第四部分　程序编译与代码优化

第 10 章　前端编译与优化

从计算机程序出现的第一天起，对效率的追逐就是程序员天生的坚定信仰，这个过程犹如一场没有终点、永不停歇的 F1 方程式竞赛，程序员是车手，技术平台则是在赛道上飞驰的赛车。

10.1　概述

在 Java 技术下谈“编译期”而没有具体上下文语境的话，其实是一句很含糊的表述，因为它可能是指一个前端编译器（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 Java 虚拟机的即时编译器（常称 JIT 编译器，Just In Time Compiler）运行期把字节码转变成本地机器码的过程；还可能是指使用静态的提前编译器（常称 AOT 编译器，Ahead Of Time Compiler）直接把程序编译成与目标机器指令集相关的二进制代码的过程。下面笔者列举了这 3 类编译过程里一些比较有代表性的编译器产品：

·前端编译器：JDK 的 Javac、Eclipse JDT 中的增量式编译器（ECJ）[1]。

·即时编译器：HotSpot 虚拟机的 C1、C2 编译器，Graal 编译器。

·提前编译器：JDK 的 Jaotc、GNU Compiler for the Java（GCJ）[2]、Excelsior JET[3]。

这 3 类过程中最符合普通程序员对 Java 程序编译认知的应该是第一类，本章标题中的“前端”指的也是这种由前端编译器完成的编译行为。在本章后续的讨论里，笔者提到的全部“编译期”和“编译器”都仅限于第一类编译过程，我们会把第二、三类编译过程留到第 11 章中去讨论。限制了“编译期”的范围后，我们对于“优化”二字的定义也需要放宽一些，因为 Javac 这类前端编译器对代码的运行效率几乎没有任何优化措施可言（在 JDK 1.3 之后，Javac 的-O 优化参数就不再有意义），哪怕是编译器真的采取了优化措施也不会产生什么实质的效果。因为 Java 虚拟机设计团队选择把对性能的优化全部集中到运行期的即时编译器中，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措施所带来的性能红利。但是，如果把“优化”的定义放宽，把对开发阶段的优化也计算进来的话，Javac 确实是做

了许多针对 Java 语言编码过程的优化措施来降低程序员的编码复杂度、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，而不是依赖字节码或者 Java 虚拟机的底层改进来支持。我们可以这样认为，Java 中即时编译器在运行期的优化过程，支撑了程序执行效率的不断提升；而前端编译器在编译期的优化过程，则是支撑着程序员的编码效率和语言使用者的幸福感的提高。

[1] JDT 官方站点：http://www.eclipse.org/jdt/。
[2] GCJ 官方站点：http://gcc.gnu.org/java/。
[3] Excelsior JET：https://en.wikipedia.org/wiki/Excelsior_JET。

10.2 Javac 编译器

分析源码是了解一项技术的实现内幕最彻底的手段，Javac 编译器不像 HotSpot 虚拟机那样使用 C++语言（包含少量 C 语言）实现，它本身就是一个由 Java 语言编写的程序，这为纯 Java 的程序员了解它的编译过程带来了很大的便利。

10.2.1 Javac 的源码与调试

在 JDK 6 以前，Javac 并不属于标准 Java SE API 的一部分，它实现代码单独存放在 tools.jar 中，要在程序中使用的话就必须把这个库放到类路径上。在 JDK 6 发布时通过了 JSR 199 编译器 API 的提案，使得 Javac 编译器的实现代码晋升成为标准 Java 类库之一，它的源码就改为放在 JDK_SRC_HOME/langtools/src/share/classes/com/sun/tools/javac 中[1]。到了 JDK 9 时，整个 JDK 所有的 Java 类库都采用模块化进行重构划分，Javac 编译器就被挪到了 jdk.compiler 模块（路径为：JDK_SRC_HOME/src/jdk.compiler/share/classes/com/sun/tools/javac）里面。虽然程序代码的内容基本没有变化，但由于本节的主题是源码解析，不可避免地会涉及大量的路径和包名，这就要选定 JDK 版本来讨论了，本次笔者将会以 JDK 9 之前的代码结构来进行讲解。

Javac 编译器除了 JDK 自身的标准类库外，就只引用了 JDK_SRC_HOME/langtools/src/share/classes/com/sun/*里面的代码，所以我们的代码编译环境建立时基本无须处理依赖关系，相当简单便捷。以 Eclipse IDE 作为开发工具为例，先建立一个名为“Compiler_javac”的 Java 工程，然后把 JDK_SRC_HOME/langtools/src/share/classes/com/sun/*目录下的源文件全部复制到工程的源码目录中，如图 10-1 所示。

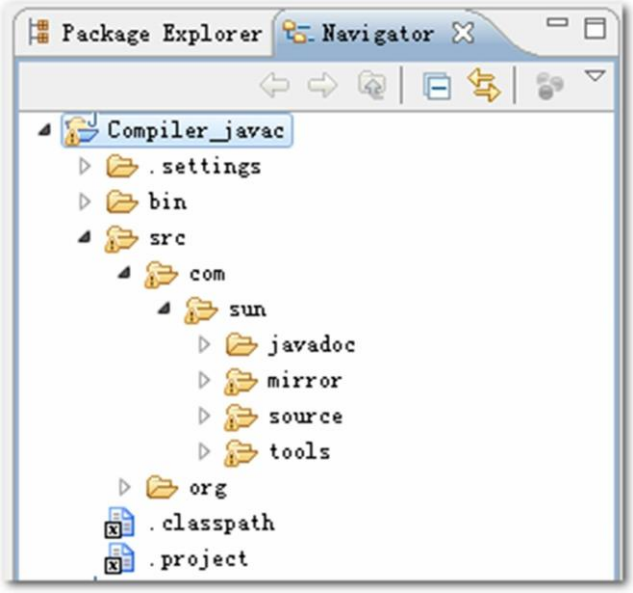

图 10-1　Eclipse 中的 Javac 工程

导入代码期间，源码文件“AnnotationProxyMaker.java”可能会提示“Access Restriction”，被 Eclipse 拒绝编译，如图 10-2 所示。

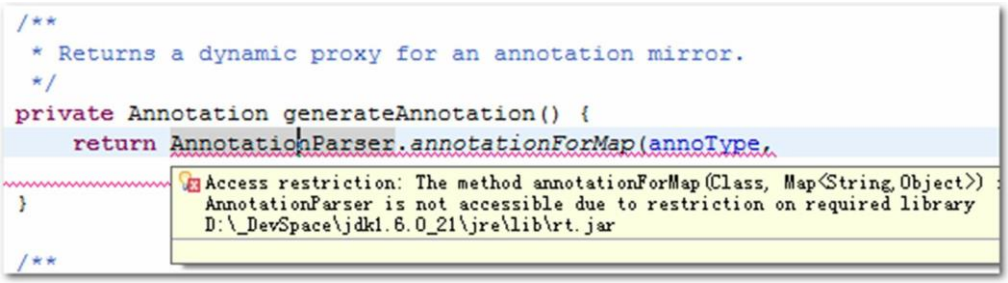

图 10-2　AnnotationProxyMaker 被拒绝编译

这是由于 Eclipse 为了避免开发人员引用非标准 Java 类库可能导致的兼容性问题，在“JRE System Library”设置中默认包含了一系列的代码访问规则（Access Rules），如果代码中引用了这些访问规则所禁止引用的类，就会提示这个错误。我们可以通过添加一条允许访问 JAR 包中所有类的访问规则来解决该问题，如图 10-3 所示。

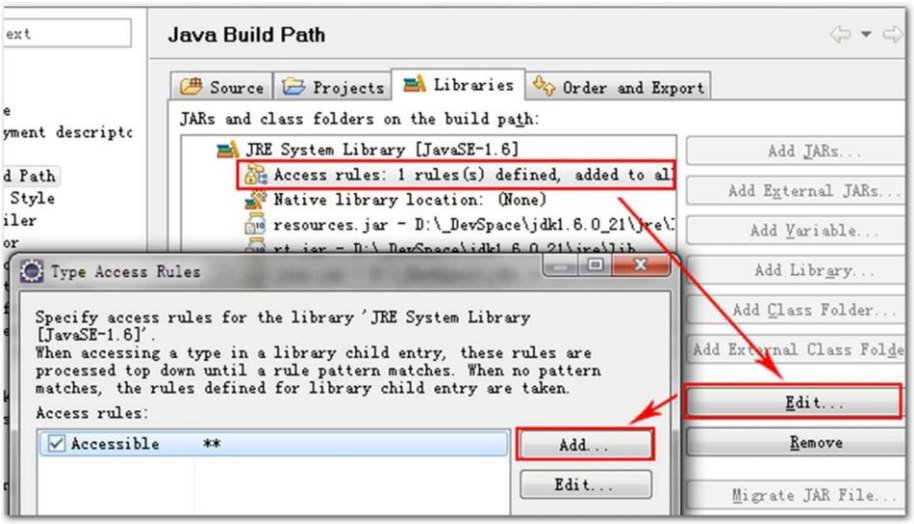

图 10-3　设置访问规则

导入了 Javac 的源码后，就可以运行 com.sun.tools.javac.Main 的 main()方法来执行编译了，可以使用的参数与命令行中使用的 Javac 命令没有任何区别，编译的文件与参数在 Eclipse 的“Debug Configurations”面板中的“Arguments”页签中指定。

《Java 虚拟机规范》中严格定义了 Class 文件格式的各种细节，可是对如何把 Java 源码编译为 Class 文件却描述得相当宽松。规范里尽管有专门的一章名为“Compiling for the Java Virtual Machine”，但这章也仅仅是以举例的形式来介绍怎样的 Java 代码应该被转换为怎样的字节码，并没有使用编译原理中常用的描述工具（如文法、生成式等）来对 Java 源码编译过程加以约束。这是给了 Java 前端编译器较大的实现灵活性，但也导致 Class 文件编译过程在某种程度上是与具体的 JDK 或编译器实现相关的，譬如在一些极端情况下，可能会出现某些代码在 Javac 编译器可以编译，但是 ECJ 编译器就不可以编译的问题（反过来也有可能，后文中将会给出一些这样的例子）。

从 Javac 代码的总体结构来看，编译过程大致可以分为 1 个准备过程和 3 个处理过程，它们分别如下所示。

准备过程：初始化插入式注解处理器。

解析与填充符号表过程，包括：

·词法、语法分析。将源代码的字符流转变为标记集合，构造出抽象语法树。

·填充符号表。产生符号地址和符号信息。 3）插入式注解处理器的注解处理过程：插入式注解处理器的执行阶段，本章的实战部分会设计一个插入式注解处理器来影响 Javac 的编译行为。

4）分析与字节码生成过程，包括：

·标注检查。对语法的静态信息进行检查。

·数据流及控制流分析。对程序动态运行过程进行检查。

·解语法糖。将简化代码编写的语法糖还原为原有的形式。

·字节码生成。将前面各个步骤所生成的信息转化成字节码。

上述 3 个处理过程里，执行插入式注解时又可能会产生新的符号，如果有新的符号产生，就必须转回到之前的解析、填充符号表的过程中重新处理这些新符号，从总体来看，三者之间的关系与交互顺序如图 10-4 所示。

图 10-4　Javac 的编译过程[2]

我们可以把上述处理过程对应到代码中，Javac 编译动作的入口是 com.sun.tools.javac.main.JavaCompiler 类，上述 3 个过程的代码逻辑集中在这个类的 compile()和 compile2() 方法里，其中主体代码如图 10-5 所示，整个编译过程主要的处理由图中标注的 8 个方法来完成。

```
initProcessAnnotations(processors);          准备过程：初始化插入式注解处理器
// These method calls must be chained to avoid memory leaks
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

case BY_TODO:
    while (!todo.isEmpty())
        generate(desugar(flow(attribute(todo.remove()))));
    break;

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
```

图 10-5　Javac 编译过程的主体代码

接下来，我们将对照 Javac 的源代码，逐项讲解上述过程。

[1] 如何获取 OpenJDK 源码请参考本书第 1 章的相关内容。
[2] 图片来源：http://openjdk.java.net/groups/compiler/doc/compilation-overview/index.html，笔者做了汉化处理。

10.2.2　解析与填充符号表

解析过程由图 10-5 中的 parseFiles()方法（图 10-5 中的过程 1.1）来完成，解析过程包括了经典程序编译原理中的词法分析和语法分析两个步骤。

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）集合的过程，单个字符是程序编写时的最小元素，但标记才是编译时的最小元素。关键字、变量名、字面量、运算符都可以作为标记，如“int a=b+2”这句代码中就包含了 6 个标记，分别是 int、a、=、b、+、2，虽然关键字 int 由 3 个字符构成，但是它只是一个独立的标记，不可以再拆分。在 Javac 的源码中，词法分析过程由 com.sun.tools.javac.parser.Scanner 类来实现。

语法分析是根据标记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，抽象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（Syntax Construct），例如包、类型、修饰符、运算符、接口、返回值甚至连代码注释等都可以是一种特定的语法结构。

图 10-6 是 Eclipse AST View 插件分析出来的某段代码的抽象语法树视图，读者可以通过这个插件工具生成的可视化界面对抽象语法树有一个直观的认识。在 Javac 的源码中，语法分析过程由 com.sun.tools.javac.parser.Parser 类实现，这个阶段产出的抽象语法树是以 com.sun.tools.javac.tree.JCTree 类表示的。

经过词法和语法分析生成语法树以后，编译器就不会再对源码字符流进行操作了，后续的操作都建立在抽象语法树之上。

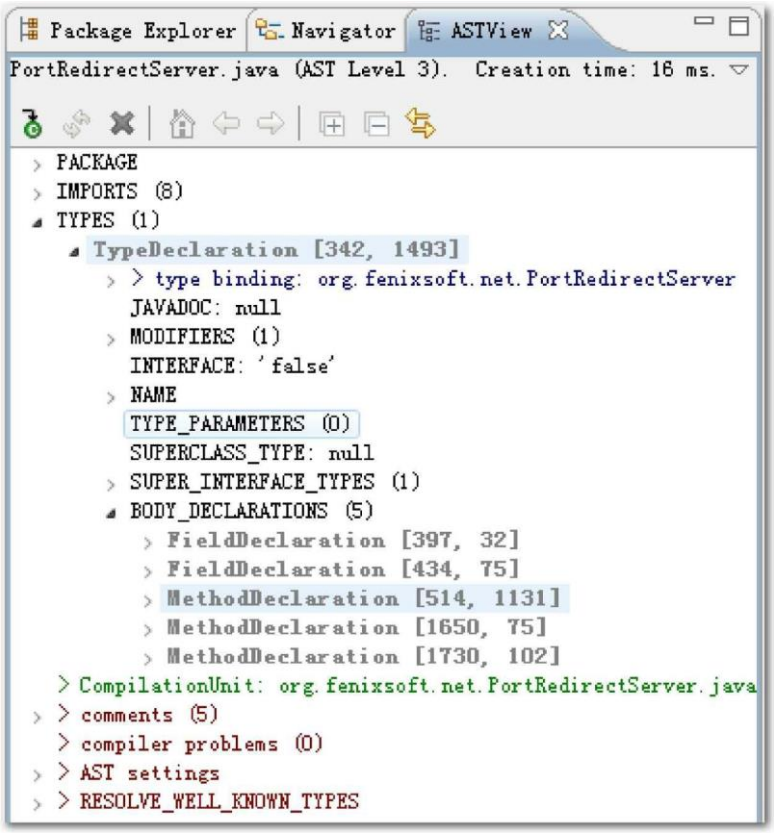

图 10-6　抽象语法树结构视图

2.填充符号表

完成了语法分析和词法分析之后，下一个阶段是对符号表进行填充的过程，也就是图 10-5 中 enterTrees()方法（图 10-5 中注释的过程 1.2）要做的事情。符号表（Symbol Table）是由一组符号地址和符号信息构成的数据结构，读者可以把它类比想象成哈希表中键值对的存储形式（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、栈结构符号表等各种形式）。符号表中所登记的信息在编译的不同阶段都要被用