JavaCC、解析树和 XQuery 语法，第 2 部分

使用 JJTree 构建和遍历定制解析树

Howard Katz (howardk@fatdog.com), 所有人, Fatdog Software

简介： 本文的第 1 部分简要讨论了语法、解析器和 BNF。然后它介绍了 JavaCC，一个流行的解析器生成器。第 2 部分演示了如何修改第 1 部分中的样本代码，这样就可以使用附加工具 JJTree 来构建相同解析的解析树表示。您将探索这种方法的优点，并研究如何编写 Java 代码在运行时遍历该解析树以便恢复其状态信息，并对正在解析的表达式求值。本文结尾将演示如何开发通用例程，用于遍历从一小部分 XQuery 语法生成的解析树，并对其求值。

发布日期： 2002 年 12 月 01 日
级别： 初级

使用 JavaCC 解析器生成器有一个严重缺点：许多或大多数客户机端 Java 代码需要嵌入到 .jj 语法脚本中，该脚本编码了您的 BNF（巴科斯-诺尔范式，Backus-Naur Form）。这意味着您失去了在开发周期中合适的 Java IDE 可以向您提供的许多优点。

开始使用 JJTree 吧，它是 JavaCC 的伙伴工具。JJTree 被设置成提供一个解析器，该解析器在运行时的主要工作不是执行嵌入的 Java 操作，而是构建正在解析的表达式的独立解析树表示。这样，您就可以独立于生成该解析树的解析代码，捕捉在运行时易于遍历和查询的单个树中的解析会话的状态。使用解析树表示还会使调试变得更容易，并缩短开发时间。JJTree 是作为 JavaCC 分发版（distribution）的一部分发布的（请参阅 参考资料）。

在我们继续之前，我要特别提一下，术语 **解析树**和 **抽象语法树**（或 AST）描述了非常相似的语法结构。严格地讲，对于我在下面提到的解析树，语言理论家更精确地把它称作 AST。

要使用 JJTree，您需要能够：

1. 创建 JJTree 作为输入获取的 .jjt 脚本
2. 编写客户机端代码以遍历在运行时生成的解析树并对其求值

本文演示了如何执行这两种操作。它并没有涵盖所有内容，但肯定能带您入门。

JJTree 基础知识

JJTree 是一个预处理器，为特定 BNF 生成解析器只需要简单的两步：

1. 对所谓的 .jjt 文件运行 JJTree；它会产生一个中间的 .jj 文件
2. 用 JavaCC 编译该文件（ 第 1 部分中讨论了这个过程）

幸运的是，.jjt 文件的结构只是我在第 1 部分中向您显示的 .jj 格式的较小扩展。主要区别是 JJTree 添加了一个新的语法 **node-constructor**构造，该构造可以让您指定在解析期间在哪里以及在什么条件下生成解析树节点。换句话说，该构造管理由解析器构造的解析树的形状和内容。

清单 1 显示了一个简单的 JavaCC .jj 脚本，它类似于您在第 1 部分中看到的脚本。为简便起见，我只显示了结果。

清单 1. simpleLang 的 JavaCC 语法

```
void simpleLang()   : {}    { addExpr() <EOF> }
void addExpr()      : {}    { integerLiteral() ( "+" integerLiteral() )? }
void integerLiteral()   : {}    { <INT> }
SKIP  : { " " | "\t" | "\n" | "\r" }
TOKEN : { < INT : ( ["0" - "9"] )+ > }
```

该语法说明了该语言中的合法表达式包含：

1. 单个整数文字，或
2. 一个整数文字，后面跟一个加号，再跟另一个整数文字。

对应的 JJTree .jjt 脚本（再次声明，略有简化）看上去可能如下：

清单 2. 等价于清单 1 中的 JavaCC 语法的 JJTree

```
SimpleNode simpleLang() : #Root       {}  { addExpr() <EOF> { return jjtThis; }}
void addExpr()          :             {}  { integerLiteral()
                                            ( "+" integerLiteral() #Add(2) )? }
void integerLiteral()   : #IntLiteral {}  { <INT> }
SKIP  : { " " | "\t" | "\n" | "\r" }
TOKEN : { < INT : ( ["0" - "9"] )+ > }
```

该脚本对您已经看到的脚本添加了一些新的语法特性。现在，我们只讨论突出显示的部分。以后，我会详细说明。

逐句说明 JJTree 语法

首先请注意，最顶层的 `simpleLang()` 结果的 JavaCC 的过程性语法现在指定了一个返回类型： `SimpleNode` 。它与嵌入的 Java 操作 `return jjtThis` （有一点为 JJTree 虚张声势）一起指定了从应用程序代码调用解析器的 `simpleLang()` 方法将返回解析树的根，然后这个根将用于树遍历。

在 JavaCC 中，解析器调用看上去如下：

```
SimpleParser parser = new SimpleParser(new StringReader( expression ));
 parser.simpleLang();
```

而现在看上去象下面这样：

```
SimpleParser parser = new SimpleParser(new StringReader( expression ));
 SimpleNode rootNode = parser.simpleLang();
```

注：所抓取的根节点并不仅仅是 `SimpleNode` 类型。它其实是 `Root` 类型，正如 清单 2 中的 `#Root` 伪指令所指定的（虽然您不会在上述调用代码中那样使用）。 `Root` 是 `SimpleNode` 子代，就象 JJTree 生成的解析器构造的每个节点一样。我将在下面向您显示 `SimpleNode` 的一些内置方法。

`addExpr()` 结果中的 `#Add(2)` 构造与上述的 `#Root` 伪指令不同，体现在以下几方面；

- 它是参数化的。树构建器在构造树期间使用节点堆栈；没有参数的节点构建器的缺省行为是将自己放在正在构造的解析树的顶部，将所有节点弹出在同一个 *节点作用域* 中创建的节点堆栈，并把自己提升

到那些节点父代的位置。参数 2 告诉新的父节点（在此示例中是一个 Add 节点）要恰好采用 *两个*子节点，它们是 下一段文字 中描述的两个 IntLiteral 子节点。JJTree 文档更详细地描述了这个过程。使用好的调试器在运行时遍历解析树是另一个宝贵的辅助方法，它有助于理解树构建在 JJTree 中是如何工作的。

- 将 #Root 伪指令放在其结果的主体之外表示 *每次* 遍历该结果时都会生成一个 Root 节点（而在此特定示例中，只允许发生一次），这一点具有同等的重要性 。但是，将 #Add(2) 伪指令放在可选的“零或一”项中表示仅当在解析期间遍历包含它的选择子句时 — 换句话说，当该结果表示一个真的加法操作时 — 才 *有条件地* 生成一个 Add 节点。当发生这种情况时，会遍历 integerLiteral() 两次，每次调用时都将一个 IntLiteral 节点添加到树上。这两个 IntLiteral 节点都成为调用它们的 Add 节点的子节点。但是，如果正在解析的表达式是单个整数，那么作为结果的 IntLiteral 节点将直接成为 Root 的一个子节点。

一图胜千言（引用一句古老的谚语）。以下是由上述语法生成的两种类型的解析树的图形表示：

图 1：单个整数表达式的解析树

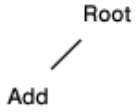
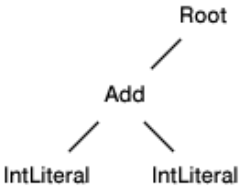


图 2：加法操作的解析树



让我们更详细地研究 SimpleNode 的类层次结构。

使用解析树

在 .jjt 脚本中声明的每个节点都指示解析器生成 JJTree SimpleNode 的一个子类。接下来， SimpleNode 又实现名为 Node 的 Java 接口。这两个类的源文件都是由 JJTree 脚本和定制 .jj 文件一起自动生成的。 清单 1 显示了定制 .jj 文件的当前示例。在当前示例中，JJTree 还提供了您自己的 Root 、 Add 和 IntLiteral 类以及没有在这里看到的一些附加的助手类的源文件。

所有 SimpleNode 子类都继承了有用的行为。 SimpleNode 方法 dump() 就是这样一个例子。它还充当了我以前的论点（使用解析树使调试更容易，从而缩短开发时间）的示例。以下三行客户机端代码的代码片段实例化了解析器、调用解析器、抓取所返回的解析树，并且将一个简单的解析树的文本表示转储到控制台：

```
SimpleParser parser = new SimpleParser(new StringReader( expression ));
 SimpleNode rootNode = parser.simpleLang();
 rootNode.dump();
```

图 2中的树的调试输出是：

```
Root
     Add
         IntLiteral
         IntLiteral
```

辅助导航

另一个有用的内置 SimpleNode 方法是 jjtGetChild(int) 。当您在客户机端向下浏览解析树，并且遇到 Add 节点时，您会要抓取它的 IntLiteral 子节点、抽取它们表示的整数值，并将这些数字加到一起 — 毕竟，那是用来

练习的。假设下一段代码中显示的 addNode 是表示我们感兴趣的 Add 类型节点的变量，那我们就可以访问 addNode 的两个子节点。（ lhs 和 rhs 分别是 *左边（left-hand side）*和 *右边（right-hand side）*的常用缩写。）

```
SimpleNode lhs = addNode.jjtGetChild( 0 );
 SimpleNode rhs = addNode.jjtGetChild( 1 );
```

即使到目前为止您已经执行了所有操作，但您仍然没有足够的信息来计算该解析树表示的算术运算的结果。您的当前脚本已经省略了一个重要的细节：树中的两个 IntLiteral 节点实际上不包含它们声称要表示的整数。那是因为当记号赋予器（tokenizer）在输入流中遇到它们时，您没有将它们的值保存到树中；您需要修改 integerLiteral() 结果来执行该操作。您还需要将一些简单的存取器方法添加到 SimpleNode 。

保存和恢复状态

要将所扫描的记号的值存储到适当的节点中，将以下修改添加到 SimpleNode ：

```
public class SimpleNode extends Node
 {
     String m_text;
     public void   setText( String text ) { m_text = text; }
     public String getText()              { return m_text; }
     ...
 }
```

将 JJTree 脚本中的以下结果：

```
void integerLiteral() : #IntLiteral {} <INT> }
```

更改成：

```
void integerLiteral() : #IntLiteral { Token t; }
                                    { t=<INT> { jjtThis.setText( t.image );} }
```

该结果抓取它刚在 t.image 中遇到的整数的原始文本值，并使用您的 setText() setter 方法将该字符串存储到当前节点中。 清单 5 中的客户机端 eval() 代码显示了如何使用相应的 getText() getter 方法。

可以很容易地修改 SimpleNode.dump() ，以提供任何节点的 m_text 值供其在解析期间存储 — 我把这作为一个众所周知的练习留给您来完成。这将让您更形象地理解在进行调试时解析树看起来是什么样子。例如，如果您解析了“42 + 1”，略经修改的 dump() 例程可以生成以下有用的输出：

```
Root
     Add
         IntLiteral[42]
         IntLiteral[1]
```

组合：XQuery 的 BNF 代码片段

让我们通过研究实际语法的一个代码片段来进行组合和总结。我将向您演示一段非常小的 XQuery 的 BNF 子集，这是 XML 的查询语言的 W3C 规范（请参阅 参考资料）。我在这里所说的大多数内容也适用于 XPath，因为这两者共享了许多相同的语法。我还将简要地研究运算符优先级的问题，并将树遍历代码推广到成熟的递归例程中，该例程可以处理任意复杂的解析树。

清单 3显示了您要使用的 XQuery 语法片段。这段 BNF 摘自 2002 年 11 月 15 日的工作草案：

清单 3：一部分 XQuery 语法

```
[21]  Query              ::= QueryProlog QueryBody
   ...
[23]  QueryBody          ::= ExprSequence?
[24]  ExprSequence       ::= Expr ( "," Expr )*
[25]  Expr               ::= OrExpr
   ...
[35]  RangeExpr          ::= AdditiveExpr ( "to"  AdditiveExpr )*
[36]  AdditiveExpr       ::= MultiplicativeExpr (("+" | "-") MultiplicativeExpr )*
[37]  MultiplicativeExpr ::= UnionExpr (("*" | "div" | "idiv" | "mod") UnaryExpr )*
      ...
```

您将要构建一个刚好适合的 JJTree 语法脚本来处理结果 [36] 和 [37] 中的 + 、 - 、 * 和 div 运算符，而且简单地假设该语法所知道的唯一数据类型是整数。该样本语法 *非常*小，并不能妥善处理 XQuery 支持的丰富的表达式和数据类型。但是，如果您要为更大、更复杂的语法构建解析器，它应该能给您使用 JavaCC 和 JJTree 的入门知识。

清单 4 显示了 .jjt 脚本。请注意该文件顶部的 options{} 块。这些选项（还有许多其它可用选项开关）指定了其中树构建在本例中是以 *多重* 方式运行的，即节点构造器用于显式地命名所生成节点的类型。备用方法（不在这里研究）是结果只将 SimpleNode 节点提供给解析树，而不是它的子类。如果您想要避免扩散节点类，那么该选项很有用。

还请注意原始的 XQuery BNF 经常将多个运算符组合到同一个结果中。在 清单 4中，我已经将这些运算符分离到 JJTree 脚本中的单独结果中，因为这让客户机端的代码更简单。要进行组合，只需存储已扫描的运算符的值，就象对整数所进行的操作一样。

清单 4：清单 3 中的 XQuery 语法的 JJTree 脚本

```
options {
   MULTI=true;
   NODE_DEFAULT_VOID=true;
   NODE_PREFIX="";
}
PARSER_BEGIN( XQueryParser )
package examples.example_2;
public class XQueryParser{}
PARSER_END( XQueryParser )
SimpleNode query()      #Root       : {} { additiveExpr() <EOF> { return jjtThis; }}
void additiveExpr()                 : {} { subtractiveExpr()
                                           ( "+" subtractiveExpr() #Add(2) )* }
void subtractiveExpr()              : {} { multiplicativeExpr()
                                           ( "-" multiplicativeExpr() #Subtract(2) )* }
void multiplicativeExpr()           : {} { divExpr() ( "*" divExpr() #Mult(2) )* }
void divExpr()                      : {} { integerLiteral()
                                           ( "div" integerLiteral() #Div(2) )* }
void integerLiteral() #IntLiteral :      { Token t; }
                                           { t=<INT> { jjtThis.setText(t.image); }}
SKIP  : { " " | "\t" | "\n" | "\r" }
TOKEN : { < INT : ( ["0" - "9"] )+ > }
```

该 .jjt 文件引入了几个新的特性。例如，该语法中的运算结果现在是 *迭代的*：通过使用 * （零次或多次）发生指示符来表示它们的可选的第二项，这与 清单 2 中的 ? （零次或一次）表示法相反。该脚本所提供的解析器可以解析任意长的表达式，如“1 + 2 * 3 div 4 + 5”。

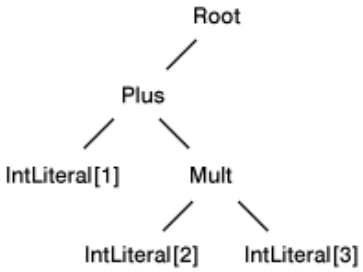
实现优先级

该语法还知道 *运算符优先级*。例如，您期望乘法的优先级比加法高。在实际例子中，这表示诸如“1 + 2 * 3”这样的表达式将作为“1 + (2 * 3)”进行求值，而不是“(1 + 2) * 3”。

优先级是通过使用级联样式实现的，其中每个结果会调用紧随其后的较高优先级的结果。级联样式和节点构造的位置和格式保证了以正确的结构生成解析树，这样树遍历可以正确执行。用一些直观图形也许更易于理解这一点。

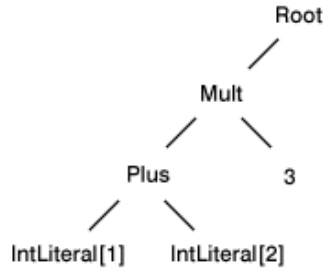
图 3 显示了由此语法生成的解析树，它可以让您正确地将“1 + 2 * 3”当作“1 + (2 * 3)”进行求值。请注意，`Mult` 运算符与它的项之间的联系比 `Plus` 更紧密，而这正是您希望的：

图 3. 结构正确的树



而 图 4显示的树（该语法 *不会*生成这样的树）表示您（错误地）想要将该表达式当作“(1 + 2) * 3”求值。

图 4. 结构不正确的树



遍历解析树客户机端

就象我曾答应的，我将用客户机端代码的清单作为总结，该清单将调用该解析器并遍历它生成的解析树，它使用简单而功能强大的递归 `eval()` 函数对树遍历时遇到的每个节点执行正确操作。 清单 5中的注释提供了关于内部 JJTree 工作的附加详细信息。

清单 5. 可容易泛化的 eval() 例程

```
// return the arithmetic result of evaluating 'query'
 public int parse( String query )
 //------------------------------
 {
     SimpleNode root = null;
  // instantiate the parser
     XQueryParser parser = new XQueryParser( new StringReader( query ));
         try {
         // invoke it via its topmost production
         // and get a parse tree back
         root = parser.query();
         root.dump("");
     }
     catch( ParseException pe ) {
         System.out.println( "parse(): an invalid expression!" );
     }
     catch( TokenMgrError e )  {
         System.out.println( "a Token Manager error!" );
     }
     // the topmost root node is just a placeholder; ignore it.
     return eval( (SimpleNode) root.jjtGetChild(0) );
 }
 int eval( SimpleNode node )
 //--------------------------
 {
```

```
        // each node contains an id field identifying its type.
        // we switch on these. we could use instanceof, but that's less efficient
        // enum values such as JJTINTLITERAL come from the interface file
        // SimpleParserTreeConstants, which SimpleParser implements.
        // This interface file is one of several auxilliary Java sources
        // generated by JJTree. JavaCC contributes several others.
        int id = node.id;
        // eventually the buck stops here and we unwind the stack
        if ( node.id == JJTINTLITERAL )
            return Integer.parseInt( node.getText() );
        SimpleNode lhs =  (SimpleNode) node.jjtGetChild(0);
        SimpleNode rhs =  (SimpleNode) node.jjtGetChild(1);
        switch( id )
        {
            case JJTADD :       return eval( lhs ) + eval( rhs );
            case JJTSUBTRACT :  return eval( lhs ) - eval( rhs );
            case JJTMULT :      return eval( lhs ) * eval( rhs );
            case JJTDIV :       return eval( lhs ) / eval( rhs );
            default :
                throw new java.lang.IllegalArgumentException(
                                "eval(): invalid operator!" );
        }
    }
```

结束语

如果您想要查看可以处理许多实际 XQuery 语法的功能更丰富的 `eval()` 函数版本，欢迎下载我的开放源码 XQuery 实现（XQEngine）的副本（请参阅 参考资料 ）。它的 `TreeWalker.eval()` 例程 *例举*了 30 多种 XQuery 节点类型。还提供了一个 .jjt 脚本。

参考资料

- 您可以参阅本文在 developerWorks 全球站点上的 英文原文.
- 请加入本文的 论坛。（您也可以单击本文顶部或底部的 **讨论**来访问论坛。）
- 有关语法、解析器和 BNF 的简要讨论和 JavaCC 的介绍，请回顾本文的 第 1 部分。您还会找到使用 JavaCC 构建定制解析器的样本代码，它从语法的 BNF 描述开始。
- 请访问免费（但不是开放源码） JavaCC 和 JJTree 的分发版。
- 请在 XML Query 主页上寻找关于 W3C 的 XQuery 和 XPath 规范的更多信息。
- XQEngine是作者编写的 XQuery 引擎的基于 Java 的开放源码实现。
- 想要了解 BNF 的更多知识吗？请访问 Wikipedia.org。
- 请在 developerWorks XML和 Java 技术专区中寻找本文中所涵盖技术的更多信息。
- IBM WebSphere Studio提供了以 Java 和其它语言自动进行 XML 开发的一组工具。它与 WebSphere Application Server紧密集成，而且还可以用于其它 J2EE 服务器。
- 了解您怎样可以成为一名 IBM 认证的 XML 及相关技术开发人员。

关于作者

Howard Katz 居住在加拿大温哥华，他是 Fatdog Software 的唯一业主，该公司专门致力于开发搜索 XML 文档的软件。在过去的大约 35 年里，他一直是活跃的程序员（一直业绩良好），并且长期为计算机贸易出版机构撰写技术文章。Howard 是 Vancouver XML Developer's Association 的共同主持人，还是 Addison

Wesley 即将出版的书籍 *The Experts on XQuery*的编辑，该书由 W3C 的 Query 工作组成员合著，概述了有关 XQuery 的技术前景。他和他的妻子夏天去划船，冬天去边远地区滑雪。可以通过 howardk@fatdog.com与 Howard 联系。

关闭 [x]

developerWorks：登录

IBM ID：
需要一个 IBM ID？
忘记 IBM ID？

密码：
忘记密码？
更改您的密码

☐ 保持登录。

单击提交则表示您同意developerWorks 的条款和条件。 使用条款

提交 取消

当您初次登录到 developerWorks 时，将会为您创建一份概要信息。**您在 developerWorks 概要信息中选择公开的信息将公开显示给其他人，但您可以随时修改这些信息的显示状态。**您的姓名（除非选择隐藏）和昵称将和您在 developerWorks 发布的内容一同显示。

所有提交的信息确保安全。

关闭 [x]

请选择您的昵称：

当您初次登录到 developerWorks 时，将会为您创建一份概要信息，您需要指定一个昵称。您的昵称将和您在 developerWorks 发布的内容显示在一起。

昵称长度在 3 至 31 个字符之间。 您的昵称在 developerWorks 社区中必须是唯一的，并且出于隐私保护的原因，不能是您的电子邮件地址。

昵称： （长度在 3 至 31 个字符之间）

单击**提交**则表示您同意developerWorks 的条款和条件。 使用条款.

提交 取消

所有提交的信息确保安全。

平均分 (3个评分)

1 星

1 星
2 星

2 星

○3 星

3 星
○4 星

4 星
○5 星

5 星

提交

添加评论:

请 登录 或 注册 后发表评论。

注意：评论中不支持 HTML 语法

□有新评论时提醒我剩余 1000 字符

发布

快来添加第一条评论