第三章

编译 Java 虚拟机

Java 虚拟机被设计为支持 Java 编程语言。Oracle 的 JDK 软件包含从 Java 编程语言编写的源代码到 Java 虚拟机指令集的编译器，以及实现 Java 虚拟机本身的运行时系统。理解一个编译器如何利用 Java 虚拟机对于未来的编译器编写者以及试图理解 Java 虚拟机本身的人都很有用。本章中编号的章节是不规范的。

请注意，当指从 Java 虚拟机的指令集到特定 CPU 的指令集的翻译器时，有时会使用术语“编译器”。这种翻译器的一个例子是即时(JIT)代码生成器，它只在加载 Java 虚拟机代码之后生成特定于平台的指令。本章不涉及与代码生成相关的问题，只涉及将 Java 编程语言编写的源代码编译成 Java 虚拟机指令的问题。

3.1 示例格式

本章主要由一些源代码示例和带注释的 Java 虚拟机代码清单组成，这些代码是 Oracle 的 JDK 1.0.2 版本中的 javac 编译器为这些示例生成的。Java 虚拟机代码是用 Oracle 的 javap 实用程序输出的非正式“虚拟机汇编语言”编写的，随 JDK 发行版分发。您可以使用 javap 生成编译方法的其他示例。

阅读过汇编代码的任何人都应该熟悉示例的格式。每个指令的形式如下：

```
<index> <opcode> [ <operand1> [ <operand2>... ]] [<comment>]
```

<index>是数组中指令操作码的索引，该数组包含此方法的 Java 虚拟机代码字节。或者，可以将 < index > 视为从方法开始的字节偏移量。 < opcode > 是指令操作码的助记符，零个或多个 < operandN > 是指令的操作数。可选的<comment>在行尾注释语法中给出：

```
8 bipush 100  // Push int constant 100
```

注释中的一些内容是由 javap 发出的;其余部分由作者提供。每个指令前面的<index>可以用作控制传输指令的目标。例如，goto 8 指令将控制转移到索引为 8 的指令。注意，Java 虚拟机控制传输指令的实际操作数是这些指令的操作码地址的偏移量; 这些操作数由 javap 显示（本章中显示），以便更容易地将偏移量读入其方法。

我们在表示运行时常量池索引的操作数的前面加上一个哈希符号，并在指令之后加上一个标识引用的运行时常量池项的注释，如下所示：

```
10 ldc #1 // Push float constant 100.0
```

或者：

```
9 invokevirtual #4    // Method Example.addTwo(II)I
```

就本章的目的而言，我们不必担心指定操作数大小等细节。

3.2 常量、局部变量和控制结构的使用

Java 虚拟机代码展示了一组由 Java 虚拟机的类型设计和使用强加的通用特征。在第一个例子中，我们遇到了许多这样的例子，并对它们进行了一些详细的考虑。

spin 方法简单地围绕空的 for 循环执行 100 次:

```
void spin() {
    int i;
    for (i = 0; i < 100; i++) {
        ;    // Loop body is empty
    }
}
```

编译器可以将 spin 编译为:

```
0        iconst_0          Push int constant 0
1        istore_1          Store into local variable 1 (i=0)
2        goto 8            First time through don't increment
5        iinc 1 1          Increment local variable 1 by 1 (i++)
8        iload_1           Push local variable 1 (i)
9        bipush 100        Push int constant 100
11       if_icmplt 5       Compare and loop if less than (i < 100)
14       return            Return void when done
```

Java 虚拟机是面向栈的，大多数操作从 Java 虚拟机当前帧的操作数栈中获取一个或多个操作数，或者将结果推回操作数栈。每次调用一个方法都会创建一个新的帧，并随之创建一个新的操作数栈和一组局部变量供该方法使用(§2.6)。因此，在计算的任何一点上，每个控制线程都可能有许多帧和相同数量的操作数栈，对应于许多嵌套的方法调用。只有当前帧中的操作数栈是活动的。

Java 虚拟机的指令集通过对各种数据类型的操作使用不同的字节码来区分操作数类型。 方法 spin 只对 int 类型的值起作用。其编译代码中用于操作类型化数据(iconst_0、istore_1、iinc、iload_1、if_icmpult)的指令都专门用于 int 类型。

使用两条不同的指令将 spin 中的两个常量 0 和 100 压入操作数栈。0 是使用 iconst_0 指令推送的，它是 iconst_<i>指令家族中的一个。使用 bipush 指令推送 100，该指令获取它推送的值作为立即操作数。

Java 虚拟机经常利用某些操作数的可能性(在 iconst_<i>指令的情况下，int 常量-1、0、1、2、3、4 和 5)，使这些操作数隐式地出现在操作代码中。因为 iconst_0 指令知道它将推入一个 int 0，所以 iconst_0 不需要存储操作数来告诉它要推入什么值，也不需要获取或解码操作数。将 push 0 编译为 bipush 0 是正确的，但会使 spin 的编译代码多一个字节。 简单的虚拟机在循环中每次都会花费额外的时间获取和解码显式操作数。隐式操作数的使用使编译代码更加紧凑和高效。

spin 中的 int i 存储为 Java 虚拟机局部变量 1。因为大多数 Java 虚拟机指令对从操作数栈弹出的值进行操作，而不是直接对局部变量进行操作，在为 Java 虚拟机编译的代码中，在局部变量和操作数堆栈之间传输值的指令很常见。这些操作在指令集中也有特殊的支持。在 spin 中，使用 istore_1 和 iload_1 指令将值传递到局部变量或从局部变量传递值，每个指令都隐式地对局部变量 1 进行操作。istore_1 指令从操作数栈中弹出一个整数，并将其存储

在局部变量 1 中。iload_1 指令将局部变量 1 中的值推送到操作数栈。

局部变量的使用（和重用）是编译器编写者的责任。专门的加载和存储指令应鼓励编译器编写者尽可能多地重用局部变量。生成的代码更快、更紧凑，并且在帧中使用更少的空间。

Java 虚拟机专门针对局部变量的某些非常频繁的操作。iinc 指令将局部变量的内容递增一字节有符号值。spin 中的 iinc 指令将第一个局部变量（其第一个操作数）递增 1（其第二个操作数）。iinc 指令在实现循环构造时非常方便。

spin 的 for 循环主要由以下指令完成：

```
5 iinc 1 1          // Increment local variable 1 by 1 (i++)
8 iload_1           // Push local variable 1 (i)
9 bipush 100        // Push int constant 100
11 if_icmplt 5      // Compare and loop if less than (i < 100)
```

bipush 指令将值 100 作为整数推送到操作数栈上，然后 if_icmplt 指令从操作数栈中弹出该值，并将其与 i 进行比较。如果比较成功（变量 i 小于 100），则控制转移到索引 5，并开始 for 循环的下一次迭代。 否则，控制传递到 if_icmplt 之后的指令。

如果 spin 示例对循环计数器使用了 int 以外的数据类型，则编译的代码必须更改以反映不同的数据类型。例如，如果 spin 示例使用的不是 int，而是 double，如下所示：

```
void dspin() {
  double i;
  for (i = 0.0; i < 100.0; i++) {
    ;// Loop body is empty
  }
}
```

编译的代码是：

```
Method void
0  dconst 0         // Push double constant 0.0
1  dstore_1         // Store into  local variables 1 and 2
2  goto 9           // First time  through don't increment
5  dload_1          // Push local  variables 1 and 2
6  dconst_1         // Push double constant 1.0
7  dadd             // Add; there  is no dinc instruction
8  dstore_1         // Store result in local variables 1 and 2
9  dload_1          // Push local  variables 1 and 2
10 ldc2_w #4        // Push double constant 100.0
13 dcmpg            // There is no if_dcmplt instruction
14 iflt 5           // Compare and loop if less than (i < 100. 0)
17 return           // Return void when done
```

操作类型化数据的指令现在专用于类型 double。(ldc2_w 指令将在本章后面讨论。)

回想一下，double 值占用两个局部变量，尽管它们只能使用两个局部变量中较小的索引进行访问。对于 long 类型的值也是如此。再次举例来说，

```
double doubleLocals(double d1, double d2) {
  return d1 + d2;
}
```

编译成

```
Method double doubleLocals(double,double)
```

```
0   dload_1       // First argument in local variables 1 and 2
1   dload_3       // Second argument in local variables 3 and 4
2   dadd
3   dreturn
```

请注意，用于在 doubleLocals 中存储 double 值的局部变量对的局部变量绝对不能单独操作。

Java 虚拟机的操作码大小为 1 字节，因此编译的代码非常紧凑。然而，1 字节操作码也意味着 Java 虚拟机指令集必须保持较小。作为折衷方案，Java 虚拟机不为所有数据类型提供同等的支持：它不是完全正交的（表 2.11.1-A）。

例如，可以使用单个 if_icmplt 指令来实现示例 spin 的 for 语句中 int 类型值的比较； 但是，Java 虚拟机指令集中没有一条指令对 double 类型的值执行条件分支。因此，dspin 必须使用一个 dcmpg 指令和一个 iflt 指令来实现 double 类型值的比较。

Java 虚拟机为 int 类型的数据提供了最直接的支持。这部分是因为预期 Java 虚拟机的操作数栈和局部变量数组的有效实现。它也是由典型程序中频繁使用 int 数据引起的。其他整数类型的直接支持较少。例如，没有 store,load 或 add 指令的 byte,char 或 short 版本。下面是用 short 写的 spin 例子:

```
void sspin() {
    short i;
    for (i = 0; i < 100; i++) {
        ;    // Loop body is empty
    }
}
```

它必须为 Java 虚拟机编译，如下所示，使用操作另一种类型(很可能是 int)的指令，根据需要在 short 和 int 值之间进行转换，以确保对 short 数据的操作结果保持在适当的范围内:

```
Method void sspin()
0  iconst_0
1  istore_1
2  goto 10
5  iload_1          // The short is treated as though an int
6  iconst_1
7  iadd
8  i2s              // Truncate int to short
9  istore_1
10 iload_1
11 bipush 100
13 if_icmplt 5
16 return
```

Java 虚拟机中缺少对 byte、char 和 short 类型的直接支持并不是特别痛苦，因为这些类型的值在内部被提升为 int (byte 和 short 被符号扩展为 int, char 被零扩展)。因此，可以使用 int 指令对 byte、char 和 short 数据进行操作。唯一的额外成本是将 int 运算的值截断为有效范围。

long 类型和浮点类型在 Java 虚拟机中具有中间级别的支持，仅缺少条件控制传输指令的完整补充。

3.3 计算

Java 虚拟机通常对其操作数栈进行运算。(唯一的例外是 iinc 指令，它直接递增局部变量的值。) 例如，align2grain 方法将 int 值与给定的 2 的幂对齐:

```
int align2grain(int i, int grain) {
  return ((i + grain-1) & ~(grain-1));
}
```

算术运算的操作数从操作数栈中弹出，运算结果被推回操作数栈。因此，算术子计算的结果可以作为嵌套计算的操作数。例如，~（grain-1）的计算由以下指令处理:

```
5   iload_2         // Push grain
6   iconst_1        // Push int constant 1
7   isub            // Subtract; push result
8   iconst_m1       // Push int constant -1
9   ixor            // Do XOR; push result
```

首先使用局部变量 2 的内容和一个即时 int 值 1 计算 grain-1。这些操作数从操作数栈中弹出，它们的差值被推回到操作数栈中。因此，这个差异可以立即作为 ixor 指令的一个操作数使用。(回想一下~x == -1^x。) 类似地，ixor 指令的结果成为后续 iand 指令的操作数。

整个方法的代码如下:

```
Method int align2grain(int,int)
0 iload_1
1 iload_2
2 iadd
3 iconst_1
4 isub
5 iload_2
6 iconst_1
7 isub
8 iconst_m1
9 ixor
10 iand
11 ireturn
```


3.4 访问运行时常量池

许多数值常量以及对象、字段和方法都是通过当前类的运行时常量池访问的。对象访问稍后会被考虑(§3.8)。int、long、float 和 double 类型的数据，以及对 String 类实例的引用，使用 ldc、ldc_w 和 ldc2_w 指令进行管理。

ldc 和 ldc_w 指令用于访问运行时常量池(包括 String 类的实例)中除 double 和 long 以外的类型的值。只有当有大量运行时常量池项，并且需要一个更大的索引来访问一个项时，才使用 ldc_w 指令来代替 ldc。ldc2_w 指令用于访问 double 和 long 类型的所有值;没有不广泛的变体。

byte、char 或 short 类型的整型常量，以及小的 int 值，可以使用 bipush、sipush 或 iconst_<i>指令进行编译(§3.2)。某些小型浮点常量可以使用 fconst_<f>和 dconst_<d>指令编译。

在所有这些情况下，编译都很简单。例如，以下的常量:

```
void useManyNumeric() {
    int i = 100;
    int j = 1000000;
    long l1 = 1;
    long l2 = 0xffffffff;
    double d = 2.2;
    ...do some calculations...
}
```

设置如下:

```
Method void useManyNumeric()
0 bipush 100      // Push small int constant with bipush
2 istore_1
3 ldc #1          // Push large int constant (1000000) with ldc
5 istore_2
6 lconst_1        // A tiny long value uses small fast lconst_1
7 lstore_3
8 ldc2_w #6       // Push long 0xffffffff (that is, an int -1)
          // Any long constant value can be pushed with ldc2_w
11 lstore 5
13 ldc2_w #8   // Push double constant 2.200000
          // Uncommon double values are also pushed with ldc2_w
16 dstore 7
...do those calculations...
```

3.5 更多控制示例

for 语句的编译已在前面的章节(§3.2)中展示。Java 编程语言的大多数其他控制结构(if-then-else、do、while、break 和 continue)也采用明显的方式进行编译。switch 语句的编译是在单独的章节中处理的(§3.10)，异常的编译(§3.12)和 finally 子句的编译(§3.13)也是如此。

作为进一步的例子，虽然 Java 虚拟机提供的特定控制传输指令因数据类型而异，但它以一种明显的方式编译 while 循环。像往常一样，对 int 类型的数据有更多的支持，例如:

```
void whileInt() {
    int i = 0;
    while (i < 100) {
      i++;
    }
}
```

编译为:

```
Method void whileInt()
0 iconst_0
1   istore_1
2   goto 8
5   iinc 1 1
8   iload_1
9   bipush 100
11  if_icmplt 5
14  return
```

注意，while 语句的测试(使用 if_icmplt 指令实现)位于循环的 Java 虚拟机代码的底部。(在之前的 spin 例子中也是如此。) 位于循环底部的测试强制使用 goto 指令在循环的第一次迭代之前进行测试。如果该测试失败，并且没有进入循环体，那么这条额外的指令就浪费了。然而，虽然循环通常在预期运行其主体时使用，但通常是多次迭代。对于后续的迭代，将测试放在循环的底部，每次循环都会保存一条 Java 虚拟机指令: 如果测试位于循环的顶部，则循环体将需要一个尾随的 goto 指令来返回到顶部。

涉及其他数据类型的控件构造以类似的方式编译，但必须使用这些数据类型可用的指令。这将导致代码的效率降低，因为需要更多的 Java 虚拟机指令，例如:

```
void whileDouble() {
    double i = 0.0;
    while (i < 100.1) {
      i++;
    }
}
```

编译为:

```
Method void whileDouble()
 0  dconst_0
 1  dstore_1
 2  goto 9
 5  dload_1
 6  dconst_1
 7  dadd
 8  dstore_1
 9  dload_1
//  10 ldc2_w #4   Push double constant 100.1
//  13 dcmpg       To compare and branch we have to use...
    14 iflt 5          // ...two instructions
    17 return
```

每种浮点类型都有两个比较指令:float 类型使用 fcmpl 和 fcmpg, double 类型使用 dcmpl 和 dcmpg。这些变体的区别仅在于它们对 NaN 的处理。NaN 是无序的（§2.3.2），因此所有浮点比较都会失败，如果其中一个操作数是 NaN。编译器为适当类型选择比较指令的变体，无论对非 NaN 值的比较失败还是遇到 NaN，该变体都会产生相同的结果。例如:

```
int lessThan100(double d) {
    if (d < 100.0) {
      return 1;
    } else {
      return -1;
    }
}
```

编译为:

```
Method int lessThan100(double)
0 dload_1
1 ldc2_w #4 //Push double constant 100.0
4 dcmpl     //Push -1 if d is NaN or d < 100.0
            //push 0 if d == 100.0
5 ifle 10   //Branch on 0 or -1
```

```
8 iconst_1
9 ireturn
10 iconst_m1
11 ireturn
```

如果 d 不是 NaN 并且小于 100.0,dcmpg 指令将一个 int -1 压入操作数栈，并且 ifge 指令不进行分支。无论 d 是否大于 100.0 或为 NaN, dcmpg 指令都将 int 1 压入操作数栈，执行 ifge 分支。如果 d 等于 100.0,dcmpg 指令将一个 int 0 压入操作数栈，然后执行 ifge 分支。

如果反转比较，则 dcmpl 指令可达到相同的效果:

```
int greaterThan100(double d) {
    if (d > 100.0) {
        return 1;
    } else {
        return -1;
    }
}
```

编译为:

```
Method int greaterThan100(double)
0   dload 1
1   ldc2_w #4       // Push double constant 100.0
4   dcmpl           // Push -1 if d is NaN or d <
                    // push 0 if d == 100.0
5   ifle 10         // Branch on 0 or -1
8   iconst_1
9   ireturn
10 iconst_m1
11 ireturn
```

同样，无论对非 NaN 值的比较失败，还是因为传递了 NaN, dcmpl 指令都会将一个 int 值压入操作数栈，从而导致 ifle 发生分支。如果两个 dcmp 指令都不存在，那么其中一个示例方法将不得不做更多的工作来检测 NaN。

3.6 接收参数

如果向实例方法传递了 n 个参数，按照惯例，它们将在为新方法调用创建的帧中编号为 1 到 n 的局部变量中接收。接收参数的顺序与传递参数的顺序相同。例如:

```
int addTwo(int i, int j) {
  return i + j;
}
```

编译为:

```
Method int addTwo(int,int)
0   iload_1         //  Push value  of local variable 1  (i)
1   iload_2         //  Push value  of local variable 2  (j)
2   iadd            //  Add; leave  int result on operand stack
3   ireturn         //  Return int result
```

按照约定，实例方法在局部变量 0 中传递对其实例的 reference。在 Java 编程语言中，可以通过 this 关键字访问实例。

类（静态）方法没有实例，因此对它们来说，不需要使用局部变量 0。类方法开始使用索引 0 处的局部变量。如果 addTwo 方法是类方法，则其参数将以与第一个版本类似的方式传递：

```
static int addTwoStatic(int i, int j) {
  return i + j;
}
```

编译为：

```
Method int addTwoStatic(int,int)
0   iload_0
1   iload_1
2   iadd
3   ireturn
```

唯一的区别是方法参数以局部变量 0 而不是 1 开始。

3.7 调用方法

实例方法的正常方法调用根据对象的运行时类型进行调度。（用 C++术语来说，它们是 virtual 的。）这种调用是使用 invokevirtual 指令实现的，该指令将运行时常量池条目的索引作为其参数，该索引给出了对象的类类型的二进制名称的内部形式、要调用的方法的名称以及该方法的描述符（§4.3.3）。要调用前面定义为实例方法的 addTwo 方法，我们可以编写：

```
int add12and13() {
    return addTwo(12, 13); }
```

编译为：

```
Method int add12and13()
0 aload_0           // Push local variable 0 (this)
1 bipush 12         // Push int constant 12
3 bipush_13         //Push int constant 13
5 invokevirtual #4 //Method Example.addtwo(II)I
8 ireturn           // Return int on top of operand stack;
                    // it is the int result of addTwo()
```

调用是通过首先将对当前实例的引用 this 推入操作数栈来设置的。然后推送方法调用的参数，int 值 12 和 13。当创建 addTwo 方法的帧时，传递给该方法的参数将成为新帧局部变量的初始值。也就是说，this 和两个参数的引用(由调用者推入操作数栈)将成为被调用方法的局部变量 0、1 和 2 的初始值。

最后，调用 addTwo。当它返回时，它的 int 返回值被压入调用者的 add12and13 方法帧的操作数栈。因此，返回值将立即返回给 add12and13 的调用者。

add12and13 的返回是由 add12and13 的 ireturn 指令处理的。ireturn 指令接受 addTwo 在当前帧的操作数栈上返回的 int 值，并将其压入调用者的帧的操作数栈。然后它将控制权返回给调用者，使调用者的帧处于当前状态。Java 虚拟机为许多数值和引用数据类型提供

了不同的返回指令，也为没有返回值的方法提供了返回指令。所有类型的方法调用都使用同一组返回指令。

invokevirtual 指令的操作数（在本例中，运行时常量池索引#4）不是类实例中方法的偏移量。编译器不知道类实例的内部布局。相反，它生成对实例方法的符号引用，这些引用存储在运行时常量池中。这些运行时常量池项在运行时解析，以确定实际的方法位置。对于访问类实例的所有其他 Java 虚拟机指令也是如此。

调用 addTwoStatic（addTwo 的类（静态）变体）类似，如下所示：

```
int add12and13() {
    return addTwoStatic(12, 13);
}
```

尽管使用了不同的 Java 虚拟机方法调用指令：

```
Method int add12and13()
0 bipush 12
2 bipush 13
4 invokestatic #3       // Method Example.addTwoStatic(II)I
7 ireturn
```

编译类(静态)方法的调用非常类似于编译实例方法的调用，只不过这不是由调用者传递的。方法参数将从局部变量 0 开始接收(§3.6)。invokestatic 指令总是用于调用类方法。

必须使用 invokespecial 指令调用实例初始化方法(§3.8)。在调用超类（super）中的方法时也使用它。例如，给定类 Near 和 Far 声明为：

```
class Near {
    int it;
    int getItNear() {
      return it;
    }
}
class Far extends Near {
    int getItFar() {
      return super.getItNear();
    }
}
```

方法 Far.getItFar (它调用了超类的方法) 编译成：

```
Method int getItFar()
0 aload_0
1 invokespecial #4      // Method Near.getItNear()I
4 ireturn
```

注意，使用 invokespecial 指令调用的方法总是将 this 作为第一个参数传递给被调用的方法。通常，它在局部变量 0 中被接收。

要调用方法句柄的目标，编译器必须形成记录实际参数和返回类型的方法描述符。编译器不能对参数执行方法调用转换；相反，它必须根据它们自己的未转换类型将它们推到栈上。

编译器通常会在参数之前安排对方法句柄对象的引用，并将其推送到栈上。编译器发出 invokevirtual 指令，该指令引用描述参数和返回类型的描述符。 通过方法解析(§5.4.3.3)的特殊安排，如果方法描述符语法正确且描述符中命名的类型可以被解析，则调用 invokeExact 方法或 java.lang.invoke.MethodHandle 的方法的 invokevirtual 指令将始终链接。

3.8 使用类实例

Java 虚拟机类实例是使用 Java 虚拟机的新指令创建的。回想一下，在 Java 虚拟机级，构造函数显示为具有编译器提供的名称<init>的方法。这种特别命名的方法称为实例初始化方法(§2.9)。给定类可能存在多个实例初始化方法，对应于多个构造函数。一旦创建了类实例，并且其实例变量（包括类及其所有超类的实例变量）已初始化为默认值，则调用新类实例的实例初始化方法。例如:

```
Object create() {
    return new Object();
}
```

编译为:

```
Method java.lang.Object create()
0   new #1              // Class java.lang.Object
3   dup
4   invokespecial #4    // Method java.lang.Object.<init>()V
7   areturn
```

类实例的传递和返回（作为 reference 类型）非常类似于数值，尽管类型 reference 有自己的指令补充，例如:

```
int i;                                  // An instance variable
MyObj example() {
    MyObj o = new MyObj();
    return silly(o);
}
MyObj silly(MyObj o) {
  if (o != null) {
     return o;
  } else {
     return o;
  }
}
```

编译为:

```
Method MyObj example()
0  new #2              // Class MyObj
3  dup
4  invokespecial #5    // Method MyObj.<init>()V
7  astore_1
8  aload_0
9  aload_1
10 invokevirtual #4    // Method Example.silly(LMyObj;)LMyObj
13 areturn
```

```
Method MyObj silly(MyObj)
0 aload_1
1 ifnull 6
4 aload_1
5 areturn
6 aload_1
7 areturn
```

使用 getfield 和 putfield 指令访问类实例的字段（实例变量）。如果 i 是 int 类型的实例变量，则方法 setIt 和 getIt 定义为：

```
void setIt(int value) {
  i = value;
}
int getIt() {
  return i;
}
```

编译为：

```
Method void setIt(int)
0   aload_0
1   iload_1
2   putfield #4    // Field Example.i I
5   return

Method int getIt()
0   aload_0
1   getfield #4    // Field Example.i I
4   ireturn
```


与方法调用指令的操作数一样，putfield 和 getfield 指令（运行时常量池索引#4）的操作数不是类实例中字段的偏移量。编译器生成对实例的字段的符号引用，这些引用存储在运行时常量池中。这些运行时常量池项在运行时被解析，以确定该字段在被引用对象中的位置。

3.9 数组

Java 虚拟机数组也是对象。数组是使用一组不同的指令创建和操作的。newarray 指令用于创建一个数值类型的数组。代码：

```
void createBuffer() {
  int buffer[];
  int bufsz = 100;
  int value = 12;
  buffer = new int[bufsz];
  buffer[10] = value;
  value = buffer[11];
}
```

可能被编译为：

```
Method void createBuffer()
0  bipush 100      // Push int constant 100 (bufsz)
2  istore_2        // Store bufsz in local variable 2
```

```
3  bipush 12        // Push int constant 12 (value)
5  istore_3         // Store value in local variable 3
6  iload_2          // Push bufsz...
7  newarray int     // ...and create new int array of that length
9  astore_1         // Store new array in buffer
10 aload_1          // Push buffer
11 bipush 10        // Push int constant 10
13 iload_3          // Push value
14 iastore          // Store value at buffer[10]
15 aload_1          // Push buffer
16 bipush 11        // Push int constant 11
18 iaload           // Push value at buffer[11]...
19 istore_3         // ...and store it in value
20 return
```

anewarray 指令用于创建一个一维的对象引用数组，例如:

```
void createThreadArray() {
    Thread threads[];
    int count = 10;
    threads = new Thread[count];
    threads[0] = new Thread();
}
```

编译为:

```
Method void createThreadArray()
  0  bipush 10          // Push int constant 10
  2  istore_2           // Initialize count to that
  3  iload_2            // Push count, used by anewarray
  4  anewarray class #1 // Create new array of class Thread
  7  astore_1           // Store new array in threads
  8  aload_1            // Push value of threads
  9  iconst_0           // Push int constant 0
  10 new #1             // Create instance of class Thread
  13 dup                // Make duplicate reference...
  14 invokespecial #5   // ...for Thread's constructor
                        // Method java.lang.Thread.<init>()V
  17 aastore            // Store new Thread in array at 0
  18 return
```

anewarray 指令也可以用来创建多维数组的第一个维度。另外，可以使用 multianewarray 指令一次创建多个维度。例如，三维数组:

```
int[][][] create3DArray() {
    int grid[][][];
    grid = new int[10][5][];
    return grid;
}
```

是由以下创建的:

```
Method int create3DArray()[][][]
0   bipush 10                // Push int 10 (dimension one)
2   iconst_5                 // Push int 5 (dimension two)
3   multianewarray #1 dim #2 // Class [[[I, a three-dimensional // int
                              array; only create the // first two dimensions
7   astore_1                 // Store new array...
```

```
8   aload_1                  // ...then prepare to return it
9   areturn
```

multianewarray 指令的第一个操作数是要创建的数组类类型的运行时常量池索引。第二个是实际创建的数组类型的维数。multianewarray 指令可用于创建类型的所有维度，如 create3DArray 的代码所示。注意，多维数组只是一个对象，因此分别由 aload_1 和 areturn 指令加载和返回。有关数组类名的信息，请参阅§4.4.1。

所有数组都有相关的长度，可通过 arraylength 指令访问。

3.10 编译 Switches

switch 语句的编译使用 tableswitch 和 lookupswitch 指令。当 switch 的情况可以有效地表示为目标偏移量表的索引时，使用 tableswitch 指令。如果 switch 表达式的值超出有效索引范围，则使用 switch 的默认目标。例如:

```
int chooseNear(int i) {
  switch (i) {
        case 0:  return  0;
        case 1:  return  1;
        case 2:  return  2;
        default: return -1;
  }
}
```

编译为:

```
Method int chooseNear(int)
0 iload_1               // Push local variable 1 (argument i)
1 tableswitch 0 to 2:   // Valid indices are 0 through 2
     0:28               // If i is 0, continue at 28
     1:30               // If i is 1, continue at 30
     2:32               // If i is  2,  continue    at  32
     default:34         // Otherwise,  continue    at  34
28 iconst_0             // i was 0; push int constant 0...
29 ireturn              //...and return it
30 iconst_1             // i was 1; push int constant 1...
31 ireturn              //...and return it
32 iconst_2             // i was 2; push int constant 2...
33 ireturn              //...and return it
34 iconst_m1            // otherwise push int constant -1...
35 ireturn              //...and return it
```

Java 虚拟机的 tableswitch 和 lookupswitch 指令仅对 int 数据进行操作。由于对 byte、char 或 short 值的操作在内部升级为 int，因此表达式计算为这些类型之一的 switch 将被编译为 int 类型。如果 chooseNear 方法是使用 short 类型编写的，则生成的 Java 虚拟机指令与使用 int 类型时相同。其他数字类型必须收窄为 int 类型，以便在 switch 中使用。

在 switch 稀疏的情况下，tableswitch 指令的表表示在空间方面变得低效。也可以使用 lookupswitch 指令。lookupswitch 指令将 int 键（case 标签的值）与表中的目标偏移量配

对。执行 lookupswitch 指令时，将 switch 表达式的值与表中的键进行比较。如果其中一个键与表达式的值匹配，则在关联的目标偏移处继续执行。如果没有匹配的键，则在默认目标上继续执行。例如，编译的代码：

```
int chooseFar(int i) {
    switch (i) {
        case  -100: return -1;
        case  0:    return  0;
        case  100:  return  1;
        default:    return -1;
    }
}
```

除了 lookupswitch 指令外，与 chooseNear 的代码类似：

```
Method int chooseFar(int)
0 iload_1
1   lookupswitch 3:
          -100: 36
             0: 38
             100: 40
             default: 42
36 iconst_m1
37  ireturn
38  iconst_0
39  ireturn
40  iconst_1
41  ireturn
42  iconst_m1
43  ireturn
```


Java 虚拟机指定 lookupswitch 指令的表必须按键排序，以便实现可以使用比线性扫描更高效的搜索。即使如此，lookupswitch 指令也必须搜索其键以查找匹配项，而不是简单地执行边界检查并索引到类似 tableswitch 的表中。因此，在空间考虑允许选择的情况下，tableswitch 指令可能比 lookupswitch 指令更有效。

3.11 操作数栈上的操作

Java 虚拟机拥有大量指令，这些指令将操作数栈的内容作为非类型化值进行操作。这些是有用的，因为 Java 虚拟机依赖于对其操作数栈的灵活操作。例如：

```
public long nextIndex() {
  return index++;
}

private long index = 0;
```

编译为：

```
Method long nextIndex()
0  aload_0          // Push this
1  dup              // Make a copy of it
2  getfield #4      // One of the copies of this is consumed
```

```
                        // pushing long field index,
                        // above the original this
      5  dup2_x1         // The long on top of the operand stack is
                        // inserted into the operand stack below the
                        // original this
      6  lconst 1        // Push long constant 1
      7  ladd            // The index value is incremented...
      8  putfield #4     // ...and the result stored in the field
      11 lreturn         // The original value of index is on top of
                        // the operand stack, ready to be returned
```

请注意，Java 虚拟机从不允许其操作数栈操作指令修改或拆分操作数栈上的单个值。

3.12 抛出和处理异常

使用 throw 关键字从程序抛出异常。它的编译很简单:

```
void cantBeZero(int i) throws TestExc {
    if (i == 0) {
        throw new TestExc();
    }
}
```

编译为:

```
Method void cantBeZero(int)
0   iload_1             //  Push argument 1 (i)
1   ifne 12             //  If i==0, allocate instanceand throw
4   new #1              //  Create instance of TestExc
7   dup                 //  One reference goes to its constructor
8   invokespecial #7    // Method TestExc.<init>()V
11  athrow              //  Second reference is thrown
12  return              //  Never get here if we threwTestExc
```

try-catch 构造的编译非常简单。例如:

```
void catchOne() {
  try {
    tryItOut();
  }
  catch (TestExc e) {
    handleExc(e);
  }
}
```

编译为:

```
Method void catchOne()
0   aload_0             // Beginning of try block
1   invokevirtual #6    // Method Example.tryItOut()V
4   return              // End of try block; normal return
5   astore_1            // Store thrown value in local var 1
6   aload_0             // Push this
7   aload_1             // Push thrown value
8   invokevirtual #5    // Invoke handler method:
                        // Example.handleExc(LTestExc;)V
11  return              // Return after handling TestExc
Exception table:
From    To      Target      Type
0       4       5           Class TestExc
```

仔细观察，try 块的编译方式与 try 不存在时的编译方式相同：

```
Method void catchOne()
0   aload_0             //  Beginning of try block
1   invokevirtual  #6   //  Method Example.tryItOut()V
4   return              //  End of try block; normal return
```

如果在执行 try 块的过程中没有抛出异常，那么它的行为就好像 try 不存在一样：调用 tryItOut 并返回 catchOne。

try 块后面是实现单 catch 子句的 Java 虚拟机代码：

```
5  astore_1         //Store thrown value in local var 1
6  aload_0          //Push this
7  aload_1          //Push thrown value
8  invokevirtual #5 //Invoke handler method:
                    //Example.handleExc(LTestExc;)V
11 return           //Return after handling TestExc
Exception table:
From   To   Target   Type
0      4    5        Class TestExc
```

handleExc 的调用（catch 子句的内容）也像普通方法调用一样进行编译。但是，catch 子句的存在会导致编译器生成异常表条目（§2.10，§4.7.3）。catchOne 方法的异常表有一个条目对应于 catchOne 的 catch 子句可以处理的一个参数（类 TestExc 的实例）。如果在执行 catchOne 中索引 0 和 4 之间的指令期间抛出了 TestExc 实例的某个值，则控制权将转移到索引 5 处的 Java 虚拟机代码，该代码实现了 catch 子句的块。如果抛出的值不是 TestExc 的实例，catchOne 的 catch 子句将无法处理它。相反，该值被重新抛出给 catchOne 的调用者。

一个 try 可以有多个 catch 子句：

```
void catchTwo() {
    try {
        tryItOut();
    } catch (TestExc1 e) {
        handleExc(e);
    } catch (TestExc2 e) {
        handleExc(e);
    }
}
```

编译给定 try 语句的多个 catch 子句时，只需将每个 catch 子句的 Java 虚拟机代码一个接一个地添加到异常表中，如下所示:

```
Method void catchTwo()
  Method void catchTwo()
  0   aload_0              // Begin try block
  1   invokevirtual #5     // Method Example.tryItOut()V
  4   return               // End of try block; normal return
  5   astore_1             // Beginning of handler for TestExc1;
                           // Store thrown value in local var 1
  6   aload_0              // Push this
  7   aload_1              // Push thrown value
  8   invokevirtual #7     // Invoke handler method:
                           // Example.handleExc(LTestExc1;)V
  11  return               // Return after handling TestExc1
  12  astore_1             // Beginning of handler for TestExc2;
                           // Store thrown value in local var 1
  13  aload_0              // Push this

  14  aload_1              // Push thrown value
  15  invokevirtual #7     // Invoke handler method:
                           // Example.handleExc(LTestExc2;)V
  18  return               // Return after handling TestExc2
  Exception table:
  From    To      Target      Type
  0       4       5           Class TestExc1
  0       4       12          Class TestExc2
```

如果在 try 子句执行期间(索引 0 到 4 之间)抛出一个与一个或多个 catch 子句的参数匹配的值(该值是一个或多个参数的实例)，则选择第一个(最内层的)这样的 catch 子句。将该 catch 子句块的控制转移到 Java 虚拟机代码。如果抛出的值与 catchTwo 的任何 catch 子句的参数不匹配，Java 虚拟机将重新抛出该值，而不调用 catchTwo 的任何 catch 子句中的代码。

嵌套 try-catch 语句编译起来非常像带有多个 catch 子句的 try 语句:

```
void nestedCatch() {
  try {
    try {
      tryItOut();
    }
    catch (TestExc1 e) {
      handleExc1(e);
    }
  } catch (TestExc2 e) {
    handleExc2(e);
  }
}
```

编译为:

```
Method void nestedCatch()
0   aload_0             // Begin try block
1   invokevirtual #8    // Method Example.tryItOut()V
4   return              // End of try block; normal return
5   astore_1            // Beginning of handler for TestExc1;
                        // Store thrown value in local var 1
6   aload_0             // Push this
7   aload_1             // Push thrown value
8   invokevirtual #7    // Invoke handler method:
                        // Example.handleExc1(LTestExc1;)V
11  return              // Return after handling TestExc1
12  astore_1            // Beginning of handler for TestExc2;
                        // Store thrown value in local var 1
13  aload_0             // Push this
14  aload_1             // Push thrown value
15  invokevirtual #6    // Invoke handler method:
                        // Example.handleExc2(LTestExc2;)V

18  return              // Return after handling TestExc2
Exception table:
From    To      Target      Type
0       4       5           Class TestExc1
0       12      12          Class TestExc2
```

catch 子句的嵌套只在异常表中表示。Java 虚拟机不强制异常表项的嵌套或排序(§2.10)。然而，因为 try-catch 构造是结构化的，所以编译器总是可以对异常处理程序表的条目进行排序，这样，对于抛出的任何异常和该方法中的任何程序计数器值，第一个匹配抛出异常的异常处理程序对应于最内层匹配的 catch 子句。

例如，如果对 tryItOut(索引 1)的调用抛出了 TestExc1 的实例，那么它将由调用 handleExc1 的 catch 子句处理。即使异常发生在外层 catch 子句(捕获 TestExc2)的作用域内，即使该外层 catch 子句本来可能能够处理抛出的值，情况也是如此。

需要注意的是，catch 子句的作用域在“from”端是包含的，在“to”端是排他的(§4.7.3)。因此，捕获 TestExc1 的 catch 子句的异常表条目不包括在偏移量 4 处的返回指令。然而，catch 子句捕获 TestExc2 的异常表条目确实覆盖了偏移量 11 处的返回指令。嵌套 catch 子句中的返回指令包含在嵌套 catch 子句覆盖的指令作用域内。

3.13 编译 `finally`

(本节假设编译器生成版本号为 50.0 或更低的 class 文件，以便使用 jsr 指令。参见§4.10.2.5)。

try-finally 语句的编译与 try-catch 语句的编译类似。在将控制转移到 try 语句外之前，无论转移是正常的还是突然的，因为抛出了异常，都必须首先执行 finally 子句。对于这个简单的例子:

```
void tryFinally() {
    try {
        tryItOut();
    } finally {
        wrapItUp();
    }
```

```
}
```

编译后的代码是:

```
Method void tryFinally()
0   aload_0             // Beginning of try block
1   invokevirtual #6    // Method Example.tryItOut()V
4   jsr 14              // Call finally block
7   return              // End of try block
8   astore_1            // Beginning of handler for any throw
9   jsr 14              // Call finally block
12  aload_1             // Push thrown value
13  athrow              // ...and rethrow value to the invoker
14  astore_2            // Beginning of finally block
15  aload_0             // Push this
16  invokevirtual #5    // Method Example.wrapItUp()V
19  ret 2               // Return from finally block
Exception table:
From    To      Target      Type
0       4       8           any
```

有四种方法可以将控制传递到 try 语句之外:通过掉入语句块的底部，通过返回，通过执行 break 或 continue 语句，或通过引发异常。如果 tryItOut 返回而不引发异常，则使用 jsr 指令将控制转移到 finally 块。索引 4 处的 jsr 14 指令对索引 14 处的 finally 块的代码进行“子例程调用”(finally 块被编译为嵌入的子例程)。当 finally 块完成时，ret 2 指令将控制权返回给索引 4 处的 jsr 指令之后的指令。

更详细地说，子例程调用的工作方式如下:jsr 指令在跳转之前将以下指令(在索引 7 处返回)的地址压入操作数栈。作为跳转目标的 astore_2 指令将操作数栈上的地址存储到局部变量 2 中。运行 finally 块的代码(在本例中是 aload_0 和 invokvirtual 指令)。假设该代码的执行正常完成，ret 指令从局部变量 2 检索地址，并在该地址继续执行。执行返回指令，tryFinally 正常返回。

带有 finally 子句的 try 语句被编译为具有一个特殊的异常处理程序，该异常处理程序可以处理 try 语句中抛出的任何异常。如果 tryItOut 抛出异常，则在 tryFinally 的异常表中搜索适当的异常处理程序。找到了特殊的处理程序，导致在索引 8 处继续执行。索引 8 处的 astore_1 指令将抛出的值存储到局部变量 1 中。下面的 jsr 指令对 finally 块的代码执行子例程调用。 假设代码正常返回，索引 12 处的 aload_1 指令将抛出的值推回操作数栈，随后的 athrow 指令重新抛出该值。

编译带有 catch 子句和 finally 子句的 try 语句更复杂:

```
void tryCatchFinally() {
  try {
    tryItOut();
  } catch (TestExc e) {
    handleExc(e);
  } finally {
    wrapItUp();
  }
}
```

编译为：

```
Method void tryCatchFinally()
0   aload_0             //  Beginning of try block
1   invokevirtual #4    //  Method Example.tryItOut()V
4   goto 16             //  Jump to finally block
7   astore_3            //  Beginning of handler for TestExc;
                        //  Store thrown value in local var 3
8   aload_0             //  Push this
9   aload_3             //  Push thrown value
10  invokevirtual #6    //  Invoke handler method:
                        //  Example.handleExc(LTestExc;)V
13  goto 16             //  This goto is unnecessary, but was
                        //  generated by javac in JDK 1.0.2
16  jsr 26              //  Call finally block
19  return              //  Return after handling TestExc
20  astore_1            //  Beginning of handler for exceptions
                        //  other than TestExc, or exceptions
                        //  thrown while handling TestExc
21  jsr 26              //  Call finally block
24  aload 1             //  Push thrown value...
25  athrow              //  ...and rethrow value to the invoker
26  astore_2            //  Beginning of finally block
27  aload_0             //  Push this
28  invokevirtual #5    //  Method Example.wrapItUp()V
31 ret 2                //  Return from finally block
Exception table:
From    To      Target          Type
0       4       7               Class TestExc any
0       16      20
```

如果 try 语句正常完成，则索引 4 处的 goto 指令跳转到索引 16 处的 finally 块的子程序调用。执行索引 26 处的 finally 块，控制返回到索引 19 处的返回指令，tryCatchFinally 正常返回。

如果 tryItOut 抛出 TestExc 的实例，则选择异常表中第一个(最内层)适用的异常处理程序来处理异常。该异常处理程序的代码从索引 7 开始，将抛出的值传递给 handleExc，并在其返回时对索引 26 处的 finally 块进行与正常情况相同的子例程调用。如果 handleExc 没有抛出异常，tryCatchFinally 会正常返回。

如果 tryItOut 抛出一个不是 TestExc 实例的值，或者如果 handleExc 本身抛出异常，则异常表中的第二个条目将处理该条件，该条目将处理索引 0 到 16 之间抛出的任何值。该异常处理程序将控制转移到索引 20，其中抛出的值首先存储在局部变量 1 中。索引 26 处的 finally 块的代码被作为子例程调用。如果它返回，则从局部变量 1 检索所抛出的值，并使用 athrow 指令重新抛出。如果在 finally 子句执行期间抛出一个新值，finally 子句会中止，tryCatchFinally 会突然返回，将新值抛给它的调用者。

3.14 同步

Java 虚拟机中的同步是通过监视器进入和退出实现的，可以显式地（通过使用

monitorenter 和 monitorexit 指令）或隐式地（使用方法调用和返回指令）。

对于用 Java 编程语言编写的代码，最常见的同步形式可能是同步方法。同步方法通常不会使用 monitorenter 和 monitorexit 实现。相反，它只是在运行时常量池中通过 ACC_SYNCHRONIZED 标志来区分，该标志由方法调用指令来检查(§2.11.10)。

monitorenter 和 monitorexit 指令允许编译同步语句。例如:

```
void onlyMe(Foo f) {
    synchronized(f) {
      doSomething();
    }
}
```

编译为:

```
Method void onlyMe(Foo)
0   aload_1             // Push f
1   dup                 // Duplicate it on the stack
2   astore_2            // Store duplicate in local variable 2
3   monitorenter        // Enter the monitor associated with f
4   aload_0             // Holding the monitor, pass this and...

5   invokevirtual #5    // ...call Example.doSomething()V
8   aload_2             // Push local variable 2 (f)
9   monitorexit         // Exit the monitor associated with f
10  goto 18             // Complete the method normally
13  astore_3            // In case of any throw, end up here
14  aload_2             // Push local variable 2 (f)
15  monitorexit         // Be sure to exit the monitor!
16  aload_3             // Push thrown value...
17  athrow              // ...and rethrow value to the invoker
18  return              // Return in the normal case
Exception table:
From    To      Target      Type
4       10      13          any
13      16      13          any
```

编译器确保在任何方法调用完成时，都会为自方法调用以来执行的每个 monitorenter 指令执行一个 monitorexit 指令。无论方法调用是正常完成(§2.6.4)还是突然完成(§2.6.5)，情况都是如此。为了在突然的方法调用完成时强制 monitorenter 和 monitorexit 指令的正确配对，编译器生成异常处理程序(§2.10)，该异常处理程序将匹配任何异常，其关联的代码将执行必要的 monitorexit 指令。

3.15 注解

§4.7.16-§4.7.22 描述了 class 文件中注解的表示。这些部分明确了如何表示对类、接口、字段、方法、方法参数和类型参数声明的注解，以及对这些声明中使用的类型的注解。 包声明的注解需要额外的规则，这里给出了这些规则。

当编译器遇到必须在运行时可用注解的包声明时，它会发出一个具有以下属性的 class 文件:

- class文件代表一个接口，也就是说，ClassFile结构的ACC_INTERFACE和ACC_ABSTRACT标志被设置(§4.1)。
- 如果 class 文件版本号小于 50.0，则不设置 ACC_SYNTHETIC 标志;如果 class 文件版本号为 50.0 或更高，则设置 ACC_SYNTHETIC 标志。
- 接口具有包访问权限(JLS§6.6.1)。
- 接口的名称是 package-name.package- info 的内部形式(§4.2.1)。
- 接口没有超接口。
- 该接口的唯一成员是 Java 语言规范，Java SE 19 版本(JLS§9.2)中隐含的成员。
- 包声明上的注解存储在 ClassFile 结构的属性表中的 RuntimeVisibleAnnotations 和 RuntimeInvisibleAnnotations 属性。

3.16 模块

一个包含模块声明(JLS§7.7)的编译单元被编译成一个包含 Module 属性的 class 文件。

按照惯例，包含一个模块声明的编译单元的名称是 module-info.java，与只包含一个包声明的编译单元的 package-info.java 的惯例相呼应。因此，按照惯例，模块声明的编译形式的名称为 module-info.class。

ClassFile 结构的 access_flags 项中的标志 ACC_MODULE (0x8000)表示这个 class 文件声明了一个模块。ACC_MODULE 扮演着类似 ACC_ANNOTATION (0x2000)和 ACC_ENUM (0x4000)的角色，将这个 class 文件标记为“不是一个普通类”。 ACC_MODULE 不描述类或接口的可访问性。

Module 属性对模块的依赖是显式的;在 ClassFile 级别没有隐含的 requires 指令。如果 requires_count 项为零，那么 Java SE 平台不会推断出 requires 表或其中任何特定条目的存在。java.base 是唯一一个零 requires_count 合法的模块，因为它是原始模块。对于每个其他模块，Module 属性必须具有长度至少为 1 的 requires 表，因为每个其他模块都依赖于 java.base。如果一个编译单元包含一个模块声明（java.base 除外），它并没有声明其对 java.base 的依赖性,则编译器必须在 requires 表中为 java.base 发出一个条目。并将其标记为 ACC_MANDATED，以表示它是隐式声明的。

对于封装，Module 属性对由普通模块导出和打开的包是显式的;普通模块在 ClassFile 级别没有隐含的 exports 或 opens 指令。如果 exports_count 项或 opens_count 项为零, 那么 Java SE 平台就不会推断 exports 表或 opens 表的存在，也不会推断其中的任何特定条目的存在。另一方面，对于一个打开的模块，Module 属性对于该模块打开的包是隐式的。一个打开模块的所有包都会对所有其他模块打开，即使 opens_count 项为零。

Module 属性是关于模块的消费和服务提供的显式属性；在 ClassFile 文件级别没有隐式 uses

或 provides 指令。