## 早期(编译器)优化

### 概述

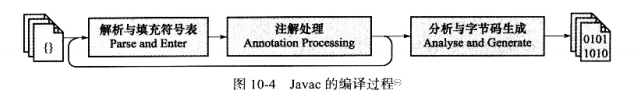
Java语言的编译期可以指3方面  
1. 前端编译器：把\*.java 文件转变为\*.class文件的过程，Sun的javac，Eclipse JDT中的增量式编译器(ECJ)

1. JIT编译器: 后端运行编译器，把字节码转换成机器码。Hotspot VM的C1,C2编译器。
2. AOT编译器：静态提前编译器，直接把\*.java文件编译成本地机器代码。GNU Compiler for the Java (GCJ),Excelsior JET。

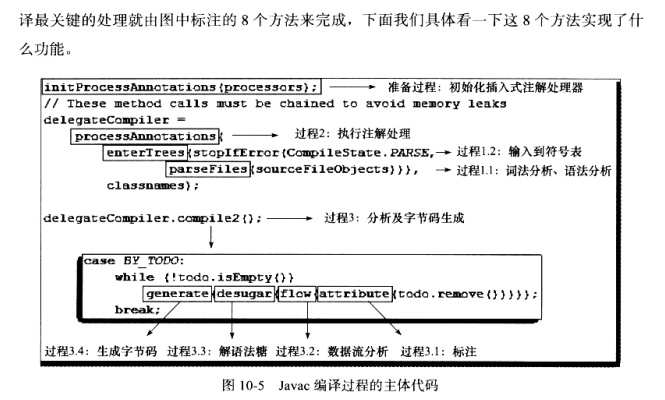
### Javac编译器

Javac实现中，编译过程大致分为3个过程：

1. 解析与填充符号表过程
2. 插入式注解处理器的注解处理过程
3. 分析与字节码生成过程。



Javac编译动作的入口是com.sun.tools.javac.main.JavaCompiler 上述逻辑集中在compile()方法中。



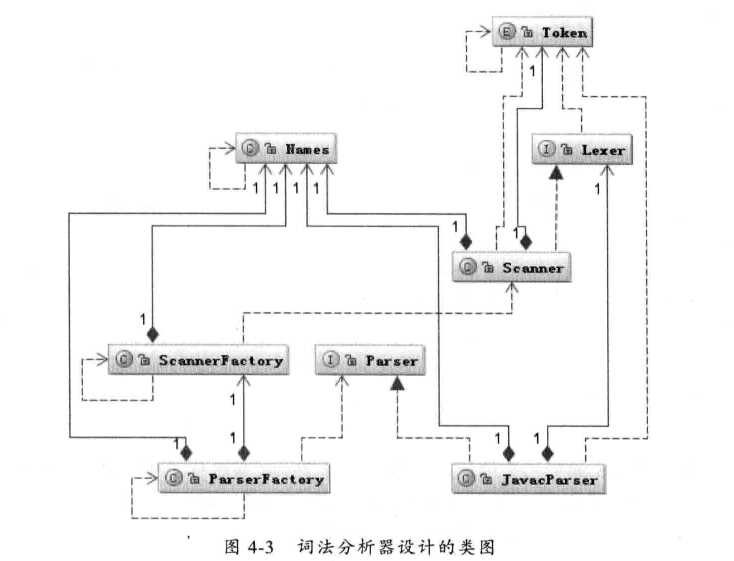
Jdk8中医没有compile2()方法。处理直接放在compile()中。

### 解析与填充符号表

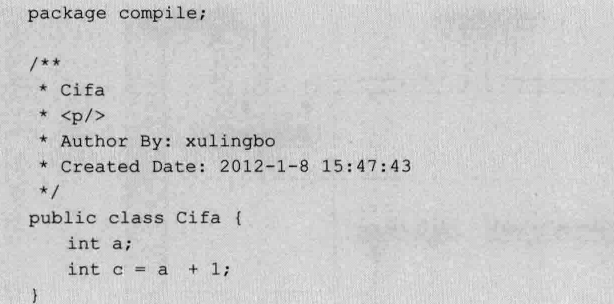
由parseFiles()方法完成。包括词法分析和语法分析两部分。

#### 词法分析

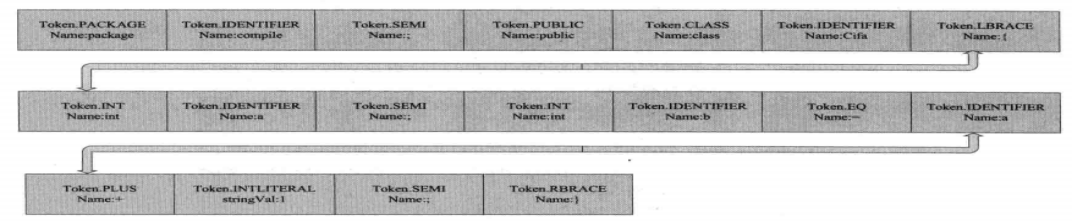
JavaCompile.parseFiles()方法调用了JavaParse.parseCompilationUnit()(此方法实际完成了词法分析和语法分析两个过程)。两个类JavaParse和Lexer(Scanner)保证了主要实现了词法分析过程.



JavaParse中的引用的Token保证了哪些词是符合Java语言规范规定的，而具体读取和归类不同此法的操作由Scanner完成。Token中有枚举类规定了java中的关键词。Names用来存储解析后的词法。词法分析是将源代码的字符流转变为标记(Token)集合，即Token中规定的枚举类。如下边的类。



经过词法分析变为：

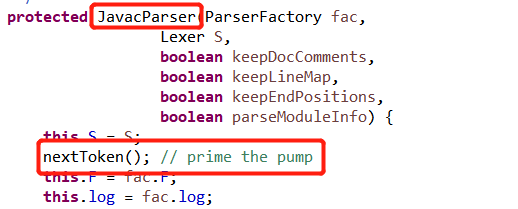


其中除了固定的Token.PACKAGE，Token.SEMI这种，还有一种Token.IDENTIFIER用于表示用户规定的名称，如类名，包名等。(token顺序由nextToken()负责维持)

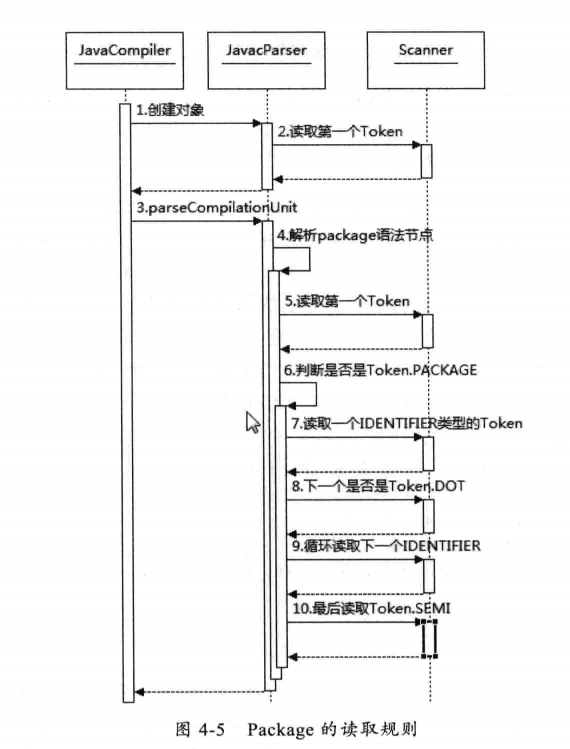
词法分析的两个问题：

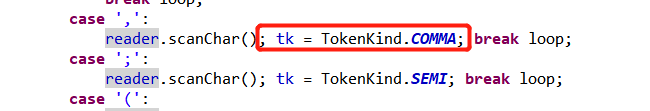
1. javac如何分辨这一个个Token，它怎么知道PACKAGE就是一个Token.PACKAGE,而不是用户定义的TokenIDENTIFIER的名称
2. Javac是如何解析一个完整的Token，如compile这个词就是一个Token，为什么不是com或者comp也就是java是如何知道哪些字符组合在一起就是一个Token的呢。

第一个问题：再创建了JavacParse对象的构造函数是，Scannel会读取第一个Token



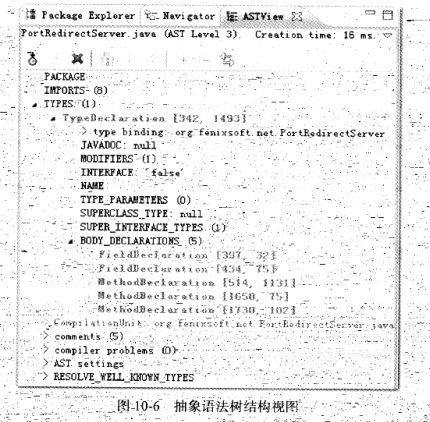
即Token.PACKAGE，Token流的顺序要符合Java的规范，TOKEN.PACKAGE接下来一定是TOKEN.IDENTIFIER....，下图为package的读取规则：



第二个问题：判断哪些字符组合是一个Token的规则是在Scanner的nextToken()中定义的，每调用一次这个方法就会构造一个Token,而这些Token必然是上述Token类中的任何元素之一。

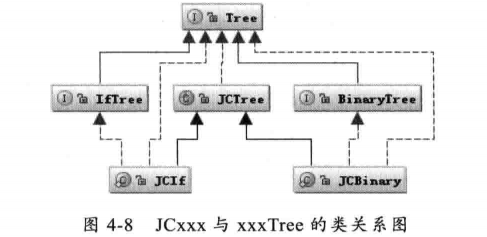
#### 语法分析

语法分析是根据词法分析生成的token流构造抽象语法树的过程。抽象语法树是一种用来描述程序代码语法结构的属性表示方式，语法数的每一个节点都代表着程序代码中的一个语法结构，例如包，类型，修饰符，运算符，接口，返回值甚至代码注释都可以是一个语法结构。由com.sun.tools.javac.JCTree表示。经过此操作，编译器基本不会对源码进行操作了。此过程和词法分析在一个方法中（parseCompilationUnit()）完成



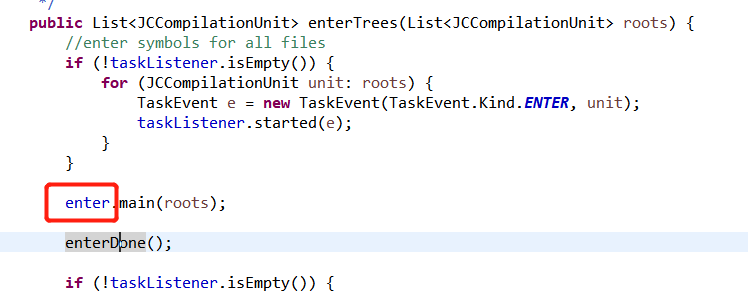
语法树有以下规则：

1. 每个语法节点都会实现一个接口xxxTree,这个接口又继承自com.sun.source.tree.Tree接口,如IFTree表示一个if类型的表达式。
2. 每个语法节点都是JCTree的子类，并且会实现第一节点中的xxxTree接口类，这个类的名称类似于JCxxx，如实现IfTree接口的实现类为JCIf。



#### 填充符号表

JavaCompiler.compile()方法中的enterTrees()方法。符号表是由一组和好地址和符号信息构成的表格，例如哈希表中的K-V值对的形式(实际上符号表不一定是哈希表实现，可以是有序表，树状表)符号表所等级的信息在编译的不同阶段都要用到，如在之后的语义分析(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。在目标代码生成阶段，当对符号名进行地址分配时，符号表是地址分配的依据。填充符号表的过程由com.sun.tools.javac.comp.Enter类实现。



这个类主要完成以下两个步骤。

1. 将在所有类中出现的符号输入到类自身的符号表中，所有类符号，类的参数类型符号(泛型参数类型)，超类符号和继承的接口类型符号等都存储到一个未处理的列表中。
2. 将这个未处理列表中所有的类都解析到各自的类符号列表中，这个操作是在MemberEnter.complete()中完成的。

解释：首先在一个类中除了类本身会定义一些符号变量如类名称，变量名称和方法名称等，还有一些符号是引用其他类的，这些符号会调用其他类的方法或者变量等，还有一些类可能会继承或者实现超类和接口等，这些符号都是在其他类中定义的，那么就需要将这些类的符号也解析到符号表中。第二步自然就是按照递归向下的顺序解析语法书。将所有的符号输入到符号表中。

在Enter类解析这一步骤中，有一个重要的步骤是添加默认的构造函数。（本鸡没有找到相关源码。找死爹了妈耶，怀疑移到了其他步骤中。）

### 注解处理器

JDK1.5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JDK1.6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在变异期间对注解进行处理(插件)。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在对语法树进行修改位置，每一次循环称为一个Round，在源码中，插入式注解处理其的初始化过程是在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doProcess()方法生成一个新的JavaCompiler对象对编译的后续步骤进行处理。

### 语义分析与字节码生成

此时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。

语义分许中分为标注检查以及数据及控制流分析两个步骤。分别由图10-5(compiler()方法)中的attribute()和flow()方法完成。

标注检查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。还有常量折叠。如int a = 1+2。常量折叠后被折叠为3。

此过程检查的点：

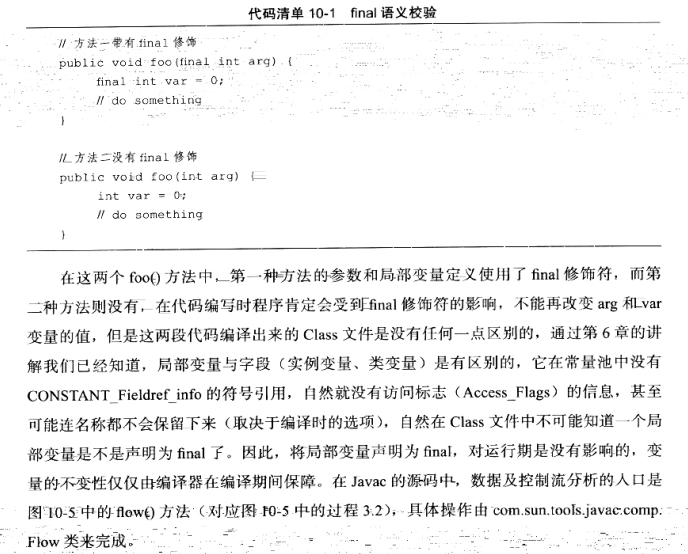
变量的类型是否匹配。

变量在使用前是否已经初始化。

能够推导出泛型方法的参数类型。

字符串常量的合并。

数据及控制流分析 他可以检查出诸如程序局部变量在使用前是否有复制，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。编译器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器或运行期才能进行。如下图：



此过程检查的点：

检查变量在使用前是否被正确复制。除了Java中的原始类型，如int,long,byte都会有默认的初始化值，其他像String类型和对象的引用都必须在使用前先赋值。

保证final修饰的变量不会被重复赋值。如果这个变量是静态变量，则在定义时必须对其赋值。

检查方法的返回值类型是否确定，并检查接受这个方法返回值的引用类型是否匹配。

所有的Checked Exception都要捕获或者向上抛出。

所有的语句都要被执行到。比如return后不能由语句。

解语法糖

语法糖：指在计算机语言中添加的某种语法，对语言功能并没有影响，但更方便程序员使用。Java中的语法糖：泛型，变长参数，自动装箱/拆箱。虚拟机在编译阶段还原回简单的基础语法结构。

字节码生成

图10-5中的generate()方法完成。经过一下两个步骤：

1. 将Java方法中的代码块转化成符合JVM语法的命令行事，JVM的操作都是基于栈的，所有的操作都必须经过出栈和进栈来完成。
2. 按照JVM的文件组织格式将字节码输出到以class为扩展名的文件中。

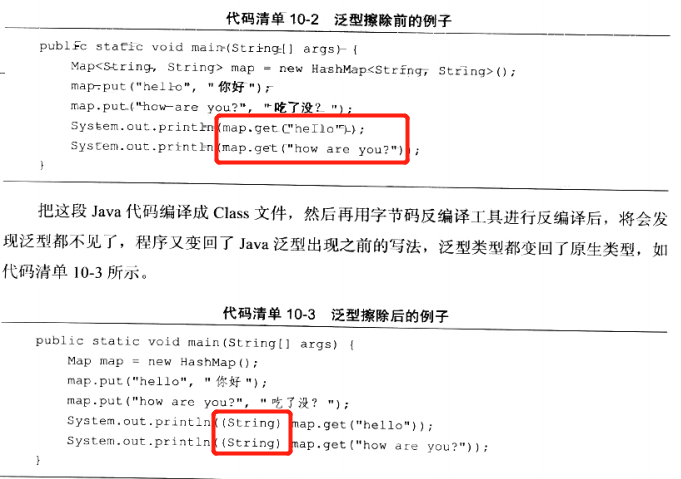
完成了对语法树的遍历和调整之后，就会把填充了所有所需信息的符号表交给com.sun.tools.javac.jvm.ClassWriter类。由这个类的writeClass()方法输出字节码。

### Java语法糖

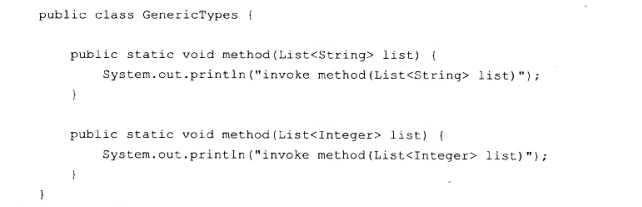
#### 泛型与类型擦出

本质是参数化类型的应用，也就是所操作的数据类型被指定为一个参数。这种参数类型可以用在类，接口和方法的创建中，分别称为泛型类，泛型接口和泛型方法。

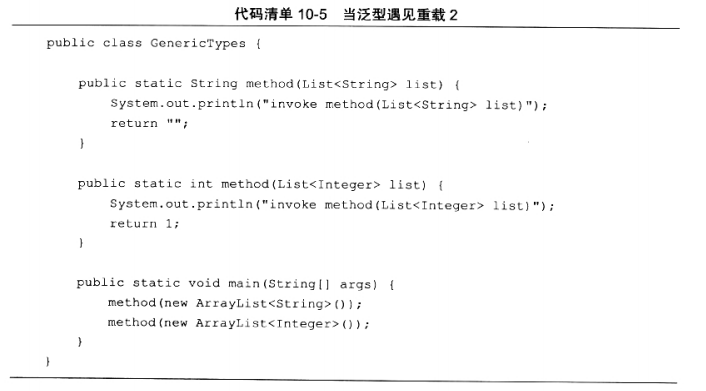
没有泛型时只能通过Object是所有类型的父类和类型强制转换两个特点的配合来实现类型泛化。泛型也有真实泛型和伪泛型。C#就为真实泛型，java为伪泛型。只在源码中存在，编译后的字节码文件中，就已经替换为原来的原生类型了，并且在相应的地方插入了强制转型代码。因此，对于运行期的Java语言来说，ArrayList<int>与ArrayList<String>就是同一个类，所以Java泛型即为一个语法糖，Java中的泛型也称为类型擦除。如下图：



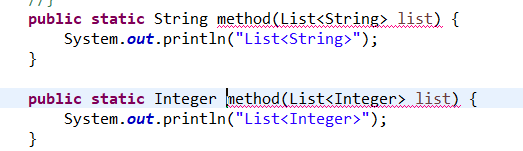
泛型遇到重载：



此两个方法并不能编译通过，泛型擦除成相同的原生类是其中一部分原因。

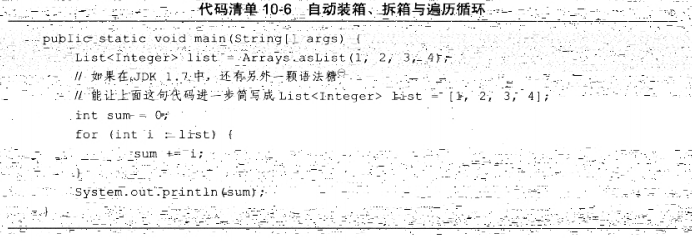


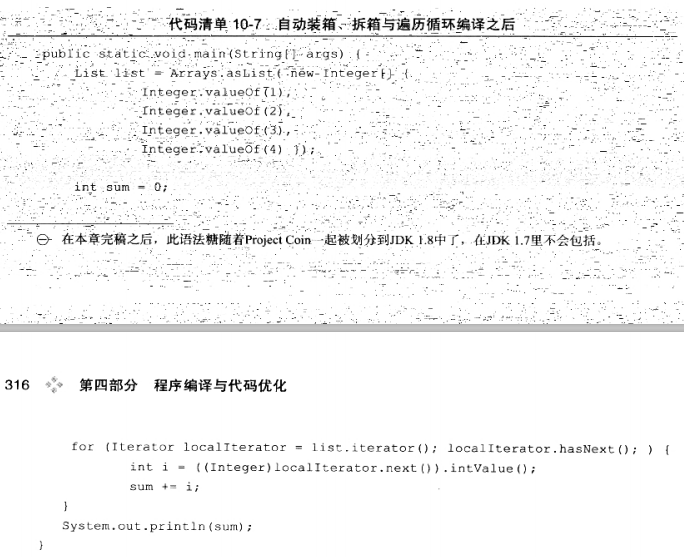
此两个方法可以编译通过，但在1.8中也不可行。



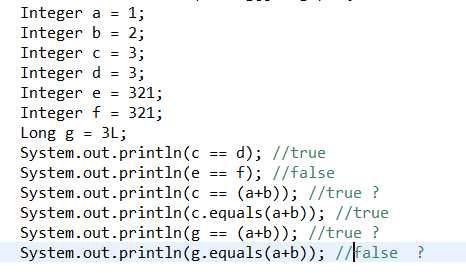
#### 自动装箱，拆箱与遍历循环

查看代码示例：





面试题：



编译后：

