链滴

深入理解 Java 即时编译器（下）

作者：Lord-X

原文链接：https://ld246.com/article/1564805932510

来源网站：链滴

□rose□rose 如果您觉得我的文章对您有帮助的话，记得在GitHub上star一波哈□
ose□rose

□rose□rose GitHub_awesome-it-blog □rose□rose

本文会介绍分层编译的机制，然后介绍即时编译器对应用启动性能的影响。

本文内容基于HotSpot虚拟机，设计Java版本的地方会在文中说明。

0 分层编译概述

在引入分层编译之前，我们需要手动的选择编译器。对于启动性能有要求的短时运行程序，我们会选C1编译器，对应参数-client，对于长时间运行的对峰值性能有要求的程序，我们会选择C2编译器，应参数-server。

Java7引入了分层编译，使用-XX:+TieredCompilation参数开启，它综合了C1的启动性能优势和C2峰值性能优势。

在Java8中默认开启了分层编译，在Java8中，无论是开启还是关闭了分层编译，-cilent和-server参都是无效的了。当关闭分层编译的情况下，JVM会直接使用C2。

分层编译将JVM中代码的执行状态分为了5个层次，五个层次分别是：

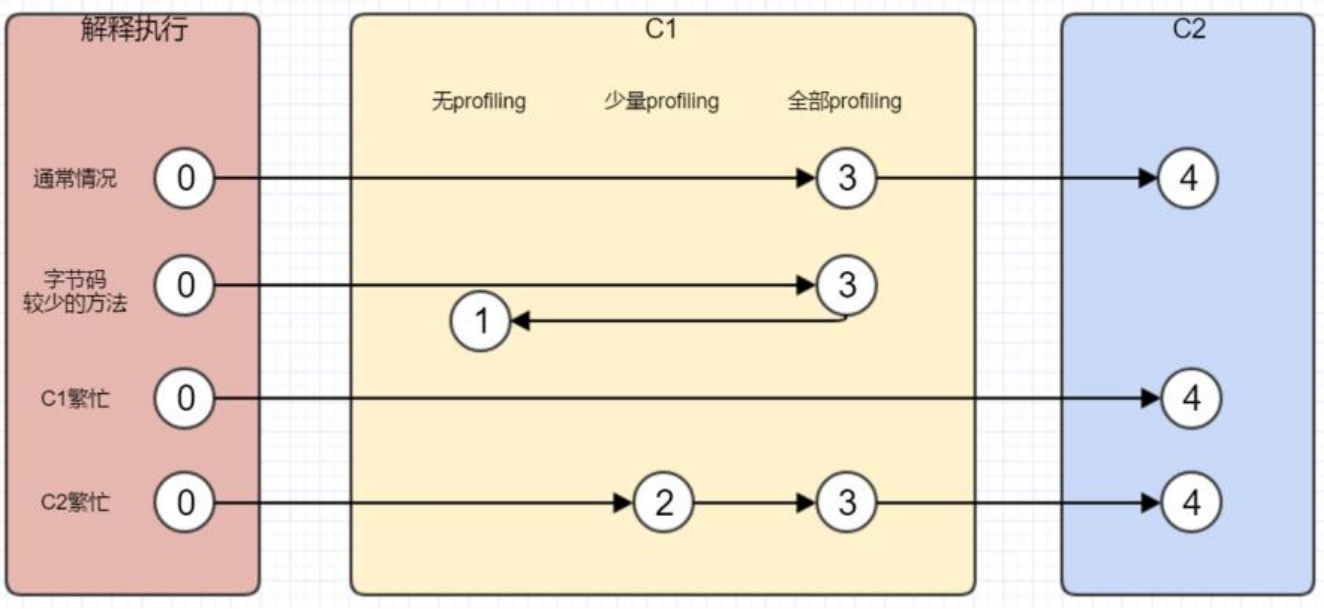
- 0 - 解释执行
- 1 - 执行不带profiling的C1代码
- 2 - 执行仅带方法调用次数和循环回边次数profiling的C1代码
- 3 - 执行带所有profiling的C1代码
- 4 - 执行C2代码

（profiling是指在程序执行过程中收集的程序执行状态数据，例如在上篇中提到的方法调用次数和循回边次数）

这几个层次的代码执行效率由高到低排序如下：4 > 1 > 2 > 3 > 0

其中，1 > 2 > 3的原因在于profiling越多，其性能开销也越大。

下图显示了几种可能的编译执行路径。



第一条执行路径，指的是在通常情况下，热点方法会被3层的C1编译，然后被4层的C2编译。

第二条执行路径，指的是字节码方法较少的情况下，如getter和setter，此时没有什么可收集的profiling，就会在3层编译后，直接交给1层来编译。

第三条执行路径，指的是C1繁忙时，JVM会在解释执行时收集profiling，然后直接有4层的C2编译。

第四条执行路径，指的是C2繁忙时，先由2层的C1编译再由3层的C1编译，这样可以减少方法在3层执行时间，最终再交给C2执行。

1 分层编译实战

1.1 分层编译的触发

本小结说明了在开启分层编译的情况下，上述的五个层次的编译分别在什么时机触发。

在上篇的第二小节，介绍了在不开启分层编译的情况下，触发即时编译的时机与-XX:CompileThreshold参数有关（具体可参考上篇）。

在开启分层编译的情况下，这个参数设定的阈值将失效，取而代之的是另一种计算阈值的方案，这个值是动态调整的（会乘一个系数s），当方法调用次数和循环回边次数满足下述两个公式的任意一个，将会触发第X层的即时编译（{X}表示第X层）。

```
method_invoke_number > Tier{X}InvocationThreshold * s
or
method_invoke_number > Tier{X}MinInvocationThreshold * s 且 method_invoke_number + loop_number > Tier{X}CompileThreshold * s
```

说明：
* method_invoke_number：方法调用次数
* loop_number：循环回边次数
* Tier{X}InvocationThreshold：由JMV参数指定，X可取3或4，第3层的默认值为200，第4层的默值为15000
* s：动态调整的系数（接下来会说明它的计算方式）
* Tier{X}MinInvocationThreshold：JVM设定的参数，X可取3或4，第3层的默认值为100，第4层的认值为600
* Tier{X}CompileThreshold：JVM设定的参数，X可取2或3或4，第2层的默认值为0，第3层的默认为2000，第4层的默认值为15000

PS：在【附加】中提供了查看JVM参数默认值的方式

系数s的计算方式：

s = compiler_method_number_{X} / (Tier{X}LoadFeedback * compiler_thread_number_{X}) + 1
* compiler_method_number_{X}：第X层待编译方法的数目
* Tier{X}LoadFeedback：JVM参数，X可取3或4，第3层的默认值为5，第4层的默认值为3
* compiler_thread_number_{X}：第X层编译线程数目

compiler_thread_number_{X}的计算方式为：

在64位的JVM中，默认情况下编译线程的总数目thread_total是根据CPU的数量来调整的，thread_total的计算方式如下所示，JVM会把这些线程按照1:2的比例分配给C1和C2。

thread_total = log2(N) * log2(log2(N)) * 3 / 2
* N为CPU核心数
例如一个4核的机器，总的编译线程数目thread_total = 3，那么会给C1分配1个线程，C2分配2个线程

由此可以计算出，JVM默认配置情况下，4核CPU，第三层触发C1即时编译的阈值为：

假设第3层有10000个待编译的方法，系数s = 10000 / (5 * 1) + 1 = 2001

那么

method_invoke_number > 200 * s = 200 * 2001 = 400200

也就是方法调用次数超过400200次的时候触发第3层的C1即时编译。

或者

method_invoke_number > 100 * s = 100 * 2001 = 200100 且 method_invoke_number + loop_umber > 2000 * s = 2000 * 2001 = 4002000

即：方法调用次数>200100 并且 方法调用次数+循环回边次数>4002000次时，触发3层的C1即时编。

同理可以计算出第4层C2的即时编译阈值：

method_invoke_number > 30015000时

或者

method_invoke_number > 1200600 且 method_invoke_number + loop_number > 30015000时

会触发第4层的C2即时编译。

1.2 分层编译日志

以上篇的一段代码为例，说明分层编译的日志。

```
/**
*    添加JVM参数：  -XX:+PrintCompilation ，打印编译日志
*/
public class JITDemo2 {

    private static Random random = new Random();

    public static void main(String[] args) throws InterruptedException {
        long start = System.currentTimeMillis();
        int count = 0;
        int i = 0;
        while (i++ < 15000) {
            count += plus();
        }
    }

    // 调用时，编译器计数器+1
    private static int plus() {
        int count = 0;
        // 每次循环，编译器计数器+1
        for (int i = 0; i < 10; i++) {
            count += random.nextInt(10);
        }
        return random.nextInt(10);
    }
}
```

执行结果如下：

```
    176    1       3       java.util.Arrays::copyOf (19 bytes)
    176    6       3       java.io.ExpiringCache::entryFor (57 bytes)
    177    7       3       java.util.LinkedHashMap::get (33 bytes)
    177    8       2       java.lang.String::hashCode (55 bytes)
    177    9       3       java.lang.String::equals (81 bytes)
    178   10       2       java.lang.CharacterData::of (120 bytes)
    178   11       2       java.lang.CharacterDataLatin1::getProperties (11 bytes)
    179   12       3       java.lang.String::<init> (82 bytes)
    179   17     n 0       java.lang.System::arraycopy (native)   (static)
    179   13       2       java.lang.String::indexOf (70 bytes)
    179    3       4       java.lang.Object::<init> (1 bytes)
    179    2       4       java.lang.AbstractStringBuilder::ensureCapacityInternal (27 bytes)
    179    5       4       java.lang.String::length (6 bytes)
    179   15       3       java.lang.Math::min (11 bytes)
    180   14       3       java.util.Arrays::copyOfRange (63 bytes)
    180    4       4       java.lang.String::charAt (29 bytes)
    180   16       3       java.lang.String::indexOf (7 bytes)
    180   18       3       java.util.HashMap::hash (20 bytes)
    180   19       3       java.lang.String::substring (79 bytes)
```

```
   181   20       4       java.util.TreeMap::parentOf (13 bytes)
   181   21       3       java.lang.Character::toUpperCase (6 bytes)
   181   22       3       java.lang.Character::toUpperCase (9 bytes)
   181   23       3       java.lang.CharacterDataLatin1::toUpperCase (53 bytes)
   181   24       3       java.lang.String::getChars (62 bytes)
   182   25       3       java.io.File::isInvalid (47 bytes)
   184   26       3       java.lang.String::startsWith (7 bytes)
   184   28       3       sun.nio.cs.UTF_8$Encoder::encode (359 bytes)
   184   31  s    4       java.lang.StringBuffer::append (13 bytes)
   184   32       4       java.lang.AbstractStringBuilder::append (29 bytes)
   185   33       4       java.io.WinNTFileSystem::isSlash (18 bytes)
   185   29       3       java.lang.String::indexOf (166 bytes)
   185   27       3       java.lang.String::startsWith (72 bytes)
   186   30       3       java.lang.String::toCharArray (25 bytes)
   186   34       3       java.lang.StringBuffer::<init> (6 bytes)
   186   35       3       java.lang.AbstractStringBuilder::<init> (12 bytes)
   186   37     n 0       sun.misc.Unsafe::getObjectVolatile (native)
   186   36       3       java.util.concurrent.ConcurrentHashMap::tabAt (21 bytes)
   187   38     n 0       sun.misc.Unsafe::compareAndSwapLong (native)
   187   41       3       java.util.Random::nextInt (74 bytes)
   187   39       3       java.util.concurrent.atomic.AtomicLong::get (5 bytes)
   187   40       3       java.util.concurrent.atomic.AtomicLong::compareAndSet (13 bytes)
   187   42       3       java.util.Random::next (47 bytes)
   188   43       1       java.util.concurrent.atomic.AtomicLong::get (5 bytes)
   188   39       3       java.util.concurrent.atomic.AtomicLong::get (5 bytes)   made not entrant
   188   44       1       java.util.concurrent.atomic.AtomicLong::compareAndSet (13 bytes)
   188   46       4       java.util.Random::nextInt (74 bytes)
   188   40       3       java.util.concurrent.atomic.AtomicLong::compareAndSet (13 bytes)   mad
 not entrant
*  188   45       3       com.example.demo.gcdemo.JITDemo2::plus (36 bytes)
   188   47       4       java.util.Random::next (47 bytes)
   189   42       3       java.util.Random::next (47 bytes)   made not entrant
*  189   48       4       com.example.demo.gcdemo.JITDemo2::plus (36 bytes)
   189   41       3       java.util.Random::nextInt (74 bytes)   made not entrant
*  191   45       3       com.example.demo.gcdemo.JITDemo2::plus (36 bytes)   made not entran
```

说明一下日志格式（最前面的*号忽略，这是为了标记出plus方法）：

- 第一列：时间（毫秒）
- 第二列：JVM维护的编译ID
- 第三列：一些标识，比如上面出现的n和s，n表示是否是native方法，显示在日志中为true，没显为false。s表示是否是synchronized方法。此外还有：%表示是否是OSR编译，!表示是否包含异常处器，b表示是否阻塞应用线程。
- 第四列：编译的层次，0-4层
- 第五列：编译的方法名
- made not entrant：之前被编译过的方法发生了“去优化”，这个在上篇中已经提到过

从日志可以观察出，plus方法首先触发了3层的C1即时编译，然后触发了4层的C2的即时编译，最后标记为made not entrant，即plus方法发生了去优化。

这里为什么会发生去优化呢，笔者猜想，made not entrant也就是不会再被进入，因为即时编译器

将编译完的代码存入CodeCache，而CodeCache是在堆外内存的，JVM进程的结束不会释放这块堆内存，这样会造成内存泄漏。那么为了释放CodeCache，就需要在JVM结束前对其所有内存进行回，而CodeCache中的内容被回收的依据是所有线程都退出被标记为made not entrant方法时，该方的CodeCache就可以被回收。

PS：通过下面代码可以在程序中获取CodeCache的使用情况

```
// 查看Code Cache使用量
List<MemoryPoolMXBean> beans = ManagementFactory.getMemoryPoolMXBeans();
for (MemoryPoolMXBean bean : beans) {
    if ("Code Cache".equalsIgnoreCase(bean.getName())) {
        System.out.println("max: " + bean.getUsage().getMax() + " bytes, used: " + bean.getUsa
e().getUsed() + " bytes");
    }
}
```

2 即时编译器对应用程序启动的影响

先来说一下发现的问题：应用启动后，CPU使用率和负载飙升，导致部分请求失败，频繁报警，大概持续1分钟左右。

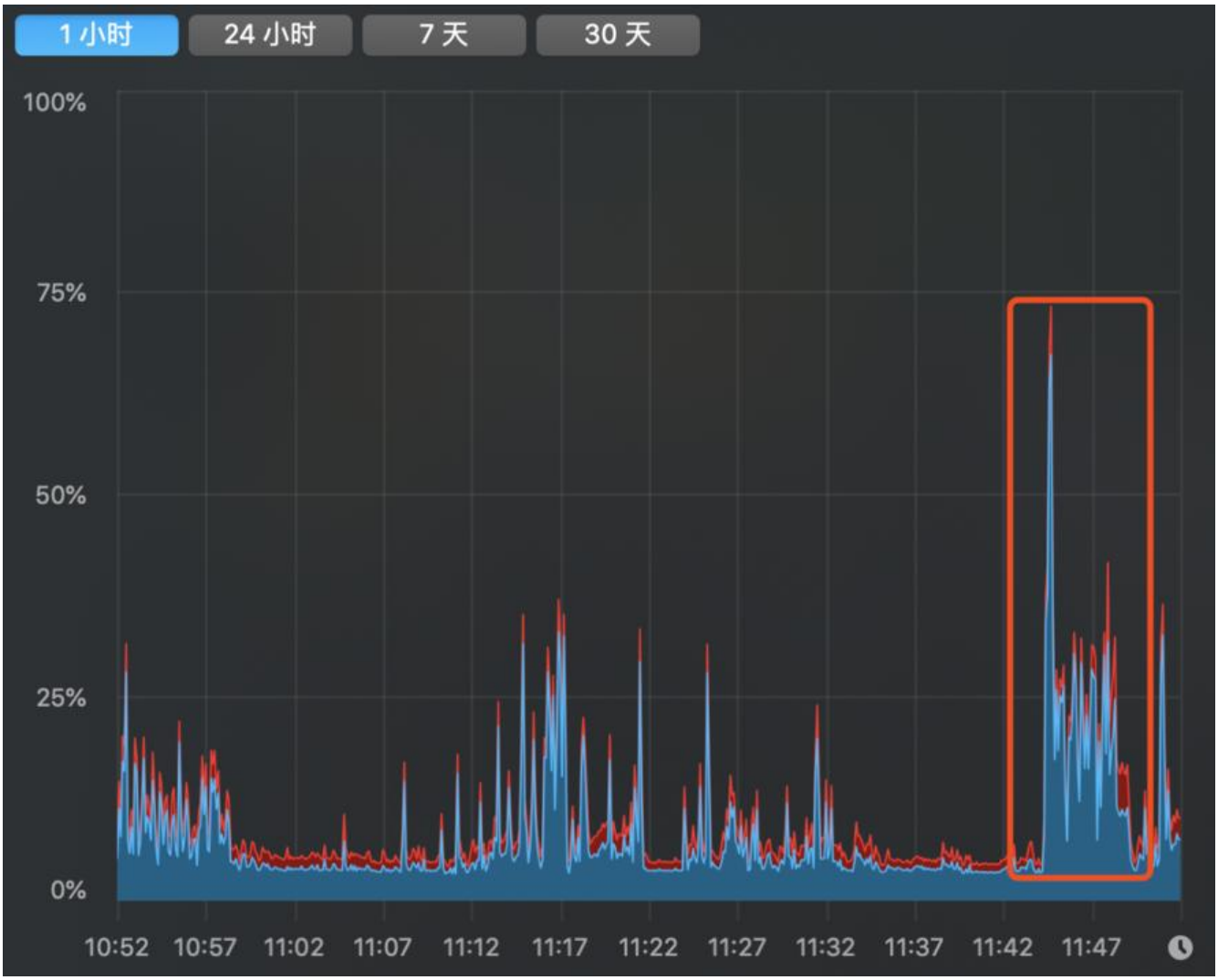
然后考虑是否是即时编译器的影响。当时我们在生产环境使用的是jdk1.7.0_67，且没有开启分层编译然后想到java8对编译器做了一些优化，并且是默认开启分层编译的，然后将其中的一台机器升级到ja a8，再重新启动，发现CPU使用率和负载都降低了。

由于当时的截图没有了，这里我自己做了一个web程序的小demo。

下面会分别比较java7环境和java8环境的启动后CPU使用率和负载变化。

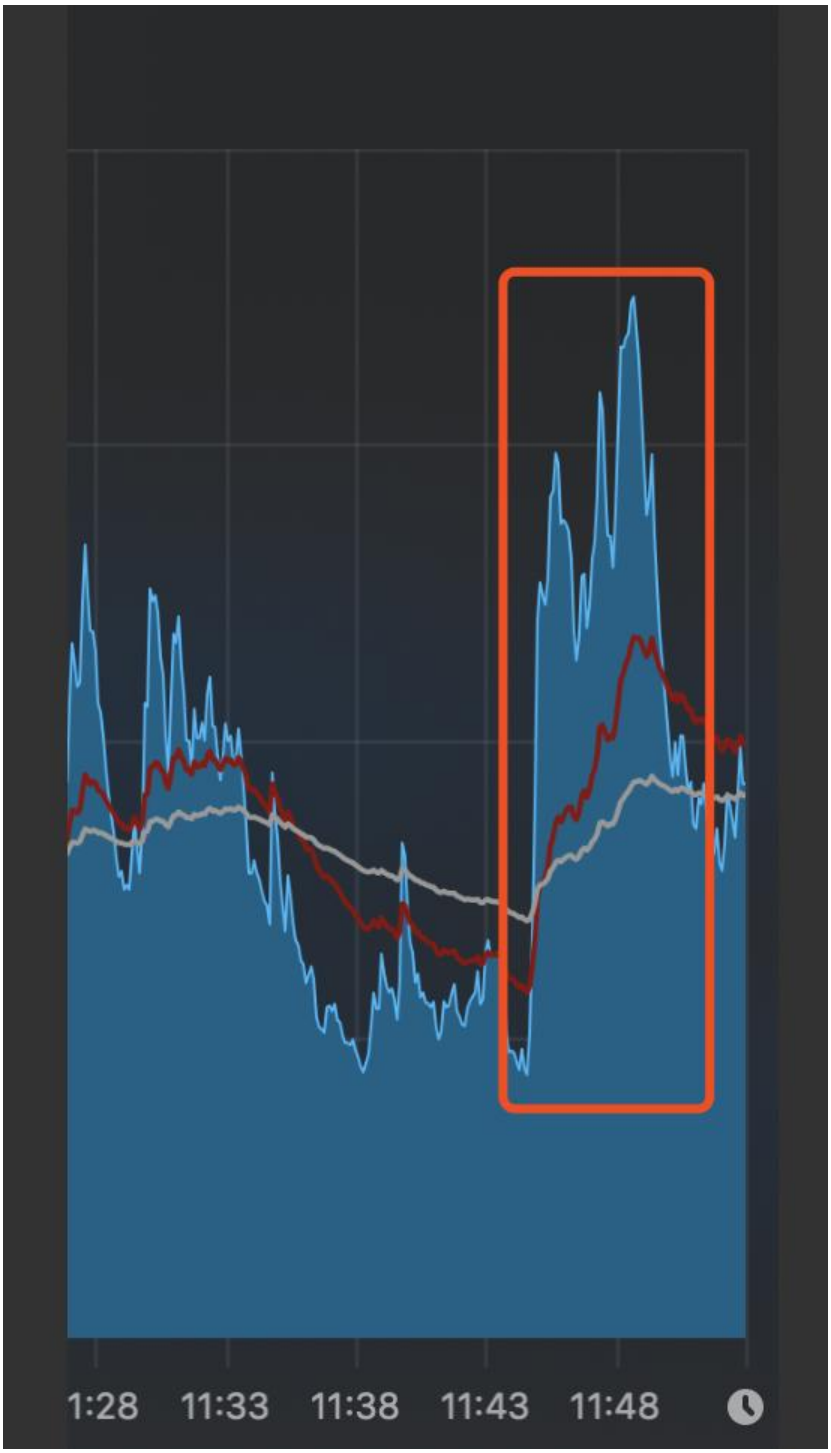
java7默认JVM参数情况下的CPU使用率和复杂变化（不开启分层编译）：

CPU使用率：



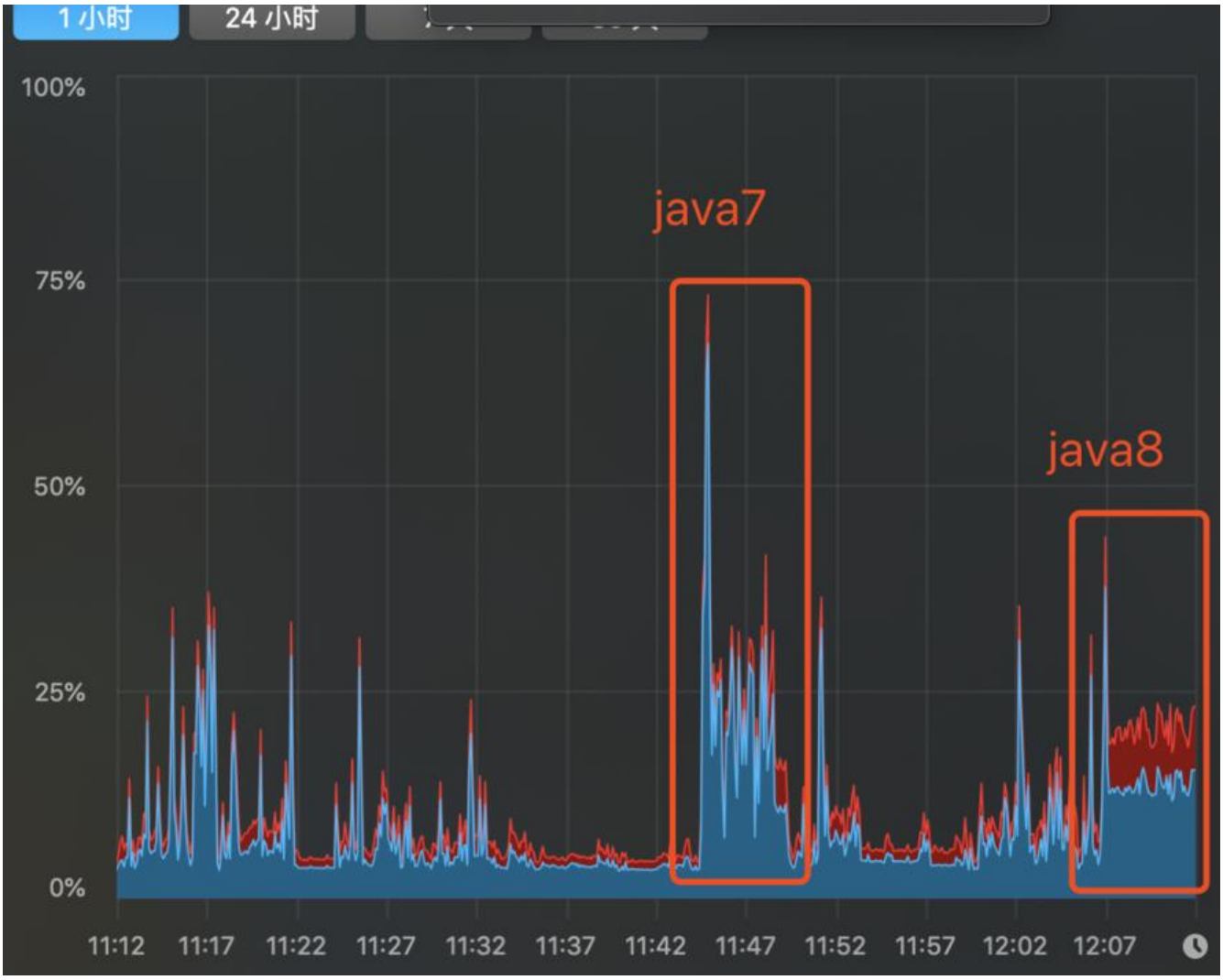
CPU负载：

原文链接：深入理解 Java 即时编译器（下）



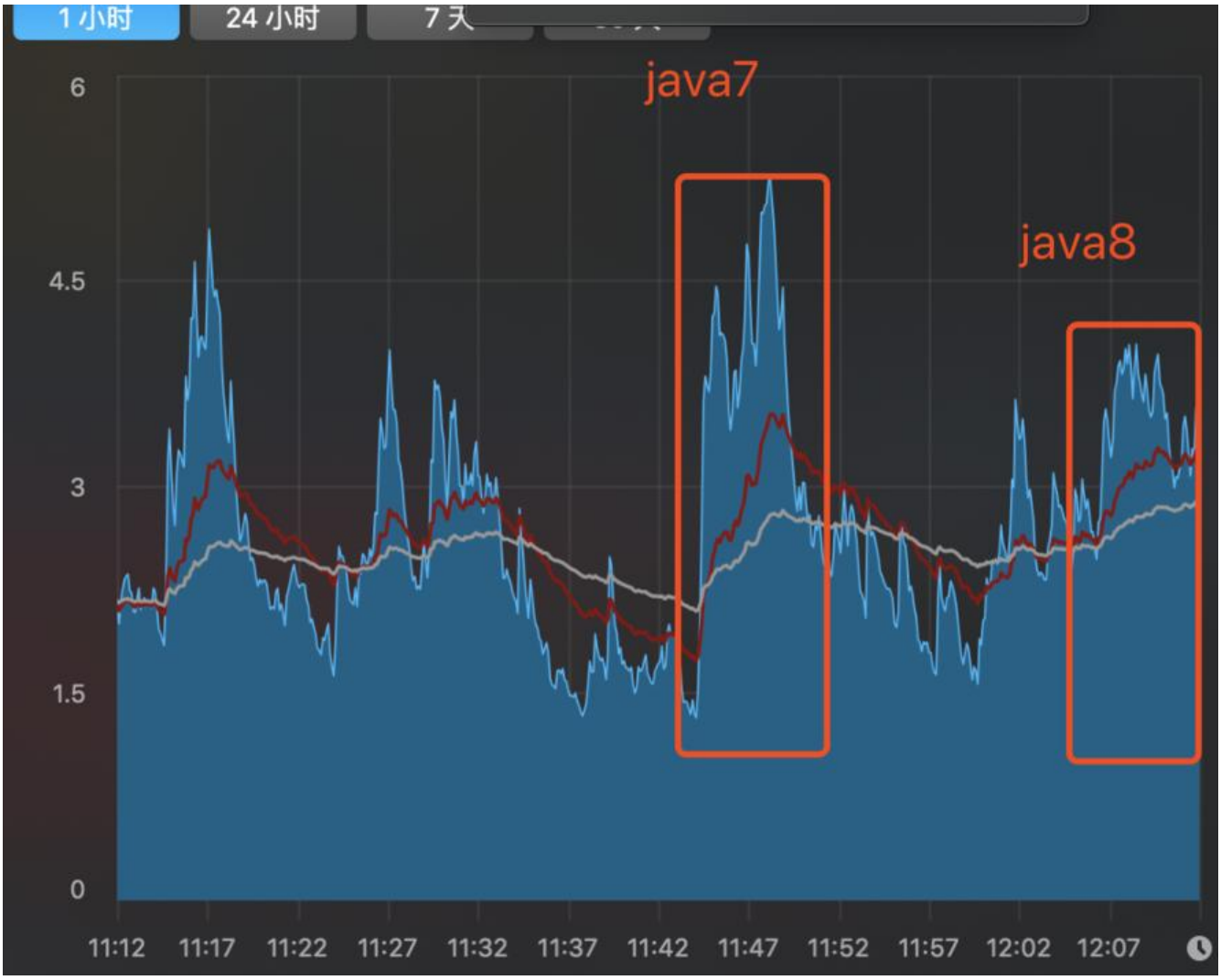
Java8默认JVM参数情况下的CPU使用率和复杂变化（开启分层编译）：

CPU使用率：



CPU负载：

原文链接：深入理解 Java 即时编译器（下）



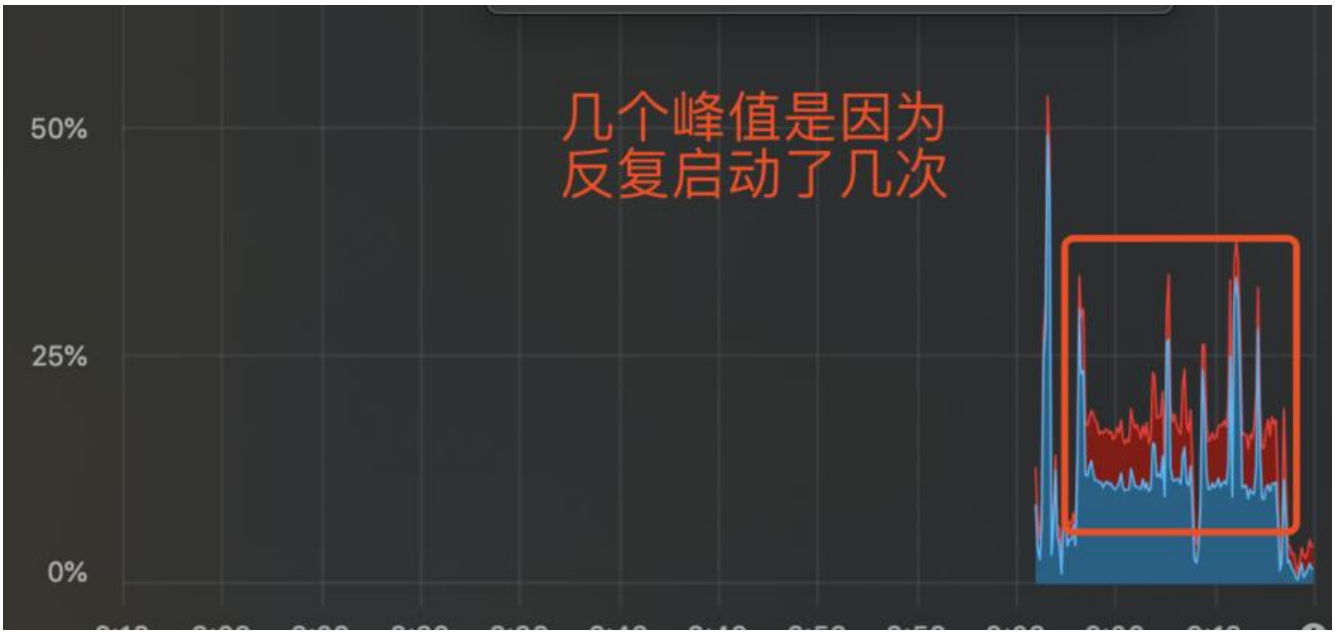
由此可以看出，同为即时编译器默认参数情况下，java8在启动性能上提升了很多。

那如何确定分层编译是否会影响启动性能呢？因为在java7中已经支持了分层编译，所以在java7环境将分层编译打开，就可以进行比对。

需要说明的是，这个比对并不严格，java7在CodeCache的回收上做的不好，这方面在java8中得到改进，除此之外还有一些其他方面的改进，所以这是一个不严格的测试，但大体能说明问题。

将java7的启动参数加上-XX:+TieredCompilation，下面是CPU使用率和CPU负载的变化情况。

CPU使用率的变化：



CPU负载的变化：

由此可见分层编译的开启有利于提升应用的启动性能。

3 思考：分层编译对代码执行性能的影响

3.1 从分层编译的模式考虑

- 在不开启分层编译的情况下，代码以混合模式执行，当方法调用次数和循环回边次数达到设定的阈时，会触发对应编译器的即时编译，这个设定的阈值是固定的。
- 在开启分层编译的情况下，每一层即时编译触发的阈值是动态计算的，而且会根据JVM当前执行状的不同，选用不同的编译器编译，例如C1繁忙时，会直接提交给C2执行，C2繁忙时，会先有C1编译在逐步的提交给C2执行。

3.2 CodeCache方面

- 不开启分层编译的情况下，64位JVM的CodeCache默认大小为48M
- 开启分层编译的情况下，64位JVM的CodeCache的默认大小为256M

由于CodeCache如果越小，GC的次数越频繁，越影响编译器的性能，CodeCache过大也不好，会提单词GC需要的时间，所以CodeCache尽可能要调整成最合适的大小。

PS：CodeCache的GC笔者没有研究过，所以这里GC对其的影响也是一个猜测。

4 附加

查看JVM默认值的方式：

-XX:+PrintFlagsFinal

例如：java -XX:+PrintFlagsFinal -version > options.txt

结果如下：

```
     intx ThreadPriorityPolicy                      = 0                                   {product}
     bool ThreadPriorityVerbose                     = false                               {product}
    uintx ThreadSafetyMargin                        = 52428800                            {product}
     intx ThreadStackSize                           = 1024                                {pd product}
    uintx ThresholdTolerance                        = 10                                  {product}
     intx Tier0BackedgeNotifyFreqLog                = 10                                  {product}
     intx Tier0InvokeNotifyFreqLog                  = 7                                   {product}
     intx Tier0ProfilingStartPercentage             = 200                                 {product}
     intx Tier23InlineeNotifyFreqLog                = 20                                  {product}
     intx Tier2BackEdgeThreshold                    = 0                                   {product}
     intx Tier2BackedgeNotifyFreqLog                = 14                                  {product}
     intx Tier2CompileThreshold                     = 0                                   {product}
     intx Tier2InvokeNotifyFreqLog                  = 11                                  {product}
     intx Tier3BackEdgeThreshold                    = 60000                               {product}
     intx Tier3BackedgeNotifyFreqLog                = 13                                  {product}
     intx Tier3CompileThreshold                     = 2000                                {product}
     intx Tier3DelayOff                             = 2                                   {product}
     intx Tier3DelayOn                              = 5                                   {product}
     intx Tier3InvocationThreshold                  = 200                                 {product}
     intx Tier3InvokeNotifyFreqLog                  = 10                                  {product}
     intx Tier3LoadFeedback                         = 5                                   {product}
     intx Tier3MinInvocationThreshold               = 100                                 {product}
     intx Tier4BackEdgeThreshold                    = 40000                               {product}
     intx Tier4CompileThreshold                     = 15000                               {product}
     intx Tier4InvocationThreshold                  = 5000                                {product}
     intx Tier4LoadFeedback                         = 3                                   {product}
     intx Tier4MinInvocationThreshold               = 600                                 {product}
     bool TieredCompilation                         = true                                {pd product}
     intx TieredCompileTaskTimeout                  = 50                                  {product}
     intx TieredRateUpdateMaxTime                   = 25                                  {product}
     intx TieredRateUpdateMinTime                   = 1                                   {product}
     intx TieredStopAtLevel                         = 4                                   {product}
```

5 参考文档

[1] 极客时间《深入拆解Java虚拟机》 郑雨迪

原文链接：深入理解 Java 即时编译器（下）