JAVA脚本解释器SmallJava设计开发

刘峻松

## 概述

本文讲述了以Java语言的语法为设计目标，在JVM的基础之上，使用Java语言开发实现一个Java语法的脚本解释器的设计思路和实现算法，对如何实现任意一种特定语法的脚本解释器进行了理论探讨。

## 前言

Java语言是当今世界非常流行和广泛使用的计算机编程语言，它在1995年被Sun公司发明出来，后来Sun公司被Oracle收购，目前归属于Oracle公司所有。JAVA语言设计为一个跨平台的编程语言，源代码被编译成中间代码，中间代码在JVM（JAVA虚拟机）上运行，JAVA语言在商业上获得了巨大的成功。

在实际使用中，很多时候需要提供动态的执行特性，也就是需要执行动态的JAVA代码，为满足这一需要，我们设计一个独立的JAVA脚本解释器，这个脚本解释器可以读入JAVA源代码作为脚本执行，实现JAVA语言的脚本化。

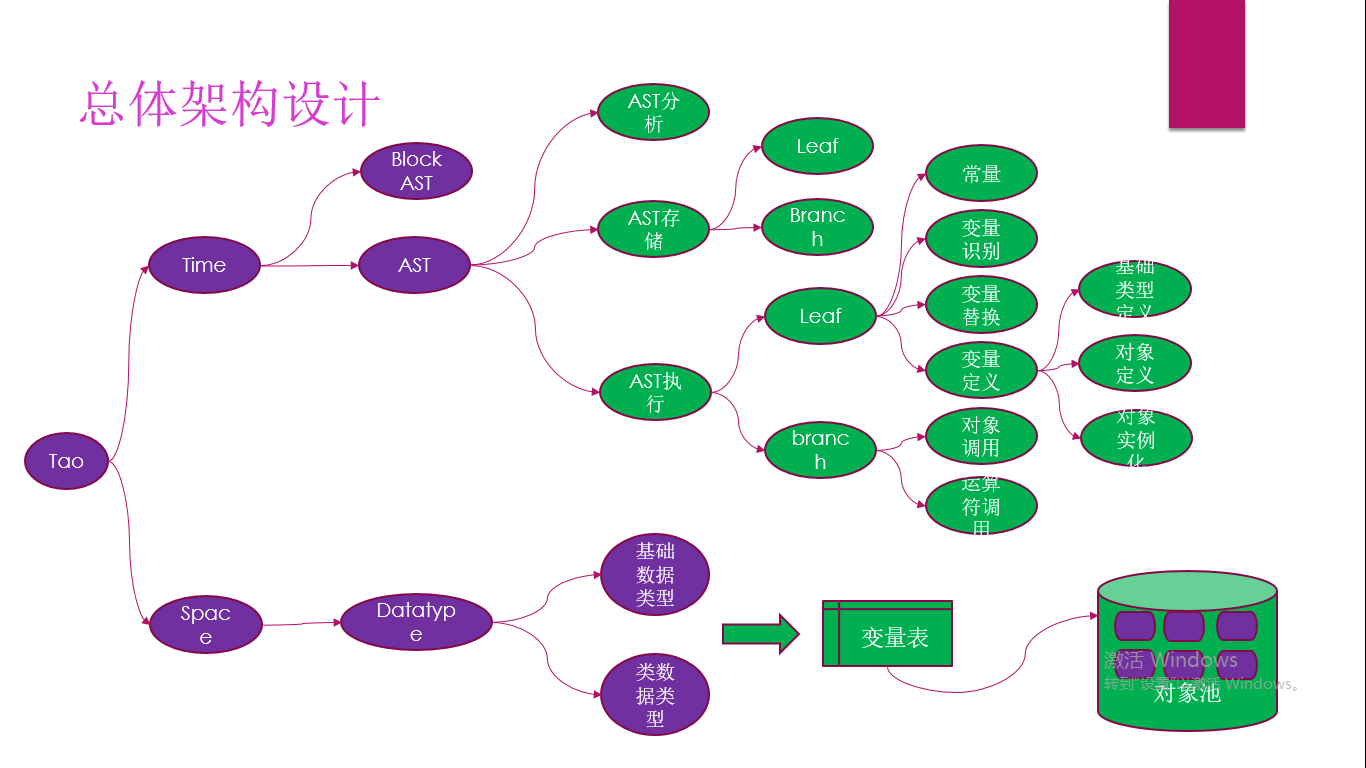
基于编译原理的基本理论，我们针对JAVA语言的一个很小的功能子集，设计了一个通用的脚本解释器，命名为smallJava，本文将讲述如何使用Java语言作为主体，设计开发smallJava脚本解释器的基本原理和核心实现，这一实现过程并不依赖于Java语言本身，理论上可以使用其他语言和环境来替代实现。

这一实现过程也可以用来使用JAVA语言来开发其他语言的脚本解释器。

## 理论基础与术语定义

smallJava脚本解释器的基本设计理论基于“编译原理”这一计算机基础性理论，本文在此不再赘述。smallJava的设计灵感来自《Java编程艺术》一书中使用Java语言编写smallBasic的语言解释器，在此给予致敬。

Java语言解释器的总体设计架构，如下图所示。



这里针对总体的架构设计进行概念性的解释和说明。

* Tao,这里借用《编程之道》里面道的概念，借以表达程序的本源。
* Time，时间，是指程序的动态部分，可以执行的部分，这部分在CPU内执行；
* Space,空间，是指程序的静态部分，一般可以指代为内存，代表存储部分；
* Block AST,这个指代的是一段脚本，JAVA脚本由多个语句组成，block定义为Statement的组合，并根据程序的跳转控制命令，构成整体的程序块，在脚本解释器执行时，先将整个的输入脚本，按照“编译原理”的基本理论，拆解成基础的Statement，Block块的分析，提供了程序流程跳转控制的功能；
* AST，即Abstract Syntax Tree,这里指的是将一个最基本的表达式（statement），按照JAVA语言的语法，拆解分析成AST的层次化表达，由更基础的AST Left节点和AST Branch节点组成；
* AST分析，指将一个最基本的JAVA表达式分解成标准AST数的过程，这一过程的算法采用标准的编译原理作为理论基础，并在其基础上进行适度简化；
* AST存储，这个指的是基于编译原理的理论，定义的AST存储结构，整个AST树是一个层次化的树装结构，最基层的节点是Leaf(叶子)节点，其他的节点称为Branch(分支)节点；
* Leaf节点，这个节点是AST树的基础性节点，这个节点已经不可按照AST的规则进行再次拆分了；
* Branch节点，这些节点是按照AST的规则进行二次拆分的节点，拆分结束后这个节点将拥有自己的子节点，子节点可能是Leaf节点，也可能是Branch节点，可以再次进行拆分。整个AST树就是从一个Branch节点开始分解，层层分解到所有部分都不再可分解，就完成了整颗AST树的分解过程。
* AST执行，在AST树分解完毕以后，可以对AST树本身进行执行，AST树执行以后会记录本身的返回值到AST树上。AST树的执行，根据AST节点的特性走向不同的分支。
* AST Leaf节点执行，AST的叶子节点已经是最基础的节点了，这里的叶子节点分为常量、变量（识别）、变量替换、变量定义四大类型；
* AST-Leaf-常量，常量是指在程序运行过程中，不会随着程序运行的时间顺序而发生含义变化的值，这里的常量包括基础类型常量（根据基础类型来限定）和字符串常量（字符串本身是Java语言的一个标准类，所以不归属于基础类型常量），常量的值是不会发生变化的。
* AST-Leaf-变量识别，一般的表达式中，所有的变量需要进行值替换，根据变量表，将变量替换为实际的数值，才能进行下一步的计算，这里的变量识别是指用于变量赋值的左操作符，这种情况下，这个变量需要被识别出来，待下一步进行赋值，但本身不需要进行值的替换。例如：【i = 100】这个表达式之中，i本身是一个变量，但这个变量由于处在【=】的左边，因此需要进行变量识别，在left节点计算时，不需要去【变量表】中进行值替换，只需要标识出这是一个【变量】就可以了。这个变量的具体值，将在【=】这个操作符进行计算时实现具体的赋值操作。
* AST-Leaf-变量替换，在一般表达式中，所有的变量都需要进行值替换，值替换基于【变量表】进行，根据变量类型，基础类型的变量替换成基础类型的值，而类类型的变量被替换成实际的对象值，这里唯一的例外是【变量识别】。
* AST-Leaf-变量定义，这个表达式的特定格式是 【变量类型】+【空格】+【变量名】，例如int i就代表要定义一个名为i的int类型，这一过程的具体解释执行，是通过在【变量表】中记录相关信息来实现的。变量定义可以基于基础数据类型，也可以基于类来进行变量定义。正是通过基于【类】的变量定义，使得脚本解释器具有了面向对象的支持特性。
* AST-Leaf-对象实例化，这个表达式的特定格式是[new]+ [空格]+[类名称]，例如[new String()]，这个脚本目前仅支持不带构造函数的对象实例化，未来会考虑支持带参数的构造函数调用。
* AST-Branch-对象调用，这个表达式是针对实例化对象的方法调用，格式为[对象名]+[.]+[方法名]+[调用参数]，例如 [s1.length()]，这里通过变量替换的方法将变量名替换为特定的JAVA实例，通过方法名和调用参数，并通过JAVA语言的反射机制来实现AST-Branch-对象调用的具体过程。这个对象调用可以理解为是一个多元的【.】操作符调用；
* AST-Branch-操作符调用，在对象调用之外，其他的操作符可以理解为两个或多个基础类型数值之间的连接符号，一般操作符定义如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 操作符分类 | 操作符 | 操作符说明 | 备注 |
| 算术操作符 | + | 加法 |  |
| 算术操作符 | - | 减法 |  |
| 算术操作符 | \* | 乘法 |  |
| 算术操作符 | / | 除法 |  |
| 布尔操作符 | && | 逻辑与 |  |
| 布尔操作符 | || | 逻辑或 |  |
| 布尔操作符 | ！ | 逻辑非 |  |
| 布尔操作符 | == | 相等 |  |
| 布尔操作符 | > | 大于 |  |
| 布尔操作符 | >= | 大于等于 |  |
| 布尔操作符 | < | 小于 |  |
| 布尔操作符 | <= | 小于等于 |  |
| 字符串操作符 | + | 字符串拼接 |  |
| 赋值操作符 | = | 赋值 | 可用于基础数据类型，字符串类型，对象类型 |

* Datatype，数据类型分为基础数据类型，字符串数据类型，类数据类型三种，字符串也是使用Java中String类型来实现的，但这个类型非常常用，因此单独出来进行说明；

|  |  |  |  |
| --- | --- | --- | --- |
| 数据类型分类 | 数据类 | 数据类型说明 | 备注 |
| 数值型 | Int | 整型 |  |
| 数值型 | Long | 长整型 |  |
| 数值型 | Float | 浮点型 |  |
| 数值型 | double | 双精度型 |  |
| 逻辑型 | Boolean | 布尔型 | True/false |
| 字符串型 | String | 字符串 |  |
| 类类型 | 根据定义决定 | JAVA类库提供 |  |
|  |  |  |  |

* 变量表，变量表是脚本解释器执行时所占用的内存空间的抽象，将所有脚本定义的变量名和变量的具体值存储于其中。在smallJava脚本解释器中，这个变量表是由解释器负责实现的；
* 对象池，对象池是实际存储生成的JAVA对象的地方，按照JAVA语言的规范定义，所有程序产生的JAVA对象都是存放在JVM的Heap（堆）里面的，我们这里直接使用了JVM所提供的堆来实现，并没有自己去重新实现一个对象池，因此这里的对象池是一个逻辑上抽象出来的概念。

## 核心数据结构说明

smallJava本身的程序代码不算很多，在V1.0版本只有不到2000行代码，但即便如此，要在一篇论文中面面俱到来介绍也是非常困难的，本章只对核心的数据结构和算法进行简要说明。

### AST树节点（AstTreeNode.java）

smallJava设计的一个要点，是不仅把AstTreeNode作为一个AST的存储点，同时也把它当成了一个AST计算的中间结果存放点，这样设计的好处是可以避免在程序中引入Stack（堆栈）的概念，这样程序的设计和实现会变得更加简单。Asttreenode的变量定义如下：

|  |
| --- |
| // 要进行分析的字符串信息  String nodeString = "";  // 节点的级别  **int** nodelevel = 0;  // 是否经过了语法分析 true 已经分析，false 未分析  **boolean** syntaxflag = **false**;  // 这个字符串存放语法分析以后产生的错误信息  String errorinfo = "";  // 分析以后产生的子节点，默认为null对象  ArrayList<ASTTreeNodeV1\_1> children = **null**;  // 判断当前这个节点是不是一个要设置的变量，如果是，就不可进行变量替换  **private** **boolean** varflag = **false**;  // stringresult现在只在一种情况下有效：变量赋值时，存储变量名  String stringresult = **null**;  // 如果不是操作符，而是一个操作数，则写入operdata;  OperElementData operdataresult = **null**; |

这里的ASTTreeNode对应到上图中的AST存储点，通过存储本身的children节点，最终构建成一个完整的树状结构。

这个AST节点定义是对应于一个基本的Java表达式（statement）;

### Block树节点（BaseBlockNode.java）

这里的Block节点和Ast节点是对应的，AST节点对应的是Java表达式，而Block树节点是对应于一个Java代码块，这一个块是由多个Java表达式组成的。这个Java类的变量定义如下：

|  |
| --- |
| //代码块分类  **private** String blocktype="";  // 代码块的文本内容  **private** String blockcontent="";  // \* 代码块的级别  **private** **int** blocklevel=0;  // 本地变量表  **private** VarTableNodeV1\_1 varmaps = **new** VarTableNodeV1\_1(**null**);  // 实际计算用的computecontent;  // 原始的blockcontent不可变，计算时使用computecontent  **private** String computecontent;  // \* 分解以后得到的子节点  **private** ArrayList<BaseBlockNodeV1\_1> children = **null**; |

这里设计的要点是将Block的AST分解和Statement的AST分解剥离开，在脚本解释时，先按照Block的级别进行AST树的分解，这样分解的最小单位是一个statement,原则上每个statement都以;来结束，然后在运行到这一步骤时，再将statement进行二次AST分解，分解为AST节点，再根据传入的变量表，对AST节点进行评估计算。

### 变量值定义（VarValue.java）

变量值是构建变量表的基本元素，这里使用一个统一的变量封装对象来封装所有各种可能的变量值。其属性定义如下。

|  |
| --- |
| //变量名  **private** String varname=**null**;  //变量基础类型  **private** String vartype=**null**;  //变量值，用字符串表示字符串类型的值  **private** String varsvalue="";  //变量值，用来存储数值型数值  **private** **double** vardvalue=0;  //变量值，用来存储bool型数值  **private** **boolean** varbvalue=**false**;  //变量值，用来存储  **private** Object varobjvalue=**null**; |

### 变量表（VarTableNode.java）

变量表是整个解释器的空间数据结构(Space)。它的属性定义如下。

|  |
| --- |
| /\*\*  \* 递归查找的父级节点，默认为null，代表是没有上级变量节点的  \*/  **private** VarTableNode parentnode = **null**;  /\*\*  \* 内部的HashMap存储，这个存储目前只存储基础类型，因此使用String来存储具体数据  \*/  **private** HashMap<String, VarValue> myvarmap = **new** HashMap<String, VarValue>(); |

这里的变量表设计要点在于把变量表设计成一个层次性的变量表，每个变量表都指向一个自己的父节点，但这个父节点可以是null值，这样就实现了变量的作用域被限定在语法的定义范围之内，同时在本作用域内可以间接访问到上层的变量定义。

## 其他讨论

## 附录

### 参考文献

《编译原理（第二版）》，机械工业出版社，[美] Alfred V.Aho，[美] Monica S.Lam，[美] Ravi Sethi 等 著，赵建华，郑滔 等 译。

《编程之道》，电子工业出版社，[美]Geoffrey James

《JAVA编程艺术》清华大学出版社,Herbert Schildt 邓劲生译

《Java编程思想（第4版）》 **作者:[美] Bruce Eckel出版社:机械工业出版社**