

图数据库 JanusGraph 实战

版本号	V0.3.0
文档 ID	
日期	2019 年 04 月 21 日
作者	胡佳辉（昵称：家辉）
版权所有，未经作者许可，不得用于商业目的。 如需合作，欢迎添加作者微信（cddennishu）	
<p style="text-align: center;">几点说明</p> <p>第一：《图数据库 JanusGraph 实战》是一个还未完成的版本，后续还会持续更新；</p> <p>第二：这本电子书原计划是要出一个课程，但由于作者近期太忙没有时间录制，又担心一直搁置而耽误广大数据库爱好者的学习，经慎重决定，将其免费开放给大家！如果本教程确实对你有所帮助，欢迎扫描下方二维码或者支付宝二维码打赏作者，请作者喝杯咖啡，在此以表感谢！</p> <p>第三：本教程的最新版本会更新到“图数据库与图计算”QQ 群里！</p>	
微信	支付宝
	

版本号	修订人	修订日期	修订描述
V0.3.0	胡佳辉	20190420	对外发布的第一个版本（非完整版）
V0.2.0	胡佳辉	20190108	更新基础内容，建立教程框架
V0.1.0	胡佳辉	20180816	创建文档

目录

1 引言	8
1.1 目的	8
1.2 背景	8
1.3 参考资料	8
1.3.1 参考书籍	8
1.3.2 参考网址	8
1.4 术语	8
2 图数据库基础	8
2.1 图数据库基本概念	8
2.1.1 什么是图数据库	8
2.1.2 属性图	8
2.2 数据库的分类	9
2.3 图数据库与图计算	9
2.3.1 图数据库与图计算的区别	9
2.3.2 主流图数据库介绍	9
2.3.3 图数据库的派系	9
3 JANUSGRAPH 入门	10
3.1 JANUSGRAPH 简介	10
3.1.1 JanusGraph 概述	10
3.1.2 JanusGraph 与 Tinkerpop 的关系	11
3.1.3 JanusGraph 与 Neo4j 的对比	11
3.2 JANUSGRAPH 架构	11
3.3 JANUSGRAPH 安装	12
3.3.1 JanusGraph 安装说明	12
3.3.2 TinkerGraph 内存图数据库	13
3.3.3 Windows 下 JanusGraph+Berkeley+ES 安装	13
3.3.3.1 版本选择	13
3.3.3.2 安装包下载	13
3.3.3.3 安装与配置	13
3.3.3.4 基本使用	14
3.4 LINUX 下 JANUSGRAPH 的安装	15
3.4.1 Linux 下 JanusGraph 的安装说明	15
3.4.2 Linux 下 JanusGraph 的安装步骤	15
3.4.3 JanusGraph 的启动	15
3.4.3.1 启动 ElasticSearch	15
3.4.3.2 JanusGraph 的基本使用	15
4 JANUSGRAPH 建模	16
4.1 图数据库建模	16
4.1.1 图数据库的基本模型	16
4.2 定义 JANUSGRAPH SCHEMA	17
4.2.1 顶点标签	17
4.2.1.1 顶点标签的概念	17
4.2.1.2 顶点标签的创建	17
4.2.1.3 顶点标签的查看	17
4.2.1.4 顶点标签的修改	18
4.2.1.5 顶点标签的删除	18
4.2.1.6 删除整个 graph	18
4.2.2 边标签	18
4.2.2.1 边标签的概念	18
4.2.2.2 边标签的创建	19

4.2.2.3 边标签的查看.....	19
4.2.2.4 边标签的删除.....	19
4.2.3 属性键.....	19
4.2.3.1 属性键的类型.....	19
4.2.3.2 属性的基数.....	20
4.2.3.3 属性键的创建.....	20
4.2.3.4 属性键的查看.....	20
4.2.3.5 属性键的修改.....	20
4.2.3.6 属性键的删除.....	20
4.3 自动创建 SCHEMA.....	20
4.3.1 模型快速查看.....	20
4.4 ID 管理.....	21
5 GREMLIN 查询语言.....	21
5.1 GREMLIN 概述.....	21
5.2 GRAPH STRUCTURE API.....	23
5.2.1 创建.....	23
5.2.2 其他.....	23
5.2.2.1 查看当前图.....	23
5.2.2.2 查看图的 features.....	23
5.2.2.3 给图添加变量.....	25
5.3 GRAPH PROCESS API.....	25
5.4 GREMLIN CONSOLE.....	25
5.4.1 Gremlin 帮助.....	25
5.5 图事务.....	26
5.5.1 提交事务.....	26
5.5.2 回滚事务.....	26
5.5.3 查询事务.....	26
5.6 GREMLIN 增删改查.....	26
5.6.1 创建顶点.....	26
5.6.1.1 创建无标签的顶点.....	26
5.6.1.2 创建带标签的顶点.....	26
5.6.1.3 创建带标签和属性的顶点.....	26
5.6.2 查看顶点.....	27
5.6.2.1 查看所有顶点.....	27
5.6.2.2 根据标签查看顶点.....	27
5.6.2.3 根据属性查看顶点.....	27
5.6.2.4 根据 id 查看顶点.....	27
5.6.2.5 查看顶点的标签.....	27
5.6.2.6 查看顶点的属性键值对.....	27
5.6.2.7 查看顶点的所有信息.....	27
5.6.2.8 根据标签和属性一起查.....	27
5.6.3 创建边.....	28
5.6.3.1 创建边.....	28
5.6.3.2 查看边.....	28
5.6.4 顶点属性.....	28
5.6.4.1 添加或更新属性.....	28
5.6.4.2 查看所有属性值.....	28
5.6.4.3 查看单个属性值.....	28
5.6.4.4 查看多个属性值.....	28
5.6.4.5 删除某个属性.....	28
5.6.5 边属性.....	28
5.6.5.1 添加/更新边属性.....	28
5.6.5.2 查看边属性.....	28
5.6.6 删除.....	28
5.6.6.1 删除所有的顶点.....	28
5.6.6.2 删除所有的边.....	29

5.6.7 退出.....	29
5.7 GREMLIN 谓词.....	29
5.7.1 常规谓词.....	29
5.7.1.1 group.....	29
5.7.1.2 Limit/tail.....	29
5.7.1.3 range.....	29
5.7.1.4 between.....	29
5.7.1.5 skip.....	30
5.7.1.6 as.....	30
5.7.1.7 select.....	30
5.7.1.8 project.....	30
5.7.1.9 order.....	30
5.7.1.10 统计 count/sum/max/min/mean.....	30
5.7.1.11 and/or/not/is.....	30
5.7.1.12 where.....	30
5.7.1.13 union.....	30
5.7.2 特有谓词.....	30
5.7.2.1 next.....	30
5.7.2.2 toList/toSet/toBulkSet.....	31
5.7.2.3 fill.....	31
5.7.2.4 fold/unfold.....	31
5.7.2.5 path.....	31
5.7.2.6 by.....	31
5.7.2.7 dedup.....	31
5.7.2.8 repeat.....	31
5.7.2.9 join.....	31
5.7.2.10 local.....	31
5.7.2.11 choose.....	31
5.7.2.12 option.....	32
5.7.2.13 match.....	32
5.7.2.14 constant.....	32
5.7.2.15 sideEffect.....	32
5.7.2.16 aggregate.....	32
5.7.2.17 inject.....	32
5.7.2.18 coalesce.....	32
5.7.2.19 optional.....	32
5.7.2.20 both/bothV/bothE/otherV.....	32
5.7.2.21 emit.....	32
5.7.2.22 cyclicPath().....	32
5.7.2.23 Gremlin' s scientific calculator math.....	32
5.7.2.24 Lambda 函数.....	33
5.7.2.25 filter.....	33
5.7.2.26 sack.....	33
5.7.2.27 store.....	34
5.8 其他搜索谓词.....	34
5.9 GREMLIN 语言变体.....	34
6 JANUSGRAPH 索引.....	34
6.1 图索引.....	34
6.1.1 索引的状态.....	34
6.1.2 复合索引 <i>Composite Index</i>	35
6.1.2.1 创建索引.....	35
6.1.2.2 查看索引状态.....	35
6.1.2.3 更新索引状态.....	35
6.1.2.4 索引分析.....	36
6.1.2.5 多字段索引.....	37
6.1.2.6 唯一性索引.....	37
6.1.2.7 相等索引.....	38
6.1.3 混合索引 <i>Mixed Index</i>	39
6.1.3.1 创建索引.....	39
6.1.3.2 查看状态.....	40
6.1.3.3 索引重建.....	40

6.1.3.4 检查索引.....	40
6.1.3.5 添加索引到新建的键.....	40
6.2 标签约束.....	40
6.3 以顶点为中心的索引.....	41
6.4 索引重建 REINDEXING.....	41
6.4.1 MapReduce 索引重建.....	41
6.5 索引禁用与删除.....	41
6.5.1 索引查看.....	41
6.5.1.1 查看所有顶点索引.....	41
6.5.1.2 查看所有边索引.....	41
6.5.2 索引禁用.....	42
6.5.3 索引删除.....	42
7 JANUSGRAPH 数据库管理.....	42
7.1 JANUSGRAP 管理系统.....	42
7.2 JANUSGRAPH 后端存储.....	42
7.2.1 Oracle Berkley DB.....	42
7.2.2 Apache Cassandra.....	42
7.2.3 ScyllaDB.....	42
7.2.4 Apache HBase.....	42
7.2.5 后端存储的选型问题.....	42
7.3 JANUSGRAPH 后端索引.....	43
7.3.1 后端索引概述.....	43
7.3.1.1 一般索引.....	43
7.3.1.2 文本索引.....	43
7.3.1.3 空间索引.....	43
7.3.1.4 索引使用例子.....	43
7.3.1.5 支持的数据类型.....	43
7.3.2 Elastic Search.....	44
7.3.3 Apache Solr.....	44
7.3.4 Apache Lucene.....	44
7.4 JANUSGRAPH 部署模式.....	44
7.4.1 嵌入式.....	44
7.4.2 单机模式.....	44
7.4.3 集群模式.....	44
7.5 JANUSGRAPH+HBASE+ELASTICSEARCH 安装.....	44
7.5.1 版本选择.....	44
7.5.2 软件下载.....	44
7.5.3 JanusGraph+HBase 独立模式+单机 ElasticSearch.....	44
7.5.3.1 JanusGraph 安装.....	44
7.5.3.2 ElasticSearch 安装.....	45
7.5.3.3 HBase 安装.....	45
7.5.3.4 安装验证.....	46
7.5.4 Hadoop+HBase 伪分布式模式.....	46
7.5.5 Hadoop+HBase 完全分布式模式.....	46
7.6 JANUSGRAPH+CASSANDRA+ELASTICSEARCH.....	46
7.6.1 Cassandra 安装.....	46
7.7 JANUSGRAPH 配置与日志.....	46
7.8 JANUSGRAPH 备份与恢复.....	46
7.9 JANUSGRAPH 导入与导出.....	46
7.9.1 csv 导入.....	47
7.9.2 自定义导入类.....	47
7.9.3 bulk load.....	47
7.9.3.1 storage.batch-loading.....	47
7.9.3.2 id 分配优化.....	47
7.9.3.3 优化读写.....	47

7.9.4 GraphML.....	47
7.9.4.1 导入.....	47
7.9.4.2 导出.....	47
7.9.5 GraphSON.....	47
7.9.5.1 导入.....	47
7.9.5.2 导出.....	47
7.9.6 Gryo.....	48
8 JANUSGRAPH 开发.....	48
8.1 嵌入式模式.....	48
8.1.1 Graph Structure API.....	48
8.1.2 graph traversal 方法.....	49
8.2 JANUSGRAPH SERVER.....	50
8.2.1 Windows 环境.....	50
8.2.2 Linux 环境.....	51
8.2.2.1 Cassandra+ES(默认).....	51
8.2.2.2 Berkeley+es.....	51
8.3 JAVA 开发(GREMLIN-DRIVER).....	52
8.4 PYTHON 开发 (GREMLIN-PYTHON)	52
8.5 REST 开发(HTTP).....	53
8.5.1 JavaWebsocket.....	53
8.5.1.1 工程搭建.....	53
8.5.1.2 maven 引用.....	53
8.5.1.3 代码.....	53
8.5.2 定制 Gremlin-Server 的配置文件.....	53
8.6 JANUSGRAPH 认证.....	53
8.7 JANUSGRAPH 与 SPARK 集成.....	53
8.7.1 插件安装.....	53
8.7.2 配置.....	54
9 JANUSGRAPH 图计算.....	54
9.1 HADOOP 与 JANUSGRAPH 集成.....	54
9.1.1 Hadoop 伪分布式环境搭建.....	54
9.1.2 启动 JanusGraph.....	54
9.1.3 Spark 图计算.....	54
10 JANUSGRAPH 可视化.....	54
10.1 可视化概述.....	54
10.2 GRAPHEXP 集成.....	55
10.2.1 源码安装方式.....	55
10.2.2 Docker 安装方式.....	55
10.3 GEPHI 插件集成.....	55
10.3.1 下载与安装.....	55
10.3.2 集成.....	55
11 JANUSGRAPH 集群.....	56
12 应用案例分析.....	56
13 性能优化.....	56
13.1 EXPLAIN.....	56
13.2 PROFILE.....	56
13.3 CLOCK/CLOCKWITHRESULT.....	57

1 引言

1.1 目的

供图数据库 JanusGraph 初学者使用！

1.2 背景

图数据库被誉为大数据时代的高铁，JanusGraph 作为开源图数据库中的优秀代表，已经被 Uber、RedHat、360、58、同盾等众多大公司应用于商业环境。

同时，JanusGraph 基于 Tinkerpop 实现，一旦掌握了 JanusGraph，很容易对 Tinkerpop 系的其他图数据库触类旁通。学习图数据库时，选择 JanusGraph 的理由：

- 完全开源，且支持分布式；
- 基于 Tinkerpop，掌握后容易对其他图数据库触类旁通；
- 已经在众多公司生产环境中经过检验，性能可靠；
- 灵活的架构，支持多种组合；

1.3 参考资料

1.3.1 参考书籍

- [1] 《Neo4j 权威指南》 张帆、庞国明、胡佳辉 等 清华大学出版社 2017 年
- [2] 《Neo4j 3.x 入门经典》 张帆、庞国明、胡佳辉 等 清华大学出版社 2019 年

1.3.2 参考网址

- [1] <http://janusgraph.org/>
- [2] <http://tinkerpop.apache.org/>
- [3] <http://kelvinlawrence.net/book/Gremlin-Graph-Guide.html>
- [4] <https://cloud.google.com/blog/products/gcp/developing-a-janusgraph-backed-service-on-google-cloud-platform>

1.4 术语

- [1] 图：Graph
- [2] 顶点：Vertex
- [3] 标签：Label
- [4] 边：Edge
- [5] 属性：Property
- [6] 领域特定语言：DSL（Domain Specific Language）

2 图数据库基础

2.1 图数据库基本概念

2.1.1 什么是图数据库

要理解什么是图数据库，首先要弄清楚什么是图。这里的图是指 Graph，源自数学中的图论。即由顶点和边（也称节点和关系）构成的关系网络。

图数据库是基于图论实现的一种新型 NoSQL 数据库，它是一种提供了图数据存储、读取和分析的数据库系统。

2.1.2 属性图

图数据要存入图数据库系统中，需要涉及特定的图数据模型，即：如何存、采用什么实现方式来存图数据。常用的有三种：属性图（Property Graphs）、超图（Hypergraphs）和三元组（Triples）。

属性图直观更易于理解，能描述绝大部分图使用场景，也是当下最流行的图数据模型。

JanusGraph、Neo4j 等采用的都是这种属性图模型。符合下列特征的图数据模型就称为属性图：

- 它包含顶点和边；
- 顶点可以有属性（键值对）；
- 顶点可以有一个或多个标签；
- 边有标签和方向，并总是有一个开始节点和一个结束节点；

2.2 数据库的分类

数据库从大的方面可以分为两类：即关系型数据库（RDBMS）和非关系型（NoSQL）。

常见的关系型数据库包括：

- Oracle
- MySQL
- SQL Server
- PostgreSQL
- IBM DB2

非关系型数据库又可以分为：

- 键值型数据库：Redis、MemCached 等
- 文档型数据库：MongoDB
- 列式数据库：HBase、Cassandra 等
- 图数据库：JanusGraph、Neo4j 等

2.3 图数据库与图计算

2.3.1 图数据库与图计算的区别

图数据库的核心在于数据的存储与读取，需要支持事务（OLTP：Online Transaction Processing），满足 ACID，即数据库事务正确执行的四个基本要素的缩写。包含：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。

OLTP 通常来说是对少量数据的操作，操作的响应时间较短，一般在毫秒和秒级。

而图计算则侧重于图的分析（OLAP：Online Analytical Processing），通常需要跨越整个图，需要读取大量的图数据、并且需要强大的集群节点来支持分析。

OLAP 通常对大量数据的分析，操作的响应时间也长，少则数分钟，多则数小时。

JanusGraph 提供了与 Hadoop、Spark 等工具的集成来支持图计算。

2.3.2 主流图数据库介绍

目前市场上主流的图数据库主要包含只支持图存储的图数据库系统（Graph DBMS）和多模图数据库系统（Multi-model DBMS）。

图数据库系统只支持图模型的存储，主流的图数据库系统按市场占有率排序先后为：Neo4j、JanusGraph、Apache Giraph、DGraph、TigerGraph，以及百度 2018 年 8 月开源的 HugeGraph。

多模数据库系统除了支持图模型存储之外，还支持其他模型，比如键值对、文档型等。主流的多模图数据库系统按市场占有率排序先后为：

Microsoft Azure Cosmos DB、Datastax Enterprise、OrientDB、ArangoDB、Virtuoso 等。

这里需要提及的一点是：Titan 是一个很有名的开源图数据库、但 Titan 被 Datastax 收购之后就不再开源，它被纳入到 Datastax Enterprise 版本中。Titan 的后续开源项目就是 JanusGraph。

2.3.3 图数据库的派系

目前单纯的图数据库系统主要分为两个流派：Neo4j 派和 Tinkerpop 派。

Neo4j 是目前公认的最优秀的图数据库之一，目前市场占有率为 45.56%，排名第一。Neo4j 背后是 Neo Technology 公司，目前 Neo4j 社区版开源、企业版已于 2018 年 11 月开始不再开源。

Tinkerpop 是 Apache 下面的一个图数据库与图计算框架，它自己只提供了一个基于内存实现的 TinkerGraph 图数据库系统，大多情况下用于测试。它本身没有提供一个产品级的图数据库系统，这为其他图数据库的发展提供了广阔的空间，由此衍生出 Tinkerpop 派系的很多产品。

如：Apache 下面的 Giraph、JanusGraph、微软的 Cosmos DB、百度开源的 HugeGraph、华为的 Graph Engine 等等。

3 JanusGraph 入门

3.1 JanusGraph 简介

3.1.1 JanusGraph 概述

JanusGraph 是一个开源的分布式图数据库。它的前身是著名的开源图数据库 Titan，但 Titan 被 DataStax 收购之后就不再开源了。JanusGraph 是在原 Titan 的基础上继续以开源的形式开发和发布，它的授权许可是 Apache2 license，具有很好的商用友好性。

JanusGraph 于 2017 年发布 0.1.0 版本，当前（截止 2018-12-06）最新版本为 0.3.1。JanusGraph 具有很好的扩展性，通过多机集群可支持存储和查询数亿的顶点和边的图数据。JanusGraph 是一个事务数据库，支持大量用户高并发地执行复杂的实时图遍历。

它提供了如下特性：

- 支持数据和用户增长的弹性和线性扩展；
- 通过数据分发和复制来提过性能和容错；
- 支持多数据中心的高可用和热备份；
- 支持 ACID 和最终一致性；
- 支持多种后端存储：
 - Apache Cassandra
 - Apache HBase
 - Google Cloud Bigtable
 - Oracle BerkeleyDB
- 支持全局的图数据分析，报表以及和如下大数据平台的 ETL 集成：
 - Apache Spark
 - Apache Giraph
 - Apache Hadoop
- 支持 geo, 数值范围以及通过如下工具进行全文检索：
 - ElasticSearch
 - Apache Solr
 - Apache Lucene
- 支持与 Apache TinkerPop 图栈进行原生集成：
 - Gremlin 图查询语言
 - Gremlin 图服务器
 - Gremlin 应用
- 在 Apache 2 license 下开源
- 可以通过如下工具来可视化存储在 JanusGraph 中的图数据：
 - Cytoscape
 - Apache TinkerPop 的 Gephi 插件
 - Graphexp
 - KeyLines by Cambridge Intelligence
 - Linkurious

详细参见 JanusGraph 官网：<http://janusgraph.org/>

3.1.2 JanusGraph 与 Tinkerpop 的关系

想要深入了解 JanusGraph，必须了解 Tinkerpop。Tinkerpop 是 Apache 基金会下的一个开源的图数据库与图计算框架（OLTP 与 OLAP），JanusGraph 与 Tinkerpop 的关系为 JanusGraph 是基于 Tinkerpop 这个框架来开发的。

Tinkerpop 有个组件叫 Gremlin，它是一门用于图操作和图遍历的语言（也称查询语言）。Gremlin Console 和 Gremlin Server 分别提供了控制台和远程执行 Gremlin 查询语言的方式。Gremlin Server 在 JanusGraph 中被成为 JanusGraph Server。

Tinkerpop 这个图数据库与图计算框架被很多厂商采用，比如百度开源的 HugeGraph，华为的图引擎服务 GES 等。

详细参见 Tinkerpop 官网：<http://tinkerpop.apache.org/>

3.1.3 JanusGraph 与 Neo4j 的对比

相同点

Neo4j 和 JanusGraph 都是非常优秀的图数据库，都支持 ACID，都是基于 Java 语言开发等等。

不同点

[1] 市场占比

Neo4j 目前图数据库市场份额排名第一的图数据库，占比达 45%。

JanusGraph 是图数据库中的一颗新星，发展速度很快，目前已经从 2017 年第 13 名上升到第 7 名，占比 1.26%。

排名参考：<https://db-engines.com/en/ranking/graph+dbms>

[2] 技术特点

Neo4j 为用户提供了一站式服务，生态成熟、工具齐全，而 JanusGraph 由于其灵活的架构，对后端存储和索引提供了很多选择，需要用户自己组合。从某种程度上来说，使用 JanusGraph 对团队的技术能力要求更高。

[3] 查询语言

Neo4j 使用 Cypher 查询语言操作和遍历图数据，而 JanusGraph 用 Gremlin 语言。

[4] 开源情况

Neo4j 区分社区版和企业版，社区版开源，企业版闭源。而 JanusGraph 无版本区分，均开源。

[5] 分布式

Neo4j 支持高可用集群，但不支持分布式。JanusGraph 是一个分布式的图数据库。

3.2 JanusGraph 架构

JanusGraph 是一个图数据库引擎。JanusGraph 本身专注在图数据的序列化、丰富图数据建模和高效查询的执行。此外，JanusGraph 使用 Hadoop 来做图分析和批量图数据处理。JanusGraph 为数据的持久化、数据索引和客户端访问实现了健壮的，模块化的接口。JanusGraph 的模块化架构使得它可以广泛地与各类存储、索引和客户端技术交互操作。它也简化了 JanusGraph 扩展支持新系统的过程。

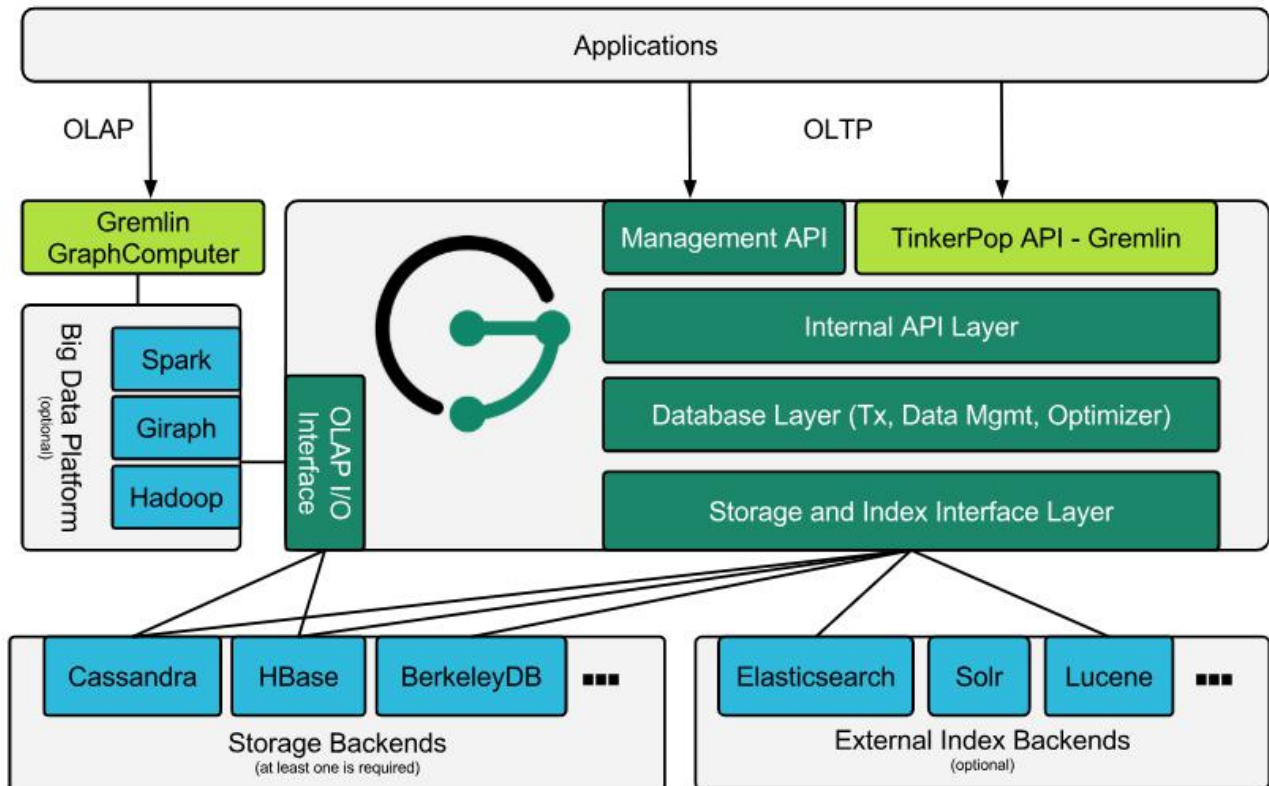
JanusGraph 支持多个存储和索引适配器，目前它支持如下适配器，但 JanusGraph 模块化的架构使得它也容易支持其他第三方适配器。

- 数据存储
 - Apache Cassandra
 - Apache HBase
 - Oracle BerkeleyDB Java Edition
- 索引
 - Elasticsearch
 - Apache Solr
 - Apache Lucene

一般来说，应用程序可以通过如下两种方式与 JanusGraph 交互：

- 嵌入式 JanusGraph：它与执行 Gremlin 查询语言的应用程序运行在同一个 JVM 中。查询执行、JanusGraph 图缓存和事务处理都发生在同一个 JVM 中，但后端的数据存储可以是本地也可以在远程。
- JanusGraph 服务器：通过提交 Gremlin 语言到 JanusGraph Server 来交互。

JanusGraph 的架构图如下所示：



JanusGraph 架构图

JanusGraph 主要包含四个部分：

- 后端存储：负责 JanusGraph 图数据的存储
- 外部索引：负责 JanusGraph 的外部索引，是可选的
- OLTP：负责图数据库相关接口及 API 操作
- OLAP：负责图计算相关接口及与其他计算框架的集成

从上面的图可以看出，JanusGraph 的架构非常灵活，对于各种第三方组件支持自由组合，可根据自己当前所在的业务需要和技术平台来选择适合自己的技术架构。

3.3 JanusGraph 安装

3.3.1 JanusGraph 安装说明

JanusGraph 的源代码是用 Java 开发的，因此需要安装 JDK。推荐使用 Oracle JDK1.8，这里就不介绍了。

JanusGraph 由于其灵活的架构，因此部署方式也有很多种。采用不同的后端存储和索引框架，部署方式就不同。最常见的部署模式有三种：

- JanusGraph+Berkeley+ES：主要用于测试环境
- JanusGraph+Cassandra+ES：适用于无 Hadoop 环境
- JanusGraph+HBase+ES：适用于与 Hadoop 集成的环境

另外, JanusGraph 的安装也可以只配置后端存储框架, 不配置外部索引框架。JanusGraph 支持两种索引, 即 Composite Index 和 Mixed Index, 后端存储默认支持 Composite Index, 如果不配置外部索引框架, 那么就不支持 Mixed Index。

3.3.2 TinkerGraph 内存图数据库

Tinkerpop 作为一个图数据库与图计算框架, 定义了一套完整的 API, 它自身也把这套 API 做了一个名为 TinkerGraph 的简单实现。Tinkerpop 是基于内存的图数据库, JanusGraph 在集成 Tinkerpop 的时候也保留了 TinkerGraph。

如果你只是做一些 Gremlin 语言的基本练习, 不需要存储练习的图数据, 那么就可以采用它。TinkerGraph 在启动 Gremlin Console 时会默认加载。

```
gremlin> graph = JanusGraphFactory.open('inmemory')
==>standardjanusgraph[inmemory:[127.0.0.1]]
gremlin>
或者
graph = JanusGraphFactory.build().set("storage.backend","inmemory").open()
```

上面和下面的区别在哪里? 上面的代码来自 Gremlin 书

答案: 可以通过 graph.features() 来查看区别。

另外, Tinkerpop 内置了一个测试数据, 可以通过如下方式加载:

```
gremlin> graph = TinkerFactory.createModern()
==>tinkergraph[vertices:6 edges:6]
gremlin> g = graph.traversal()
==>graphtraversalsource[tinkergraph[vertices:6 edges:6], standard]
gremlin>
gremlin> g.V().count()
==>6
gremlin> g.V().valueMap(true)
==>[id:1,name:[marko],age:[29],label:person]
==>[id:2,name:[vadas],age:[27],label:person]
==>[id:3,name:[lop],lang:[java],label:software]
==>[id:4,name:[josh],age:[32],label:person]
==>[id:5,name:[ripple],lang:[java],label:software]
==>[id:6,name:[peter],age:[35],label:person]
gremlin>
```

3.3.3 Windows 下 JanusGraph+Berkeley+ES 安装

前置条件: Oracle JDK8 已安装好! 考虑到大部分朋友使用的是 Windows 作为开发环境, 这里以 Windows 下安装为例。

3.3.3.1 版本选择

JanusGraph 当前(2018-12-3)的最新版本为 0.3.1。与之兼容的 BerkeleyJE 版本为 7.4.5, Elasticsearch 版本为 6.0.1, 这两个版本已经包含在安装包里了。

3.3.3.2 安装包下载

JanusGraph 0.3.1 到这里下载 <https://github.com/JanusGraph/janusgraph/releases/>。大约 261M。

3.3.3.3 安装与配置

第一步: 解压 JanusGraph

将 JanusGraph 安装包解压到某个目录, 例如: H:\ssdgreen\janusgraph-0.3.1-hadoop2

第二步: 下载 Hadoop 的本地工具

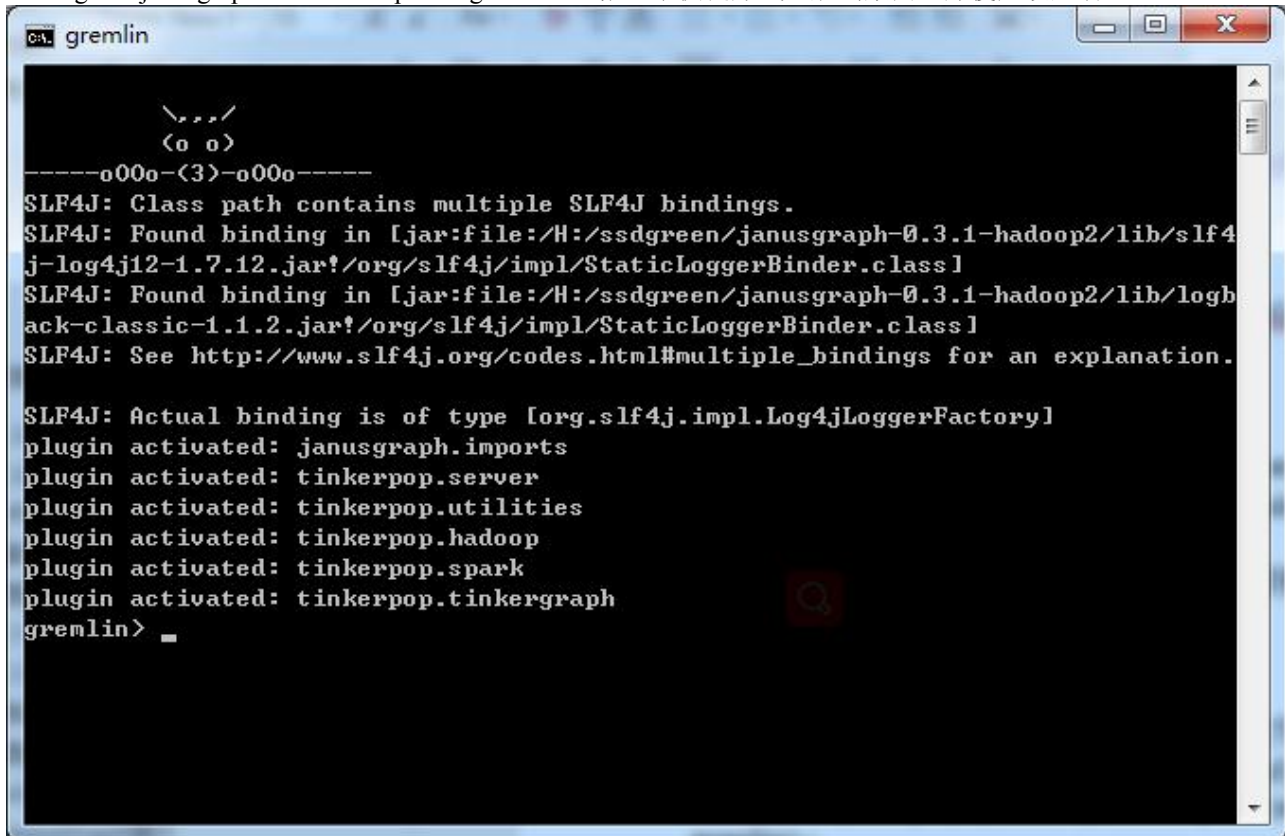
下载 <http://public-repo-1.hortonworks.com/hdp-win-alpha/winutils.exe>, 将其放入 janusgraph-0.3.1-hadoop2\bin 目录下。

第三步: 启动 ElasticSearch

JanusGraph 自带了 ElasticSearch, 进入 JanusGraph 解压目录的 elasticsearch\bin 下面, 直接双击 elasticsearch.bat 即可启动 ElasticSearch。

第四步: 启动 JanusGraph 控制台

找到 JanusGraph 解压目录 bin 下的 gremlin.bat, 右键, 选择“以管理员身份”运行。也可以将 H:\ssdgreen\janusgraph-0.3.1-hadoop2\bin\gremlin.bat 做一个快捷方式, 放到桌面, 方便后续启动。



```
gremlin

  \_ _ _/
  < o o >
-----o00o-(3)-o00o-----
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/H:/ssdgreen/janusgraph-0.3.1-hadoop2/lib/slf4j-log4j12-1.7.12.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/H:/ssdgreen/janusgraph-0.3.1-hadoop2/lib/logback-classic-1.1.2.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.

SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
plugin activated: janusgraph.imports
plugin activated: tinkrpop.server
plugin activated: tinkrpop.utilities
plugin activated: tinkrpop.hadoop
plugin activated: tinkrpop.spark
plugin activated: tinkrpop.tinkergraph
gremlin> _
```

注意:

- [1] 运行的时候, 需要以管理员身份运行, 需要注册一些信息, 后续直接双击运行即可。
- [2] 除了控制台之外, 还可以通过 gremlin-server.bat 启动 JanusGraph Server 接收远程访问。

3.3.3.4 基本使用

在 Gremlin 控制台模式下, 以 berkeley+es 的方式开启 Graph 实例, 如下:

```
gremlin> graph = JanusGraphFactory.open('conf/janusgraph-berkeleyje-es.properties')
==>standardjanusgraph[berkeleyje:H:\ssdgreen\janusgraph-0.3.1-hadoop2\conf\../db/berkeley]
gremlin>
```

JanusGraph 自带了一个很有名的图数据案例, 叫做“The Graph of the Gods”, 也称罗马诸神。JanusGraphFactory 提供了静态的方法来加载图数据。

```
gremlin> GraphOfTheGodsFactory.load(graph)
==>null
```

获取图遍历句柄

```
gremlin> g = graph.traversal()
==>graphtraversalsource[standardjanusgraph[berkeleyje:H:\ssdgreen\janusgraph-0.3.1-hadoop2\conf\../db/berkeley], standard]
```

通过图遍历查询罗马神话中指农神萨图努斯 saturn 的信息

```
gremlin> g.V().has('name', 'saturn').valueMap()
==>[name:[saturn],age:[10000]]
```


创建一个顶点（也称节点）

```
gremlin> g.addV('person').property('name','Dennis')
==>v[4176]
```

查询刚创建的顶点

```
gremlin> g.V().has('name', 'Dennis').valueMap()
==>[name:[Dennis]]
gremlin>
```

3.4 Linux 下 JanusGraph 的安装

3.4.1 Linux 下 JanusGraph 的安装说明

Linux 下 JanusGraph 与 Windows 下的安装有两点不同：

- [1] ElasticSearch 因为是压缩包的方式，只能以非 root 用户启动，而 Windows 下可以用管理员启动；如果是 rpm 包安装，可以用 root 启动；
- [2] Linux 下 JanusGraph 自带了一个 JanusGraph Server 的配置和脚本，可以直接启动 JanusGraph Server；

3.4.2 Linux 下 JanusGraph 的安装步骤

说明：这里假设你已经阅读了 Windows 下的安装，并已下载好安装包，了解了 JanusGraph 的基本部署模式。

- [1] 将压缩包上传到用户根目录下并解压

注意：这里假设用户名为 dennis，不能用 root，前面已说明。

解压

```
[dennis@hadoop03 ~]$ cd /opt/
[dennis@hadoop03 opt]$ sudo unzip ~/janusgraph-0.3.1-hadoop2.zip
```

- [2] 修改权限

修改安装包的权限，以便 dennis 用户能够访问/opt 下的 janusgraph 包

```
[dennis@hadoop03 opt]$ sudo chown -R dennis:dennis janusgraph-0.3.1-hadoop2
```

3.4.3 JanusGraph 的启动

本文采用的是 JanusGraph+Berkeley+ES 的部署模式，也就是说后端存储采用 BerkeleyDB、外部索引采用 ElasticSearch。因此，BerkeleyDB 是嵌入式的，不需要单独启动，但 ElasticSearch 需要在 JanusGraph 之前启动。

3.4.3.1 启动 ElasticSearch

JanusGraph 自带了 ElasticSearch 的安装包，先进入该目录

```
[dennis@hadoop03 ~]$ cd /opt/janusgraph-0.3.1-hadoop2/elasticsearch/
```

加上&以便在后台启动

```
[dennis@hadoop03 elasticsearch]$ bin/elasticsearch &
```

3.4.3.2 JanusGraph 的基本使用

JanusGraph 的使用方式通常包括：

- [1] 以嵌入式开发(Java)的方式访问；
- [2] 通过 Gremlin Console 控制台访问；
- [3] 通过 JanusGraph Server 的方式访问；

这里先只介绍 Gremlin Console 的方式，其他方式将在后面陆续介绍。

JanusGraph Gremlin Console 的使用

- [1] 启动 Gremlin Console

```
[dennis@hadoop03 ~]$ cd /opt/janusgraph-0.3.1-hadoop2/
```

```
[dennis@hadoop03 janusgraph-0.3.1-hadoop2]$ bin/gremlin.sh
```

[2] 开启一个图数据库实例

```
gremlin> graph = JanusGraphFactory.open('conf/janusgraph-berkeleyje-es.properties')
==>standardjanusgraph[berkeleyje:/opt/janusgraph-0.3.1-hadoop2/conf/./db/berkeley]
gremlin>
```

JanusGraph 默认有很多配置，这里采用文前提到的配置模式。

[3] 获取图遍历句柄

```
gremlin> g = graph.traversal()
==>graphtraversalsource[standardjanusgraph[berkeleyje:/opt/janusgraph-0.3.1-hadoop2/conf/./db/berkeley],
standard]
gremlin>
```

[4] 通过图遍历句柄来进行各种图操作

新增一个顶点(vertex)

```
gremlin> g.addV('person').property('name','Dennis')
==>v[4104]
```

查询刚刚创建的顶点

```
gremlin> g.V().has('name', 'Dennis').values()
```

4 JanusGraph 建模

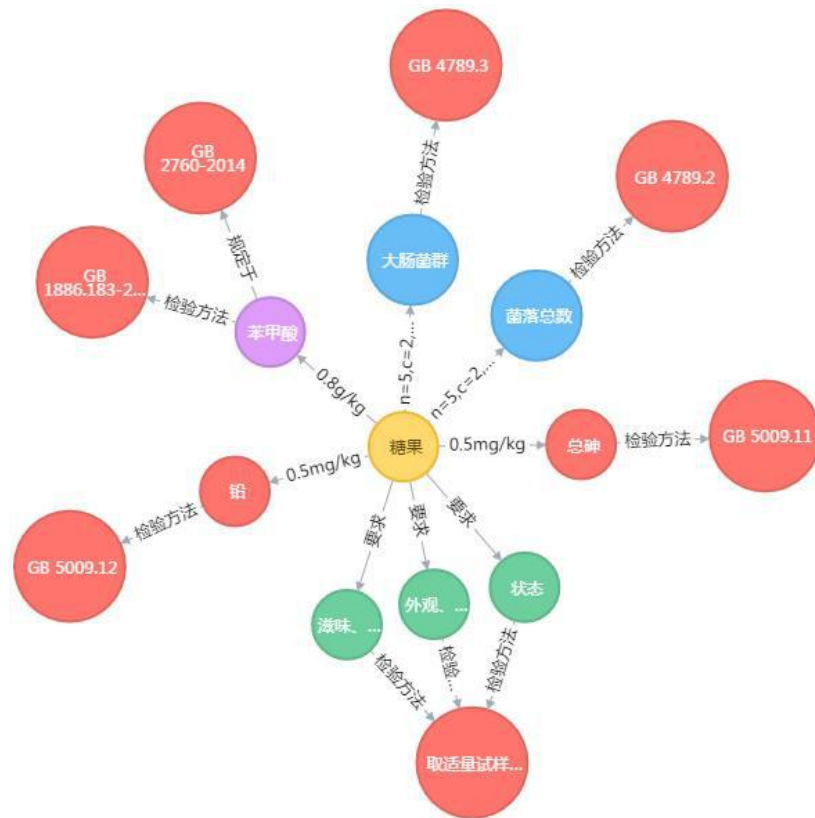
4.1 图数据库建模

4.1.1 图数据库的基本模型

图数据库的基本模型如下：

- 顶点：用于表达实体
- 边：用于表达两个实体之间的关系
- 标签：分为顶点标签和边标签。顶点标签是顶点的类型，边标签是实体之间关系的类型。
- 属性：用于丰富顶点和边的信息。

一个简易的图模式示例如下：



4.2 定义 JanusGraph Schema

JanusGraph 的每一个图都有特定的模型（Schema），主要包括顶点标签（VertexLabel）、边标签（EdgeLabel）和属性键（PropertyKey）。Schema 可以显式地或者隐式定义。推荐显式地定义图的 Schema，这样可以增强图的健壮性。

所谓显式地定义就是事先定义好图的 Schema，隐式定义就是直接创建图数据，这时图的 Schema 会自动根据数据中的模型来创建。

4.2.1 顶点标签

4.2.1.1 顶点标签的概念

顶点的标签的概念有部分类似于关系型数据中的表名，可以为顶点分类。一个顶点可以有多个标签，也可以没有标签。比如一个表达人的顶点，他可以既是一名学生，又同时是一名志愿者。

注意：Neo4j 支持多标签，JanusGraph 只支持一个标签，可以通过属性来补充。

尽管顶点的标签是可选的，但是 JanusGraph 内部会默认给所有的顶点分配一个默认标签(vertex)。顶点的标签名在图中必须是唯一的。

4.2.1.2 顶点标签的创建

下面的命令创建了一个名为'person'的标签。

```
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@656c5c3
gremlin> person = mgmt.makeVertexLabel('person').make()
==>person
gremlin> mgmt.commit()
==>null
```

4.2.1.3 顶点标签的查看

查看指定的顶点标签

```
gremlin> mgmt = graph.openManagement();
```

```
==>org.janusgraph.graphdb.database.management.ManagementSystem@634e1b39
gremlin> person = mgmt.getVertexLabel('person')
==>person
```

查看所有的顶点标签

```
gremlin> labels = mgmt.getVertexLabels()
==>person
```

4.2.1.4 顶点标签的修改

下面的命令将顶点标签 `person` 的名字改为 `person1`。

```
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@335f5c69
gremlin> person = mgmt.getVertexLabel('person')
==>person
gremlin> mgmt.changeName(person, 'person1')
==>null
gremlin> mgmt.commit()
==>null
```

获取修改后的顶点标签

```
gremlin> mgmt = graph.openManagement();
==>org.janusgraph.graphdb.database.management.ManagementSystem@74c9e11
gremlin> person = mgmt.getVertexLabel('person')
==>null
gremlin> person1 = mgmt.getVertexLabel('person1')
==>person1
```

根据返回值可以看出，获取顶点标签 `person` 返回 `null`，而 `person1` 返回了结果。

4.2.1.5 顶点标签的删除

删除顶点标签 `person1`

```
gremlin> mgmt.getVertexLabel("person1").remove();
==>null
gremlin> mgmt.commit()
==>null
```

4.2.1.6 删除整个 graph

删除整个图，包含 `graph`、`data`、`schema` 等

```
gremlin> JanusGraphFactory.drop(graph);
==>null
```

警告：这一步需要谨慎操作，一旦操作之后无法恢复。

4.2.2 边标签

4.2.2.1 边标签的概念

每一条连接两个顶点之间的边都有一个标签，用于定义顶点之间关系的语义。例如 `friend` 标签表达了两个顶点的个体之间的朋友关系。

边标签在图中也必须是唯一的。

每个边标签有一个多样性（multiplicity），它可以约束一对顶点之间的边数量及方向等。

边标签的多样性

- MULTI: 允许一对顶点之间有任何多个相同标签的边。换句话说就是没有约束。
- SIMPLE: 允许最多一个相同标签的边。
- MANY2ONE: 该边标签只允许有一条外向的边，但对向内的边的条数没有限制。
- ONE2MANY: 该边标签只允许有一条向内的边，但对外向的边的条数没有限制。
- ONE2ONE: 该边标签只允许一条向内和一条向外的边。

边标签默认的多多样性为 MULTI。

4.2.2.2 边标签的创建

```
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@1ec88aa1
```

```
gremlin> follow = mgmt.makeEdgeLabel('follow').multiplicity(MULTI).make()
==>follow
```

```
gremlin> mgmt.makeEdgeLabel('mother').multiplicity(MANY2ONE).make()
==>mother
```

```
gremlin> mgmt.commit()
==>null
```

4.2.2.3 边标签的查看

查看指定的标签

```
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@32ea16b7
gremlin> mgmt.getEdgeLabel("follow");
==>follow
```

查看所有标签

```
gremlin> mgmt.getRelationTypes(EdgeLabel.class)
==>follow
==>mother
```

4.2.2.4 边标签的删除

```
gremlin> mgmt.getEdgeLabel("follow").remove()
==>null
gremlin> mgmt.commit()
==>null
```

4.2.3 属性键

顶点和边都可以有属性。属性是以键值对的方式存在。比如属性 name="Dennis" 的 key 为 name，值为 "Dennis"。

属性键可以约束数据的类型和基数（cardinality）。

4.2.3.1 属性键的类型

JanusGraph 原生支持的数据类型表

属性键的类型	类型描述
String	字符串序列
Character	单个字符
Boolean	布尔类型：true 或者 false
Byte	字节
Short	短整型
Integer	整型
Long	长整型
Float	浮点型
Double	双精度型

Date	时间类型, java.util.Date 的实例
GeoShape	地理的 shape 类型, 如 point、circle 或者 box
UUID	全局唯一标识符 (java.util.UUID)

4.2.3.2 属性的基数

- SINGLE: 只允许有一个单值
- LIST: 值可以是一个列表
- SET: 值可以有多个, 但不能重复

4.2.3.3 属性键的创建

```
gremlin> name = mgmt.makePropertyKey('name').dataType(String.class).cardinality(Cardinality.SINGLE).make()
==>name
gremlin> mgmt.commit()
==>null
```

4.2.3.4 属性键的查看

查看指定的属性键

```
gremlin> mgmt.getPropertyKey('name')
==>name
```

4.2.3.5 属性键的修改

下面的命令将 name 属性改为 firstname

```
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@6cbb175
gremlin> name = mgmt.getPropertyKey('name')
==>name
gremlin> mgmt.changeName(name, 'firstname')
==>null
gremlin> mgmt.commit()
==>null
```

4.2.3.6 属性键的删除

```
gremlin> mgmt = graph.openManagement();
==>org.janusgraph.graphdb.database.management.ManagementSystem@4d464510
gremlin>
gremlin> mgmt.getPropertyKey('name').remove()
==>null
gremlin> mgmt.commit()
==>null
```

4.3 自动创建 Schema

JanusGraph 默认支持自动模型创建, 它是通过 DefaultSchemaMaker 来实现的。默认情况下, 边标签可以有多个, 属性键的基数为 SINGLE, 数据类型为 Object.class。用户也可以自己去实现和注册 DefaultSchemaMaker 来控制自动创建模型。

为了保证图数据的一致性, 强烈建议关闭自动创建模型, 并显式地定义所有的模型元素。关闭方法为在配置文件中加入 schema.default=none。

4.3.1 模型快速查看

```
g.V().label().dedup()
g.E().label().dedup()
```

查看顶点的所有属性

```
gremlin> g.V().properties().key().dedup()
```

查看边的所有属性

```
gremlin> g.E().properties().key().dedup()
```

4.4 ID 管理

在 JanusGraph 中，每个顶点、每条边、每个标签以及每个属性都有唯一的 ID。ID 唯一标识这个对象。

获取 id

```
gremlin> g.V().has('code','DFW').id()  
==>v[4128]
```

```
gremlin> g.V().hasId(4128)  
==>v[4128]
```

或者

```
gremlin> g.V().has(id, 4128)  
==>v[4128]
```

```
g.V().hasId(between(1,6))
```

还可以

```
g.V(3)  
g.V(3,6,8,15)
```

```
a=[3,6,8,15]  
g.V(a).values('code')
```

5 Gremlin 查询语言

<http://tinkerpop.apache.org/javadocs/3.3.3/core/>

<http://tinkerpop.apache.org/docs/current/reference/#traversal>

<https://static.javadoc.io/org.janusgraph/janusgraph-core/0.3.0/overview-summary.html>

<http://tinkerpop.apache.org/javadocs/3.2.3/core/overview-summary.html>

<http://tinkerpop.apache.org/javadocs/3.2.3/core/org/apache/tinkerpop/gremlin/process/traversal/dsl/graph/GraphTraversalSource.html>

<http://tinkerpop.apache.org/javadocs/3.2.3/core/org/apache/tinkerpop/gremlin/process/traversal/dsl/graph/GraphTraversal.html>

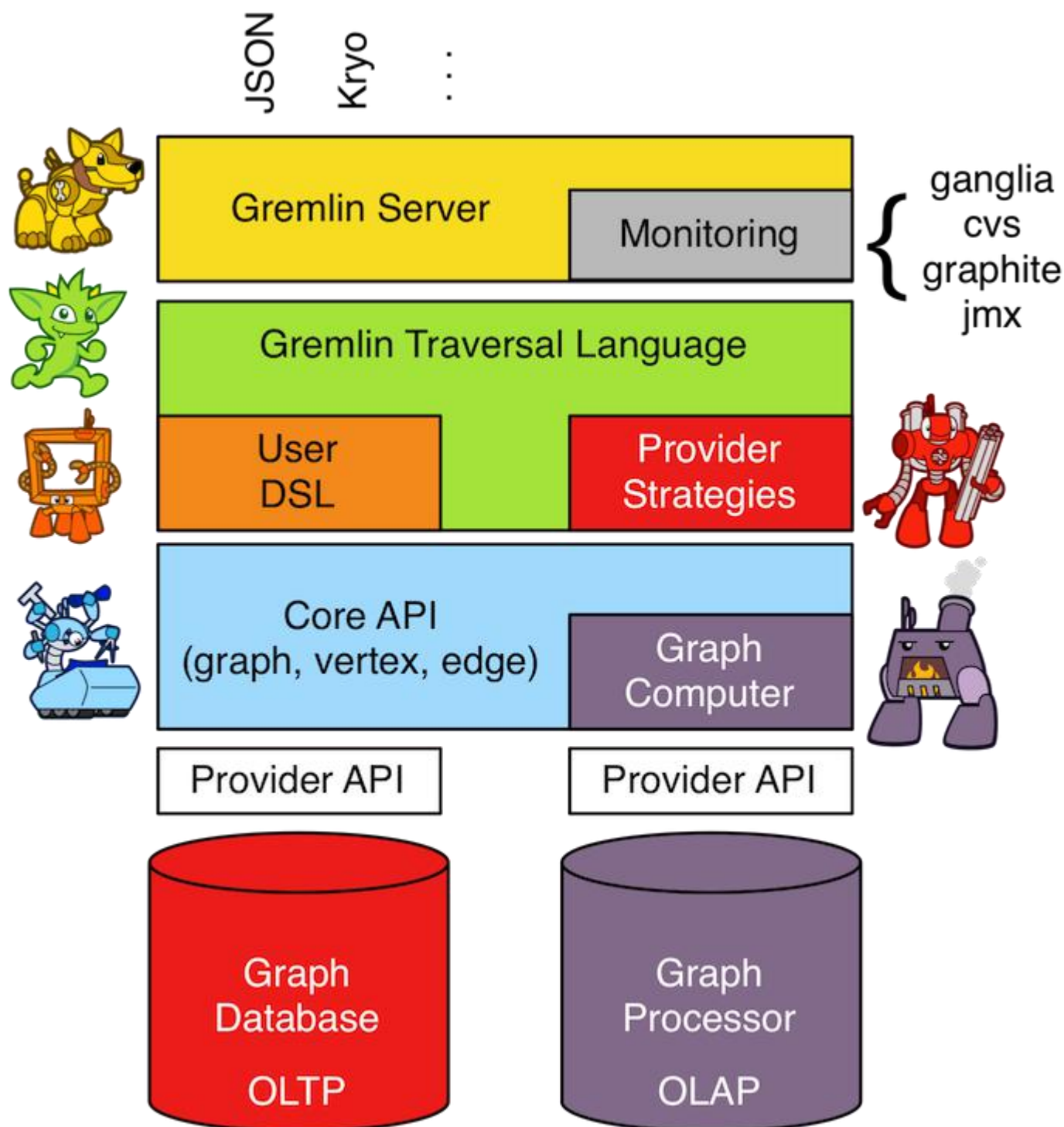
5.1 Gremlin 概述

参考: <http://groovy-lang.org>

Gremlin 是 JanusGraph 的查询语言，用于检索和修改图数据。Gremlin 是一门面向路径(path-oriented)的语言，它可以很简洁地表达复杂的图遍历。Gremlin 也是一门函数式语言，通过遍历算子按路径的方式链接在一起来构成像路径一样的表达式。

Gremlin 其实是 Apache Tinkerpop 的一个组件，它是独立于 JanusGraph 的开发，而且在很多其他图数据库中也支持 Gremlin。

Tinkerpop3 提供了两套 API 操作：Graph Structure 和 Graph Process。它们之间的关系可以借助如下图来表示：



图系统集成

其中，

- 图结构 Graph Structure: 图结构 API 也称为 “Core TinkerPop3 API”，它定义了图的数据模型，包括 Graph、Vertex、Edge 以及事务等。
- 图处理 Graph Process: 图处理 API 构建于图结构 API 之上，提供了更灵活的遍历方式。

图结构通常以 **graph** 表示，图处理以 **g** 来表达。建议只用 **graph** 来做图的模型定义及数据库管理相关操作。图的数据操作，包括创建、更新、删除及便利都用 **g** 来操作。

5.2 Graph Structure API

- Graph: 维护了顶点和边的集合以及事务相关
- Element: 维护了一个属性列表和一个元素类型的标签
 - Vertex: 扩展了 Element, 并维护了进和出的边集合
 - Edge: 扩展了 Element, 并维护了一个进和出的顶点
- Property<V>: 字符串键及 V 值
 - VertexProperty<V>: a string key associated with a V value as well as a collection of Property<U> properties (vertices only)

5.2.1 创建

```
graph.addVertex(T.label,'student', 'name', 'Dennis')
```

student 是标签, name 为属性的 key, Dennis 为属性的 value。

不推荐这么用, 原因是 graph API 并不总是对用户可见。推荐使用 g.addV() 的 API。

5.2.2 其他

5.2.2.1 查看当前图

```
gremlin> graph.toString()
==>standardjanusgraph[berkeleyje:H:\ssdgreen\janusgraph-0.3.1-hadoop2\conf\../db
/berkeley]
gremlin>
```

5.2.2.2 查看图的 features

```
gremlin> graph.features()
==>FEATURES
> GraphFeatures
>-- Transactions: true
>-- Computer: true
>-- ConcurrentAccess: true
>-- ThreadedTransactions: true
>-- Persistence: true
> VariableFeatures
>-- Variables: true
>-- MapValues: true
>-- MixedListValues: false
>-- SerializableValues: false
>-- UniformListValues: false
>-- BooleanValues: true
>-- ByteValues: true
>-- DoubleValues: true
>-- FloatValues: true
>-- IntegerValues: true
>-- LongValues: true
>-- BooleanArrayValues: true
>-- ByteArrayValues: true
>-- DoubleArrayValues: true
>-- FloatArrayValues: true
>-- IntegerArrayValues: true
>-- StringArrayValues: true
>-- LongArrayValues: true
>-- StringValues: true
> VertexFeatures
>-- AddVertices: true
>-- RemoveVertices: true
>-- MultiProperties: true
>-- DuplicateMultiProperties: true
>-- MetaProperties: true
>-- UuidIds: false
```

```
>-- AnyIds: false
>-- AddProperty: true
>-- RemoveProperty: true
>-- NumericIds: true
>-- StringIds: false
>-- CustomIds: false
>-- UserSuppliedIds: false
> VertexPropertyFeatures
>-- UuidIds: false
>-- AnyIds: false
>-- RemoveProperty: true
>-- NumericIds: false
>-- StringIds: true
>-- CustomIds: true
>-- UserSuppliedIds: false
>-- Properties: true
>-- MapValues: true
>-- MixedListValues: false
>-- SerializableValues: false
>-- UniformListValues: false
>-- BooleanValues: true
>-- ByteValues: true
>-- DoubleValues: true
>-- FloatValues: true
>-- IntegerValues: true
>-- LongValues: true
>-- BooleanArrayValues: true
>-- ByteArrayValues: true
>-- DoubleArrayValues: true
>-- FloatArrayValues: true
>-- IntegerArrayValues: true
>-- StringArrayValues: true
>-- LongArrayValues: true
>-- StringValues: true
> EdgeFeatures
>-- AddEdges: true
>-- RemoveEdges: true
>-- UuidIds: false
>-- AnyIds: false
>-- AddProperty: true
>-- RemoveProperty: true
>-- NumericIds: false
>-- StringIds: false
>-- CustomIds: true
>-- UserSuppliedIds: false
> EdgePropertyFeatures
>-- Properties: true
>-- MapValues: true
>-- MixedListValues: false
>-- SerializableValues: false
>-- UniformListValues: false
>-- BooleanValues: true
>-- ByteValues: true
>-- DoubleValues: true
>-- FloatValues: true
>-- IntegerValues: true
>-- LongValues: true
>-- BooleanArrayValues: true
>-- ByteArrayValues: true
>-- DoubleArrayValues: true
```



```
>-- FloatArrayValues: true
>-- IntegerArrayValues: true
>-- StringArrayValues: true
>-- LongArrayValues: true
>-- StringValues: true
```

```
gremlin>
```

5.2.2.3 给图添加变量

```
gremlin> graph.variables().set('creator','Dennis')
==>null
gremlin> graph.variables().keys()
==>creator
gremlin> graph.variables().asMap()
==>creator=Dennis
gremlin> graph.variables().get('creator')
==>Optional[Dennis]
```

删除变量

```
gremlin> graph.variables().remove('creator')
==>null
gremlin> graph.variables().asMap()
gremlin>
```

5.3 Graph Process API

- TraversalSource: 特定图的遍历器, 领域特定语言的 执行引擎
 - Traversal<S,E>: 将 S 对象转为 E 对象的数据流函数
- GraphComputer: 以并行及跨越多机集群方式处理图的系统, 即图计算
 - VertexProgram: 以逻辑并行方式在所有顶点执行的代码, 通过消息传递进行相互通信
 - MapReduce: 并行地分析图中的所有顶点, 并产生一个简化的结果的处理框架

5.4 Gremlin console

5.4.1 Gremlin 帮助

在 gremlin 控制台, 输入':help'、':h'或者':?'都可以查看帮助。

查看帮助

```
gremlin> ?
```

查看版本

```
gremlin> Gremlin.version
==>3.3.3
```

查看已导入的包

```
gremlin> :show imports
```

关闭日志输出

```
gremlin> g.V().count();[]
```

在语句末尾加上";[]"即可关闭额外的输出。

如果执行结果返回"==>null", 则表示调用了一个返回 void 的方法, 且没有异常发生。

可以执行任何 Groovy 代码, 也可以这样执行: `gremlin -e mycode.groovy`

还可以在 console 中加载 groovy 脚本, 比如:

```
gremlin> :load init.groovy
```

记录会话到日志文件

```
gremlin> :record start mylog.txt
Recording session to: "mylog.txt"
==>mylog.txt
gremlin> 1+1
==>2
gremlin> :record stop
Recording stopped; session saved as: "mylog.txt" (137 bytes)
==>mylog.txt
gremlin> :help
```

5.5 图事务

任何的图操作都会自动开启一个事务。如果事务没有提交，那么操作就不会生效。提交事务的方法很简单，对于 Graph Structure API 和 Graph Process API 分别如下：

5.5.1 提交事务

```
graph.tx().commit()
g.tx().commit()
```

5.5.2 回滚事务

如下：

```
graph.tx().rollback()
g.tx().rollback()
```

5.5.3 查询事务

查看当前正在进行的事务，如下：

```
graph.getOpenTransactions()
```

如果想要结束当前所有的事务，可以用：

```
for (tx in graph.getOpenTransactions()) tx.rollback()
```

5.6 Gremlin 增删改查

5.6.1 创建顶点

5.6.1.1 创建无标签的顶点

创建顶点的时候，标签是可选的。例如：创建一个没有任何标识的顶点。

```
gremlin> g.addV()
==>v[4336]
```

返回的是创建成功的顶点的 id。

5.6.1.2 创建带标签的顶点

```
gremlin> g.addV("person")
==>v[8432]
```

注：标签一旦创建之后不可更改。

5.6.1.3 创建带标签和属性的顶点

```
gremlin> g.addV("person").property('name','Dennis')
==>v[8392]
```

同时添加多个属性

```
gremlin> g.addV("person").property('name','Dennis').property('city','Chengdu')
```

```
==>v[4208]
```

5.6.2 查看顶点

5.6.2.1 查看所有顶点

```
gremlin> g.V()
18:42:45 WARN org.janusgraph.graphdb.transaction.StandardJanusGraphTx - Query
requires iterating over all vertices [()]. For better performance, use indexes
==>v[8392]
==>v[8328]
```

注意：不推荐这样使用。当数据量很大的时候会卡死。

5.6.2.2 根据标签查看顶点

查看所有 person 顶点

```
gremlin> g.V().hasLabel('person')
23:15:46 WARN org.janusgraph.graphdb.transaction.StandardJanusGraphTx - Query
requires iterating over all vertices [(~label = person)]. For better performance
, use indexes
==>v[4320]
gremlin>
```

注：不推荐这样使用，尤其是数据量很大的情况。

5.6.2.3 根据属性查看顶点

```
g.V().has('code','DFW')
```

变体

```
g.V().hasNot('region') 等价于 g.V().not(has('region'))
```

5.6.2.4 根据 id 查看顶点

```
gremlin> g.V(4208).valueMap()
==>[city:[Chengdu],name:[Dennis]]
```

5.6.2.5 查看顶点的标签

```
gremlin> g.V().has('name','Dennis').label()
23:51:58 WARN org.janusgraph.graphdb.transaction.StandardJanusGraphTx - Query requires iterating over all
vertices [(name = Dennis)]. For better performance, use indexes
==>person
```

5.6.2.6 查看顶点的属性键值对

```
gremlin> g.V().has('name','Dennis').valueMap()
==>[name:[Dennis]]
```

只查看属性值

```
gremlin> g.V().has('name','Dennis').values()
==>Dennis
```

5.6.2.7 查看顶点的所有信息

除了包含顶点的属性键值对之外，还包含 id 和标签

```
gremlin> g.V().valueMap(true)
==>[id:40972336,name:[Dennis],label:person]
gremlin>
```

查看数据库中所有顶点的标签

```
g.V().label()
```

5.6.2.8 根据标签和属性一起查

`g.V().hasLabel('airport').has('code','DFW')`和 `g.V().has('airport','code','DFW')`等价

5.6.3 创建边

5.6.3.1 创建边

创建边的时候，边的标签是必须的。

```
gremlin> x02 = g.V().has('name', 'xiaoming02').id()
==>4184
gremlin> x01 = g.V().has('name', 'xiaoming01').id()
==>40964152
gremlin> g.addEdge("coll").from(g.V(40964152)).to(g.V(4184))
==>e[odyx3-oe054-6nf9-388][40964152-coll->4184]
gremlin>
```

说明："coll"为边标签，表示同事关系。

5.6.3.2 查看边

```
g.V().has('code','MIA').outE().as('e').inV().has('code','DFW').select('e')
```

5.6.4 顶点属性

values 值看属性值，valueMap 看属性键值对，valueMap(true)加上 id 和 label。

还可以这样：g.V().has('code','AUS').valueMap(true,'region')

类似于 select *,valueMap()中可以添加你想查看到的字段名，不添加就默认查看所有。也可以这样，更清晰：g.V().has('code','AUS').valueMap().select('code','icao','desc')

5.6.4.1 添加或更新属性

先通过 id 查询到该顶点，然后用 property 添加或者更新属性。

```
gremlin> g.V(4120).property('name', 'Mike')
==>v[4120]
```

说明：如果 name 属性存在就更新，不存在就添加改属性。

5.6.4.2 查看所有属性值

```
gremlin> g.V().has('airport','code','DFW').values()
```

5.6.4.3 查看单个属性值

```
gremlin> g.V().has('airport','code','DFW').values('city')
```

5.6.4.4 查看多个属性值

```
gremlin> g.V().has('airport','code','DFW').values('runways','icao')
```

5.6.4.5 删除某个属性

```
gremlin> g.V(4328).properties('name').drop()
```

5.6.5 边属性

注：边属性的操作基本跟顶点属性的操作类似。

5.6.5.1 添加/更新边属性

```
gremlin> g.E('4w74-35s-4r9-394').property('updated', '2019-01-10')
==>e[4w74-35s-4r9-394][4096-followedBy->4216]
```

5.6.5.2 查看边属性

```
gremlin> g.E('4w74-35s-4r9-394').valueMap(true)
==>[id:4w74-35s-4r9-394,weight:19,label:followedBy]
```

5.6.6 删除

5.6.6.1 删除所有的顶点

```
g.V().drop()
```

或者

```
gremlin> g.V().drop().iterate()
```

5.6.6.2 删除所有的边

```
g.E().drop()
```

5.6.7 退出

:quit 或者 :exit 都可以退出。

5.7 Gremlin 谓词

5.7.1 常规谓词

5.7.1.1 group

分组查看

```
g.V().groupCount().by(label)
```

或者

```
g.V().label().groupCount()
```

或

```
g.V().group().by(label).by(count())
```

5.7.1.2 Limit/tail

5.7.1.3 range

range(10, -1) -1 表示所有剩下的。

5.7.1.4 between

lt 表示 less than

eq

neq

Table 3. Predicates that test values or ranges of values

eq	Equal to
neq	Not equal to

gt	Greater than
gte	Greater than or equal to
lt	Less than
lte	Less than or equal to
inside	Inside a lower and upper bound, neither bound is included.
outside	Outside a lower and upper bound, neither bound is included.
between	Between two values inclusive/exclusive (upper bound is excluded)
within	Must match at least one of the values provided. Can be a range or a list
without	Must not match any of the values provided. Can be a range or a list

5.7.1.5 skip

5.7.1.6 as

就是别名

5.7.1.7 select

通常和 as 结合起来用

```
g.V().has('code','DFW').as('from').out().  
has('region','US-CA').as('to').  
select('from','to')
```

5.7.1.8 project

映射，给结果的列重命名

```
g.V().has('type','airport').limit(10).  
project('IATA','Region','Routes').  
by('code').by('region').by(out().count())
```

多个 as 与 select

```
g.V(1).as('a').V(2).as('a').select(first,'a')  
first 可以换为 last all
```

5.7.1.9 order

```
.order().by('code')
```

参考: 3.12

5.7.1.10 统计 count/sum/max/min/mean

5.7.1.11 and/or/not/is

```
g.V().and(has('code','AUS'),has('icao','KAUS'))  
g.V().or(has('code','AUS'),has('icao','KDFW'))  
g.V().not(hasLabel('airport')).count()  
g.V().is(hasLabel('person')).count()
```

5.7.1.12 where

```
g.V().where(values('runways').is(gt(5)))
```

5.7.1.13 union

identity 没看懂

5.7.2 特有谓词

5.7.2.1 next

获取真正的值，默认每一步查询返回的都是查询本身，以便继续后续的查询。

```
gremlin> g.V().has("name", "Dennis").getClass()  
==>class org.apache.tinkerpop.gremlin.process.traversal.dsl.graph.DefaultGraphTraversal
```

```
gremlin> g.V().has("name", "Dennis").next().getClass()  
==>class org.janusgraph.graphdb.vertices.StandardVertex
```

next()中还可以带数字，表示获取几个值。如 next(3)

It is necessary to add a call to next to the end of the query in order for this to work. Forgetting to add the call to next is a very commonly made mistake by people getting used to the Gremlin query language. The call to next terminates the traversal part of the query and generates a concrete result that can be stored in a variable. There are other steps such as toList and toSet that also

5.7.2.2 toList/toSet/toBulkSet

The difference between a bulkSet and a set is that bulkSet is a so called weighted set. A bulkSet stores every value but includes a count of how many of each type is present.

asBulk()

```
setb= g.V().has('airport','region','US-TX').values('runways').toBulkSet()

// How many unique values are in the set?
setb.uniqueSize()
6

// How many total values are present?
setb.size()
26
```

5.7.2.3 fill

```
a = []
g.V().has('airport','region','US-TX').values('runways').fill(a)
```

5.7.2.4 fold/unfold

the fold step puts all of the results into a list for us
转为 list

unfold 就是分开成一个一个的结果

5.7.2.5 path

After you have done some graph walking using a query you can use path to get a summary back of where you went.

就是查询所经过的路径

5.7.2.6 by

```
g.V().has('airport','code','AUS').out().out().path().by('code')
```

5.7.2.7 dedup

去重

5.7.2.8 repeat

times until

5.7.2.9 join

以某种方式将结果连接起来
比如.toList().join(',')

5.7.2.10 local

```
g.V().has('region','GB-SCT').order().by('code').local(values('code','city').fold())
```

有点像局部做的意思，分别做的意思

5.7.2.11 choose

类似 if else

5.7.2.12 option

类似 switch case

5.7.2.13 match

模式匹配

5.7.2.14 constant

5.7.2.15 sideEffect

The sideEffect step allows you to do some additional processing as part of a query without changing what gets passed on to the next stage of the query.

```
g.V().has('code','SFO').sideEffect{println "I'm working on it"}.values('desc')
```

```
g.V(3).sideEffect(out().count().store('a')).out().out().count().as('b').select('a','b')
```

5.7.2.16 aggregate

5.7.2.17 inject

```
g.inject(1,2,3,4,5).mean()
```

5.7.2.18 coalesce

如果 coalesce 中有值就返回，否则就返回后面部分的值。

```
g.V(1).coalesce(has('region','US-TX').values('desc'),constant("Not in Texas"))
```

5.7.2.19 optional

类似于 coalesce，但是指后者如果有值就返回，否则返回前者。

```
g.V().has('code','AUS').optional(out().has('code','SYD')).values('city')
```

5.7.2.20 both/bothV/bothE/otherV

5.7.2.21 emit

输出中间结果

5.7.2.22 cyclicPath()

循环路径

```
g.V().has('code','AUS').  
out().out().cyclicPath().  
limit(10).path().by('code')
```

5.7.2.23 Gremlin' s scientific calculator math

注意：Apache TinkerPop 3.3.1 及以上版本的才支持这个功能。

Table 4. Scientific calculator operators

+	Arithmetic plus.
-	Arithmetic minus.
*	Arithmetic multiply.
/	Arithmetic divide.
%	Arithmetic modulo (remainder).
^	Raise to the power. (n^x).
abs	Absolute value
acos	Arc (inverse) cosine in radians.
asin	Arc (inverse) sine in radians.

atan	Arc (inverse) tangent in radians.
cbrt	Cube root
ceil	Returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer.
cos	Cosine of angle given in radians.
cosh	Hyperbolic cosine.
exp	Returns Euler's number "e" raised to the given power (e^x)
floor	Returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer.
log	Natural logarithm (base e)
log10	Logarithm (base 10)
log2	Logarithm (base 2)
signum	Returns the <i>signum</i> function of the argument; zero if the argument is zero, 1.0 if the argument is greater than zero, -1.0 if the argument is less than zero.
sin	Sine of angle given in radians.
sinh	Hyperbolic sine
sqrt	Square root
tan	Tangent of angle given in radians.
tanh	Hyperbolic tangent.

```
gremlin> g.inject(12).math('_/3')
==>4.0
```

用下划线引用前面的结果。

5.7.2.24 Lambda 函数

5.7.2.25 filter

5.7.2.26 sack

5.7.2.27 store

5.8 其他搜索谓词

Table 15. Additional JanusGraph text search predicates

textContains	True if a whole word matches the search string provided.
textContainsPrefix	True if at least one word starts with the search string provided.
textContainsRegex	True if at least one word matches the regular expression provided.
textContainsFuzzy	True if a word matches the fuzzy search text provided.
textPrefix	True if the string being inspected starts with the search text.
textRegex	True if the string being inspected matches the regular expression provided.
textFuzzy	True if the string being inspected matches the fuzzy search text.

5.9 Gremlin 语言变体

以如何使用 Python 版为例

<https://github.com/Microsoft/spring-data-gremlin>
spring.data.gremlin

6 JanusGraph 索引

参考: <https://docs.janusgraph.org/latest/indexes.html>

<https://developer.ibm.com/dwblog/2018/janusgraph-composite-mixed-indexes-traversals/>

JanusGraph 支持两种类型的索引: 图索引 (graph indexes) 和以顶点为中心的索引 (vertex-centric indexes)。

图索引是指针对整个图的全局索引, 它允许通过顶点或者边的属性的条件筛选来有效地检索顶点或边。而以顶点为中心的索引通常用于加速遍历包含很多边的那些顶点。

6.1 图索引

图索引又可以分为两类: 复合 (composite) 索引和混合 (mixed) 索引。

复合索引非常快速且高效, 但仅限于特定的, 已经定义的属性键组合的相等查找。混合索引可用于对索引键的任何组合进行查找, 并且除了相等之外还支持多个条件谓词, 具体取决于后端的索引引擎。

这两类索引都是通过 JanusGraph 管理系统 JanusGraphManagement.buildIndex(String, Class)构建。它的第一个参数定义了索引的名称, 第二个参数指定了被索引元素的类型, 比如 Vertex.class。

如果构建的图索引已经在被使用了, 那么需要对已有的数据做索引重建(Reindexing): 就是对已存在的图数据建立索引的过程。只有等索引重建完成之后, 索引才可以使用。

推荐在最开始创建 Schema 的时候就定义图索引。

注意: 建立和删除索引的时候都必须确保当前所有的事务都已经结束, 否则可能导致失败。

提示: 如果未建立任何索引, 为了检索期望的顶点列表, JanusGraph 默认会遍历整个图。尽管这会产生正确的结果, 但是这是非常低效的, 在生产环境中会导致极大地影响系统的性能。因此, 建议在生产环境中开启 force-index 配置, 禁止全图遍历。

配置方法: query.force-index=true

6.1.1 索引的状态

索引的状态

- INSTALLED
- REGISTERED
- ENABLED
- DISABLED

6.1.2 复合索引 Composite Index

复合索引不需要配置额外的索引后端，它与图数据存储在一起。复合索引只能用于相等的情况。

强烈建议对任何经常需要在查询中使用的属性键都建立复合索引以提高性能。有了索引可以加速图的查询，而不是每次都去遍历整个图。

注意：这里一般来说要区分两种情况：

第一种：创建索引的属性在之前不存在，在创建 Schema 之后就马上创建索引。也就是不需要重建索引。

第二种：要创建索引的属性已存在，且已经有属性相关的数据。需要索引重建。

这里先介绍第一种情况，后续将介绍第二种。

6.1.2.1 创建索引

警告：创建索引之前需要确保所有的事务都已结束。可以通过如下查询和关闭的方式来处理。

```
gremlin> graph.getOpenTransactions()
==>standardjanusgraphtx[0x675ec28b]
==>standardjanusgraphtx[0x63dda940]
gremlin> for (tx in graph.getOpenTransactions()) tx.rollback()
==>null
gremlin> graph.getOpenTransactions()
gremlin>

gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@2443abd6
gremlin> name = mgmt.makePropertyKey('name').dataType(String.class).cardinality(Cardinality.SINGLE).make()
==>name
gremlin> mgmt.buildIndex('byNameComposite', Vertex.class).addKey(name).buildCompositeIndex()
==>byNameComposite

gremlin> mgmt.commit()
==>null
```

6.1.2.2 查看索引状态

```
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@463afa6e
gremlin> name = mgmt.getPropertyKey('name')
==>name
gremlin> byName = mgmt.getGraphIndex('byNameComposite')
==>byNameComposite
gremlin> status = byName.getIndexStatus(name)
==>ENABLED
gremlin>
```

因为是在 Schema 之后直接创建的索引，在这种情况下，所以创建立即就会变成 Enabled 状态。

6.1.2.3 更新索引状态

<https://www.cnblogs.com/Uglthinx/p/9630779.html>

第二种情况：对已存在数据的属性建索引。

第一步：准备数据

```
gremlin> g = graph.traversal()
gremlin> g.addV().property("city", "Chengdu")
==>v[4248]
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@8840c98
gremlin> g.tx().commit()
==>null
```

注意：如果缺少 `g.tx().commit()` 这个步骤，是无法查询到 `city` 这个属性键的。

```
gremlin> city = mgmt.getPropertyKey('city')
==>city
gremlin> mgmt.commit()
==>null
```

第二步：创建索引

```
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@41f964f9
gremlin> mgmt.buildIndex('byCityComposite', Vertex.class).addKey(city).buildCompositeIndex()
==>byCityComposite
gremlin> mgmt.commit()
==>null
```

第三步：查询状态

```
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@7bb4ed7
gremlin> city = mgmt.getPropertyKey('city')
==>city
gremlin> byCity = mgmt.getGraphIndex('byCityComposite')
==>byCityComposite
gremlin> status = byCity.getIndexStatus(city)
==>INSTALLED
```

第四步：更新为 REGISTER 状态（暂时为可选，因为发现第三步获取就已经为 REGISTERED 了）

```
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@538b3c88
gremlin>
gremlin> mgmt.updateIndex(mgmt.getGraphIndex('byCityComposite'), SchemaAction.REGISTER_INDEX).get()
==>null
gremlin> mgmt.commit()
==>null
```

第五步：索引重建

```
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@33e50ff2
gremlin> mgmt.updateIndex(mgmt.getGraphIndex("byCityComposite"), SchemaAction.REINDEX).get()
==>org.janusgraph.diskstorage.keycolumnvalue.scan.StandardScanMetrics@78128dd0
gremlin> mgmt.commit()
==>null
```

第六步：查询状态

```
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@53dba9cd
gremlin> city = mgmt.getPropertyKey('city')
==>city
gremlin> byCity = mgmt.getGraphIndex('byCityComposite')
==>byCityComposite
gremlin> status = byCity.getIndexStatus(city)
==>ENABLED
```

6.1.2.4 索引分析

未建索引的查询如下

```
gremlin> g.addV().property('name', 'Dennis')
==>v[4184]
gremlin> g.V().has('name', 'Dennis').profile()
```

```

gremlin> g.V().has('name', 'Dennis').profile()
20:30:39 WARN org.janusgraph.graphdb.transaction.StandardJanusGraphTx - Query requires iterating over all vertices [(name = Dennis)]. For better performance, use indexes
==>Traversal Metrics
Step                                     Count Traversers      Time (ms)    % Dur
-----
JanusGraphStep([],[name.eq(Dennis)])      1           1         12.164     100.00
  \_condition=(name = Dennis)
  \_isFitted=false
  \_query=[]
  \_orders=[]
  \_isOrdered=true
  optimization 0.077
  optimization 0.112
  scan 0.000
  \_condition=VERTEX
  \_query=[]
  \_fullscan=true
>TOTAL - - 12.164 -

```

已建索引的查询

```
gremlin> g.V().has('city', 'Chengdu').profile()
```

```

gremlin> g.V().has('city', 'Chengdu').profile()
==>Traversal Metrics
Step                                     Count Traversers      Time (ms)    % Dur
-----
JanusGraphStep([],[city.eq(Chengdu)])      1           1          0.945     100.00
  \_condition=(city = Chengdu)
  \_isFitted=true
  \_query=multiKSQ[1]@2147483647
  \_index=byCityComposite
  \_orders=[]
  \_isOrdered=true
  optimization 0.019
  optimization 0.352
  optimization 0.945
>TOTAL - - 0.945 -

```

6.1.2.5 多字段索引

```
mgmt.buildIndex('byNameAndAgeComposite', Vertex.class).addKey(name).addKey(age).buildCompositeIndex()
```

6.1.2.6 唯一性索引

复合索引可用于为图定义唯一性约束。如果某个键被定义为唯一性，那么相应的顶点或者边就只允许有一个。例如，下面将定义一个 name 唯一性约束。

第一步：先确保所有的事务已结束

```

gremlin> graph.getOpenTransactions()
==>standardjanusgraphtx[0x14e750c5]
gremlin>
gremlin> for (tx in graph.getOpenTransactions()) tx.rollback()
==>null

```

第二步：创建唯一性索引

```

gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@2bf2d6eb
gremlin> name = mgmt.getPropertyKey('name')
==>name
gremlin> mgmt.buildIndex('byNameUnique', Vertex.class).addKey(name).unique().buildCompositeIndex()
==>byNameUnique
gremlin>
gremlin> mgmt.commit()
==>null
gremlin> ManagementSystem.awaitGraphIndexStatus(graph, 'byNameUnique').call()
==>GraphIndexStatusReport[succes=true, indexName='byNameUnique', targetStatus=[REGISTERED],
notConverged={}, converged={name=REGISTERED}, elapsed=PT0.01S]

```

第三步：重建索引

```

gremlin> graph.getOpenTransactions()
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@55d8c2c4
gremlin> mgmt.updateIndex(mgmt.getGraphIndex("byNameUnique"), SchemaAction.REINDEX).get()
==>org.janusgraph.diskstorage.keycolumnvalue.scan.StandardScanMetrics@62cf6a84
gremlin> mgmt.commit()
==>null

```


gremlin>

如何知道索引已重建完成？索引状态为 ENABLED 就认为重建完成了

查询索引

```
gremlin> graph.getOpenTransactions()
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@6ed18d80
gremlin> name = mgmt.getPropertyKey('name')
==>name
gremlin> byNameUnique = mgmt.getGraphIndex('byNameUnique')
==>byNameUnique
gremlin> status = byNameUnique.getIndexStatus(name)
==>ENABLED
gremlin>
```

检查唯一性约束

```
g.addV().property('name', 'Dennis')
gremlin> g.addV().property('name', 'Dennis')
Adding this property for key [name] and value [Dennis] violates a uniqueness constraint [byNameUnique]
Type :help or :h for help.
Display stack trace? [yN]
```

6.1.2.7 相等索引

复合索引只能用于相等的情况。举例如下：

第一步：创建一个顶点，含有 age 属性

```
gremlin> g.addV().property('name', 'Mike').property('age', 21)
==>v[4312]
gremlin> g.V().has('age', inside(20, 50))
21:40:12 WARN org.janusgraph.graphdb.transaction.StandardJanusGraphTx - Query requires iterating over all
vertices [(age > 20 AND age < 50)]. For better performance, use indexes
==>v[4312]
gremlin>
gremlin> g.tx().commit()
==>null
```

第二步：建立 age 索引

```
gremlin> graph.getOpenTransactions()
==>standardjanusgraphtx[0x6af78a48]
gremlin> for (tx in graph.getOpenTransactions()) tx.rollback()
==>null
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@5a4464c5
gremlin> age = mgmt.getPropertyKey('age')
==>age
gremlin> mgmt.buildIndex('byAgeComposite', Vertex.class).addKey(age).buildCompositeIndex()
==>byAgeComposite
gremlin> mgmt.commit()
==>null
```

查看状态

```
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@1b17d8ab
gremlin> age = mgmt.getPropertyKey('age')
==>age
gremlin> byAgeComposite = mgmt.getGraphIndex('byAgeComposite')
==>byAgeComposite
gremlin> status = byAgeComposite.getIndexStatus(age)
==>REGISTERED
```

```
gremlin> mgmt.commit()
==>null
```

重建索引

```
gremlin> graph.getOpenTransactions()
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@51424203
gremlin> mgmt.updateIndex(mgmt.getGraphIndex("byAgeComposite"), SchemaAction.REINDEX).get()
==>org.janusgraph.diskstorage.keycolumnvalue.scan.StandardScanMetrics@3962ec84
gremlin> mgmt.commit()
==>null
```

查看状态

```
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@3855d9b2
gremlin> age = mgmt.getPropertyKey('age')
==>age
gremlin> byAgeComposite = mgmt.getGraphIndex('byAgeComposite')
==>byAgeComposite
gremlin> status = byAgeComposite.getIndexStatus(age)
==>ENABLED
gremlin>
```

检查索引

```
gremlin> g.V().has('age', inside(20, 50))
22:00:30 WARN org.janusgraph.graphdb.transaction.StandardJanusGraphTx - Query requires iterating over all
vertices [(age > 20 AND age < 50)]. For better performance, use indexes
==>v[4312]
依然提示没有使用上索引。但是，如果使用相等，则可以使用上索引。
gremlin> g.V().has('age', 21)
==>v[4312]
```

```
gremlin> g.V().has('age', 21).profile()
==>Traversal Metrics
Step
```

	Count	Traversers	Time (ms)	% Dur
JanusGraphStep([], [age.eq(21)])	1	1	0.640	100.00
\condition=(age = 21)				
\isFitted=true				
\query=multiKSQ[1]@2147483647				
\index=byAgeComposite				
\orders=[]				
\isOrdered=true				
optimization			0.023	
optimization			0.248	
>TOTAL	-	-	0.640	-

```
gremlin>
```

接下来，我们看看如何用混合索引来解决非等的索引。

6.1.3 混合索引 Mixed Index

混合索引支持任意属性键的组合，它提供了更多的灵活性，并且支持非等的谓词。但是，混合索引的速度要比复合索引慢一些。

与复合索引不同的事，混合索引必须配置专门的索引后端。

6.1.3.1 创建索引

```
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@652e345
gremlin> age = mgmt.getPropertyKey('age')
==>age
gremlin> mgmt.buildIndex('age', Vertex.class).addKey(age).buildMixedIndex("search")
==>age
gremlin> mgmt.commit()
==>null
```

6.1.3.2 查看状态

```
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@1b3a9ef4
gremlin> age = mgmt.getPropertyKey('age')
==>age
gremlin> byAgeMixed = mgmt.getGraphIndex('age')
==>age
gremlin> status = byAgeMixed.getIndexStatus(age)
==>INSTALLED
gremlin> mgmt.commit()
==>null
gremlin>
```

如果不变为 REGISTERED，则执行如下语句更新状态。

```
mgmt.updateIndex(mgmt.getGraphIndex('age'), SchemaAction.REGISTER_INDEX).get()
```

6.1.3.3 索引重建

```
mgmt = graph.openManagement()
mgmt.updateIndex(mgmt.getGraphIndex("age"), SchemaAction.REINDEX).get()
mgmt.commit()
```

查看状态

如果一直未变为 enable，可以使用 `ManagementSystem.awaitGraphIndexStatus(graph, 'age').call()` 然后等待变为 ENABLED。

6.1.3.4 检查索引

```
gremlin> g.V().has('age', lt(50)).profile()
```

```
gremlin> g.V().has('age', lt(50)).profile()
==>Traversal Metrics
Step
```

	Count	Traversers	Time (ms)	% Dur
JanusGraphStep([],[age.lt(50)])	1	1	3.640	100.00
_condition=(age < 50)				
_isFitted=true				
_query=[(age < 50)]:age				
_index=age				
_orders=[]				
_isOrdered=true				
_index_impl=search				
optimization			0.052	
optimization			0.598	
>TOTAL	-	-	3.640	-

```
gremlin> g.V().has('age', inside(20, 50)).profile()
```

```
gremlin> g.V().has('age', inside(20, 50)).profile()
==>Traversal Metrics
Step
```

	Count	Traversers	Time (ms)	% Dur
JanusGraphStep([],[age.and(>(20), <(50))])	1	1	1.337	100.00
_condition=(age > 20 AND age < 50)				
_isFitted=true				
_query=[(age > 20 AND age < 50)]:age				
_index=age				
_orders=[]				
_isOrdered=true				
_index_impl=search				
optimization			0.035	
optimization			0.733	
>TOTAL	-	-	1.337	-

6.1.3.5 添加索引到新建的键

参考: 11.1.2.1. Adding Property Keys

6.2 标签约束

参考: 11.1.4. Label Constraint

6.3 以顶点为中心的索引

以顶点为中心的索引是每个顶点单独构建的局部索引结构。在大图中，顶点可能有数千个入射边。遍历这些顶点可能非常慢，因为必须检索大的事件边缘子集，然后在内存中进行过滤以匹配遍历的条件。以顶点为中心的索引可以通过使用本地化索引结构仅检索需要遍历的边缘来加速此类遍历。

在罗马诸神的例子中，假设除了捕获的三个怪物之外，Hercules 还与数百个怪兽作战。如果没有以顶点为中心的索引，则要查询哪些怪物在时间点 10 和 20 之间作战，尽管只有一些边满足条件，也需要遍历所有 battled 的边。

```
h = g.V().has('name', 'hercules').next()
g.V(h).outE('battled').has('time', inside(10, 20)).inV()
```

正如其名，以顶点为中心的索引是围绕一个顶点而创建的索引，它通常用于当一个顶点有大量的关联边的场景。一般来说，在建图模型的时候就需要创建图索引，只有在需要的时候才会创建以顶点为中心的索引。

以顶点为中心的索引，对于解决超级节点问题具有很大的帮助。

6.4 索引重建 Reindexing

在 JanusGraph 中，索引重建可以有两种框架来执行：

- MapReduce
- JanusGraphManagement

基于 MapReduce 的索引重建支持大规模、水平分布式数据库。而 JanusGraphManagement 的索引重建只能在单机运行。

鉴于效率考虑，推荐使用 MapReduce 来做索引重建。

6.4.1 MapReduce 索引重建

6.5 索引禁用与删除

参考: <https://docs.janusgraph.org/latest/index-admin.html>

6.5.1 索引查看

6.5.1.1 查看所有顶点索引

可以用罗马诸神的例子查看

先导入罗马诸神的数据，如下所示：

```
gremlin> GraphOfTheGodsFactory.load(graph)
==>null
```

然后查看

```
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@b01cb8d
gremlin> mgmt.getGraphIndexes(Vertex.class)
==>name
==>vertices
gremlin> mgmt.commit()
==>null
```

6.5.1.2 查看所有边索引

```
gremlin> mgmt = graph.openManagement()
==>org.janusgraph.graphdb.database.management.ManagementSystem@c5e69a5
gremlin> mgmt.getGraphIndexes(Edge.class)
==>edges
gremlin> mgmt.commit()
==>null
```

6.5.2 索引禁用

6.5.3 索引删除

索引的删除分为两个阶段：

第一阶段：将索引的状态改为 DISABLED。这时，JanusGraph 将停止使用索引并停止更新索引；索引相关的数据已经保留，但会被忽略。

第二阶段：这个阶段依赖于索引的类型。复合索引可以通过 JanusGraph 删除。与索引重建一样，索引的删除也可以通过 MapReduce 或者 JanusGraphManagement 来完成。然而，混合索引必须在索引后端手动删除。

JanusGraph 没有提供自动删除后端索引的机制。

索引删除后，除了索引 schema 之外，其他相关的数据都会被删除。并且该索引的 schema 是 DISABLED 状态。

7 JanusGraph 数据库管理

注意：JanusGraph 本身没有自己的进程，只是一堆 jar 包。它并没有以服务的方式，要么通过在代码中直接调用，那么通过 console 调用或者通过 Gremlin Server 接受调用。

7.1 JanusGraph 管理系统

通常所说的 JanusGraph 管理系统是指 JanusGraphManagement，它可以通过 graph 实例获得。如下所示：

```
mgmt = graph.openManagement()
```

通过 JanusGraphManagement，可以进行数据库的管理。例如：创建 Schema、构建索引等等。

7.2 JanusGraph 后端存储

7.2.1 Oracle Berkley DB

一般应用于数据规模较小的单机环境。Oracle Berkeley DB 是一个嵌入式的数据库。

7.2.2 Apache Cassandra

Apache Cassandra 是一个开源分布式的 NoSQL 数据库系统，类似于 Google 的 BigTable、Apache 的 HBase 等。它是一个非常优秀的数据库，是 JanusGraph 默认支持的后端存储系统。

注意：其实 Cassandra 也可以嵌入式模式运行。

7.2.3 ScyllaDB

ScyllaDB 是 C++ 版 Cassandra。

IBM 的采用 JanusGraph+ScyllaDB cluster 模式。

7.2.4 Apache HBase

Apache HBase 是基于 HDFS 的面向列的分布式数据库。

7.2.5 后端存储的选型问题

如果是小规模数据的使用，同时对复杂的分布式存储系统不熟悉，这种情况建议直接使用嵌入式的 Berkeley DB。这样会比较好，可以省去学习和维护后端存储的时间。

如果数据规模比较大，通常需要分布式支持，那么通常根据使用 JanusGraph 的团队以及公司的环境来决定。优先选择熟悉的分布式存储系统，比如团队已经部署有 HBase 集群，那么肯定优选 HBase，Cassandra 同理。

在对后端存储系统都不熟悉的情况下，建议使用 JanusGraph 默认的 Cassandra。

提示：百度的 HugeGraph 图数据库的后端也是采用的 Cassandra。

7.3 JanusGraph 后端索引

参考: <https://docs.janusgraph.org/latest/index-backends.html>

7.3.1 后端索引概述

7.3.1.1 一般索引

JanusGraph 支持的索引如下:

- eq (equal) 相等, 符号=
- neq (not equal) 不等, 符号!=
- gt (greater than) 大于, 符号>
- gte (greater than or equal) 大于等于, 符号>=
- lt (less than) 小于, 符号<
- lte (less than or equal) 小于等于, 符号<=

All comparison predicates are supported by String, numeric, Date and Instant data types. Boolean and UUID data types support the eq and neq comparison predicates.

eq and neq can be used on Boolean and UUID.

7.3.1.2 文本索引

文本索引一般包括两类:

部分匹配: 不区分大小写

- textContains: 搜索指定的字符串是否至少出现一次
- textContainsPrefix: is true if (at least) one word inside the text string begins with the query string
- textContainsRegex: 是否满足指定的正则表达式
- textContainsFuzzy: 查询是否有相似的字符串, 是否相似基于 Levenshtein 编辑距离算法计算。

全部匹配: 区分大小写

- textPrefix: if the string value starts with the given query string
- textRegex: if the string value matches the given regular expression in its entirety
- textFuzzy: if the string value is similar to the given query string (based on Levenshtein edit distance)

7.3.1.3 空间索引

空间索引主要包括如下:

- geoIntersect which holds true if the two geometric objects have at least one point in common (opposite of geoDisjoint).
- geoWithin which holds true if one geometric object contains the other.
- geoDisjoint which holds true if the two geometric objects have no points in common (opposite of geoIntersect).
- geoContains which holds true if one geometric object is contained by the other.

7.3.1.4 索引使用例子

7.3.1.5 支持的数据类型

JanusGraph 的复合索引支持能够存在 JanusGraph 的所有类型, 但混合索引只支持如下类型:

- Byte
- Short
- Integer
- Long
- Float
- Double
- String
- Geoshape
- Date
- Instant
- UUID

支持的空间数据类型如下: point, circle, box, line, polygon, multi-point, multi-line and multi-polygon

空间数据的索引只能通过混合索引来实现。

7.3.2 Elastic Search

7.3.3 Apache Solr

7.3.4 Apache Lucene

7.4 JanusGraph 部署模式

7.4.1 嵌入式

嵌入式

7.4.2 单机模式

7.4.3 集群模式

7.5 JanusGraph+HBase+ElasticSearch 安装

7.5.1 版本选择

JanusGraph0.3.1 兼容的各个软件版本如下:

Apache Cassandra 2.1.20, 2.2.10, 3.0.14, 3.11.0

Apache HBase 1.2.6, 1.3.1, 1.4.4

Google Bigtable 1.0.0, 1.1.2, 1.2.0, 1.3.0, 1.4.0

Oracle BerkeleyJE 7.4.5

Elasticsearch 1.7.6, 2.4.6, 5.6.5, 6.0.1

Apache Lucene 7.0.0

Apache Solr 5.5.4, 6.6.1, 7.0.0

Apache TinkerPop 3.3.3

Java 1.8

根据本次环境搭建, 选择如下:

JanusGraph 0.3.1

HBase 1.2.6

ElasticSearch 6.0.1

Java 1.8

7.5.2 软件下载

JanusGraph 0.3.1 到这里下载 <https://github.com/JanusGraph/janusgraph/releases/>。大约 261M。

HBase1.2.6 到这里下载 <http://archive.apache.org/dist/hbase/>, 选择 1.2.6。hbase-1.2.6-bin.tar.gz, 大约 100M 左右。我的百度网盘也有: 链接: https://pan.baidu.com/s/12UtC2cXbGjTn_nPfOuB2JA 提取码: h52b

7.5.3 JanusGraph+HBase 独立模式+单机 ElasticSearch

这里以 CentOS7 作为安装环境。

7.5.3.1 JanusGraph 安装

[1] 解压

```
[root@centos7 ~]# cd /opt/
```

```
[root@centos7 opt]# unzip ~/janusgraph-0.3.1-hadoop2.zip
```

[2] 重命名解压文件夹

```
[root@centos7 opt]# mv janusgraph-0.3.1-hadoop2/ janusgraph-0.3.1
```

[3] 设置环境变量

将如下部分加入到/etc/profile 的末尾

```
export JANUSGRAPH_HOME=/opt/janusgraph-0.3.1
```

```
export PATH=$JANUSGRAPH_HOME/bin:$PATH
```

[4] 让环境变量生效

```
[root@centos7 ~]# . /etc/profile
```

7.5.3.2 Elasticsearch 安装

说明: JanusGraph 安装包里自带了 Elasticsearch, 因此只需要直接启动即可, 但自带的压缩包版本 Elasticsearch 只能用非 root 用户启动。

7.5.3.3 HBase 安装

[1] 解压 hbase 压缩包

```
[root@centos7 ~]# cd /opt/
```

```
[root@centos7 opt]# tar xzf ~/hbase-1.2.6-bin.tar.gz
```

解压后的目录为: /opt/hbase-1.2.6

[2] 配置 hbase-env.sh

配置文件位于 /opt/hbase-1.2.6/conf/hbase-env.sh

修改 JAVA_HOME

```
export JAVA_HOME=/usr/java/jdk1.8.0_171-amd64/jre
```

修改 HBASE_PID_DIR

```
export HBASE_PID_DIR=/opt/hbase-1.2.6/data/pids
```

注释掉 PermSize 配置

```
# Configure PermSize. Only needed in JDK7. You can safely remove it for JDK8+
```

```
#export HBASE_MASTER_OPTS="$HBASE_MASTER_OPTS -XX:PermSize=128m -XX:MaxPermSize=128m"
```

```
#export HBASE_REGIONSERVER_OPTS="$HBASE_REGIONSERVER_OPTS -XX:PermSize=128m -XX:MaxPermSize=128m"
```

[3] 配置 hbase-site.xml

配置文件位于 /opt/hbase-1.2.6/conf/hbase-site.xml

```
<configuration>
  <property>
    <name>hbase.rootdir</name>
    <value>file:///opt/hbase-1.2.6/data/hbase</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>/opt/hbase-1.2.6/data/zookeeper</value>
  </property>
</configuration>
```

说明:

- 上面配置的两个目录不需要事先创建, hbase 会自动创建;
- hbase.rootdir 是 RegionServer 的共享目录, 用来持久化 HBase;
- 如果不配置上面的目录, 默认存放在 /tmp 目录下, 机器重启后可能被删除掉。

[4] 配置环境变量

在 /etc/profile 末尾追加如下两行:

```
export HBASE_HOME=/opt/hbase-1.2.6
```

```
export PATH=:$HBASE_HOME/bin:$PATH
```

执行” . /etc/profile”让环境变量即时生效。

[5] 启动 hbase

```
[root@centos7 ~]# start-hbase.sh
```

[6] 检查 hbase 是否启动成功

先检查进程是否在，用 `jps` 看是否有 HMaster 进程。

```
[root@centos7 ~]# jps
761 Elasticsearch
2077 HMaster
```

在独立模式下，HMaster、HRegionServer 和 ZooKeeper 都运行在同一个 JVM 中。

注意：这里有个坑。一定要在 `/etc/hosts` 配置 IP 地址和 `/etc/hostname` 的映射，否则会导致 hbase 启动失败。

7.5.3.4 安装验证

通过启动控制台，然后打开上述的配置实例来验证安装的正确性。

[1] 启动 gremlin 控制台

```
[root@centos7 ~]# gremlin.sh
```

[2]

```
gremlin> graph = JanusGraphFactory.open('conf/janusgraph-hbase-es.properties')
==>standardjanusgraph[hbase:[127.0.0.1]]
gremlin>
gremlin> g = graph.traversal()
==>graphtraversalsource[standardjanusgraph[hbase:[127.0.0.1]], standard]
gremlin> g.V().count()
==>0
```

如果能查询出来当前图数据库的节点总数，那么就表明安装是没问题的。

```
bin/gremlin.sh
gremlin> graph = JanusGraphFactory.open('conf/janusgraph-hbase-es.properties')
gremlin> g = graph.traversal()
gremlin> g.addV('person').property('name','Dennis')
gremlin> g.tx().commit()
```

7.5.4 Hadoop+HBase 伪分布式模式

7.5.5 Hadoop+HBase 完全分布式模式

7.6 JanusGraph+Cassandra+ElasticSearch

提示：百度自己的 JanusGraph 集群后端存储用的是 Cassandra。

7.6.1 Cassandra 安装

Cassandra

7.7 JanusGraph 配置与日志

7.8 JanusGraph 备份与恢复

7.9 JanusGraph 导入与导出

http://tinkerpop.apache.org/docs/3.3.0/reference/#_gremlin_i_o
<https://github.com/IBM/janusgraph-utils>

https://github.com/IBM/janusgraph-utils/blob/master/doc/users_guide.md
<https://www.datastax.com/dev/blog/powers-of-ten-part-i>

7.9.1 csv 导入

http://tinkerpop.apache.org/docs/current/tutorials/getting-started/#_loading_data
https://blog.csdn.net/qq_32140547/article/details/82387885?tdsourcetag=s_pcqq_aiomsg JanusGraph-Import

7.9.2 自定义导入类

<https://stackoverflow.com/questions/50956162/load-data-into-janusgraph>

7.9.3 bulk load

<http://tinkerpop.apache.org/docs/current/reference/#bulkloadvertexprogram>

7.9.3.1 storage.batch-loading

注意: 开启 storage.batch-loading 需要用户确保导入的数据内在的一致性以及与数据库中已经存在的数据的一致性。

自动类型创建容易导致数据的完整性受损, 强烈建议关闭自动类型创建: `schema.default = none`

7.9.3.2 id 分配优化

设置 `ids.block-size`

`ids.authority.wait-time`

`ids.renew-timeout`

7.9.3.3 优化读写

`storage.buffer-size`

7.9.4 GraphML

说明: Gephi 也支持 GraphML。

7.9.4.1 导入

```
graph.io(graphml()).readGraph('air-routes-latest.graphml')
```

7.9.4.2 导出

```
graph.io(graphml()).writeGraph('my-graph.graphml')
```

7.9.5 GraphSON

GraphSON 目前有三个版本, JanusGraph 都支持。最初的 1.0 版本不包含任何类型信息。2.0 版本在 JSON 中引入了数据类型。GraphJSON3.0 作为 Tinkerpop 的一部分添加了一些额外类型。

7.9.5.1 导入

```
graph.io(IoCore.graphson()).readGraph('my-graph.json')
```

生成 adjacency list form of GraphSON

生成 wrapped adjacency list form of GraphSON.

```
// Create a single (wrapped) JSON file
```

```
fos = new FileOutputStream("my-graph.json")
```

```
GraphSONWriter.build().wrapAdjacencyList(true).create().writeGraph(fos, graph)
```

7.9.5.2 导出

```
graph.io(graphson()).writeGraph("airport.json")
```

7.9.6 Gryo

Kryo is a popular serialization package for the JVM. Gremlin-Kryo is a binary Graph serialization format for use on the JVM by JVM languages.

8 JanusGraph 开发

JanusGraph 开发通常来说可以分为两种模式：嵌入式模式和 JanusGraph Server 模式。它们的区别如下：

- 嵌入式模式：JanusGraph 与客户端运行在同一个 JVM 中，没有独立的 JanusGraph 服务或者进程；
- JanusGraph Server 模式：JanusGraph 与客户端运行在不同的 JVM 中，客户端通过 WebSocket 或者 HTTP 来访问 JanusGraph。

<https://github.com/JanusGraph/janusgraph/tree/master/janusgraph-examples>

8.1 嵌入式模式

<http://www.k6k4.com/chapter/show/aafiizxav1531746415578>

在 Gremlin 语言的章节中讲过，操作 JanusGraph 有两套 API，分别是 Graph Structure 和 Graph Process。建议只用 graph 来做图的模型定义及数据库管理相关操作。图的数据操作，包括创建、更新、删除及便利都用 g 来操作。如果想用 API 来大批量地操作数据，可以跳过 JanusGraph，直接写入后端存储。

下面将分别介绍这两类 API 的操作。

8.1.1 Graph Structure API

注意：本课程中所有的 Java 工程都用 Maven 来管理，这也是目前大多数公司所采用的方式。

前提条件：

- [1] 安装并配置好了 Maven 环境；
- [2] 安装并配置好了 IntelliJ IDEA；

第一步：使用 Maven 命令创建 JanusGraphStruceBerkeley 工程

工程名的含义可这样分解：Janus 代表 Janus 图数据库，GraphStrutuce 代表 Graph Structure API，Berkeley 代表后端存储使用 BerkeleyJE。

在 Maven 命令行执行如下命令：

```
mvn archetype:generate -DgroupId=cn.dennishucd -DartifactId=JanusGraphStruceBerkeley -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

第二步：将 Maven 工程导入到 IDEA

File-->New-->Project from Existing Sources...，找到刚创建的 Maven 工程，选中 pom.xml，然后点击“OK”在弹出的窗口中，将“Import Maven projects automatically”勾上，然后一路确定就可以了。

第三步：添加工程所需的 Maven 依赖

```
<dependency>
  <groupId>org.janusgraph</groupId>
  <artifactId>janusgraph-core</artifactId>
  <version>0.3.1</version>
</dependency>
<dependency>
  <groupId>org.janusgraph</groupId>
  <artifactId>janusgraph-berkeleyje</artifactId>
  <version>0.3.1</version>
</dependency>
```


第四步：编写代码

```
package cn.dennishucd;

import
org.apache.tinkerpop.gremlin.process.traversal.dsl.graph.GraphTraversalSource;
import org.janusgraph.core.JanusGraph;
import org.janusgraph.core.JanusGraphFactory;
import org.janusgraph.core.JanusGraphTransaction;

public class JanusDemo
{
    public static void main( String[] args ) {
        JanusGraph graph = JanusGraphFactory.build()
            .set("storage.backend", "berkeleyje")
            .set("storage.directory", "db/graph")
            .open();

        //add a vertex
        JanusGraphTransaction tx = graph.newTransaction();
        tx.addVertex("user").property("name", "Jesson");
        tx.commit();

        GraphTraversalSource g = graph.traversal();
        System.out.println("Vertex count = " + g.V().count().next());

        System.exit(0);
    }
}
```

要点讲解：

- 数据库目录
- 创建实例的方式
- 创建边
- 两种事务方式都可以

8.1.2 graph traversal 方法

```
代码 package cn.dennishucd;

import
org.apache.tinkerpop.gremlin.process.traversal.dsl.graph.GraphTraversalSource;
import org.janusgraph.core.JanusGraph;
```

```
import org.janusgraph.core.JanusGraphFactory;

public class JanusDemo
{
    public static void main( String[] args ) {
        JanusGraph graph = JanusGraphFactory.build()
            .set("storage.backend", "berkeleyje")
            .set("storage.directory", "db/graph")
            .open();

        GraphTraversalSource g = graph.traversal();
        g.addV("user").property("name", "Dennis").next();
        g.tx().commit();
        g.tx().close();

        System.out.println("Vertex count = " + g.V().count().next());

        System.exit(0);
    }
}
```

8.2 JanusGraph Server

参考: <https://docs.janusgraph.org/latest/server.html>

JanusGraph 是使用的 Tinkerpop 的 Gremlin Server 做为 JanusGraph Server。JanusGraph Server 可以配置为 WebSocket 和 HTTP 两种访问接口。

JanusGraph Server 需要手动启动。JanusGraph 的安装包中自带了一个以 Cassandra 为后端存储和 ElasticSearch 为外部索引的配置，可以直接通过 `bin/janusgraph.sh start` 启动。

8.2.1 Windows 环境

[1] 需要先配置 Gremlin-Server，将 JanusGraph 对应的版本注册到 Gremlin-Server，以便找到对应的依赖。
`bin\gremlin-server.bat -i org.janusgraph janusgraph-all 0.3.1`

[2] 然后创建配置文件

`janusgraph-berkeleyje-es.properties`

`conf/gremlin-server-janusgraph.yaml`

`host: localhost`

`port: 8182`

`graphs: {`

`graph: conf/janusgraph.properties}`

`scriptEngines: {`

`gremlin-groovy: {`

`plugins: { org.janusgraph.graphdb.tinkerpop.plugin.JanusGraphGremlinPlugin: {},`

`org.apache.tinkerpop.gremlin.server.jsr223.GremlinServerGremlinPlugin: {},`

`org.apache.tinkerpop.gremlin.tinkergraph.jsr223.TinkerGraphGremlinPlugin: {}},`

```
    org.apache.tinkerpop.gremlin.jsr223.ImportGremlinPlugin: {classImports: [java.lang.Math], methodImports:
[java.lang.Math#*]},
    org.apache.tinkerpop.gremlin.jsr223.ScriptFileGremlinPlugin: {files: [scripts/empty-sample.groovy]}}}}
serializers:
- { className: org.apache.tinkerpop.gremlin.driver.ser.GryoMessageSerializerV3d0, config: { ioRegistries:
[org.janusgraph.graphdb.tinkerpop.JanusGraphIoRegistry] }}
- { className: org.apache.tinkerpop.gremlin.driver.ser.GryoMessageSerializerV3d0, config:
{ serializeResultToString: true }}
- { className: org.apache.tinkerpop.gremlin.driver.ser.GraphSONMessageSerializerV3d0, config: { ioRegistries:
[org.janusgraph.graphdb.tinkerpop.JanusGraphIoRegistry] }}
metrics: {
  slf4jReporter: {enabled: true, interval: 180000}}
```

[3] 启动

bin\gremlin-server.bat conf/gremlin-server-janusgraph.yaml

[4] 验证

控制台验证

```
gremlin> :remote connect tinkerpop.server conf/remote.yaml
==>Configured localhost/127.0.0.1:8182
gremlin> :> g.V().count()
==>12
gremlin>
```

REST 方式验证

直接在浏览器中输入:

[http://localhost:8182?gremlin=g.V\(\).has\('name','Dennis'\).valueMap\(\)](http://localhost:8182?gremlin=g.V().has('name','Dennis').valueMap())

```
gremlin> :remote connect tinkerpop.server conf/remote.yaml
==>Configured localhost/127.0.0.1:8182
gremlin>
```

8.2.2 Linux 环境

8.2.2.1 Cassandra+ES(默认)

配置接口方式

conf/gremlin-server/gremlin-server.yaml

测试

```
curl "http://192.168.142.164:8182?gremlin=g.V().count()"
```

```
curl "http://192.168.142.164:8182?gremlin=g.addV('person').property('name','Dennis')"
```

```
curl "http://192.168.142.164?gremlin=g.V().count()"
```

8.2.2.2 Berkeley+es

第一步: 启动 elasticsearch

[1] chown -R dennis:dennis elasticsearch/

[2]

su - dennis

\$ bin/elasticsearch &

第二步: 配置 elasticsearch 参数

配置文件为 conf/gremlin-server/janusgraph-berkeleyje-es-server.properties
index.search.hostname 的默认值为 elasticsearch, 修改为 localhost 如下:

```
index.search.hostname=localhost
```

第三步: 配置 channelizer

注: 这一步可选, 如果不配置, 那么 Gremlin Server 就只支持 WebSocket 接口, 不支持 HTTP 接口。

配置文件为 conf/gremlin-server/gremlin-server-berkeleyje.yaml

channelizer 的默认值为 org.apache.tinkerpop.gremlin.server.channel.WebSocketChannelizer, 只支持 WebSocket 方式, 这里修改为如下, 同时支持 WebSocket 和 HTTP 接口。

channelizer: org.apache.tinkerpop.gremlin.server.channel.WsAndHttpChannelizer

第四步: 启动 Gremlin Server

```
bin/gremlin-server.sh conf/gremlin-server/gremlin-server-berkeleyje.yaml &
```

第四步: 访问 Gremlin Server

[1] HTTP 接口方式访问

可以直接在命令行通过 curl 获取, 例如:

查询当前数据库的顶点数:

```
curl "http://192.168.142.164:8182/?gremlin=g.V().count()"
```

创建一个 person 顶点 ()

```
curl "http://192.168.142.164:8182?gremlin=g.addV('person').property('name','Dennis')"
```

查询

```
curl "http://192.168.142.164:8182?gremlin=g.V().has('person').property('name','Dennis')"
```

[2] WebSocket 接口方式访问

由于 Gremlin Console 采用的是 WebSocket 接口, 所以可以通过它验证 WebSocket 接口。

启动 Gremlin Console

```
bin/gremlin.sh
```

```
gremlin> :remote connect tinkerpop.server conf/remote.yaml
```

```
==>Configured localhost/127.0.0.1:8182
```

```
gremlin>
```

```
gremlin> :> g.V().count()
```

```
==>1
```

8.3 Java 开发(Gremlin-Driver)

Gremlin-Java: <http://tinkerpop.apache.org/docs/3.4.0/reference/#gremlin-java>

Java API: <http://tinkerpop.apache.org/javadocs/current/full/>

8.4 Python 开发 (Gremlin-Python)

```
pip install gremlinpython
```

源码

```
from gremlin_python.driver import client
```

```
client = client.Client('ws://192.168.142.164:8182/gremlin', 'g')
```

```
gremlin = "g.V().count()"
```

```
result_set = client.submit(gremlin)
```

```
future_results = result_set.all()
```

```
results = future_results.result()
```

```
print (results)
```

```
print ("End")
```

8.5 REST 开发(HTTP)

8.5.1 JavaWebsocket

8.5.1.1 工程搭建

```
mvn archetype:generate -DgroupId=cn.dennishucd -DartifactId=JanusWebsocket -DarchetypeArtifactId=maven-archetype-quickstart -interactiveMode=false
```

8.5.1.2 maven 引用

```
<dependency>
  <groupId>org.apache.tinkerpop</groupId>
  <artifactId>gremlin-core</artifactId>
  <version>3.3.3</version>
</dependency>
<dependency>
  <groupId>org.apache.tinkerpop</groupId>
  <artifactId>gremlin-driver</artifactId>
  <version>3.3.3</version>
</dependency>
```

8.5.1.3 代码

<https://github.com/JanusGraph/janusgraph/tree/master/janusgraph-examples/example-remotegraph>

8.5.2 定制 Gremlin-Server 的配置文件

```
mvn archetype:generate -DgroupId=cn.dennishucd -DartifactId=JanusRestClient -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

GraphSONMapper 可以对结果进行格式化

8.6 JanusGraph 认证

8.7 JanusGraph 与 Spark 集成

https://docs.janusgraph.org/latest/hadoop-tp3.html#_olap_traversals_with_spark_local

<http://tinkerpop.apache.org/docs/current/reference/#interacting-with-spark>

<http://tinkerpop.apache.org/docs/current/reference/#spark-plugin>

Spark 插件是作为 spark-gremlin 的一部分来安装的。插件的具体定义在 SparkGraphComputer 部分，而这部分是通过 Hadoop-Plugin 一起安装的。

8.7.1 插件安装

参考: <http://tinkerpop.apache.org/docs/3.3.3/reference/#hadoop-gremlin>

```
[root@centos7 ~]# gremlin
```

```
gremlin> :install org.apache.tinkerpop hadoop-gremlin 3.3.3
```

```
==>a module with the name hadoop-gremlin is already installed
```

```
gremlin>
```

安装完之后需要重启 Gremlin Console, 如下:

```
gremlin> :q
```

```
gremlin> :plugin use tinkerpop.hadoop
```

```
==>tinkerpop.hadoop activated
gremlin>
```

```
:install org.apache.tinkerpop spark-gremlin 3.3.3
```

```
:plugin use tinkerpop.spark
```

8.7.2 配置

conf/hadoop-graph/hadoop-script.properties

gremlin.hadoop.defaultGraphComputer 设置默认图计算引擎

9 JanusGraph 图计算

<https://docs.janusgraph.org/latest/hadoop-tp3.html>

<https://tinkerpop.apache.org/docs/3.3.3/reference/#hadoop-gremlin>

9.1 Hadoop 与 JanusGraph 集成

9.1.1 Hadoop 伪分布式环境搭建

略

然后确保 hdfs 已启动

```
export HADOOP_HOME=/opt/hadoop-2.7.3
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export CLASSPATH=$HADOOP_CONF_DIR
```

9.1.2 启动 JanusGraph

```
cd /opt/janusgraph-0.3.1-hadoop2/
```

```
bin/gremlin.sh
```

```
gremlin> hdfs
```

```
==>storage[DFS[DFSClient[clientName=DFSClient_NONMAPREDUCE_940431940_1, ugi=dennis
(auth:SIMPLE)]]]
```

```
gremlin>
```

如果能看到 DFSClient，就说明配置成功。

9.1.3 Spark 图计算

```
gremlin> graph = GraphFactory.open('conf/hadoop-graph/read-cassandra-3.properties')
```

```
==>hadoopgraph[cassandra3inputformat->gryooutputformat]
```

```
gremlin> g = graph.traversal().withComputer(SparkGraphComputer)
```

```
==>graphtraversalsource[hadoopgraph[cassandra3inputformat->gryooutputformat], sparkgraphcomputer]
```

```
gremlin>
```

```
gremlin> g.V().count()
```

```
[Stage 0:=====> ==>0 (1 + 4) / 5]
```

```
gremlin>
```

10 JanusGraph 可视化

尝试下 graphEXP

<https://github.com/bricaud/graphexp>

10.1 可视化概述

<https://www.compose.com/articles/announcing-the-data-browser-for-janusgraph/>

Compose 被 IBM 收购了。

<https://www.ibm.com/cloud/compose/janusgraph>

<https://www.compose.com/>

KeyLines by Cambridge Intelligence 是收费的，参考：<https://cambridge-intelligence.com/keylines/janusgraph/>

10.2 Graphexp 集成

10.2.1 源码安装方式

第一步：下载和安装源码

修改 graphexp.html 中的，将 localhost 改为 IP 地址。

第二步：安装和配置 nginx

```
server
{
    listen    8083;
    server_name 172.27.9.140;
    index index.html;

    location /
    {
        root /opt/graphexp;
    }
}
```

sudo nginx -s reload

第三步：浏览器中访问

[1] 访问 <http://192.168.142.164:8183/graphexp.html>

点击“Get graph info”按钮，可以获取到 JanusGraph 中的图数据。

[2] 加载罗马诸神图

为了更好地展示图的效果，通过 Gremlin Console 加载

gremlin> :remote connect tinkerserver conf/remote.yaml

10.2.2 Docker 安装方式

10.3 Gephi 插件集成

注意大坑：<https://github.com/gephi/gephi/issues/1832> 不能用中文，必须恢复英文。

<http://tinkerpop.apache.org/docs/current/reference/#gephi-plugin>

这篇文章不错：Gephi 教程：使用 Graph Streaming 实现图可视化

<https://xiuxiuing.gitee.io/blog/2018/10/22/gephi/>

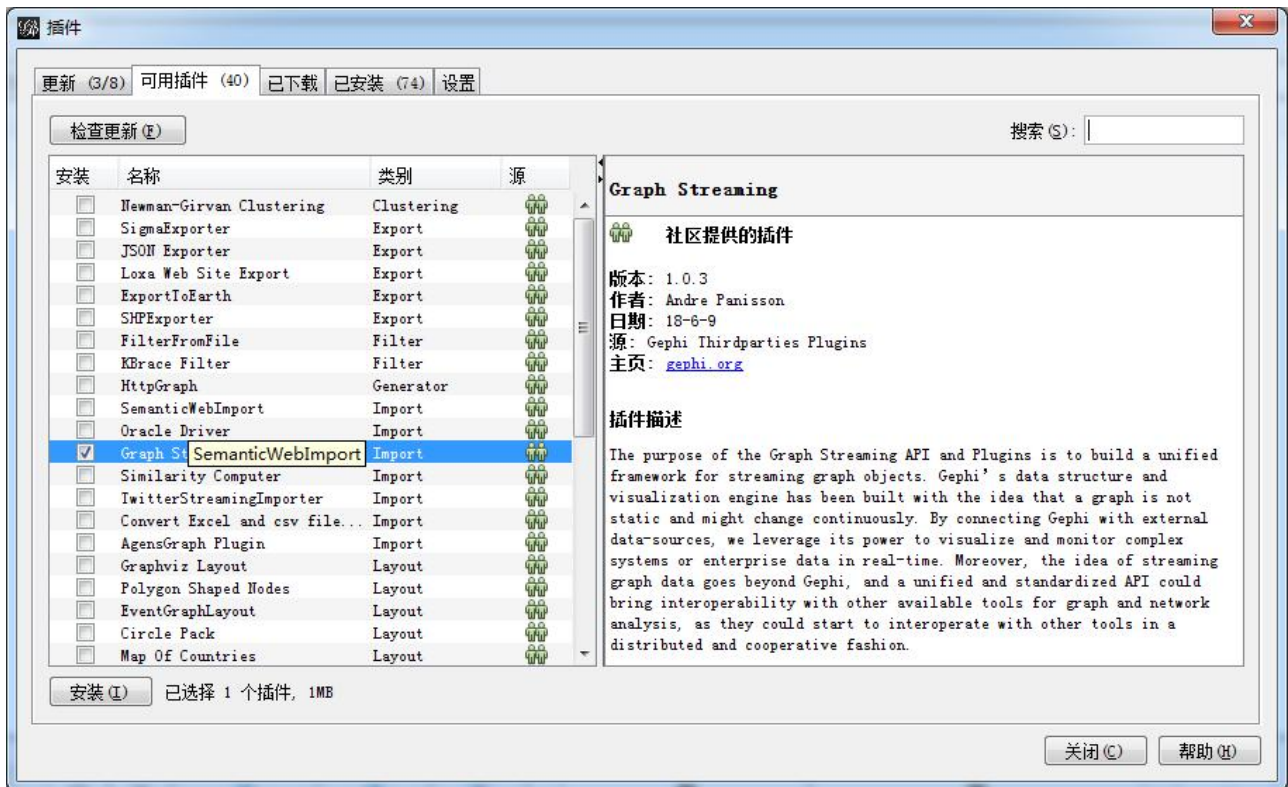
10.3.1 下载与安装

下载 <https://gephi.org/>，当前最新版本为 gephi-0.9.2-windows，大约 73M。

10.3.2 集成

新建工程

工具->插件->可用插件中，选中“Graph Streaming”，点击左下方的安装按钮，如下所示：



安装完成需要重启 Gephi

参考:

- [1] <https://gephi.org/>
- [2] <https://www.udemy.com/gephi/>

可以试下这个

<https://github.com/bricaud/graphexp>

11 JanusGraph 集群

待写

12 应用案例分析

待写

13 性能优化

13.1 explain

Explain 可以放在 Gremlin 部分

在末尾加上 explain()之后, 语句不会被执行。

疑问: `g.addV().property('name','Mike').property('name','mike').explain()`为啥只添加了一个 name?

13.2 profile

You can use the profile step to ask Gremlin to give you a more fine grained summary of where the time is spent processing your query.

作者 QQ:18037707

13.3 clock/clockWithResult

```
clock(1){g.V().has('code', 'LHR')}
```

clock 括号中的数字为执行的次数，有时候为了获取多次执行的平均时间，这样会更准确一些。