**Hive源码分析（六）-语法分析器和语义分析器**

2017年05月07日 15:46:58

找到cli的执行main方法：   
https://insight.io/github.com/apache/hive/blob/master/cli/src/java/org/apache/hadoop/hive/cli/CliDriver.java?line=685

public static void main(String[] args) throws Exception {

int ret = new CliDriver().run(args);

System.exit(ret);

}

main方法调用了CliDriver实体的run method：   
在run methond中最后返回的是executeDriver方法

public int run(String[] args) throws Exception {

。。。。。。。。。略 。。。

return executeDriver(ss, conf, oproc);

。。。。。。。。。略

继续跟进executeDriver（）：

private int executeDriver(CliSessionState ss, HiveConf conf, OptionsProcessor oproc)

throws Exception {

。。。。。。。。略

while ((line = reader.readLine(curPrompt + "> ")) != null) {

if (!prefix.equals("")) {

prefix += '\n';

}

if (line.trim().startsWith("--")) {

continue;

}

if (line.trim().endsWith(";") && !line.trim().endsWith("\\;")) {

line = prefix + line;

ret = cli.processLine(line, true);

prefix = "";

curDB = getFormattedDb(conf, ss);

curPrompt = prompt + curDB;

dbSpaces = dbSpaces.length() == curDB.length() ? dbSpaces : spacesForString(curDB);

} else {

prefix = prefix + line;

curPrompt = prompt2 + dbSpaces;

continue;

}

}

。。。。。。。。。。略

executeDriver方法将一条sql用“；”拆分成多条语句，每条语句执行 **ret = cli.processLine(line, true);**

/\*\*

\* Processes a line of semicolon separated commands

\* @param line The commands to process

\* @param allowInterrupting When true the function will handle SIG\_INT (Ctrl+C) by interrupting the processing and

\* returning -1

\* @return 0 if ok

\*/

public int processLine(String line, boolean allowInterrupting) {

。。。。。。。。。略

ret = processCmd(command);

。。。。。。。。。略

然后进入processCmd方法：

public int processCmd(String cmd) {

。。。。。。。。。略。。。。。。。

if (cmd\_trimmed.toLowerCase().equals("quit") || cmd\_trimmed.toLowerCase().equals("exit")) {

。。。。。。。。。略。。。。。。。

} else if (tokens[0].equalsIgnoreCase("source")) {

。。。。。。。。。略。。。。。。。

} else if (cmd\_trimmed.startsWith("!")) {

。。。。。。。。。略。。。。。。。

} else { // local mode

try {

CommandProcessor proc = CommandProcessorFactory.get(tokens, (HiveConf) conf)*;*

ret = processLocalCmd(cmd, proc, ss)*;*

} catch (SQLException e) {

console.printError("Failed processing command " + tokens[0] + " " + e.getLocalizedMessage(),

org.apache.hadoop.util.StringUtils.stringifyException(e))*;*

ret = 1*;*

}

}

ss.resetThreadName()*;*

return ret*;*

首先processCmd判断是不是退出命令，然后是source和“！”开始的特殊命令（非SQL）的处理，最后是sql的处理逻辑，   
CommandProcessor proc = CommandProcessorFactory.get(tokens, (HiveConf) conf);这句生成了一个CommandProcessor ，那么CommandProcessor 是个什么呢？进入get方法看看：

public static CommandProcessor get(String[] cmd, HiveConf conf)

throws SQLException {

CommandProcessor result = getForHiveCommand(cmd, conf);

if (result != null) {

return result;

}

if (isBlank(cmd[0])) {

return null;

} else {

if (conf == null) {

return new Driver();*//此处返回的是一个Driver，即Driver是CommandProcessor 的下属类型。*

}

Driver drv = mapDrivers.get(conf);

if (drv == null) {

drv = new Driver();

mapDrivers.put(conf, drv);

} else {

drv.resetQueryState();

}

drv.init();

return drv;

}

}

所以

CommandProcessor proc = CommandProcessorFactory.get(tokens, (HiveConf) conf);

ret = processLocalCmd(cmd, proc, ss);

这里的proc是一个Driver，进入processLocalCmd：

int processLocalCmd(String cmd, CommandProcessor proc, CliSessionState ss) {

int tryCount = 0;

boolean needRetry;

int ret = 0;

do {

try {

needRetry = false;

if (proc != null) {

if (proc instanceof Driver) {*//一定为true*

Driver qp = (Driver) proc;

PrintStream out = ss.out;

long start = System.currentTimeMillis();

if (ss.getIsVerbose()) {

out.println(cmd);

}

qp.setTryCount(tryCount);

ret = qp.run(cmd).getResponseCode();*//此处调用的是Driver的run*

if (ret != 0) {

qp.close();

return ret;

}

。。。。。。。。。。略

进入run方法

@Override

public CommandProcessorResponse run(String command)

throws CommandNeedRetryException {

return run(command, false);

}

public CommandProcessorResponse run(String command, boolean alreadyCompiled)

throws CommandNeedRetryException {

CommandProcessorResponse cpr = runInternal(command, alreadyCompiled);

if(cpr.getResponseCode() == 0) {

return cpr;

}

SessionState ss = SessionState.get();

if(ss == null) {

return cpr;

}

MetaDataFormatter mdf = MetaDataFormatUtils.getFormatter(ss.getConf());

if(!(mdf instanceof JsonMetaDataFormatter)) {

return cpr;

}

CommandProcessorResponse cpr = runInternal(command, alreadyCompiled);执行sql的编译和返回结果：

private CommandProcessorResponse runInternal(String command, boolean alreadyCompiled)

throws CommandNeedRetryException {

。。。。。略

*// compile internal will automatically reset the perf logger*

ret = compileInternal(command, true);

。。。。。。。略

compileInternal方法：

private int compileInternal(String command, boolean deferClose) {

。。。。。。略。。。。

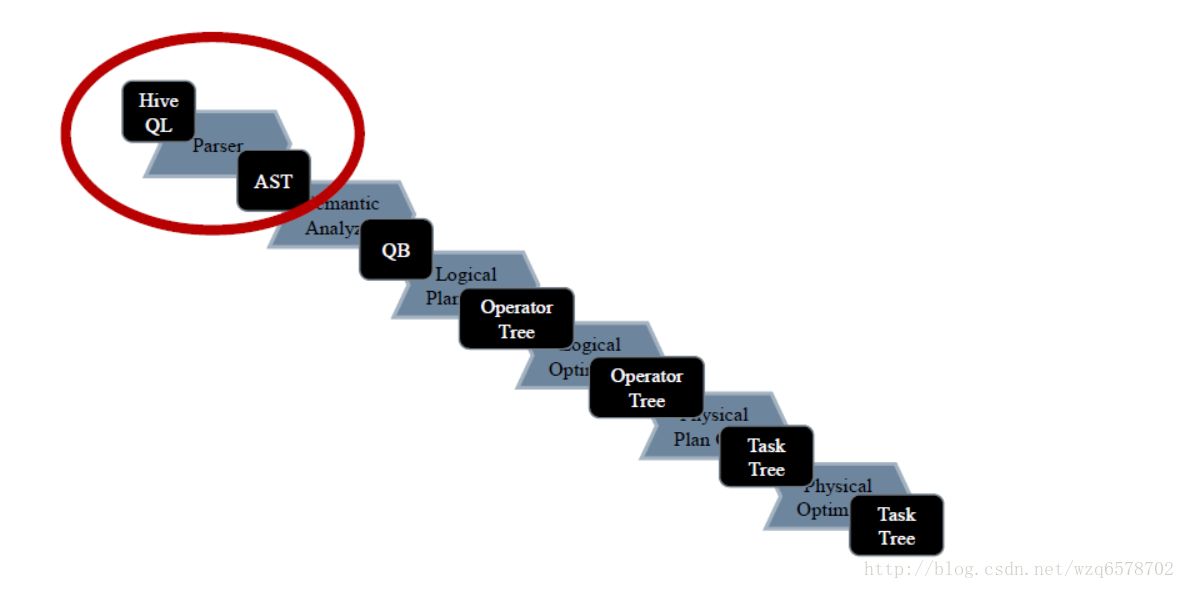
try {

ret = compile(command, true, deferClose);

} finally {

compileLock.unlock();

}

**compile(command, true, deferClose);**   
就是hive的入口了。   
Driver的run方法最终会执行compile()操作,Compiler作语法解析和语义分析。   
回顾一下解析步骤：   
**第一部分：语法分析**   
语法解析Parser   


tree = ParseUtils.parse(command, ctx);【源码】ParseUtils封装了ParseDriver 对sql的解析工作，ParseUtils的parse方法：

/\*\* Parses the Hive query. \*/

public static ASTNode parse(

String command, Context ctx, boolean setTokenRewriteStream) throws ParseException {

ParseDriver pd = new ParseDriver();

ASTNode tree = pd.parse(command, ctx, setTokenRewriteStream);

tree = findRootNonNullToken(tree);

handleSetColRefs(tree);

return tree;

}

ParseDriver 对command进行词法分析和语法解析（统称为语法分析），返回一个抽象语法树AST，进入parseDriver的parse方法：

public ASTNode parse(String command, Context ctx, boolean setTokenRewriteStream)

throws ParseException {

if (LOG.isDebugEnabled()) {

LOG.debug("Parsing command: " + command);

}

HiveLexerX lexer = new HiveLexerX(new ANTLRNoCaseStringStream(command));*//词法分析*

TokenRewriteStream tokens = new TokenRewriteStream(lexer);*//根据词法分析的结果得到tokens的，此时不只是单纯的字符串，而是具有特殊意义的字符串的封装，其本身是一个流。*

if (ctx != null) {

if ( setTokenRewriteStream) {

ctx.setTokenRewriteStream(tokens);

}

lexer.setHiveConf(ctx.getConf());

}

HiveParser parser = new HiveParser(tokens);

if (ctx != null) {

parser.setHiveConf(ctx.getConf());

}

parser.setTreeAdaptor(adaptor);

HiveParser.statement\_return r = null;

try {

r = parser.statement();

} catch (RecognitionException e) {

e.printStackTrace();

throw new ParseException(parser.errors);

}

if (lexer.getErrors().size() == 0 && parser.errors.size() == 0) {

LOG.debug("Parse Completed");

} else if (lexer.getErrors().size() != 0) {

throw new ParseException(lexer.getErrors());

} else {

throw new ParseException(parser.errors);

}

ASTNode tree = (ASTNode) r.getTree();*//生成AST返回*

tree.setUnknownTokenBoundaries();

return tree;

}

Antlr对Hive SQL解析的代码如上述代码逻辑，HiveLexerX，HiveParser分别是Antlr对语法文件HiveLexer.g编译后自动生成的词法解析和语法解析类，在这两个类中进行复杂的解析。   
这是解析的第一步，生辰了一个ATS。   
看一下之后的词法分析，   
**词法分析器Lexer - HiveLexerX**   
输入：一堆字符，这里是HiveSQL   
输出：一串Toker，这里是TokenRewriteStream   
也称词法分析器 **Lexical Analyzer(LA)**或者Scanner   
建议翻阅《编译原理》   
上文提到HiveLexer.g，即文法分析依靠一个文件HiveLexer.g：   
文件定义了一些hive的关键字，form、where，数字的定义格式【0–9】，分隔符，比较符之类的etc。每一个关键字都会变成一个token。   
例如：规定hive中以数字或者下划线开头：

CharSetName

:

'\_' (Letter | Digit | '\_' | '-' | '.' | ':' )+

;

如果你对这个规则不满意可以修改它。   
**语法解析 HiveParser:**   
如何获得ASTNode   
HiveParser.statement().getTree()   
https://www.codatlas.com/github.com/apache/hive/master/ql/src/java/org/apache/hadoop/hive/ql/parse/ParseDriver.java?line=197   
HiveParser是Antlr根据HiveParser.g生成的文件   
进入HiveParser .java看到第一行：

// $ANTLR 3.5.2 org/apache/hadoop/hive/ql/parse/HiveParser.g 2017-05-03 10:08:46

此Java文件在2017-05-03被生成的。但是HiveParser.g我们进去看一下：   
用select字句举例：

selectStatement

:

a=atomSelectStatement

set=setOpSelectStatement[$atomSelectStatement.tree]?

o=orderByClause?

c=clusterByClause?

d=distributeByClause?

sort=sortByClause?

l=limitClause?

{

if(set == null){

$a.tree.getFirstChildWithType(TOK\_INSERT).addChild($o.tree)*;*

$a.tree.getFirstChildWithType(TOK\_INSERT).addChild($c.tree)*;*

$a.tree.getFirstChildWithType(TOK\_INSERT).addChild($d.tree)*;*

$a.tree.getFirstChildWithType(TOK\_INSERT).addChild($sort.tree)*;*

$a.tree.getFirstChildWithType(TOK\_INSERT).addChild($l.tree)*;*

}

}

-> {set == null}?

{$a.tree}

-> {o==null && c==null && d==null && sort==null && l==null}?

{$set.tree}

-> ^(TOK\_QUERY

^(TOK\_FROM

^(TOK\_SUBQUERY

{$set.tree}

{adaptor.create(Identifier, generateUnionAlias())}

)

)

^(TOK\_INSERT

^(TOK\_DESTINATION ^(TOK\_DIR TOK\_TMP\_FILE))

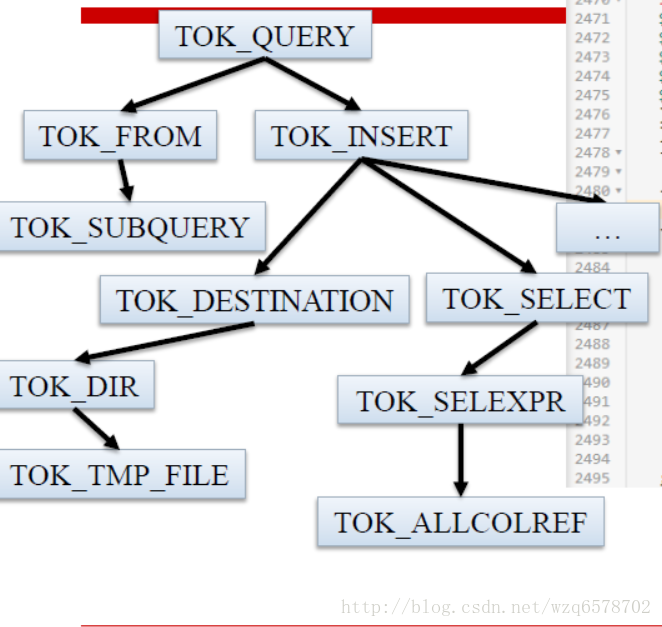
^(TOK\_SELECT ^(TOK\_SELEXPR TOK\_SETCOLREF))

$o? $c? $d? $sort? $l?

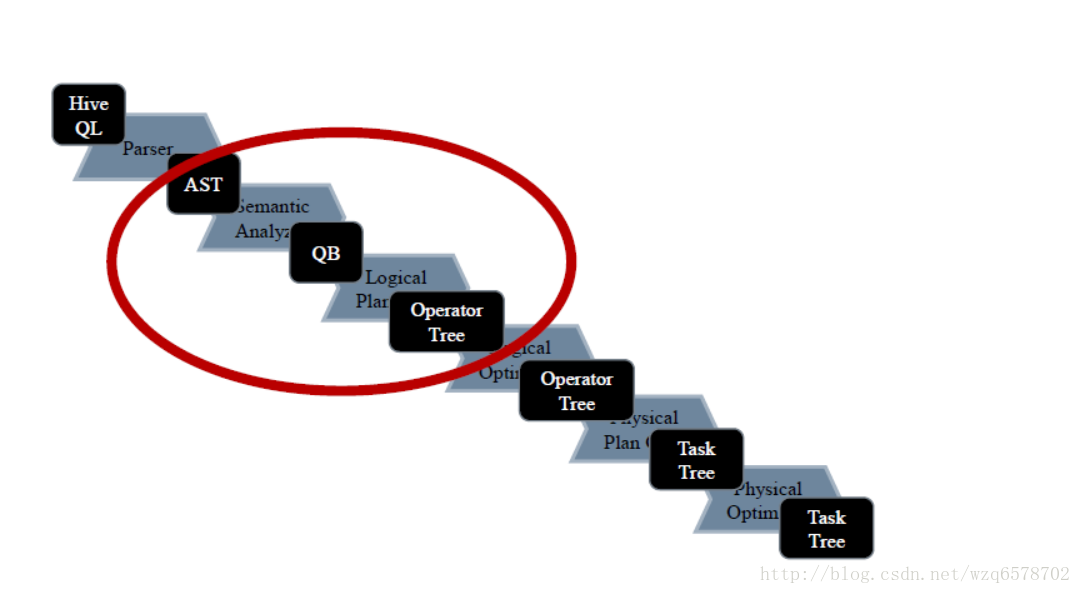
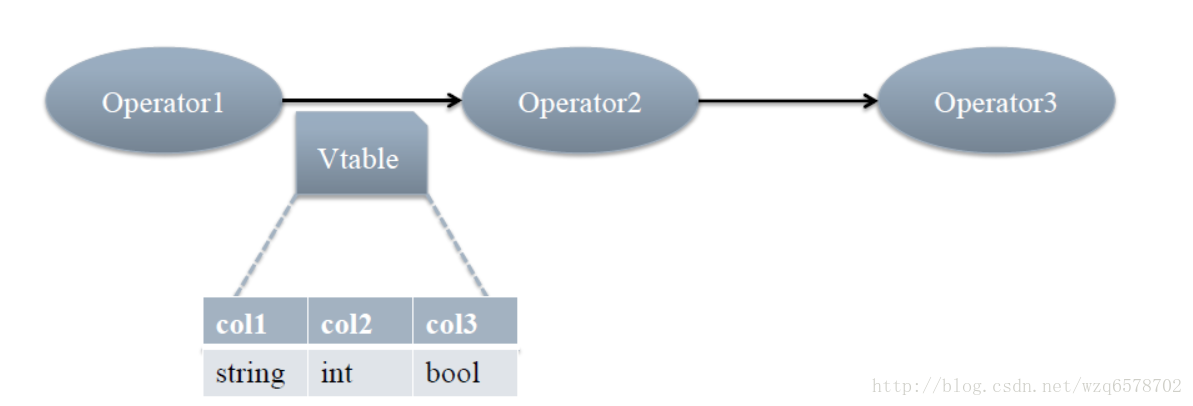
)

)

*;*

用图形表示：   
   
TMP\_FIEL是输出路径，hive是基于mr的上层框架，mr必须要有一个数据文件，mr任务完毕之后结果会存放在TMP\_FIEL此路径下边，然后cli回去读取这个结果文件，展示数据结果。而另一个框架瞅准了hive的这个弱点，没有临时文件，impala边执行边输出结果。

**增加一种语法**这时候，你知道了……   
如果我们想为Hive增加一种新的语法……   
第一步……   
就是修改HiveParser.g   
如果要引入关键字，还需要修改HiveLexer.g

**第二部分：语义解析初步 - SemanticAnalyzer**   
  
**SQL执行顺序**   
一个SQL大致分为以下7部分，按顺序执行   
(5)SELECT  
(6)DISTINCT   
(1)FROM  
(2)WHERE   
(3)GROUP BY   
(4)HAVING   
(7) ORDER BY   
**Operators对应SQL**  
  
**Operator过程**  
ο每个步骤对应一个逻辑运算符(Operator)   
ο每个Operator输出一个虚表(VirtualTable)   
  
**语义解析器 - SemanticAnalyzer**  
ο语义解析器：   
ν输入AST树（见3.3.2）   
ν输出Operator图   
ο回到Compiler代码，看入口在哪里   
νhttps://www.codatlas.com/github.com/apache/hive/master/ql/src/java/org/apache/hadoop/hive/ql/Driver.java?line=503  
νSemanticAnalyzer.analyze()   
νSemanticAnalyzer.analyzeInternal()

|  |
| --- |
|  |

在回到Driver.java:

...........ignore

BaseSemanticAnalyzer sem = SemanticAnalyzerFactory.get(queryState, tree)*;*

............ignore

// Do semantic analysis and plan generation

if (saHooks != null && !saHooks.isEmpty()) {

HiveSemanticAnalyzerHookContext hookCtx = new HiveSemanticAnalyzerHookContextImpl()*;*

hookCtx.setConf(conf)*;*

hookCtx.setUserName(userName)*;*

hookCtx.setIpAddress(SessionState.get().getUserIpAddress())*;*

hookCtx.setCommand(command)*;*

for (HiveSemanticAnalyzerHook hook : saHooks) {

tree = hook.preAnalyze(hookCtx, tree)*;*

}

sem.analyze(tree, ctx)*;*

hookCtx.update(sem)*;*

for (HiveSemanticAnalyzerHook hook : saHooks) {

hook.postAnalyze(hookCtx, sem.getAllRootTasks())*;*

}

} else {

sem.analyze(tree, ctx)*;*

}

进入sem.analyze(tree,ctx)【https://insight.io/github.com/apache/hive/blob/master/ql/src/java/org/apache/hadoop/hive/ql/parse/BaseSemanticAnalyzer.java?line=255】:

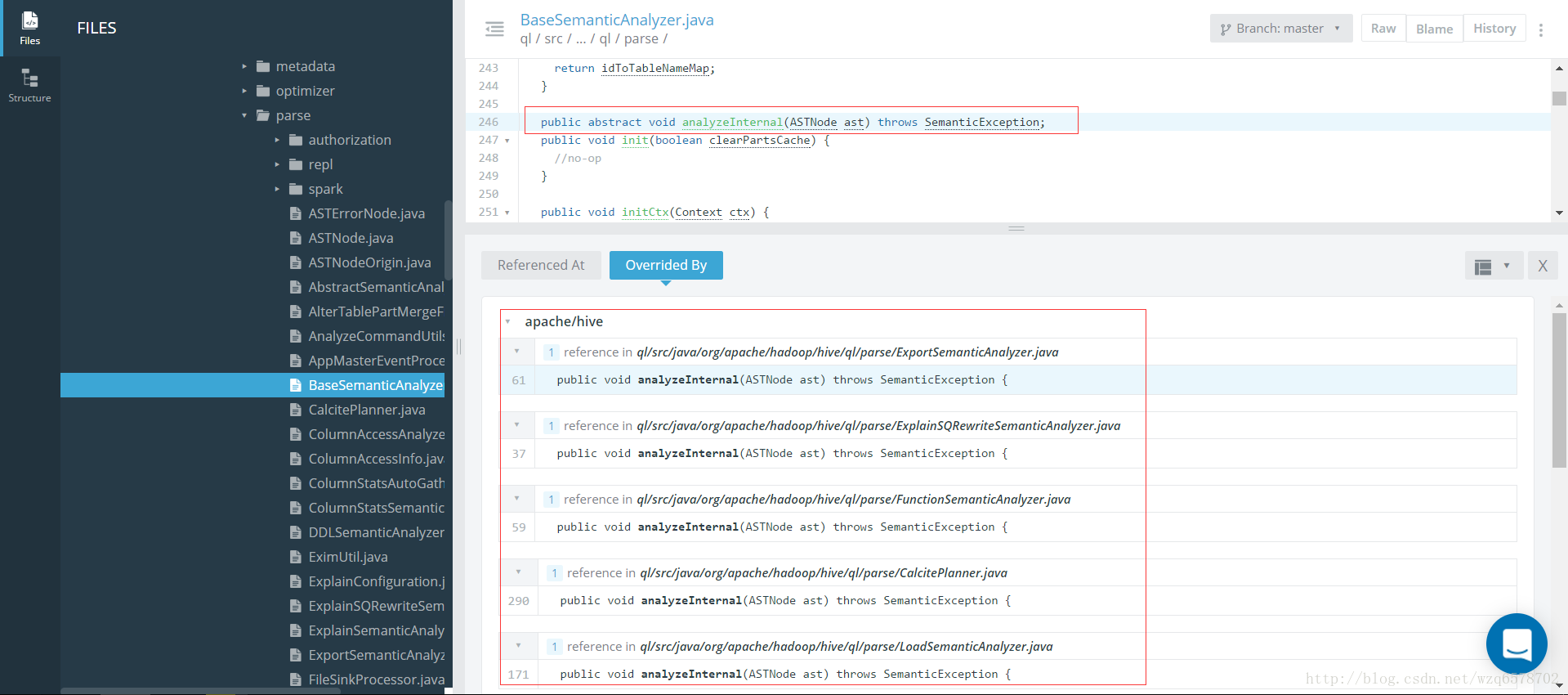
public void analyze(ASTNode ast, Context ctx) throws SemanticException {

initCtx(ctx);*//初始化上下文*

init(true);

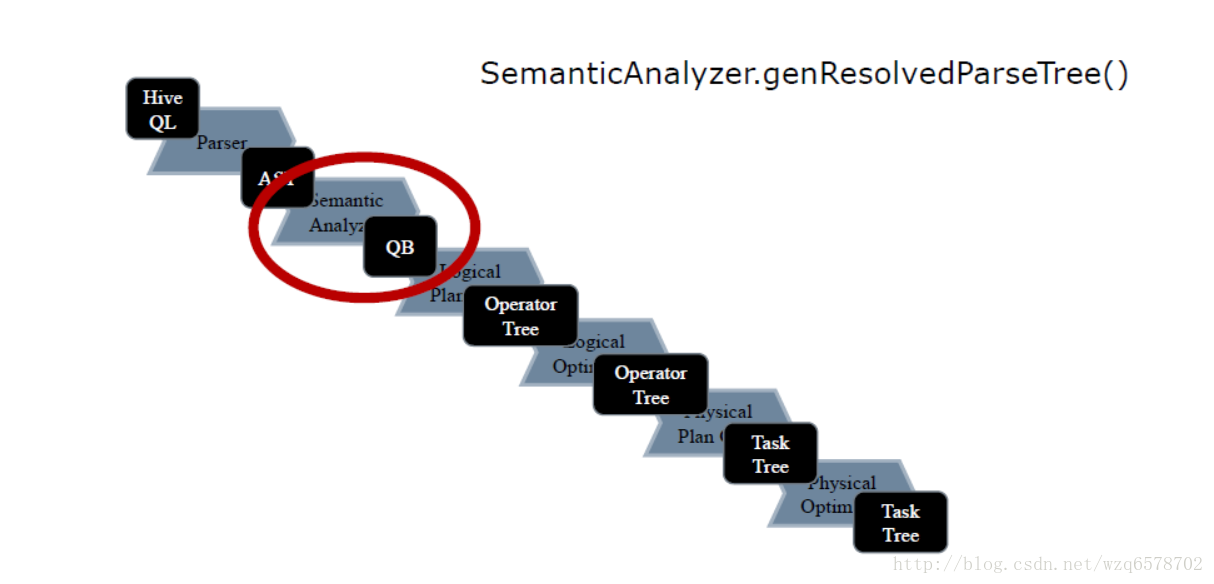
analyzeInternal(ast);*//此方法在BaseSemanticAnalyzer中为一个抽象方法【 public abstract void analyzeInternal(ASTNode ast) throws SemanticException;】*

}

analyzeInternal方法有很多实现：   


**用于查询的 SemanticAnalyzer**   
继承自BaseSemanticAnalyzer的语义分析器有很多种   
其中最重要的是用于查询的SemanticAnalyzer类（很奇怪这种命名，不应该是叫QuerySemanticAnalyzer么？不应该把抽象类的Base一词去掉么？忍吧）   
他们有很多是replaction的，有些是ddl的，有些是做查询的，我们在此处关注做查询的。

**一万多行语义分析**   
看到了么？截止2016年11月20日   
SemanticAnalyzer有13623行   
Hive优化的秘密全在于此   
不要急，慢慢来   
注意：输入的ASTTree后续的QB的生成，逻辑执行计划、逻辑执行计划的优化、物理执行计划的切分、物理执行计划的优化、以及mr任务的生成全部都在这1万多行的代码里边的逻辑中。

**生成QB - genResolvedParseTree()**   


关注SemanticAnalyzer 的 analyzeInterna方法：

public void analyzeInternal(ASTNode ast) throws SemanticException {

analyzeInternal(ast, new PlannerContext());

}

进入analyzeInternal(ast, new PlannerContext());

void analyzeInternal(ASTNode ast, PlannerContext plannerCtx) throws SemanticException {

*// 1. Generate Resolved Parse tree from syntax tree*

LOG.info("Starting Semantic Analysis");

*//change the location of position alias process here*

processPositionAlias(ast);

if (!genResolvedParseTree(ast, plannerCtx)) {

return;

}

。。。。。。。。。。。略。。。。。。

进入genResolvedParseTree(ast, plannerCtx)

boolean genResolvedParseTree(ASTNode ast, PlannerContext plannerCtx) throws SemanticException {

。。。。。。。略。。。。。。。。

// 4. continue analyzing from the child ASTNode.

Phase1Ctx ctx\_1 = initPhase1Ctx();

preProcessForInsert(child, qb);

if (!doPhase1(child, qb, ctx\_1, plannerCtx)) {

// if phase1Result false return

return false;

}

。。。。。。。。。。。。略。。。。。。

如果doPhase1执行成功那么就会得到一个QB，进入doPhase1方法：

/\*\*

\* Phase 1: (including, but not limited to):

\* 1. Gets all the aliases for all the tables / subqueries and makes the

\* appropriate mapping in aliasToTabs, aliasToSubq 2. Gets the location of the

\* destination and names the clause "inclause" + i 3. Creates a map from a

\* string representation of an aggregation tree to the actual aggregation AST

\* 4. Creates a mapping from the clause name to the select expression AST in

\* destToSelExpr 5. Creates a mapping from a table alias to the lateral view

\* AST's in aliasToLateralViews

\*

\* @param ast

\* @param qb

\* @param ctx\_1

\* @throws SemanticException

\*/

@SuppressWarnings({"fallthrough", "nls"})

public boolean doPhase1(ASTNode ast, QB qb, Phase1Ctx ctx\_1, PlannerContext plannerCtx)

throws SemanticException {

。。。。。。。。。。。。。。。。略。。。。。。。。

case HiveParser.TOK\_SELECT:*//select类型的token*

qb.countSel();*//对qb做标记*

qbp.setSelExprForClause(ctx\_1.dest, ast);

。。。。。。。。。。。。。。。。。略。。。。。。

case HiveParser.TOK\_WHERE:*//where类型token*

*//对where的孩子进行处理，为什么是ast.getChild(0)？这个是和之前的HiveParser.g结构相辅相成的。*

qbp.setWhrExprForClause(ctx\_1.dest, ast);

if (!SubQueryUtils.findSubQueries((ASTNode) ast.getChild(0)).isEmpty())

queryProperties.setFilterWithSubQuery(true);

break;

。。。。。。。。。。。。。。。。略。。。。。。。。

case HiveParser.TOK\_GROUPBY:

case HiveParser.TOK\_ROLLUP\_GROUPBY:

case HiveParser.TOK\_CUBE\_GROUPBY:

case HiveParser.TOK\_GROUPING\_SETS:

。。。。。。。。。。。。略。。。。。。。。

if (!skipRecursion) {

*// Iterate over the rest of the children*

int child\_count = ast.getChildCount();

for (int child\_pos = 0; child\_pos < child\_count && phase1Result; ++child\_pos) {

*// Recurse*

phase1Result = phase1Result && doPhase1(

(ASTNode)ast.getChild(child\_pos), qb, ctx\_1, plannerCtx);

}

}

。。。。。。。。。。。。。。。略。。。。。。。。。

参数qb是一个空的QB，在不同case类型下对齐进行填满。   
doPhase1对ASTTree中的每个元素的TOK类型进行case，针对于不同的case对节点数据进行填充。   
for遍历整棵ASTTree，中间对每个元素递归调用doPhase1，这种方式是一种深度优先搜索的算法。   
经过一轮深度优先遍历，不带元数据的QB树就生成了。   
doPhase1执行完毕之后得到QB，QB里边的只是一些关键字还有一些表的名字，但是和hdfs的文件路径对应不起来，所以需要metaData映射关系，之后在SemanticAnalyzer中调用了 getMetaData ：

public void getMetaData(QB qb) throws SemanticException {

getMetaData(qb, false);

}

public void getMetaData(QB qb, boolean enableMaterialization) throws SemanticException {

try {

if (enableMaterialization) {

getMaterializationMetadata(qb);

}

getMetaData(qb, null);

} catch (HiveException e) {

*// Has to use full name to make sure it does not conflict with*

*// org.apache.commons.lang.StringUtils*

LOG.error(org.apache.hadoop.util.StringUtils.stringifyException(e));

if (e instanceof SemanticException) {

throw (SemanticException)e;

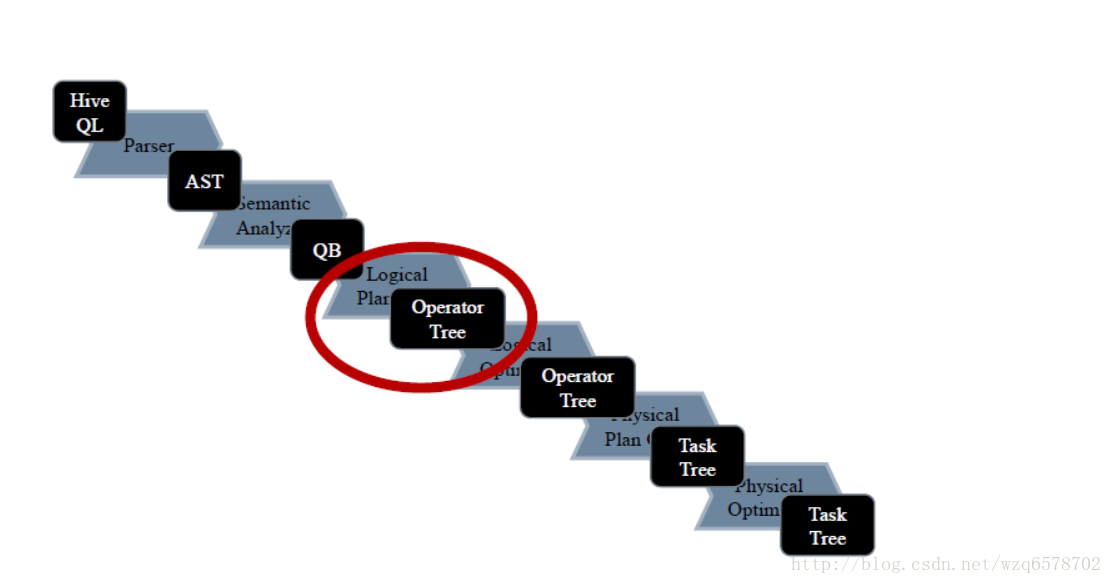
}

throw new SemanticException(e.getMessage(), e);

}

}

getMetaData又会递归的去取元数据（从mysql中），经过doPhase1和getMetaData得到一个完整的QB，接下来就是逻辑执行技术的生成。

**Logical Plan Generator - SemanticAnalyzer.genPlan()**   
  
**Logical Plan Generator - genPlan**   
genPlan()实现QB->Operator   
genPlan() 也是深度优先的递归   
Operator sinkOp = genOPTree(ast, plannerCtx);【https://insight.io/github.com/apache/hive/blob/master/ql/src/java/org/apache/hadoop/hive/ql/parse/SemanticAnalyzer.java?line=11235】生成op：

Operator genOPTree(ASTNode ast, PlannerContext plannerCtx) throws SemanticException {

*// fetch all the hints in qb*

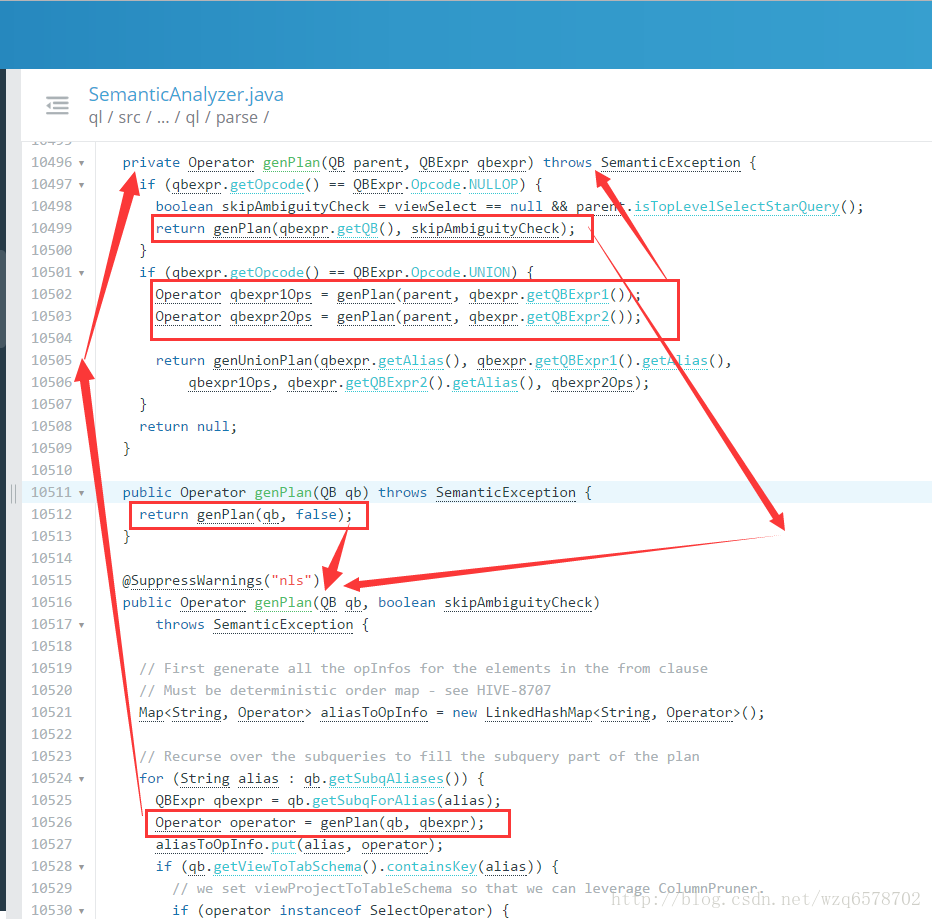
List<ASTNode> hintsList = new ArrayList<>();

getHintsFromQB(qb, hintsList);

getQB().getParseInfo().setHintList(hintsList);

return genPlan(qb);

}

大体的递归过程：   


**表达式分析**   
•类型推倒 100  INT 100.1  DOUBLE ‘Hello’  STRING TRUE  BOOL   
•隐式类型转换 对于fun(DOUBLE, DOUBLE),有输入A—INT, B—DOUBLE fun(double(A), B) 如1+2.5  double(1) + 2.5   
NULL值类型转换   
•表达式求值 f(g(A), B)  A, g(), B, f() 逆波兰表达式   
•BOOL表达式分析 合取范式 (C1 and C2) or C3  (C1 or C3) and (C2 or C3) SELECT \* FROM T,P WHERE (T.A>10 AND P.B<100) OR T.B>10   SELECT \* FROM T,P WHERE (T.A>10 OR T.B>10) AND (P.B<100 OR T.B>10) 当条件变换为合取范式时，可以对AND连接的每一项进行下推优化   
UDFToLong

public LongWritable evaluate(Text i) {

*//有三种情况为null*

*//第一Text是null*

if (i == null) {

return null;

} else {

*//猜测不是数字，返回null*

if (!LazyUtils.isNumberMaybe(i.getBytes(), 0, i.getLength())) {

return null;

}

try {

longWritable.set(LazyLong.parseLong(i.getBytes(), 0, i.getLength(), 10));*//使用LazyLong装换，没有用jdk的API*

return longWritable;

} catch (NumberFormatException e) {

*// MySQL returns 0 if the string is not a well-formed numeric value.*

*// return LongWritable.valueOf(0);*

*// But we decided to return NULL instead, which is more conservative.*

*//出错返回null*

return null;

}

}

}

UDFToInteger

public IntWritable evaluate(Text i) {

*//Text 为null，返回null*

if (i == null) {

return null;

} else {

*//猜测失败，返回null*

if (!LazyUtils.isNumberMaybe(i.getBytes(), 0, i.getLength())) {

return null;

}

try {

intWritable.set(LazyInteger.parseInt(i.getBytes(), 0, i.getLength(), 10));*//使用LazyInteger，未使用jdk的API*

return intWritable;

} catch (NumberFormatException e) {

*// MySQL returns 0 if the string is not a well-formed numeric value.*

*// return IntWritable.valueOf(0);*

*// But we decided to return NULL instead, which is more conservative.*

*//报错返回null*

return null;

}

}

}