**第10章 早期（编译期）优化**

Java语言的编译期可能有如下几个操作过程：

（1）前端编译器（或编译器前端，如Sun的javac、Eclipse JDT中的增量式编译器ECJ）把\*Java编译成\*class文件

（2）虚拟机的后端运行期编译期（JIT编译器，Just In Time Compiler，如HotSpot VM的C1、C2编译器）把字节码转变成机器码

（3）静态提前编译器（AOT编译器，Ahead Of Time Compiler，如GCJ、Excelsior JET）直接把\*.java文件编译成本地机器代码的过程。

一、Sun Javac编译器

编译过程分成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

\*.Java

解析与填充符号表

Parse and Enter

注解处理

Annotation Processing

分析与字节码生成

Analyse and Generate

01010

Javac的编译过程

Javac编译动作的入口是：com.sun.tools.javac.main.JavaCompiler类，上述三个过程的代码逻辑集中于这个类的compile()和compile2()方法中，主体代码如下，整个编译最关键的处理由如下标注的8个方法来完成。

initProcessAnnotations(processors);//准备过程：初始化插入式注解处理器

delegateCompiler =

processAnnotations( //过程2：执行注解处理

enterTrees(stopIfError(CompileState.PARSE, //过程1.2：输入到符号表

parseFiles(sourceFileObjects))), //过程1.1：词法分析、语法分析

classnames);

delegateCompiler.compile2(); //过程3：分析及字节码生成

过程3.4：生成字节码 过程3.3：解语法糖 过程3.2：数据流分析 过程3.1：标注

case BY\_TODO:

while (!todo.isEmpty())

generate(desugar(flow(attribute(todo.remove()))));

break;

（1）解析与填充符号表

1、词法、语法分析 —— parseFiles()方法

词法分析将源代码字符流转变为标记（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，包含了6个标记，分别为int、a、=、b、+、2。在Javac源码中，词法分析过程由com.sun.tools.javac.parser.Scanner类来实现。

语法分析根据Token序列构造抽象语法树，抽象语法树（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，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。语法分析过程由com.sun.tools.javac.parser.Parser类实现。产出的语法树由com.sun.tools.javac.tree.JCTree类表示。经过这个步骤后，编译器就基本不会对源码文件进行操作了，后续的操作都建立在抽象语法树上。

2、填充符号表 —— enterTrees()方法

完成词法、语法分析后，接下来进行符号表填充。符号表（Symbol Table）是由一组符号地址和符号信息构成的表格，类似于K-V值对形式。符号表中所登记的信息在编译的不同阶段都会用到。在语义分析中，符号表所登记的内容将用于语义检查和产生中间代码。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。

出口：待处理列表（To Do List），包含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。

（2）注解处理器

JDK1.5后，Java语言提供了对注解（Annotation）的支持。这些注解与普通Java代码一样，是在运行期发挥作用的。在JDK1.6中提供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，可以认为这是一组编译器的插件，在这些插件里，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。也就是回环过程。

有了编译器注解处理的标准API后，用户代码才有可能干涉编译器额的行为，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中访问到，所以通过插入式注解处理器实现的插件在功能呢跟上有很大的发挥空间。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。

在Javac源码中，插入式注解处理器的初始化过程是initProcessAnnotations()方法，执行过程是在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，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。

（3）语义分析与字节码生成

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。

1、标注检查 —— attribute()方法

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，另外还会进行常量折叠：如代码int a = 1 + 2，会在编译期被常量折叠为int a = 3。所以在代码里定义int a = 1 + 2比直接定义 int a = 3并不会增加程序运行期哪怕仅仅依据CPU指令的运算量。

标注检查步骤在Javac源码中的实现类是com.sun.tools.javac.comp.Attr类和com.sun.tools.javac.comp.Check类

2、数据及控制流分析 —— flow()方法

检查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。在Javac源码中，数据及控制流分析的具体操作由com.sun.tools.javac.comp.Flow类来完成。

3、解语法糖 —— desugar()方法

语法糖（Syntactic Sugar），也称糖衣语法。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。使用语法糖能够增强程序的可读性，从而减少程序代码出错的机会。

Java属于“低糖语言”，Java中常用的语法糖主要为泛型、变长参数、自动装箱/拆箱等，虚拟机运行时并不支持这些语法，它们在编译阶段会被还原为简单的基础语法结构，这个过程称为解语法糖。

在Javac源码中，解语法糖过程在com.sun.tools.javac.comp.TransTypes类和com.sun.tools.javac.comp.Lower类中完成。

4、字节码生成

Javac编译过程的最后一个阶段，在Javac源码中由com.sun.tools.javac.jvm.Gen类来完成。字节码生成阶段不仅将前面所生成的语法树、符号表转化成字节码写到磁盘，编译器还进行了少量的代码添加和转换操作。如：

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。（注：这里的实例构造器并非指默认构造函数，如果用户代码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（对于实例构造器则为“{}”块，对于类构造器则为“static{}”块）、变量（实例变量和类变量）的初始化、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>()方法中经常会发生调用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，上述的动作由Gen.normalizeDefs()方法实现。

另外除了生成构造器，还会有其他一些代码替换工作用于优化程序实现逻辑，如把字符串的加操作替换为StringBuffer或StringBuilder的append()操作等。

完成了对语法树的遍历和调整后，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。到此早期编译工作宣告结束。

二、Java语法糖的味道（部分）

（1）泛型与类型擦除

JDK1.5 新特性，参数化类型的引用。即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、方法的创建中，分别称为泛型类、泛型接口和泛型方法。

Java中的泛型只在程序源码存在，在编译期进行**类型擦除**，即编译后的字节码文件中，就已经替换为原来的原生类型（Raw Type，裸类型）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。

擦除法所谓的擦除，仅仅是对方法表的Code属性中的字节码进行擦除，实际上元数据还是保留了泛型信息。JCP组织引入了Signature等属性用于解决泛型参数类型的识别问题，Signature的作用就是存储一个方法在字节码层面的特征签名，这个属性保存的参数类型并不是原生类型，而是包括了参数化类型的信息。因此这也是能够通过反射手段取得参数化类型的根本依据。

需要注意，Java属于伪泛型。C#属于真实泛型。

（2）自动装箱、拆箱与遍历循环

见AutoBoxingUnBoxingAndForeach.java

AutoBoxingUnBoxingAndForeach2.java

AutoBoxingTrap.java

（3）条件编译

Java语言没有像C/C++一样使用预处理器指示符（#ifdef）来完成条件编译。Java语言进行条件编译的方法就是使用条件为常量的if语句。

拓展：Java程序命令应当符合下列格式的书写规范

类/接口：符合驼式命名法，首字母大写

方法：符合驼式命名法，首字母小写

字段：类或实例变量：符合驼式命名法，首字母小写

常量：要求全部由大写字母或下划线构成，且第一个字母不能是下划线