Count');
    return g.V().count().next().
        then(data => {
            console.log("Vertex Count Result: %s", JSON.stringify(data));
        });
}
function addVertex1()
{
    console.log('Running Add Vertex 1');
    return g.addV('gdb_vertex_test_label')
        .property(t.id, 'gdb_vertex_test_id_1')
        .property('name','Jack').toList()
        .then(data => {
            console.log("Add Vertex 1 Result: %s", JSON.stringify(data));
        });
}
function addVertex2()
{
    console.log('Running Add Vertex 2');
    return g.addV('gdb_vertex_test_label')
        .property(t.id, 'gdb_vertex_test_id_2')
        .property('name','Lucy').toList()
        .then(data => {
            console.log("Add Vertex 2 Result: %s", JSON.stringify(data));
        });
}
function addEdge()
{
    console.log('Running Add Edge');
    return g.addE('gdb_edge_test_label')
        .from_(__.V('gdb_vertex_test_id_1'))
        .to(__.V('gdb_vertex_test_id_2'))
        .property(t.id, 'gdb_edge_test_id')
        .property('relation','lover').toList()
        .then(data => {
            console.log("Add Edge Result: %s", JSON.stringify(data));
        });
}
function dropVertex()
{
    console.log('Running Drop');
    return g.V().drop().iterate();
}
function finish()
{
    console.log("Finished, Press any key to exit")
    process.stdin.resume();
    process.stdin.on('data', process.exit.bind(process, 0));
}
async function allTasks()
{
  await vertexCount();
```

```
  await addVertex1();
  await addVertex2();
  await addEdge();
  await dropVertex();
  await finish();
}
allTasks();
```

以上示例使用顺序串行的方式依次完成点数量查询，添加两个点和一个边，再删除所有数据，最后等待用户键入退出。示例只是演示 `script` 和 `bytecode` 完成相同功能的代码片段，同时也推荐您在异步事件中嵌入GDB的操作代码。

> **注意** 使用 `bytecode` 方式需要在DSL末尾添加一个终结符

最佳实践

- GDB支持 `script` 和 `bytecode` 两种方式访问，但更推荐使用 `script` 方式。
- 使用 `script` 方式访问GDB时，最好将DSL中可变参数模版化，可以获得较好的性能。

1.8. 配置Gremlin请求超时时间

图数据库GDB支持修改客户端向图数据库GDB发送请求数据时的超时时间，您可以根据业务场景和数据量等修改超时时间。本文介绍不同语言环境中配置Gremlin请求超时时间的方法。

超时时间简介

- 客户端向图数据库GDB发送请求数据时，如果请求时间超过图数据库GDB设置的超时时间（默认为1800毫秒），则会发生超时现象，造成结果无法返回的现象。
- 超时时间的值为Long类型，单位为毫秒。
- 建议您不要配置过大的超时时间。

  > **说明**
  >
  > 复杂任务中需要设置过大的超时时间时，您可以使用图数据库GDB的图分析功能代替。

- 当客户端和图数据库GDB的连接在请求超时前断开时，会导致新的请求不生效。

使用各种语言配置超时时间

以下介绍各语言接入使用scripts请求的超时时间配置，示例配置超时时间为5000毫秒。

- Java
  i. 在客户端安装Java和Maven工具。
     a. 执行如下命令，安装Java（以1.8.0版本为例）。

```
sudo yum install java-1.8.0-devel
```

b. 执行如下命令，添加具有Maven程序包的存储库。

```
wget http://repos.fedorapeople.org/repos/dchen/apache-maven/epel-apache-maven.repo -O /etc/yum.repos.d/epel-apache-maven.repo
```

c. 执行如下命令，设置存储库的版本号（以4.0.0为例）。

```
sudo sed -i s/4.0.0/6/g /etc/yum.repos.d/epel-apache-maven.repo
```

d. 执行如下命令，安装Maven工具。

```
sudo yum install -y apache-maven
```

> **说明**
>
> 如果回显信息中报类似XXX的错误，您可以执行以下任意一种命令安装Maven。
>
> ```
> sudo yum install -y apache-maven --skip-broken
> ```
>
> ```
> sudo yum install -y apache-maven --nobest
> ```

ii. 创建用于连接图数据库GDB实例的配置文件（以pom.xml为例）。

a. 执行如下命令，创建并进入配置文件所在目录。

```
mkdir gdb-gremlin-test
cd gdb-gremlin-test
```

b. 执行如下命令，创建pom.xml文件。

```
touch pom.xml
```

c. 执行如下命令，编辑pom.xml文件。

```
vi pom.xml
```

输入i，将以下内容复制至pom.xml文件中。

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2
001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/m
aven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.gdb.alibaba</groupId>
  <artifactId>GdbGremlinExample</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>GdbGremlinExample</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
       <groupId>org.apache.tinkerpop</groupId>
       <artifactId>gremlin-driver</artifactId>
       <version>3.4.0</version>
    </dependency>
  </dependencies>
  <build>
     <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>2.0.2</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>exec-maven-plugin</artifactId>
            <version>1.3</version>
            <configuration>
                <mainClass>com.gdb.alibaba.Test</mainClass>
                <complianceLevel>1.8</complianceLevel>
            </configuration>
        </plugin>
    </plugins>
  </build>
</project>
```

按Esc键，输入:wq退出vi编辑器。

iii. 创建用于连接Java客户端的配置文件（以Test.java为例）。

a. 执行如下命令，创建并进入配置文件所在目录。

```
mkdir -p src/main/java/com/gdb/alibaba/
cd src/main/java/com/gdb/alibaba
```

b. 执行如下命令，创建Test.java文件。

```
touch Test.java
```

c. 执行如下命令，编辑Test.java文件。

```
vi Test.java
```

输入i，将以下内容根据业务需求在本地修改完成后复制粘贴至Test.java文件中。

```
package com.gdb.alibaba;
import org.apache.tinkerpop.gremlin.driver.Cluster;
import org.apache.tinkerpop.gremlin.driver.Client;
import org.apache.tinkerpop.gremlin.driver.Result;
import org.apache.tinkerpop.gremlin.driver.ResultSet;
import java.util.List;
import java.util.Map;
import java.util.HashMap;
import java.io.File;
public class Test
{
    public static void main( String[] args )
    {
      try {
        if(args.length != 1) {
            System.out.println("gdb-remote.yaml path needed");
            return;
        }
        String yaml = args[0];

        // 1. 初始化客户端。
        Cluster cluster = Cluster.build(new File(yaml)).create();
        Client client = cluster.connect().init();

        // 2. 配置超时时间为5000毫秒。
        RequestOptions.Builder options = RequestOptions.build().timeout(5000);
        options.addParameter("G___id", "sand131_id_5_99");
        String dsl = "g.V(G___id)";

        // 3. 发送Gremlin请求到图数据库GDB服务端。
        String dsl = "g.addV(yourlabel).property(propertyKey, propertyValue)"; //
yourlabel为点的标签；propertyKey为点的属性；propertyValue为点属性的值。
        Map<String,Object> parameters = new HashMap<>();
        parameters.put("yourlabel","area");
        parameters.put("propertyKey","wherence");
        parameters.put("propertyValue","shenzheng");
        ResultSet results = client.submit(dsl,parameters);
        List<Result> result = results.all().join();
        result.forEach(p -> System.out.println(p.getObject()));

        // 4. 关闭客户端。
        cluster.close();
      } catch (Exception e) {
        System.out.println(e.getMessage());
      }
  }
}
```

按Esc键，输入:wq退出vi编辑器。

iv. 创建用于连接Java客户端和图数据库GDB实例的配置文件（以gdb-remote.yaml为例）。

a. 执行如下命令，创建gdb-remote.yaml文件。

```
touch gdb-remote.yaml
```

b. 执行如下命令，编辑gdb-remote.yaml文件。

```
vi gdb-remote.yaml
```

输入i，将以下内容按照业务需求在本地修改完成后复制粘贴至gdb-remote.yaml文件中。

```
hosts: <your_gdb_endpoint> //<your_gdb_endpoint>为图数据库GDB实例的连接地址。
port: <port> //<port>为图数据库GDB实例的端口号。
username: <username> //<username>为图数据库GDB实例的帐号。
password: <password> //<password>为图数据库GDB实例的帐号密码。
serializer: {
  className: org.apache.tinkerpop.gremlin.driver.ser.GraphBinaryMessageSerializer
V1,
  config: { serializeResultToString: false }
}
```

v. 编译并执行Java程序。

a. 执行如下命令，进入pom.xml文件所在目录（即用于连接图数据库GDB实例的配置文件所在目录）。

```
cd /root/gdb-gremlin-test
```

b. 执行如下命令，编译并执行Java程序。

```
mvn compile exec:java  -Dexec.args="/root/apache-tinkerpop-gmlin-console-3.4.0/co
nf/gdb-remote.yaml"
```

- Python

i. 在客户端安装Python。

a. 执行如下命令，下载Python。

b. 执行如下命令，

ii. （可选）在客户端安装pip。

iii. 连接Python客户端和图数据库GDB实例。

a. 执行如下命令，安装gremlinpython程序包。

b. 执行如下命令，创建用于连接Python客户端和图数据库GDB实例的配置文件（以test.py为例）。

c. 执行如下命令，编辑test.py文件。

d. 执行如下命令，运行test.py文件。

1. 执行如下命令，初始化客户端。

```
client = client.Client('ws://${your-gdb-endpoint}:8182/gremlin', 'g', username="${usern
ame}", password="${password}")
```

2. 执行如下命令，配置超时时间为5000毫秒。

```
dsl = "g.V(G___id)"
parameters = {}
parameters["G___id"] = "sand131_id_5_99"
message = RequestMessage('', 'eval', {'gremlin': dsl, 'scriptEvaluationTimeout': 5000,
'bindings': parameters})
```

3. 执行如下命令，发送请求至图数据库GDB服务端。

```
results = client.submit(message).all().result()
```

4. 执行如下命令，关闭客户端。

```
client.close()
```

5. .Net

   i. 执行如下命令，初始化客户端。

   ```
   var gremlinServer = new GremlinServer(endpoint, port, username: username, password:
   password);
   var gremlinClient = new GremlinClient(gremlinServer);
   ```

   ii. 执行如下命令，配置超时时间为5000毫秒。

   ```
   string dsl = "g.V(G___id)";
   Dictionary<string, object> bindings = new Dictionary<string, object> {{"G___id", "s
   and131_id_5_99"}};
   var msgBuilder = RequestMessage.Build(Tokens.OpsEval).AddArgument(Tokens.ArgsGremli
   n, dsl)
   .AddArgument(Tokens.ArgsBindings, bindings).AddArgument("scriptEvaluationTimeout",
   5000);
   ```

   iii. 执行如下命令，发送请求至图数据库GDB服务端。

   ```
   await gremlinClient.SubmitAsync<T>(msgBuilder.Create()).ConfigureAwait(false);
   ```

   iv. 执行如下命令，关闭客户端。

6. Go

   i. 执行如下命令，初始化客户端。

   ```
   settings := &goClient.Settings{
           Host:     host,
           Port:     port,
           Username: username,
           Password: password,
       }
   client := goClient.NewClient(settings)
   ```

ii. 执行如下命令，配置超时时间为5000毫秒。

```
parameters := make(map[string]interface{})
parameters["G___id"] = "sand131_id_5_99"
options := graph.NewRequestOptionsWithBindings(bindings)
options.SetTimeout(5000)
```

iii. 执行如下命令，发送请求至图数据库GDB服务端。

```
dsl := "g.V(G___id)"
results, err := client.SubmitScriptOptions(dsl, options)
```

iv. 执行如下命令，关闭客户端。

```
client.Close()
```

其他方法配置超时时间

Java bytecode

1. 执行如下命令，初始化客户端。

```
Cluster cluster = Cluster.build(new File(yaml)).create();
Client client = cluster.connect().init();
```

2. 执行如下命令，配置超时时间为5000毫秒。

```
g.with('scriptEvaluationTimeout', 5000).V().limit(1).toList();
```

3. 执行如下命令，发送请求至图数据库GDB服务端。

```
List<Vertex> list = g.V().has("person","name","marko").out("knows").toList();
```

4. 执行如下命令，关闭客户端。

```
cluster.close();
```

Java 请求超时时间配置

```
// 1. 初始化客户端 client

// 2. 配置请求参数，包括超时时间，bindings参数化
RequestOptions.Builder options = RequestOptions.build().timeout(5000);
options.addParameter("G___id", "sand131_id_5_99");

String dsl = "g.V(G___id)";
// 3. 发送请求到GDB服务端，同时提供请求的配置
ResultSet results = client.submit(dsl, options.create());

// 4. 结束任务后关闭客户端 client
```

> 说明
>
> 将sand131_id_5_99替换为实际请求参数。

Gremlin-console 请求超时时间配置

```
:remote config timeout 5000
```

Python 请求超时时间配置

```
from gremlin_python.driver.request import RequestMessage

# 1. 初始化客户端 client

dsl = "g.V(G___id)";
parameters = {}
parameters["G___id"] = "sand131_id_5_99"

# 2. 配置请求，包括超时时间，bindings参数化，请求语句
message = RequestMessage('', 'eval', {'gremlin': dsl, 'scriptEvaluationTimeout': 5000, 'bin
dings': parameters})

# 3. 发送请求到GDB服务端，同时包含有请求的配置
results = client.submit(message).all().result()

# 4. 结束任务后关闭客户端 client
```

> 说明
>
> 将sand131_id_5_99替换为实际请求参数。

.Net 请求超时时间配置

```
// 1. 初始化客户端 client

string dsl = "g.V(G___id)";
Dictionary<string, object> bindings = new Dictionary<string, object> {{"G___id", "sand131_i
d_5_99"}};

// 2. 配置请求，包括超时时间，bindings参数化，请求语句
var msgBuilder = RequestMessage.Build(Tokens.OpsEval).AddArgument(Tokens.ArgsGremlin, dsl)
        .AddArgument(Tokens.ArgsBindings, bindings).AddArgument("scriptEvaluationTimeout",
5000);

// 3. 发送请求到GDB服务端，同时包含有请求的配置
await gremlinClient.SubmitAsync<T>(msgBuilder.Create()).ConfigureAwait(false);

// 4. 结束任务后关闭客户端 client
```

> **说明**
>
> 将sand131_id_5_99替换为实际请求参数。

Go 请求超时时间配置

```
import "github.com/aliyun/alibabacloud-gdb-go-sdk/gdbclient/graph"

// 1. 初始化客户端 client

parameters := make(map[string]interface{})
parameters["G___id"] = "sand131_id_5_99"

// 2. 配置请求参数，包括超时时间，bindings参数化
options := graph.NewRequestOptionsWithBindings(bindings)
options.SetTimeout(5000)

// 3. 发送请求到GDB服务端，同时提供请求的配置
dsl := "g.V(G___id)"
results, err := client.SubmitScriptOptions(dsl, options)

// 4. 结束任务后关闭客户端 client
```

> **说明**
>
> 将sand131_id_5_99替换为实际请求参数。

bytecode请求超时时间配置

图数据库支持bytecode请求的超时时间配置，Java 环境使用如下(其他环境暂不支持)。

```
g.with('scriptEvaluationTimeout', 2000).V().limit(1).toList()
```

1.9. Gremlin请求参数模版化

阿里云GDB推荐使用 `String-based` Gremlin scripts请求与GDB服务端交互，在使用各SDK发送scripts请求时，需要将请求中参数部分模版化以提升查询性能。

> **注意** 参数模版化是使用一个 `Map` 结构参数bindings，将scripts请求中可变数据参数用占位符替换，同时再将占位符和对应的数据值放到bindings，达到使用相同scripts请求、不同bindings完成同类型Gremlin访问。

各语言SDK的参数模版化写法相似，以下给出关键部分代码示例。

Java参数模版化

```
String dsl ="g.addV(G___label).property(id,G___id).property('name',G___name)";
Map<String, Object> parameters = new HashMap<>();
parameters.put("G___id","sand131_id_5_99"); // 点id参数
parameters.put("G___label","person");       // 点label参数
parameters.put("G___name","Jack");          // 点属性'name'值参数
ResultSet results = client.submit(dsl, parameters);
```

python参数模版化

```
dsl ="g.addV(G___label).property(id,G___id).property('name',G___name)"
parameters = {}
parameters["G___id"] = "sand131_id_5_99"
parameters["G___label"] = "person"
parameters["G___name"] = "Jack"
results = client.submit(dsl, bindings=parameters).all().result()
```

.Net参数模版化

```
string dsl ="g.addV(G___label).property(id,G___id).property('name',G___name)";
Dictionary<string, object> parameters = new Dictionary<string, object> {
                    {"G___id", "sand131_id_5_99"},
                    {"G___label", "person"},
                    {"G___name", "Jack"}};
gremlinClient.SubmitAsync<dynamic>(dsl, parameters);
```

Nodejs参数模版化

```
function addVertex1()
{
    return client.submit("g.addV(G___label).property(id,G___id).property('name',G___name)",
{
        G___id: "sand131_id_5_99",
        G___label: "person",
        G___name: "Jack"
    }).then(data => {
        console.log("Add Vertex Result: %s\n", JSON.stringify(data));
    });
}
```

Go参数模版化

```
dsl := "g.addV(G___label).property(id,G___id).property('name',G___name)"
parameters := make(map[string]interface{})
parameters["G___id"] = "sand131_id_5_99"
parameters["G___label"] = "person"
parameters["G___name"] = "Jack"
results, err := client.SubmitScriptBound(dsl, parameters)
```

常见问题

- 模版化参数个数限制

  GDB支持bindings中模版化参数的个数最大值是128，不可修改。

- 模版化中参数的选取

  一类scripts请求通常使用在固定场景时，数据参数中仅有部分是可变的，我们只需要给可变部分建立占位符，固定数据部分可以直接硬编码到请求中。

  以下示例请求是添加标签为'person'，带'name'和'age'两个属性的点。

```
dsl = "g.addV('person').property('name', G___PV1).property('age', G___PV2)"
bindings = {"G___PV1":"Jack","G___PV2":29}
client.submit(dsl, bindings)
```

  参数dsl不变，使用获取到的'name'和'age'数据填充bindings。提交请求写入点数据。

- List参数支持

  部分 `Gremlin Step` 参数不支持数组类型参数，比如ValueMap，需要使用多个占位符分别指定多值类型中的元素。

```
# valueMap参数不支持数组，参数错误
bindings = {"G___PKs":["name","age"]}
dsl = "g.V().hasLabel('person').valueMap(G___PKs)"
client.sumbit(dsl, bindings)
# 使用两个占位符表示两个参数，正确
bindings = {"G___PK1":"name","G___PK2":"age"}
dsl = "g.V().hasLabel('person').valueMap(G___PK1,G___PK2)"
client.submit(dsl, bindings)
```

1.10. GDB用户侧事务控制

阿里云GDB支持事务，默认一个DSL请求的所有操作都在一个事务内。同时GDB也支持用户侧事务控制，同一事务内的多个操作保证原子性，同时成功或失败，事务的隔离级别是Read-Committed。

以下介绍各语言SDK用户控制事务的使用，类似的也只给出关键部分代码。

Java 事务接口

阿里云GDB的Java-SDK提供了方便易用的事务接口，可以参照demo中的 `scriptSessionTest` 示例。

```
Demo demo = new Demo(yaml, true);
GdbClient txClient = demo.client.get();
try {
    // init parameters
    Map<String, Object> parameters = new HashMap();
    parameters.put("vertexStart", vertexStart);
    parameters.put("vertexEnd", vertexEnd);
    parameters.put("vertexLabel", vertexLabel);
    parameters.put("edgeLabel", edgeLabel);

    txClient.batchTransaction((tx, g) -> {
        // add vertex 1
        String dsl = "g.addV(vertexLabel).property(T.id, vertexStart)";
        tx.exec(dsl, parameters, DEFAULT_TIMEOUT_MILLSECOND);

        // add vertex 2
        String dsl2 = "g.addV(vertexLabel).property(T.id, vertexEnd)";
        tx.exec(dsl2, parameters, DEFAULT_TIMEOUT_MILLSECOND);

        // add edge
        String dsl3 = "g.addE(edgeLabel).from(V(vertexEnd)).to(V(vertexStart))";
        tx.exec(dsl3, parameters, DEFAULT_TIMEOUT_MILLSECOND);
    });
} catch (Exception ex) {
    throw new RuntimeException(ex);
} finally {
    // close Session GdbClient
    demo.close();
}
```

上述在接口 `batchTransaction` 中的所有操作（添加两个点和一个边）都在同一事务内，所有操作都返回成功并且无异常抛出，最终更新才会写入。

Go 事务接口

阿里云GDB的Go-SDK也提供方便易用的事务接口，同样可以参照demo的使用示例。

```
// connect GDB with auth
sessionId, _ := uuid.NewUUID()
client := goClient.NewSessionClient(sessionId.String(), settings)

client.BatchSubmit(func(c goClient.ClientShell) error {
    bindings := make(map[string]interface{})
    bindings["GDB___label"] = "goTest"
    bindings["GDB___PK1"] = "name"
    bindings["GDB___PK2"] = "age"
    dsl := "g.addV(GDB___label).property(id, GDB___id).property(GDB___PK1, GDB___PV1).prope
rty(GDB___PK2, GDB___PV2)"

     // 添加点100
    bindings["GDB___id"] = "100"
    bindings["GDB___PV1"] = "Jack"
    bindings["GDB___PV2"] = 32
    _, err := c.SubmitScriptBound(dsl, bindings)
    if err != nil {
      return err
    }

    // 添加点101
    bindings["GDB___id"] = "101"
    bindings["GDB___PV1"] = "Luck"
    bindings["GDB___PV2"] = 34
    _, err = c.SubmitScriptBound(dsl, bindings)
    if err != nil {
      return err
    }

    dsl = "g.addE(GDB___label).from(__.V(GDB___from)).to(__.V(GDB___to)).property(id, GDB__
_id).property(GDB___PK1, GDB___PV1)"
    // 添加边201
    bindings["GDB___id"] = "201"
    bindings["GDB___PV1"] = "created"
    bindings["GDB___from"] = "100"
    bindings["GDB___to"] = "101"
    _, err = c.SubmitScriptBound(dsl, bindings)
    if err != nil {
      return err
    }

    return nil
})
client.Close()
```

上述 `BatchSubmit` 接口中的所有操作（添加两个点和一个边）都在同一个事务内完成，所有操作成功返回 `nil` 则最后更新写入到GDB实例，其中任一处返回 `err(err != nil)` ，整个更新都不会写到GDB实例。

Python 事务接口

事务接口依赖session，`gremlin-python`最新版（v3.4.7）添加了session支持，可以使用session接口手动实现事务功能。

```
session_id = str(uuid.uuid4())
client = Client('ws://${gdb_endpoint}:8182/gremlin', 'g', username=${username}, password=${
password}, session=session_id)

try:
    # 开启事务
    client.submit('g.tx().open()')
    dsl ="g.addV('person').property(id, GDB___id).property('name', GDB___PV)"

    # 添加点100，同步操作
    client.submit(dsl, {'GDB___id':'100', 'GDB___PV':'Jack'}).all().result()

    # 添加点101，同步操作
    client.submit(dsl, {'GDB___id':'101', 'GDB___PV':'Luck'}).all().result()

    # 添加边201，同步操作
    client.submit("g.addE('friend').from(V('100')).to(V('101')).property(id, '201').propert
y('time','2020-03-08')").all().result()

    # 提交事务更新
    client.sumbit('g.tx().commit()')
except Exception:
    # 回滚事务更新
    client.submit('g.tx().rollback()')
client.close()
```

上述使用session机制实现事务操作接口，手动控制事务的开启、提交或回滚。在事务内完成添加2个点一个边的操作，所有操作返回成功无异常就执行最后的事务提交`g.tx().commit()`请求，完成更新到GDB实例的写入。出现异常会进入到回滚请求流程，撤销所有更新。

.Net 事务接口

事务接口依赖session，`gremlin-dotnet`最新版（v3.4.7）添加了session支持，可以使用session接口手动实现事务功能。

```
var gremlinServer = new GremlinServer(${gdb_endpoint}, 8182, username=${username}, password
=${password});
var sessionId = Guid.NewGuid().ToString();
using (var gremlinClient = new GremlinClient(gremlinServer, sessionId: sessionId))
{
    try
    {
        // 开启事务
        await gremlinClient.SubmitAsync("g.tx().open()");

        string dsl ="g.addV('person').property(id,G___id).property('name',G___name)";
        Dictionary<string, object> parameters1 = new Dictionary<string, object> {{"G___id",
"100"},{"G___name", "Jack"}};
        // 添加点100
        await gremlinClient.SubmitAsync<dynamic>(dsl, parameters1);

        Dictionary<string, object> parameters2 = new Dictionary<string, object> {{"G___id",
"101"},{"G___name", "Luck"}};
        // 添加点101
        await gremlinClient.SubmitAsync<dynamic>(dsl, parameters2);

        // 添加边
        string dsl2 = "g.addE('known').property(id,'201').from(V('100')).to(V('101'))";
        await gremlinClient.SubmitAsync<dynamic>(dsl2);

        // 提交更新事务
        await gremlinClient.SubmitAsync("g.tx().commit()");
    }
    catch (Exception ignored)
    {
        // 回滚更新事务
        await gremlinClient.SubmitAsync("g.tx().rollback()");
    }
}
```

Nodejs 事务接口

事务接口依赖session，`gremlin-javascript`最新版（v3.4.7）添加了session支持，可以使用session接口手动实现事务功能。

```
const gremlin = require('gremlin');
const authenticator = new gremlin.driver.auth.PlainTextSaslAuthenticator(${username}, ${pas
sword});
const client = new gremlin.driver.Client(
    'ws://${gdb_endpoint}:8182/gremlin',
    {'authenticator': authenticator,
     'session': ${uuid-session-id}}
);

function addVertex1()
{
    return client.submit("g.addV(GDB___label).property(id, GDB___id).property('name', GDB__
 pv)",{
```

```
        GDB___id: "gdb_vertex_test_id_1",
        GDB___label: "gdb_vertex_test_label",
        GDB___pv: "Jack"
    }).then(data => {
        console.log("Vertex 1: %s\n", JSON.stringify(data));
    });
}

function addVertex2()
{
    return client.submit("g.addV(GDB___label).property(id, GDB___id).property('name', GDB__
_pv)",{
        GDB___id: "gdb_vertex_test_id_2",
        GDB___label: "gdb_vertex_test_label",
        GDB___pv: "Lucy"
    }).then(data => {
        console.log("Vertex 2: %s\n", JSON.stringify(data));
    });
}

function addEdge()
{
    return client.submit("g.addE(GDB___label).from(V(GDB___from)).to(V(GDB___to)).property(
id, GDB___id)",{
        GDB___id: "gdb_edge_test_id",
        GDB___label: "gdb_edge_test_label",
        GDB___from: "gdb_vertex_test_id_1",
        GDB___to: "gdb_vertex_test_id_2"
    }).then(data => {
        console.log("Edge: %s\n", JSON.stringify(data));
    });
}

function beginTx()
{
    return client.submit("g.tx().open()",{
    }).then(data => {
        console.log("open tx\n");
    });
}

function doCommit()
{
    return client.submit("g.tx().commit()",{
    }).then(data => {
        console.log("commit tx\n");
    });
}

function doRollback()
{
    return client.submit("g.tx().rollback()",{
    }).then(data => {
        console.log("commit tx\n");
```

```
    });
}

client.open()
    .then(beginTx)
    .then(addVertex1)
    .then(addVertex2)
    .then(addEdge)
    .then(doCommit)
    .catch((error) => {
        doRollback()
        console.error("Error running query...");
        console.error(error);
    }).then((res) => {
        client.close()
        console.log("Finished, Press any key to exit")
        process.stdin.resume();
        process.stdin.on('data', process.exit.bind(process, 0));
    }).catch((err) => {
        console.error("Fatal error:", err);
    });
```

上述通过回调方式在事务中依次完成添加点和关联边的操作，并提交事务更新。如果中途出现异常进入 `catch` 流程会回滚事务的更新。

常见问题

1. GDB事务接口不支持并发

GDB事务接口底层依赖session机制，同一事务内的操作使用同一个 `session-client` 顺序发送到服务端。

因此 `session-client` 不支持多线程并发请求。

如果您有并发需求，可以使用普通自动事务client。或者创建多个 `session-client` ，每个线程使用一个 `session-client` 顺序发送请求，多个线程实现并发效果。

> **注意**
>
> 多个线程并发操作的事务隔离级别是Read-Committed，即只能读取到已经提交的数据。

2. 操作完成后确保事务时关闭状态

使用客户端控制事务接口时，服务端不会主动提交或回滚事务，请确保所有开启的事务在操作完成后都处于关闭状态（commit/rollback），未完成的事务可能导致数据丢失，同时影响到其他数据项的更新。

GDB在 `session-client` 连接断开后并不会主动关闭该session上的未关闭事务，只有在 `session-client.close()` 调用正常退出时才会关闭。

如果 `session-client` 异常断开时有未关闭事务，服务端默认10min后操作回滚。

3. `session-client` 连接配置

尽量配置 `session-client` 的连接数为1，不主动关闭连接。在后续版本中GDB服务端会更新session模式时连接断开后回滚所有未关闭的事务。

4. 事务接口只支持scripts请求

只支持scripts请求使用事务接口，不支持bytecode方式。

1.11. Gremlin多值属性示例

本文为您介绍Gremlin多值属性示例。

多值属性

- Gremlin语法中对顶点提供了多值属性支持，在添加或者更新属性时可以通过cardinality参数指定属性的类型。
- Gremlin提供了两种类型的多值属性：set和list。GDB从1.0.20版本开始支持set属性。

注意事项

- 只有顶点支持多值属性，边不支持。
- 一个set属性的多个value可以是不同类型，但是GDB在判断值是否相同时是按照语义进行比较的，类型不同但是值相等的两个value会被认为是相同的（例如1和1.0），因而只会保留一个。
- 属性的cardinality可以改变，以最后一次调用property()更新属性为准。set属性更新为single类型后，只保留后者的值；single属性更新为set类型后，前后两个值都保留。
- set属性查询结果中的多个值是按照语义排序的。如果是数值型，按值升序排列；如果是字符串型，按字典序排列。
- set属性和single属性一样，会自动建立索引，查询性能跟single属性相近。

Set属性用法

Console

- 设置属性

```
g.addV('person').
property(id, '11111').
property('name', 'marko').
property(set, 'email', 'marko@a.com').
property(set, 'email', 'marko@b.com')
```

- 查询属性

```
g.V('11111').properties('email')
==>vp[email->marko@a.com]
==>vp[email->marko@b.com]
```

- 删除一个值

```
g.V('11111').properties('email').hasValue('marko@a.com').drop()
```

- 删除所有值

```
g.V('11111').properties('email').drop()
```

Java

```
package com.gdb.alibaba;

import org.apache.tinkerpop.gremlin.driver.Cluster;
import org.apache.tinkerpop.gremlin.driver.Client;
import org.apache.tinkerpop.gremlin.driver.Result;
import org.apache.tinkerpop.gremlin.driver.ResultSet;
import org.apache.tinkerpop.gremlin.structure.util.detached.DetachedVertex;
import org.apache.tinkerpop.gremlin.structure.util.detached.DetachedVertexProperty;

import java.util.List;
import java.util.Map;
import java.util.HashMap;
import java.io.File;

public class Test {
  public static void main(String[] args) {
    try {
      if (args.length != 1) {
        System.out.println("gdb-remote.yaml path needed");
        return;
      }
      String yaml = args[0];
      Cluster cluster = Cluster.build(new File(yaml)).create();
      Client client = cluster.connect().init();
      String dsl = "g.addV(yourLabel).property(propertyKey, propertyValue).property(set, se
tPropertyKey, setPropertyValue0).property(set, setPropertyKey, setPropertyValue1)";
      Map<String, Object> parameters = new HashMap<>();
      parameters.put("yourLabel", "person");
      parameters.put("propertyKey", "name");
      parameters.put("propertyValue", "marko");
      parameters.put("setPropertyKey", "email");
      parameters.put("setPropertyValue0", "marko@a.com");
      parameters.put("setPropertyValue1", "marko@b.com");
      ResultSet results = client.submit(dsl, parameters);
      List<Result> result = results.all().join();
      if (result.size() > 0) {
        String vertexId = (String) ((DetachedVertex) result.get(0).getObject()).id();
        parameters.put("yourId", vertexId);
      }

      dsl = "g.V(yourId).properties(setPropertyKey)";
      results = client.submit(dsl, parameters);
      result = results.all().join();
      result.forEach(r -> {
        Object element = r.getObject();
        if (element instanceof DetachedVertexProperty) {
          DetachedVertexProperty property = (DetachedVertexProperty) element;
          System.out.println(property.key() + ": " + property.value());
        }
```

```
      });

      cluster.close();
    } catch (Exception e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Python

- 用法1：

```
from gremlin_python.process.anonymous_traversal import traversal
from gremlin_python.driver.driver_remote_connection import DriverRemoteConnection
from gremlin_python.process.traversal import T, Cardinality

g = traversal().withRemote(
    DriverRemoteConnection('ws://HOST:PORT/gremlin',
                           'g',
                           username=USER,
                           password=PASS))

v = g.addV('person').property(T.id_, "123").property("name", "marko") \
    .property(Cardinality.set_, "email", "marko@gmail.com") \
    .property(Cardinality.set_, "email", "marko@hotmail.com").iterate()

properties = g.V('123').properties('email')
for prop in properties:
    print(prop.key + ": " + str(prop.value))

g.V('123').drop().iterate()
```

- 用法2：

```
from gremlin_python.driver import client

client = client.Client('ws://HOST:PORT/gremlin',
                       'g',
                       username=USER,
                       password=PASS)

dsl = 'g.addV("person").property(id, "123").property("name", "marko").property(set, "emai
l", "marko@a.com").property(set, "email", "marko@b.com")'
callback = client.submitAsync(dsl)
for result in callback.result():
    print(result)

dsl = 'g.V("123").properties("email")'
callback = client.submitAsync(dsl)
for result in callback.result():
    for prop in result:
        print(prop.key + ": " + str(prop.value))
```

Go

```
    bindings := make(map[string]interface{})
    bindings["GDB___id"] = "22"
    bindings["GDB___label"] = "goTest"
    bindings["GDB___PK"] = "name"
    bindings["GDB___PV"] = "Jack"
    bindings["GDB___PK_PHONE"] = "phone"
    bindings["GDB___PV_PHONE1"] = "111111"
    bindings["GDB___PV_PHONE2"] = "222222"

    dsl := "g.addV(GDB___label).property(id, GDB___id).property(GDB___PK, GDB___PV).propert
y(set, GDB___PK_PHONE, GDB___PV_PHONE1).property(set, GDB___PK_PHONE, GDB___PV_PHONE2)"
    results, err := client.SubmitScriptBound(dsl, bindings)
    if err != nil {
        log.Fatalf("Error while querying: %s\n", err.Error())
    }

    // get response, add vertex should return a Vertex
    for _, result := range results {
        v := result.GetVertex()
        log.Printf("get vertex: %s", v.String())

        // read vertex property
        for _, p := range v.VProperties() {
            log.Printf("prop: %s", p.String())
        }
    }
```

Javascript

```
const gremlin = require('gremlin');
const DriverRemoteConnection = gremlin.driver.DriverRemoteConnection;
const Graph = gremlin.structure.Graph;
const __ = gremlin.process.statics;
const t = gremlin.process.t;
const cardinality = gremlin.process.cardinality
const authenticator =
    new gremlin.driver.auth.PlainTextSaslAuthenticator(USER, PASS);

const graph = new Graph();
const g = graph.traversal().withRemote(
    new DriverRemoteConnection('ws://HOST:PORT/gremlin', {authenticator}));

g.addV('person')
      .property(t.id, '2222')
      .property('name', 'james')
      .property(cardinality.set, 'phone', '111111')
      .property(cardinality.set, 'phone', '222222')
      .iterate()
      .then(data => {
        g.V('2222').properties().toList().then(
            properties => { console.log(properties); });
      });
```

2.Cypher SDK参考

2.1. GDB Cypher实现的兼容性

本文介绍图数据库GDB Cypher实现的兼容性。

协议版本

GDB目前支持bolt-v3协议，建议您使用 `neo4j-java-driver 4.0.0` 版本，使用时除了将GDB域名配置成 `bolt://gdb-endpoint:gdb-port` 之外，其他与原生Cypher使用方式保持一致。

数据类型

GDB Cypher支持整数、浮点数、字符串、布尔值等基本数据类型和空间坐标Point类型，整体如下：

分类	类型	支持情况
Property types	Number / String / Boolean / Spatial / Temporal	不支持Temporal
Structural types	Nodes / Relationships / Paths	支持
Composite types	Lists / Maps	支持

不支持或不完全兼容的类型如下：

- 不支持时间（temporal）类型。
- GDB支持空间坐标（Point）类型，但是目前不支持使用Point类型作为点和边的属性。
- List类型。

List类型只能用作点的属性值，不支持作为边属性。在使用List作为属性时，GDB内部会把List当作Set来处理，即List中多个重复的值会只会被存储一次，支持用户按List中某一元素作为条件进行索引查找。例如：

```
neo4j> create (n:X {p: [1,2,1]}) return n;
+------------------+
| n                |
+------------------+
| (:X {p: [1, 2]}) |
+------------------+
```

子句

操作	支持情况	备注
MATCH	支持	无
OPTIONAL MATCH	支持	无
RETURN	支持	无
WITH	支持	无
UNWIND	支持	无
WHERE	支持	无
ORDER BY	支持	无
SKIP	支持	无
LIMIT	支持	无
CREATE	支持	无
DELETE	支持	无
SET	支持	GDB不支持Label修改以及多Label。 **说明** • 一个点/边的label一旦创建，无法修改，如果要想修改，只能删除点重建。 • 多label中一个点/边只能指定一个label类型。
REMOVE	支持	属性操作，不支持删除Label。
FOREACH	支持	无

操作	支持情况	备注
MERGE	支持	无
CALL {subquery}	不支持	Cypher 4.0新引入， GDB不支持。
CALL procedure	不支持	取决于procedure的实现情况
UNION	支持	无
USE	不支持	Cypher 4.0新引入多图，GDB不支持。
LOAD CSV	不支持	GDB支持基于OSS、DataWorks等导入。

内置函数

GDB对Cypher内置函数的支持如下：

函数	分类	支持情况
all()	谓词	不支持
any()	谓词	不支持
exists()	谓词	是
single()	谓词	不支持
none()	谓词	是
coalesce()	标量	是
endNode()	标量	是
head()	标量	是
id()	标量	是

函数	分类	支持情况
last()	标量	是
length()	标量	是
randomUUID()	标量	不支持
properties()	标量	是
size()	标量	是
startNode()	标量	是
timestamp()	标量	否
toBoolean()	标量	是
toFloat()	标量	是
toInteger()	标量	是
type()	标量	是
avg()	聚合	是
collect()	聚合	是
count()	聚合	是
max()	聚合	部分支持
min()	聚合	部分支持
percentileCont()	聚合	部分支持

函数	分类	支持情况
percentileDisc()	聚合	部分支持
stDev()	聚合	不支持
stDevP()	聚合	不支持
sum()	聚合	支持
keys()	列表	支持
labels()	列表	支持
nodes()	列表	支持
range()	列表	支持
reduce()	列表	支持
relationships()	列表	支持
reverse()	列表	支持
tail()	列表	支持
left()	字符串	不支持
lTrim()	字符串	不支持
reverse()	字符串	支持
right()	字符串	不支持
rTrim()	字符串	不支持

函数	分类	支持情况
split()	字符串	支持
subString()	字符串	支持
toLower()	字符串	支持
toString()	字符串	支持
toUpper()	字符串	支持
trim()	字符串	支持
abs()	数值	支持
ceil()	数值	不支持
floor()	数值	不支持
rand()	数值	不支持
round()	数值	支持
sign()	数值	不支持
e()	对数函数	不支持
exp()	对数函数	不支持
log()	对数函数	不支持
log10()	对数函数	不支持
sqrt()	对数函数	支持

函数	分类	支持情况
acos()	三角函数	不支持
atan()	三角函数	不支持
atan2()	三角函数	不支持
cos()	三角函数	不支持
cot()	三角函数	不支持
degrees()	三角函数	不支持
haversin()	三角函数	不支持
pi()	三角函数	不支持
radians()	三角函数	不支持
sin()	三角函数	不支持
tan()	三角函数	不支持
point()	空间	支持
distance()	空间	支持

说明

- max()/min()/percentileCount()/percentileDisc()四个函数只支持输入参数为相同类型的List，不支持混合类型，也不支持包含NULL的List。
- GDB不支持时间类型，因此时间相关的函数没有列在表里。

不完全兼容的用法

- LOAD CSV。GDB提供其他数据导入方式，因而不支持LOAD CSV子句以及相关的 `linenumber()` 和 `file()` 函数。
- 索引机制。GDB会自动为所有属性建立自动索引，用户也可通过接口进行复合索引以及点中心索引的设置。不支持通过Cypher方式建立索引。
- 不支持Bookmark。
- GDB不支持用户自定义函数（User-defined Function）和APOC。
- driver。用户可使用包括Java、Python、Go、Javascript、C#等语言的官方driver来访问GDB，也可以通过Spring方式来接入GDB。
- Cypher中其他跟管理相关功能都使用和Gremlin接入兼容的方式，包括创建和删除数据库、索引管理、属性约束（constraints）、访问控制等。

事务支持

目前支持Simple和Async Session模式的事务，暂不支持Reactive Session。

Session模式	Atomic Transaction	Transaction Function
Simple Sessions	支持	支持
Async Sessions	支持	支持
Reactive Sessions	不支持	不支持

Spring支持

GDB支持通过spring方式访问，目前spring的访问接口支持程度如下：

Spring-data-neo4j 默认函数	支持情况
S save(S entity);	支持
Iterable saveAll (Iterable entities);	支持
Optional findById(ID id);	支持
boolean existsById(ID id);	支持

Spring-data-neo4j 默认函数	支持情况
Iterable findAll();	支持
Iterable findAllById(Iterable ids);	支持
long count();	支持
void deleteById(ID id);	支持
void delete(T entity);	支持
void deleteAll(Iterable<? extends T> entities);	支持
void deleteAll();	支持
Iterable findAll(Sort sort);	支持
Page findAll(Pageable pageable);	支持
S extends T> S save(S s, int depth);	支持
<S extends T> Iterable<S> save(Iterable<S> entities, int depth);	支持
Optional findById(ID id, int depth);	支持
Iterable findAll(int depth);	支持
Iterable findAll(Sort sort, int depth);	支持
Iterable findAllById(Iterable ids, int depth);	支持

Spring-data-neo4j 默认函数	支持情况
Iterable findAllById(Iterable ids, Sort sort);	支持
Iterable findAllById(Iterable ids, Sort sort, int depth);	支持
Page findAll(Pageable pageable, int depth);	支持

可视化

目前提供开源客户端使用，后续会集成在GDB DMS系统中。

2.2. Cypher SDK用法示例

GDB支持OpenCypher查询语言，并兼容bolt协议，可以通过Neo4j driver进行访问。以下为各种常用driver访问GDB的示例。

Java

环境准备

- 安装JDK 1.8以上版本。
- 安装maven。

代码示例

1. 创建并进入项目目录。

```
mkdir cypher-java-example
cd cypher-java-example
```

2. 创建pom.xml文件，内容如下。

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XM
LSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v
4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.gdb.alibaba</groupId>
  <artifactId>GdbGremlinExample</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>GdbGremlinExample</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
       <groupId>org.neo4j.driver</groupId>
       <artifactId>neo4j-java-driver</artifactId>
       <version>4.0.1</version>
    </dependency>
  </dependencies>
  <build>
     <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>2.0.2</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>exec-maven-plugin</artifactId>
            <version>1.3</version>
            <configuration>
                <mainClass>com.example.gdb.HelloWorld</mainClass>
                <complianceLevel>1.8</complianceLevel>
            </configuration>
        </plugin>
    </plugins>
  </build>
</project>
```

3. 创建源码目录。

```
mkdir -p src/main/java/com/example/gdb/
```

4. 编辑文件src/main/java/com/example/gdb/HelloWorld.java。其中GDB_HOST、GDB_PORT、GDB_USER、GDB_PASSWORD分别替换成对应实例的地址、端口、用户名和密码。

```
package com.example.gdb;
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.Session;
import org.neo4j.driver.Result;
import org.neo4j.driver.Transaction;
import org.neo4j.driver.TransactionWork;
import static org.neo4j.driver.Values.parameters;
public class HelloWorld implements AutoCloseable
{
    private final Driver driver;
    public HelloWorld( String uri, String user, String password )
    {
        driver = GraphDatabase.driver( uri, AuthTokens.basic( user, password ) );
    }
    @Override
    public void close() throws Exception
    {
        driver.close();
    }
    public void printGreeting( final String message )
    {
        try ( Session session = driver.session() )
        {
            String greeting = session.writeTransaction( new TransactionWork<String>()
            {
                @Override
                public String execute( Transaction tx )
                {
                    Result result = tx.run( "CREATE (a:Greeting) " +
                                            "SET a.message = $message " +
                                            "RETURN a.message + ', from node '
+ id(a)",
                            parameters( "message", message ) );
                    return result.single().get( 0 ).asString();
                }
            } );
            System.out.println( greeting );
        }
    }
    public static void main( String... args ) throws Exception
    {
        try ( HelloWorld greeter = new HelloWorld( "bolt://GDB_HOST:GDB_PORT", "GDB_USE
R", "GDB_PASSWORD" ) )
        {
            greeter.printGreeting( "hello" );
        }
    }
}
```

5. 编译运行。

```
mvn compile exec:java
```

实例运行结果如下：

```
hello, from node 1000010
```

Python

环境准备

- 安装CPython 2.7或3.4以上，建议Python3。
- 安装pip。
- 安装neo4j driver。如果同时安装了多个版本的python和pip，需要用pip2或者pip3为指定版本安装。

```
pip install neo4j --user
```

代码示例

1. 编辑文件cypher-python-example.py，内容如下。其中GDB_HOST、GDB_PORT、GDB_USER、GDB_PASSWORD分别替换成对应实例的地址、端口、用户名和密码。

```
from neo4j import GraphDatabase
class HelloWorldExample(object):
    def __init__(self, uri, user, password):
        self._driver = GraphDatabase.driver(uri,
                                            auth=(user, password),
                                            encrypted=False)
    def close(self):
        self._driver.close()
    def print_greeting(self, message):
        with self._driver.session() as session:
            greeting = session.write_transaction(
                self._create_and_return_greeting, message)
            print(greeting)
    @staticmethod
    def _create_and_return_greeting(tx, message):
        result = tx.run(
            "CREATE (a:Greeting) "
            "SET a.message = $message "
            "RETURN a.message + ', from node ' + id(a)",
            message=message)
        return result.single()[0]
e = HelloWorldExample('bolt://GDB_HOST:GDB_PORT', 'GDB_USER', 'GDB_PASSWORD')
r = e.print_greeting('hello')
```

2. 编译运行。

```
python cypher-python-example.py
```

运行结果如下：

```
hello, from node 1000003
```

Neo4j Python driver的更多信息，请参阅Neo4j官网文档。

Go

环境准备

- 安装go 1.11以上。
- 安装seabolt。Neo4j go driver依赖于seabolt，推荐从源码编译安装。安装方法以Mac OS为例，其他安装方法参考seabolt的Github主页。

```
# 安装openssl
brew install openssl
# 设置环境变量OPENSSL_ROOT_DIR
export OPENSSL_ROOT_DIR=/usr/local/opt/openssl
# 解压seabolt源码包
tar zxvf seabolt-1.7.4.tar.gz
cd seabolt-1.7.4
mkdir build && cd build
cmake ..
make install
```

代码示例

1. 创建和初始化项目目录。

```
mkdir cypher-go-example
cd cypher-go-example
go mod init cypher-go-example
```

2. 编辑main.go文件，内容如下。其中GDB_HOST、GDB_PORT、GDB_USER、GDB_PASSWORD分别替换成对应实例的地址、端口、用户名和密码。

```
package main
import (
    "fmt"
    "github.com/neo4j/neo4j-go-driver/neo4j"
)
func helloWorld(uri, username, password string) (string, error) {
    var (
        err      error
        driver   neo4j.Driver
        session  neo4j.Session
        result   neo4j.Result
        greeting interface{}
    )
    driver, err = neo4j.NewDriver(uri, neo4j.BasicAuth(username, password, ""))
    if err != nil {
        return "", err
    }
    defer driver.Close()
    session, err = driver.Session(neo4j.AccessModeWrite)
    if err != nil {
        return "", err
    }
    defer session.Close()
    greeting, err = session.WriteTransaction(func(transaction neo4j.Transaction) (inter
face{}, error) {
        result, err = transaction.Run(
            "CREATE (a:Greeting) SET a.message = $message RETURN a.message + ', from no
de ' + id(a)",
            map[string]interface{}{"message": "hello"})
        if err != nil {
            return nil, err
        }
        if result.Next() {
            return result.Record().GetByIndex(0), nil
        }
        return nil, result.Err()
    })
    if err != nil {
        return "", err
    }
    return greeting.(string), nil
}
func main() {
    greeting, err := helloWorld("bolt://GDB_HOST:GDB_PORT", "GDB_USER", "GDB_PORT")
    if err != nil {
        fmt.Println(err)
    }
    fmt.Println(string(greeting))
}
```

3. 编译并运行示例，go会自动下载安装driver并添加依赖关系到go.mod文件中。

```
go build ./.. && ./cypher-go-example
# 或者
go run main.go
```

示例运行结果如下：

```
hello, from node 1000004
```

.Net

环境准备：下载安装.net，版本2.0以上。

代码示例

1. 创建项目，并进入项目目录。

```
dotnet new console -o cypher-dotnet-example
cd cypher-dotnet-example
```

2. 安装Neo4j.Driver。

```
dotnet add package Neo4j.Driver.Simple
```

3. 编辑Program.cs，内容如下。其中GDB_HOST、GDB_PORT、GDB_USER、GDB_PASSWORD分别替换成对应实例的地址、端口、用户名和密码。

```
using System;
using System.Linq;
using Neo4j.Driver;
namespace cypherTest
{
    public class HelloWorldExample : IDisposable
    {
        private readonly IDriver _driver;
        public HelloWorldExample(string uri, string user, string password)
        {
            _driver = GraphDatabase.Driver(uri, AuthTokens.Basic(user, password));
        }
        public void PrintGreeting(string message)
        {
            using (var session = _driver.Session())
            {
                var greeting = session.WriteTransaction(tx =>
                {
                    var result = tx.Run("CREATE (a:Greeting) " +
                                        "SET a.message = $message " +
                                        "RETURN a.message + ', from node ' + id(a)",
                        new {message});
                    return result.Single()[0].As<string>();
                });
                Console.WriteLine(greeting);
            }
        }
        public void Dispose()
        {
            _driver?.Dispose();
        }
        public static void Main()
        {
            using (var greeter = new HelloWorldExample("bolt://GDB_HOST:GDB_PORT", "GDB
_USER", "GDB_PASSWORD"))
            {
                greeter.PrintGreeting("hello");
            }
        }
    }
}
```

4. 编译运行。

```
dotnet run
```

实例运行结果如下：

```
hello, from node 1000007
```

Neo4j .Net driver的更多信息，请参阅Neo4j官方文档。

Node.js

环境准备：安装Node.js，建议使用LTS版本（10.x、12.x）。

代码示例

1. 新建并进入项目目录。

```
mkdir cypher-js-example
cd cypher-js-example
```

2. 添加Neo4j driver。

```
npm install neo4j-driver
```

3. 编辑demo.js，内容如下。其中GDB_HOST、GDB_PORT、GDB_USER、GDB_PASSWORD分别替换成对应实例的地址、端口、用户名和密码。

```
const neo4j = require("neo4j-driver");
const driver = neo4j.driver('bolt://GDB_HOST:GDB_PORT',
                            neo4j.auth.basic('GDB_USER', 'GDB_PASSWORD'));
async function helloWorld() {
  const session = driver.session();
  try {
    const result = await session.writeTransaction(
        tx => tx.run(
            'CREATE (a:Greeting) SET a.message = $message RETURN a.message + ", from no
de " + id(a)',
            {message : 'hello'}));
    const singleRecord = result.records[0];
    const greeting = singleRecord.get(0);
    return greeting;
  } finally {
    await session.close();
  }
}
helloWorld().then(msg => {
  console.log(msg);
  driver.close();
});
```

4. 编译运行。

```
node demo.js
```

实例运行结果如下：

```
hello, from node 1000008
```

Neo4j Javascript driver的更多信息，请参阅Neo4j官方文档。