JIT编译器是“动态编译器”的一种，相对的“静态编译器”则是指的比如：C/C++的编译器

JIT并不是JVM的必须部分，JVM规范并没有规定JIT必须存在，更没有限定和指导JIT。但是，JIT性能的好坏、代码优化程度的高低却是衡量一款JVM是否优秀的最关键指标之一，也是虚拟机中最核心且最能体现虚拟机技术水平的部分。

解释器：Interpreter

编译器：Compiler

jvm

C1:Client Compiler

C2:server Compiler

分层编译（Tiered Compilation）

实施分层编译后，C1和C2将会同时工作，许多代码会被多次编译，用C1获取更高的编译速度，用C2来获取更好的编译质量，且在解释执行的时候解释器也无须再承担收集性能监控信息的任务

1. 什么是Just In Time编译器?

Hot Spot 编译

当 JVM 执行代码时，它并不立即开始编译代码。这主要有两个原因：

首先，如果这段代码本身在将来只会被执行一次，那么从本质上看，编译就是在浪费精力。因为将代码翻译成 java 字节码相对于编译这段代码并执行代码来说，要快很多。

当 然，如果一段代码频繁的调用方法，或是一个循环，也就是这段代码被多次执行，那么编译就非常值得了。因此，编译器具有的这种权衡能力会首先执行解释后的代 码，然后再去分辨哪些方法会被频繁调用来保证其本身的编译。其实说简单点，就是 JIT 在起作用，我们知道，对于 Java 代码，刚开始都是被编译器编译成字节码文件，然后字节码文件会被交由 JVM 解释执行，所以可以说 Java 本身是一种半编译半解释执行的语言。Hot Spot VM 采用了 JIT compile 技术，将运行频率很高的字节码直接编译为机器指令执行以提高性能，所以当字节码被 JIT 编译为机器码的时候，要说它是编译执行的也可以。也就是说，运行时，部分代码可能由 JIT 翻译为目标机器指令（以 method 为翻译单位，还会保存起来，第二次执行就不用翻译了）直接执行。

第二个原因是最优化，当 JVM 执行某一方法或遍历循环的次数越多，就会更加了解代码结构，那么 JVM 在编译代码的时候就做出相应的优化。

我 们将在后面讲解这些优化策略，这里，先举一个简单的例子：我们知道 equals() 这个方法存在于每一个 Java Object 中（因为是从 Object class 继承而来）而且经常被覆写。当解释器遇到 b = obj1.equals(obj2) 这样一句代码，它则会查询 obj1 的类型从而得知到底运行哪一个 equals() 方法。而这个动态查询的过程从某种程度上说是很耗时的。

在主流商用JVM（HotSpot、J9）中，Java程序一开始是通过解释器（Interpreter）进行解释执行的。当JVM发现某个方法或代码块运行特别频繁时，就会把这些代码认定为“热点代码（Hot Spot Code）”，然后JVM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（Just In Time Compiler，JIT）

JIT编译器是“动态编译器”的一种，相对的“静态编译器”则是指的比如：C/C++的编译器

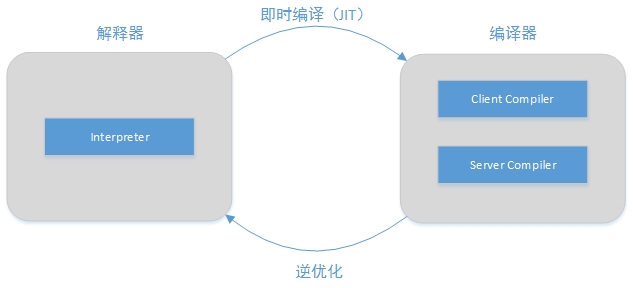
JIT并不是JVM的必须部分，JVM规范并没有规定JIT必须存在，更没有限定和指导JIT。但是，JIT性能的好坏、代码优化程度的高低却是衡量一款JVM是否优秀的最关键指标之一，也是虚拟机中最核心且最能体现虚拟机技术水平的部分。

2. 编译器与解释器

首先，不是所有JVM都采用编译器和解释器并存的架构，但主流商用虚拟机，都同时包含这两部分。

2.1 配合过程

1. 当程序需要迅速启动然后执行的时候，解释器可以首先发挥作用，编译器不运行从而省去编译时间，立即执行程序
2. 在程序运行后，随着时间的推移，编译器逐渐发挥作用，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
3. 当程序运行环境中内存资源限制较大（如部分嵌入式系统中），可以使用解释执行来节约内存；反之，则可以使用编译执行来提升效率。
4. 同时，解释器还可以作为编译器（C2才会激进优化）激进优化时的一个“逃生门”，让编译器根据概率选择一些大多数时候都能提升运行速度的优化手段，当激进优化假设不成立。如：加载了新类后，类型继承结构出现变化，出现“罕见陷阱（Uncommon Trap）”时，可以通过逆优化（Deoptimization）退回到解释状态继续执行   
   (部分没有解释器的虚拟机，也会采用不进行激进优化的C1编译器担任“逃生门”的角色)



2.2 解释器 - Interpreter

Interpreter解释执行class文件，好像JavaScript执行引擎一样

特殊的例子：

* 最早的Sun Classic VM只有Interpreter
* BEA JRockit VM则只有Compiler，但它主要面向服务端应用，部署在其上的应用不重点关注启动时间

2.3 编译器 - Compiler

只说HotSpot JVM

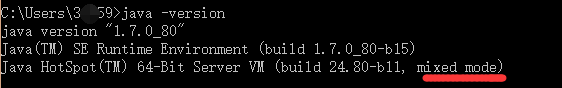
1. C1和C2：

HotSpot虚拟机内置了两个即时编译器，分别称为Client Compiler和Server Compiler，习惯上将前者称为C1，后者称为C2

2. 使用C1还是C2？

HotSpot默认采用解释器和其中一个编译器直接配合的方式工作，使用那个编译器取决于虚拟机运行的模式，HotSpot会根据自身版本和宿主机器硬件性能自动选择模式，用户也可以使用“-client”或”-server”参数去指定

1. 混合模式（Mixed Mode）   
   默认的模式，如上面描述的这种方式就是mixed mode
2. 解释模式（Interpreted Mode）   
   可以使用参数“-Xint”，在此模式下全部代码解释执行
3. 编译模式（Compiled Mode）   
   参数“-Xcomp”，此模式优先采用编译，但是无法编译时也会解释（在最新的HotSpot中此参数被取消）

可以看到，我的JVM现在是mixed mode   


重要：↓

在JDK1.7（1.7仅包括Server模式）之后，HotSpot就不是默认“采用解释器和其中一个编译器”配合的方式了，而是采用了分层编译，分层编译时C1和C2有可能同时工作

3. 分层编译

3.1 为什么要分层编译？

由于编译器compile本地代码需要占用程序时间，要编译出优化程度更高的代码所花费的时间可能更长，且此时解释器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

所以，为了在程序启动响应时间与运行效率之间达到最佳平衡，HotSpot在JDK1.6中出现了分层编译（Tiered Compilation）的概念并在JDK1.7的Server模式JVM中作为默认策略被开启

3.2 编译层 tier（或者叫级别）

分层编译根据编译器编译、优化的规模与耗时，划分了不同的编译层次（不只以下3种），包括：

* 第0层，程序解释执行（没有编译），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

实施分层编译后，C1和C2将会同时工作，许多代码会被多次编译，用C1获取更高的编译速度，用C2来获取更好的编译质量，且在解释执行的时候解释器也无须再承担收集性能监控信息的任务

4. 编译对象与触发条件

1. 谁被编译了？

编译对象就是之前说的“热点代码”，它有两类：

1. 被多次调用的方法   
   * 一个方法被多次调用，理应称为热点代码，这种编译也是虚拟机中标准的JIT编译方式
2. 被多次执行的循环体   
   * 编译动作由循环体出发，但编译对象依然会以整个方法为对象；
   * 这种编译方式由于编译发生在方法执行过程中，因此形象的称为：栈上替换（On Stack Replacement- OSR编译，即方法栈帧还在栈上，方法就被替换了）

2. 触发条件

1. 综述

上面的方法和循环体都说“多次”，那么多少算多？换个说法就是编译的触发条件。

判断一段代码是不是热点代码，是不是需要触发JIT编译，这样的行为称为：热点探测（Hot Spot Detection），有几种主流的探测方式：

1. 基于计数器的热点探测（Counter Based Hot Spot Detection）   
   虚拟机会为每个方法（或每个代码块）建立计数器，统计执行次数，如果超过阀值那么就是热点代码。缺点是维护计数器开销。
2. 基于采样的热点探测（Sample Based Hot Spot Detection）   
   虚拟机会周期性检查各个线程的栈顶，如果某个方法经常出现在栈顶，那么就是热点代码。缺点是不精确。
3. 基于踪迹的热点探测（Trace Based Hot Spot Detection）   
   Dalvik中的JIT编译器使用这种方式

2. HotSpot

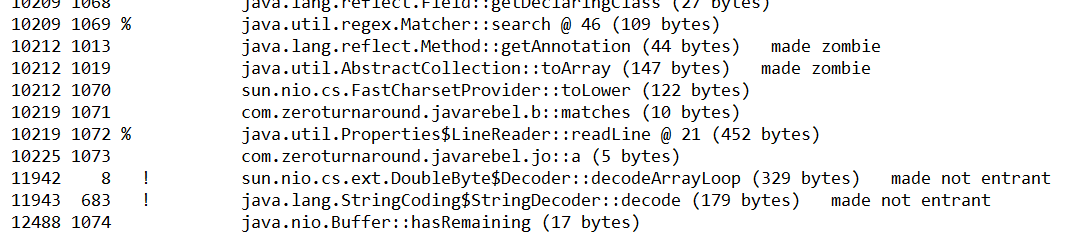
HotSpot使用的是第1种，因此它为每个方法准备了两类计数器：方法调用计数器（Invocation Counter）和回边计数器（Back Edge Counter）

1. 方法计数器
   * 默认阀值，在Client模式下是1500次，Server是10000次，可以通过参数“-XX:CompileThreshold”来设定
   * 当一个方法被调用时会首先检查是否存在被JIT编译过得版本，如果存在则使用此本地代码来执行；如果不存在，则将方法计数器+1，然后判断“方法计数器和回边计数器之和”是否超过阀值，如果是则会向编译器提交一个方法编译请求
   * 默认情况下，执行引擎并不会同步等待上面的编译完成，而是会继续解释执行。当编译完成后，此方法的调用入口地址会被系统自动改写为新的本地代码地址
   * 还有一点，热度是会衰减的，也就是说不是仅仅+，也会-，热度衰减动作是在虚拟机的GC执行时顺便进行的
2. 回边计数器
   * 回边，顾名思义，只有执行到大括号”}”时才算+1
   * 默认阀值，Client下13995，Server下10700
   * 它的调用逻辑和方法计数器差不多，只不过遇到回边指令时+1、超过阀值时会提交OSR编译请求以及这里没有热度衰减

5. 编译过程

编译过程是在后台线程（daemon）中完成的，可以通过参数“-XX:-BackgroundCompilation”来禁止后台编译，但此时执行线程就会同步等待编译完成才会执行程序

1. Client Compiler   
   C1编译器是一个简单快速的三段式编译器，主要关注“局部性能优化”，放弃许多耗时较长的全局优化手段   
   过程：class -> 1. 高级中间代码 -> 2. 低级中间代码 -> 3. 机器代码
2. Server Compiler   
   C2是专门面向服务器应用的编译器，是一个充分优化过的高级编译器，几乎能达到GNU C++编译器使用-O2参数时的优化强度。

使用参数“-XX:+PrintCompilation”会让虚拟机在JIT时把方法名称打印出来，如图：   


6. Java和C/C++的编译器对比

这里不是比Java和C/C++谁快这种大坑问题，只是比较编译器（我认为开发效率上Java快，执行效率上C/C++快）

这种对比代表了经典的即时编译器与静态编译期的对比，其实总体来说Java编译器有优有劣。主要就是动态编译时间压力大能做的优化少，还要做一些动态校验。而静态编译器无法实现一些开发上很有用的动态特性

JVM模式有：-Xint, -Xcomp, 和 -Xmixed

* \* -Xint\*   
  在解释模式(interpreted mode)下，-Xint标记会强制JVM执行所有的字节码，当然这会降低运行速度，通常低10倍或更多。
* \* -Xcomp\*   
  -Xcomp参数与它（-Xint）正好相反，JVM在第一次使用时会把所有的字节码编译成本地代码，从而带来最大程度的优化。这听起来不错，因为这完全绕开了缓慢的解释器
* **-Xmixed**   
  混合模式(mixed mode)，是JVM的默认工作模式。它会同时使用编译模式和解释模式。对于字节码中多次被调用的部分，JVM会将其编译成本地代码以提高执行效率；而被调用很少（甚至只有一次）的方法在解释模式下会继续执行，从而减少编译和优化成本。JIT编译器在运行时创建方法使用文件，然后一步一步的优化每一个方法，有时候会主动的优化应用的行为。这些优化技术，比如积极的分支预测（optimistic branch prediction），如果不先分析应用就不能有效的使用。这样将频繁调用的部分提取出来，编译成本地代码，也就是在应用中构建某种热点（即HotSpot，这也是HotSpot JVM名字的由来）。使用混合模式可以获得最好的执行效率。

[寄存器](https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8)是[中央处理器](https://baike.baidu.com/item/%E4%B8%AD%E5%A4%AE%E5%A4%84%E7%90%86%E5%99%A8)内的组成部分。寄存器是有限存贮容量的高速存贮部件，它们可用来暂存[指令](https://baike.baidu.com/item/%E6%8C%87%E4%BB%A4)、[数据](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE/33305)和[地址](https://baike.baidu.com/item/%E5%9C%B0%E5%9D%80/80420)。在[中央处理器](https://baike.baidu.com/item/%E4%B8%AD%E5%A4%AE%E5%A4%84%E7%90%86%E5%99%A8)的控制部件中，包含的寄存器有[指令寄存器](https://baike.baidu.com/item/%E6%8C%87%E4%BB%A4%E5%AF%84%E5%AD%98%E5%99%A8)(IR)和[程序计数器](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E8%AE%A1%E6%95%B0%E5%99%A8)(PC)。在[中央处理器](https://baike.baidu.com/item/%E4%B8%AD%E5%A4%AE%E5%A4%84%E7%90%86%E5%99%A8)的算术及逻辑部件中，存器有[累加器](https://baike.baidu.com/item/%E7%B4%AF%E5%8A%A0%E5%99%A8)(ACC)。

导入导出Excel

1、用HSSFWorkbook打开或者创建“Excel文件对象”

2、用HSSFWorkbook对象返回或者创建Sheet对象

3、用Sheet对象返回行对象，用行对象得到Cell对象

4、对Cell对象读写。