16 即时编译（上）

在专栏的第一篇中，我曾经简单地介绍过即时编译。这是一项用来提升应用程序运行效率的技术。通常而言，代码会先被 Java 虚拟机解释执行，之后反复执行的热点代码则会被即时编译成为机器码，直接运行在底层硬件之上。

今天我们便来详细剖析一下 Java 虚拟机中的即时编译。

分层编译模式

HotSpot 虚拟机包含多个即时编译器 C1、C2 和 Graal。

其中，Graal 是一个实验性质的即时编译器，可以通过参数 -XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler 启用，并且替换 C2。

在 Java 7 以前，我们需要根据程序的特性选择对应的即时编译器。对于执行时间较短的，或者对启动性能有要求的程序，我们采用编译效率较快的 C1，对应参数 -client。

对于执行时间较长的，或者对峰值性能有要求的程序，我们采用生成代码执行效率较快的 C2，对应参数 -server。

Java 7 引入了分层编译（对应参数 -XX:+TieredCompilation）的概念，综合了 C1 的启动性能优势和 C2 的峰值性能优势。

分层编译将 Java 虚拟机的执行状态分为了五个层次。为了方便阐述，我用“C1 代码”来指代由 C1 生成的机器码，“C2 代码”来指代由 C2 生成的机器码。五个层级分别是：

1. 解释执行；
2. 执行不带 profiling 的 C1 代码；
3. 执行仅带方法调用次数以及循环回边执行次数 profiling 的 C1 代码；
4. 执行带所有 profiling 的 C1 代码；
5. 执行 C2 代码。

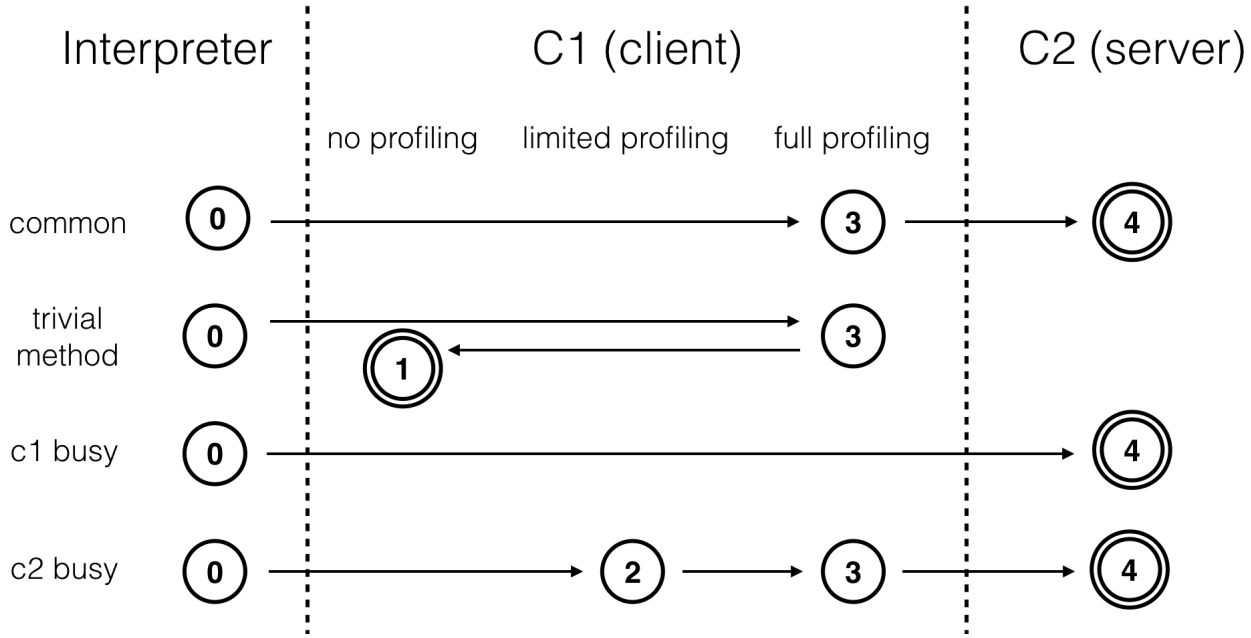
通常情况下，C2 代码的执行效率要比 C1 代码的高出 30% 以上。然而，对于 C1 代码的三种状态，按执行效率从高至低则是 1 层 > 2 层 > 3 层。

其中 1 层的性能比 2 层的稍微高一些，而 2 层的性能又比 3 层高出 30%。这是因为 profiling 越多，其额外的性能开销越大。

这里解释一下，profiling 是指在程序执行过程中，收集能够反映程序执行状态的数据。这里所收集的数据我们称之为程序的 profile。

你可能已经接触过许许多多的 profiler，例如 JDK 附带的 hprof。这些 profiler 大多通过注入（instrumentation）或者 JVMTI 事件来实现的。Java 虚拟机也内置了 profiling。我会在下一篇中具体介绍 Java 虚拟机的 profiling 都在做些什么。

在 5 个层次的执行状态中，1 层和 4 层为终止状态。当一个方法被终止状态编译过后，如果编译后的代码并没有失效，那么 Java 虚拟机是不会再次发出该方法的编译请求的。



不同的编译路径，图片来源于我之前一篇介绍 Graal 的博客。

这里我列举了 4 个不同的编译路径（Igor 的演讲列举了更多的编译路径）。通常情况下，热点方法会被 3 层的 C1 编译，然后再被 4 层的 C2 编译。

如果方法的字节码数目比较少（如 getter/setter），而且 3 层的 profiling 没有可收集的数据。

那么，Java 虚拟机断定该方法对于 C1 代码和 C2 代码的执行效率相同。在这种情况下，Java 虚拟机会在 3 层编译之后，直接选择用 1 层的 C1 编译。由于这是一个终止状态，因此 Java 虚拟机不会继续用 4 层的 C2 编译。

在 C1 忙碌的情况下，Java 虚拟机在解释执行过程中对程序进行 profiling，而后直接由 4 层

的 C2 编译。在 C2 忙碌的情况下，方法会被 2 层的 C1 编译，然后再被 3 层的 C1 编译，以减少方法在 3 层的执行时间。

Java 8 默认开启了分层编译。不管是开启还是关闭分层编译，原本用来选择即时编译器的参数 -client 和 -server 都是无效的。当关闭分层编译的情况下，Java 虚拟机将直接采用 C2。

如果你希望只是用 C1，那么你可以在打开分层编译的情况下使用参数 -XX:TieredStopAtLevel=1。在这种情况下，Java 虚拟机会在解释执行之后直接由 1 层的 C1 进行编译。

即时编译的触发

Java 虚拟机是根据方法的调用次数以及循环回边的执行次数来触发即时编译的。前面提到，Java 虚拟机在 0 层、2 层和 3 层执行状态时进行 profiling，其中就包含方法的调用次数和循环回边的执行次数。

这里的循环回边是一个控制流图中的概念。在字节码中，我们可以简单理解为往回跳转的指令。（注意，这并不一定符合循环回边的定义。）

```
public static void foo(Object obj) {
  int sum = 0;
  for (int i = 0; i < 200; i++) {
    sum += i;
  }
}
```

举例来说，上面这段代码将被编译为下面的字节码。其中，偏移量为 18 的字节码将往回跳至偏移量为 7 的字节码中。在解释执行时，每当运行一次该指令，Java 虚拟机便会将该方法的循环回边计数器加 1。

```
public static void foo(java.lang.Object);
  Code:
     0: iconst_0
     1: istore_1
     2: iconst_0
     3: istore_2
     4: goto 14
     7: iload_1
     8: iload_2
     9: iadd
    10: istore_1
    11: iinc 2, 1
    14: iload_2
    15: sipush 200
    18: if_icmplt 7
    21: return
```

在即时编译过程中，我们会识别循环的头部和尾部。在上面这段字节码中，循环的头部是偏移量为 14 的字节码，尾部为偏移量为 11 的字节码。

循环尾部到循环头部的控制流边就是真正意义上的循环回边。也就是说，C1 将在这个位置插入增加循环回边计数器的代码。

解释执行和 C1 代码中增加循环回边计数器的位置并不相同，但这并不会对程序造成影响。

实际上，Java 虚拟机并不会对这些计数器进行同步操作，因此收集而来的执行次数也并非精确值。不管如何，即时编译的触发并不需要非常精确的数值。只要该数值足够大，就能说明对应的方法包含热点代码。

具体来说，在不启用分层编译的情况下，当方法的调用次数和循环回边的次数的和，超过由参数 -XX:CompileThreshold 指定的阈值时（使用 C1 时，该值为 1500；使用 C2 时，该值为 10000），便会触发即时编译。

当启用分层编译时，Java 虚拟机将不再采用由参数 -XX:CompileThreshold 指定的阈值（该参数失效），而是使用另一套阈值系统。在这套系统中，阈值的大小是动态调整的。

所谓的动态调整其实并不复杂：在比较阈值时，Java 虚拟机会将阈值与某个系数 s 相乘。该系数与当前待编译的方法数目成正相关，与编译线程的数目成负相关。

系数的计算方法为：
`s = queue_size_X / (TierXLoadFeedback * compiler_count_X) + 1`

其中 `X` 是执行层次，可取 `3` 或者 `4`；
`queue_size_X` 是执行层次为 `X` 的待编译方法的数目；
`TierXLoadFeedback` 是预设好的参数，其中 `Tier3LoadFeedback` 为 `5`，`Tier4LoadFeedback` 为 `3`；
`compiler_count_X` 是层次 `X` 的编译线程数目。

在 64 位 Java 虚拟机中，默认情况下编译线程的总数目是根据处理器数量来调整的（对应参数 -XX:+CICompilerCountPerCPU，默认为 true；当通过参数 -XX:+CICompilerCount=N 强制设定总编译线程数目时，CICompilerCountPerCPU 将被设置为 false）。

Java 虚拟机会将这些编译线程按照 1:2 的比例分配给 C1 和 C2（至少各为 1 个）。举个例子，对于一个四核机器来说，总的编译线程数目为 3，其中包含一个 C1 编译线程和两个 C2 编译线程。

```
对于四核及以上的机器，总的编译线程的数目为：
n = log2(N) * log2(log2(N)) * 3 / 2
其中 N 为 CPU 核心数目。
```

当启用分层编译时，即时编译具体的触发条件如下。

```
当方法调用次数大于由参数 -XX:TierXInvocationThreshold 指定的阈值乘以系数，或者当方法调用次

触发条件为：
i > TierXInvocationThreshold * s || (i > TierXMinInvocationThreshold * s  && i + b >
```

其中 i 为调用次数，b 为循环回边次数。

OSR 编译

可以看到，决定一个方法是否为热点代码的因素有两个：方法的调用次数、循环回边的执行次数。即时编译便是根据这两个计数器的和来触发的。为什么 Java 虚拟机需要维护两个不同的计数器呢?

实际上，除了以方法为单位的即时编译之外，Java 虚拟机还存在着另一种以循环为单位的即时编译，叫做 On-Stack-Replacement（OSR）编译。循环回边计数器便是用来触发这种类型的编译的。

OSR 实际上是一种技术，它指的是在程序执行过程中，动态地替换掉 Java 方法栈桢，从而使得程序能够在非方法入口处进行解释执行和编译后的代码之间的切换。事实上，去优化（deoptimization）采用的技术也可以称之为 OSR。

在不启用分层编译的情况下，触发 OSR 编译的阈值是由参数 -XX:CompileThreshold 指定的阈值的倍数。

该倍数的计算方法为：

```
(OnStackReplacePercentage - InterpreterProfilePercentage)/100
```

```
其中 -XX:InterpreterProfilePercentage 的默认值为 33，当使用 C1 时 -XX:OnStackReplacePer
```

也就是说，默认情况下，C1 的 OSR 编译的阈值为 13500，而 C2 的为 10700。

在启用分层编译的情况下，触发 OSR 编译的阈值则是由参数 -XX:TierXBackEdgeThreshold 指定的阈值乘以系数。

OSR 编译在正常的应用程序中并不多见。它只在基准测试时比较常见，因此并不需要过多了解。

总结与实践

今天我详细地介绍了 Java 虚拟机中的即时编译。

从 Java 8 开始，Java 虚拟机默认采用分层编译的方式。它将执行分为五个层次，分为为 0 层解释执行，1 层执行没有 profiling 的 C1 代码，2 层执行部分 profiling 的 C1 代码，3 层执行全部 profiling 的 C1 代码，和 4 层执行 C2 代码。

通常情况下，方法会首先被解释执行，然后被 3 层的 C1 编译，最后被 4 层的 C2 编译。

即时编译是由方法调用计数器和循环回边计数器触发的。在使用分层编译的情况下，触发编译的阈值是根据当前待编译的方法数目动态调整的。

OSR 是一种能够在非方法入口处进行解释执行和编译后代码之间切换的技术。OSR 编译可以用来解决单次调用方法包含热循环的性能优化问题。

今天的实践环节，你可以使用参数 -XX:+PrintCompilation 来打印你项目中的即时编译情况。

```
88   15       3       CompilationTest::foo (16 bytes)
88   16       3       java.lang.Integer::valueOf (32 bytes)
88   17       4       CompilationTest::foo (16 bytes)
88   18       4       java.lang.Integer::valueOf (32 bytes)
89   15       3       CompilationTest::foo (16 bytes)   made not entrant
89   16       3       java.lang.Integer::valueOf (32 bytes)   made not entrant
90   19 %     3       CompilationTest::main @ 5 (33 bytes)
```

简单解释一下该参数的输出：第一列是时间，第二列是 Java 虚拟机维护的编译 ID。

接下来是一系列标识，包括 %（是否 OSR 编译），s（是否 synchronized 方法），！（是否包含异常处理器），b（是否阻塞了应用线程，可了解一下参数 -Xbatch），n（是否为 native 方法）。再接下来则是编译层次，以及方法名。如果是 OSR 编译，那么方法名后面还会跟着 @以及循环所在的字节码。

当发生去优化时，你将看到之前出现过的编译，不过被标记了“made not entrant"。它表示该方法不能再被进入。

当 Java 虚拟机检测到所有的线程都退出该编译后的“made not entrant”时，会将该方法标记为“made zombie”，此时可以回收这块代码所占据的空间了。

上一页 下一页