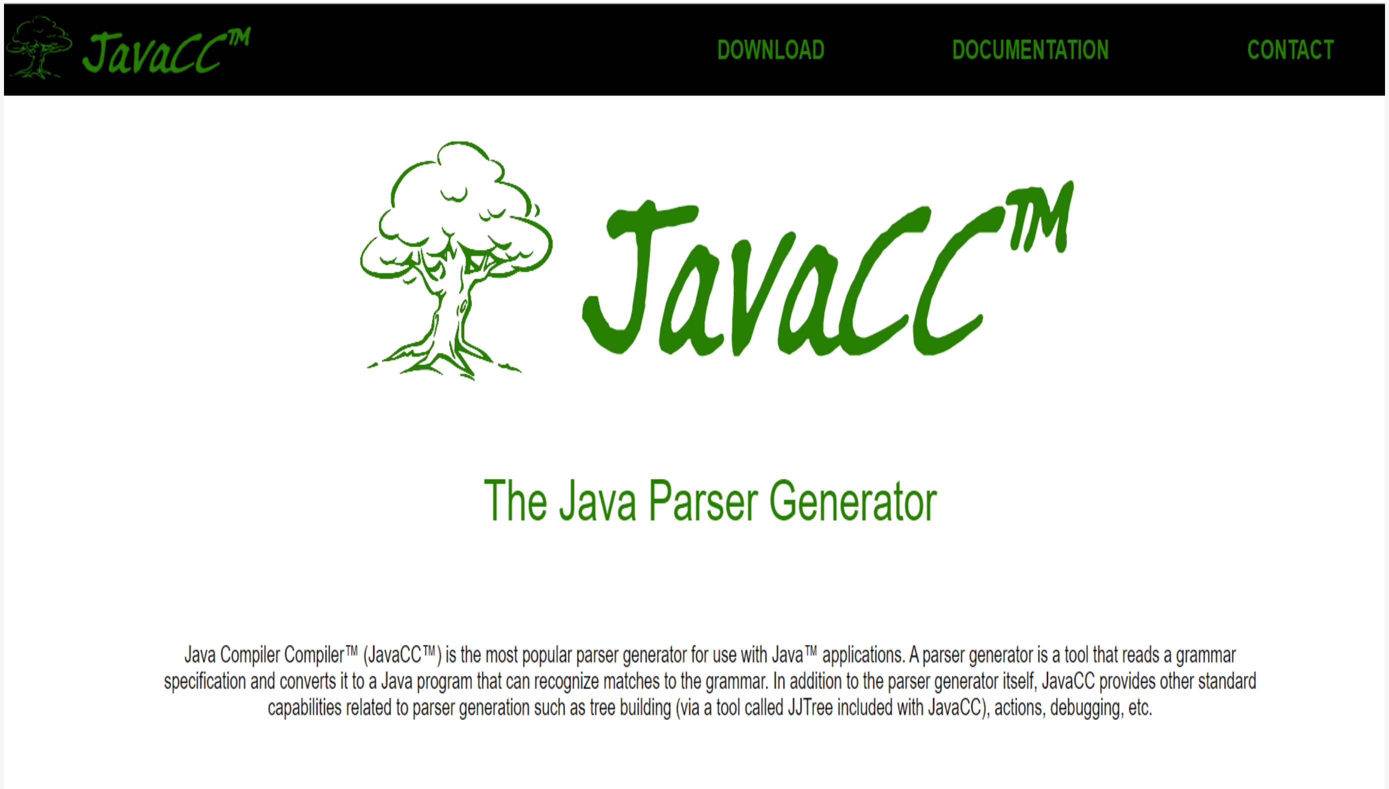
编译实现



The Java Parser Generator

Java Compiler Compiler™ (JavaCC™) is the most popular parser generator for use with Java™ applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building (via a tool called JJTree included with JavaCC), actions, debugging, etc.

Java编译器编译器是用于Java应用程序的最流行的解析器生成器。语法分析器生成器是一种工具，它读取语法规范并将其转换为Java程序，该程序可以识别与语法匹配的内容。除了解析器生成器本身之外，JavaCC还提供了与解析器生成相关的其他标准功能，如树构建（通过JavaCC附带的JJTree椅子）、操作、调试等。

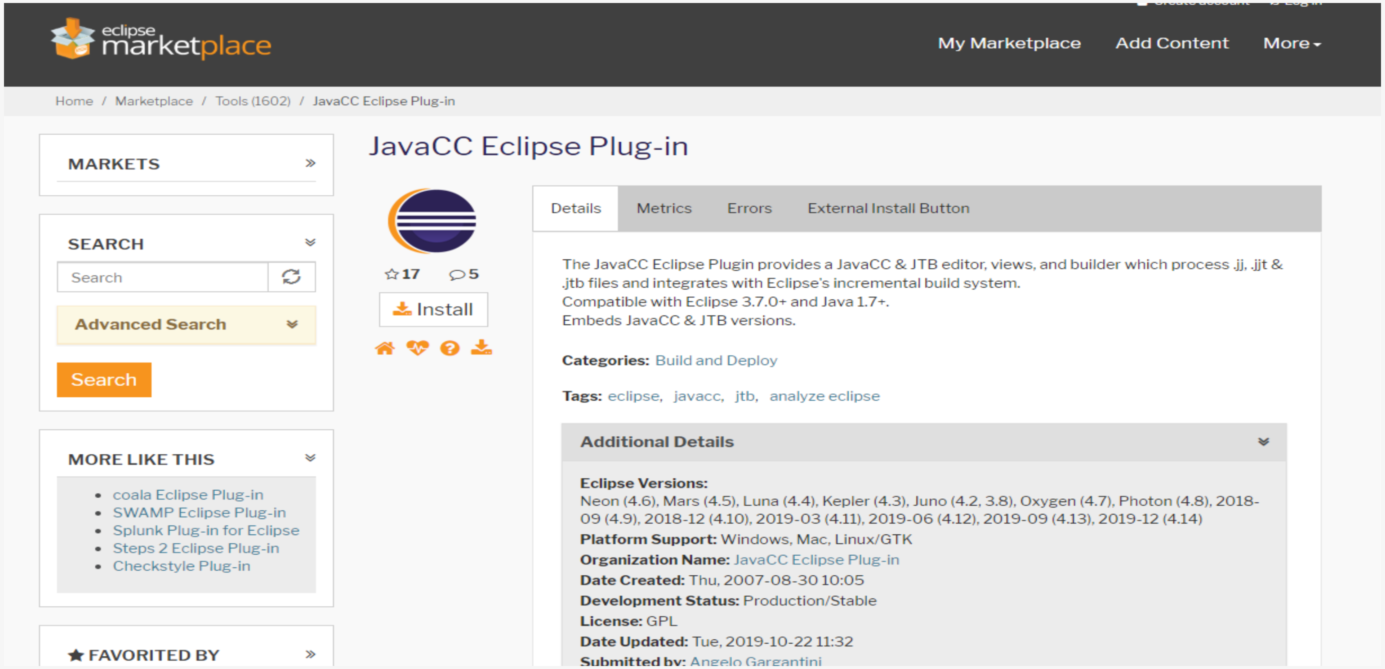
javacc安装

javacc下载地址：https://javacc.org/

（1）eclipse下载地址：https://www.eclipse.org/downloads/

（2）javacc for eclipse下载地址：http://marketplace.eclipse.org/content/javacc-eclipse-plug

（3）资料https://jingyan.baidu.com/article/636f38bb6954dbd6b84610e3.html



JavaCC Eclipse Plug-in

- coala Eclipse Plug-in
- SWAMP Eclipse Plug-in
- Splunk Plug-in for Eclipse
- Steps 2 Eclipse Plug-in
- Checkstyle Plug-in

The JavaCC Eclipse Plugin provides a JavaCC & JTB editor, views, and builder which process .jj, .jjt & .jtb files and integrates with Eclipse's incremental build system.
Compatible with Eclipse 3.7.0+ and Java 1.7+.
Embeds JavaCC & JTB versions.

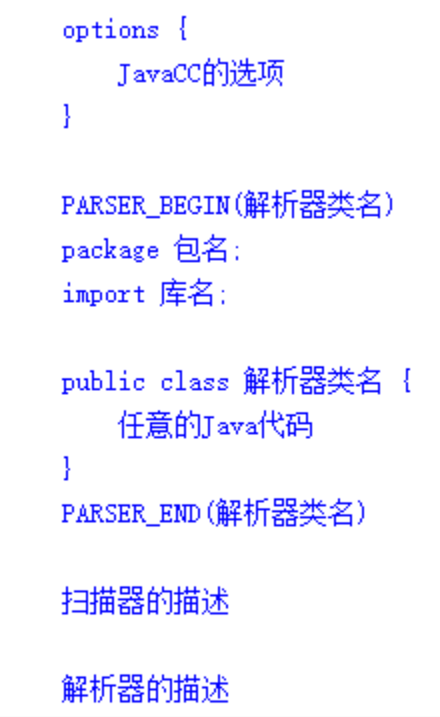
Categories: Build and Deploy

Tags: eclipse, javacc, jtb, analyze eclipse

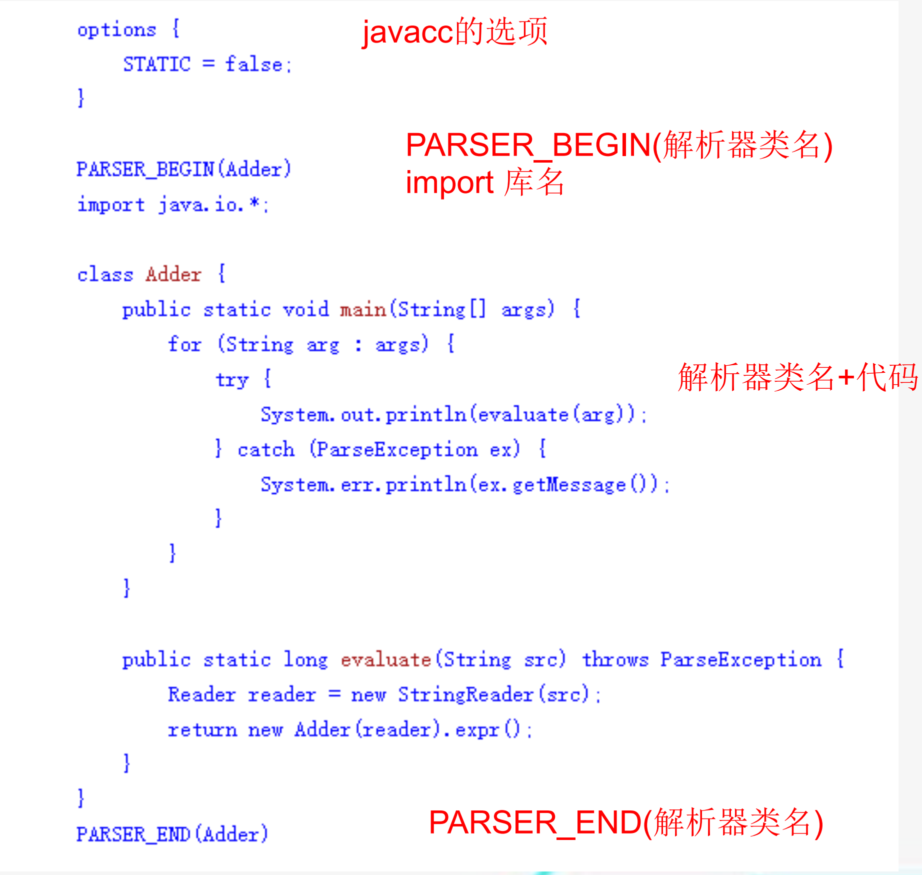
Eclipse Versions:
Neon (4.6), Mars (4.5), Luna (4.4), Kepler (4.3), Juno (4.2, 3.8), Oxygen (4.7), Photon (4.8), 2018-09 (4.9), 2018-12 (4.10), 2019-03 (4.11), 2019-06 (4.12), 2019-09 (4.13), 2019-12 (4.14)
Platform Support: Windows, Mac, Linux/GTK
Organization Name: JavaCC Eclipse Plug-in
Date Created: Thu, 2007-08-30 10:05
Development Status: Production/Stable
License: GPL
Date Updated: Tue, 2019-10-22 11:32
Submitted by: Angelo Gargantini

javacc使用

- javacc的核心是后缀名为jj的语法文件，在按指定规则编写完jj文件后并运行javacc程序，javacc便会根据jj文件生成对应的解析器java文件。
- jj文件的语法结构如下。



```
options {
    JavaCC的选项
}

PARSER_BEGIN(解析器类名)
package 包名;
import 库名;

public class 解析器类名 {
    任意的Java代码
}
PARSER_END(解析器类名)

扫描器的描述

解析器的描述
```



```
options {
    STATIC = false;
}

PARSER_BEGIN(Adder)
import java.io.*;

class Adder {
    public static void main(String[] args) {
        for (String arg : args) {
            try {
                System.out.println(evaluate(arg));
            } catch (ParseException ex) {
                System.err.println(ex.getMessage());
            }
        }
    }

    public static long evaluate(String src) throws ParseException {
        Reader reader = new StringReader(src);
        return new Adder(reader).expr();
    }
}
PARSER_END(Adder)
```

javacc的选项

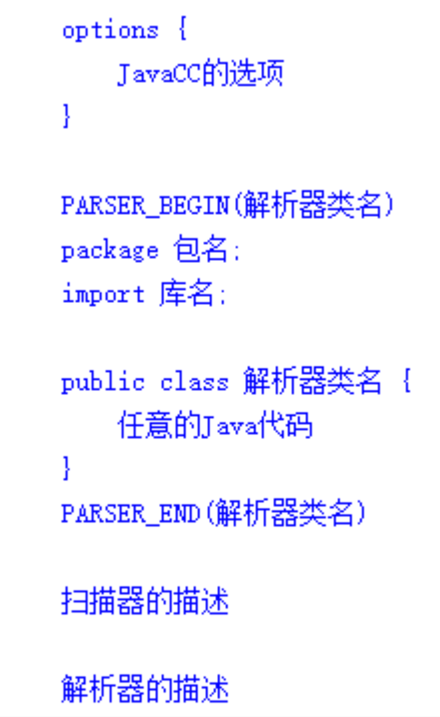
PARSER_BEGIN(解析器类名)
import 库名

解析器类名+代码

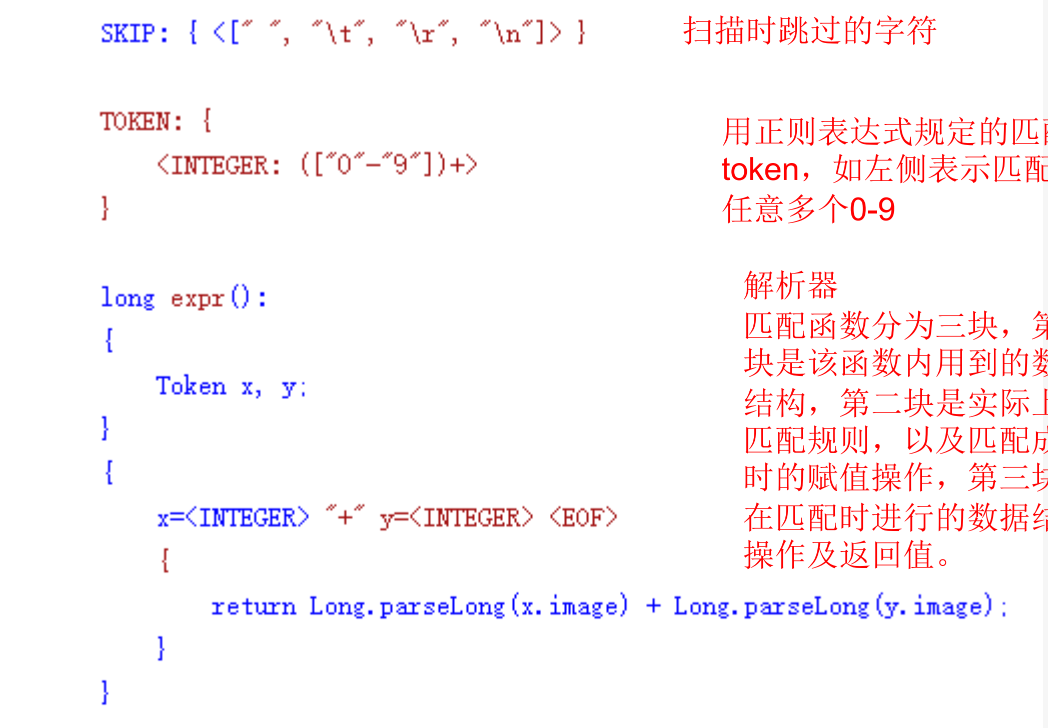
PARSER_END(解析器类名)

javacc使用

- javacc的核心是后缀名为jj的语法文件，在按指定规则编写完jj文件后并运行javacc程序，javacc便会根据jj文件生成对应的解析器java文件。
- jj文件的语法结构如下。



```
options {
    JavaCC的选项
}

PARSER_BEGIN(解析器类名)
package 包名;
import 库名;

public class 解析器类名 {
    任意的Java代码
}
PARSER_END(解析器类名)

扫描器的描述

解析器的描述
```



```
SKIP: { <[" ", "\t", "\r", "\n"]> }

TOKEN: {
    <INTEGER: (["0"-"9"])+>
}

long expr():
{
    Token x, y;
}
{
    x=<INTEGER> "+" y=<INTEGER> <EOF>
    {
        return Long.parseLong(x.image) + Long.parseLong(y.image);
    }
}
```

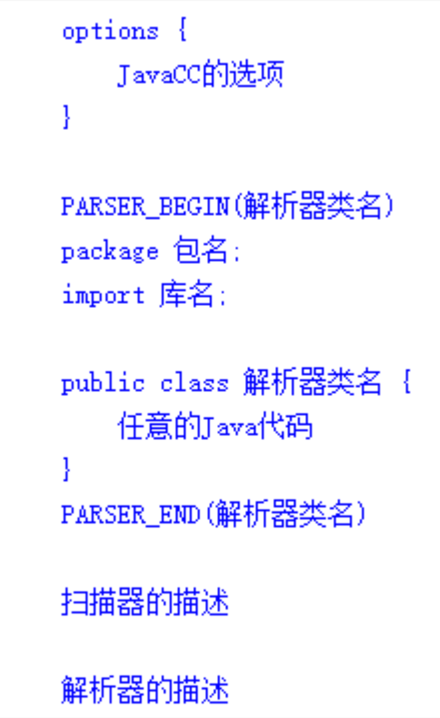
扫描时跳过的字符

用正则表达式规定的匹配token，如左侧表示匹配任意多个0-9

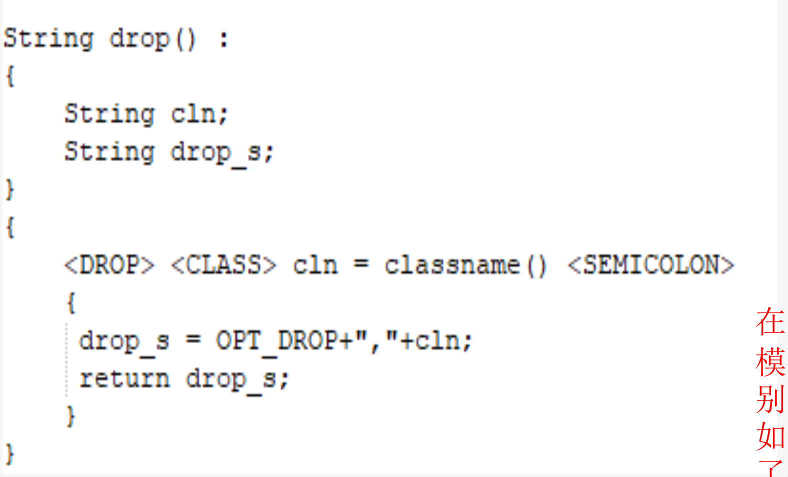
解析器
匹配函数分为三块，第一块是该函数内用到的数据结构，第二块是实际上的匹配规则，以及匹配成功时的赋值操作，第三块是在匹配时进行的数据结构操作及返回值。

javacc使用

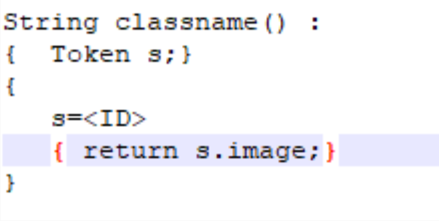
- javacc的核心是后缀名为jj的语法文件，在按指定规则编写完jj文件后并运行javacc程序，javacc便会根据jj文件生成对应的解析器java文件。
- jj文件的语法结构如下。



```
options {
    JavaCC的选项
}

PARSER_BEGIN(解析器类名)
package 包名;
import 库名;

public class 解析器类名 {
    任意的Java代码
}
PARSER_END(解析器类名)

扫描器的描述

解析器的描述
```



```
String drop() :
{
    String cln;
    String drop_s;
}
{
    <DROP> <CLASS> cln = classname() <SEMICOLON>
    {
     drop_s = OPT_DROP+","+cln;
     return drop_s;
    }
}
```



```
String classname() :
{   Token s;}
{
    s=<ID>
    { return s.image;}
}
```

在函数的匹配模块（第二模块），可以递归地调用别的函数，继续进行匹配，如这边就在drop()中调用了classname()继续进行匹配。

语句类型

定义语言

1．创建类：CREATE CLASS
2．创建代理类：CREATE DEPUTYCLASS
3．删除类：DROPCLASS

操作语言

1．插入对象：INSERT INTO
2．删除对象：DELETE FROM
3．更新对象：UPDATE
4．对象的查询：SELECT … FROM

标准格式：

```
CREATE CLASS<class_name>([ATTRIBUTE]({<column><type>}));
```

示例：

CREATE CLASS product (id int, name char , price int);

数据结构：CreateStmt
String NodeTag； ->create origin
String classname；-> product
ArrayList<Relattr> cols; -> id int

数据结构： RelAttr
String attrname;
String attrtype;

标准格式：

```
CREATE SELECTDEPUTY  <class_name>
[ATTRIBUTE({<column><type>})] ##定义实属性
SELECT <attr> <switch_express> AS <attr_name>FROM S WHERE
wherrCluase##定义虚属性
```

示例：

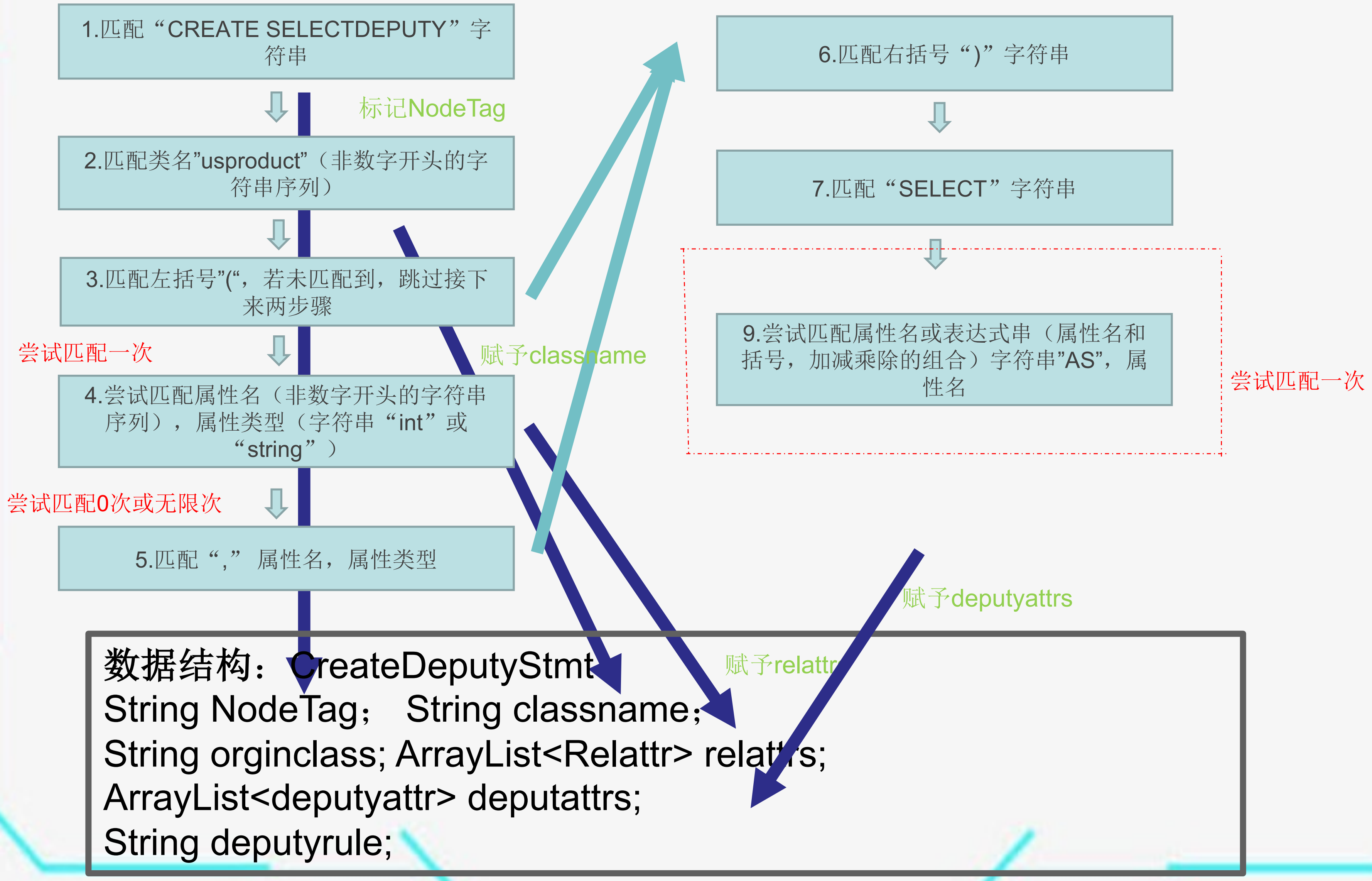
CREATE SELECTDEPUTY usproduct (sales int) SELECT name AS name, (price/7) AS usprice FROM product WHERE price>5000;

数据结构：CreateDeputyStmt
String NodeTag； -> selectdeputy
String classname；-> usproduct
String orginclass; -> product
ArrayList<Relattr> relattrs;-> sales int
ArrayList<deputyattr> deputyattrs;->name name (price/7) usprice
String deputyrule; -> price>5000

数据结构： RelAttr
String attrname;
String attrtype;

数据结构： DeputyAttr
String attrname; usprice
String orginclass; product
String switchrule; (price/7)

CREATE SELECTDEPUTY usproduct (sales int) SELECT name AS name, (price/7) AS usprice FROM product WHERE price>5000;



1.匹配“CREATE SELECTDEPUTY”字符串

标记NodeTag

2.匹配类名”usproduct”（非数字开头的字符串序列）

3.匹配左括号”(“，若未匹配到，跳过接下来两步骤

尝试匹配一次

赋予classname

4.尝试匹配属性名（非数字开头的字符串序列），属性类型（字符串“int”或“string”）

尝试匹配0次或无限次

5.匹配“,”属性名，属性类型

6.匹配右括号“)”字符串

7.匹配“SELECT”字符串

9.尝试匹配属性名或表达式串（属性名和括号，加减乘除的组合）字符串”AS”，属性名

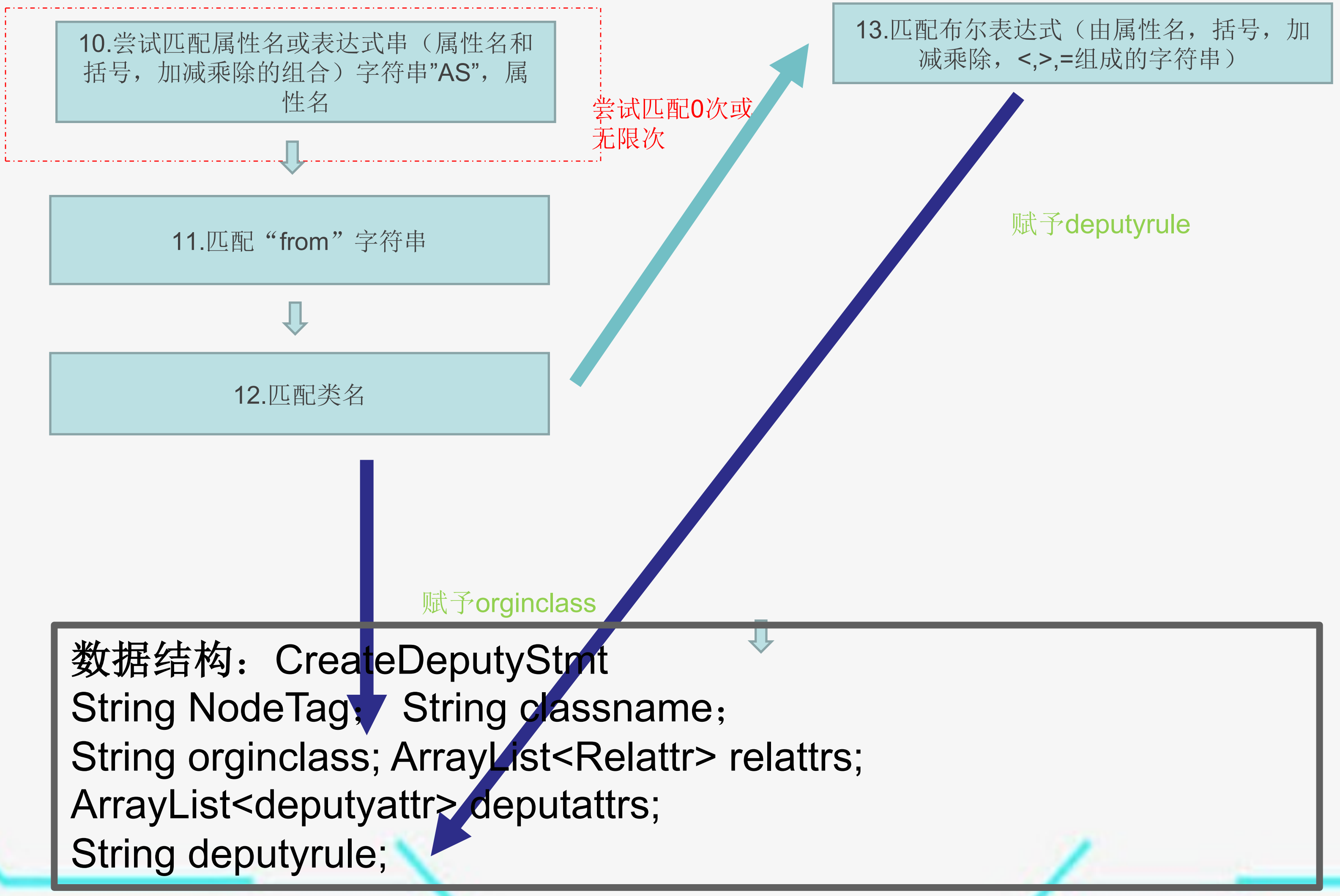
尝试匹配一次

赋予deputyattrs

赋予relattr

数据结构：CreateDeputyStmt
String NodeTag； String classname;
String orginclass; ArrayList<Relattr> relattrs;
ArrayList<deputyattr> deputattrs;
String deputyrule;

示例：
CREATE SELECTDEPUTY usproduct (sales int) SELECT name AS name, (price/7) AS usprice FROM product WHERE price>5000;



10.尝试匹配属性名或表达式串（属性名和括号，加减乘除的组合）字符串”AS”，属性名

尝试匹配0次或无限次

11.匹配“from”字符串

12.匹配类名

13.匹配布尔表达式（由属性名，括号，加减乘除，<,>,=组成的字符串）

赋予deputyrule

赋予orginclass

数据结构：CreateDeputyStmt
String NodeTag； String classname；
String orginclass; ArrayList<Relattr> relattrs;
ArrayList<deputyattr> deputattrs;
String deputyrule;

```
CreateStmt create():
{
    String create_s;
    createstmt = new CreateStmt();
    createselstmt = new CreateSelStmt();
int count;
}
{
  (
    <CREATE> originclass() <SEMICOLON>
  {
    createstmt.NodeTag="CREATEORIGIN";
    return createstmt;
  }
  )
|
    (<CREATE> selectdeputy() <SEMICOLON>)
    {
      createselstmt.NodeTag="CREATEDEPUTY";
      return createselstmt;
    }
}
```

Create语句有两种，一种为创建源类，一种为创建代理类。

为这两种语句分别准备了不同的结构体以及匹配规则

示例： CREATE SELECTDEPUTY usproduct (sales int) SELECT name AS name, (price/7) AS usprice FROM product WHERE price>5000;

```
int selectdeputy() :
{String cln;
RelAttr rl=new RelAttr();
String attr_s;
String attrtype_s;}
{
   (cln = classname() { createselstmt.classname=cln; }  匹配类名
    <LEFT_BRACKET>     匹配左括号
    attr_s = attr() { rl=new RelAttr(); rl.attrname=attr_s; }匹配属性名
    attrtype_s = attrtype() { rl.attrtype=attrtype_s;
createselstmt.relattrs.add(rl); }  匹配属性类型，已完成一个属性匹配，赋予结构体
    (<COMMA> attr_s = attr() { rl=new RelAttr();
rl.attrname=attr_s; } attrtype_s = attrtype() {
rl.attrtype=attrtype_s; createselstmt.relattrs.add(rl); })*
                        带上“,”，重复上述匹配逻辑
<RIGHT_BRACKET>        匹配右括号，结束对实属性的
                       匹配。
```

CREATE SELECTDEPUTY usproduct (sales int)

情况1：有实属性： CREATE SELECTDEPUTY usproduct (sales int) SELECT

```
 <RIGHT_BRACKET>  <SELECT>
directselect() { return 0;}
  )
|
(
   <SELECT>  directselect() {
return 0;}
)
}
```

分两种情况，因为代理类可以有实属性，也可以没有实属性，所以在类名后如果匹配到了左括号，执行一套逻辑，如果直接匹配到了select，就执行另一套逻辑。

情况2：无实属性： CREATE SELECTDEPUTY usproduct SELECT name AS name, (price/7) AS usprice FROM product WHERE price>5000;

```
void directselect() :
{String dattr_s;
String attr_s;
String expr_s;
String value_s = "";
DeputyAttr deputyattr;
String cln;
String cond;}
{(
    expr_s = expression() { deputyattr=new DeputyAttr(); deputyattr.switchrule=expr_s;}   匹配切换表达式

    <AS>   匹配AS字符串
    dattr_s = dattr() {deputyattr.deputyname=dattr_s;   匹配虚属性名字
createselstmt.deputyattrs.add(deputyattr);}
    (<COMMA>  expr_s = expression() {deputyattr=new DeputyAttr();
deputyattr.switchrule=expr_s;}
    <AS> dattr_s = dattr(){deputyattr.deputyname=dattr_s;     带上“,”重复上述匹配过程
createselstmt.deputyattrs.add(deputyattr);})*

    <FROM>        匹配字符串“from”
    cln = classname() {createselstmt.originname=cln; }   匹配类名
    <WHERE>                                               匹配字符串“where”
    cond = condition() {createselstmt.whereclause=cond; }
}                                                          匹配字符串布尔表达式
```

CREATE SELECTDEPUTY usproduct (sales int) SELECT name AS name, (price/7) AS usprice FROM product WHERE price>5000;

删除类

标准格式：

```
DROP CLASS<class_name>
```

示例：

DROP CLASS singer；

数据结构：DropStmt
String NodeTag； ->drop
String classname；-> singer

插入对象

标准格式：

```
INSERT INTO <class_name> <target_attrs_list>
VALUES(<value_list>);
```

示例：

INSERT INTO product (id , name , price) VALUES (1 , "mac" , 14000);

数据结构：InsertStmt

String NodeTag； ->insert

String classname；-> product

ArrayList<String> attrnames; -> id name price

ArrayList<String> attrvalues; ->1 mac 14000

删除对象

标准格式：

```
DELETE FROM <class_name> [<where_stmt>];
```

示例：

```
DELETE FROM product WHERE name="mi" ;
```

数据结构：deleteStmt
String type； ->delete
String classname；-> product
Whereclause -> name=“mi”

更新对象

标准格式：

```
UPDATE <class_name> SET {<attr> = <expr>}
[<where_stmt>];
```

示例：

```
UPDATE product SET price=4900 WHERE name="iphone";
```

数据结构：updateStmt
String type；
String classname； -> product
ArrayList<String> attrs -> price
ArrayList<String> values ->4900
Whereclause -> `name="iphone"`

跨类查询

标准格式：

SELECT <class_name>{ -> <class_name>} [.<attr>] [where_stmt];

示例：

```
SELECT  jpproduct->product->usproduct.sales FROM jpproduct
WHERE id=2;
```

数据结构：selectStmt
String type； ->select
String classname；-> jpproduct
ArrayList<attrcontext> attrs
String Whereclause -> id=1

数据结构：attrcontext
String type； ->direct or indirect
String classname；jpproduct
String attr；->sales
ArrayList<String> crossclass ->{jpproduct，product，usproduct}