第1讲 | Java代码是怎么运行的?

2018-07-20 郑雨迪

第1讲 | Java代码是怎么运行的?
朗读人：郑雨迪　10′36″ | 6.09M

我们学院的一位教授之前去美国开会，入境的时候海关官员就问他：既然你会计算机，那你说说你用的都是什么语言吧?

教授随口就答了个 Java。海关一看是懂行的，也就放行了，边敲章还边说他们上学那会学的是 C+。我还特意去查了下，真有叫 C+ 的语言，但是这里海关官员应该指的是 C++。

事后教授告诉我们，他当时差点就问海关，是否知道 Java 和 C++ 在运行方式上的区别。但是又担心海关官员拿他的问题来考别人，也就没问出口。那么，下次你去美国，不幸地被海关官员问这个问题，你懂得如何回答吗?

作为一名 Java 程序员，你应该知道，Java 代码有很多种不同的运行方式。比如说可以在开发工具中运行，可以双击执行 jar 文件运行，也可以在命令行中运行，甚至可以在网页中运行。当然，这些执行方式都离不开 JRE，也就是 Java 运行时环境。

实际上，JRE 仅包含运行 Java 程序的必需组件，包括 Java 虚拟机以及 Java 核心类库等。我们 Java 程序员经常接触到的 JDK（Java 开发工具包）同样包含了 JRE，并且还附带了一系列开发、诊断工具。

然而，运行 C++ 代码则无需额外的运行时。我们往往把这些代码直接编译成 CPU 所能理解的代码格式，也就是机器码。

比如下图的中间列，就是用 C 语言写的 Helloworld 程序的编译结果。可以看到，C 程序编译而成的机器码就是一个个的字节，它们是给机器读的。那么为了让开发人员也能够理解，我们可以用反汇编器将其转换成汇编代码（如下图的最右列所示）。

```
; 最左列是偏移；中间列是给机器读的机器码；最右列是给人读的汇编代码
0x00:  55                   push   rbp
0x01:  48 89 e5             mov    rbp,rsp
0x04:  48 83 ec 10          sub    rsp,0x10
0x08:  48 8d 3d 3b 00 00 00 lea    rdi,[rip+0x3b]
                                   ; 加载 "Hello, World!\n"
0x0f:  c7 45 fc 00 00 00 00 mov    DWORD PTR [rbp-0x4],0x0
0x16:  b0 00                mov    al,0x0

0x18:  e8 0d 00 00 00       call   0x12
                                   ; 调用 printf 方法
0x1d:  31 c9                xor    ecx,ecx
0x1f:  89 45 f8             mov    DWORD PTR [rbp-0x8],eax
0x22:  89 c8                mov    eax,ecx
0x24:  48 83 c4 10          add    rsp,0x10
0x28:  5d                   pop    rbp
0x29:  c3                   ret
```

既然 C++ 的运行方式如此成熟，那么你有没有想过，为什么 Java 要在虚拟机中运行呢，Java 虚拟机具体又是怎样运行 Java 代码的呢，它的运行效率又如何呢？

今天我便从这几个问题入手，和你探讨一下，Java 执行系统的主流实现以及设计决策。

为什么 Java 要在虚拟机里运行？

Java 作为一门高级程序语言，它的语法非常复杂，抽象程度也很高。因此，直接在硬件上运行这种复杂的程序并不现实。所以呢，在运行 Java 程序之前，我们需要对其进行一番转换。

这个转换具体是怎么操作的呢？当前的主流思路是这样子的，设计一个面向 Java 语言特性的虚拟机，并通过编译器将 Java 程序转换成该虚拟机所能识别的指令序列，也称 Java 字节码。这里顺便说一句，之所以这么取名，是因为 Java 字节码指令的操作码（opcode）被固定为一个字节。

举例来说，下图的中间列，正是用 Java 写的 Helloworld 程序编译而成的字节码。可以看到，它与 C 版本的编译结果一样，都是由一个个字节组成的。

并且，我们同样可以将其反汇编为人类可读的代码格式（如下图的最右列所示）。不同的是，Java 版本的编译结果相对精简一些。这是因为 Java 虚拟机相对于物理机而言，抽象程度更高。

```
# 最左列是偏移；中间列是给虚拟机读的机器码；最右列是给人读的代码
0x00:  b2 00 02         getstatic java.lang.System.out
0x03:  12 03            ldc "Hello, World!"
0x05:  b6 00 04         invokevirtual java.io.PrintStream.println
0x08:  b1               return
```

Java 虚拟机可以由硬件实现 [1]，但更为常见的是在各个现有平台（如 Windows_x64、Linux_aarch64）上提供软件实现。这么做的意义在于，一旦一个程序被转换成 Java 字节码，那么它便可以在不同平台上的虚拟机实现里运行。这也就是我们经常说的“一次编写，到处运行”。

虚拟机的另外一个好处是它带来了一个托管环境（Managed Runtime）。这个托管环境能够代替我们处理一些代码中冗长而且容易出错的部分。其中最广为人知的当属自动内存管理与垃圾回收，这部分内容甚至催生了一波垃圾回收调优的业务。

除此之外，托管环境还提供了诸如数组越界、动态类型、安全权限等等的动态检测，使我们免于书写这些无关业务逻辑的代码。

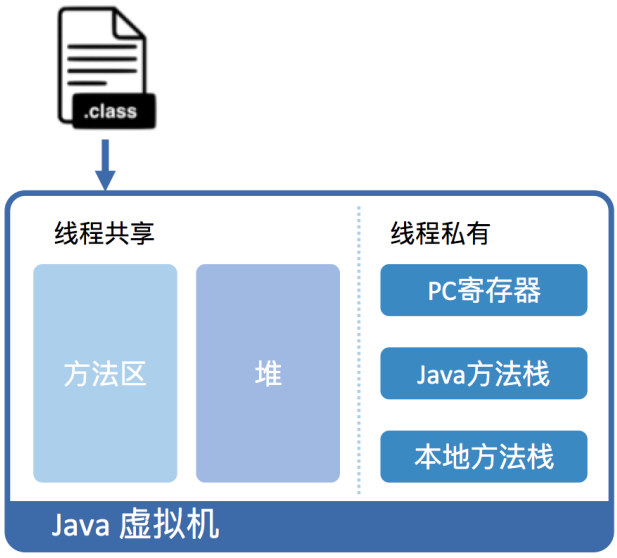
Java 虚拟机具体是怎样运行 Java 字节码的？

下面我将以标准 JDK 中的 HotSpot 虚拟机为例，从虚拟机以及底层硬件两个角度，给你讲一讲 Java 虚拟机具体是怎么运行 Java 字节码的。

从虚拟机视角来看，执行 Java 代码首先需要将它编译而成的 class 文件加载到 Java 虚拟机中。加载后的 Java 类会被存放于方法区（Method Area）中。实际运行时，虚拟机会执行方法区内的代码。

如果你熟悉 X86 的话，你会发现这和段式内存管理中的代码段类似。而且，Java 虚拟机同样也在内存中划分出堆和栈来存储运行时数据。

不同的是，Java 虚拟机会将栈细分为面向 Java 方法的 Java 方法栈，面向本地方法（用 C++ 写的 native 方法）的本地方法栈，以及存放各个线程执行位置的 PC 寄存器。

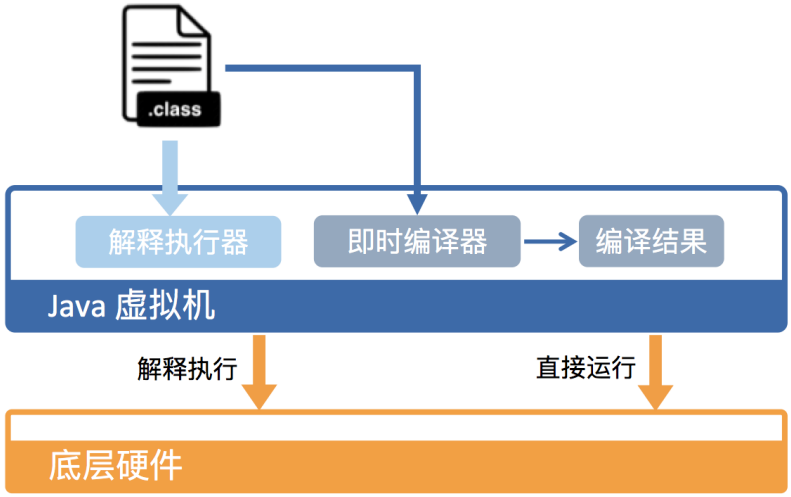


在运行过程中，每当调用进入一个 Java 方法，Java 虚拟机会在当前线程的 Java 方法栈中生成一个栈帧，用以存放局部变量以及字节码的操作数。这个栈帧的大小是提前计算好的，而且 Java 虚拟机不要求栈帧在内存空间里连续分布。

当退出当前执行的方法时，不管是正常返回还是异常返回，Java 虚拟机均会弹出当前线程的当前栈帧，并将之舍弃。

从硬件视角来看，Java 字节码无法直接执行。因此，Java 虚拟机需要将字节码翻译成机器码。

在 HotSpot 里面，上述翻译过程有两种形式：第一种是解释执行，即逐条将字节码翻译成机器码并执行；第二种是即时编译（Just-In-Time compilation，JIT），即将一个方法中包含的所有字节码编译成机器码后再执行。



前者的优势在于无需等待编译，而后者的优势在于实际运行速度更快。HotSpot 默认采用混合模式，综合了解释执行和即时编译两者的优点。它会先解释执行字节码，而后将其中反复执行的热点代码，以方法为单位进行即时编译。

Java 虚拟机的运行效率究竟是怎么样的？

HotSpot 采用了多种技术来提升启动性能以及峰值性能，刚刚提到的即时编译便是其中最重要的技术之一。

即时编译建立在程序符合二八定律的假设上，也就是百分之二十的代码占据了百分之八十的计算资源。

对于占据大部分的不常用的代码，我们无需耗费时间将其编译成机器码，而是采取解释执行的方式运行；另一方面，对于仅占据小部分的热点代码，我们则可以将其编译成机器码，以达到理想的运行速度。

理论上讲，即时编译后的 Java 程序的执行效率，是可能超过 C++ 程序的。这是因为与静态编译相比，即时编译拥有程序的运行时信息，并且能够根据这个信息做出相应的优化。

举个例子，我们知道虚方法是用来实现面向对象语言多态性的。对于一个虚方法调用，尽管它有很多个目标方法，但在实际运行过程中它可能只调用其中的一个。

这个信息便可以被即时编译器所利用，来规避虚方法调用的开销，从而达到比静态编译的 C++ 程序更高的性能。

为了满足不同用户场景的需要，HotSpot 内置了多个即时编译器：C1、C2 和 Graal。Graal 是 Java 10 正式引入的实验性即时编译器，在专栏的第四部分我会详细介绍，这里暂不做讨论。

之所以引入多个即时编译器，是为了在编译时间和生成代码的执行效率之间进行取舍。C1 又叫做 Client 编译器，面向的是对启动性能有要求的客户端 GUI 程序，采用的优化手段相对简单，因此编译时间较短。

C2 又叫做 Server 编译器，面向的是对峰值性能有要求的服务器端程序，采用的优化手段相对复杂，因此编译时间较长，但同时生成代码的执行效率较高。

从 Java 7 开始，HotSpot 默认采用分层编译的方式：热点方法首先会被 C1 编译，而后热点方法中的热点会进一步被 C2 编译。

为了不干扰应用的正常运行，HotSpot 的即时编译是放在额外的编译线程中进行的。HotSpot 会根据 CPU 的数量设置编译线程的数目，并且按 1:2 的比例配置给 C1 及 C2 编译器。

在计算资源充足的情况下，字节码的解释执行和即时编译可同时进行。编译完成后的机器码会在下次调用该方法时启用，以替换原本的解释执行。

总结与实践

今天我简单介绍了 Java 代码为何在虚拟机中运行，以及如何在虚拟机中运行。

之所以要在虚拟机中运行，是因为它提供了可移植性。一旦 Java 代码被编译为 Java 字节码，便可以在不同平台上的 Java 虚拟机实现上运行。此外，虚拟机还提供了一个代码托管的环境，代替我们处理部分冗长而且容易出错的事务，例如内存管理。

Java 虚拟机将运行时内存区域划分为五个部分，分别为方法区、堆、PC 寄存器、Java 方法栈和本地方法栈。Java 程序编译而成的 class 文件，需要先加载至方法区中，方能在 Java 虚拟机中运行。

为了提高运行效率，标准 JDK 中的 HotSpot 虚拟机采用的是一种混合执行的策略。

它会解释执行 Java 字节码，然后会将其中反复执行的热点代码，以方法为单位进行即时编译，翻译成机器码后直接运行在底层硬件之上。

HotSpot 装载了多个不同的即时编译器，以便在编译时间和生成代码的执行效率之间做取舍。

下面我给你留一个小作业，通过观察两个条件判断语句的运行结果，来思考 Java 语言和 Java 虚拟机看待 boolean 类型的方式是否不同。

下载 asmtools.jar [2]，并在命令行中运行下述指令（不包含提示符 $）：

```
$ echo
```

```
$ echo '
public class Foo {
 public static void main(String[] args) {
  boolean flag = true;
  if (flag) System.out.println("Hello, Java!");
  if (flag == true) System.out.println("Hello, JVM!");
 }
}' > Foo.java
$ javac Foo.java
$ java Foo
$ java -cp /path/to/asmtools.jar org.openjdk.asmtools.jdis.Main Foo.class > Foo.jasm.1
$ awk 'NR==1,/iconst_1/{sub(/iconst_1/, "iconst_2")} 1' Foo.jasm.1 > Foo.jasm
$ java -cp /path/to/asmtools.jar org.openjdk.asmtools.jasm.Main Foo.jasm
$ java Foo
```

[1] : https://en.wikipedia.org/wiki/Java_processor

[2]: https://wiki.openjdk.java.net/display/CodeTools/asmtools

通过留言可与作者互动