第11章 晚期（运行期）优化

当虚拟机发现某个方法或代码块的运行特别频繁时，就会把这些代码认定为热点代码，为了提高热点低吗的执行效率，在运行时虚拟机就会把这些代码编译成与本地平台相关的机器码，并进行各种层次的优化，完成这个任务的编译器称为即时编译器。

解释器与编译器

许多主流的商用虚拟机如HotSpot和J9都同时包含解释器与编译器。

解释器和编译器各有优势：当程序需要迅速启动和执行时，解释器可以首先发挥作用，省去编译时间，立即执行。在程序运行后，随着时间的推移，编译器逐渐返回作用，把越来越多的代码编译成本地代码之后，可以获得更高的执行效率。

解释执行节约内存，编译执行提升效率。

HotSpot虚拟机中内置了两个即时编译器：分别成为Client Complier和Server Complier，简称为C1和C2编译器。程序使用哪个编译器，取决于虚拟机的运行模式。

当然可以使用-client或-server参数强制指定虚拟机运行在client模式或server模式。

混合模式：解释器和编译器下载搭配使用的方式。

当然可以使用参数-Xint强制虚拟机运行于解释模式，这时编译器完全不介入工作，全部代码都使用解释方式执行。

可以使用参数-Xcomp强制虚拟机运行于编译模式，这时将优先采用比阿尼亚方式执行程序，但是解释器仍然要在编译无法进行的情况下介入执行过程。

虚拟机的-version命令可以输出显示模式：

F:\Html\mannual\_files\_all\jvm\_study\笔记>java -version

java version "1.8.0\_102"

Java(TM) SE Runtime Environment (build 1.8.0\_102-b14)

Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14, mixed mode)

F:\Html\mannual\_files\_all\jvm\_study\笔记>java -Xint -version

java version "1.8.0\_102"

Java(TM) SE Runtime Environment (build 1.8.0\_102-b14)

Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14, interpreted mode)

F:\Html\mannual\_files\_all\jvm\_study\笔记>java -Xcomp -version

java version "1.8.0\_102"

Java(TM) SE Runtime Environment (build 1.8.0\_102-b14)

Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14, compiled mode)

即时编译器编译本地代码需要占用程序运行时间，要编译出优化程度更高的代码所花费的时间可能更长，而且想要编译出优化程度更高的代码，解释器可能还要替编译器收集性能监控信息，这对解释执行的速度也有影响。为了在程序启动响应速度与运行效率之间达到最佳平衡，HotSpot虚拟机还会逐渐启用分层编译的策略。

分层编译分为：

第0层：程序解释执行，解释器不开启性能监控功能，可触发第1层编译

第1层：也称为C1编译，将字节码编译为本地代码，进行简单、可靠的优化，如有必要将加入性能监控的逻辑

第2层（或2层以上）：也称为C2编译，也是将字节码编译为本地代码，但是会启用一些编译耗时较长的优化，甚至会根据性能监控信息进行一些不可靠的激进优化。

分层编译后，C1和C2将会同时工作，许多代码可能会被多次编译，用C1获取更高的编译速度，用C2获取更好的编译质量，在解释执行的时候也无须在承担收集性能监控信息的任务。

在运行过程中会被即时编译器编译的热点代码有两类：

1. 被多次调用的方法

由方法调用触发JIT编译，会以整个方法作为编译对象

1. 被多次执行的循环体

由循环体触发JIT编译，但编译器依然会以整个方法而不是单独的循环体作为编译对象，也被称为栈上替换，简称OSR编译，即方法栈帧还在栈上，方法就被替换了。

热点探测：判断一段代码是不是热点代码，是不是需要触发即使编译。目前主要的热点探测方式有两种：

1. 基于采样点的热点探测
2. 基于计数器的热点探测：HotSpot采用的就是这种

HotSpot为每个方法准备了两个计数器：

1. 方法调用计数器
2. 回边计数器

在确定虚拟机运行参数的前提下，这两个计数器都有一个确定的阈值，当计数器超过阈值溢出了，就会触发JIT编译。

方法调动计数器：顾名思义，这个计数器就是用来统计方法被调用的次数，它得默认阈值在Client模式下是1500次，在Server模式下是10000次。

这个阈值可以通过虚拟机参数-XX：CompileThreshold来人为设定

调用计数器热度衰减：方法掉调用计数器统计的并不是方法被调用的绝对次数，而是一个相对的执行频率，即一段时间之内方法被调用的次数。当超过一定的时间限度，如果方法调用次数仍然不足以让它提交给即时编译器编译，那这个方法的调用计数器就会被减少一半。这段时间称为此方法统计的半衰周期。

可以通过虚拟机参数-XX:-UseCounterDecay来关闭热度衰减，这时方法计数器统计的即为方法被调用的绝对次数。

可以使用虚拟机参数-XX:CounterHalfLifeTime参数设置半衰周期的时间，单位是秒

回边计数器：统计一个方法中循环体代码执行的次数。

调整回边计数器，可以通过虚拟机参数-XX：OnStackReplacePercentage来间接调整回边计数器的阈值。

回边计数器阈值计算公式：

1. 在C1模式下

为方法调用计数器阈值\*OSR比率/100

OnStackReplacePercentage(OSR比率)默认值为933，如果都取默认值那么回边计数器的默认值为13995

1. 在C2模式下：

为方法调用计数器阈值\*OSR比率-解释器监控比率/100

OnStackReplacePercentage(OSR比率)默认值为140，InterpreterProfilePercentage默认值为33，如果斗气默认值那么回边计数器的默认值为10700.

不同于方法计数器，回边计数器统计的就是方法循环执行的绝对次数。

无论是那种方式触发的JIT编译请求，当编译器还未完成编译之前，都仍然按照解释方式执行，而编译动作则在后台的编译线程中进行。

可以通过虚拟机参数-XX：-BackgroundCompilation来禁止后台编译。这时如果达到了阈值，提交编译请求后执行线程就会阻塞，知道编译完成。

查看及分析即时编译结果

虚拟机通过解释执行代码还是编译执行代码，执行结果没有影响，速度上会有很大差别。

使用虚拟机参数-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。

public class JITDemo {  
 public static final int *NUM*=15000;  
 public static int doubleValue(int i){  
 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*();  
 }  
 }  
}

查看输出：

…

239 33 % 3 com.test.jvm\_study.JITDemo::doubleValue @ 2 (18 bytes)

240 34 3 com.test.jvm\_study.JITDemo::doubleValue (18 bytes)

240 35 % 4 com.test.jvm\_study.JITDemo::doubleValue @ 2 (18 bytes)

241 36 1 java.lang.Object::<init> (1 bytes)

241 33 % 3 com.test.jvm\_study.JITDemo::doubleValue @ -2 (18 bytes) made not entrant

…

243 39 % 4 com.test.jvm\_study.JITDemo::calcSum @ 4 (26 bytes)

还可以加上参数-XX:+PrintInlining要求虚拟机输出方法的内联信息

虚拟机提供了一组通用的反汇编接口，可以接入各种平台下的反汇编适配器来使用。

可以通过虚拟机参数-XX:+PrintAssembly要求虚拟机打印编译方法的反汇编代码

-XX:+PrintCompilation -XX:+UnlockDiagnosticVMOptions -XX:+PrintInlining -XX:+PrintAssembly -XX:+UnlockDiagnosticVMOptions

编译优化技术：

编译方式执行代码比解释方式更快，虚拟机设计团队几乎把第代码的所有优化措施都集中在了即时编译器之中。一般来说即时编译器产生的本地代码（机器码）会比javac产生的字节码更加优秀。