关于antlr的使用

工具：antlrworks-1.4.jar ，针对antlr3.5的语法

1. ANTLR是ANother Tool for Language Recognition的缩写“又一个语言识别工具”，读[ 'æntlə ]。从名字上可以看出在ANTLR出现之前已经存在其它语言识别工具了（如LEX1，GCC ,YACC2 ）。Antlr通过自己的语法来定义此法规则和语法规则，然后将这些语法规则生成相应的Java/C++代码，分别是一个EELexer.java（词法解析器）和EEParser.java（语法解析器），其中EE是EE.g文件的文件名，这两个文件可以直接拿来使用，具体的demo如下 。
2. Antlr的词法规则

词法是一些正则表达式的东东，可以根据规则将一篇文章切分成一个个的“单词”，又叫token，such as ：

Identifier : ('a'..'z' | 'A'..'Z' | '\_') ('a'..'z' | 'A'..'Z' | '\_' | '0'..'9')\*;

STRING : '\'' (~'\'')\* '\'';

INT : '0'..'9'+;

WS : ( ' ' | '\t' | '\r' | '\n' )+ { Skip(); } ;

其中WS也是一个词法规则，但是会被过滤掉，可以看作是词之间的分隔符 。

词法规则使用大写来表示，一般写在antlr \*\*.g文件的末尾 。

1. Antlr的语法规则

语法规则是将一系列词法规则组合起来用的，语法规则可以嵌套语法规则，但是有个最顶层的语法规则，最终对于输入要生成这个语法规则，不然会报错 。

**语法树重写：**

insertStatement : insertClause selectClause fromClause whereClause?

-> ^(INSERT\_STATEMENT insertClause selectClause fromClause whereClause?);

这种叫做语法重写，是根据相应的识别相应的语法规则，然后重写这个规则的语法树，主要是为了使得结果的语法树更清晰 。

^(\*)表示需要有根节点、叶子节点的语法树，如果不加这个标志的，则解析后的每个token都是一个叶子节点。

其中的第一个token是这棵树的根节点 。

1. Demo

对于一个StormQ1.g文件 ： ，可以使用antlrworks-1.4.jar打开

在antlrwork中，可以使用Generate—>Genterate Code 来生成相应的Java代码，

由于antlrwork的一些bug，生成的Java代码需要改动一些东西 ，修改一个方法的大小写。

然后就可以直接用这两个类了 。

使用的demo如下 ：

String sql="INSERT INTO desttable SELECT aa , bb ,cc FROM log\_alb\_sum WHERE cc>1000 AND bb=dd AND ss IS NULL AND cc IS NOT NULL";

InputStream in =**new** ByteArrayInputStream(sql.getBytes());

ANTLRInputStream input = **new** ANTLRInputStream(in);

Lexer lexer = **new** StormQlLexer(input);

CommonTokenStream tokens = **new** CommonTokenStream(lexer);

StormQlParser parser = **new** StormQlParser(tokens);

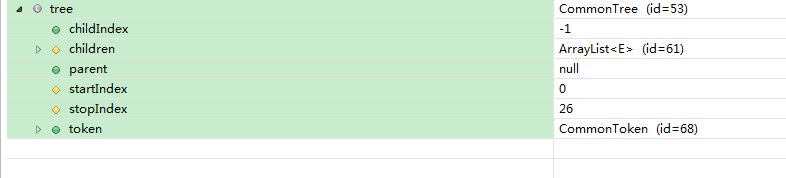
StormQlParser.dmlStatement\_return r =parser.dmlStatement();

Token start=r.start;

Token end=r.stop;

BaseTree tree=(BaseTree)r.tree;

r是解析后返回的一个值，其中最终要的属性值r.tree ，是解析后的语法树的根节点，其结构如下：



主要是children子树 。

Token是表示每个词，其中主要属性是其文本，开始、结束位置以及type，主要是在语法报错时候会用到，type的值在生成的StormQl.tokens会有定义，表示某个token对应的值是多少，主要可以用在后面语法解析的时候，用这个值找到自己定义的token（主要是重写为子树的根节点），然后实现自己的语法解析，一般这种token是用户自己的语法关键点，语义分析也都是遍历这些关键token点（可以参考hive的语法中的token）

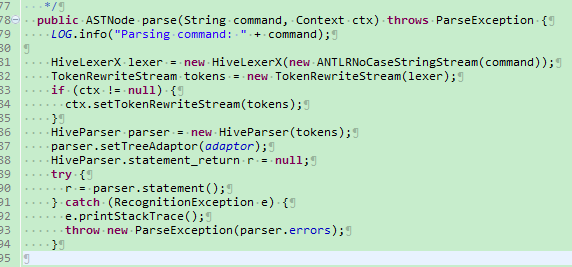
1. HIVE的语法文件

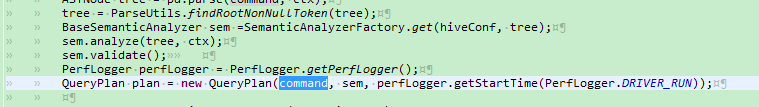


1. 整个sql语句编译、执行的步骤，执行的入口方法是在Driver.run(String command)方法中，执行的参数也就是一个sql字符串 ，主要的方法是：

**int** ret = compile(command); //编译，主要是将sql字符串翻译成ast树，然后翻译成可执行的task树 ，然后再优化执行树

ret = execute(); //执行所有的task

1. **Hive中调用antlr类的代码org.apache.hadoop.hive.ql.parse.ParseDriver类** 返回的HiveParser.statement\_return和上面一样，是棵ast的语法树 ,具体语法树的接口可以参见相应的HiveParse.g文件
2. **得到语法树之后，会根据语法树根节点的类型来选择相应的SemanticAnalyzer**

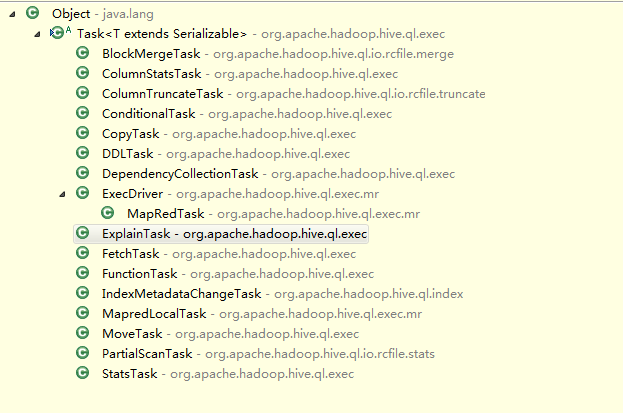
 主要是根据根节点的语法树类型来选择相应的analyzer，具体的选择analyzer代码如下：



对于DDL操所，得到的就是DDLSemanticAnalyzer ，对于一般的insert(hive中存select语句会被翻译成一个insert tmpDirectory的语句)得到的就是SemanticAnalyzer 。

1. **然后调用SemanticAnalyzer.analyze(tree,ctx)来将语法树翻译成可执行的执行计划**

可执行的计划存储在 protected List<Task<? extends Serializable>> rootTasks 属性中， Task的executeTask()方法是可以直接执行的，最终实际的执行也是调用每个task的executeTask方法，依赖以及调度是在上层控制的，Task的集成关系如下：



Task是一个树形结构，每个task有一堆child task ，这些child是在执行顺序上依赖于自己的task ，rootTasks中存储的就是整个执行计划中需要最开始执行的task list ，一棵”倒着的执行依赖树” 。

1. **执行task:Driver.execute()为入口**

****

将可执行的task放入runnable中，初始为root task list ，runnable表示正在运行running的task 。

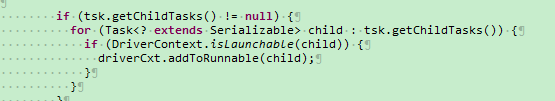
具体的执行流程如下：

* 不断去遍历runnable，选出一个执行launchTask(tsk, queryId, noName, running, jobname, jobs, driverCxt) ，在这个方法中，启动task，其实就是调用task的executeTask() 方法 。



这个里面hive是支持并发执行task的，若是需要并发的话每个task被封装成一个Thread的子类，然后自行启动。

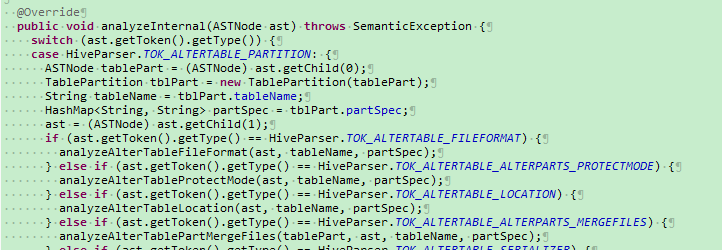
* 找出执行完成的task，然后遍历该task的子task，选出可执行（pre task已经执行完）task 放入runnable中 ，然后重复上一个步骤 。



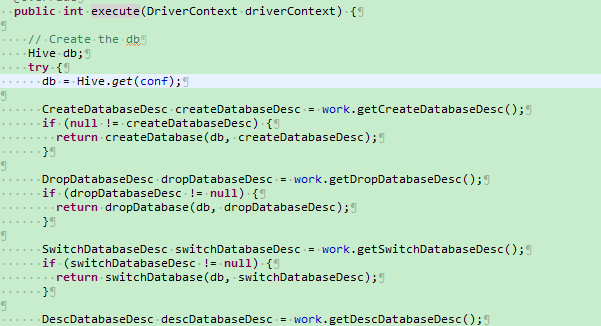
对于一些有多个pre task 的child task，会在最后一个pre task执行完后被启动，所以在这会被在child中过掉。

* 待补充

1. **待补充**
2. 关于DDLSemanticAnalyzer：解析ddl语句并生成相应执行DDLTask来执行



根据相应的ast树类型，生成一个执行该ddl需要信息的对象DDLWork ，DDLWork是一个union的数据结构，里面有各种操作信息的引用，但是只有一个有用的 。随后DDLWork封装到一个DDLTask中，DDLTask执行具体execute方法的时候，在根据DDLWork中得到的值判断具体该执行那一种操作（那种操作需要的信息的引用!=null,则执行哪种引用），具体的执行其实是调用Hive对象相应的方法 。



1. SemanticAnalyzer:将dml的ast树翻译成具体的执行计划的analyzer
2. 其它几个LoadSemanticAnalyzer、ExportSemanticAnalyzer、ImportSemanticAnalyzer较简单，暂未细看
3. 待补充