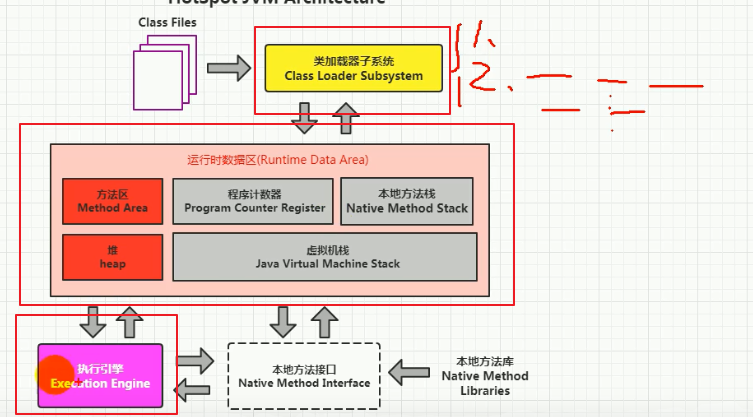
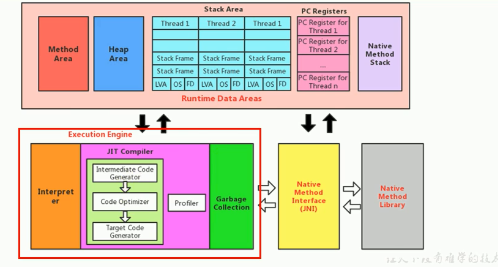
# JVM核心组成部分

1. 类加载器子系统
2. 运行时数据区
3. 执行引擎



# 执行引擎概述



执行引擎属于JVM的下层，里面包括：解释器、即时编译器、垃圾回收器

## 虚拟机和物理机的区别

* 物理机的执行引擎是直接建立在处理器、缓存、指令集和操作系统层面上。
* 虚拟机的执行引擎则是由软件自行实现的，因此可以不受物理条件制约来定制指令集和执行引擎的结构体系，**能够执行那些不被硬件直接支持的指令集格式。**

## JVM的任务

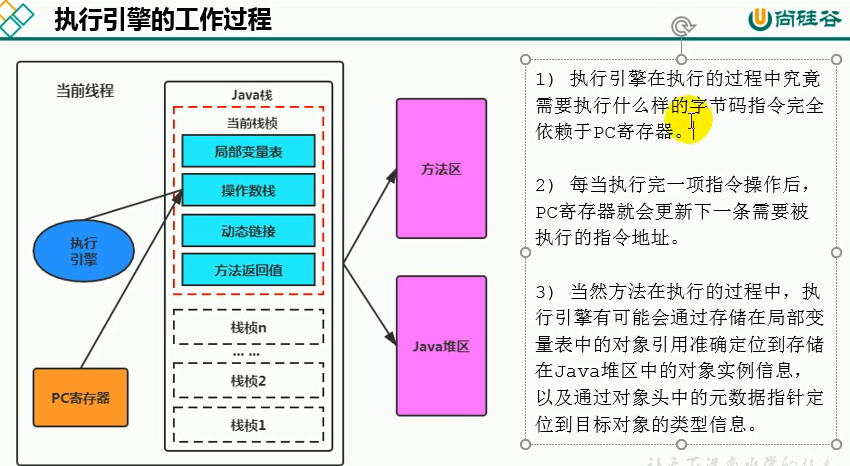
加载字节码文件到虚拟机内存，字节码指令并非等价于本地机器的机器指令，只能被JVM所识别，而不能直接运行在物理机上，因此需要执行引擎来将字节码指令**解释、编译**为对应的物理机的本地机器指令才可以。



因此JVM的执行引擎就是将高级语言翻译为机器语言



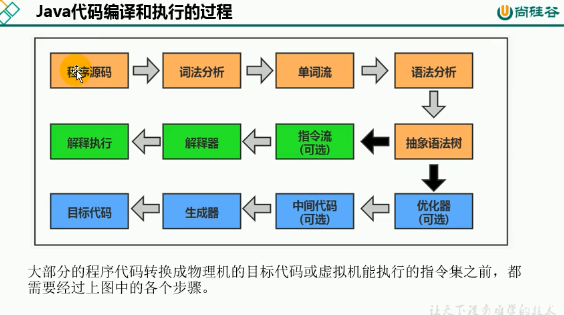
## 执行引擎的工作过程



* 执行引擎根据PC寄存器来明确要执行什么样的字节码指令
* 每当执行完一项指令操作后，PC寄存器就会更新下一条需要被执行的指令地址
* 当方法在执行过程中，执行引擎有可能会通过存储在局部变量表中的对象引用准确定位到存储在Java堆中的对象实例信息，以及通过对象头中的元数据指针定位到方法区中的类型信息。

从外观来看，所有的JVM的执行引擎的输入输出都是一致的，输入的是字节码二进制流，处理过程是字节码解析执行的

# Java代码编译和执行过程

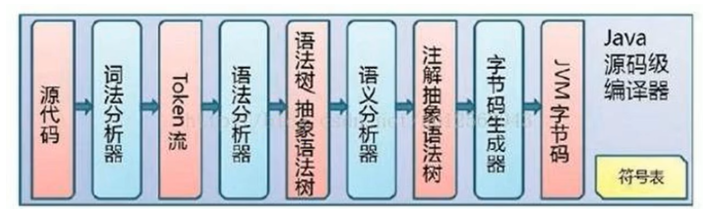


图中橙色的是编译过程

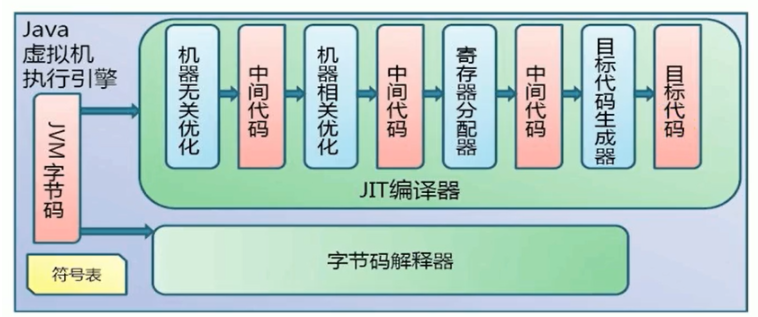
绿色的是解释过程

蓝色的是后期编译过程

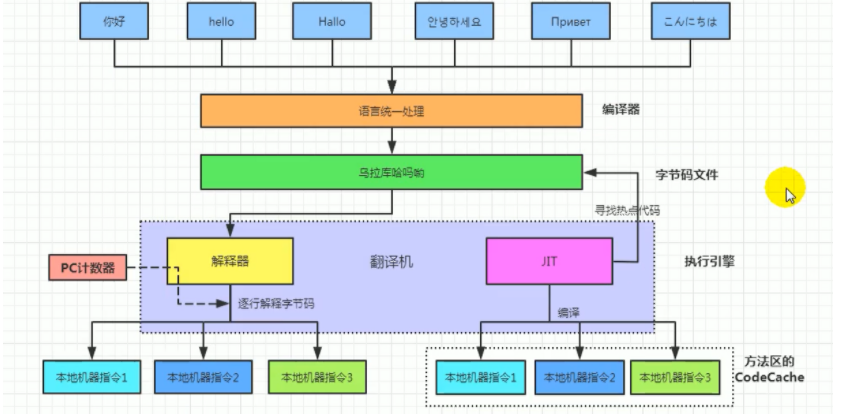
Java代码编译是由Java源码编译器来完成，流程图如下所示：



Java字节码的执行是由JVM执行引擎来完成，流程图如下所示



我们用一个总的图，来总结解释器和编译器



## 解释器（Interpreter）和JIT编译器

* 解释器（Interpreter）：当java虚拟机启动的时候会根据预定义的规范**对字节码采用逐行解释的方式执行**，将每条字节码文件的内容“翻译”为对应平台的本地机器指令执行。 **逐行翻译执行**
* JIt编译器(Just In Time Compiler)：虚拟机将源代码直接编译成和本地机器平台相关的机器语言。**编译完成后才可执行。**

### \*思考：为什么Java是半编译半解释型语言？

Jdk1.0时，JVM执行引擎只有解释器，因此是解释执行。后来JVM发展出了可以直接升成本地代码的编译器。

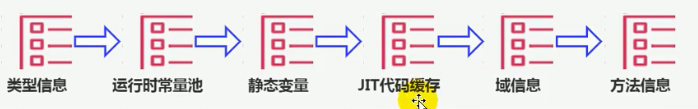
如今JVM在执行Java程序的时候，通常会将解释执行和编译执行二者结合起来进行。

1. 解释器的特点：按行边解释边执行，所以当程序启动后，解释器响应速度快，省去了编译的时间。
2. 编译器的特点：把代码全部编译成本地机器指令后，才能执行。这个过程需要一定的时间，但是编译为本地机器指令后，执行效率高。

在JVM启动的时候，解释器能够首先发挥作用，而不必等待即时编译器全部编译完成后再执行，这样可以省去许多不必要的编译时间。随着时间的推移，编译器发挥作用，根据**热点探测功能**，将有价值的字节码编译为本地机器指令，从而获得更高的执行效率。

### \*思考：为什么方法区中要存储即时编译后的代码缓存？

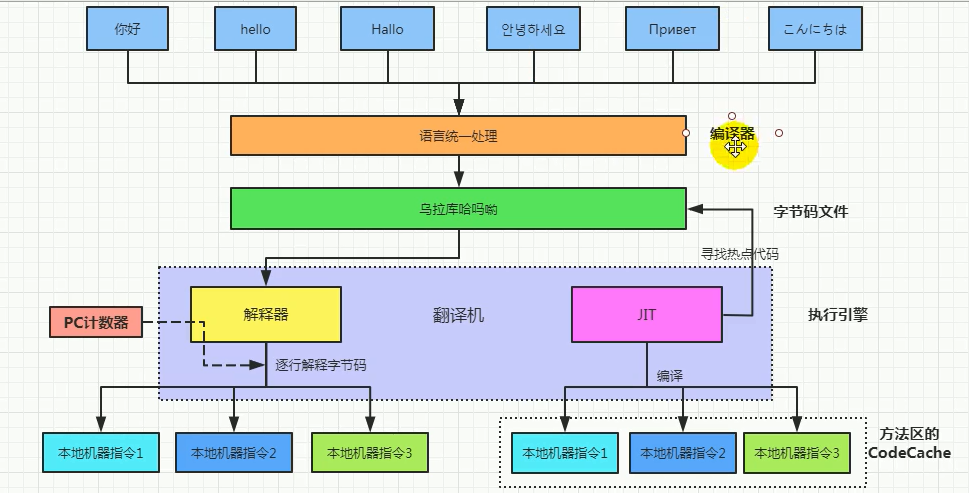
结合方法区的存储内容：类信息、静态变量（JDK8后在堆中）、运行时常量池、即时编译后的代码缓存



即时编译将字节码编译成机器语言，在方法区中做缓存，以后再调用这段代码时可以直接调用机器指令，效率会更高。

所以JIT编译器主要用于处理热点代码。

# 机器码、指令、汇编语言



高级语言---> 编译--->字节码文件--->执行引擎--->本地机器指令

## 机器码

各种用二进制编码方式表示的指令，就是机器指令码，机器语言。

机器码执行速度最快。

机器指令受限于CPU，不同的CPU机器指令也不同。

## **指令**

机器码是二进制的序列，可读性差，发明了指令。

指令就是机器码中的0 1序列，简化成对应的指令，可读性要好些

不同的机器执行同一个操作的时候，机器码可能不同，所以不同的机器的同一个指令，对应的机器码也不同。

指令相当于对机器码的封装，屏蔽了底层因机器不同而不同的机器码的差异。

## **指令集**

不同的硬件平台，各自支持的指令，是有差别的。因此每个平台所支持的指令，称之为对应平台的指令集。如常见的

1. x86指令集，对应的是x86架构的平台
2. ARM指令集，对应的是ARM架构的平台

## **汇编语言**

指令的可读性依旧很差，于是发明了汇编语言。

汇编语言用**助记符**代替机器指令的**操作码**，用**地址符号**或者**标号**代替指令或者**操作数的地址。**

不同的硬件平台，汇编语言对应着不同的机器语言指令集，通过汇编过程转换成机器指令。

由于计算机只能认识指令码，所以用汇编语言编写的程序还必须翻译成机器指令码，计算机才能识别和执行。

## 高级语言

为了让计算机用户编程更容易，于是出现了各种高级计算机语言，高级语言比机器语言、汇编语言更接近人类语言。

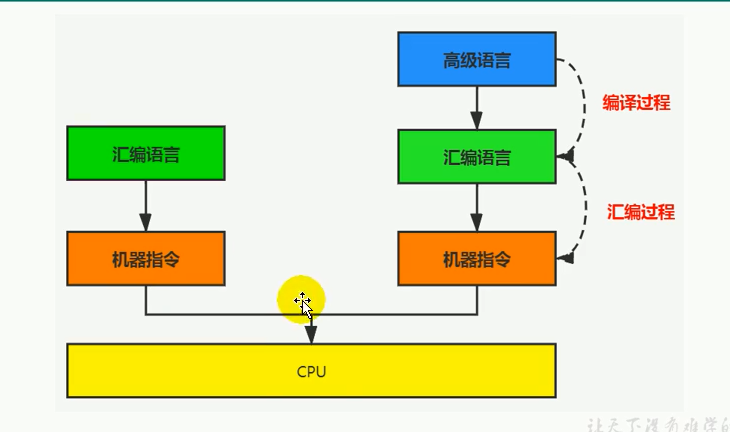
计算机执行高级语言编写的程序时，需要**把程序解释和编译成机器的指令吗。这个过程就叫解释或者编译。**

高级语言也不是直接翻译成 机器指令

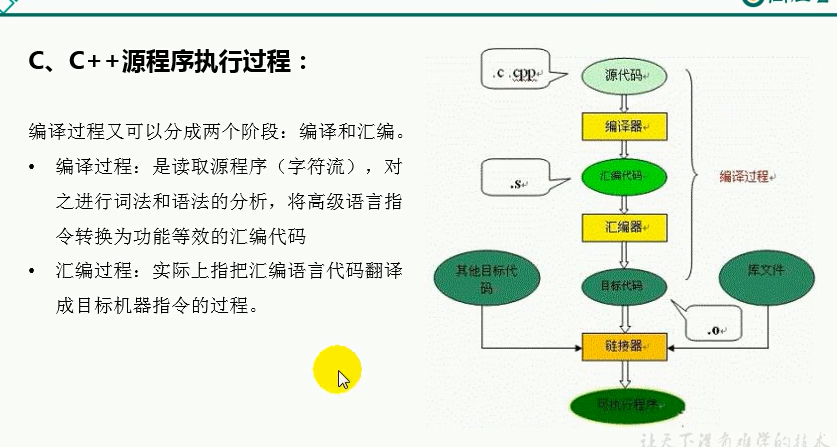
首先需要通过编译阶段，将高级语言翻译为为汇编语言

再通过汇编阶段，将生成的汇编语言翻译成机器指令

如下面说的C和C++



## C++源程序的执行过程

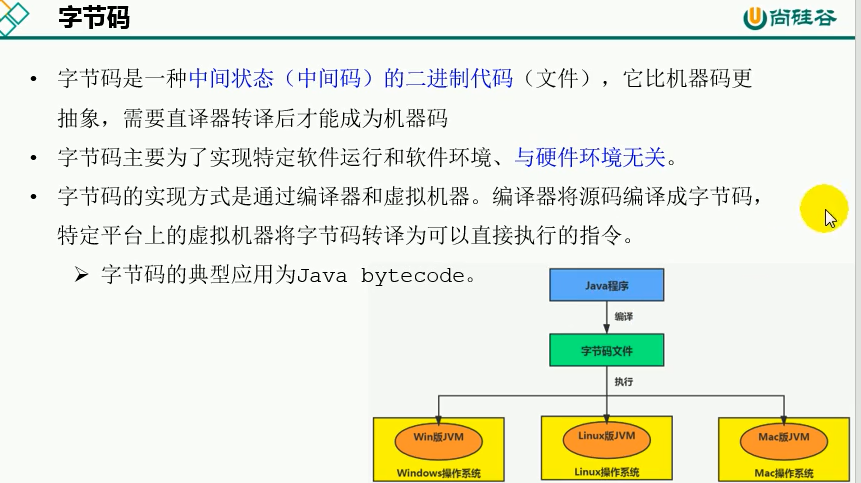


**编译过程又可以分成两个阶段：编译和汇编。**

1. 编译过程：是读取源程序（字符流），对之进行词法和语法的分析，将高级语言指令转换为功能等效的汇编代码
2. 汇编过程：实际上指把汇编语言代码翻译成目标机器指令的过程。

源代码--->编译-->汇编代码-->汇编--->机器指令

## 字节码



# 解释器

JVM设计者们的初衷仅仅只是单纯地为了满足Java程序实现跨平台特性，因此避免采用静态编译的方式直接生成本地机器指令，从而诞生了实现解释器在运行时采用逐行解释字节码执行程序的想法。

解释器相当于同声翻译，将字节码文件的内容逐条翻译为对应平台的本地机器指令执行。

当一条字节码指令被解释执行后，PC寄存器会转移到下一条需要被执行的字节码指令，接着解释器根据PC寄存器来执行相应的操作。

HotSpot VM中，解释器主要有Interpreter模块和Code模块构成：

* Interpreter模块：实现了解释器的核心功能
* Code模块：用于管理HotSpot VM在运行时生成的本地机器指令

### 解释器的缺点：

低效，因为边解释边执行，执行效率受解释影响。

### JVM的改进：

为了改进解释器执行效率低下的缺点，JVM提供了一种即时编译的技术，即时编译的目的是避免函数被解释执行，而是将整个函数体编译称为机器码，每次函数执行时，只执行编译后的机器码即可，这种方式可以是执行效率大幅度提升。

### Java的两种执行方式

1. 解释运行：将源代码编译成字节码文件，然后在运行时通过解释器将字节码文件转为机器码执行
2. 编译后运行：编译执行（直接编译成机器码）。现代虚拟机为了提高执行效率，会使用即时编译技术（JIT，Just In Time）将方法编译成机器码后再执行

# JIT编译器

Just In Time

JIT解释器用于编译执行，直接编译成机器码，就是说**全部编译成机器码之后再执行**。

HotSpot VM采用解释器和JIT编译器并存的架构。

**JVM运行时，解释器和JIT编译器相互协作，各自取长补短，选择最合适的方式来权衡编译本地代码的时间和直接解释执行代码的时间。**

Java运行性能已经达到了可以和C C++程序一较高下的地步。

JIT的优势：

**速度更快，有代码缓存**

## 思考：为什么JVM要让解释器和JIT编译器共存？

1. 解释器的特点：按行边解释边执行，所以当程序启动后，解释器响应速度快，省去了编译的时间。
2. 编译器的特点：把代码全部编译成本地机器指令后，才能执行。这个过程需要一定的时间，但是编译为本地机器指令后，执行效率高。

**在JVM启动的时候，解释器能够首先发挥作用，而不必等待即时编译器全部编译完成后再执行，这样可以省去许多不必要的编译时间。随着时间的推移，编译器发挥作用，根据热点探测功能，将有价值的字节码编译为本地机器指令，从而获得更高的执行效率。**

JRockit VM不包含解释器，字节码全部依靠即时编译器编译后执行。

JRockit VM程序的执行性能非常高效，但是程序在启动的时候必然会花费更长的时间进行编译。

对于服务端的应用来说，启动时间并非是关注重点，但是对于那些看重启动时间的应用场景，就可以将解释器和JIT编译器并存来获取平衡。

编译器在编译的时候，激进优化不成立的时候，解释器用作编译器的逃生门。

## 案例

注意案例中解释执行和编译执行在线上环境微妙的辩证关系。

机器在热机状态可以承受的负载要远大于冷机状态。如果以热机状态时的流量进行切流，可能会导致处于冷机状态的服务器因无法承载流量而假死。

在生产环境发布过程中，以分批的方式发布，根据机器的数量划分成多个批次，每个批次的机器数量只能占整个集群的1/8，如果在平台发布不小心按照两批发布，如果是热机状态，正常情况下一半的机器勉强能承受住流量，但由于刚启动的冷机状态的JVM都是解释执行，还没有进行热点代码统计和JIT动态编译，会导致机器启动后，当前1/2发布成功的服务器马上全部宕机，此故障说明了JIT的存在。

也就是说热机状态的JVM中，已经有热点代码缓存了，在来新的程序流量的时候，能勉强承受住。如果是冷机状态的，来新的程序流量时，还没有热点代码缓存，全部要进行解释执行，这样会导致机器死机。

## 编译器的分类

1. 前端编译器：把.java文件转变成.class文件
2. 后端运行期编译器：JIT编译器，把字节码转变成机器码
3. 静态提前编译器（AOT,Ahead of time Compiler）:直接把.java文件编译成本地机器代码。

# JVM如何判断选择使用哪种方式执行？

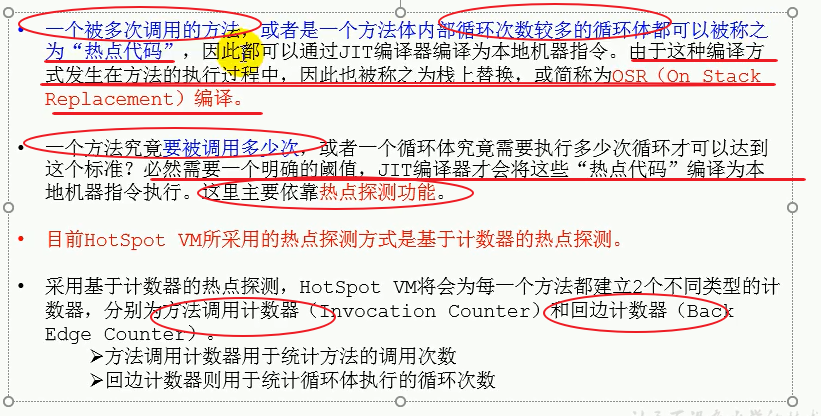
### 热点代码及探测方式

* 热点代码

是否需要启动JIT编译器将字节码直接编译为对应平台的本地机器指令，则需要根据代码被调用执行的频率而定。

关于那些需要被编译为本地代码的字节码，也被称之为“热点代码”，JIT编译器在运行时会针对那些频繁被调用的“热点代码”做出深度优化，将其直接编译为对应平台的本地机器指令，以此提升Java程序的执行性能。

一个方法究竟要被调用多少次，或者一个循环体究竟需要执行多少次循环才可以达到这个标准？必然需要一个明确的阈值，JIT编译器才会将这些“热点代码”编译为本地机器指令执行。这里主要依靠热点探测功能。



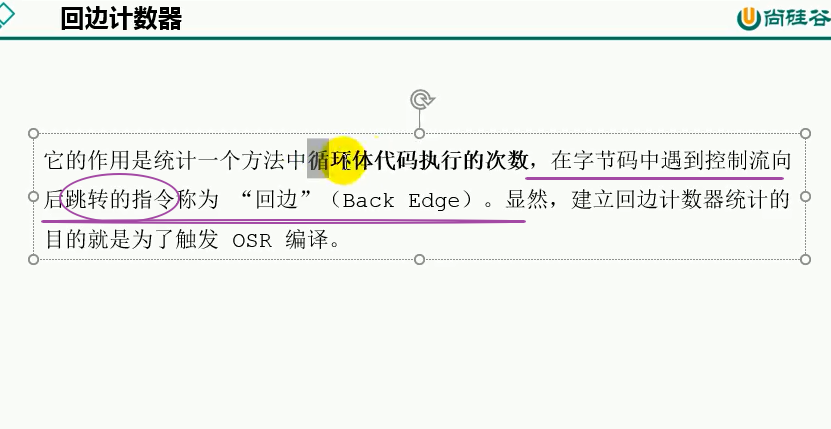
* 热点探测技术

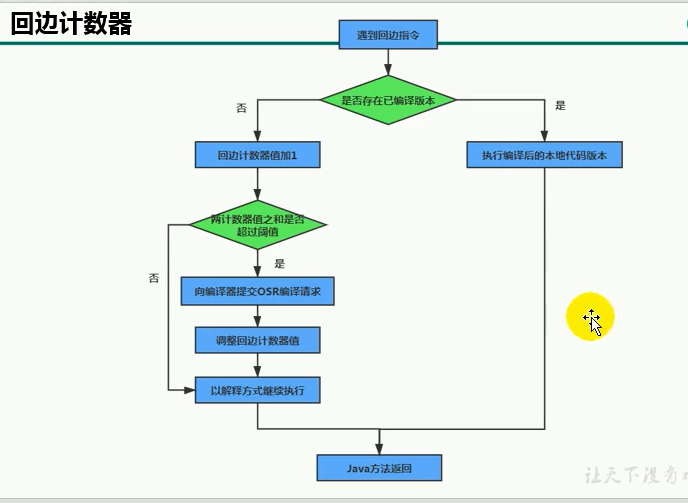
目前HotSpot VM所采用的热点探测方式是**基于计数器的热点探测**。

采用基于计数器的热点探测，HotSpot VM将会为每一个方法都建立2个不同类型的计数器，分别为**方法调用计数器（Invocation Counter）**和**回边计数器（Back Edge Counter）**。

* 方法调用计数器用于统计方法的调用次数
* 回边计数器则用于统计循环体执行的循环次数

### 回边计数器





### 方法调用计数器

这个计数器就用于统计方法被调用的次数，它的默认阀值在Client模式下是1500次，在Server模式下是10000次。超过这个阈值，就会触发JIT编译。

这个阀值可以通过虚拟机参数 **-XX:CompileThreshold** 来人为设定。

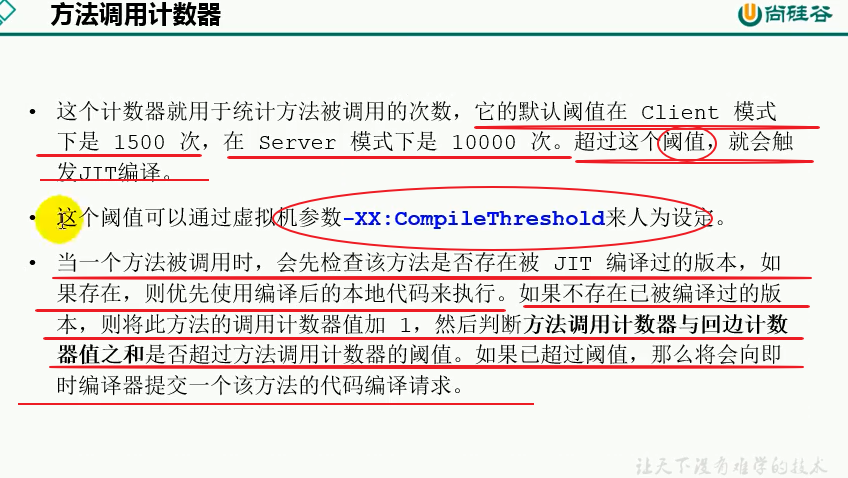
当一个方法被调用时，会先检查该方法是否存在被JIT编译过的版本

如果存在，则优先使用编译后的本地代码来执行

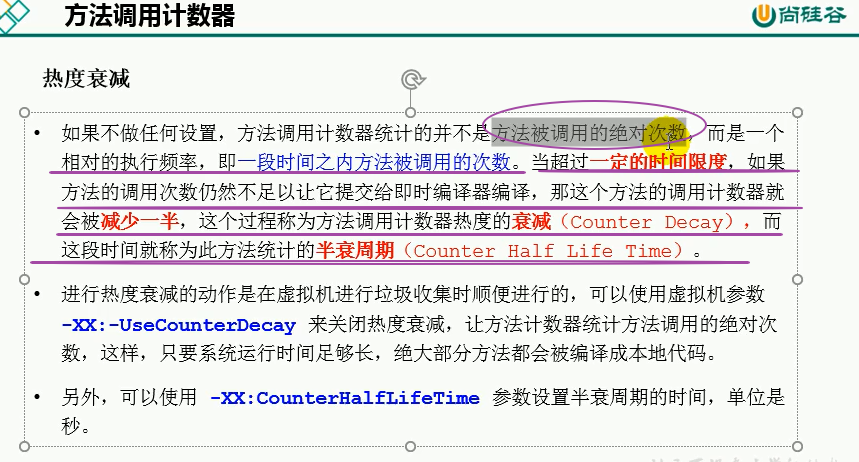
如果不存在已被编译过的版本，则将此方法的调用计数器值加1，然后判断方法调用计数器与回边计数器值之和是否超过方法调用计数器的阀值。

如果已超过阈值，那么将会向即时编译器提交一个该方法的代码编译请求。

如果未超过阈值，则使用解释器对字节码文件解释执行



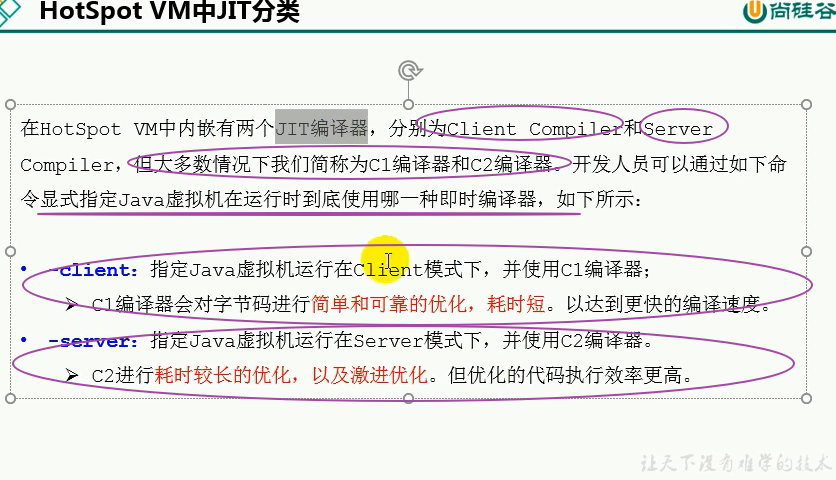




### HotSpot 虚拟机设置执行方式

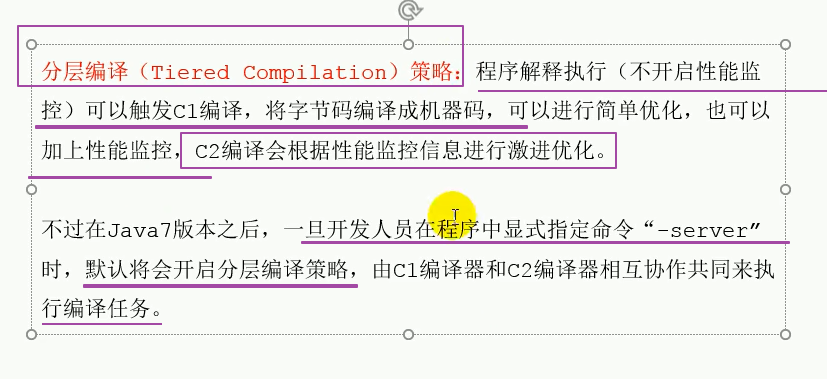


HotSpot VM中 JIT的分类

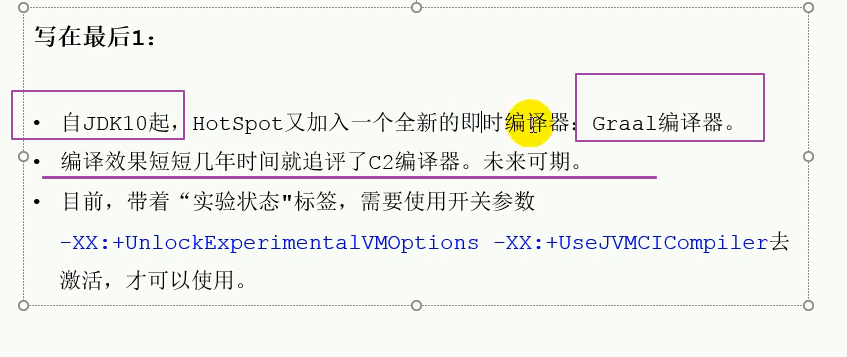


64位操作系统，JVM只能是server端 。





Graal编译器是C2编译器



AOT和JIT编译器是同等级

