Titan 数据库学习分析报告

Titan 是一个可扩展的图形数据库，完美结合HBase、Cassandra、BerkeleyDB提供存储功能，ES、Lucene、Solar提供索引功能以及利用Hadoop计算框架对图数据进行分析、统计。经过优化，可用于存储和索引分布于多节点集群的百亿级顶点和边的图，同时，Titan又是一个事务数据库，可以支持数千个并发用户实时执行复杂图形遍历。本文对Titan数据库的架构、核心实现、存储模型等进行学习分析，并对Titan 2.0升级进行了简单的介绍，最后介绍了目前Titan中的各种限制。

# 一 Titan数据库简介

## 1.1简介

从某种程度上来讲，Titan并不是数据库，可以称之为一个数据库之上的客户端库，它既不提供存储功能也不提供索引功能。它依赖于底层的存储引擎，比如：HBase、Cassandra、BerkeleyDB，来存储其数据，同时依赖于索引引擎，比如：[Lucene](https://baike.baidu.com/item/Lucene)、[ElasticSearch](https://baike.baidu.com/item/ElasticSearch" \t "_blank)或者Solr，提供的索引服务，这些也为Titan的可扩展性的标记。非常容易搭建大规模的Titan开发和生产环境，只要个人或者公司的技术栈中存在上面提到的技术，就可以搭建出Titan的分布式数据库系统。

Titan是一个**开箱即用**的**配置系统**，也就是说，只要本地或者集群中已成功运行HBase、Cassandra和[Lucene](https://baike.baidu.com/item/Lucene" \t "_blank)、[ElasticSearch](https://baike.baidu.com/item/ElasticSearch" \t "_blank)、Solr中的组件就可以单独或者组合提供服务，Titan提供相应的配置文件，对底层选用的组件进行配置。

Titan除了提供存储、查询和事物功能还提供如下功能：

* 不断增长的数据和用户群弹性扩展；
* 采用数据分布式存储和备份机制提供性能、容错的保证；
* 多数据中心高可用性和热备份；
* 支持ACID和最终的一致性；
* 支持各种存储后端：

Apache Cassandra

Apache HBase

Oracle BerkeleyDB

* 通过与大数据平台的集成，支持全局图形数据分析，报告和ETL：

Apache Spark

Apache Giraph

Apache Hadoop

* 通过对以下组件的集成支持地理，数字范围和全文搜索：

ElasticSearch

Solr

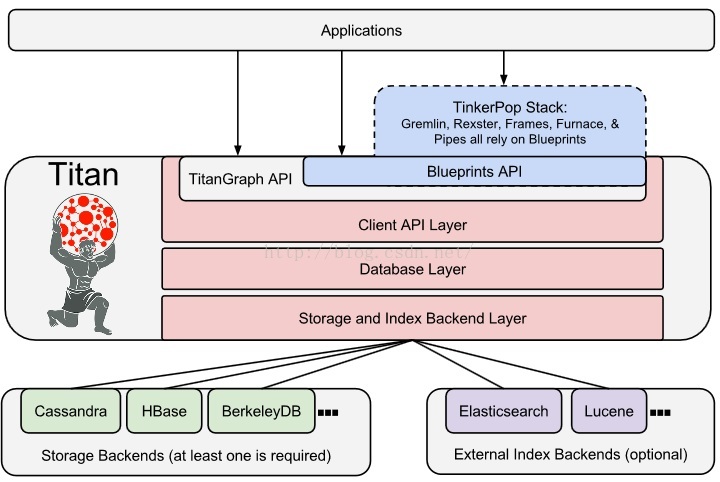
Lucene

* 开放源码，供二次开发。

## 1.2 框架简介

Titan是一个图形数据库引擎，其本身专注于紧凑图表序列化，丰富的图形数据建模和高效的查询执行。此外，Titan还利用Hadoop进行图形分析和批处理。同时，实现强大的模块化接口，用于数据持久性，数据索引和客户端访问。Titan的模块化架构使其能够与广泛的存储，索引和客户端技术进行互操作，它也简化了扩展Titan来支持新的操作。

其架构如下图所示：



Titan与底层磁盘之间存在一个或多个存储和索引适配器，至少需要Apache Cassandra 、Apache HBase、Oracle BerkeleyDB中的一个或者其它第三方组件作为存储后端，提供存储功能；至少需要ElasticSearch、Solr、Lucene中的一个或者其它第三方组件作为索引引擎提供索引功能。

Titan提供三种交互接口，分别是：TitanGraph API、TinkerPop Stack和Blueprints API。通过TitanGraph API将Titan嵌入应用程序，直接执行Gremlin查询操作图，在查询执行的过程中，Titan的缓存和事务处理都发生在与应用程序相同的JVM中，而存储后端执行的数据检索可能位于本地或远程。另外，通过向服务器提交Gremlin查询来与本地或远程Titan实例进行交互，Titan本身支持Tinkerpop堆栈的Gremlin Server组件。

Titan具有以下特性：

1. 支持大规模图集，其规模随着机群中的节点扩展而扩展；
2. 支持大规模并发事务及图操作，Titan的事务处理能力随着集群节点的扩展而扩展，并且对于复杂的遍历和查询达到毫秒级的反应速度；
3. 通过Hadoop框架集成支持对图形的全局图分析和批处理操作；
4. 支持超大图形上的地理，数字范围和对顶点和边的全文搜索；
5. 天然支持由TinkerPop曝露的主流图数据模型；
6. 天然支持图形遍历语言Gremlin；
7. 与Gremlin图服务器轻松集成，实现非语言相关性编程；
8. 对于许多图级配置项提供性能优化调节支持；
9. 以顶点为中心的索引提供顶点级查询，以缓解臭名昭着的超级节点问题；
10. 提供磁盘优化指标，以便有效利用存储，提高访问速度。

## 1.3 简单使用

下载Titan并解压即可执行gremlin命令进行操作，可参考Titan内置样例“诸神图”，不再赘述。

关于Tinkerpop的使用参见：<http://tinkerpop.apache.org/docs/3.0.1-incubating/> 、Gremlin的使用参见：[http://gremlindocs.spmallette.documentup.com/、 SQL2Gremlin](http://gremlindocs.spmallette.documentup.com/、%20SQL2Gremlin)的使用参加：<http://sql2gremlin.com/>

$ unzip titan-1.0.0-hadoop1.zip //下载的Titan安装包解压

Archive: titan-1.0.0-hadoop1.zip

creating: titan-1.0.0-hadoop1/

...

$ cd titan-1.0.0-hadoop1//进入解压后的目录，期目录结构如下：

$ bin/gremlin.sh//启动gremlin客户端服务，用于执行gremlin命令，操作titan数据库

\,,,/

(o o)

-----oOOo-(3)-oOOo-----

09:12:24 INFO org.apache.tinkerpop.gremlin.hadoop.structure.HadoopGraph - HADOOP\_GREMLIN\_LIBS is set to: /usr/local/titan/lib

plugin activated: tinkerpop.hadoop

plugin activated: aurelius.titan

JAVA API 使用参见下面文档，不再赘述：



## 1.4简单配置

Titan 配置文件是基于我们选择的存储和索引组件的，配置文件位于Titan的安装目录下：

titan-2.0.0-hdh4.0.0 /conf 下面，由于我们选择使用HBase作为后端存储，ES作为索引引擎，因此需要关注的配置文件是：

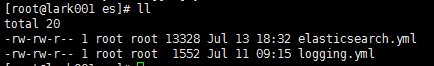


配置Titan存储相关，不是HBase的配置。



此外，Titan中可以对ES进行配置位于：

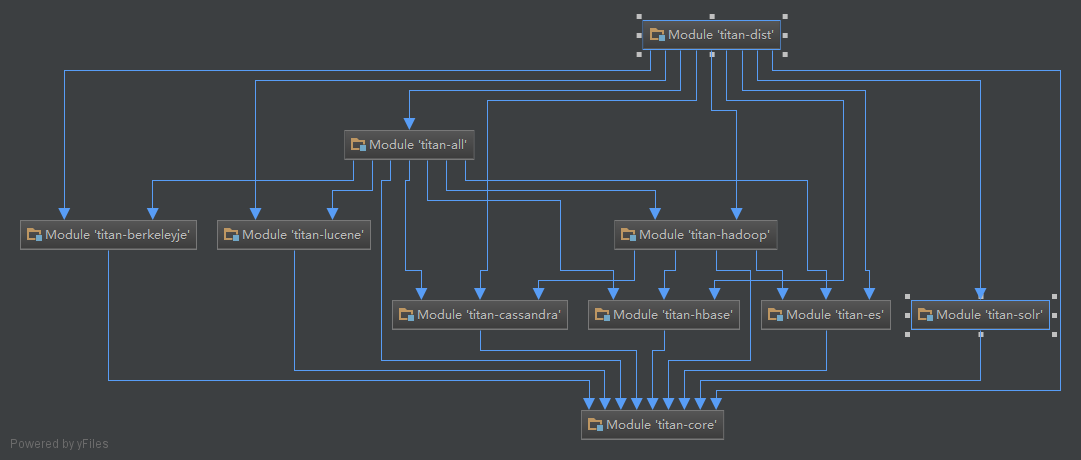
/ titan-2.0.0-hdh4.0.0/conf/es



格式为yaml文件，由于我们使用HDH平台中的ES此处不需要单独配置。

# 二 Titan模块设计

和大多数组件一样Titan采用模块化设计，根据功能将整个项目划为不同的模块，其结构如下图所示：



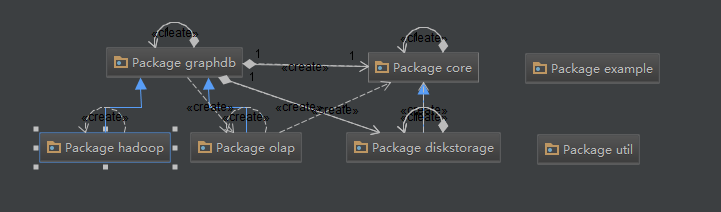
titan-dist负责整个项目的打包发布，titan提供两个打包的路径：titan-dist和titan-parent目录，执行如下命令：

*mvn clean install –Dmaven.test.skip=true*

建议在titan-parent目录下执行打包操作，如果在titan-dist目录下执行打包操作需要titan其他模块已打好包且推到在远端仓库或者本地仓库，支持达成zip和deb/rpm包。目前，titan除了定义Titan具体操作的核心模块titan-core之外还包含以下几个关键模块，提供索引支持的titan-es、titan-solr和titan-lucene，提供存储服务的titan-berkeleyje、titan-cassandre和titan-hbase，以及titan-hadoop。下面将分别介绍其具体作用。

## 2.1 Titan-core

titan-core 为Titan的核心模块定义titan的公共和核心操作包括：属性、接口，持久化、配置、索引以及与底层存储、适配、操作接口，其目录结构如下图所示：



根据具体功能由被封装成:core、diskstorage、example、graphdb、util等包，此外，hadoop位于单独模块。

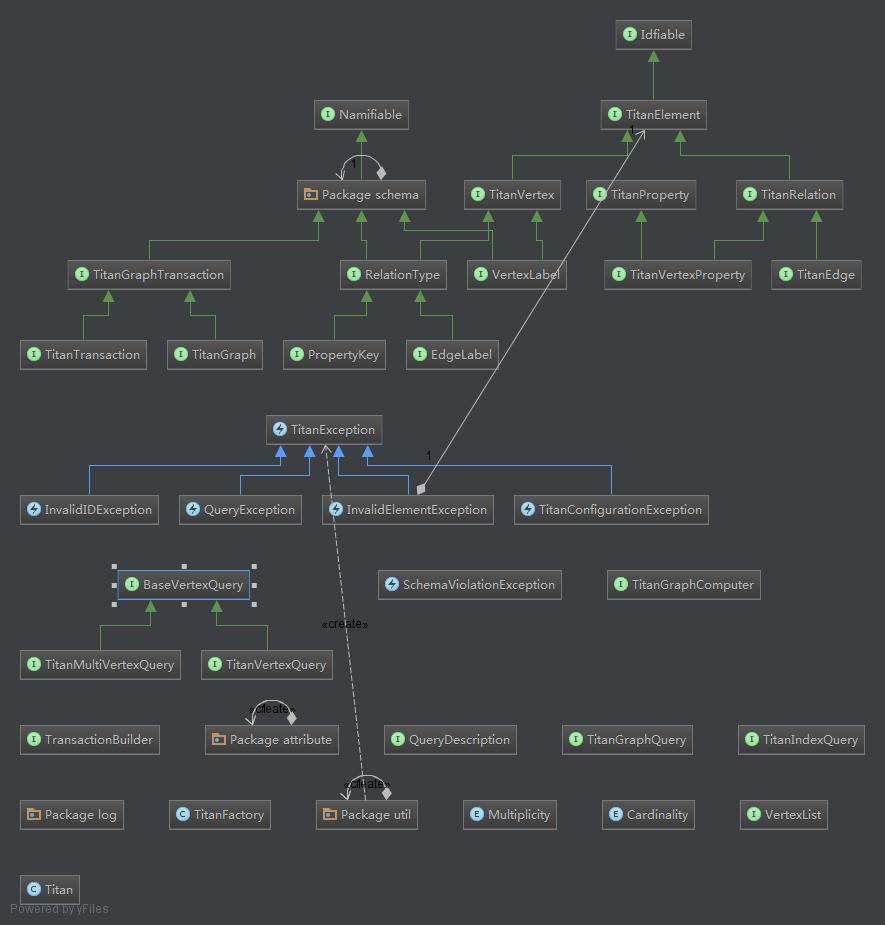
**Core包**

core包定义了操作Titan数据库的核心接口，TitanGraph、TitanTransaction、RelationType、各种Label接口及其工厂接口。

其中的schema包定义了操作schema的接口，包括TitanIndex、TitanGraphIndex、SchemaManager等接口，Schema提供了schema相关的EdgeLabel 、PropertyKey、 RelationType、SchemaElement 、VertexLabel的定义类以及SchemaContainer。每个Titan图都有一个有EdgeLabel、VertexLabel和PropertyKey组成的Schema，Titan Schema可以被显式或隐式定义，官方鼓励用户在应用程序开发过程中明确定义Graph Schema。显式定义的Graph Schema是一个应用程序鲁棒性Graph APP的重要组成部分，可大大提高协同软件开发能力。需要注意的是，TitanSchema可以随时间演进而不会中断正常的数据库操作。扩展Schema不会减慢查询应答速度，而且不会因数据库停机而加大时间消耗。

Schema type如：EdgeLabel， PropertyKey和 VertexLabel在图首次创建时就被关联到相应的元素: Edge、Property和Vertex，为了保证系统的稳定性已分配的Schema type不能为特定元素更改。

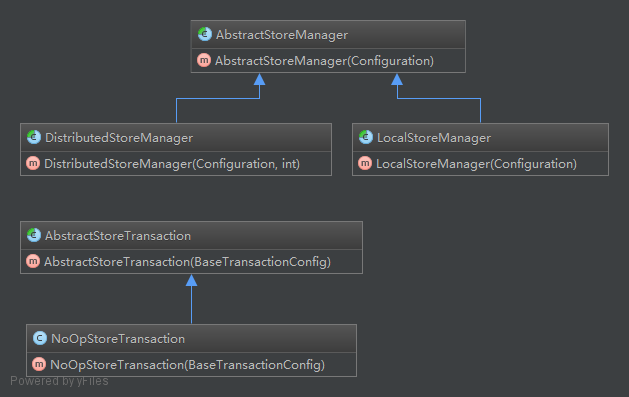
attribute包定义了Geo、Text、Cmp等的枚举以及属性序列化接口，util定义了ManagementUtil、TitanId以及clean工具类。包结构如下图所示：



**Diskstorage包**

Diskstorage包定义了存储相关的操作、各种buffer接口、meta声明以及基本的异常定义。

**common**定义了分布式和本地存储的管理和事务类：



Titan采用多层数据缓存来促进图形的快速遍历。Titan数据缓存分为Transaction-Level Caching 、Database Level Caching和Storage Backend Caching，缓存越接近Transaction，缓存访问速度越快，内存占用和维护开销越高。

**Configuration**定义了配置相关的操作，Titan区分本地和全局配置选项，本地配置选项适用于单个Titan实例，全局配置选项适用于集群中的所有实例。确切的Titan区分以下五个配置选项范围：

* Local：这些选项仅适用于单个Titan实例，并在初始化Titan实例时提供的配置中指定；
* MASKABLE：本地配置文件可以为单个Titan实例覆盖这些配置选项，如果本地配置文件未指定该选项，则其值将从全局Titan集群配置中读取；
* GLOBAL：这些选项始终从集群配置读取，不能在实例的基础上覆盖；
* GLOBAL\_OFFLINE：像GLOBAL一样，但更改这些选项需要重新启动集群，以确保整个集群的值相同；
* FIXED：像GLOBAL一样，但一旦Titan集群被初始化，该值就不能被更改。

当集群中的第一个Titan实例启动时，全局配置选项将从提供的本地配置文件初始化，随后，通过Titan的管理API完成全局配置选项的更改。可通过：g.getManagementSystem()打开一个Titan实例的Handle类，通过下面的示例修改配置：

mgmt = graph.openManagement()

*mgmt.get('cache.db-cache')*

*// Prints the current config setting*

*mgmt.set('cache.db-cache', true)*

*// Changes option*

*mgmt.get('cache.db-cache')*

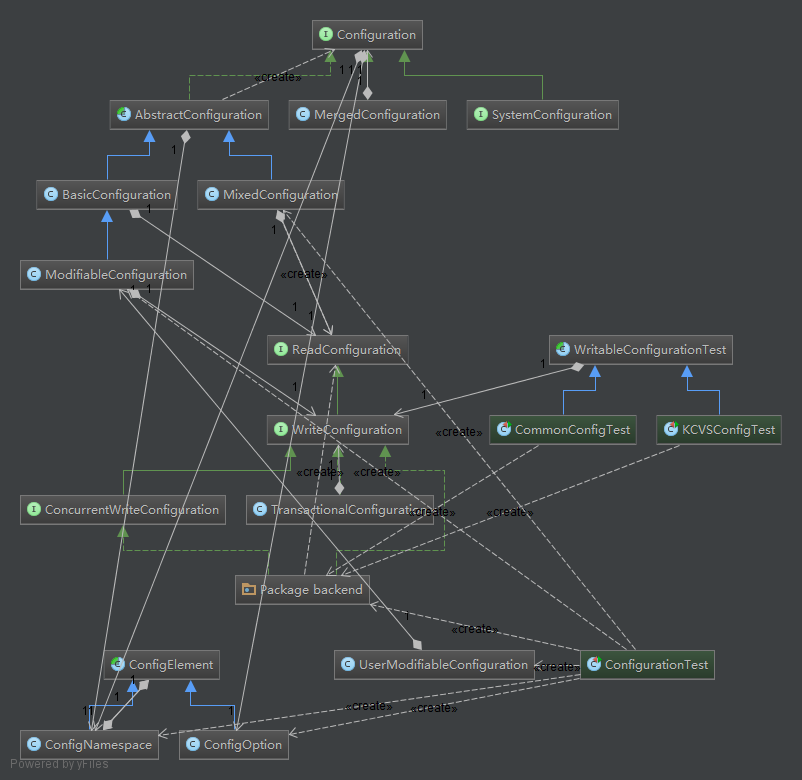
*// Prints 'true'*

*mgmt.commit()*

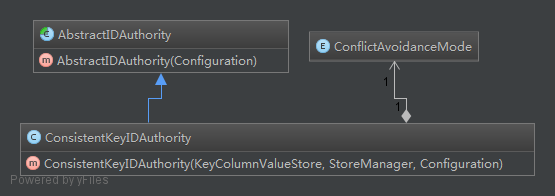
*// Changes take effect*

对于新启动的实例更改配置选项不会影响运行实例，但是更改GLOBAL\_OFFLINE配置选项需要重新启动群集，以便更改对所有实例立即生效。要更改GLOBAL\_OFFLINE配置项，需要按一下步骤执：

* 关闭集群中除Titan之外的所有实例
* 连接到单个实例
* 确保所有运行的事务都已关闭
* 确保没有新事务被启动（即集群必须脱机）
* 打开管理API
* 更改配置选项
* 调用commit将自动关闭图形实例
* 重新启动所有实例

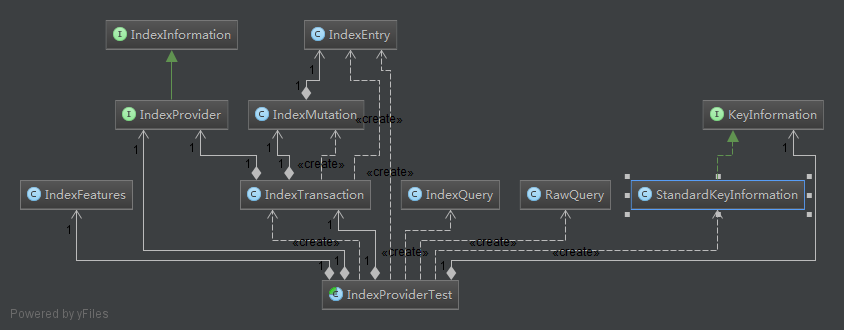


Idmanagement定义了Titan内部id及一致性Key操作：

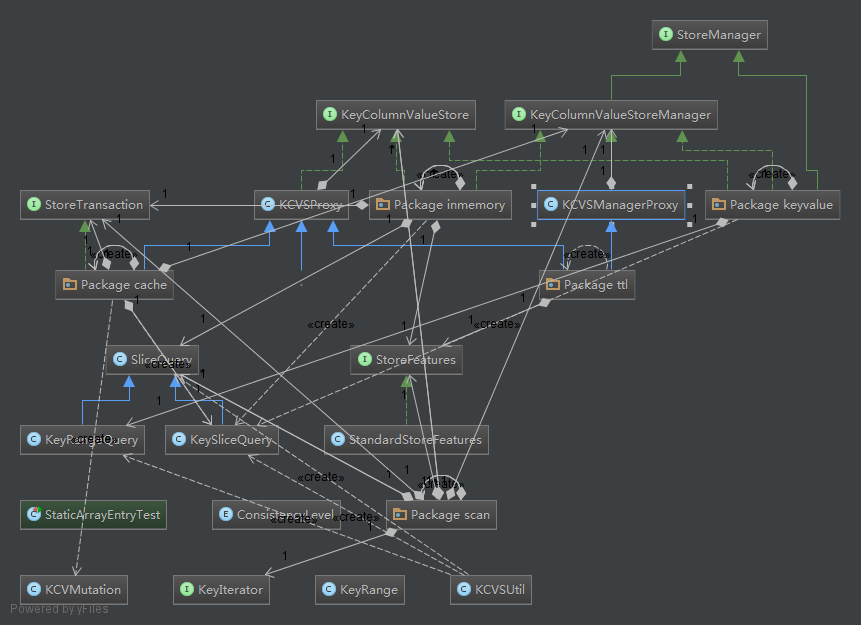


Indexing定义了索引相关操作和接口，TitanIndex是建立在Titan数据库之上的索引，其目的在于便于有效的检索Graph。Titan支持两种不同的索引来加快查询处理：Graph indexes 和 Vertex-centric indexes。大多数图查询从其属性标识的顶点或边的列表中开始遍历，Graph indexes使全局检索操作适用于大图。

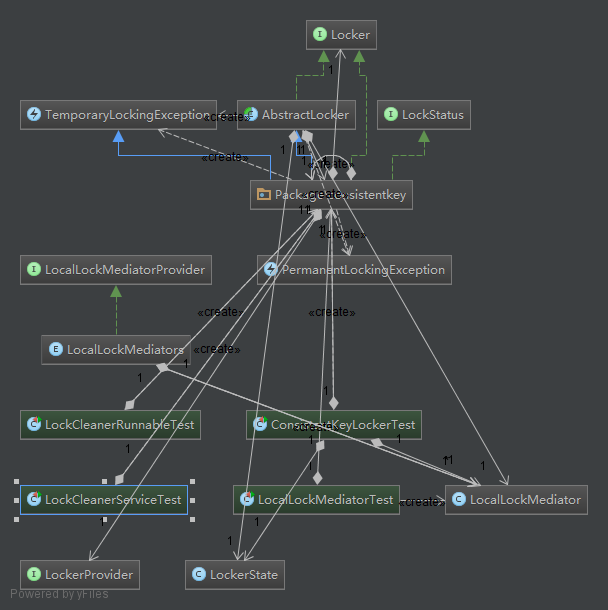
Vertex-centric indexes加速了图中的实际遍历，特别是当遍历具有许多入射边的顶点时。



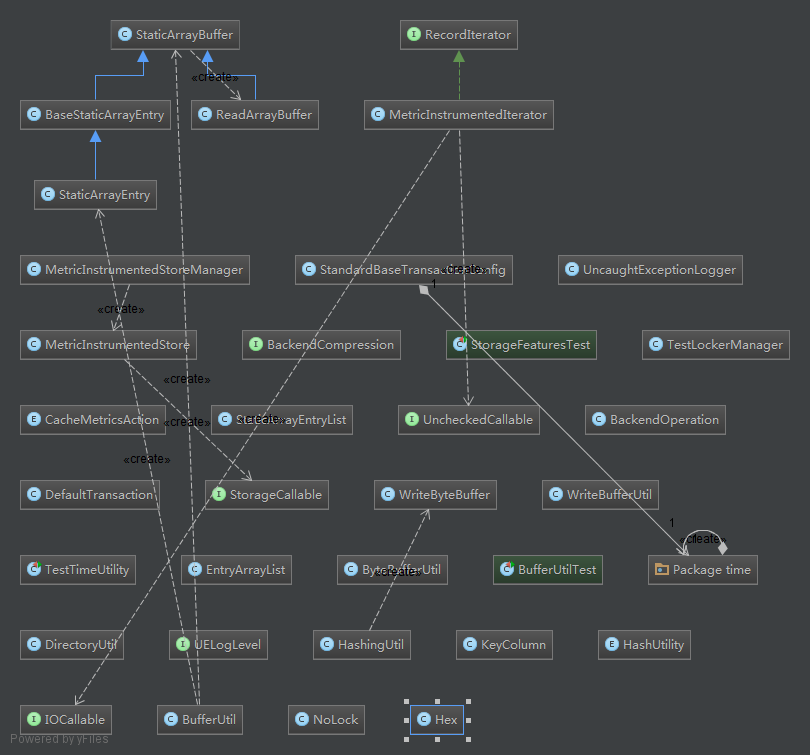
Keycolumnvalue定义各种存储相关操作如：缓存、TTL、磁盘存储管理、scan先关操作和接口：



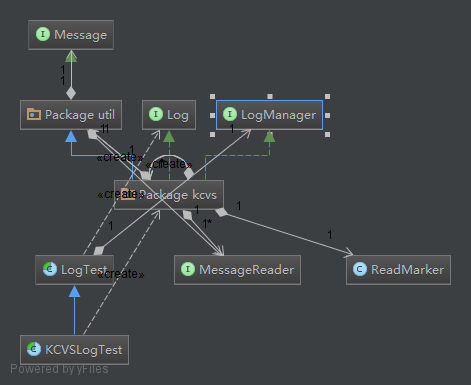
Locking定义了事务先关的“锁”接口和操作：



Util定义了一个工具类及操作：



此外，log定义了log相关操作的接口和定义的消息体：



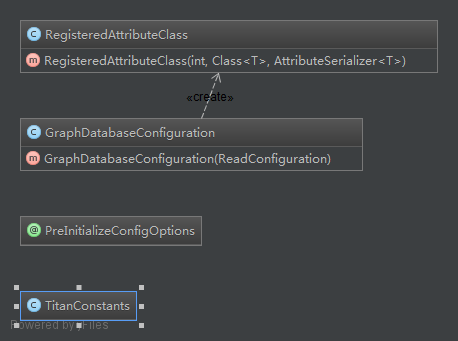
**Example包**

Example包定义了Titan官方文档中的“诸神图”的例子，可以参考其实现学习相关接口的使用。

**Graphdb包**

**Graphdb**定义了titan中关于数据库配置、数据库操作、ID管理、OLAP、查询、schema、Transaction、relations、vertices以及交互接口tinkerpop操作及接口：

Configuration定义注册、图数据库配置操作以及Titan的一些常量数据。



**Databas**e定义了数据库缓存、ID管理、标准索引信息、以及一些序列化、管理操作，此处着重介绍下Titan缓存。Titan采用多层数据缓存来促进图形的快速遍历。Titan数据缓存分为Transaction-Level Caching 、Database Level Caching和Storage Backend Caching，缓存越接近Transaction，缓存访问速度越快，内存占用和维护开销越高。

Transaction-Level Caching

在一个已打开的Transaction中titan维护两个缓存Vertex Cache和Index Cache：

* Vertex Cache：缓存已访问的顶点及其邻接列表（或其子集），以加快同一Transaction中的后续访问，该缓存加速了迭代遍历；
* Index Cache：缓存索引查询的结果，以便后续索引可以直接从内存中获取数据，而不是调用后端存储。

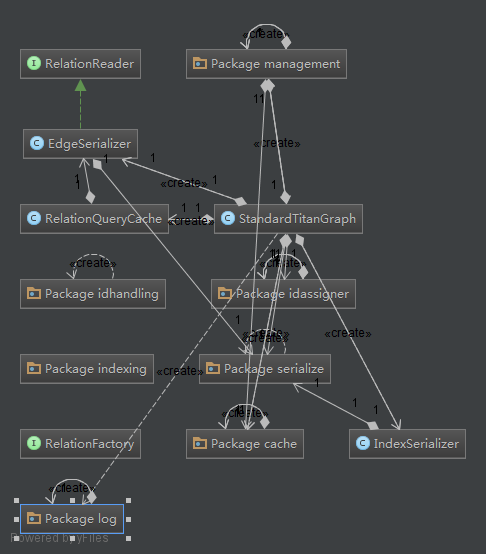
这两者的大小由事务transaction缓存大小决定，事务缓存大小可以通过cache.tx-cache-size或每个事务的基础上通过事务构建器graph.buildTransction（）打开一个事务并使用setVertexCacheSize（int）方法进行配置。

 Database Level Caching

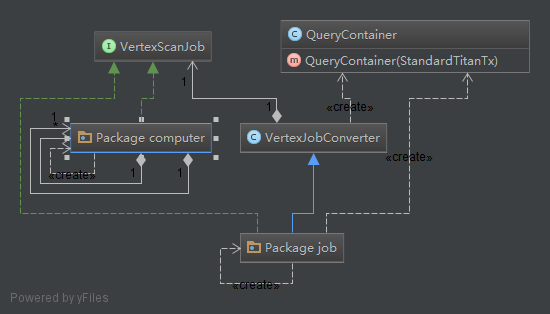
数据库级缓存跨多个transaction，保留邻接列表（或其子集），并超出单个事务的存活时间，由数据库中的所有事务共享。它有比Transaction-Level Caches更高的空间利用率，但其访问速率较慢，与Transaction-Level Caching相反，数据库级别缓存在关闭事务后不会立即过期。 因此，数据库级缓存大大加快了跨越事务处理读取繁重工作负载的图遍历。

Storage Backend Caching

在Titan中每个存储后端都维护自己的数据缓存层。这些缓存能够从数据压缩，数据紧凑性，协调过期等方面为Titan数据库的性能提升带来好处，此外，这些通常是被保留在堆中，这意味着可以使用大型缓存而不会遇到垃圾回收问题。虽然这些缓存明显大于数据库级缓存，但它们访问速度也较慢。

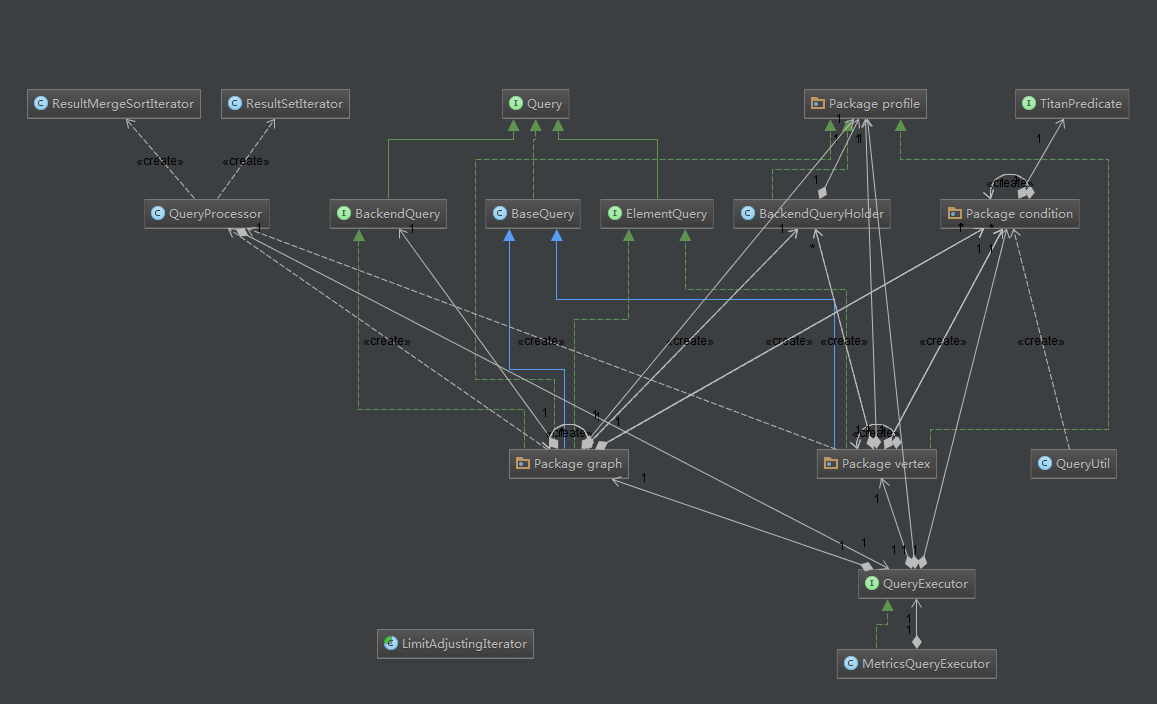


**Olap**定义job、计算先关操作:

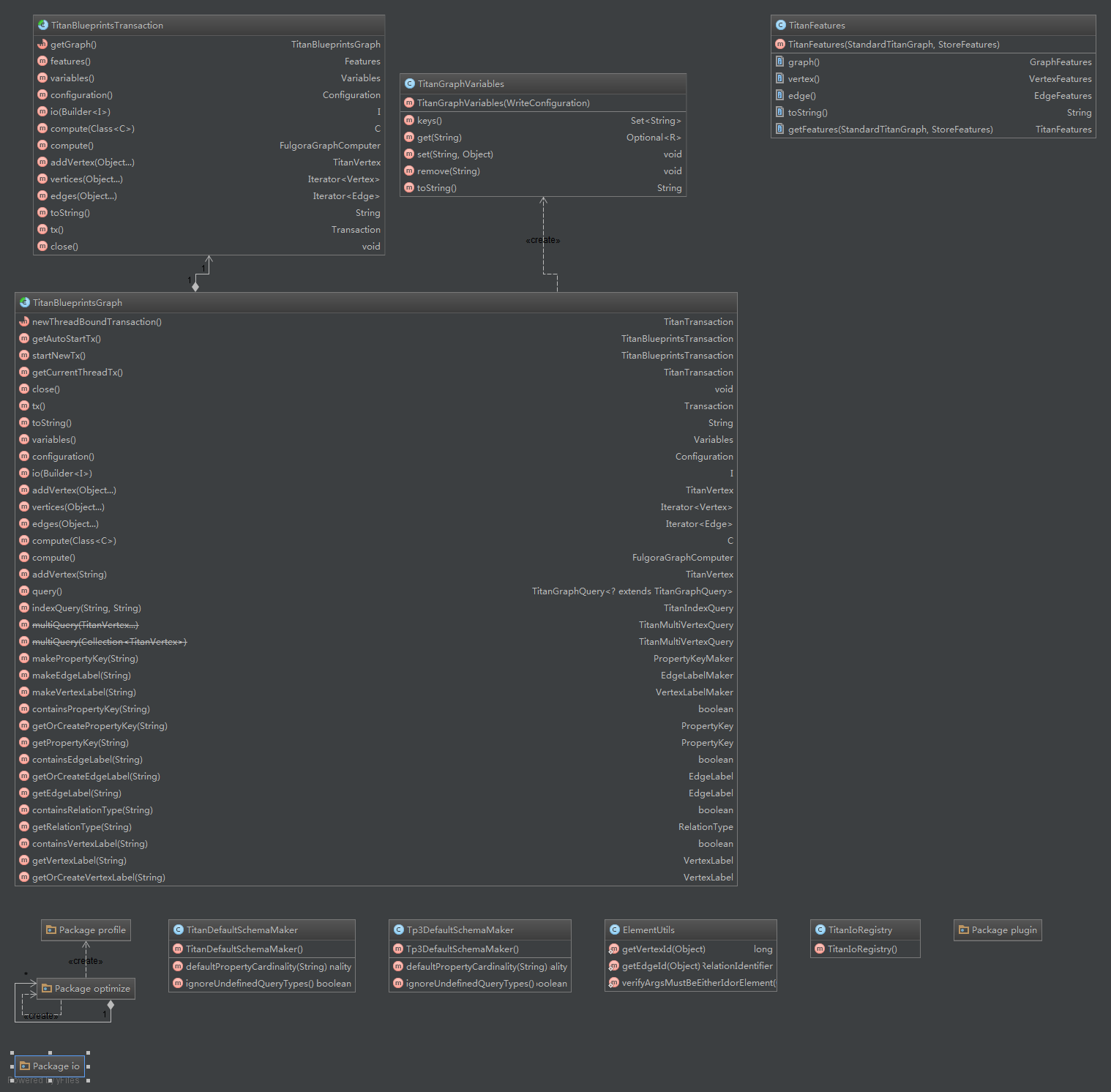


**Query**定义了查询先关的接口以及结果处理类，Titan把Gremlin作为其查询语言，用于从图中检索数据和修改数据。Gremlin是一种面向路径的语言，简洁地表达了复杂的图遍历和变异操作，同时Gremlin也是一种功能性语言，其中遍历运算符链接在一起以形成路径样表达式，比如：

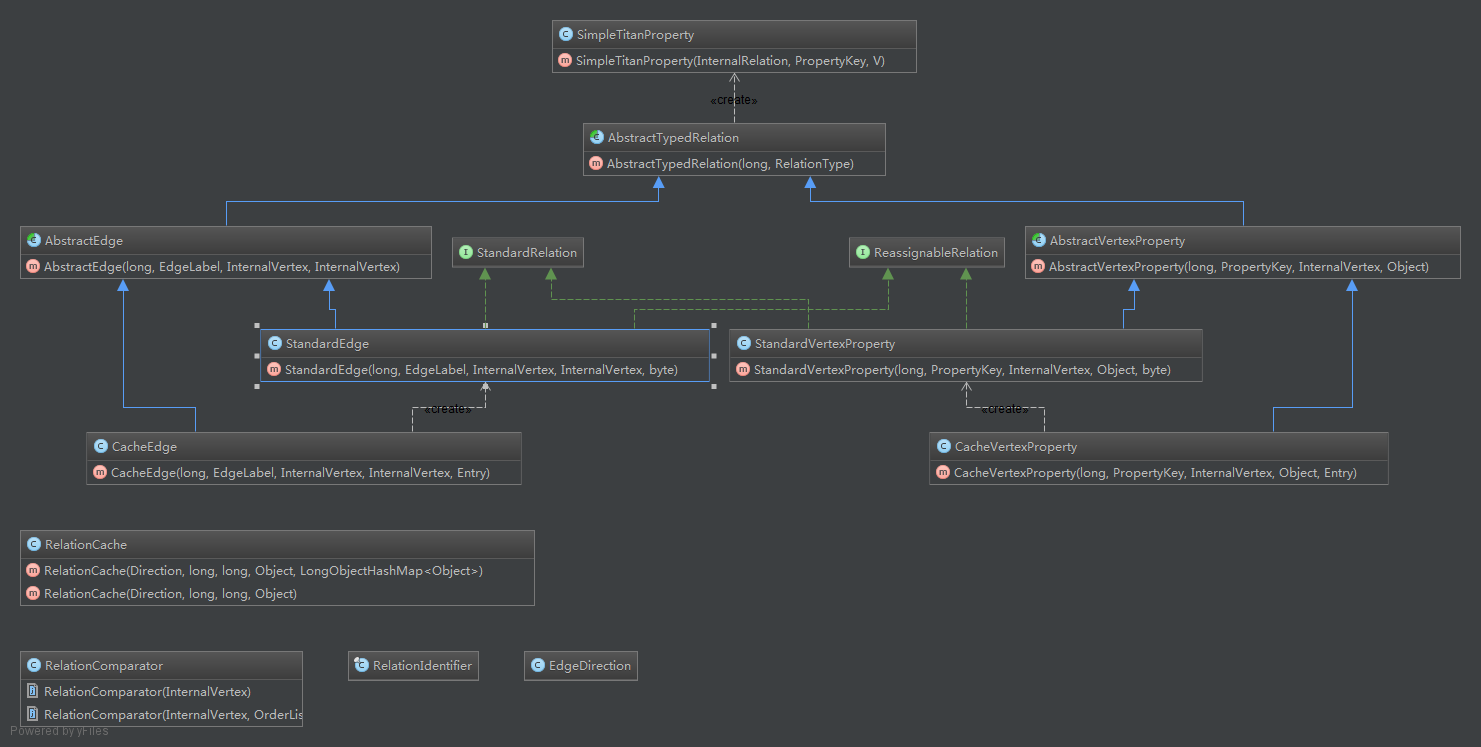
*Vertex grandSonOfSaturn = traversalSource.V(vertex).in("father").in("father").next();*



Gremlin独立于Titan开发，并由大多数图形数据库支持，通过Gremlin查询语言在Titan之上构建应用程序，用户可以避免供应商封锁，因为它们的应用程序可以迁移到支持Gremlin的其他图形数据库。

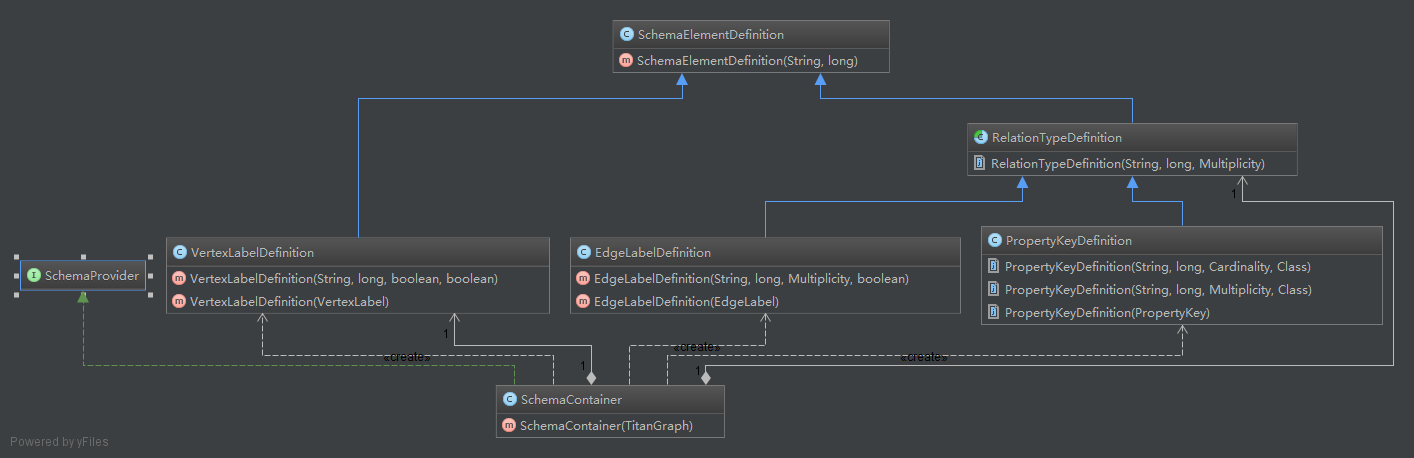


**Relations**定义顶点（Vertex）、边（Edge）等的关系属性、定义及相关操作：

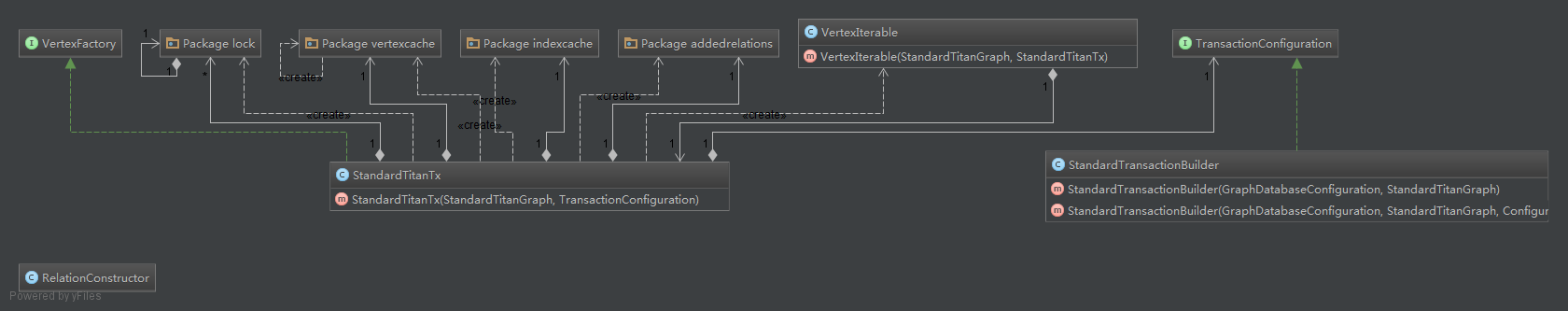


**Schema**提供了schema相关的EdgeLabel 、PropertyKey、 RelationType、SchemaElement 、VertexLabel的定义类以及SchemaContainer。每个Titan图都有一个有EdgeLabel、VertexLabel和PropertyKey组成的Schema，Titan Schema可以被显式或隐式定义，官方鼓励用户在应用程序开发过程中明确定义Graph Schema。显式定义的Graph Schema是一个应用程序鲁棒性Graph APP的重要组成部分，可大大提高协同软件开发能力。需要注意的是，TitanSchema可以随时间演进而不会中断正常的数据库操作。扩展Schema不会减慢查询应答速度，而且不会因数据库停机而加大时间消耗。

Schema type如：EdgeLabel， PropertyKey和 VertexLabel在图首次创建时就被关联到相应的元素: Edge、Property和Vertex，为了保证系统的稳定性已分配的Schema type不能为特定元素更改。



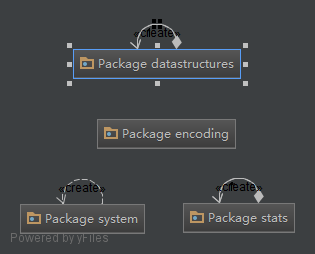
Transaction定义了Transaction相关的一些操作，包括一个AddedRelationsContainer接口及其实现，Index和Vertex缓存和缓存替换策略 的定义，以及保证transaction操作的原子性的锁操作：



其他还包含一个index、vertex type属性及接口和操作的包types和一个工具包util以及缓存、删除、加载vertex的vertices包。

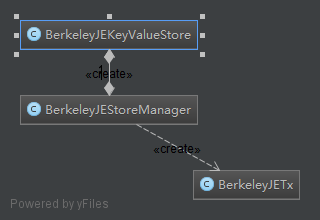
**Util包**

Util包定义了titan的数据结构、编解码方式以及一些系统属性及操作：



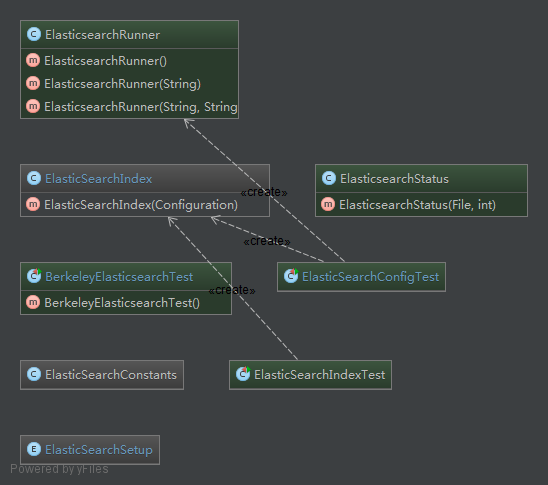
## 2.2 titan-berkeleyje

titan-berkeleyje定义了底层存储选择berkeleyje时对keyvalue存储、存储管理以及Transaction的一些操作：

****

## 2.3 titan-es

Titan-es定义了对es索引的操作，包括配置文件加载、索引创建及检索：

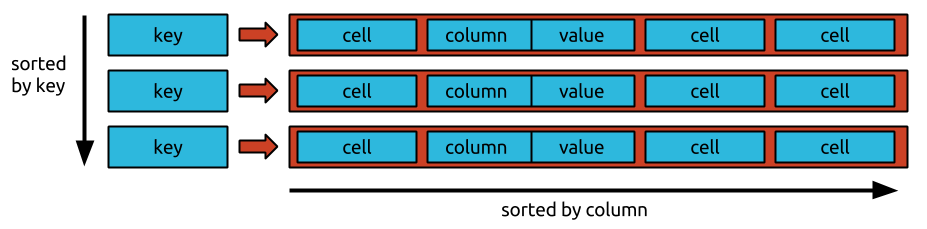
****

此外几个模块包括：hbase、hadoop、solr、lucene都是对相关组件API接口的一些操作，再次不再赘述。

# 三Titan 数据模型

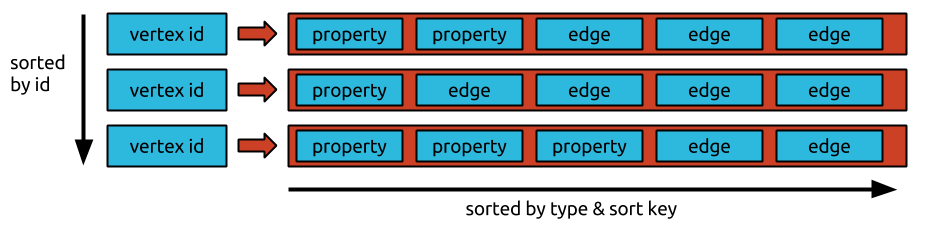
Titan以邻接列表的形式存储图数据，也就是说，一张图存储为Vertex邻接列表的集合。Vertex的邻接列表包含Vertex及其关联的Edges和属性。通过这种存储方式，Titan可确保将所有顶点的入口edge和属性紧凑地存储在存储后端中，从而加速遍历，这样做的缺点是每个edge需要存储两次，分别是作为vertex的“头”和“尾”存储。此外，Titan按照排序的“KEY”定义每个顶点的相邻列表，并按顺序对边缘标签进行排序，排序使得可以使用顶点中心索引来有效地检索邻接列表及其子集。Titan支持任何大数据存储的组件来存储图的邻接列表。

## 3.1 大数据模型



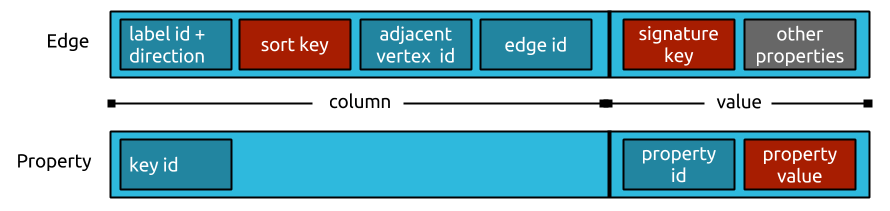
在BigTable数据模型下，每个表都是一组行，每一行都由一个键唯一标识，每行由任意（大但是有限的）单元组成，单元格由列和值组成，单元格由给定行中的列唯一标识。 BigTable模型中的行称为“宽行”，因为它们支持大量单元格，并且这些单元格的列不必按照关系数据库中的要求进行定义。Titan对BigTable数据模型有额外要求：单元格必须按其列排序，并且必须通过列索引指定的单元格子集（例如使用索引结构，skiplist或二进制搜索）进行有效检索。

## 3.2 Titan数据模型



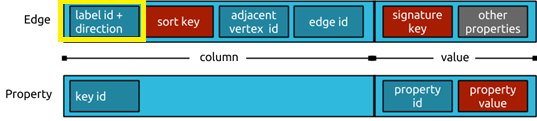
Titan将每个vertex的邻接列表存储在后端存储中的一行，采用64bit的唯一vertexid标示每一个vertex，作为存储的行健，每一edge和property存储在单独的cell中，并允许有效的插入和删除。因此，后端存储中每行允许的单元格的最大数量也是titan可以支持的最大vertex数量。如果存储后端支持按主键排序，则邻接列表将通过vertexid排序，而Titan也可以通过分配vertexid进行有效地分区。 每个vertex被关联上唯一的id，使得经常共存的vertex具有小的绝对差异的id。

## 3.3 Edge数据模型

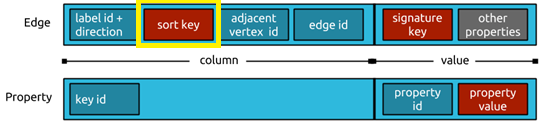


每个Edge及其property被存储在被其邻接vertex标示的行的一个cell中，这些数据会被序列化使得列的字节顺序代表edgelabel的排序，为了保证每个edge/cell占用的存储空间尽可能小采用可变长度的id编码模式并压缩对象序列化数据。如上图所示：深蓝色的框表示采用可变长度编码方案编码的数字，以减少它们消耗的字节数；红色框表示一个或多个属性值（即对象），这些属性值是在相关联的属性键中引用的压缩元数据进行序列化的； 灰色框表示未压缩属性值（即序列化对象）。

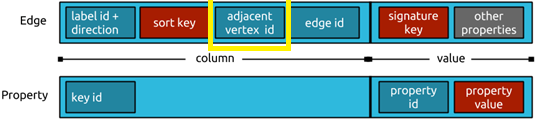
Edge序列化从edge的唯一label开始，id的最后一个是一个offset代表该edge是一个incoming edge还是outcoming edge，如下图标示：



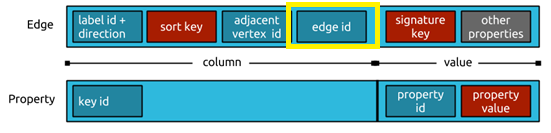
接下来存储的是已排序的属性值，排序键用边缘标签定义，因此排序键对象元数据可以引用到边缘标签，



之后，存储相邻顶点的id：



Titan不存储实际的vertexid，而是存储与拥有该邻接列表的vertex的id之间的差异。Vertexid之后是edgeid：

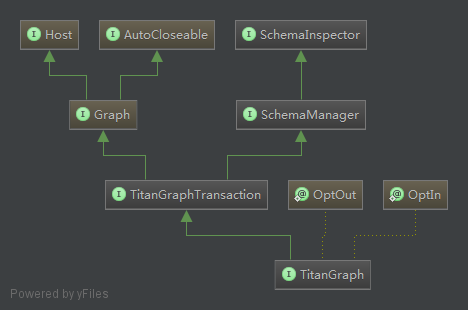


每一个edge都被分配唯一的edgeid，作为该cell的列值，之后是签名属性和其他未序列化的属性。

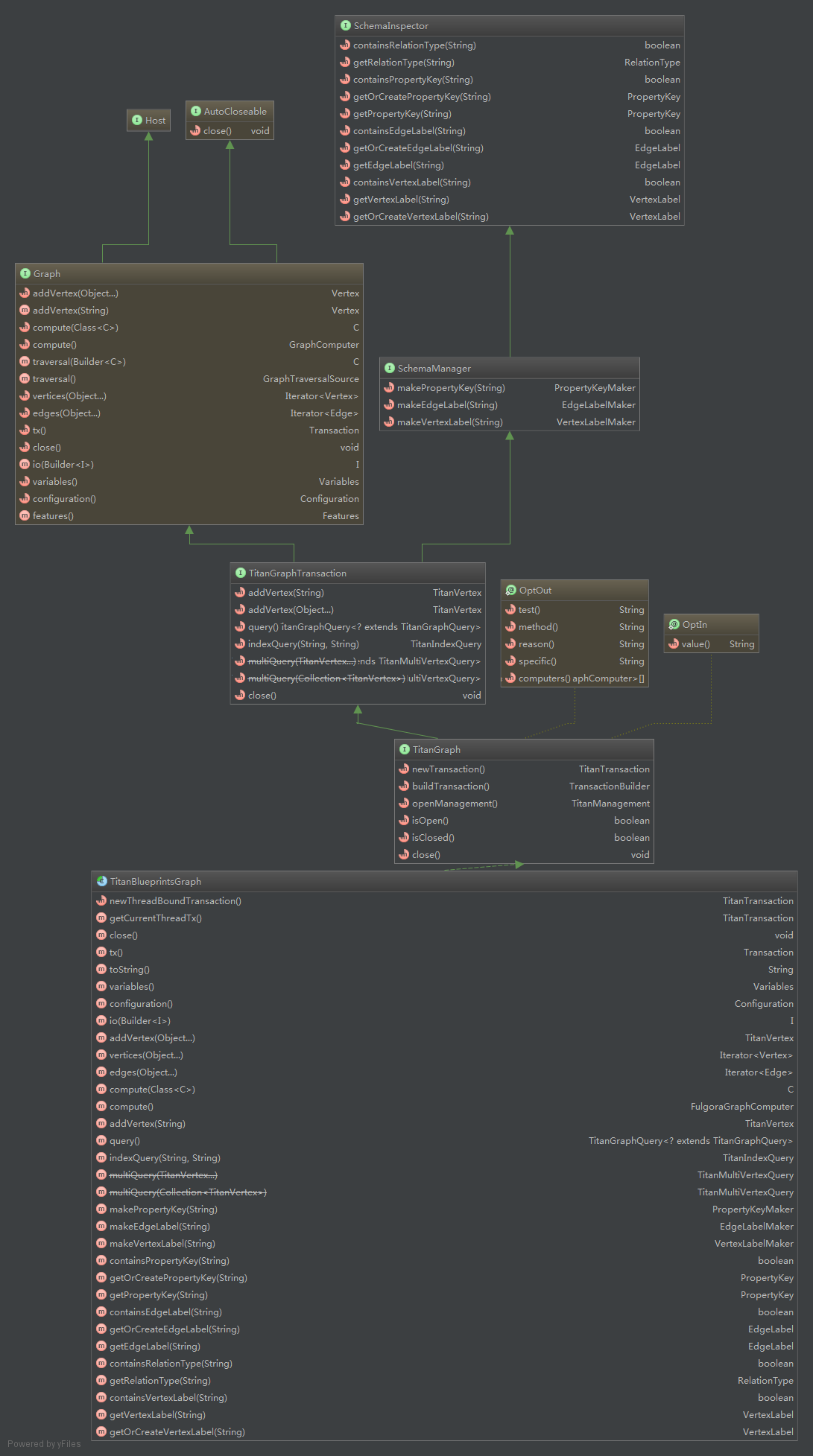
属性的序列化表示更简单，只包含属性的列中的键ID。 属性id和属性值存储在该值中。 如果属性键定义为list（），则属性id也存储在列中。

# 四 核心操作分析

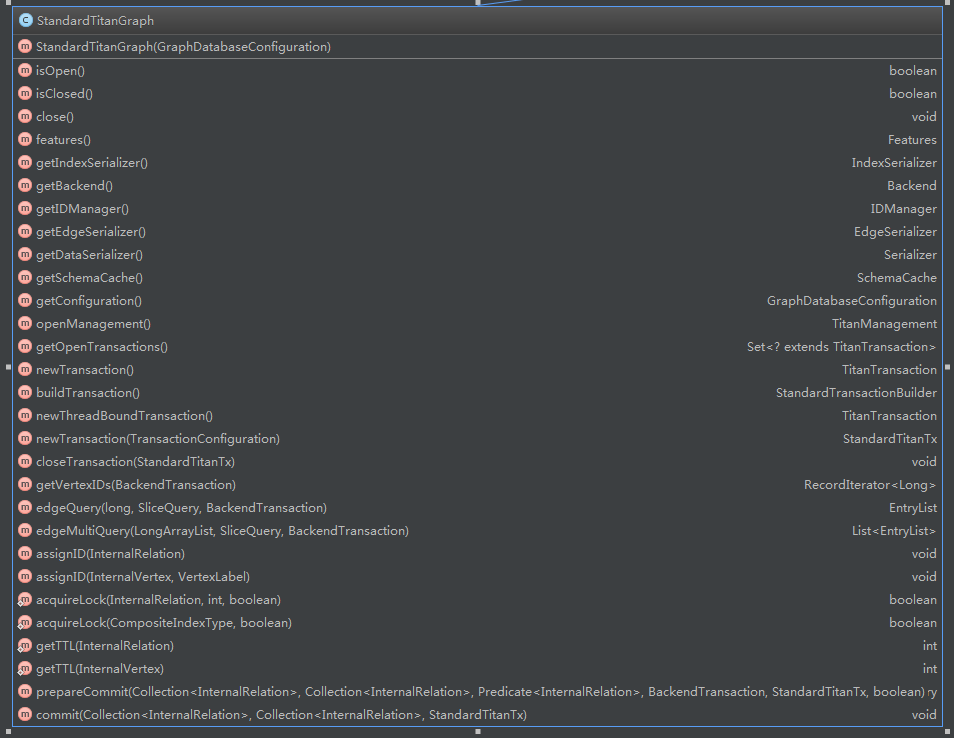
## 4.1 TitanGraph

操作Titan数据库的入口是一个TitanGraph实例，根据config配置数据实例化，该conf在编码过程中传入也可读取本地的配置文件加载： 

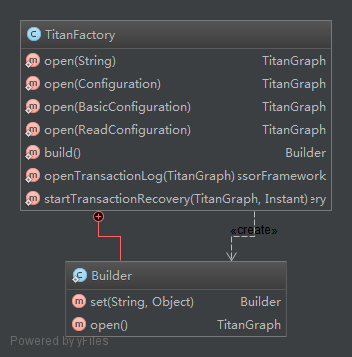
它有两个实现，一个直接实现TitanGraph接口的TitanBlueprintsGraph：



封装了关于Transaction和Schema相关的操作。包括图数据库基本的属性如：顶点、边、标签、属性等的操作。另外一个又继承自TitanBlueprintsGraph的实现StandardTitanGraph：



通过调用TitanFactory的open()函数可返回TitanGraph实例：



TitanFactory根据传入的参数提供四种实例化TitanGraph的方法，最常用的是通过内部类Builder对象传入参数如：

*TitanFactory.Builder config = TitanFactory.build();*

*config.set("storage.backend", "hbase");*

*config.set("storage.directory", directory);*

*……*

*TitanGraph graph = config.open();*

此外，TitanFactory提供基于配置文件的：

*public static TitanGraph open(String shortcutOrFile) {return open(getLocalConfiguration(shortcutOrFile));}*

和基于配置对象的：

*public static TitanGraph open(Configuration configuration) {*

*return open(new CommonsConfiguration(configuration));}*

*public static TitanGraph open(BasicConfiguration configuration) {*

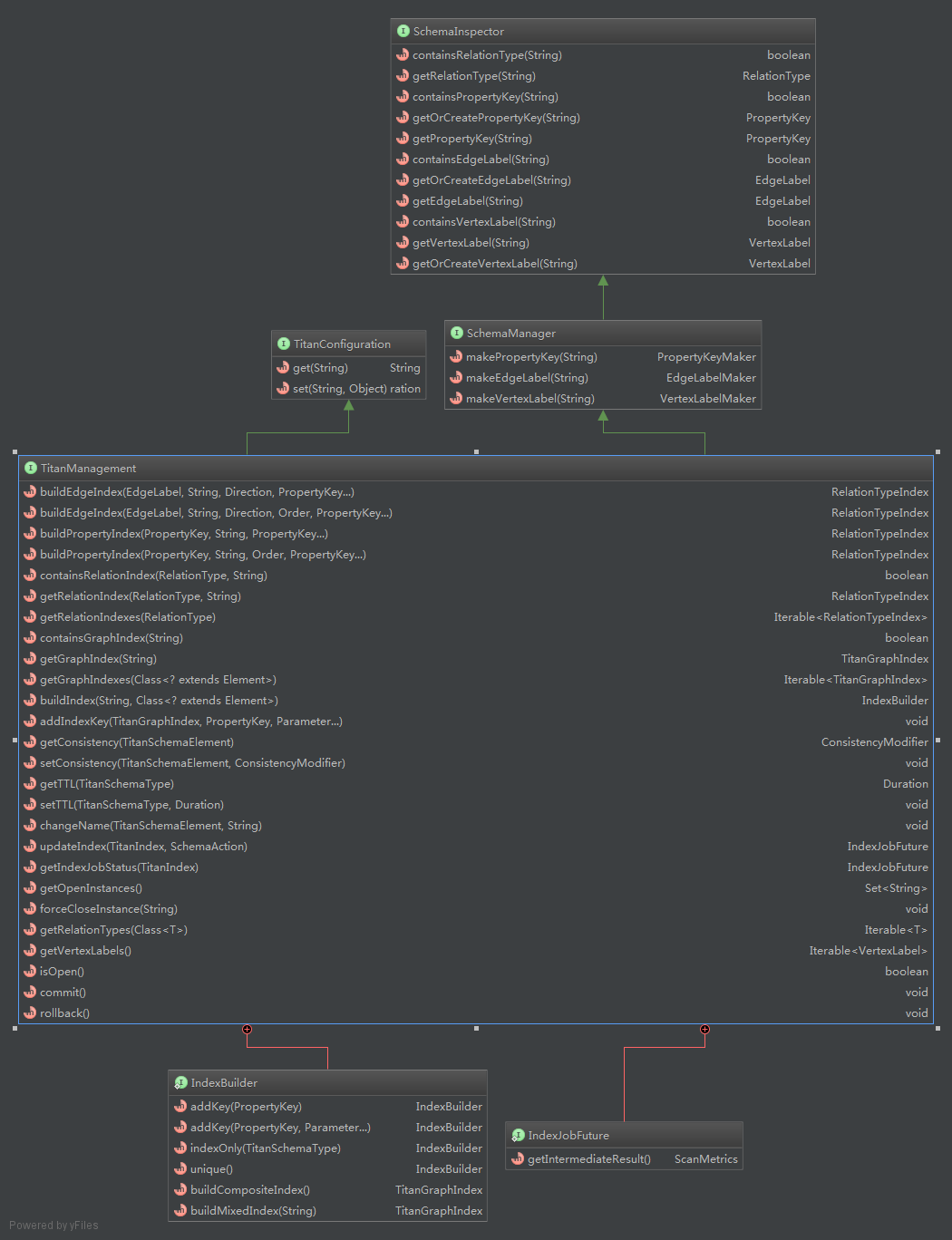
*return open(configuration.getConfiguration());}*

*public static TitanGraph open(ReadConfiguration configuration) {*

*return new StandardTitanGraph(new GraphDatabaseConfiguration(configuration));}*

## 4.2 TitanManagement

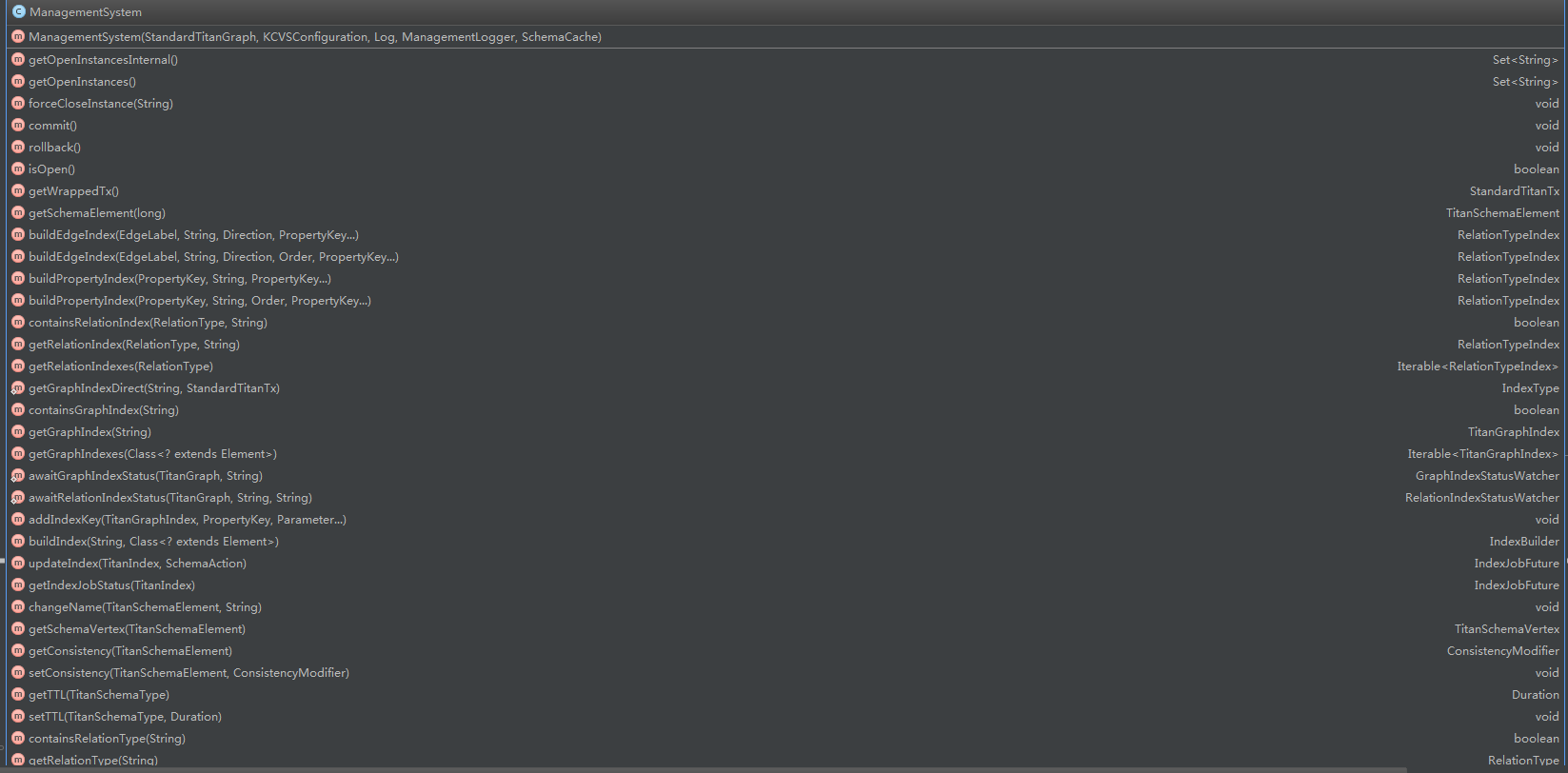
TitanManagement接口提供了定义、更新、检查Titan Schema的操作，接口定义如下：



由于TitanManagement包装了一个TitanTransaction对象，因此其copy了很多与Schema定义、检查相关的函数。TitanManagement的行为更像是一个事务，因为它打开了一个事务范围，用于读取Schema并对其进行更改，因此，它需要通过调用commit()或rollback()方法显式关闭。TitanManagement可通过TitanGraph的openManagement()打开，其提供如下种类的操作：

1. Schema Types，查看，更新和创建顶点标签，vertex labels、edge labels和 property keys；
2. Relation Type Index，在edge labels和property keys上查看并创建以顶点为中心的索引；
3. Graph Index，查看并创建图形索引，用于有效的元素检索；
4. Consistency Management，一致性管理设置但Schema元素的一致性管理策略。

通过TitanGraph的openManagement()函数可返回一个TitanManagement实现类ManagementSystem的对象，其不仅实现接口基本函数还扩充部分函数：



其操作如下示例代码：

*TitanManagement mgmt = graph.openManagement();****//获取TitanManagement 对象***

*final PropertyKey name = mgmt.makePropertyKey("name").dataType(String.class).make();* ***//创建PropertyKey***

*TitanManagement.IndexBuilder nameIndexBuilder = mgmt.buildIndex("name", Vertex.class).addKey(name);*  ***//IndexBuilder，index创建及管理***

*if (uniqueNameCompositeIndex) nameIndexBuilder.unique();*

*TitanGraphIndex namei = nameIndexBuilder.buildCompositeIndex();//Graph index创建*

*mgmt.setConsistency(namei, ConsistencyModifier.LOCK);//Lock*

*final PropertyKey age = mgmt.makePropertyKey("age").dataType(Integer.class).make();*

*if (null != mixedIndexName)*

*mgmt.buildIndex("vertices",Vertex.class).addKey(age).buildMixedIndex(mixedIndexName);*

*final PropertyKey time = mgmt.makePropertyKey("time").dataType(Integer.class).make();*

*final PropertyKey reason = mgmt.makePropertyKey("reason").dataType(String.class).make();*

*final PropertyKey place = mgmt.makePropertyKey("place").dataType(Geoshape.class).make();*

*if (null != mixedIndexName)*

*mgmt.buildIndex("edges",Edge.class).addKey(reason).addKey(place).buildMixedIndex(mixedIndexName);*

*mgmt.makeEdgeLabel("father").multiplicity(Multiplicity.MANY2ONE).make();//edgelablel 创建*

*mgmt.makeEdgeLabel("mother").multiplicity(Multiplicity.MANY2ONE).make();*

*EdgeLabel battled = mgmt.makeEdgeLabel("battled").signature(time).make();*

*mgmt.buildEdgeIndex(battled, "battlesByTime", Direction.BOTH, Order.decr, time);*

*mgmt.makeEdgeLabel("lives").signature(reason).make();*

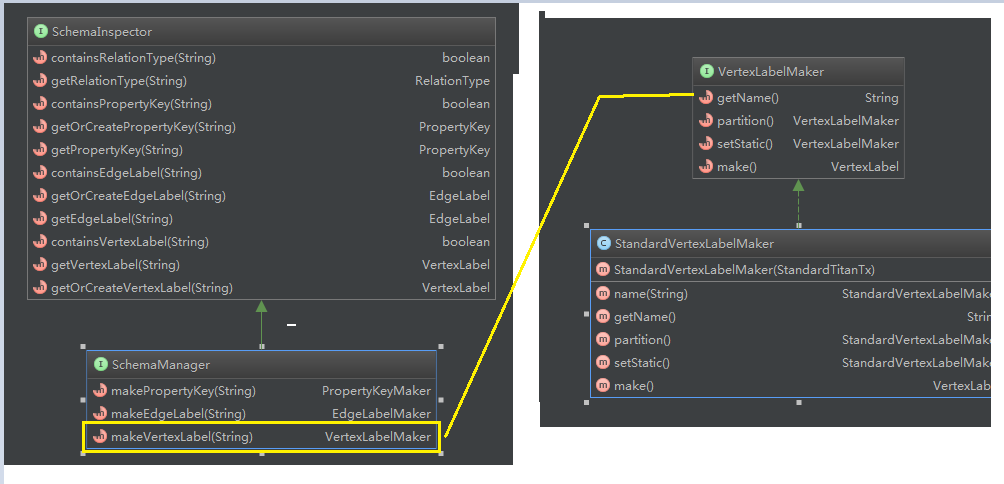
*mgmt.makeVertexLabel("titan").make();//VertexLabel*

*mgmt.makeVertexLabel("location").make();*

*mgmt.makeVertexLabel("god").make();*

*mgmt.commit();*

*VertexLabel、PropertyKey、EdgeLabel的创建由SchemaManager接口提供的makexxx()接口实现，返回对应接口的实例，其创建流程相似，下面我们以VertexLabel的创建*为例大概讲下具体操作：

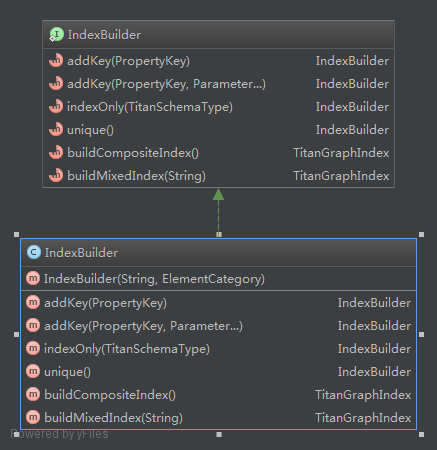
*mgmt.makeVertexLabel("location").make();*

*TitanManagement mgmt = graph.openManagement()* 获取的是一个ManagementSystem对象，该对象实现了TitanManagement接口，而TitanManagement接口继承自SchemaManager接口，因此其有一个返回VertexLabelMaker的接口实现StandardVertexLabelMaker实例，在该接口中定义了一个make()函数返回一个VertexLabel实现的实例。

TitanGraphIndex被称为图索引，加载到图上以便能够通过其properties有效地检索图元素。通过以下命令：

*TitanManagement.IndexBuilder nameIndexBuilder = mgmt.buildIndex( "name", Vertex.class).addKey(name);*

返回一个IndexBuilder实例，为该Graph添加上索引。

**

然后通过：

*TitanGraphIndex namei = nameIndexBuilder.buildCompositeIndex();//Graph index创建*

*mgmt.setConsistency(namei, ConsistencyModifier.LOCK);*

创建TitanGraphIndex，该索引可以是复合索引或混合索引。此外，TitanGraphIndex支持基于现有索引创建，可通过以下接口实现：

*TitanManagement.getGraphIndex(String)} 或者TitanManagement.getGraphIndexes(Class).*

可通过 mgmt.setConsistency(namei, ConsistencyModifier.LOCK)为TitanGraph创建一致性管理级别。

关于Relation Type Index的创建则通过以下接口实现：

*public RelationTypeIndex buildEdgeIndex(EdgeLabel label, String name, Direction direction, Order sortOrder, PropertyKey... sortKeys);*

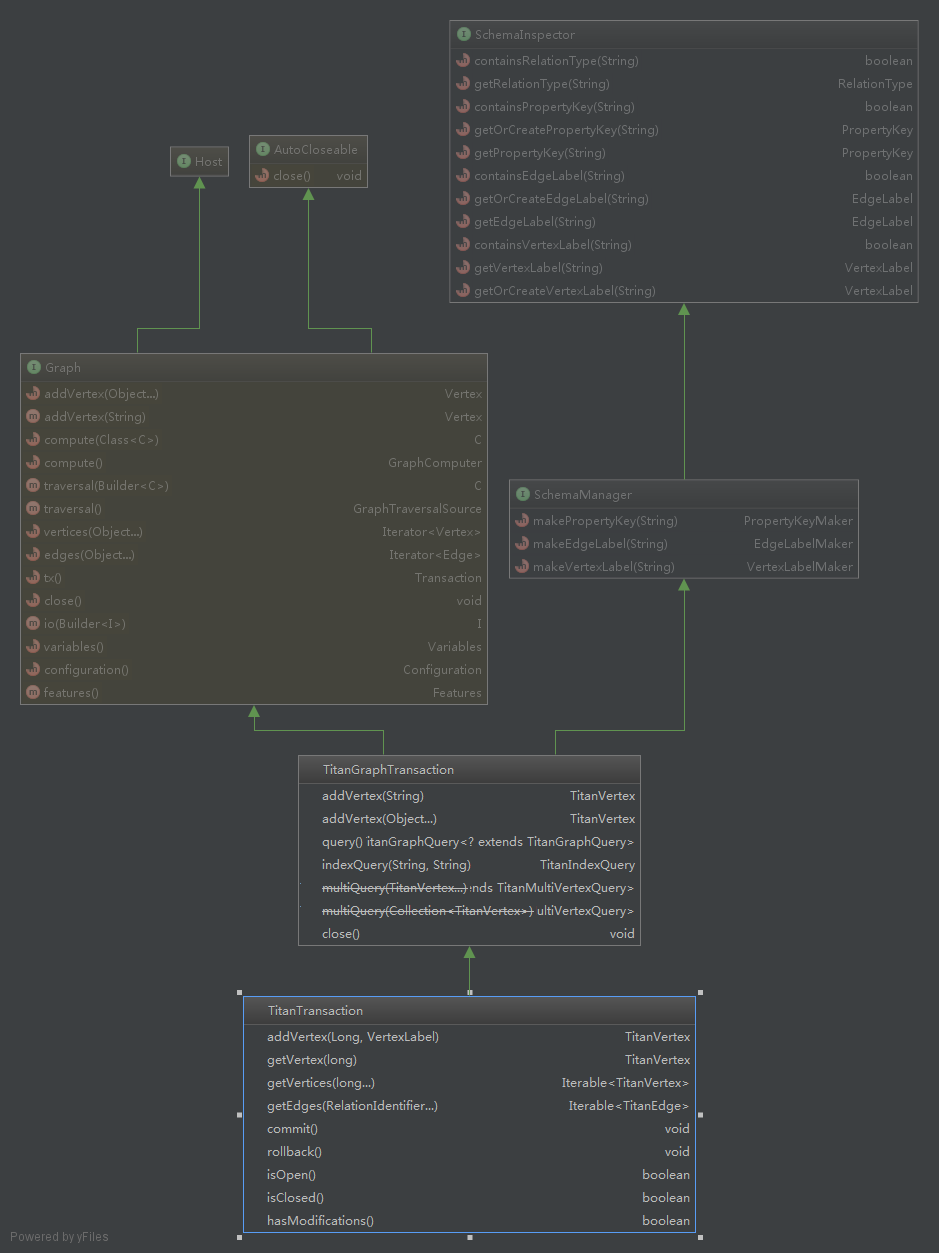
*public RelationTypeIndex buildPropertyIndex(PropertyKey key, String name, PropertyKey... sortKeys);*

## 4.3 TitanTransaction

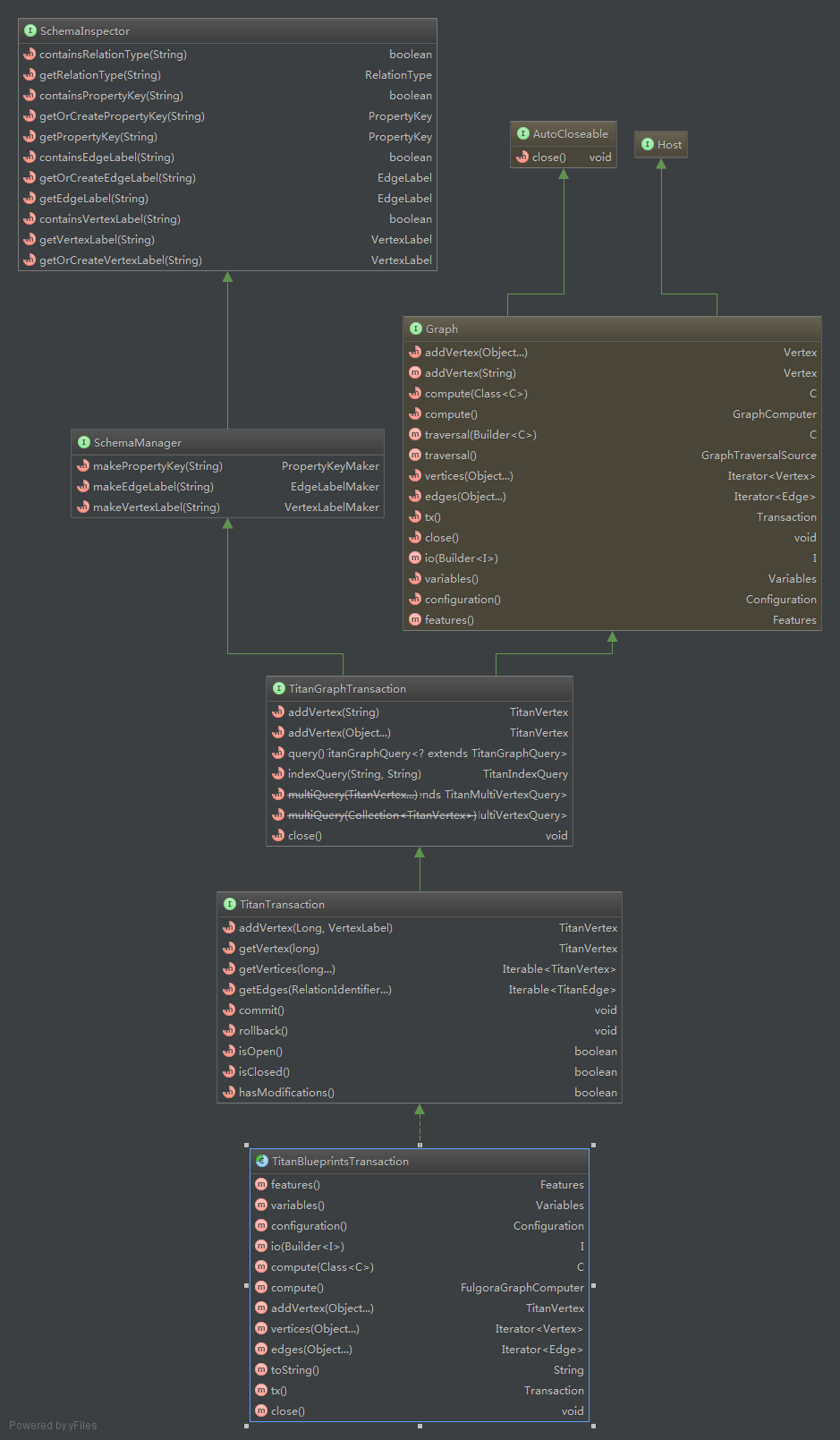
TitanTransaction定义了TitanGraph的事务上下文，几乎所有与Titan的交互操作都与特定Transaction相关联，TitanTransaction对多个线程并发使用是安全的。所有TitanGraph实例上的方法，如graph.v（...）和graph.commit（）会执行ThreadLocal 尝试检索或创建与调用线程相关联的Transaction，调用者可以选择放弃ThreadLocal Transaction管理，通过graph.newTransaction（）返回对Transaction对象的引用，该方法定义了读/写图形数据和提交或回滚的方法。Titan数据库不会严格遵守ACID事务机制。

TitanGraph是一个事务数据库，所有与图的交互操作都有TitanTransaction来缓解，所有Vertex和Edge检索都通过图形事务引导，将所有这些检索，创建和删除捆绑到一个事务中。Titan的图事务类似于数据库事务，通过存储后端来配置隔离级别和ACID支持，这意味着存储后端支持的任何级别的隔离都由Graph Transaction作为镜像。Graph Transaction支持以下操作，也就是说所有关于下面的操作将通过TitanTransaction提供给的函数来实现：

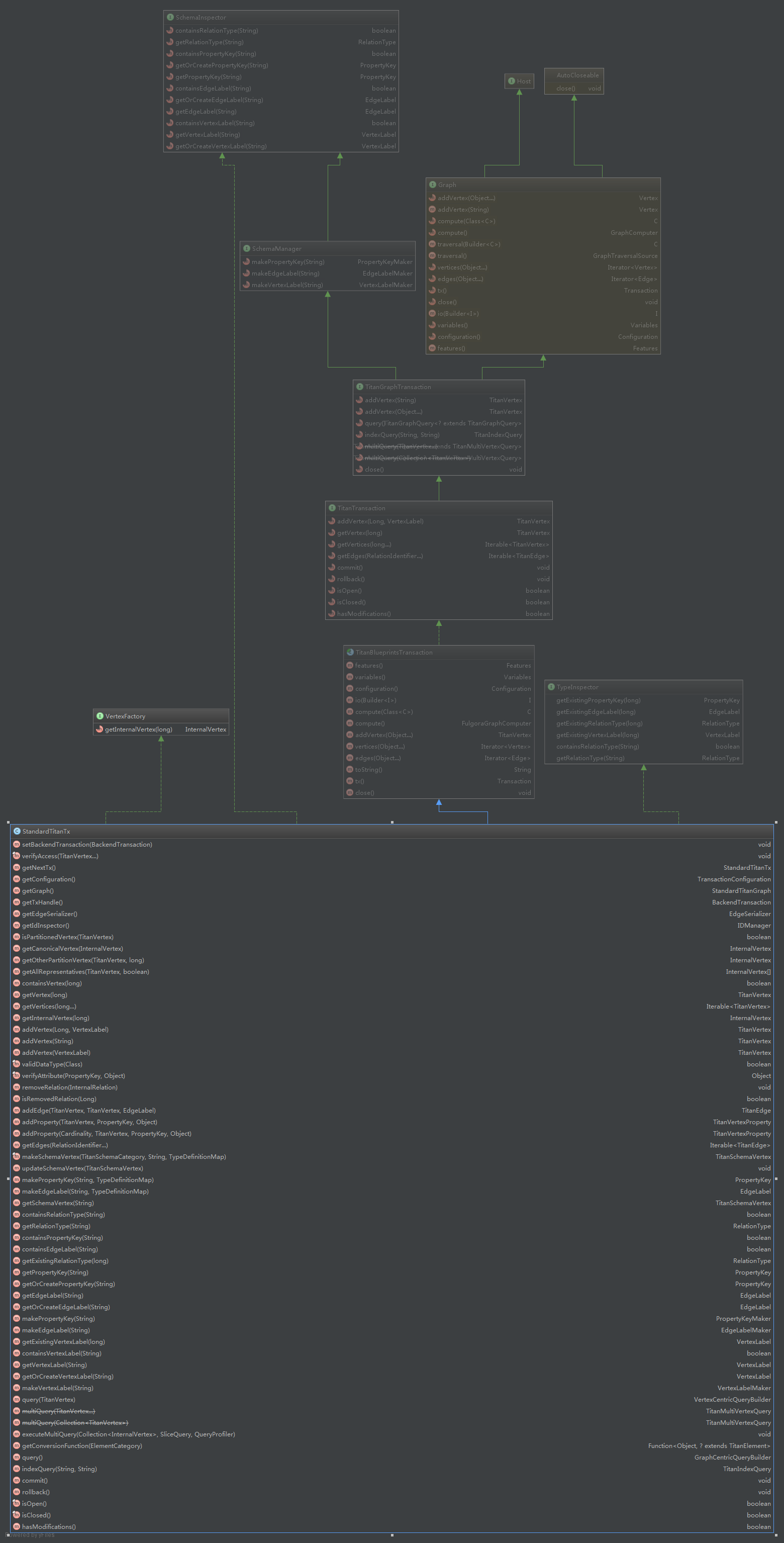
1. 创建vertices， properties 和 edges；
2. 创建types；
3. 基于索引的vertices检索；
4. 查询edges和vertices；
5. 终止transaction的提交。

****

TitanTransaction接口定义如下，TitanTransaction接口继承自TitanGraphTransaction接口，同时有两种实现，分别是直接实现TitanBlueprintsTransaction

****

以及继承自 TitanBlueprintsTransaction的StandardTitanTx

****

代码示例：

*TitanTransaction tx = graph.newTransaction();//获取TitanTransaction对象*

*Vertex saturn = tx.addVertex(T.label, "titan", "name", "saturn", "age", 10000);//创建vertices*

*Vertex sky = tx.addVertex(T.label, "location", "name", "sky");*

*jupiter.addEdge("father", saturn);//创建 edges*

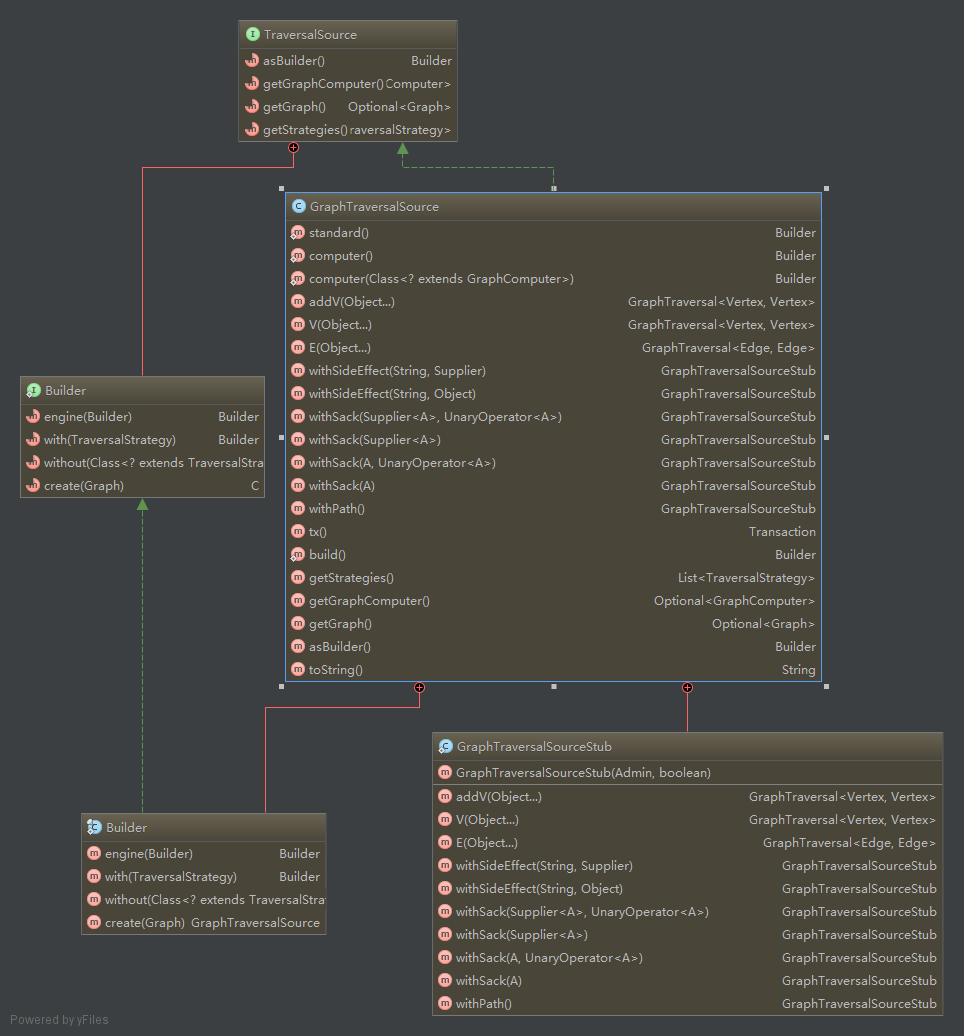
*jupiter.addEdge("lives", sky, "reason", "loves fresh breezes");*

*hercules.addEdge("battled", cerberus, "time", 12, "place", Geoshape.point(39f, 22f));*

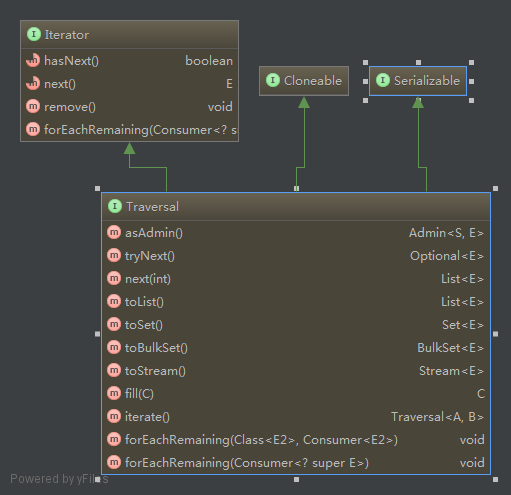
*tx.commit();//提交任务commit the transaction to disk*

## 4.4 GraphTraversalSource

GraphTraversalSource定义了Graph遍历操作的源，实现TraversalSource接口。TraversalSource负责生成Traversal，一旦构建TraversalSource成功，就可以生成任意数量的Traversal实例。每个遍历DSL将维护相应的 TraversalSource，指定"fluent-chaining"遍历步骤的方法。



其中有一个关键接口GraphTraversal继承自Traversal，Traversal作为tinkerpop.gremlin所有遍历的基本接口其每个扩展接口都可视为一个DSL，每一个Traversal都代表了一个Graph的定向遍历。GraphTraversal定义MAP STEPS、FILTER STEPS、SIDE-EFFECT STEPS 、BRANCH STEPS、UTILITY STEPS等一系列操作。



示例代码：

*TitanTransaction tx = graph.newTransaction();*

*GraphTraversalSource traversal = tx.traversal();*

*//由TitanTransaction获取到GraphTraversalSource对象*

*GraphTraversal<Vertex, Vertex> vertices = traversal.V().hasLabel("Person").has("id", "370126200001010010");//由GraphTraversalSource对象获取GraphTraversal*

*while (vertices.hasNext()) {*

*Vertex v = vertices.next();//遍历GraphTraversal得到目标Vertex*

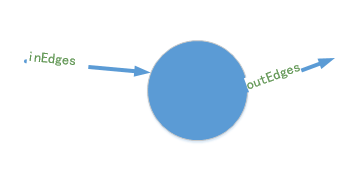
*System.out.println("v.id(): " + v.id());*

*// do something*

*}*

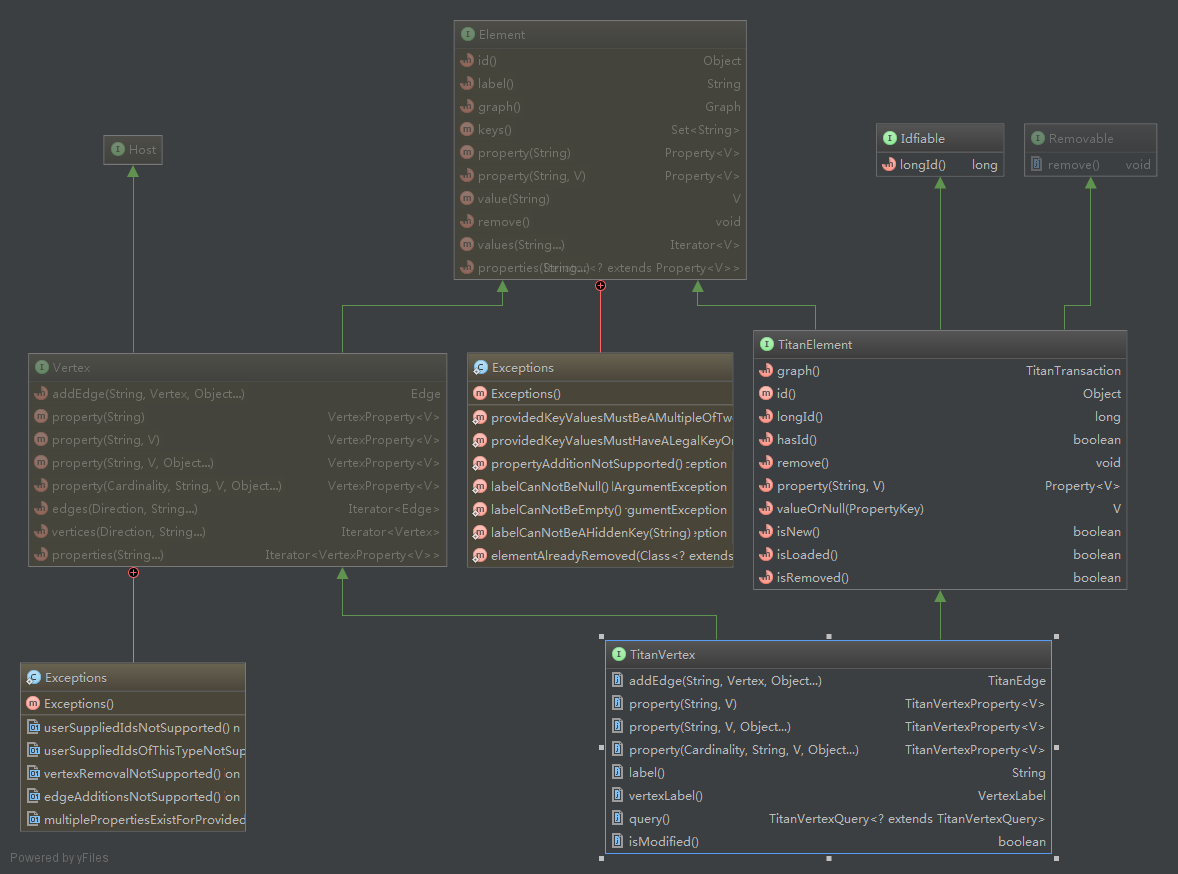
（注：关于Vertex）

Vertex维护着一组指向“incoming 和 outgoing”的Edge的对象指针，outgoing edge是指以该Vertex为结尾的那些edge，outgoing edge指的是以该Vertex为头的那些edge。



## 4.5 TitanVertex

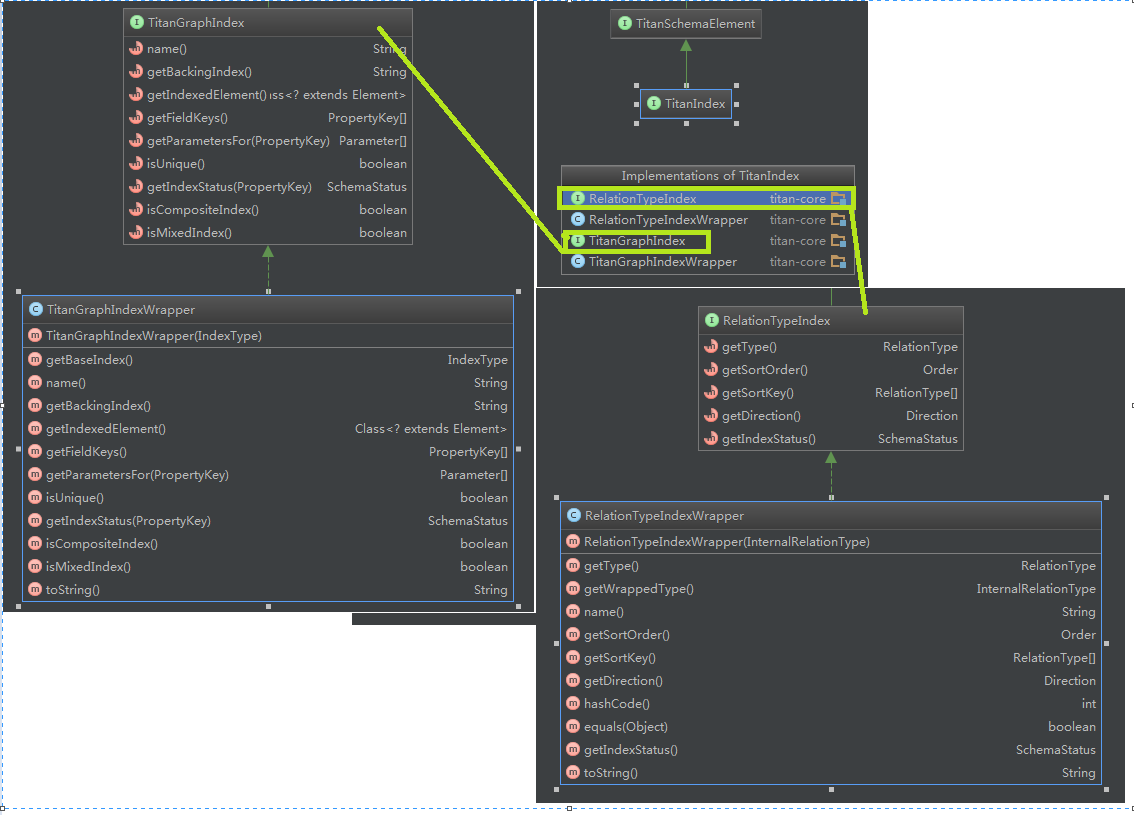
TitanVertex是TitanGraph的基本单位，它扩展了Blueprint Vertex的方法实现，Vertex具有edges和properties。 边缘将Vertex连接到其他Vertex。 属性以键值对的形式附加到此顶点上和TitanRelation一样，Vertex有一个Vertex Label。



## 4.5 TitanIndex

TitanIndex是建立在Titan数据库之上的索引，其目的在于便于有效的检索Graph。Titan支持两种不同的索引来加快查询处理：Graph indexes 和 Vertex-centric indexes。大多数图查询从其属性标识的顶点或边的列表中开始遍历，Graph indexes使全局检索操作适用于大图。

Vertex-centric indexes加速了图中的实际遍历，特别是当遍历具有许多入射边的顶点时。



TitanGraphIndex是建立在Graph上的索引，其目的在于能够通过其属性有效地检索图形元素。GraphIndex是整个图形上的全局索引结构，可以通过其特性为顶点或边缘进行有效的检索，从而获得足够的选择性条件。例如以下查询：

*g.V().has('name', 'hercules')*

*g.E().has('reason', textContains('loves'))*

第一个查询要求具有名称hercules的所有顶点。第二个要求所有edge，其中属性*'reason'* 包含*'loves'*。如果没有GraphIndex能够快速返回结果，那么这些查询需要对图形中的所有顶点或边缘进行全面扫描，以找到与给定条件匹配的条件，显然对于大图来说其效率非常低，甚至是不可行的。因此Titan中有两种GraphIndex即：复合索引（Composite indexes）和混合索引（Mixed indexes）。

复合索引非常快速和高效，但限于特定的基于已定义的属性键组合的等式查找。混合索引可用于对索引键的任意组合进行查找，并且除依赖于后备索引存储之外，还可以支持多个条件谓词。这两种类型的索引都是通过TitanManagement和由TitanManagement.buildIndex（String，Class）返回的索引构建器创建的，其中第一个参数定义了索引的名称，第二个参数指定要索引的元素的类型（例如：vertex.class）。GraphIndex的名称必须是全局唯一的。

为新定义的Property keys构建的GraphIndex，可立即使用，针对已经在使用的属性键构建的图表索引需要执行重新索引过程以确保索引包含所有以前添加的元素。在“reindex”操作完成之前，索引将不可用。鼓励相同的事务中定义图形索引作为初始Schema。

针对已经在使用的属性键构建的图表索引需要执行重新索引过程以确保索引包含所有以前添加的元素。

该接口允许引用现有的图表索引。 现有的图表索引可以通过TitanManagement.getGraphIndex（String）或TitanManagement.getGraphIndexes（Class）}来创建。

RelationTypeIndex是建立在RelationType上的一个索引，用于加速该Type的verte-center的索引。可通过：

*RelationTypeIndexTitanManagement.buildEdgeIndex(com.thinkaurelius.titan.core.EdgeLabel, String, com.tinkerpop.gremlin.structure.Direction, com.thinkaurelius.titan.graphdb.internal.Order, com.thinkaurelius.titan.core.RelationType...)}*

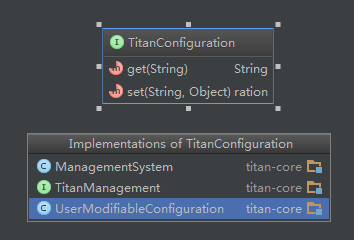
创建EdgeLabel索引；通过：

*TitanManagement.buildPropertyIndex(com.thinkaurelius.titan.core.PropertyKey, String, com.thinkaurelius.titan.graphdb.internal.Order, com.thinkaurelius.titan.core.RelationType...)}*

创建PropertyKey索引。

## 4.6 TitanConfiguration

TitanConfiguration用于读取和更改Titan的全局配置，从Graph实例中读取配置选项并影响整个数据库。



Titan区分本地和全局配置选项，本地配置选项适用于单个Titan实例，全局配置选项适用于集群中的所有实例。确切的Titan区分以下五个配置选项范围：

* Local：这些选项仅适用于单个Titan实例，并在初始化Titan实例时提供的配置中指定；
* MASKABLE：本地配置文件可以为单个Titan实例覆盖这些配置选项，如果本地配置文件未指定该选项，则其值将从全局Titan集群配置中读取；
* GLOBAL：这些选项始终从集群配置读取，不能在实例的基础上覆盖；
* GLOBAL\_OFFLINE：像GLOBAL一样，但更改这些选项需要重新启动集群，以确保整个集群的值相同；
* FIXED：像GLOBAL一样，但一旦Titan集群被初始化，该值就不能被更改。

当集群中的第一个Titan实例启动时，全局配置选项将从提供的本地配置文件初始化，随后，通过Titan的管理API完成全局配置选项的更改。可通过：g.getManagementSystem()打开一个Titan实例的Handle类，通过下面的示例修改配置：

mgmt = graph.openManagement()

*mgmt.get('cache.db-cache')*

*// Prints the current config setting*

*mgmt.set('cache.db-cache', true)*

*// Changes option*

*mgmt.get('cache.db-cache')*

*// Prints 'true'*

*mgmt.commit()*

*// Changes take effect*

对于新启动的实例更改配置选项不会影响运行实例，但是更改GLOBAL\_OFFLINE配置选项需要重新启动群集，以便更改对所有实例立即生效。要更改GLOBAL\_OFFLINE配置项，需要按一下步骤执：

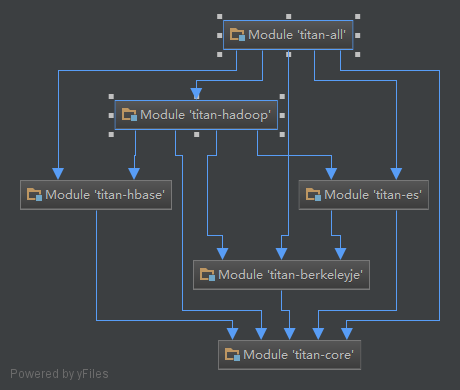
* 关闭集群中除Titan之外的所有实例
* 连接到单个实例
* 确保所有运行的事务都已关闭
* 确保没有新事务被启动（即集群必须脱机）
* 打开管理API
* 更改配置选项
* 调用commit将自动关闭图形实例
* 重新启动所有实例

# 五Titan 2.0 升级

目前Titan数据库官方发布的版本为1.0.0，兼容的组件及其版本如下所示：

* Cassandra 2.1.9
* HBase 0.94.25, 0.96.2-hadoop2, 0.98.8-hadoop2, 1.0.2
* Hadoop 1.2.1
* BerkeleyJE 5.0.73
* Elasticsearch 1.5.1
* Lucene 4.10.4
* Solr 5.2.1
* TinkerPop 3.0.1-incubating
* Java 1.8

目前大数据平台HDH的组件HBase、hadoop和ES的版本较高，同时Titan中有些组件暂时用不到比如：Cassandra、Solr，因此，为了让Titan能够更好的工作在大数据平台上需要对Titan进行升级，目标版本2.0.0-hdh4.0.0。升级后的版本包括以下模块：



升级工作设计已下几个方面：

1. 删除暂时用不到的模块；
2. 升级组件到目标版本；
3. 升级中版本冲突解决；
4. 升级之后版本保证功能的可用性。

## 5.1 暂时不用的组件删除

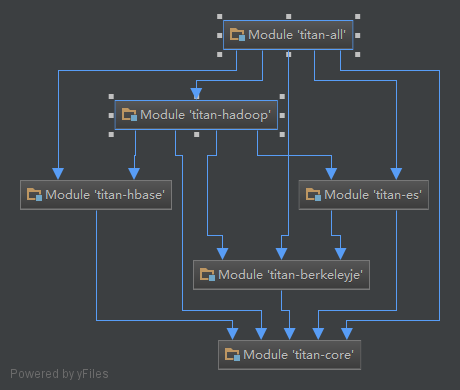
目前我们的大数据平台使用的索引引擎为Elasticsearch，底层数据库为HBase，因此在Titan2.0版中仅保留两个及与其相关的组件因此在版本中保留如下组件：

**titan-berkeleyje、 titan-es 、titan-hadoop、titan-hbase。**

暂时去除：

**titan-cassandra、** **titan-lucene、 titan-solr。**

升级版本模块如下图所示：

****

## 5.2 组件升级

需要根据平台版本对组件版本进行相应的升级，需要升级的组件有：ES、Hadoop、Hbase以及这三个组件以来的相应底层比如：lucene版本。

### 5.2.1 Elasticsearch升级

当前版本：**1.5.1**，目标版本：**2.1.1，**涉及的工作：

1. JAR依赖升级；
2. 接口修改，原来在1.5.1的版本中的部分接口在在2.1.1中废除或者不建议使用，因此需要修改Titan中的代码，根据新接口事项相应功能；
3. 版本绑定修改，在Titan数据库中各组件版本都做了严格的校验与绑定，而升级版本需要对这部分的操作进行解绑与重新校验。

### 5.2.2 Lucene版本升级

ES的实现依赖于Lucene，因次ES版本升级需要相应升级Lucene版本：

当前版本：4.10.4 目标版本：5.3.1

### 5.2.3 HBase版本升级

当前版本：1.0.2 目标版本：1.2.0-cdh5.7.0，涉及的工作：

1. 冲突解决；
2. 1.2.0-cdh5.7.0版本数据库操作接口的实现；
3. 版本绑定解绑与重新校验。

### 5.2.4 Hadoop版本升级

当前版本：1.2.1 目标版本：2.6.0-cdh5.7.0，涉及的工作：

1. 冲突解决；
2. 新版本数据操作接口修改；
3. 功能测试。

## 5.3 冲突解决

由于在升级的过程中牵扯到多个组件的升级和删除，涉及到大量第三方依赖的重新引入、新增和删除，这就导致大量的依赖存在多个版本，结果是整个工程不可用，因此需要花费大量的时间来排查和修改依赖的冲突。

## 5.4 功能的测试

升级的基本要求是保证功能的可用性，期间牵扯到新接口的引入，部分功能的重新编码以及新组建对整个项目可用性的影响，因此需要新增测试用例对组件功能进行测试。

# 六 Titan数据库的限制

Titan数据库的扩展性很好的解决了图数据在存储、索引、检索等方面的限制，但并不是说它就没有了缺点，在使用Titan数据库是我们需要注意其各种限制和“陷阱”，其中有一些限制是必要的设计，而其它的一些问题将随着Titan的发展而得以纠正。目前Titan数据库在以下几个方面存在缺陷或者待优化。

## 6.1 设计上的限制

设计上的限制是一个长期权衡的结果，目前，想要改变这些限制是有困难的或不切实际的，也就是说这些限制会一直存在，在以后的也不会被删除。

* Size Limitation，Titan可以存储多达0.2\*2 ^ 60的edgs和0.5\*2 ^ 60的vertexs，这个限制是由Titan的id Schema限定的；
* DataType Definitions，当使用dataType（Class）声明属性键的数据类型时，Titan将强制该键的所有属性都具有声明的类型，除非该类型为Object.class；
* Edge Retrievals 效率O(log(k))；
* Type 定义不能被修改。

## 6.2 临时限制

* 限制对混合索引 （Mixed Index）的支持，目前混合索引仅支持部分Titan数据库支持的数据类型，此外，混合索引还不支持SET和LIST的属性键；
* 批量加载速度，目前Titan提供基于配置文件的批量加载模式。但是，此批处理模式只能更快地加载到存储后端，它不会使用存储后端特定批量加载技术。因此，Titan中的批量装载目前比单机数据库提供的批量加载模式慢。