



超级畅销书全新升级，第1版两年内重印近10次，Java图书领域公认的经典著作，  
繁体版台湾发行

基于最新JDK 1.7，围绕内存管理、执行子系统、程序编译与优化、高效并发等核  
心主题对JVM进行全面而深入的分析，深刻揭示JVM的工作原理

以实践为导向，通过大量与实际生产环境相结合的案例展示了解决各种常见JVM问  
题的技巧和最佳实践



第2版

# 深入理解 Java 虚拟机

JVM高级特性与最佳实践

*Understanding the JVM*

Advanced Features and Best Practices, Second Edition

周志明 著



机械工业出版社  
China Machine Press











































































































































































































































































































































































































































































































## 7.3 类加载的过程

接下来我们详细讲解一下Java虚拟机中类加载的全过程，也就是加载、验证、准备、解析和初始化这5个阶段所执行的具体动作。

### 7.3.1 加载

“加载”是“类加载”（Class Loading）过程的一个阶段，希望读者没有混淆这两个看起来很相似的名词。在加载阶段，虚拟机需要完成以下3件事情：

- 1 ) 通过一个类的全限定名来获取定义此类的二进制字节流。
- 2 ) 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
- 3 ) 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。

虚拟机规范的这3点要求其实并不算具体，因此虚拟机实现与具体应用的灵活度都是相当大的。例如“通过一个类的全限定名来获取定义此类的二进制字节流”这条，它没有指明二进制字节流要从一个Class文件中获取，准确地说是根本没有指明要从哪里获取、怎样获取。虚拟机设计团队在加载阶段搭建了一个相当开放的、广阔的“舞台”，Java发展历程中，充满创造力的开发人员则在这个“舞台”上玩出了各种花样，许多举足轻重的Java技术都建立在这一基础之上，例如：

从ZIP包中读取，这很常见，最终成为日后JAR、EAR、WAR格式的基础。

从网络中获取，这种场景最典型的应用就是Applet。

运行时计算生成，这种场景使用得最多的就是动态代理技术，在java.lang.reflect.Proxy中，就是用了ProxyGenerator.generateProxyClass来为特定接口生成形式为“`*$Proxy`”的代理类的二进制字节流。

由其他文件生成，典型场景是JSP应用，即由JSP文件生成对应的Class类。

从数据库中读取，这种场景相对少见些，例如有些中间件服务器（如SAP Netweaver）可以选择把程序安装到数据库中来完成程序代码在集群间的分发。

.....

相对于类加载过程的其他阶段，一个非数组类的加载阶段（准确地说，是加载阶段中获取类的二进制字节流的动作）是开发人员可控性最强的，因为加载阶段既可以使用系统提供的引导类加载器来完成，也可以由用户自定义的类加载器去完成，开发人员可以通过定义自己的类加载器去控制字节流的获取方式（即重写一个类加载器的loadClass（）方法）。

对于数组类而言，情况就有所不同，数组类本身不通过类加载器创建，它是由Java虚拟机直接创建的。但数组类与类加载器仍然有很密切的关系，因为数组类的元素类型（Element

Type，指的是数组去掉所有维度的类型 ) 最终是要靠类加载器去创建，一个数组类(下面简称为C) 创建过程就遵循以下规则：

如果数组的组件类型( Component Type，指的是数组去掉一个维度的类型)是引用类型，那就递归采用本节中定义的加载过程去加载这个组件类型，数组C将在加载该组件类型的类加载器的类名称空间上被标识(这点很重要，在7.4节会介绍到，一个类必须与类加载器一起确定唯一性)。

如果数组的组件类型不是引用类型(例如int[]数组)，Java虚拟机将会把数组C标记为与引导类加载器关联。

数组类的可见性与它的组件类型的可见性一致，如果组件类型不是引用类型，那数组类的可见性将默认为public。

关于类加载器的话题，笔者将在本章的7.4节专门讲述。

加载阶段完成后，虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在方法区之中，方法区中的数据存储格式由虚拟机实现自行定义，虚拟机规范未规定此区域的具体数据结构。然后在内存中实例化一个java.lang.Class类的对象(并没有明确规定是在Java堆中，对于HotSpot虚拟机而言，Class对象比较特殊，它虽然是对象，但是存放在方法区里面)，这个对象将作为程序访问方法区中的这些类型数据的外部接口。

加载阶段与连接阶段的部分内容(如一部分字节码文件格式验证动作)是交叉进行的，加载阶段尚未完成，连接阶段可能已经开始，但这些夹在加载阶段之中进行的动作，仍然属于连接阶段的内容，这两个阶段的开始时间仍然保持着固定的先后顺序。

### 7.3.2 验证

验证是连接阶段的第一步，这一阶段的目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

Java语言本身是相对安全的语言（依然是相对于C/C++来说），使用纯粹的Java代码无法做到诸如访问数组边界以外的数据、将一个对象转型为它并未实现的类型、跳转到不存在的代码行之类的事情，如果这样做了，编译器将拒绝编译。但前面已经说过，Class文件并不一定要求用Java源码编译而来，可以使用任何途径产生，甚至包括用十六进制编辑器直接编写来产生Class文件。在字节码语言层面上，上述Java代码无法做到的事情都是可以实现的，至少语义上是可以表达出来的。虚拟机如果不检查输入的字节流，对其完全信任的话，很可能因为载入了有害的字节流而导致系统崩溃，所以验证是虚拟机对自身保护的一项重要工作。

验证阶段是非常重要的，这个阶段是否严谨，直接决定了Java虚拟机是否能承受恶意代码的攻击，从执行性能的角度上讲，验证阶段的工作量在虚拟机的类加载子系统中又占了相当大的一部分。《Java虚拟机规范（第2版）》对这个阶段的限制、指导还是比较笼统的，规范中列举了一些Class文件格式中的静态和结构化约束，如果验证到输入的字节流不符合Class文件格式的约束，虚拟机就应抛出一个java.lang.VerifyError异常或其子类异常，但具体应当检查哪些方面，如何检查，何时检查，都没有足够具体的要求和明确的说明。直到2011年发布的《Java虚拟机规范（Java SE 7版）》，大幅增加了描述验证过程的篇幅（从不到10页增加到130页），这时约束和验证规则才变得具体起来。受篇幅所限，本书无法逐条规则去讲解，但从整体上看，验证阶段大致上会完成下面4个阶段的检验动作：文件格式验证、元数据验证、字节码验证、符号引用验证。

#### 1. 文件格式验证

第一阶段要验证字节流是否符合Class文件格式的规范，并且能被当前版本的虚拟机处理。这一阶段可能包括下面这些验证点：

是否以魔数0xCAFEBAE开头。

主、次版本号是否在当前虚拟机处理范围之内。

常量池的常量中是否有不被支持的常量类型（检查常量tag标志）。

指向常量的各种索引值中是否有指向不存在的常量或不符合类型的常量。

CONSTANT\_Utf8\_info型的常量中是否有不符合UTF8编码的数据。

Class文件中各个部分及文件本身是否有被删除的或附加的其他信息。

.....

实际上，第一阶段的验证点还远不止这些，上面这些只是从HotSpot虚拟机源码中摘抄的一小部分内容，该验证阶段的主要目的是保证输入的字节流能正确地解析并存储于方法区

之内，格式上符合描述一个Java类型信息的要求。这阶段的验证是基于二进制字节流进行的，只有通过了这个阶段的验证后，字节流才会进入内存的方法区中进行存储，所以后面的3个验证阶段全部是基于方法区的存储结构进行的，不会再直接操作字节流。

## 2. 元数据验证

第二阶段是对字节码描述的信息进行语义分析，以保证其描述的信息符合Java语言规范的要求，这个阶段可能包括的验证点如下：

这个类是否有父类（除了java.lang.Object之外，所有的类都应当有父类）。

这个类的父类是否继承了不允许被继承的类（被final修饰的类）。

如果这个类不是抽象类，是否实现了其父类或接口之中要求实现的所有方法。

类中的字段、方法是否与父类产生矛盾（例如覆盖了父类的final字段，或者出现不符合规则的方法重载，例如方法参数都一致，但返回值类型却不同等）。

.....

第二阶段的主要目的是对类的元数据信息进行语义校验，保证不存在不符合Java语言规范的元数据信息。

## 3. 字节码验证

第三阶段是整个验证过程中最复杂的一个阶段，主要目的是通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。在第二阶段对元数据信息中的数据类型做完校验后，这个阶段将对类的方法体进行校验分析，保证被校验类的方法在运行时不会做出危害虚拟机安全的事件，例如：

保证任意时刻操作数栈的数据类型与指令代码序列都能配合工作，例如不会出现类似这样的情况：在操作栈放置了一个int类型的数据，使用时却按long类型来加载入本地变量表中。

保证跳转指令不会跳转到方法体以外的字节码指令上。

保证方法体中的类型转换是有效的，例如可以把一个子类对象赋值给父类数据类型，这是安全的，但是把父类对象赋值给子类数据类型，甚至把对象赋值给与它毫无继承关系、完全不相干的一个数据类型，则是危险和不合法的。

.....

如果一个类方法体的字节码没有通过字节码验证，那肯定是有问题的；但如果一个方法体通过了字节码验证，也不能说明其一定就是安全的。即使字节码验证之中进行了大量的检查，也不能保证这一点。这里涉及了离散数学中一个很著名的问题“Halting Problem”[\[2\]](#)：通俗一点的说法就是，通过程序去校验程序逻辑是无法做到绝对准确的——不能通过程序准确地检查出程序是否能在有限的时间之内结束运行。

由于数据流验证的高复杂性，虚拟机设计团队为了避免过多的时间消耗在字节码验证阶段，在JDK 1.6之后的Javac编译器和Java虚拟机中进行了一项优化，给方法体的Code属性的属性表中增加了一项名为“StackMapTable”的属性，这项属性描述了方法体中所有的基本块（Basic Block，按照控制流拆分的代码块）开始时本地变量表和操作栈应有的状态，在字节码验证期间，就不需要根据程序推导这些状态的合法性，只需要检查StackMapTable属性中的记录是否合法即可。这样将字节码验证的类型推导转变为类型检查从而节省一些时间。

理论上StackMapTable属性也存在错误或被篡改的可能，所以是否有可能在恶意篡改了Code属性的同时，也生成相应的StackMapTable属性来骗过虚拟机的类型校验则是虚拟机设计者值得思考的问题。

在JDK 1.6的HotSpot虚拟机中提供了-XX:-UseSplitVerifier选项来关闭这项优化，或者使用参数-XX:+FailOverToOldVerifier要求在类型校验失败的时候退回到旧的类型推导方式进行校验。而在JDK 1.7之后，对于主版本号大于50的Class文件，使用类型检查来完成数据流分析校验则是唯一的选择，不允许再退回到类型推导的校验方式。

#### 4. 符号引用验证

最后一个阶段的校验发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在连接的第三阶段——解析阶段中发生。符号引用验证可以看做是对类自身以外（常量池中的各种符号引用）的信息进行匹配性校验，通常需要校验下列内容：

符号引用中通过字符串描述的全限定名是否能找到对应的类。

在指定类中是否存在符合方法的字段描述符以及简单名称所描述的方法和字段。

符号引用中的类、字段、方法的访问性（private、protected、public、default）是否可被当前类访问。

.....

符号引用验证的目的是确保解析动作能正常执行，如果无法通过符号引用验证，那么将会抛出一个java.lang.IncompatibleClassChangeError异常的子类，如java.lang.IllegalAccessError、java.lang.NoSuchFieldError、java.lang.NoSuchMethodError等。

对于虚拟机的类加载机制来说，验证阶段是一个非常重要的、但不是一定必要（因为对程序运行期没有影响）的阶段。如果所运行的全部代码（包括自己编写的及第三方包中的代码）都已经被反复使用和验证过，那么在实施阶段就可以考虑使用-Xverify:none参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

[1] 源码位置：hotspot\src\share\vm\classfile\classFileParser.cpp。

[2] 停机问题就是判断任意一个程序是否会在有限的时间之内结束运行的问题。如果这个问题可以在有限的时间之内解决，可以有一个程序判断其本身是否会停机并做出相反的行为。这时候显然不管停机问题的结果是什么都不会符合要求，所以这是一个不可解的问题。具体的证明过程可参考：<http://zh.wikipedia.org/zh/停机问题>。

### 7.3.3 准备

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存都将在方法区中进行分配。这个阶段中有两个容易产生混淆的概念需要强调一下，首先，这时候进行内存分配的仅包括类变量（被static修饰的变量），而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在Java堆中。其次，这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量的定义为：

```
public static int value=123;
```

那变量value在准备阶段过后的初始值为0而不是123，因为这时候尚未开始执行任何Java方法，而把value赋值为123的putstatic指令是程序被编译后，存放于类构造器<clinit>()方法之中，所以把value赋值为123的动作将在初始化阶段才会执行。表7-1列出了Java中所有基本数据类型的零值。

表 7-1 基本数据类型的零值

数据类型	零 值	数据类型	零 值
int	0	boolean	false
long	0L	float	0.0f
short	(short) 0	double	0.0d
char	'\u0000'	reference	null
byte	(byte) 0		

上面提到，在“通常情况”下初始值是零值，那相对的会有一些“特殊情况”：如果类字段的字段属性表中存在ConstantValue属性，那在准备阶段变量value就会被初始化为ConstantValue属性所指定的值，假设上面类变量value的定义变为：

```
public static final int value=123;
```

编译时Javac将会为value生成ConstantValue属性，在准备阶段虚拟机就会根据ConstantValue的设置将value赋值为123。

### 7.3.4 解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程，符号引用在前一章讲解Class文件格式的时候已经出现过多次，在Class文件中它以CONSTANT\_Class\_info、CONSTANT\_Fieldref\_info、CONSTANT\_Methodref\_info等类型的常量出现，那解析阶段中所说的直接引用与符号引用又有什么关联呢？

符号引用（Symbolic References）：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到内存中。各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须都是一致的，因为符号引用的字面量形式明确定义在Java虚拟机规范的Class文件格式中。

直接引用（Direct References）：直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是和虚拟机实现的内存布局相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那引用的目标必定已经在内存中存在。

虚拟机规范之中并未规定解析阶段发生的具体时间，只要求了在执行anewarray、checkcast、getfield、getstatic、instanceof、invokedynamic、invokeinterface、invokespecial、invokestatic、invokevirtual、ldc、ldc\_w、multianewarray、new、putfield和putstatic这16个用于操作符号引用的字节码指令之前，先对它们所使用的符号引用进行解析。所以虚拟机实现可以根据需要来判断到底是在类被加载器加载时就对常量池中的符号引用进行解析，还是等到一个符号引用将要被使用前才去解析它。

对同一个符号引用进行多次解析请求是很常见的事情，除invokedynamic指令以外，虚拟机实现可以对第一次解析的结果进行缓存（在运行时常量池中记录直接引用，并把常量标识为已解析状态）从而避免解析动作重复进行。无论是否真正执行了多次解析动作，虚拟机需要保证的是在同一个实体中，如果一个符号引用之前已经被成功解析过，那么后续的引用解析请求就应当一直成功；同样的，如果第一次解析失败了，那么其他指令对这个符号的解析请求也应该收到相同的异常。

对于invokedynamic指令，上面规则则不成立。当碰到某个前面已经由invokedynamic指令触发过解析的符号引用时，并不意味着这个解析结果对于其他invokedynamic指令也同样生效。因为invokedynamic指令的目的本来就是用于动态语言支持（目前仅使用Java语言不会生成这条字节码指令），它所对应的引用称为“动态调用点限定符”（Dynamic Call Site Specifier），这里“动态”的含义就是必须等到程序实际运行到这条指令的时候，解析动作才能进行。相对的，其余可触发解析的指令都是“静态”的，可以在刚刚完成加载阶段，还没有开始执行代码时就进行解析。

解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行，分别对应于常量池的CONSTANT\_Class\_info、CONSTANT\_Fieldref\_info、CONSTANT\_Methodref\_info、CONSTANT\_InterfaceMethodref\_info、CONSTANT\_MethodType\_info、CONSTANT\_MethodHandle\_info和CONSTANT\_InvokeDynamic\_info 7种常量类型。下面将讲解前面4种引用的解析过程，对于后面3种，与JDK 1.7新增的动态语言支持息息相关，由于

Java语言是一门静态类型语言，因此在没有介绍invokedynamic指令的语义之前，没有办法将它们和现在的Java语言对应上，笔者将在第8章介绍动态语言调用时一起分析讲解。

## 1.类或接口的解析

假设当前代码所处的类为D，如果要把一个从未解析过的符号引用N解析为一个类或接口C的直接引用，那虚拟机完成整个解析的过程需要以下3个步骤：

1 ) 如果C不是一个数组类型，那虚拟机将会把代表N的全限定名传递给D的类加载器去加载这个类C。在加载过程中，由于元数据验证、字节码验证的需要，又可能触发其他相关类的加载动作，例如加载这个类的父类或实现的接口。一旦这个加载过程出现了任何异常，解析过程就宣告失败。

2 ) 如果C是一个数组类型，并且数组的元素类型为对象，也就是N的描述符会是类似“[Ljava/lang/Integer”的形式，那将会按照第1点的规则加载数组元素类型。如果N的描述符如前面所假设的形式，需要加载的元素类型就是“java.lang.Integer”，接着由虚拟机生成一个代表此数组维度和元素的数组对象。

3 ) 如果上面的步骤没有出现任何异常，那么C在虚拟机中实际上已经成为一个有效的类或接口了，但在解析完成之前还要进行符号引用验证，确认D是否具备对C的访问权限。如果发现不具备访问权限，将抛出java.lang.IllegalAccessError异常。

## 2.字段解析

要解析一个未被解析过的字段符号引用，首先将会对字段表内class\_index<sup>[2]</sup>项中索引的CONSTANT\_Class\_info符号引用进行解析，也就是字段所属的类或接口的符号引用。如果在解析这个类或接口符号引用的过程中出现了任何异常，都会导致字段符号引用解析的失败。如果解析成功完成，那将这个字段所属的类或接口用C表示，虚拟机规范要求按照如下步骤对C进行后续字段的搜索。

1 ) 如果C本身就包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。

2 ) 否则，如果在C中实现了接口，将会按照继承关系从下往上递归搜索各个接口和它的父接口，如果接口中包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。

3 ) 否则，如果C不是java.lang.Object的话，将会按照继承关系从下往上递归搜索其父类，如果在父类中包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。

4 ) 否则，查找失败，抛出java.lang.NoSuchFieldError异常。

如果查找过程成功返回了引用，将会对这个字段进行权限验证，如果发现不具备对字段的访问权限，将抛出java.lang.IllegalAccessError异常。

在实际应用中，虚拟机的编译器实现可能会比上述规范要求得更加严格一些，如果有

个同名字段同时出现在C的接口和父类中，或者同时在自己或父类的多个接口中出现，那编译器将可能拒绝编译。在代码清单7-4中，如果注释了Sub类中的“public static int A=4；”，接口与父类同时存在字段A，那编译器将提示“The field Sub.A is ambiguous”，并且拒绝编译这段代码。

#### 代码清单7-4 字段解析

```
package org.fenixsoft.classloading;
public class FieldResolution{
interface Interface0{
int A=0;
}
interface Interface1 extends Interface0{
int A=1;
}
interface Interface2{
int A=2;
}
static class Parent implements Interface1{
public static int A=3;
}
static class Sub extends Parent implements Interface2{
public static int A=4;
}
public static void main(String[]args){
System.out.println(Sub.A);
}
}
```

### 3.类方法解析

类方法解析的第一个步骤与字段解析一样，也需要先解析出类方法表的class\_index<sup>[3]</sup>项中索引的方法所属的类或接口的符号引用，如果解析成功，我们依然用C表示这个类，接下来虚拟机将会按照如下步骤进行后续的类方法搜索。

1 ) 类方法和接口方法符号引用的常量类型定义是分开的，如果在类方法表中发现class\_index中索引的C是个接口，那就直接抛出java.lang.IncompatibleClassChangeError异常。

2 ) 如果通过了第1步，在类C中查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。

3 ) 否则，在类C的父类中递归查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。

4 ) 否则，在类C实现的接口列表及它们的父接口之中递归查找是否有简单名称和描述符都与目标相匹配的方法，如果存在匹配的方法，说明类C是一个抽象类，这时查找结束，抛出java.lang.AbstractMethodError异常。

5 ) 否则，宣告方法查找失败，抛出java.lang.NoSuchMethodError。

最后，如果查找过程成功返回了直接引用，将会对这个方法进行权限验证，如果发现不具备对此方法的访问权限，将抛出java.lang.IllegalAccessError异常。

### 4.接口方法解析

接口方法也需要先解析出接口方法表的class\_index<sup>[4]</sup>项中索引的方法所属的类或接口的符号引用，如果解析成功，依然用C表示这个接口，接下来虚拟机将会按照如下步骤进行后续的接口方法搜索。

1 ) 与类方法解析不同 , 如果在接口方法表中发现class\_index中的索引C是个类而不是接口 , 那就直接抛出java.lang.IncompatibleClassChangeError异常。

2 ) 否则 , 在接口C中查找是否有简单名称和描述符都与目标相匹配的方法 , 如果有则返回这个方法的直接引用 , 查找结束。

3 ) 否则 , 在接口C的父接口中递归查找 , 直到java.lang.Object类 ( 查找范围会包括Object类 ) 为止 , 看是否有简单名称和描述符都与目标相匹配的方法 , 如果有则返回这个方法的直接引用 , 查找结束。

4 ) 否则 , 宣告方法查找失败 , 抛出java.lang.NoSuchMethodError异常。

由于接口中的所有方法默认都是public的 , 所以不存在访问权限的问题 , 因此接口方法的符号解析应当不会抛出java.lang.IllegalAccessError异常。

[1] 严格来说 , CONSTANT\_String\_info和CONSTANT\_InterfaceMethodref\_info这两种类型的常量也有解析过程 , 但很简单、直观 , 不再做单独介绍。

[2] 参见第6章中关于CONSTANT\_Fieldref\_info常量的内容。

[3] 参见第6章关于CONSTANT\_Methodref\_info常量的内容。

[4] 参见第6章中关于CONSTANT\_InterfaceMethodref\_info常量的内容。

























































































































































































































## 11.2.4 查看及分析即时编译结果

一般来说，虚拟机的即时编译过程对用户程序是完全透明的，虚拟机通过解释执行代码还是编译执行代码，对于用户来说并没有什么影响（执行结果没有影响，速度上会有很大差别），在大多数情况下用户也没有必要知道。但是虚拟机也提供了一些参数用来输出即时编译和某些优化手段（如方法内联）的执行状况，本节将介绍如何从外部观察虚拟机的即时编译行为。

本节中提到的运行参数有一部分需要Debug或FastDebug版虚拟机的支持，Product版的虚拟机无法使用这部分参数。如果读者使用的是根据本书第1章的内容自己编译的JDK，注意将SKIP\_DEBUG\_BUILD或SKIP\_FASTDEBUG\_BUILD参数设置为false，也可以在OpenJDK网站上直接下载FastDebug版的JDK（从JDK 6u25之后Oracle官网就不再提供FastDebug的JDK下载了）。注意，本节中所有的测试都基于代码清单11-2所示的Java代码。

代码清单11-2 测试代码

```
public static final int NUM=15000;
public static int doubleValue( int i ){
    //这个空循环用于后面演示JIT代码优化过程
    for( int j=0; j<100000; j++ );
    return i*2;
}
public static long calcSum( ){
    long sum=0;
    for( int i=1; i<=100 ; i++ ){
        sum+=doubleValue( i );
    }
    return sum;
}
public static void main( String[]args ){
    for( int i=0; i<NUM; i++ ){
        calcSum( );
    }
}
```

首先运行这段代码，并且确认这段代码是否触发了即时编译，要知道某个方法是否被编译过，可以使用参数-XX:+PrintCompilation要求虚拟机在即时编译时将被编译成本地代码的方法名称打印出来，如代码清单11-3所示（其中带有“%”的输出说明是由回边计数器触发的OSR编译）。

代码清单11-3 被即时编译的代码

```
VM option'+PrintCompilation'
310 1 java.lang.String:charAt( 33 bytes )
329 2 org.fenixsoft.jit.Test:calcSum( 26 bytes )
329 3 org.fenixsoft.jit.Test:doubleValue( 4 bytes )
332 1%org.fenixsoft.jit.Test:main@5( 20 bytes )
```

从代码清单11-3输出的确认信息中可以确认main()、calcSum()和doubleValue()方法已经被编译，我们还可以加上参数-XX:+PrintInlining要求虚拟机输出方法内联信息，如代码清单11-4所示。

代码清单11-4 内联信息

```
VM option'+PrintCompilation'
VM option'+PrintInlining'
273 1 java.lang.String:charAt( 33 bytes )
291 2 org.fenixsoft.jit.Test:calcSum( 26 bytes )
@9 org.fenixsoft.jit.Test:doubleValue inline( hot )
294 3 org.fenixsoft.jit.Test:doubleValue( 4 bytes )
295 1%org.fenixsoft.jit.Test:main@5( 20 bytes )
@5 org.fenixsoft.jit.Test:calcSum inline( hot )
@9 org.fenixsoft.jit.Test:doubleValue inline( hot )
```

从代码清单11-4的输出中可以看到方法doubleValue( )被内联编译到calcSum( )中，而calcSum( )又被内联编译到方法main( )中，所以虚拟机再次执行main( )方法的时候（举例而已，main( )方法并不会运行两次），calcSum( )和doubleValue( )方法都不会再被调用，它们的代码逻辑都被直接内联到main( )方法中了。

除了查看哪些方法被编译之外，还可以进一步查看即时编译器生成的机器码内容，不过如果虚拟机输出一串0和1，对于我们的阅读来说是没有意义的，机器码必须反汇编成基本的汇编语言才可能被阅读。虚拟机提供了一组通用的反汇编接口<sup>[1]</sup>，可以接入各种平台下的反汇编适配器来使用，如使用32位80x86平台则选用hsdis-i386适配器，其余平台的适配器还有hsdis-amd64、hsdis-sparc和hsdis-sparcv9等，可以下载或自己编译出反汇编适配器<sup>[2]</sup>，然后将其放置在JRE/bin/client或/server目录下，只要与jvm.dll的路径相同即可被虚拟机调用。在为虚拟机安装了反汇编适配器之后，就可以使用-XX:+PrintAssembly参数要求虚拟机打印编译方法的汇编代码了，具体的操作可以参考本书4.2.7节。

如果没有HSDIS插件支持，也可以使用-XX:+PrintOptoAssembly（用于Server VM）或-XX:+PrintLIR（用于Client VM）来输出比较接近最终结果的中间代码表示，代码清单11-2被编译后部分反汇编（使用-XX:+PrintOptoAssembly）的输出结果如代码清单11-5所示。从阅读角度来说，使用-XX:+PrintOptoAssembly参数输出的伪汇编结果包含了更多的信息（主要是注释），利于阅读并理解虚拟机JIT编译器的优化结果。

### 代码清单11-5 本地机器码反汇编信息（部分）

```
....  
000 B1 : #N1<-BLOCK HEAD IS JUNK Freq:1  
000 pushq rbp  
subq rsp, #16#Create frame  
nop#nop_for_patch_verified_entry  
006 movl RAX,RDX#spill  
008 sall RAX, #1  
00a addq rsp, 16#Destroy frame  
popq rbp  
testl rax, [rip+#offset_to_poll_page]#SafePoint:poll for GC  
....
```

前面提到的使用-XX:+PrintAssembly参数输出反汇编信息需要Debug或者FastDebug版的虚拟机才能直接支持，如果使用Product版的虚拟机，则需要加入参数-XX:+UnlockDiagnosticVMOptions打开虚拟机诊断模式后才能使用。

如果除了本地代码的生成结果外，还想再进一步跟踪本地代码生成的具体过程，那还可以使用参数-XX:+PrintCFGToFile（使用Client Compiler）或-XX:+PrintIdealGraphFile（使用Server Compiler）令虚拟机将编译过程中各个阶段的数据（例如，对C1编译器来说，包括字节码、HIR生成、LIR生成、寄存器分配过程、本地代码生成等数据）输出到文件中。然后使用Java HotSpot Client Compiler Visualizer<sup>[3]</sup>（用于分析Client Compiler）或Ideal Graph Visualizer<sup>[4]</sup>（用于分析Server Compiler）打开这些数据文件进行分析。以Server Compiler为例，笔者分析一下JIT编译器的代码生成过程。

Server Compiler的中间代码表示是一种名为Ideal的SSA形式程序依赖图（Program Dependence Graph），在运行Java程序的JVM参数中加入“-XX:+PrintIdealGraphLevel=2-XX:+PrintIdealGraphFile=ideal.xml”，编译后将产生一个名为ideal.xml的文件，它包含了Server Compiler编译代码的过程信息，可以使用Ideal Graph Visualizer对这些信息进行分析。

Ideal Graph Visualizer加载ideal.xml文件后，在Outline面板上将显示程序运行过程中编译















































































































































































































