GTL空间数据管理

## 系统架构



Common模块是基础模块，定义了GTL中的基础数据结构和对象，如序列化、坐标点、几何对象、要素对象、数据访问接口等。Data模块主要实现了数据在不懂存储介质上的管理，包括内存、文件、HDFS等，其中gtl-data-storage是一种基于抽象分页存储系统的数据管理模块，本身可以实现基于内存、文件等的数据管理，也是系统的默认实现。

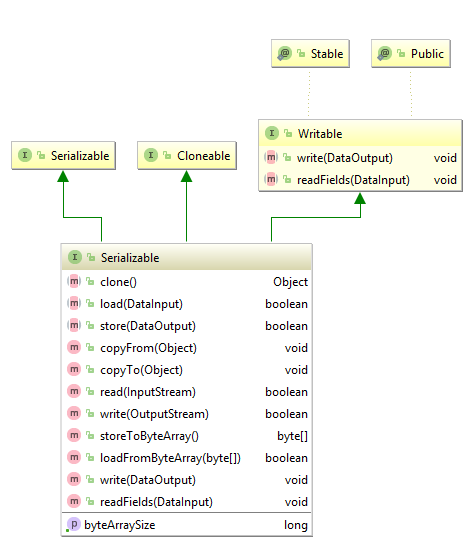
此外，在GTL中使用了部分GeoTools的模块。其中，元数据、坐标系统使用了GeoTools的实现。分别对应的是gt-metadata和gt-referencing，这两个模块又都依赖于gt-opengis。

在gtl-shp工程中，SHP文件读写暂时调用GeoTools实现，依赖于gt-shapefile。它依赖于gt-data、gt-main、gt-api、gt-referencing。

## IO接口设计

### 序列化接口

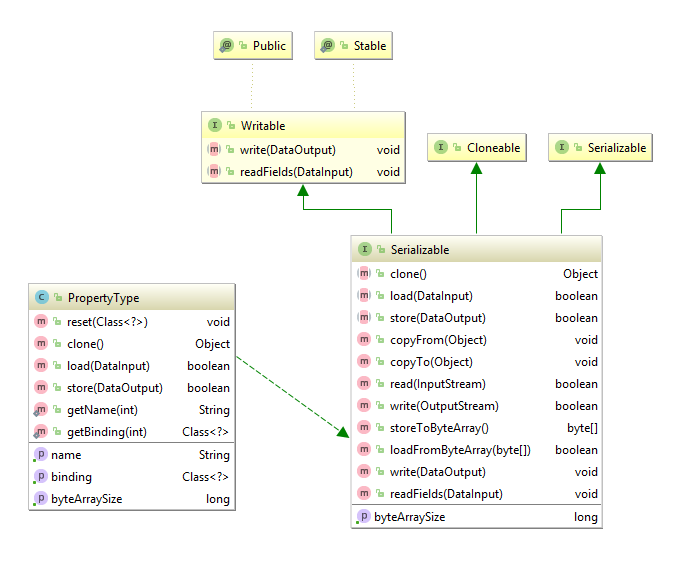
GTL中的所有对象，凡是需要进行IO操作的，都必须实现gtl.io.Serializable接口。该接口从java.io.Serializable和org.apache.hadoop.io.Writable继承，如下图所示：



之所以从Writable继承，是为了在分布式计算中能够支持Hadoop体系的序列化。在GTL中，所有需要进行IO操作的对象，都需要实现gtl.io.Serializable接口，所以，这类对象类都需要包含一个私有成员变量：

**private static final long *serialVersionUID*** = -4627070846660440083L;

例如，其中的gtl.common.PropertyType就实现了接口，如下图所示：



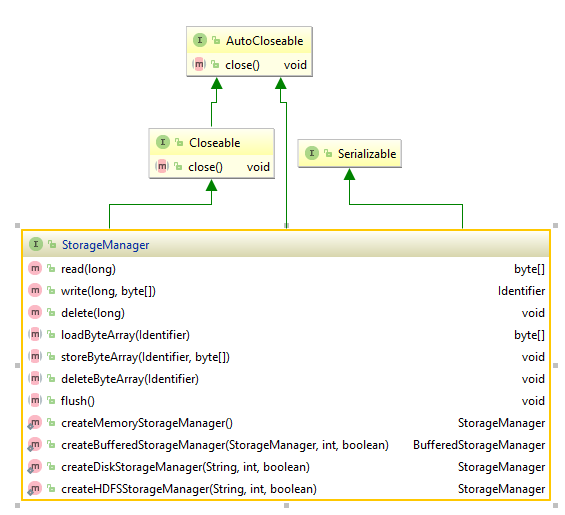
在Java 8及后续版本由于接口可以支持default方法，gtl.io.Serializable中下面3个方法没有默认实现：

Object clone() ;  
**boolean** load(DataInput in) **throws** IOException;  
**boolean** store(DataOutput out) **throws** IOException;

除了上面3个方法以外，其他方法都进行了默认实现。所以，在实现该接口的时候，上面3个方法是强制必须实现的。此外，其中的**long** getByteArraySize()方法是计算对象序列化后的字节数，默认实现为将其写入一个字节数组中，然后返回该字节数；这种实现方法消耗过大，如果一个对象的字节数是固定的，或者是比较容易计算出来的，建议重载此方法。例如，对于IdentifierImpl类，其数据成员的大小固定为8，则可以重载此方法，直接返回8。

### 分页存储机制

分页存储机制是GTL的默认存储机制。所谓的分页存储，就是将内存、文件等按照页面进行组织，读写也以页面为基本单位。页面分为公有页面和私有页面。所谓公有页面，也即在目录中可以看到页码的页面，类似于书本的章节开头所在的页面。所谓私有页面，也即在目录中不会出现页码的页面。如果一块内容一个一个页面承载不下，则写该内容的页面会有多个，这些页面中第一个是公有页面，其余的是私有页面。建议在高速存储介质，如内存中，不要采用私有页面的方式，而应该采用直接扩大公有页面的方式实现。而在低速介质中，如磁盘文件、HDSF等，则推荐采用私有页面方式。GTL的分页存储管理机制通过接口StorageManager定义，其中出现的页码均为公有页面ID，如下图所示：

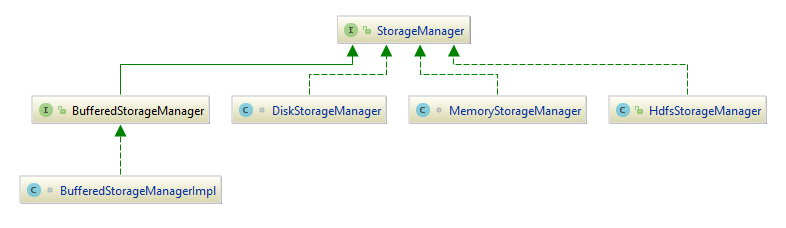


主要通过以下三个方法来实现分页读写机制：

*/\*\*  
 \* 读取页面ID为page的字节信息，存放到一个字节数组中  
 \** ***@param page*** *页面唯一ID  
 \** ***@return*** *页面中的内容，存放到字节数组中返回  
 \** ***@throws*** *IOException  
 \*/***byte**[] loadByteArray(Identifier page) **throws** IOException;  
  
*/\*\*  
 \*在页面page中写入字节数组；如果页面ID为-1，则选择一个新的页面写入，并返回新页面ID。  
 \* 如果一个页面写不完需要写入的数据，则会自动增加私有页面或扩展页面；在具体实现本方法  
 \* 的时候，如果是高速存储介质，例如内存，推荐采用扩展页面大小的方式；如果是低速介质，  
 \* 如磁盘，HDFS等，推荐采用私有页面方式。所谓的私有页面，就是跟在一个公有页面（也称  
 \* 为目录上的页码，本接口中的所有页面都是公有页面）。  
 \** ***@param page*** *页面ID，如果为-1，则选择一个新的页面写入，并修改传入page并返回  
 \* 如果为有效页面ID，则写入该页面。  
 \** ***@param data*** *需要写入的字节数组  
 \** ***@throws*** *IOException  
 \*/***void** storeByteArray(Identifier page, **byte**[] data) **throws** IOException;

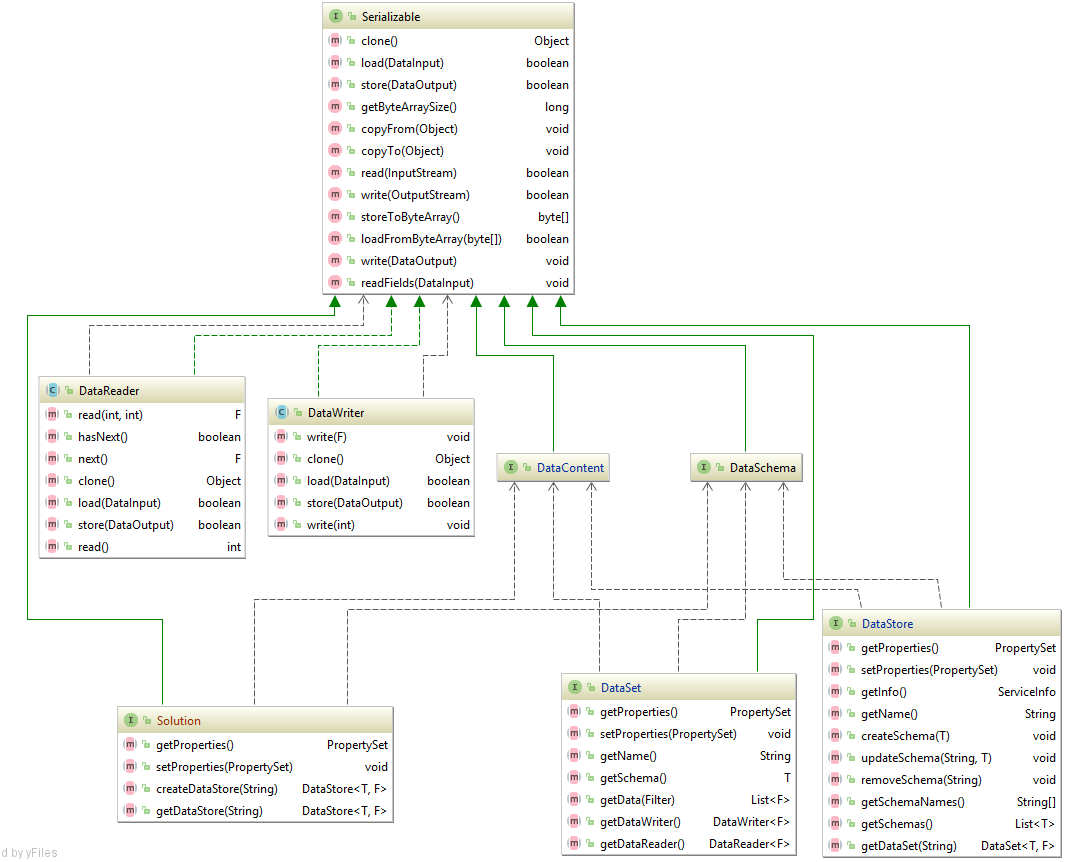
*/\*\*  
 \* 删除指定页面  
 \** ***@param page*** *\** ***@throws*** *IOException  
 \*/***void** deleteByteArray(Identifier page) **throws** IOException;

这三个函数后期会逐渐废除，推荐使用read，write，delete替换这三个函数。GTL目前实现了内存、磁盘文件和HDFS三种方式的分页存储管理器，分别为MemoryStorageManager、DiskStorageManager、HdfsStorageManager,如下图所示：



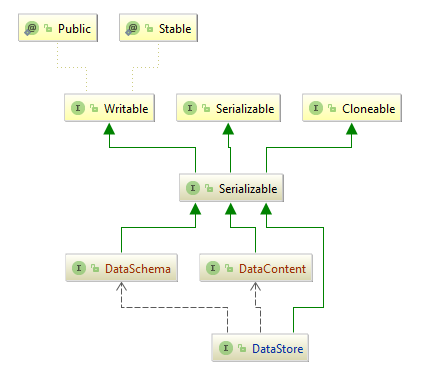
## 基础数据管理接口

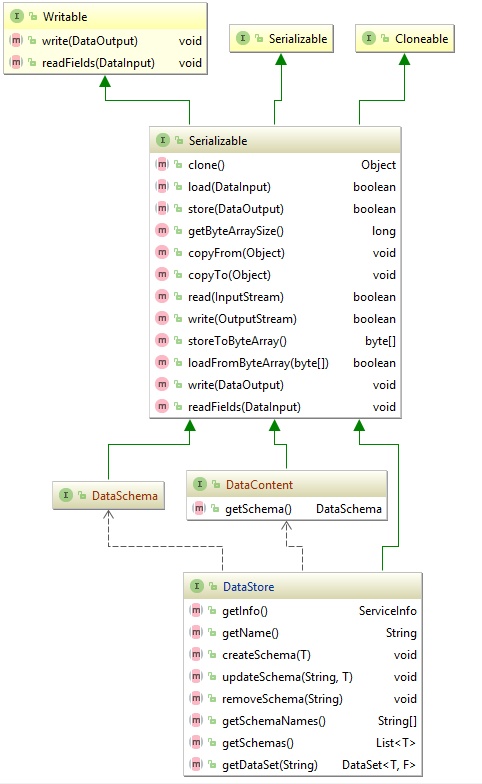
GTL中的数据管理有7个核心接口，分别是DataReader、DataWriter、DataSchema、DataContent、DataSet、DataStore和Solution,其中GTL中的Feature实现了DataContent接口，FeatureType实现了DataSchema接口，如下图所示：



### DataStore

本接口代表了一个数据库，主要由一系列的数据模式（DataSchema）和数据内容（Data Content）构成。一个数据模式（DataSchema）及其对应的一条或多条数据内容（DataContent）构成是一个数据集（DataSet）。这个DataSet类似于数据库中的数据表，在空间数据库中可以理解为一个要素类型和多个要素对象构成的数据集。所以，要素（Feature）必须实现DataContent接口，FeatureType必须实现DataSchema接口。

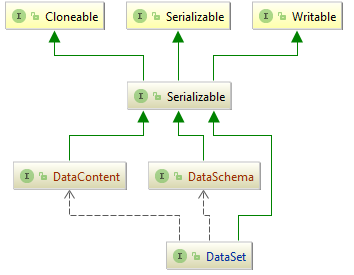


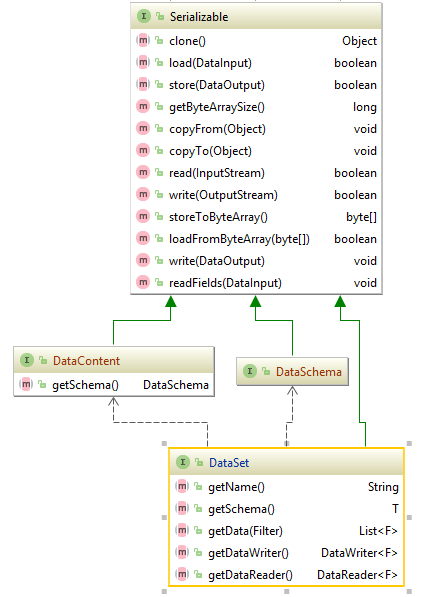


*/\*\*  
 \* 存储数据的仓库，相当于数据库  
 \* 每个Schema相当于一个SHP文件结构或一个数据表的结构  
 \* 一个DataSet相当于一个数据表或一个SHP文件  
 \** ***@param <T>*** *数据模型，例如，FeatureType是Feature的模式，也可以是数据表的结构和数据表的记录集之间的关系  
 \** ***@param <F>****数据内容，例如，Feature是FeatureType对应的数据内容，也可以是数据表的结构和数据表的记录集之间的关系  
 \*/***public interface** DataStore <T **extends** DataSchema, F **extends** DataContent> **extends** Serializable{  
 */\*\*  
 \* 获取数据库相关服务信息  
 \** ***@return*** *\*/* ServiceInfo getInfo();  
  
 */\*\*  
 \*获取数据库的名称  
 \** ***@return*** *数据库的名称  
 \*/* String getName();  
  
 */\*\*  
 \* 在数据库中创建一个数据模式  
 \** ***@param featureType*** *\** ***@throws*** *IOException  
 \*/* **void** createSchema(T featureType) **throws** IOException;  
  
 */\*\*  
 \* 更新一个指定名称的数据模式  
 \** ***@param typeName*** *\** ***@param featureType*** *\** ***@throws*** *IOException  
 \*/* **void** updateSchema(String typeName, T featureType) **throws** IOException;  
  
 */\*\*  
 \* 删除指定的数据模式，模式对应的数据集也要随之删除  
 \** ***@param typeName*** *\** ***@throws*** *IOException  
 \*/* **void** removeSchema(String typeName) **throws** IOException;  
  
 */\*\*  
 \* 获取所有的数据模式名称  
 \** ***@return*** *\** ***@throws*** *IOException  
 \*/* String[] getSchemaNames() **throws** IOException;  
  
 */\*\*  
 \* 返回数据库中所有的数据模式  
 \** ***@return*** *\** ***@throws*** *IOException  
 \*/* List<T> getSchemas() **throws** IOException;  
  
 */\*\*  
 \* 获取指定模式的数据集  
 \** ***@param typeName*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* DataSet<T,F> getDataSet(String typeName) **throws** IOException;  
}

### DataSet

本接口代表了DataStore中的一个数据集（DataSet）。一个数据集包含一个数据模式（DataSchema）和多条数据内容（DataContent）。在空间数据库中，一个数据集由一个要素类型（FeatureType）以及多个这种类型的要素（Feature）对象构成。

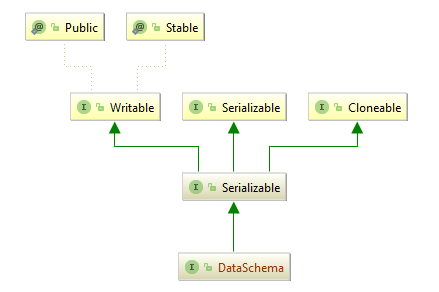




*/\*\*  
 \* 对DataStore中存储数据的操作,  
 \* 相当于一个数据表或SHP文件  
 \** ***@param <T>*** *\** ***@param <F>*** *\*/***public interface** DataSet <T **extends** DataSchema, F **extends** DataContent> **extends** Serializable{  
 */\*\*  
 \*  
 \** ***@return*** *数据集的名称  
 \*/* String getName();  
  
 */\*\*  
 \*  
 \** ***@return*** *数据集的结构  
 \** ***@throws*** *IOException  
 \*/* T getSchema() **throws** IOException;  
  
 */\*\*  
 \* 根据过滤条件，返回符合条件的对象列表  
 \** ***@param filter*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* List<F> getData(Filter filter) **throws** IOException;  
  
 */\*\*  
 \* 返回该数据集上的Writer对象  
 \** ***@return*** *\** ***@throws*** *IOException  
 \*/* DataWriter<F> getDataWriter() **throws** IOException;  
  
 */\*\*  
 \* 返回该数据集上的Reader  
 \** ***@return*** *\** ***@throws*** *IOException  
 \*/* DataReader<F> getDataReader() **throws** IOException;  
}

### DataSchema

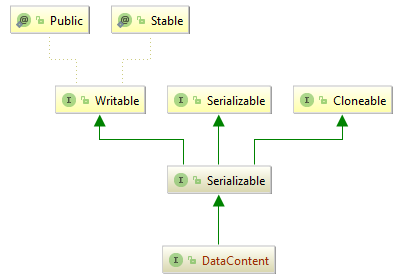
本接口代表一个数据模式，可以是一个数据表的结构，或者是一个要素类型（FeatureType）。



*/\*\*  
 \* 数据模式接口，可以数据库中数据表的结构，  
 \* 或者是SHP文件的属性结构，也即FeatureType  
 \* 一个DataSchema对应0个或多个DataContent  
 \*/***public interface** DataSchema **extends** Serializable {  
}

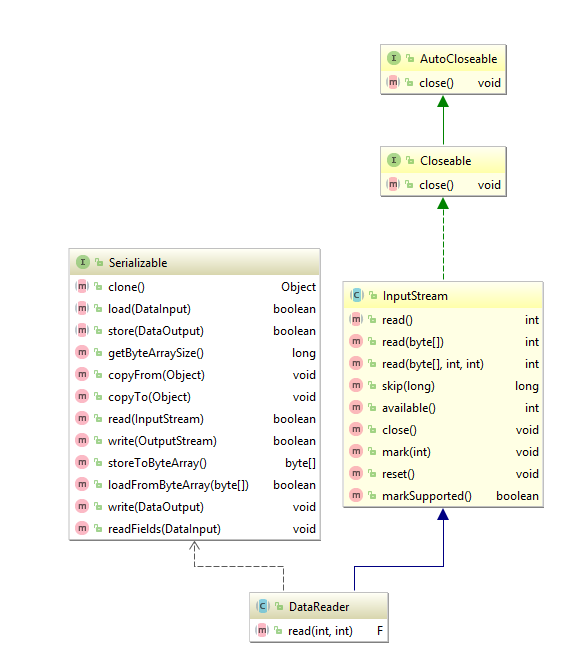
### DataContent

本接口代表一条数据内容，可以是一个数据表的一个记录，或者是一个要素对象（Feature）。



### DataReader

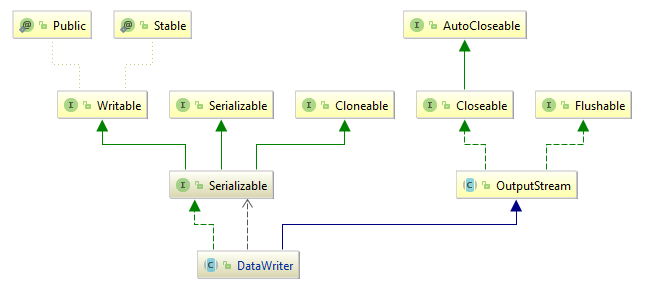
本抽象类从InputStream继承，主要用于实现了gtl.io.Serializable接口的对象读取。



**public abstract class** DataReader <F **extends** Serializable> **extends** InputStream **implements** Serializable{  
  
 **private static final long *serialVersionUID*** = 6341437970744016771L;  
  
 */\*\*  
 \* 在Reader，可能是文件、内存或管道，从offset位置开始读取byteCount个字节内容，并构建对象返回  
 \** ***@param offset*** *\** ***@param byteCount*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* **public abstract** F read(**int** offset, **int** byteCount) **throws** IOException;  
  
 @Override  
 **public abstract** Object clone();  
  
 @Override  
 **public abstract boolean** load(DataInput in) **throws** IOException;  
  
 @Override  
 **public abstract boolean** store(DataOutput out) **throws** IOException ;  
  
 @Override  
 **public abstract int** read() **throws** IOException ;  
}

### DataWriter

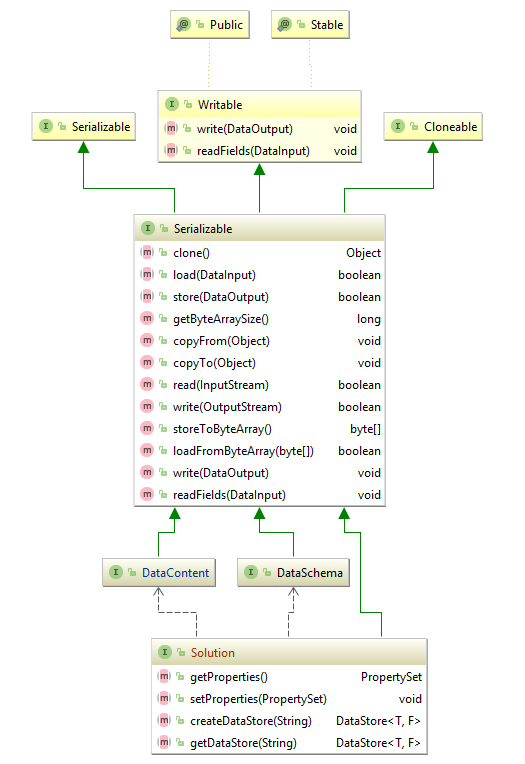
本抽象类从OutputStream继承，主要用于实现了gtl.io.Serializable接口的对象写出。



**public abstract class** DataWriter<F **extends** Serializable> **extends** OutputStream **implements** Serializable {  
 **private static final long *serialVersionUID*** = -2511879492829177520L;  
  
 */\*\*  
 \* 将对象s写入Writer，可能是文件或内存，还有可以能是网络管道  
 \** ***@param s*** *\** ***@throws*** *IOException  
 \*/* **public abstract void** write(F s) **throws** IOException;  
  
 @Override  
 **public abstract** Object clone() ;  
  
 @Override  
 **public abstract boolean** load(DataInput in) **throws** IOException ;  
  
 @Override  
 **public abstract boolean** store(DataOutput out) **throws** IOException ;  
  
 @Override  
 **public abstract void** write(**int** b) **throws** IOException;  
}

### Solution

一个方案包括若干个数据库（DataStore）。



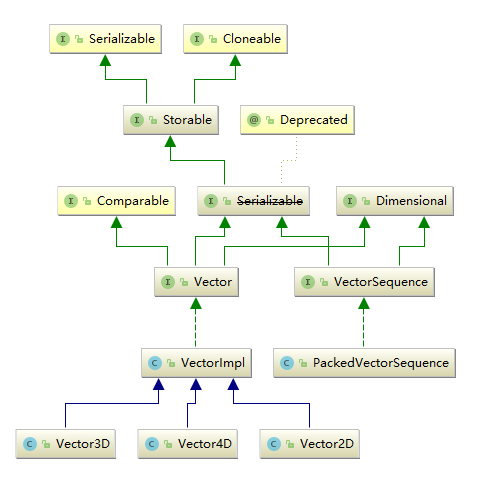
**public interface** Solution <T **extends** DataSchema, F **extends** DataContent> **extends** Serializable{  
 */\*\*  
 \* 获取属性集  
 \** ***@return*** *\*/* PropertySet getProperties();  
  
 */\*\*  
 \* 设置属性集  
 \** ***@param ps*** *\*/* **void** setProperties(PropertySet ps);  
  
 */\*\*  
 \* 创建一个数据库库  
 \** ***@param storeName*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* DataStore<T,F> createDataStore(String storeName) **throws** IOException;  
  
 */\*\*  
 \* 获取方案中存在的数据库  
 \** ***@param storeName*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* DataStore<T,F> getDataStore(String storeName) **throws** IOException;  
}

## 几何数据管理接口

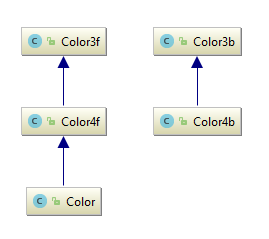
所有的几何类，都从Geometry类继承。

### 基础数据结构

#### 节点

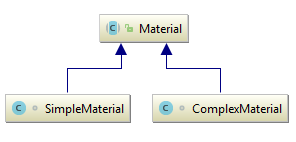


#### 颜色

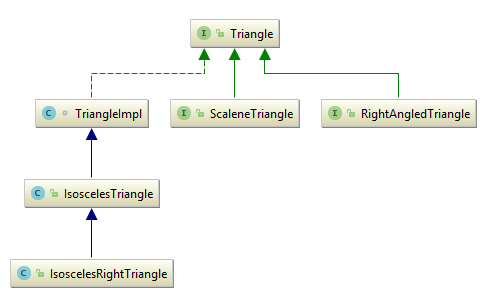


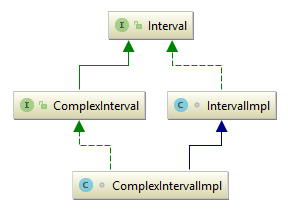
#### 纹理

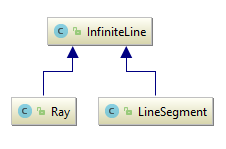
#### 材质



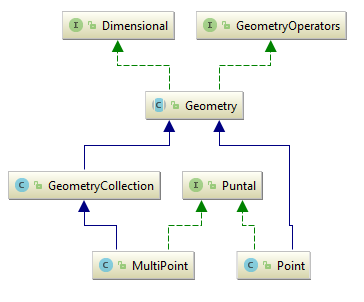
#### 其他



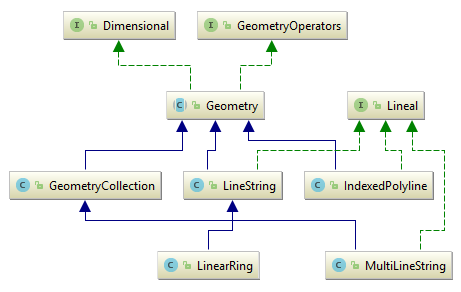




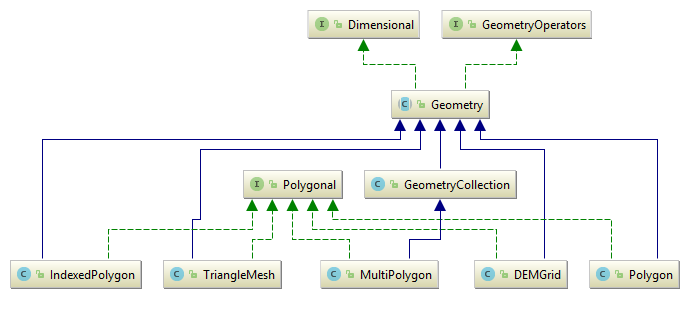
### 点状几何对象（Puntal）



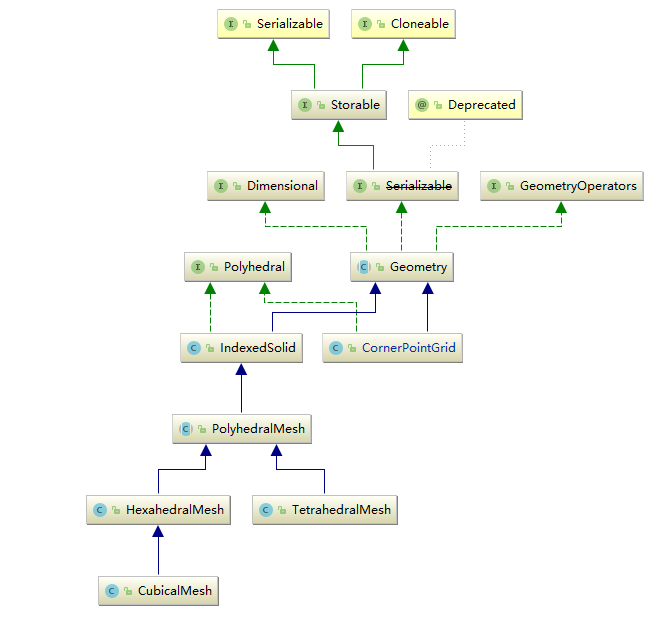
### 线状几何对象（Lineal）



### 面状几何对象（Polygonal）

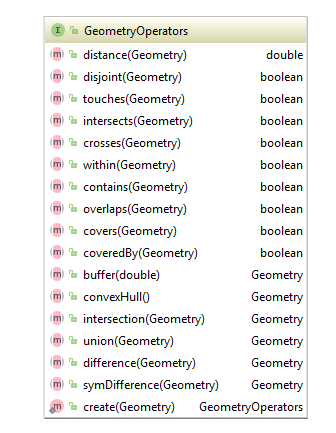


### 体状几何对象（Polyhedral）



### 几何对象之间的关系计算

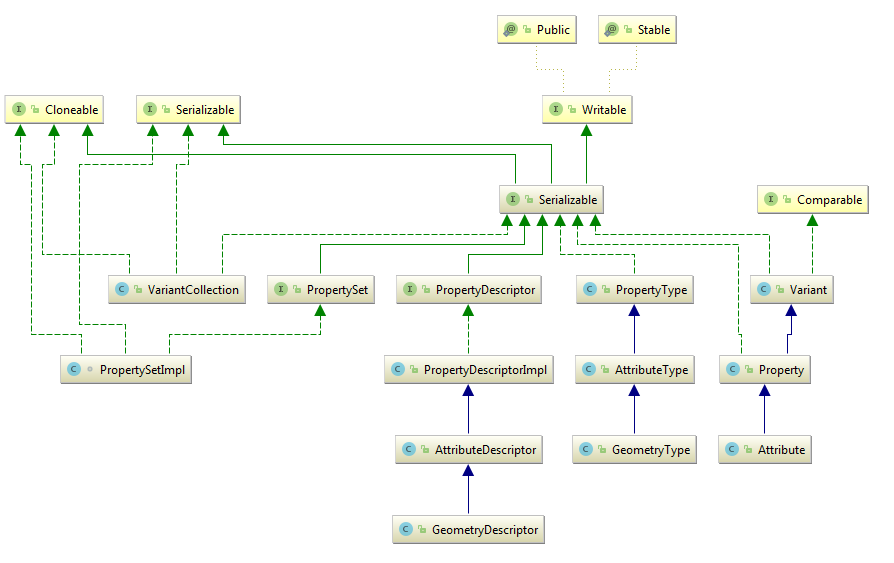
几何对象之间的关系操作的接口为GeometryOperators，定义如下：

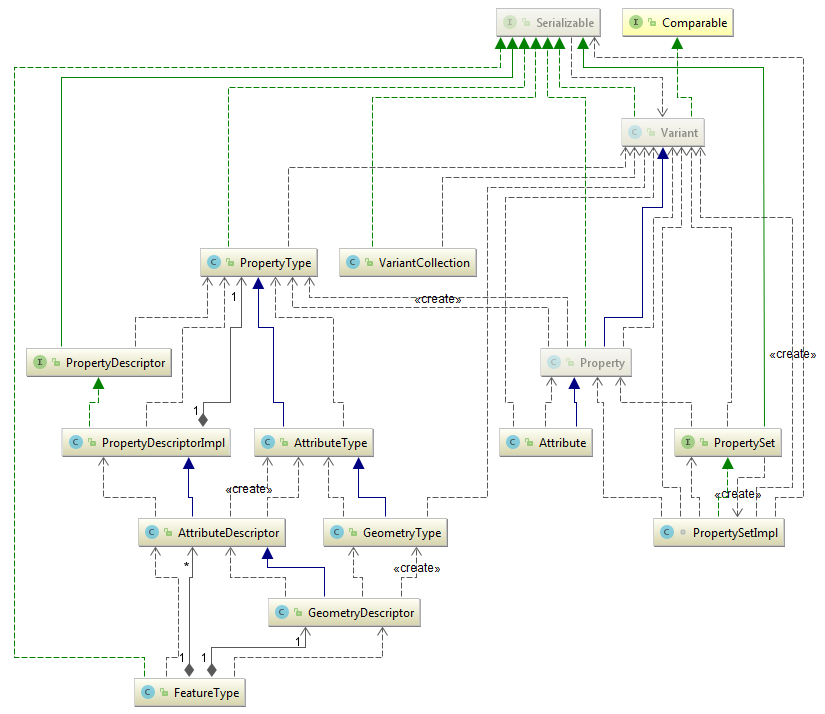


目前只有二维的Point、MultiPoint、LineString、MultiLineString、Polygon、MultiPolygon、GeometryCollection等7种几何对象借助JTS实现了上述接口中定义的几何操作。其他的几何对象均未实现上述几何操作。

还需要实现其他15种三维几何对象之间的关系计算。

## 属性数据管理接口





### Variant

### VariantCollection

### Property

### PropertySet

### Attribute

### PropertyType

### AttributeType

### GeometryType

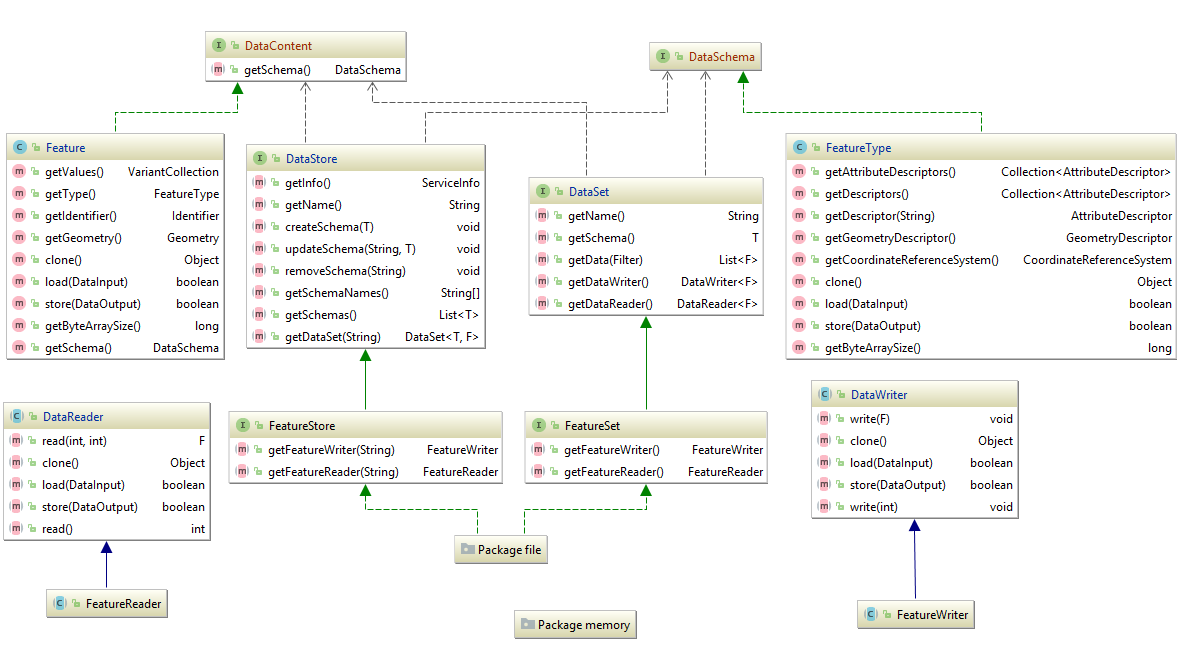
### PropertyDescriptor

### AttributeDescriptor

### GeometryDescriptor

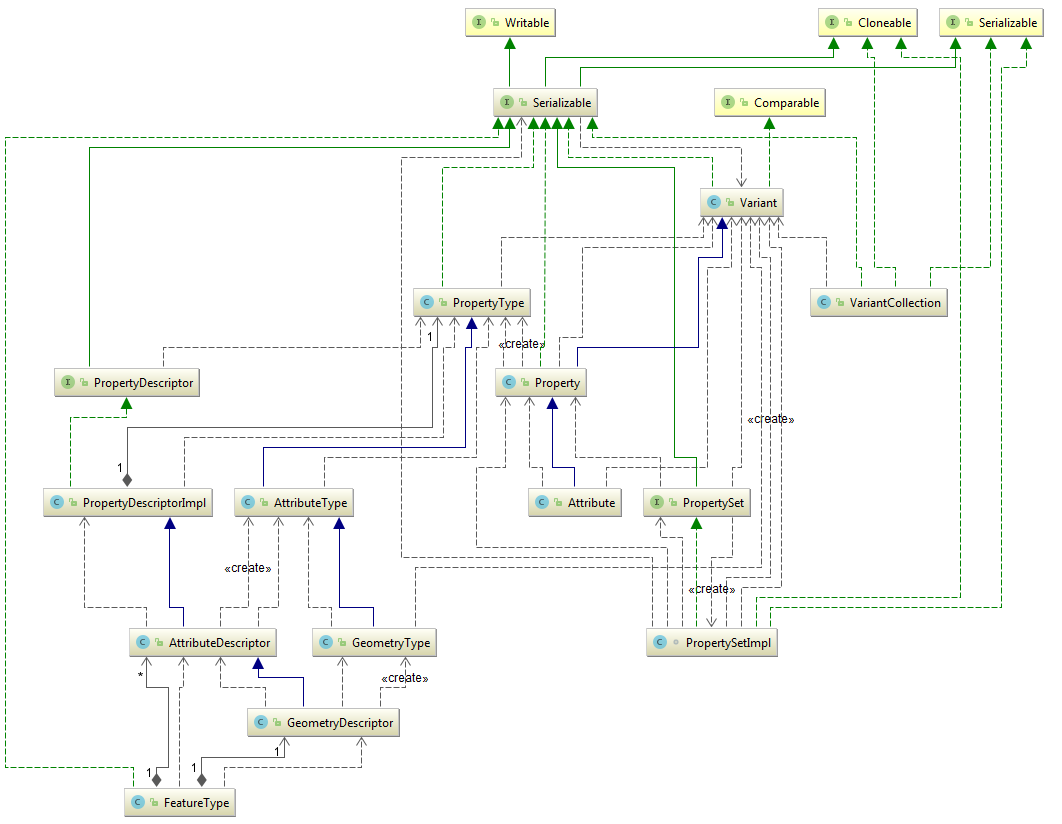
## 空间要素管理接口

GTL中的空间要素管理有6个核心接口，分别是FeatureReader、FeatureWriter、FeatureType、Feature 、FeatureSet和FeatureStore,其中GTL中的Feature实现了DataContent接口，FeatureType实现了DataSchema接口，如下图所示：



这六个接口构成了一个比较完整的空间数据库。其中空间数据库（FeatureStore）、空间要素集（FeatureSet）、空间要素读取（FeatureReader）和空间要素写入（FeatureWriter）这四个接口实现了空间数据库的基本读写操作。GTL中任何版本的空间要素数据库，都是对这四个接口的不同实现。

### FeatureType



### Feature

### FeatureStore

**public interface** FeatureStore **extends** DataStore<FeatureType, Feature> {  
 */\*\*  
 \* 打开要素库,如果该要素库不存在，则新建一个要素库并打开  
 \** ***@param path*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* **boolean** open(Path path) **throws** IOException;  
  
 */\*\*  
 \* 关闭要素库  
 \*/* **void** close();  
 */////////////////////////////////////////////////////////////////////////////  
 //FeatureType operations begin  
 /////////////////////////////////////////////////////////////////////////////  
 /\*\*  
 \* 构建一个FeatureTypeBuilder，并返回  
 \* 通过FeatureTypeBuilder构建的FeatureType 必须添加到本Store中  
 \** ***@return*** *\*/* **default** FeatureTypeBuilder getFeatureTypeBuilder(){  
 **return new** FeatureTypeBuilder();  
 }  
  
 */\*\*  
 \* 获取FeatureType的查找器，可以通过名称查找该要素类型对象  
 \** ***@return*** *\*/* FeatureTypeFinder getFeatureTypeFinder();  
  
 */\*\*  
 \* 传入FeatureType，并将其加入到本Store中,  
 \* 这是FeatureType添加到FeatureStore中的唯一方法  
 \* 目的是保证FeatureType与FeatureSet的一一对应关系  
 \* 然后在本FeatureStore中创建一个FeatureSet  
 \** ***@param featureType*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* FeatureSet createFeatureSet(FeatureType featureType) **throws** IOException;  
  
 */\*\*  
 \* 获取对应的FeatureSet  
 \** ***@param featureTypeName*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* FeatureSet getFeatureSet(String featureTypeName) **throws** IOException;  
  
 */\*\*  
 \* 移除FeatureSet及其对应的FeatureType  
 \** ***@param featureTypeName*** *\** ***@throws*** *IOException  
 \*/* **boolean** removeFeatureSet(String featureTypeName) **throws** IOException;  
  
 */////////////////////////////////////////////////////////////////////////////  
 //Feature operations begin  
 /////////////////////////////////////////////////////////////////////////////  
 /\*\*  
 \* 添加要素，首先查找所属FeatureSet,  
 \* 然后打开该FeatureSet，添加进去  
 \** ***@param feature*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* Feature appendFeature(Feature feature) **throws** IOException;  
  
 */\*\*  
 \* 移除要素  
 \** ***@param feature*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* Feature removeFeature(Feature feature) **throws** IOException;  
  
 */\*\*  
 \* 获取FeatureWriter对象  
 \** ***@param featureTypeName*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* FeatureWriter getFeatureWriter(String featureTypeName) **throws** IOException;  
  
 */\*\*  
 \* 获取FeatureReader对象  
 \** ***@param featureTypeName*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* FeatureReader getFeatureReader(String featureTypeName) **throws** IOException;  
  
 */\*\*  
 \* 根据要素类型名称，构建FeatureBuilder  
 \* 传入FeatureBuilder的要素类型必须是本Store中已经存在的要素类型  
 \** ***@param featureTypeName*** *本Store中已经存在的要素类型名称  
 \** ***@return*** *\*/* **default** FeatureBuilder getFeatureBuilder(String featureTypeName){  
 FeatureTypeFinder featureTypeFinder = getFeatureTypeFinder();  
 **if**(featureTypeFinder==**null**) **return null**;  
 FeatureType ft = featureTypeFinder.find(featureTypeName);  
 **if**(ft==**null**) **return null**;  
 **return new** FeatureBuilder(ft);  
 }  
  
}

### FeatureSet

**public interface** FeatureSet **extends** DataSet<FeatureType, Feature> {  
 FeatureWriter getFeatureWriter() **throws** IOException;  
 FeatureReader getFeatureReader() **throws** IOException;  
 FeatureStore getFeatureStore() **throws** IOException;  
}

### FeatureReader

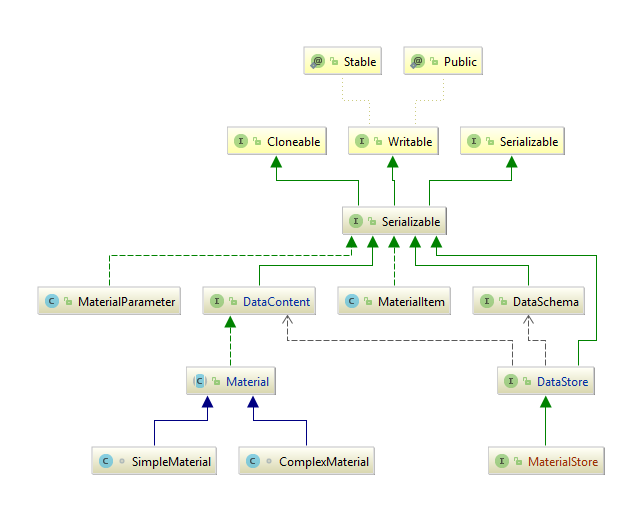
### FeatureWriter

## 材质管理接口

一个材质库，包括一个material目录，目录下包括0个或1个材质文件(materials)。

### MaterialStore

主要操作包括，初始化材质库，打开材质库，添加材质，删除材质，查找材质，关闭材质库等。

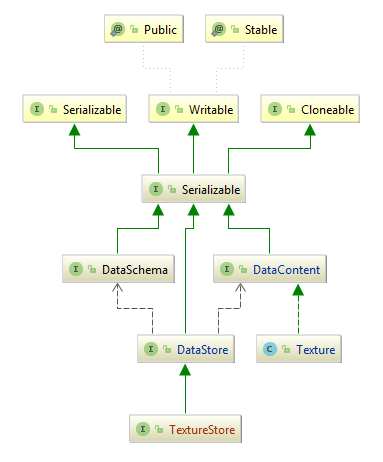


**public interface** MaterialStore **extends** DataStore<DataSchema,Material> {  
 */\*\*  
 \* 打开材质库,如果该材质库不存在，则新建一个材质库并打开  
 \** ***@param path*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* **boolean** open(Path path) **throws** IOException;  
  
 */\*\*  
 \* 添加材质  
 \** ***@param Material*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* Material append(Material Material) **throws** IOException;  
  
 */\*\*  
 \* 移除材质  
 \** ***@param identifier*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* Material remove(Identifier identifier) **throws** IOException;  
  
 */\*\*  
 \* 查找材质  
 \** ***@param identifier*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* Material find(Identifier identifier) **throws** IOException;  
  
 */\*\*  
 \* 查找材质  
 \*/* Material find(String materialName) **throws** IOException;  
  
 */\*\*  
 \* 关闭材质库  
 \*/* **void** close();  
}

## 纹理管理接口

一个纹理库包括，1个纹理目录（textures），里面包括0个或多个纹理图片文件或对象。

### TextureStore



主要操作包括，初始化纹理库，打开纹理库，添加纹理，删除纹理，查找纹理，关闭纹理库等。

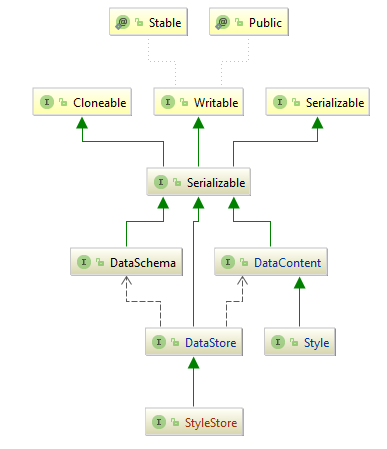
*/\*\*  
 打开纹理库  
 添加纹理  
 删除纹理  
 查找纹理  
 关闭纹理库  
  
 \*/***public interface** TextureStore **extends** DataStore<DataSchema, Texture> {  
   
 */\*\*  
 \* 打开纹理库,如果该纹理库不存在，则新建一个纹理库，并打开  
 \** ***@param path*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* **boolean** open(Path path) **throws** IOException;  
  
 */\*\*  
 \* 添加纹理  
 \** ***@param texture*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* Texture append(Texture texture) **throws** IOException;  
  
 */\*\*  
 \* 移除纹理  
 \** ***@param identifier*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* Texture remove(Identifier identifier) **throws** IOException;  
  
 */\*\*  
 \* 查找纹理  
 \** ***@param identifier*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* Texture find(Identifier identifier) **throws** IOException;  
  
 */\*\*  
 \* 查找纹理  
 \*/* Texture find(String textureName) **throws** IOException;  
  
 */\*\*  
 \* 关闭纹理库  
 \*/* **void** close();  
}

## 风格管理接口

一个风格库是一个目录，目录下包含0个或多个风格文件。

### StyleStore

主要操作包括，初始化风格库，打开风格库，添加风格，删除风格，查找风格，关闭风格库等。



## 方案管理接口

### DataSolution

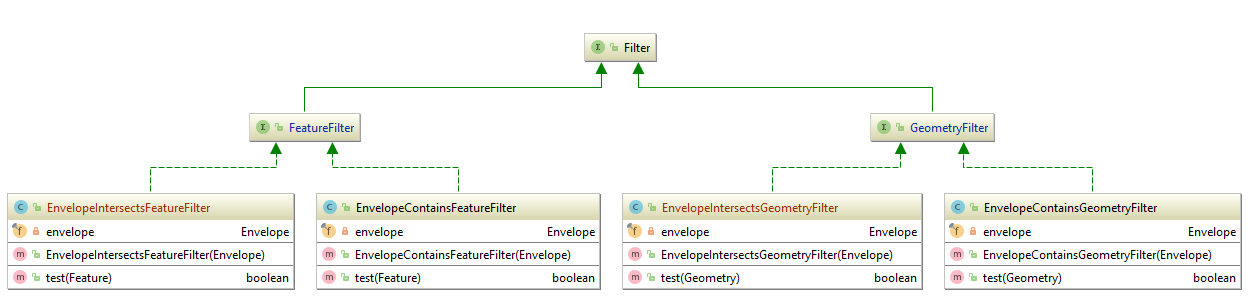
## 场景管理接口

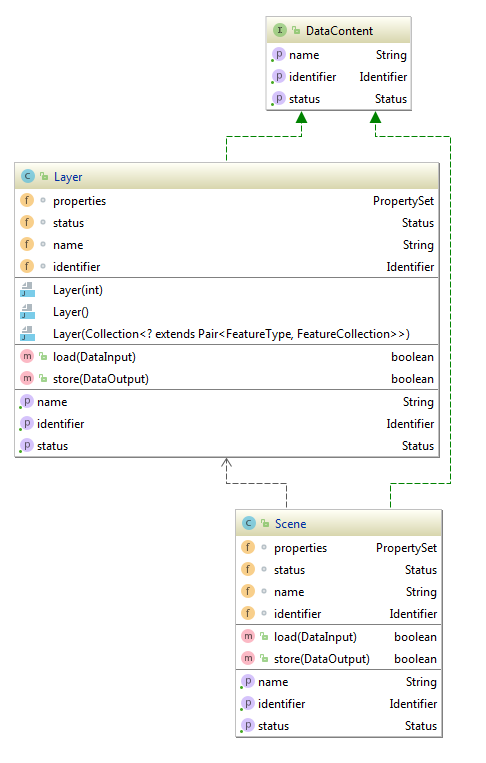
场景Scene管理方案的某个局部范围的数据。例如，在进行可视化的时候，一般当前显示窗口获取的是数据库中的一部分数据，这部分数据采用Scene来进行组织。它是存在于内存中的，可以将其理解为空间数据缓存。

## 过滤查询接口

过滤接口从java.util.function.Predicate继承，实现该接口主要需要实现函数test。

**public interface** Filter<T> **extends** Predicate<T> {  
}



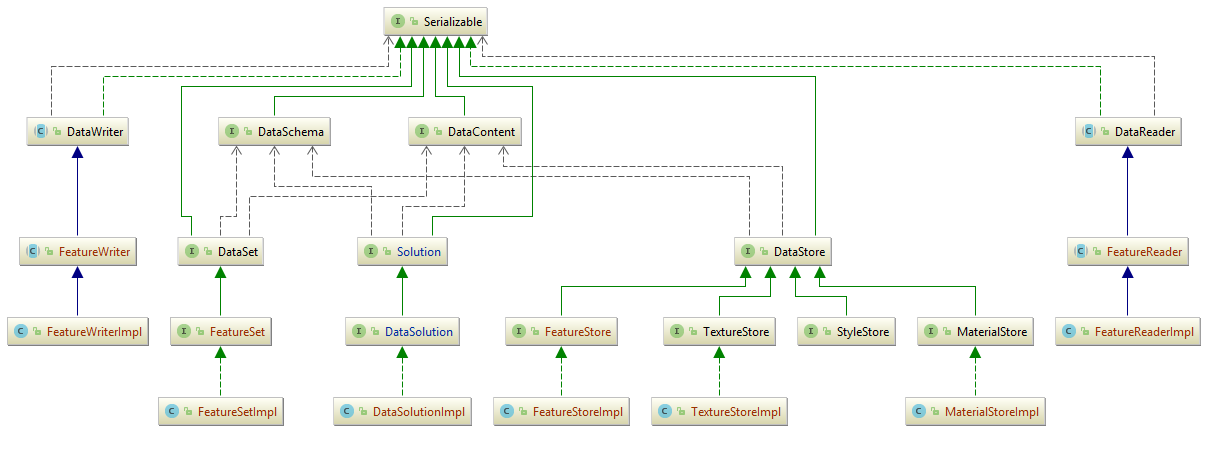


### Layer

### Scene

## 默认的空间数据管理方案

默认的空间数管理方案采用了StorageManager作为存储系统的抽象。StorageManager采用分页机制进行存储管理，包括内存、文件和HDFS等存储系统。主要包括空间要素、纹理数据、材质数据、风格数据、方案数据等多种数据内容的管理。具体实现模块为gtl-data-storage。



### 空间要素的管理

#### FeatureStoreImpl

#### FeatureSetImpl

### 纹理数据的管理

#### TextureStoreImpl

### 材质数据的管理

#### MaterialStoreImpl

### 风格数据的管理

#### StyleStoreImpl

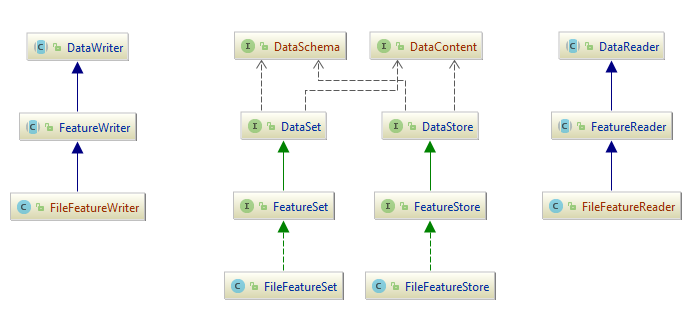
### 方案数据的管理

#### DataSolutionImpl

## 文件版本的空间数据管理方案 （李旸、黄挺，刘玉婷，实现接口，测试）

### 空间要素的管理

文件版本的空间要素数据库主要包括四个类：空间数据库（FileFeatureStore）、空间要素集（FileFeatureSet）、空间要素读取（FileFeatureReader）和空间要素写入（FileFeatureWriter）。这四个接口实现了文件版本的空间数据库的基本读写操作。GTL中任何版本的空间要素数据库，都是对这四个接口的不同实现。如下图所示：



#### FileFeatureStore

文件版本的空间要素库，一个目录，目录名称为数据库的名称，主要包含：

1. 1个要素类型文件（schemas）；
2. 0个或多个空间要素集；每个空间要素集的构成参见FileFeatureSet。

#### FileFeatureSet

每个文件版本的空间要素集包括：

1）1个要素文件（xxx\_features，xxx表示本要素集的ID）;

2）0个或1个索引文件（xxx\_index，xxx表示本要素集的ID）;

3）0个或1个LOD文件（xxx\_lod，xxx表示本要素集的ID）;

### 纹理数据的管理

每个文件版本的纹理库包括：

1. 包含一个texture目录；
2. 目录下包含一个纹理数据集文件，默认名称为data.txt，将所有的纹理id和name存入其中，每行以id+“,“+name的方式存放
3. 目录下还包含一个image文件夹，里面存放各个纹理，每个纹理以id+“+“+name问文件名，以文本文档的方式存放纹理数据，包含id、name、type、data等，以二进制码的方式存放

#### FileTextureStore

### 材质数据的管理

#### FileMaterialStore

### 风格数据的管理

#### FileStyleStore

### 方案数据的管理

文件版本的方案管理是一个目录。该目录下包含：

1. 1个或多个FeatureStore；
2. 0个或一个TextureStore；
3. 0个或一个MaterialStore；
4. 0个或一个StyleStore；

## 内存版本的空间数据管理方案

### 空间要素的管理

### 纹理数据的管理

### 材质数据的管理

### 风格数据的管理

### 方案数据的管理

## HDFS版本的空间数据管理方案

### 空间要素的管理

### 纹理数据的管理

### 材质数据的管理

### 风格数据的管理

### 方案数据的管理

## BEAM版本的空间数据管理方案

### 空间要素的管理

### 纹理数据的管理

### 材质数据的管理

### 风格数据的管理

### 方案数据的管理

## GVX插件设计（黄挺）

本插件实现GVX文件与文件版本的空间数据库的相互转换。见gtl.plugings.gvx包的GVXDataImporter和GVXDataExporter。

### GVXDataImporter

**public boolean** importSolution(DataSolution dataSolution, Path path) **throws** IOException {  
 dataSolution.open(path);  
 *//读取工程属性信息,填充PropertySet* PropertySet propertySet=**null**;  
 dataSolution.setProperties(propertySet);  
 *//每个Map对应一个FeatureStore* FeatureStore featureStore = dataSolution.createFeatureStore(**"features"**);  
 *//读取Map的属性，填充propertySet* featureStore.setProperties(propertySet);  
 *//循环读取每个图层的信息，填充propertySet，并构建FeatureType* {  
 FeatureType featureType=featureStore

.getFeatureTypeBuilder()  
 .setIdentifier(Identifier.*create*())  
 .setName(**"Road"**)  
 .setCoordinateReferenceSystem(**null**)  
 .add(**"geometry"**, LineString.**class**)  
 .add(**"道路名称"**, Variant.***STRING***)  
 .add(**"道路长度"**, Variant.***DOUBLE***)  
 .add(**"道路等级"**,Variant.***INTEGER***)  
 .build();  
 FeatureSet featureSet = featureStore.createFeatureSet(featureType);  
  
 featureSet.setProperties(propertySet);  
 *//循环读取图层中的图元信息，构建Feature* {  
 List<Feature> fs = **new** ArrayList<>();  
 {  
 LineString lineString=(LineString) Geometry.*create*(LineString.**class**,3);  
 lineString.getVertices().add(0,0,0);  
 lineString.getVertices().add(1,1,1);  
 lineString.getVertices().add(2,2,2);  
  
 Feature f = featureStore.getFeatureBuilder(featureType.getName())  
 .setIdentifier(Identifier.*create*())  
 .setName(**"Road\_1"**) *// 要素名称* .add(lineString) *//几何对象* .add(**"八一路"**) *//属性：道路名称* .add(3000.0) *//属性：道路长度* .add(2) *//属性：道路等级* .build();  
  
 fs.add(f);  
 }  
 featureSet.getFeatureWriter().write(fs);  
 }  
 }  
 dataSolution.close();  
 **return true**;  
}

### GVXDataExporter

**public class** GVXDataExporter **implements** DataExporter {  
 @Override  
 **public boolean** exportSolution(DataSolution dataSolution, Path path) **throws** IOException {  
 **return false**;  
 }  
}

## SHP插件设计（吴凡）

### SHPDataImporter

### SHPDataExporter

## GTL版本管理

### VERSION0.1

0.1版本，在gtl.io.Serializable中添加了org.appache.hadoop.io.Writable接口，这样导致common必须依赖于hadoop-common；此外，由于StorageManager有一个HDFS版本的实现，也导致了common必须依赖于hadoop-hdfs。在GeoBeam中采用了hadoop的序列化编码方式，必须依赖org.appache.hadoop.io.Writable。后续版本中，将去除common对hadoop相关类的依赖，将这部分内容放在gtl-data-hdfs中去实现。

## GTL测试环境构建

在gtl.config.Config类中设置了默认的测试数据路径，本地测试数据路径通过下列函数获取：

*/\*\**

*\* 返回本地默认数据目录*

*\** ***@return***

*\*/*

**public static** String getLocalDataDirectory(){

**return** *DEFAULT\_DATA\_DIR*+File.***separator***+**"dat"**;

}

*/\*\**

*\* 返回本地默认临时数据目录*

*\** ***@return***

*\*/*

**public static** String getLocalTemporaryDirectory(){

**return** *DEFAULT\_DATA\_DIR*+ File.***separator***+**"temp"**;

}

*/\*\**

*\* 返回本地默认数据交换目录*

*\** ***@return***

*\*/*

**public static** String getLocalSwapDirectory(){

**return** *DEFAULT\_DATA\_DIR*+File.***separator***+**"swap"**;

}

*/\*\**

*\* 返回本地交换文件名称（全名）*

*\** ***@return***

*\*/*

**public static** String getLocalSwapFile(){

**return** *getLocalSwapDirectory*()+File.***separator***+ UUID.*randomUUID*().toString();

}

gtl.config.Config.getLocalDataDirectory()函数返回的是当前用户的home目录。

在Linux操作系统下，如果用户名为vincent，则该函数返回的本地路径为/home/vincent/gtl/data/dat。

在Windows操作系统下，如果用户名为vincent，则该函数返回的本地路径为c:/Users/vincent/gtl/data/dat。

所以，https://112.74.79.128/svn/gtl/trunk/data 上的测试数据，请签到当前用户home目录下的gtl目录，例如在windows下面，当前登录用户名为vincent，则其需要checkout到C:\Users\vincent\gtl\data目录；如果是在linux下或MacX OS，需要签到~/gtl/data目录下或/home/vincent/gtl/data。请将自己的测试数据一并上传，否则别人在打包测试的时候会由于数据位置不对或缺失导致失败。

如果要获取本地临时数据路径，则通过gtl.config.Config.getLocalTemporaryDirectory()获取，linux下为

/home/vincent/gtl/data/temp，

在Windows操作系统下为

c:/Users/vincent/gtl/data/temp。

如果要获取本地数据交换路径，则通过gtl.config.Config.getLocalSwapDirectory()获取，

linux下为/home/vincent/gtl/data/swap，

在Windows操作系统下为c:/Users/vincent/gtl/data/swap。

对于HDFS 集群，不管是本地的伪分布式集群，还是HP调试集群，还是天河2集群，统一命名为hdfs://geosciences。测试数据统一放在hdfs://geosciences/gtl/data目录下。可以通过下列函数获取

*/\*\**

*\* 返回HDFS的默认数据目录*

*\** ***@return***

*\*/*

**public static** String getHdfsDataDirectory(){

**return** *DEFAULT\_HDFS\_DIR*+File.***separator***+**"dat"**;

}

*/\*\**

*\* 返回HDFS的默认临时数据目录*

*\** ***@return***

*\*/*

**public static** String getHdfsTemporaryDirectory(){

**return** *DEFAULT\_HDFS\_DIR*+ File.***separator***+**"temp"**;

}

*/\*\**

*\* 返回HDFS的默认数据交换目录*

*\** ***@return***

*\*/*

**public static** String getHdfsSwapDirectory(){

**return** *DEFAULT\_HDFS\_DIR*+File.***separator***+**"swap"**;

}

*/\*\**

*\* 返回HDFS的默认数据交换文件名称（全名）*

*\** ***@return***

*\*/*

**public static** String getHdfsSwapFile(){

**return** *getHdfsSwapDirectory*()+File.***separator***+ UUID.*randomUUID*().toString();

}

该目录与本地目录一致。