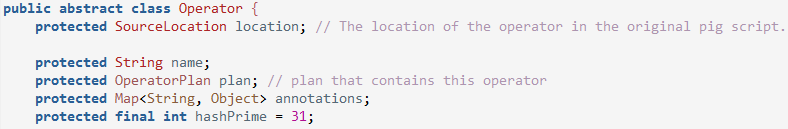
# Architecture

关键类/接口关系图



下面对关键类/接口具体实现做分析

# Operator



Operator的变量：

对name和plan提供get函数，构造函数传入name和plan。

对annotations提供get，annote，remove方法，来得到、添加、移除注释。

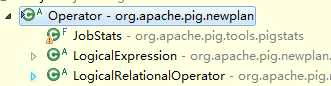
对locaiton提供get和set函数，且构造函数new SourceLocation()。

Operstor抽象方法：

主要方法为accept(PlanVisitor v)，在PlanWalker里常用到。

还提供一个isEqual方法。

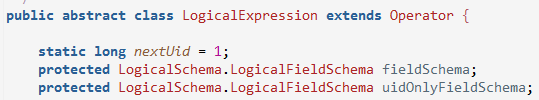
继承结构



主要看两类实现 LogicalExpression和LogicalRelationalOperator

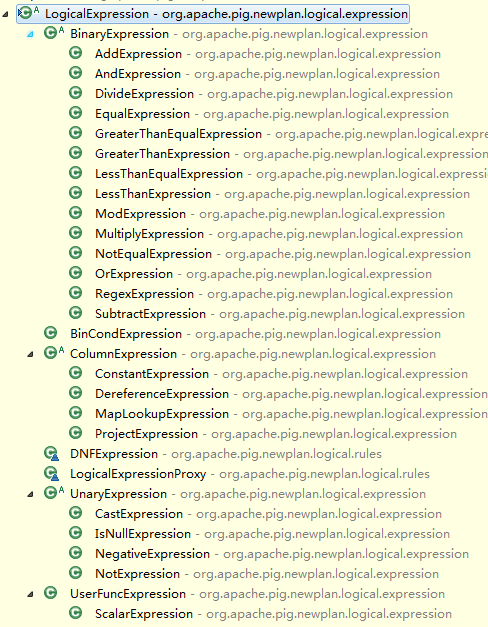
## LogicalExpression

LogicalExpression，



deepCopy()方法需要子类实现

继承结构：



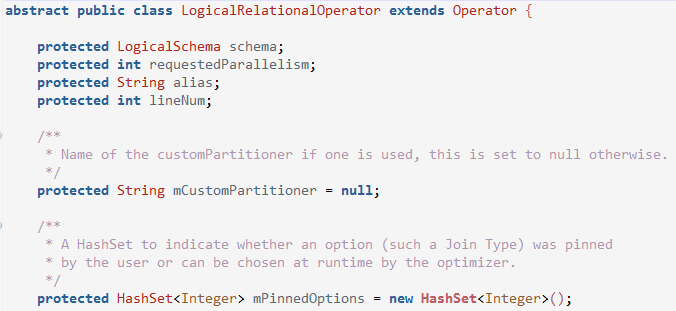
### ConstantExpression

### AddExpression

### AndExpression

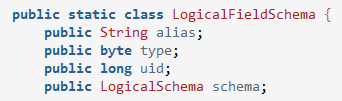
## LogicalRelationalOperator

LogicalRelationalOperator代表关系型操作，关系型操作有Schema。以下是主要变量，LogicalRelationalOperator为他们提供了一些get/set方法。

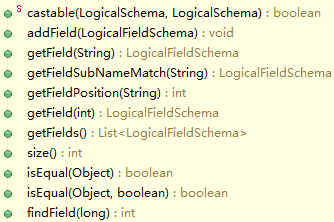


关于LogicalSchema类：

内部类LogicalFieldSchema具体表示每一个field的结构，可以看到与LogicalSchema是嵌套的。LogicalSchema维护一个List<LogicalFieldSchema>



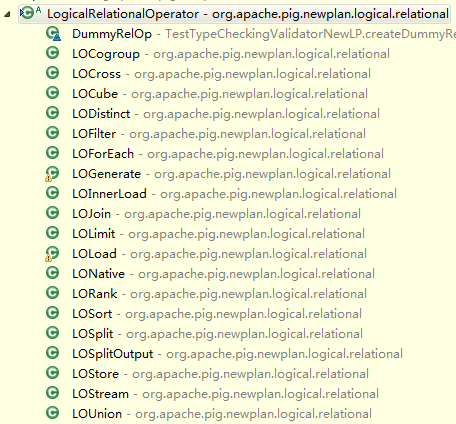
提供基本方法如下：



除了基本方法外，还提供一套merge schema的方法，举例：



继承结构：



### LOSplit

### LOSplitOutput

与LOSplit的区别

### LOStore

### LOForEach

### LOLimit

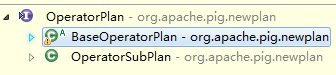
# OperatorPlan

OperatorPlan是一个接口，定义了对Operator的图操作(Graph Operations)。

罗列了所有方法之后发现，Operator类虽然没有结构，只是一个普通的VO类。但是OperatorPlan这个接口定义的以下这套图操作，使Operstor与Operator组成了一个Graph。

|  |  |  |
| --- | --- | --- |
| 类别 | 方法 | 备注 |
| 图结构有关的get方法，都返回List<Operator> | getSources();  getSinks();  getPredecessors(Operator op);  getSuccessors(Operator op);  getSoftLinkPredecessors(Operator op);  getSoftLinkSuccessors(Operator op); | 得到的是根节点、叶子节点、节点的前辈们和后辈们的操作。 |
| 添加删除节点(不涉及到Edge) | **void** add(Operator op);  **void** remove(Operator op) |  |
| connect应该就是指建立Edge | **void** connect(Operator from, **int** fromPos, Operator to, **int** toPos);  **void** connect(Operator from, Operator to);  Pair<Integer, Integer> disconnect(Operator from, Operator to) | 得到的Pair是两个position值 |
| SoftLink相关的操作 | **void** createSoftLink(Operator from, Operator to);  **void** removeSoftLink(Operator from, Operator to); |  |
| 用于遍历所有Operators | Iterator<Operator> getOperators(); |  |
| 有相同operators且结构相同，则Equal | **boolean** isEqual( OperatorPlan other ) |  |
| 所有的connection会指向新的Operator | **void** replace(Operator oldOperator, Operator newOperator) |  |
| 移除operator之后，会把他的所有后辈connect到他的前辈，或他的所有前辈connect到他的后辈 | **void** removeAndReconnect(Operator operatorToRemove) |  |
| 插入节点 | **void** insertBetween(Operator pred, Operator operatorToInsert, Operator succ) |  |
| 检测两个Operator之间是否存在path | **boolean** pathExists(Operator load, Operator store); |  |

实现结构：



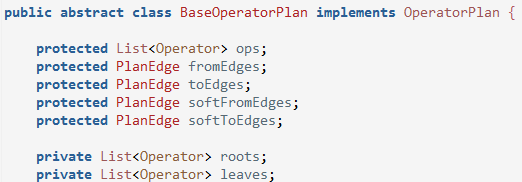
下面展开分析。

## OperatorSubPlan

OperatorSubPlan代表的是一个OperatorPlan的一个子集的视图，OperatorSubPlan只有一个实现，使用在Rule的match过程里。所以OperatorSubPlan的作用就是提供一个子Plan，用于匹配操作。

## BaseOperstorPlan

BaseOperatorPlan实现了OperatorPlan接口，具体实现了各个图操作方法，把Operator之间的关系（包括softLink关系）用PlanEdge表示，图操作方法都借助PlanEdge类表达和实现。



比如：

toEdges.**get**(op) 返回op的前辈

toEdges.**get**(op) == **null** 的op为root

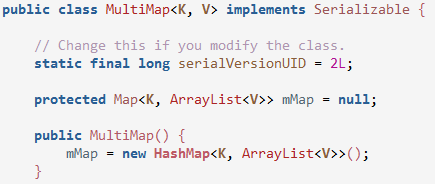
fromEdges.**get**(op) 返回op的后辈

fromEdges.**get**(op) == **null** 的op为leave

PlanEdge类的实现：

**public** **class** PlanEdge **extends** MultiMap<Operator, Operator>

这里的MultiMap是Pig自己的工具类，Pig表示不使用Apache common的MultiMap是因为不支持序列化。



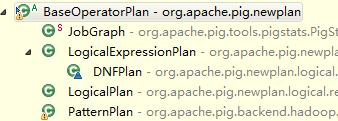
因为MultiMap的value部分使用的是ArrayList，所以使得某些图操作支持position信息，如：

**public** Pair<Integer, Integer> disconnect(Operator from, Operator to)

**public** **void** connect(Operator from, **int** fromPos, Operator to, **int** toPos)

除了实现图操作方法外，BaseOperatorPlan还提供了explain()方法，子类会使用dpumper或printer来打印输出Operators层次结构。

继承结构：



主要看下LogicalPlan和LogicalExpressionPlan两个实现类。

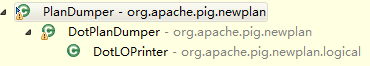
### LogicalPlan

LogicalPlan只包含关系型操作，也就是说涉及到的Operator都是LogicalRelationalOperator。

explain()方法既支持LogicalPlanPrinter的visit实现，也支持DotLOPrinter的dpump实现。

LogicalPlanPrinter是PlanVisitor的子类， LogicalPlanPrinter内部有一个PrintStream，在visit()过程中边遍历，边记录。

DotLOPrinter是DotPlanDumper的子类，DotPlanDumper是PlanDumper的子类，根据graphviz的dot algorithm，输出符合DOT格式的plan。



### LogicalExpressionPlan

LogicalExpressionPlan处理的是LogicalExpressionOperators。

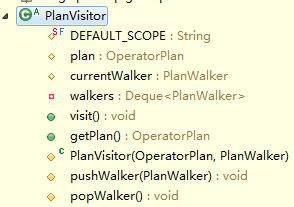
explain()方法借助LogicalPlanPrinter实现

# PlanVisitor

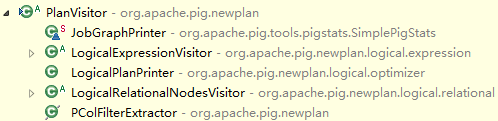
访问者机制，用于操作一个plan。

内部有一个PlanWalker双向队列，PlanWalker会按照某种顺序遍历访问传入的OperatorPlan，让plan的每个operation accept该Visitor。

PlanVisitor可以进行push和pop walker的操作。visit()方法调用的是walker.walk(this)方法。



继承结构很可观

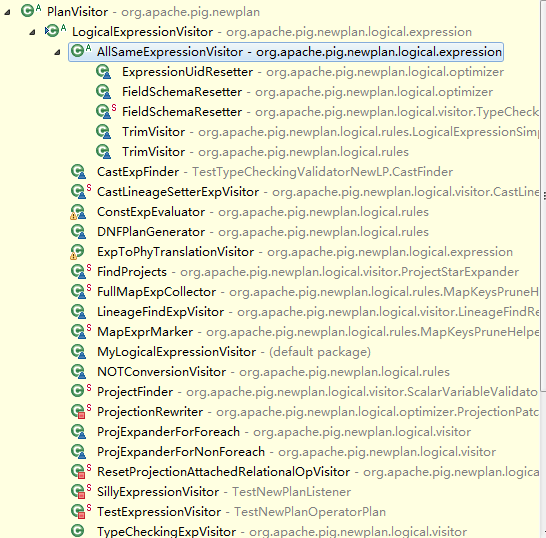


主要看LogicalExpressionVisitor、LogicalRelationalNodesVisitor这两大体系。前者访问expression plans，后者访问logical plans。

## LogicalExpressionVisitor

LogicalExpressionVisitor初始化的时候会判断传入的OperatorPlan是否是LogicalExpressionPlan的子类。visit()方法们通过多态，接受LogicalExpression的子类。

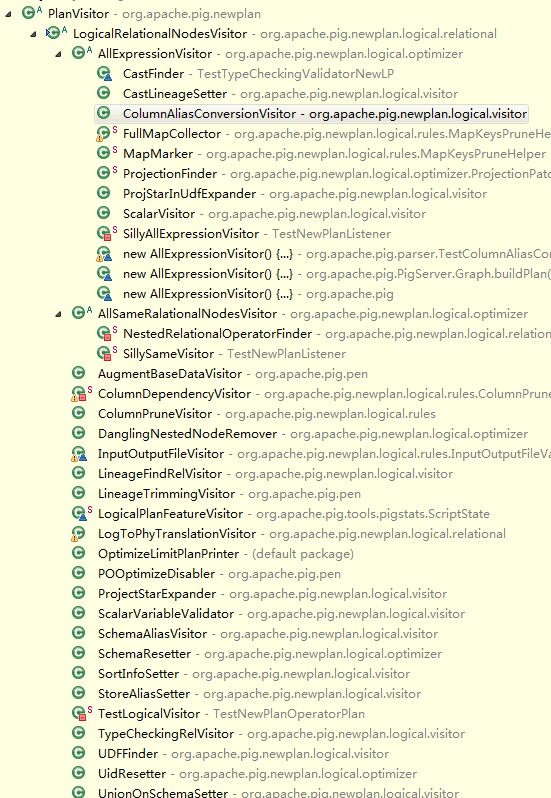
LogicalExpressionVisitor继承结构



## LogicalRelationalNodesVisitor

LogicalRelationalNodesVisitor接受的OperatorPlan必须每个operator都是LogicalRelationalOperator的子类(初始化的时候会得到operator iterator对每个进行校验，不满足就抛异常)。visit()方法们通过多态，接受LogicalRelationalOperator的实现子类。

LogicalRelationalNodesVisitor继承结构



### UidResetter

### SchemaResetter

# PlanWalker

PlanWalker提供的是遍历访问一个plan的能力。

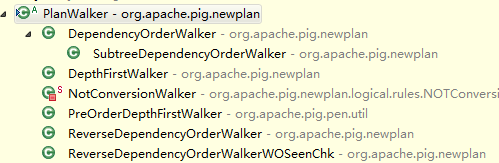
PlanWalker的子类主要实现两个方法：

**public** **abstract** **void** walk(PlanVisitor visitor) **throws** FrontendException;

**public** **abstract** PlanWalker spawnChildWalker(OperatorPlan plan);

walk()方法在子类的实现中，会以不同的顺序遍历plan，最后的结果是遍历到的节点Operator会调op.accept(visitor)接受本Visitor。

继承结构



接下来具体介绍各子类遍历能力的实现。

## DependencyOrderWalker

DependencyOrderWalker按照依赖顺序访问plan，即一个node被访问的前提是它的前辈们已经被访问过了。这个访问顺序相当于，按照拓扑顺序访问图上的节点。

@Override

**public** PlanWalker spawnChildWalker(OperatorPlan plan) {

**return** **new** **DependencyOrderWalker**(plan);

}

walk()方法通过plan.getSinks()方法得到所有的leave节点，即没有后辈的节点，然后遍历他们，获取每个节点的所有前辈，再递归前辈的前辈，从而实现把所有的节点都访问一遍，最后得到结果就是一个FIFO的List。代码里的这个Graph依赖遍历的方式很不高效，但是因为访问的图的节点少，所以可接受。

递归的过程如下



## DepthFirstWalker

DepthFirstWalker是深度优先遍历（由上而下的深度优先）

@Override

**public** PlanWalker spawnChildWalker(OperatorPlan plan) {

**return** **new** **DepthFirstWalker**(plan);

}

walk()方法通过plan.getSources()得到所有的root节点，然后遍历他们，遍历的时候获取他们的所有后辈，递归遍历。

递归过程如下：



## PreOrderDepthFirstWalker

PreOrderDepthFirstWalker即前序深度优先（由下而上的深度优先）

子Walker是深度优先

**public** PlanWalker spawnChildWalker(OperatorPlan plan) {

**return** **new** **DepthFirstWalker**(plan);

}

walk()方法是通过plan.getSinks()得到所有leave节点，然后遍历每个leave节点，获得他的前辈，并递归进行深度优先(向上)遍历。

递归操作如下：



## ReserveDependencyOrderWalker

ReservedDependencyOrderWalker是逆向的依赖顺序遍历，即一个节点访问之后才能访问它依赖的节点，即N节点要想被访问，需要依赖N节点的节点先被访问。

@Override

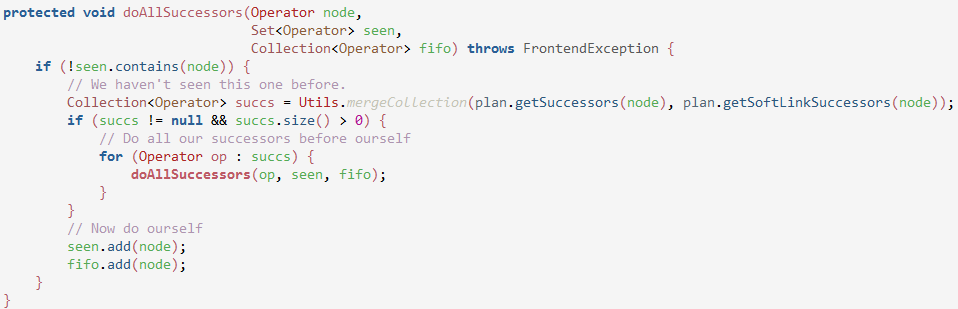
**public** PlanWalker spawnChildWalker(OperatorPlan plan) {

**return** **new** **ReverseDependencyOrderWalker**(plan);

}

walk()方法的访问模式类似DependencyOrderWalker，区别在于先获得所有的root节点，然后进行遍历操作，遍历root节点的所有后辈，递归后辈的后辈，使root节点最后访问。

递归如下：



## ReverseDependencyOrderWalkerWOSeenChk

ReverseDependencyOrderWalkerWOSeenChk也是逆向的依赖顺序遍历，同ReserveDependencyOrderWalker一样。

子Walker是ReserveDependencyOrderWalker

@Override

**public** PlanWalker spawnChildWalker(OperatorPlan plan) {

**return** **new** **ReverseDependencyOrderWalker**(plan);

}

walk()方法和ReserveDependencyOrderWalker的区别在于，每次遍历的时候不记录一个seen的Set<Operator>集。也就是说返回的List里可能会把同一个节点放入多次？没理解WOSeenChk的意思？为什么允许重复？