C 语言与 Java 语言的符号表区别和特点

C 语言

大多数编程语言都可以分成三部分：声明(declaration)，表达式(expression)，语句模块(statement). 每部分都有专门的语法来定义，在上一节中，通过语法定义了 C 语言的变量声明，并通过解析器成功实现了变量声明的语法解析。对于 C 语言中的一段函数代码，便可分割成对应于上面所说的三部分。函数声明中的函数名，返回值和输入参数例如：

```
int fun(int argl, int arg2);
```

就可以对应于上面三部分中的声明部分。函数的主体则对应于表达式和语句模块部分。

在对代码的声明部分进行语法解析时，我们需要构建一种数据结构，以便于支持具体的代码生成，这种数据结构，就是我们接下来要研究的符号表。符号表本质上是一种数据库，用来存储代码中的变量，函数调用等相关信息。该表以 key-value 的方式存储数据。变量和函数的名字就用来对应表中的 key 部分，value 部分包含一系列信息，例如变量的类型，所占据的字节长度，或是函数的返回值。当我们的解析器读取源代码，遇到声明部分时，便给符号表添加一条记录，如果变量或函数脱离了它的作用范围时，便将他们对应的记录从表中删除。例如：

```
{

int variable = 0;

}
```

在上面代码中，进入大括号时，解析器遇到变量的声明，于是便把变量 variable 的相关信息写入符号表。当解析器读取到右括号时，便把 variable 在符号表中的信息给删除，因为出了 variable 的作用范围只在括号之内。

符号表还可以用来存储类型定义(typedef)和常量声明，在词法解析的过程，词法解析器还需要和符号表交互，用于确定一个变量名是否属于一种类型定义，例如：

```
typedef char SingleByte
```

当词法解析器读取 SingleByte 这个字符串后，会在符号表中查询这个字符串所对应的记录，由于每个记录都有一个标志位，用来表明该字符串是否属于变量声明，于是词法解析器从记录中读取这个标志位，发现 SingleByte 对应的标志位被设置为 1，因此词法解析器就不会把 SingleByte 当做普通的变量处理，而是当做关键字来处理。

符号表作为一种数据库，它必须具备以下特点：

1. 速度：由于符号表会被编译器频繁写入和读取，因此记录的写入，查询速度必须足够快。为了保证速度，整个符号表会直接存储在内存中，由此符号表的设计必须仔细考虑内存消耗。
2. 维护性。符号表几乎是编译器中，最复杂的数据结构。它的设计必须灵活可扩展，使得除了编译器外，其他应用程序或模块也能良好的访问符号表。
3. 灵活性。C 语言的变量声明系统很复杂，例如它允许类型关键字的相互组合等(long int, long doube *…), 因此符号表必须能支持各种不同的变量声明方式。
4. 重复性支持。由于对大多数编程语言而言，在不同的间套下，重复的变量名是允许的：

   ```
   int variable = 0;
   {
         int variable = 1;

   }
   ```

   例如上面例子中，两个变量虽然拥有相同的名字，但却是合法的。在大括号内的变量会覆盖(shadow) 外层同名变量。因此符号表必须支持同一个 key, 但却可以映射到不同的 value.

5. 易删除。由于变量可能随时超出作用范围，因此一旦语法解析器发现变量失效后，必须能快速的将其从符号表中删除。

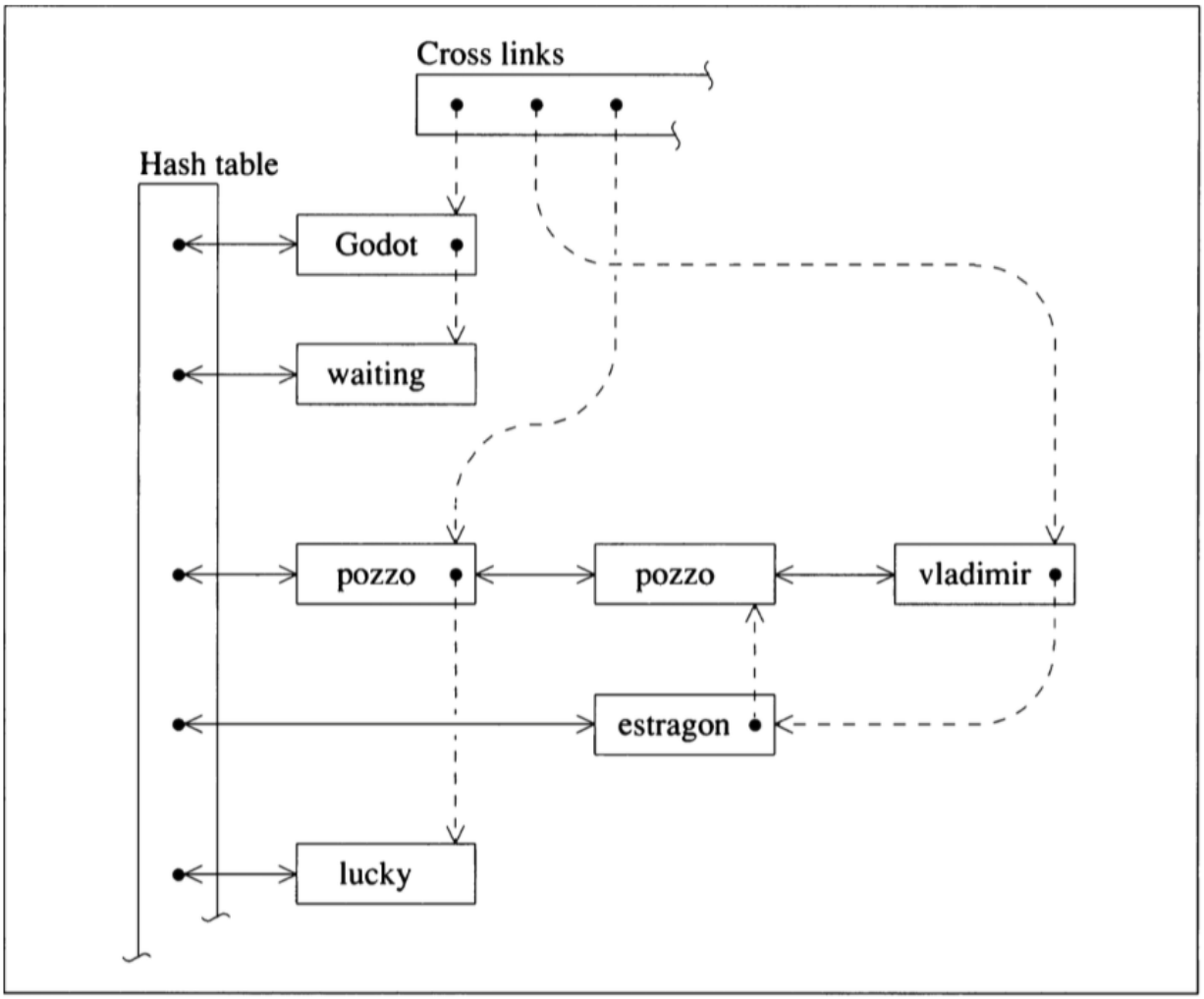
符号表的数据结构设计

由于哈希表的插入和删除平均耗时是 O(1), 因此它能满足快速的插入和删除这一要求，如果遇到作用域不同的同名变量，他们必然被哈希到同一个位置，那么我们可以用链表把哈希到同一个地方的记录串联起来，这样就解决了支持重复性的问题。举个具体例子：

```
int Godot;

void waiting(int vladmir, int estragon)

{

      int pozzo;

      while (condition)

      {
```

```
            int pozzo, lucky;
        }
}
```

在上面的代码中，Godot 和 waiting 是属于第一层的变量，函数 waiting 的参数 vladmir, estragon ,和内部变量 pozzo 属于第二层的变量。while 体内的变量 pozzo, 和 lucky 属于第三层的变量，而且两个 pozzo 是同名变量。于是通过链式哈希表来实现符号表的过程如下：



所有的变量都存储到哈希表中，同名变量 pozzo 被哈希到同一个位置，所有用队列连接起来，由于我们使用变量名做哈希，因此不同变量名也有可能哈希到同一个地方，假定 vladmir 哈希到与 pozzo 相同的地方，所以 vladmir 也在同一个队列中。

在头顶还有一个队列，用来存储不同层次的变量起始指针，例如 Godot, waiting 属于第一层次的变量，因此头部队列的第一个元素存储指针，指向第一个变量 Godot, 然后 Godot 自己又引出一个指针，指向同一层的另一个变量 wait, 由此，同一层的变量实际上是通过一个队列连接起来，这个队列的头指针就存储在 Cross link 列表中。

第二层三个变量 vladmir, estragon, pozzo ，也组成一个队列：vladmir->estragon->pozzo, 这个队列的头指针就存放在 cross link 列表的第二个元素。第三层以此类推。

Java 语言

Java 语言的"编译器"其实是一段"不确定"的操作过程,因为它可能是指一个前端编译器(其实叫"编译器的前端"更准确一些)把*.java 文件转变成*.class 文件的过程;也可能是指虚拟机的后端运行期编译器(JIT 编译器,Just In Time Compiler)把字节码转变成机器码的过程 ;还可能是指使用静态提前编译器(AOT 编译器,Ahead Of Time Compiler) 直接把*.java 文件编译成本地机器代码的过程。下面列举了这 3 类编译过程中一些比较有代表性的编译器。

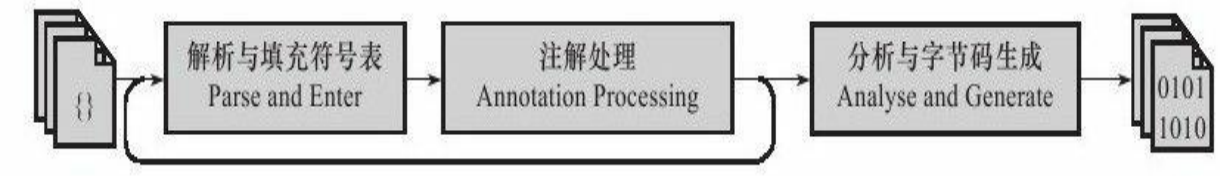
- **前端编译器**:Sun 的 Javac、 Eclipse JDT 中的增量式编译器(ECJ) 。
- **JIT 编译器**:HotSpotVM 的 C1、C2 编译器。
- **AOT 编译器**: GNU Compiler for the Java (GCJ) 、 Excelsior JET。

这 3 类过程中最符合大家对 Java 程序编译认知的应该是第一类,在本章的后续文字里, 笔者提到的"编译期"和"编译器"都仅限于第一类编译过程,把第二类编译过程留到下一章中讨论。限制了编译范围后,我们对于"优化"二字的定义就需要宽松一些,因为 Javac 这类编译器对代码的运行效率几乎没有任何优化措施(在 JDK 1.3 之 后 ,Javac 的-O 优化参数就不再有意 义)。虚拟机设计团队把对性能的优化集中到了后端的即时编译器中,这样可以让那些不是由 Javac 产生的 Class 文件 (如 JRuby、Groovy 等语言的 Class 文件)也同样能享受到编译器优化所带来的好处。但是 Javac 做了许多针对 Java 语言编码过程的优化措施来改善程序员的编码风格和提高编码效率。相当多新生的 Java 语法特性,都是靠编译器的"语法糖"来实现,而不是依赖虚拟机的底层改进来支持,可以说,Java 中即时编译器在运行期的优化过程对于程序运行来说更重要,而前端编译器在编译期的优化过程对于程序编码来说关系更加密切。

从 Sun Javac 的代码来看,编译过程大致可以分为 3 个过程,分别是:

- **解析与填充**符号表的过程。
- 插入式注解处理器的**注解处理**过程。
- **分析与字节码生成**过程。

这 3 个步骤之间的关系与交互顺序如图所示。



编译过程 1：解析与填充符号表

1.词法、语法分析

词法分析是将源代码的字符流转变为标记(Token)集合,单个字符是程序编写过程的最小元素,而标记则是编译过程的最小元素,关键字、变量名、字面量、运算符都可以成为标记,如"int a=b+2"这句代码包含了 6 个标记,分别是 int、a、=、b、+、2 ,虽然关键字 int 由 3 个字符构成,但是它只是一个 Token,不可再拆分。在 Javac 的源码中,词法分析过程由 com.sun.tools.javac.parser.Scanner 类来实现。

语法分析是根据 Token 序列构造抽象语法树的过程,抽象语法树(Abstract Syntax Tree,AST) 是一种用来描述程序代码语法结构的树形表示方式,语法树的每一个节点都代表着程序代码中的一个语法结构(Construct) ,例如包、类型、修饰符、运算符、接口、返回值甚至代码注释等都可以是一个语法结构。

2.填充符号表

完成了语法分析和此法分析后，下一步就是填充符号表的过程，符号表（Symbol Table）是由一组符号地址和符号信息构成的表格，读者可以把它想象成哈希表中 K-V 值对的形式（实际上符号表不一定是哈希表实现，可以是有序符号表、树状符号表、栈结构符号表等）。符号表中所登记的信息在编译的不同阶段都要用到。在语义分析中，符号表所登记的内容将用于语义检查（如检查一个名字的使用和原先的说明是否一致）和产生中间代码。在目标生成阶段，当对符号名进行地址分配时，符号表是地址分配的依据。

在 Javac 源代码中,填充符号表的过程由 com.sun.tools.javac.comp.Enter 类实现,此过程的出口是一个待处理列表(To Do List) ,包含了每一个编译单元的抽象语法树的顶级节点, 以及 package-info.java (如果存在的话）的顶级节点。

编译过程 2：注解处理器

在 JDK 1.5 之后,Java 语言提供了对注解(Annotation) 的支持,这些注解与普通的 Java 代码一样,是在运行期间发挥作用的。在 JDK 1.6 中实现了 JSR-269 规范 ,提供了一组插入式注解处理器的标准 API 在编译期间对注解进行处理,我们可以把它看做是一组编译器的插件 ,在这些插件里面,可以读取、修改、添加抽象语法树中的任意元素。如果这些插件在处理注解期间对语法树进行了修改,编译器将回到解析及填充符号表的过程重新处理,直到所有插入式注解处理器都没有再对语法树进行修改为止,每一次循环称为一个 Round,也就是图 10-4 中的回环过程。

有了编译器注解处理的标准 API 后 ,我们的代码才有可能干涉编译器的行为,由于语法树中的任意元素,甚至包括代码注释都可以在插件之中访问到,所以通过插入式注解处理器实现的插件在功能上有很大的发挥空间。只要有足够的创意,程序员可以使用插入式注解处理器来实现许多原本只能在编码中完成的事情,本章最后会给出一个使用插入式注解处理器的简单实战。

在 Javac 源码中,插入式注解处理器的初始化过程是在 initPorcessAnnotations() 方法中完成的,而它的执行过程则是在 processAnnotations() 方法中完成的,这个方法判断是否还有新的注解处理器需要执行，如果有的话，通过

com.sun.tools.javac.processing.JavacProcessingEnvironment 类的 doProcessing() 方法生成一个新的 JavaCompiler 对象对编译的后续步骤进行处理。

编译过程 3：语义分析与字节码生成

语法分析之后,编译器获得了程序代码的抽象语法树表示,语法树能表示一个结构正确的源程序的抽象,但无法保证源程序是符合逻辑的。而语义分析的主要任务是对结构上正确的源程序进行上下文有关性质的审查,如进行类型审查。举个例子,假设有如下的 3 个变量定义语句:

```
int a=1;
boolean b=false;
char c=2;
```

后续可能出现的赋值运算:

```
int d=a+c;
int d=b+c;
char d=a+c;
```

后续代码中如果出现了如上 3 种赋值运算的话,那它们都能构成结构正确的语法树,但是只有第 1 种的写法在语义上是没有问题的,能够通过编译,其余两种在 Java 语言中是不合逻辑的 ,无法编译 (是否合乎语义逻辑必须限定在具体的语言与具体的上下文环境之中才有意义。如在 c 语言中 ,a 、b 、c 的上下文定义不变,第 2 、3 种写法都是可以正确编译)。

1.标注检查

Javac 的编译过程中,语义分析过程分为标注检查以及数据及控制流分析两个步骤。

标注检查步骤检查的内容包括诸如变量使用前是否已被声明、变量与赋值之间的数据类型是否能够匹配等。在标注检查步骤中,还有一个重要的动作称为常量折叠,如果我们在代码中写了如下定义:

```
int a=1+2;
```

那么在语法树上仍然能看到字面量" 1"、"2"以及操作符"+",但是在经过常量折叠之后 ,它们将会被折叠为字面量"3" 。 由于编译期间进行了常量折叠 ,所以在代码里面定义"a=1+2"比起直接定义"a=3" , 并不会增加程序运行期哪怕仅仅一个 CPU 指令的运算量。

2.数据及控制流分析

数据及控制流分析是对程序上下文逻辑更进一步的验证，它可以检查出诸如程序局部变量在使用前是否有赋值、方法的每条路径是否都有返回值、是否所有的受查异常都被正确处理了等问题。编译时期的数据及控制流分析与类加载时的数据及控制流分析的目的基本上是一致的，但校验范围有所区别，有一些校验项只有在编译期或运行期才能进行。

3.解语法糖

语法糖(Syntactic Sugar) , 也称糖衣语法,是由英国计算机科学家彼得•约翰•兰达(Peter J.Landin)发明的一个术语,指在计算机语言中添加的某种语法,这种语法对语言

的功能并没有影响,但是更方便程序员使用。通常来说,使用语法糖能够增加程序的可读性,从而减少程序代码出错的机会。
Java 在现代编程语言之中属于"低糖语言" (相对于 C#及许多其他 JVM 语言来说),尤其是 JDK 1.5 之前的版本,"低糖"语法也是 Java 语言被怀疑已经"落后"的一个表面理由。Java 中最常用的语法糖主要是前面提到过的泛型(泛型并不一定都是语法糖实现,如 C#的泛型就是直接由 CLR 支持的)、变长参数、自动装箱/拆箱等,虚拟机运行时不支持这些语法 ,它们在编译阶段还原回简单的基础语法结构,这个过程称为解语法糖。
在 Javac 的源码中,解语法糖的过程由 desugar() 方法触发,在 com.sun.tools.javac.comp.TransTypes 类和 com.sun.tools.javac.comp.Lower 类中完成。

4.字节码生成
字节码生成是 Javac 编译过程的最后一个阶段,在 Javac 源码里面由 com.sun.tools.javac.jvm.Gen 类来完成。字节码生成阶段不仅仅是把前面各个步骤所生成的信息 (语法树、符号表)转化成字节码写到磁盘中,编译器还进行了少量的代码添加和转换工作。

例如,前面章节中多次提到的**实例构造器<init>()** 方法和**类构造器<clinit> ()**方法就是在这个阶段添加到语法树之中的(**注意 ,这里的实例构造器并不是指默认构造函数, 如果用户代码中没有提供任何构造函数,那编译器将会添加一个没有参数的、访问性(public、 protected 或 private) 与当前类一致的默认构造函数,这个工作在填充符号表阶段就已经完成**),这两个构造器的产生过程实际上是一个代码收敛的过程,编译器会把语句块(对于实例构造器而言是"{}"块 ,对于类构造器而言是"static{}"块)、变量初始化(实例变量和类变量)、调用父类的实例构造器 (仅仅是实例构造器,<clinit>()方法中无须调用父类的<clinit>() 方法,虚拟机会自动保证父类构造器的执行,但在<clinit>()方法中经常会生成调用 java.lang.Object 的<init>() 方法的代码) 等操作收敛到<init>() 和<clinit>() 方法之中,**并且保证一定是按先执行父类的实例构造器,然后初始化变量,最后执行语句块的顺序进行**,上面所述的动作由 Gen.normalizeDefs() 方法来实现。除了生成构造器以外,还有其他的一些代码替换工作用于优化程序的实现逻辑,如把字符串的加操作替换为 StringBuffer 或 StringBuilder (取决于目标代码的版本是否大于或等于 JDK 1.5)的 append() 操作等。

完成了对语法树的遍历和调整之后,就会把填充了所有所需信息的符号表交给 com.sun.tools.javac.jvm.ClassWriter 类 ,由这个类的 writeClass()方法输出字节码,生成最终的 Class 文件 ,到此为止整个编译过程宣告结束。

https://www.cnblogs.com/winner-0715/p/7400544.html

https://blog.csdn.net/tyler_download/article/details/52437687