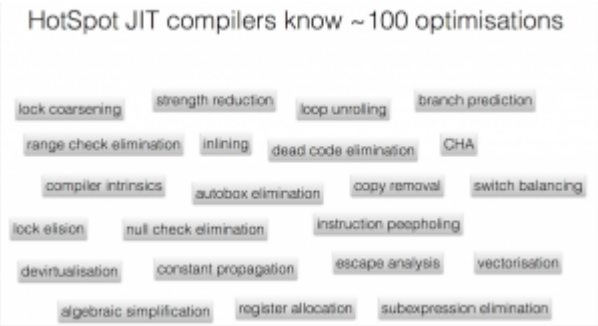
戎码一生

Java JIT性能调优

JVM自动监控这所有方法的执行，如果某个方法是热点方法，JVM就计划把该方法的字节码代码编译成本地机器代码，同时还会在后续的执行过程中进行可能的更深层次的优化，编译成机器代码的过程是在独立线程中执行的，不会影响程序的执行；除次以外，JVM还对热点方法和很小的方法内联到调用方的方法中，减少方法栈的创建。这些就是JIT（just in time）。

JIT编译器有近100种优化方式



JIT编译器有近百种优化方式

其中以下三种方式效果非常明显：

- 把bytecode编译成本地代码（native code）：编译后的代码保存在一个特殊的堆上，除非相关的类被卸载，或者本地代码的优化被取消。这个cache有一定的大小限制（可通过启动参数*-XX:ReservedCodeCacheSize*来修改cache的大小），如果这个cache被装满，则JVM无法编译出更多的本地代码，但通常说不会碰到这种情况的。
  - hot method：默认情况，方法执行次数超过10000次的方法，jvm会编译成本地二进制代码，这个数值可以通过设置启动参数*-XX:CompileThreshold=10000*来修改。
  - On Stack Replacement (OSR)：如果某个循环执行的次数非常多，那么这个循环体代码也可能会编译为本地代码
- 分支预测（Branch Prediction）：降低分支条件判断的结果的随机性，使CPU指令流水线缓存命中率提升
- 方法内联（inlining，对性能的提升很大）：方法内联可以减少方法调用，从而减少方法栈的创建。相信大家都知道循环的速度比递归快很多，就是这个原因，另外方法内联后，还使得一些JIT更深入的优化变成可能。jvm可以通过两个启动参数来控制字节码大小为多少的方法可以被内联：

- *-XX:MaxInlineSize*：能被内联的方法的最大字节码大小，默认值为35Byte，这种方法不需要频繁的调用。比如：一般pojo类中的getter和setter方法，它们不是那种调用频率特别高的方法，但是它们的字节码大小非常短，这种方法会在执行后被内联。
- *-XX:FreqInlineSize*：调用很频繁的方法能被内联的最大字节码大小，这个大小可以比MaxInlineSize大，默认值为325Byte（和平台有关，我的机器是64位mac）。

这些优化方法通常是层层依赖的，所以当JIT优化后的代码被JVM应用，就会开始尝试进行更上一层次的优化。因此我们写代码的时候，应该尽量往这些优化方式上面靠。

输出JIT编译和内联过的方法

在JVM启动参数中添加三个启动参数，比如下面的命令，把编译信息输出到inline.log文件中，便于后续使用grep命令分析：

```
1 java -XX:+PrintCompilation -XX:+UnlockDiagnosticVMOptions -XX:+PrintInlining Simp
```

inline.log中内容类似这样：

```
1 31    23 s!    sun.misc.URLClassPath::getLoader (136 bytes)  inline (hot)
```

- 第1列 31：为JVM启动后到该方法被编译相隔的时间，单位为毫秒
- 第2列 23：编译ID，用来跟踪一个方法的编译、优化、深度优化
- 第3列 s!：s是指该方法是synchronized，感叹号是指该方法有对异常的处理
- 第4列 sun.misc.URLClassPath::getLoader：被编译的方法和类名
- 第5列 (136 bytes)：方法的字节码大小
- 第6列 inline(hot)：表示该方法被内联了，且调用频率很高，这一列还有其他值，比如：
  - inline (hot): 该方法被内联了，且调用频率很高
  - too big: 该方法没有被内联，因为方法字节码比-XX:MaxInlineSize的值大
  - hot method too big: 该方法被调用的频率很高，但是方法的字节码比-XX:FreqInlineSize的值大

inline.log文件内容中的方法还以tab缩进的方式来体现方法调用链的层次结构，非常易懂。

输出JIT编译的细节信息

通过添加参数-XX:+PrintCompilation，可以看到的信息其实并不具体，比如：那些方法进行了内联，内联后的二进制代码是怎么样的都没有。而要输出JIT编译的细节信息，就需要在JVM启动参数中添加这个参数：

```
1 -XX:+UnlockDiagnosticVMOptions
2 -XX:+LogCompilation
3 -XX:+TraceClassLoading
4 -XX:+PrintAssembly
```

输出的编译信息，默认情况是在启动JVM的目录下一个名为：hotspot_pid<PID>.log的文件

如果想指定文件路径和文件名的话，可以再添加一个启动参数：

```
-XX:LogFile=<path to file>
```

输出的是一个很大的xml文件，可能有几十上百兆，下面摘出部分内容如下（文件中的汇编代码太长，就不贴了）：

```
<nmethod
compile_id='78'
compiler='C2'
level='4'
entry='0x00000001052bc060'
size='856'
address='0x00000001052bbf10'
relocation_offset='296'
insts_offset='336'
stub_offset='496'
scopes_data_offset='544'
scopes_pcs_offset='624'
dependencies_offset='848'
oops_offset='520'
method='com/github/fastxml/util/ByteUtils isValidTokenChar (B)Z'
bytes='26'
count='7722'
iicount='7722'
stamp='0.344'/>
```

这些内容很难读懂，建议使用JITWatch（https://github.com/AdoptOpenJDK/jitwatch/）的可视化界面来查看JIT编译的细节信息。同时JITWatch还可以给出很多优化建议，给我们有效的优化代码提供参考，详见下文。

JIT编译模式

上面的输出的细节编译信息inline.log文件中，有个字段上“compiler=C2”，这里的C2就是JIT的编译模式，C2表示这个方法进行了深度优化。下面介绍下JIT的编译模式

C1: 通常用于那种快速启动的GUI应用，对应启动参数：- client

C2: 通常用于长时间允许的服务端应用，对应启动参数：- server

分层编译模式（tiered compilation）：这是自从Java SE 7以后的新特性，可通过添加启动参数来开启：

```
-XX:+TieredCompilation
```

这个特性在应用启动阶段使用C1模式以达到快速启动的效果，一旦应用程序运行起来以后，C2模式将取代C1模式，以进行更深度的优化。在Java SE 8中，这个特性是默认的。

JITWatch

前面也提到了，JITWatch可以通过可视化界面来帮助我们分析JVM输出的JIT编译输出日志，还可以帮助我们静态分析jar中的代码是否符合JIT编译优化的条件，还可以以曲线图形的方式展示JIT编译的整个过程中的一些指标，还给我们的代码提意见和建议，非常好用的工具。

下载

JITWatch需要在github上把代码clone下来，然后用maven来运行，地址为：https://github.com/AdoptOpenJDK/jitwatch/

安装hsdis

如果在jvm的启动参数中添加了下面的启动参数：

```
-XX:+UnlockDiagnosticVMOptions
-XX:+LogCompilation
-XX:+TraceClassLoading
-XX:+PrintAssembly
```

但是你发现启动你的java程序后，有如下的报错信息：

```
Java HotSpot(TM) 64-Bit Server VM warning: PrintAssembly is enabled; turning on D
```

或者启动啦JITWATCH后，打开了某个编译信息log文件，但是看不到每个方法编译后的汇编信息，且那么你就需要安装hsdis。hsdis可以帮助我们查看编译后的本地代码，具体可以参考JITWatch提供的文档，根据自己的系统类型来选择安装：https://github.com/AdoptOpenJDK/jitwatch/wiki/Building-hsdis，如果你是mac，可以参考这篇文章：http://nitschinger.at/Printing-JVM-generated-Assembler-on-Mac-OS-X/。

如果安装了hsdis库后，仍然在JITWatch中看不到汇编信息，那你检查下环境变量配置是否正确，实在不行可以尝试下重启电脑。

运行JITWwatch

在代码根目录下执行`launchUI.sh`（Linux/Mac）或则`launchUI.bat`（windows）

如果你使用maven，也可以在代码根目录下这样运行（其他运行方式，请参考JITWatch的github首页）

```
mvn clean compile exec:java
```

如果你使用的是mac，而且idk版本是jdk7，且运行mvn clean compile exec:java时出现下面的错误和异常时：

```
Caused by: java.lang.NullPointerException
    at com.sun.t2k.MacFontFinder.initPSFontNameToPathMap(MacFontFinder.java:339)
    at com.sun.t2k.MacFontFinder.getFontNamesOfFontFamily(MacFontFinder.java:390)
    at com.sun.t2k.T2KFontFactory.getFontResource(T2KFontFactory.java:233)
    at com.sun.t2k.LogicalFont.getSlot0Resource(LogicalFont.java:184)
    at com.sun.t2k.LogicalFont.getSlotResource(LogicalFont.java:228)
    at com.sun.t2k.CompositeStrike.getStrikeSlot(CompositeStrike.java:86)
    at com.sun.t2k.CompositeStrike.getMetrics(CompositeStrike.java:132)
    at com.sun.javafx.font.PrismFontUtils.getFontMetrics(PrismFontUtils.java:31)
    at com.sun.javafx.font.PrismFontLoader.getFontMetrics(PrismFontLoader.java:4
    at javafx.scene.text.Text.<init>(Text.java:153)
    at com.sun.javafx.scene.control.skin.Utils.<clinit>(Utils.java:52)
    ... 13 more

[ERROR] Failed to execute goal org.codehaus.mojo:exec-maven-plugin:1.5.0:java (d
```

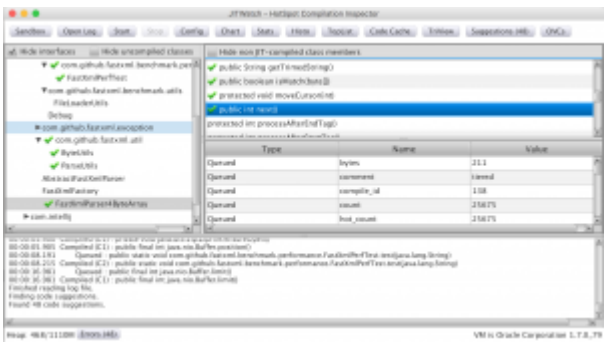
请在org.adoptopenjdk.jitwatch.launch.LaunchUI类的main函数开头处添加下面的代码（或者直接使用我fork修改好的JITWatch）：

```
final Class<?> macFontFinderClass = Class.forName("com.sun.t2k.MacFontFinder");
final java.lang.reflect.Field psNameToPathMap = macFontFinderClass.getDeclaredFi
psNameToPathMap.setAccessible(true);
if (psNameToPathMap.get(null) == null) {
    psNameToPathMap.set(
        null, new java.util.HashMap<String, String>());
}
final java.lang.reflect.Field allAvailableFontFamilies = macFontFinderClass.getD
allAvailableFontFamilies.setAccessible(true);
if (allAvailableFontFamilies.get(null) == null) {
    allAvailableFontFamilies.set(
        null, new String[] {});
}
```

然后重新运行即可看到JITWatch的界面。

用JITWatch来帮助优化代码

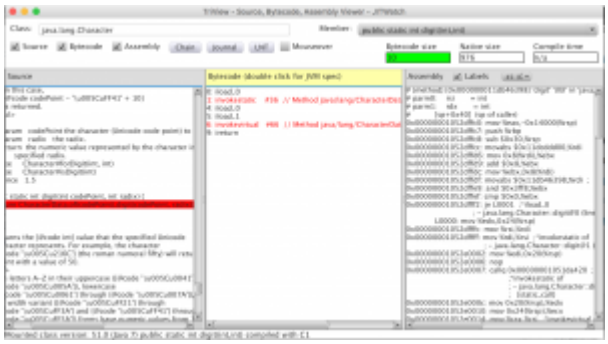
首先点击“Open Log”按钮，选择前面提到过的hotspot_pid<PID>.log文件，然后点击“Start”分析该文件。随后就会在左边生成程序运行过程中加载的类及其目录结构。选择某个类后，右侧会展示该类对应的方法。这些方法中可能部分方法前面有个绿颜色的勾，这说明这个方法被编译成本地代码，选中这个方法后，可以在下方看到该方法具体信息，比如方法调用次数，方法大小等。如下图所示：



jitwatch加载JIT编译log文件

这个界面中，顶部的工具栏都可以自己尝试一下，个人觉得“TopList”和“Suggest”比较直接，我们根据这两个就可以快速的定位需有优化哪些代码了，大体是什么原因导致未编译或者未内联。

选中方法后，点击“TriView”即可查看该方法和字节码和编译后的汇编代码，如下图：



jitwatch方法字节码和编译后的本地代码查看

如果你左边的java代码看不到，那你就需要在上一个界面中点击“Config”来添加源码路径或者源码文件以告诉JITWatch从哪里找源码；如果你右边的汇编代码看不到，说明你上面的hsdis未安装好，请重新安装。

此时，点击上面的“Chain”按钮，即可看到该方法调用了哪些方法，以及这些方法是否被编译了，是否被内联了。如下图所示：

JITWatch查看方法编译和内联状态

总结

JIT的功能能显著提升java程序的性能，尤其是编译为本地代码和内联功能。内联需要方法比较小，也就是说写代码时就尽量把方法写得更小，让方法的复用度更高，复用的越多，就越可能被编译为本地代码。高性能的框架和类库针对JVM的JIT功能进行优化是非常有必要的，JVM提供的调试输出参数和JITWatch这样友好的工具能大大帮助我们快速的发现和定位需要优化的代码，大大提升了效率。

尽管我们可以手动调整JIT相关的一些参数，来让我们的更多的方法被编译和被内联，但一般不建议这么做（大牛都这么说）。

JIT编译成本地代码的过程也是需要消耗时间的，而且编译后本地代码不一定会使用(made not entrant，如果JVM根据一段时间的执行后进行了某项优化，但是在后来的某次执行时验证之前的优化是不完备的，那么JVM会取消这个优化，继续用解释执行的方式来执行字节码)，所以并不是把所有或者大部分代码都编译一定会性能最优，那有可能也是灾难。

我所了解的JVM JIT性能调优的大致原理和方法就是这些，如有错误请指出。

性能优化永远是最后一步，不要提前过早开始性能优化。

Reference

如果你有耐心，就看看下面的文章吧，因为它们比我写的更详细

http://www.oracle.com/technetwork/articles/java/architect-evans-pt1-2266278.html

https://www.chrisnewland.com/images/jitwatch/HotSpot_Profiling_Using_JITWatch.pdf

http://www.docklandsljc.uk/presentations/2015/ChrisNewland-JITWatch.pdf

http://blog.csdn.net/hengyunabc/article/details/26898657

https://advancedweb.hu/2016/05/27/jvm_jit_optimization_techniques/

JVM启动参数：http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html

JITWatch使用文档：https://github.com/AdoptOpenJDK/jitwatch/wiki （右侧的页面目录分类很清晰）

https://advancedweb.hu/2016/05/27/jvm_jit_optimization_techniques/

https://advancedweb.hu/2016/06/28/jvm_jit_optimization_techniques_part_2/

2016年7月18日 weager Java、软件和工具 java、jvm