JavaCC 简介

索扬公关

2009 年 4 月 22 日 警察

面向解析器开发初学者的 JavaCC 简单介绍。

介绍

大多数时候，当需要解析文件或流时，程序员倾向于依赖“ `Tokenizer`”或“ `StreamTokenizer`”而不是创建解析器。当然，创建解析器非常耗时，因为它需要对所有可能的状态进行迭代测试。但是，它使您的应用程序健壮且无错误，尤其是在处理具有特定格式的文件时。一旦你开始创建一个简单的解析器，你肯定会发现它是许多繁忙情况下更好的选择。本文针对初学者进行解析器开发。这将使您了解通过合适的分步示例创建解析器。在本教程的最后，我们将解析一个 SQL 文件并提取表规范（请注意，这是为了说明目的；不支持完整的 SQL 格式）。

先决条件

JavaCC 适用于 Java VM 1.2 或更高版本。您需要安装 JavaCC。有关安装的帮助，请参阅 https://javacc.dev.java.net/doc/installhelp.html。如果您使用的是 Eclipse，则可以使用免费的 JavaCC 插件。下载并安装插件（Google for *easy-javacc-1.5.7.exe*，JavaCC 的免费 Eclipse 插件）。

什么是JavaCC?

Java Compiler Compiler 是一个用于 Java 程序的开源解析器生成器。与 YACC（Yet Another Compiler Compiler）不同，JavaCC 是一个自顶向下的 LL 类型语法解析器。所以严格来说，LR解析（左递归）在JavaCC中是不可能的。JavaCC 为上下文无关文法创建 LL 解析器（上下文无关文法包含格式为 NT -->T 的产生式规则，其中 NT 是已知终结符，t 是终结符和/或非终结符的组合）。LL 解析器从左到右解析输入，并与 LR 解析器创建句子的最右侧派生相比，创建句子的最左侧派生。这些类型的解析器使用 next 标记来做出解析决策，而无需任何回溯（Look Ahead）。所以这些LL解析器并不复杂，

在开始之前

解析器和词法分析器是描述任何字符流的语法和语义的软件组件。词法分析器识别并分离一组称为标记的预定义字符。解析器指定语句和/或不同语句中不同标记的语义。因此，解析器可以很容易地用于检查结构文件并从文件中提取特定组件。例如，在以下 Java 语句中：

爪哇 复制代码

```
String test = "Testing";
```

词法分析发现以下标记：

爪哇 复制代码

```
{ "String"|" "|"Test"|" "|"="|" "| "\"" |"Testing"| "\""|";"}
```

所以经过词法分析，我们会得到这组token：

爪哇 复制代码

```
{STRING_TYPE|SPACE|VAR_NAME|SPACE|EQUAL|SPACE|D_QUATES|VAR_VAL|D_QUATES|SEMCOLN}
```

解析器检查语法文件中指定的词法分析器标识的标记的语义。对于本身不是词法分析器或解析器的 JavaCC，它根据上下文无关文法文件中的规范为词法分析器和解析器生成 Java 代码。按照惯例，这些语法文件以扩展名.jj命名。与手写的 Java 解析器相比，这些语法文件产生更多的模块化，并且易于阅读、修改或编写，因此可以节省大量时间和精力。

文件中令牌的关联由 BNF 产品描述。Backus-Naur Form (BNF) 是一种元语法符号，用于表达上下文无关文法。大多数解析器生成器支持 BNF 生成规则，用于在无错误输出中指定令牌类型的序列。在接下来的部分中，您将看到如何使用.jj文件中的BNF 产生式指定令牌关联。

因此，让我们以一个非常简单的示例开始创建 JavaCC 文件。假设我们有一个包含 SQLCreate语句的文件。对于此示例，我们不考虑 Create Table 语句中的字段。在接下来的步骤中，我们将向文件中添加更多项目。随着文件越来越复杂，分步创建和修改语法通常被认为是一种很好的做法。

SQL 复制代码

```
Create Table test()
```

在上面的例子中，你有令牌：

复制代码

```
CTCMD :- "Create Table" //the create table command
             TNAME :- "test" //the table name followed
             OBRA :- "(" //Opening bracket
             CBRA: - ")" //Closing bracket
             EOF //End of the file
```

语法文件的结构

要生成解析器，JavaCC 的唯一输入是上下文无关语法文件。JavaCC 在输出中生成 7 个 Java 文件。JavaCC 语法文件（.jj文件）由四个部分组成。

1. 选项
2. 类声明
3. 词法分析规范
4. BNF 符号

的Options部分是指定的可选参数，例如DEBUG，STATIC等。在本例中，STATIC被设定为false使解析器类非静态，从而使解析器的多个实例可以在一个时间存在。STSTIC是true默认的。语法文件中的类声明提供了生成的解析器的主要入口点。语法文件中的另一部分用于指定词法分析的标记。语法可能包含 BNF 产生式规则，这些规则指定定义文件结构的标记的关联。为了解析上述文件，我们的语法文件如下所示：

爪哇 复制代码

```
/* Sample grammar file*/
options
{STATIC = false ;}
PARSER_BEGIN(SqlParser)
package sqlParserDemo;
class SqlParser {
{Start () ;}
}
PARSER_END (SqlParser)
```

```
SKIP: { "\n" | "\r" | "\r\n" |"\\"|"\t"|" "}

TOKEN [IGNORE_CASE]:
{
     <CTCMD :("Create Table")>
  | <TNAME :(["a"-"z"])+ >
  | <OBRA :("(")>| <CBRA:(")")>
}
void Start (): {}
{<CTCMD><TNAME><OBRA><CBRA><EOF>}
```

在上面的语法文件中，类声明部分包含在PARSER_BEGIN和PARSER_END关键字中。在本节中，我们定义包、所有导入和解析器类。在这里，“ SqlParser”类具有initParser作为入口点的方法。解析器抛出“ ParseException”和“ TokenMgrError”异常。TokenMgrError当扫描遇到未定义的令牌时抛出。如果遇到未定义的状态或产生式，解析器将抛出“ ParseException”。默认情况下，创建的解析器代码应该有一个接受“ reader”类型的构造函数。

在很多情况下，我们需要忽略文件中的一些字符，比如换行符、空白符、制表符等格式化字符。这些序列可能与含义没有关系。如果将这些字符指定为SKIP终端，则可以在扫描时跳过这些字符。在上述语法文件中，换行符、回车符（注意换行符表示因操作系统而异）和空格被指定为SKIPJA终结符。扫描器读取这些字符并忽略它们。它们不会传递给解析器。

关键字TOKEN用于指定令牌。每个标记都是一个与名称相关联的字符序列。“|” 字符用于分隔标记。JavaCC 提供了一个令牌限定符 IGNORE_CASE. 它用于使扫描仪对令牌不区分大小写。在此示例中，SQL 不区分大小写，因此在任何情况下都可以编写“创建表”命令。如果您有区分大小写和不区分大小写的标记，那么您可以在不同的TOKEN语句中指定它们。标记括在尖括号内。

在这里，我们创建了标记CTCMD、TNAME、OBRA和CBRA。标记的字符序列是用正则表达式语法定义的。(["a"-"z"])+ 表示由 "a"-"z" 组成的任意数量字符的序列。

BNF 生产规则在令牌声明之后指定。这些看起来几乎与方法语法相似。我们可以在第一组花括号中添加任何 Java 代码。通常，它们包含在产生式规则中使用的变量声明。返回类型是来自 BNF 的预期返回类型。在这个例子中，我们只检查文件结构。所以返回类型是void. 的Start在示例函数初始化解析。您应该Start从类声明中调用该方法。在这个例子中，它将文件的结构定义为：

复制代码

```
{<CTCMD><TNAME><OBRA><CBRA><EOF>}
```

文件格式的任何更改都会引发错误。

生成解析器

1. 创建语法文件后，将其保存在目录中。一般命名约定遵循“ .jj “扩展名，例如*demogrammar.jj*。
2. 转到命令提示符并将目录更改为*demogrammar.jj*文件目录。
3. 输入命令“javacc demogrammar.jj”。

如果您使用安装了 JavaCC 插件的 Eclipse，请按照以下步骤操作：

1. 使用语法文件中指定的包创建一个新项目。
2. 要创建语法文件，请右键单击包并转到新建-->其他-->JavaCC-->JavaCC 模板文件。创建语法文件并保存。
3. 构建项目。

在语法文件 (*demogrammar.jj*)上调用 Javacc 后，您将在其自己的文件中获得以下 7 个类。

1. Token: 表示令牌的类。每个令牌都与代表令牌类型的“令牌种类”相关联。字符串 'image' 表示与令牌关联的字符序列。
2. TokenMgrError: 的子类Error。对词法分析中的错误抛出异常。
3. ParseException: 的子类Exception。对解析器检测到的错误抛出异常。
4. SimpleCharStream: 词法分析器接口字符流的实现。
5. SqlParserConstants：用于定义词法分析器和解析器中使用的标记类的接口。
6. SqlParserTokenManager: 词法分析器类。
7. SqlParser: 解析器类。

编译生成的类 (*javac *.java*)。

运行解析器

编译成功后，您就可以测试示例文件了。为了测试示例，`main`向包中添加一个具有函数的类。

爪哇 复制代码

```
 /*for testing the parser class*/
public class ParseDemoTest {
  public static void main(String[] args) {
    try{SqlParserparser = new SqlParser(new FileReader(FilePath));
      parser.initParser () ;}
    catch (Exception ex)
    {ex.printStackTrace() ;}}
```

创建解析器对象时，您可以指定读取器作为构造函数参数。请注意，生成的解析器代码包含一个接受读取器的构造函数。该`InitParser`方法初始化解析。现在您可以构建并运行“ParseDemoTest“。如果给定文件路径中指定的文件不限于语法，则会引发异常。由于我们已经讨论了 JavaCC 操作的总体思路，现在我们可以向文件中添加更多项以进行解析。在此示例中，我们将提取 SQL 文件中提供的表规范（请注意，该示例仅用于说明目的，因此语法并不符合所有 SQL 语法）。新的 SQL 文件采用以下格式：

SQL 复制代码

```
##JavaccParserExample#####
CREATE TABLE STUDENT
(
    StudentName varchar (20),
    Class varchar(10),
    Rnum integer,
)

CREATE TABLE BOOKS
(
    BookName varchar(10),
    Edition integer,
    Stock integer,
)
CREATE TABLE CODES
(
    StudentKey varchar(20),
    StudentCode varchar(20),
)
```

从文件中可以看出它可以有多个`Create`语句，并且每个`Create`语句可能包含多个列。此外，还有用字符“#”括起来的注释。为了获取表格规范，我们需要一个包含表格列表及其详细信息的结构。在包中创建一个类来表示一个表：

爪哇 复制代码

```
public class TableStruct {
  String TableName;
  HashMap<String,String> Variables =
              new HashMap<String, String> ();
}
```

此类表示具有表名的表，以及映射到其数据类型的列名。下面是新文件的修改后的语法文件。

在检查新的语法文件时，您可以看到一个`SPECIAL_TOKEN`. 如前所述，特殊标记是那些没有任何意义但仍然提供信息的标记，例如评论。在这里，评论是由特殊标记定义的。词法分析器识别特殊标记并将其传递给解析器。特殊令牌没有 BNF 符号。您可以看到与 BNF 表示法关联的所有变量声明和其他 Java 代码都包含在“{}“中。一个表达式可能包含其他表达式，例如`TType = DType()`. 识别可重用表达式并单独指定它们是一种很好的做法。在变量的 BNF 符号中：

爪哇 复制代码

```
( TName = <TNAME>
   TType = DType()
   <COMMA>
```

```
   {var.put(TName.image,TType.image);}
)*
```

“*”表示可以出现任意次数的令牌序列。为了运行和测试这个语法文件，改变你的主类如下：

爪哇 复制代码

```
public class ParseDemoTest {
public static void main(String[] args) {
try{
SqlParser parser = new SqlParser (new FileReader("D:\\sqltest.txt"));
        ArrayList<TableStruct> tableList = parser.initParser();
for(TableStruct t1 : tableList)
{  System.out.println("--------------------------");
           System.out.println("Table Name :"+t1.TableName);
           System.out.println("Field names :"+t1.Variables.keySet());
           System.out.println("Data Types :"+t1.Variables.values());
           System.out.println("--------------------------");
      }
}catch (Exception ex)
{ex.printStackTrace() ;}
}
}
```

编译语法文件并运行应用程序。对于 SQL 测试文件，您将获得如下输出：

复制代码

```
--------------------------
Table Name :STUDENT
Field names :[StudentName, Rnum, Clas
--------------------------
--------------------------
Table Name :BOOKS
Field names :[Stock, Edition, BookName]
Data Types :[integer, integer, varchar]
--------------------------
--------------------------
Table Name :CODES
Field names :[StudentCode, StudentKey]
Data Types :[varchar, varchar]
--------------------------
```

结论

JavaCC 是一种广泛使用的词法和解析器组件生成工具，它遵循正则表达式和 BNF 符号语法，用于 lex 和解析器规范。创建解析器需要一个迭代步骤。永远不要期望一次性获得所需的输出。在示例中创建第一个解析器后，尝试修改它并向输入文件添加其他可能性。在处理复杂的文件时，需要一步一步的方法。另外，尽量使表达式通用且可重用。一旦您对*.jj*文件感到满意，您就可以使用更高级的工具，如**JJTree**和**JTB**和 JavaCC 来自动扩充语法。

爪哇 缩小▲ 复制代码

```
/* demo grammar.jj*/
options
{
    STATIC = false ;
}
PARSER_BEGIN (SqlParser)
   package sqlParserDemo;
   import java.util.ArrayList;
   import java.util.HashMap;
   class SqlParser {
         ArrayList<TableStruct> initParser()throws ParseException, TokenMgrError
   { return(init()) ; }
   }
```

```
PARSER_END (SqlParser)

SKIP: { "\n" | "\r" | "\r\n" |"\\"|"\t"|" "}

TOKEN [IGNORE_CASE]:
{
 <CTCMD :("Create Table")>
|<NUMBER :(["0"-"9"])+ >
|<TNAME:(["a"-"z"])+ >
|<OBRA:("(")+>
|<CBRA:(")")+>
|<COMMA:(",")>
}

SPECIAL_TOKEN : {<COMMENT:("#")+(<TNAME>)+("#")+>}

ArrayList<TableStruct> init():
{
  Token T;
ArrayList<TableStruct> tableList = new ArrayList<TableStruct>();
TableStruct tableStruct;
}
{
  (
       <CTCMD>
       T =<TNAME>
       {    tableStruct = new TableStruct ();
       tableStruct.TableName = T.image ;}
   <OBRA>
  tableStruct.Variables = Variables()
  <CBRA>
     {tableList.add (tableStruct) ;}
  )*
  <EOF>
  {return tableList;}
}

HashMap Variables():
{
   Token TName;
   Token TType;
   HashMap<String,String> var = new HashMap<String, String>();
 }

(
TName = <TNAME>
TType = DType()
<COMMA>
{var.put(TName.image,TType.image);}
)*
{return var;}
}
Token DType():
{
   Token TDType;
}
{
    TDType=<TNAME>
    [<OBRA><NUMBER><CBRA>]
   {return TDType;}
 }Create Table test
        (
        )
```

执照

分享

关于作者

索扬公关
没有提供传记
美国
手表
该会员

评论和讨论

添加评论或问题 电子邮件提醒 Search Comments

第一 页上一页 下一页

initParser 在哪里?
Adnan Siddiqi **17-Mar-15 14:28**

回复：initParser 在哪里?
Theodore S. Norvell 23-Mar-15 2:28

我的5票
Vinod Viswanath **24-Aug-12 1:18**

刷新 1

一般 新闻 建议 问题 错误 答案 笑话 赞美 咆哮 管理员

使用Ctrl+Left/Right 切换消息，Ctrl+Up/Down 切换主题，Ctrl+Shift+Left/Right 切换页面。