

【Java 技术之旅】彻底你明白什么是 JIT 编译器（Just In Time 编译器）



计算机视觉入门
分享个人java架构知识，共同学习，一起进步！

5 人赞同了该文章

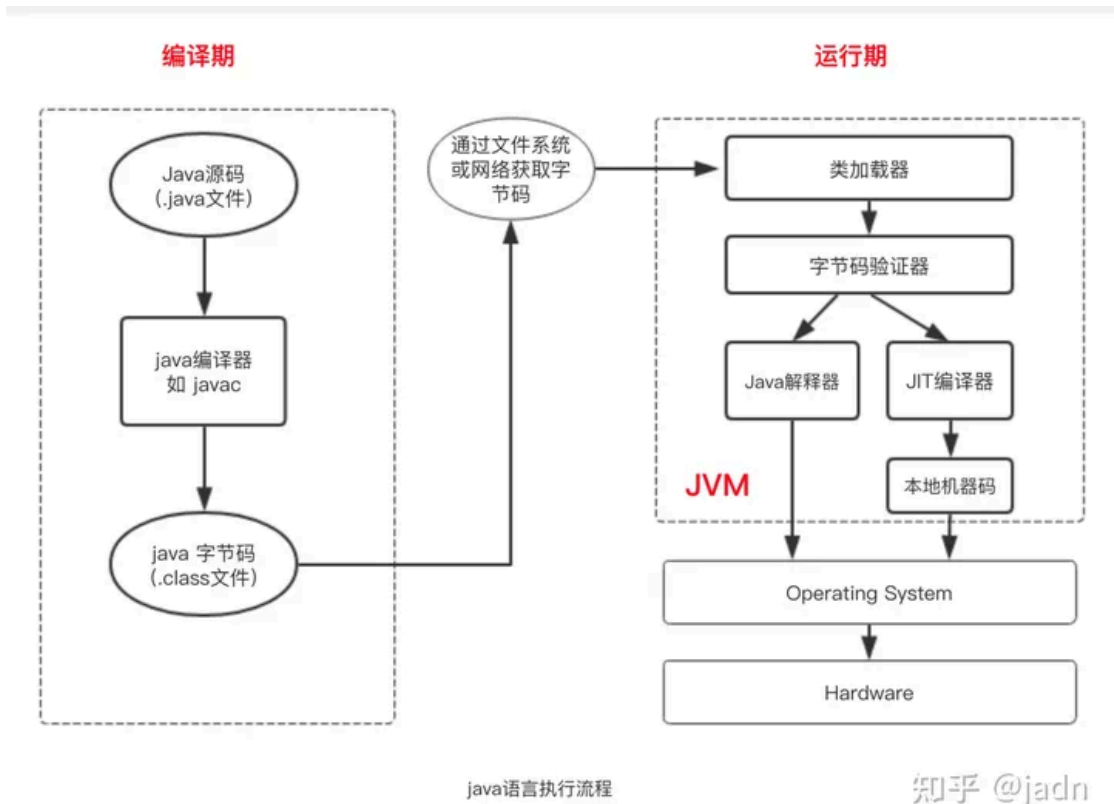


前提概要

- 我们都知道开发语言整体分为两类，一类是编译型语言，一类是解释型语言。那么你知道二者有何区别吗？编译器和解释器又有什么区别？
- 这是为了兼顾启动效率和运行效率两个方面。Java 程序最初是通过解释器进行解释运行的，当虚拟机返现某个方法或代码块的运行特别频繁时，就会把这段代码标记为热点代码，为了提供热点代码的运行效率，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。

编译器和解释器

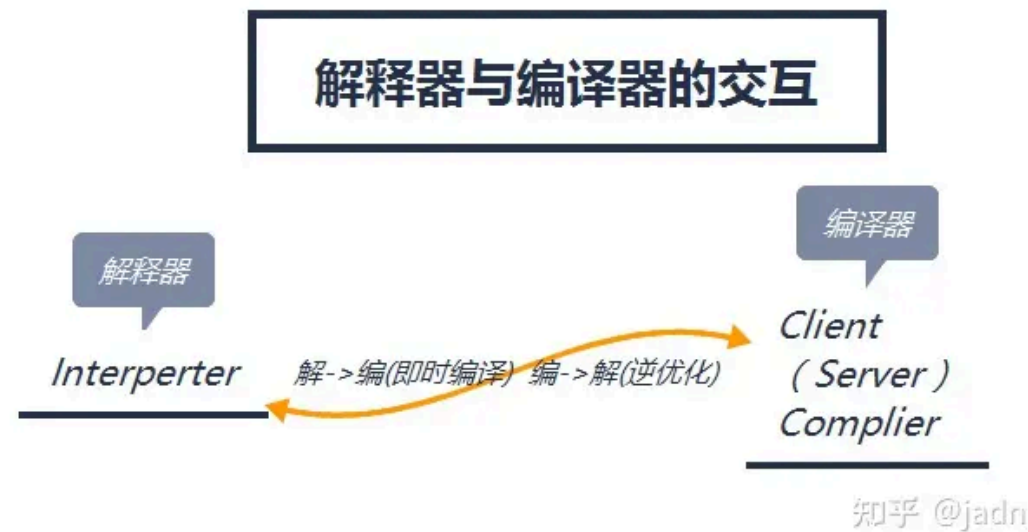
- Java 编译器 (javac) 的作用是将 java 源程序编译成中间代码字节码文件，是最基本的开发工具。
- Java 解释器 (java) (英语：Interpreter)，又译为直译器，是一种电脑程序，能够把高级编程语言一行一行直接转译运行。解释器不会一次把整个程序转译出来，只像一位“中间人”，每次运行程序时都要先转成另一种语言再作运行，因此解释器的程序运行速度比较缓慢。它每转译一行程序叙述就立刻运行，然后再转译下一行，再运行，如此不停地进行下去。



1. 当程序需要首次启动和执行的时候，解释器可以首先发挥作用，一行一行直接转译运行，但效率低下。



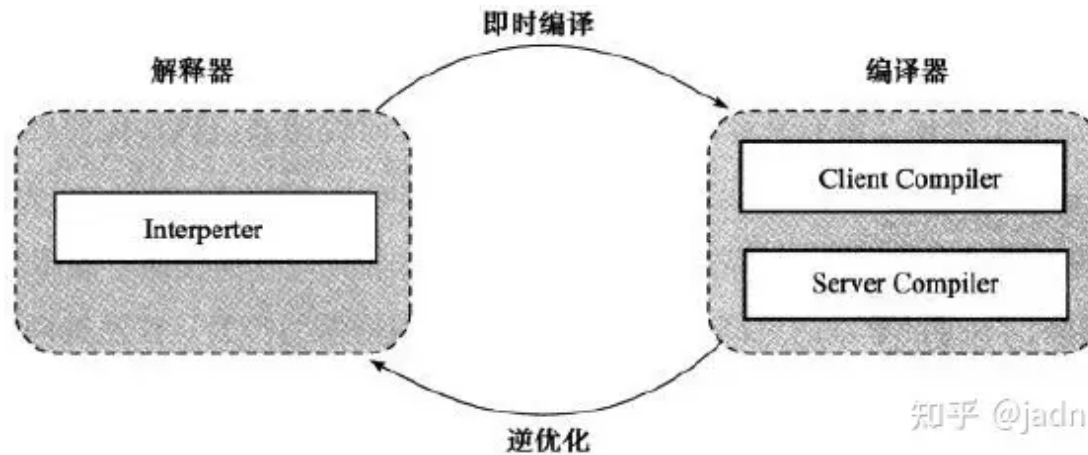
解释器与编译器的交互：



HotSpot 虚拟机中内置了两个即时编译器，分别称为 Client Compiler 和 Server Compiler，它会根据自身版本与宿主机器的硬件性能自动选择运行模式，用户也可以使用"-client"或"-server"参数去强制指定虚拟机运行在 Client 模式或 Server 模式

什么是 JIT 编译器

- 即时 (Just-In-Time) 编译器是 Java 运行时环境的一个组件，它可提高运行时 Java 应用程序的性能。JVM 中没有什么比编译器更能影响性能，而选择编译器是运行 Java 应用程序时做出的首要决定之一。
- 当编译器做的激进优化不成立，如载入了新类后类型继承结构出现变化。出现了罕见陷阱时能够进行逆优化退回到解释状态继续运行。



解释器与编译器搭配使用的方式：

HotSpot JVM 内置了两个编译器，各自是 Client Compiler 和 Server Compiler，虚拟机默认是 Client 模式。我们能够通过。

- -client: 强制虚拟机运行 Client 模式
- -server: 强制虚拟机运行 Server 模式
- 默认 (java -version 混合模式)

而不管是 Client 模式还是 Server 模式，虚拟机都会运行在解释器和编译器配合使用的混合模式下。能够通过。

- 解释模式 (java -Xint -version) 强制虚拟机运行于解释模式，仅使用解释器方式执行。
- 编译模式 (java -Xcomp -version) 优先采用编译方式执行程序，但解释器要在编译无法进行的情况下介入执行过程。

```
java -version
java version "1.8.0_121"
Java(TM) SE Runtime Environment (build 1.8.0_121-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode)
```

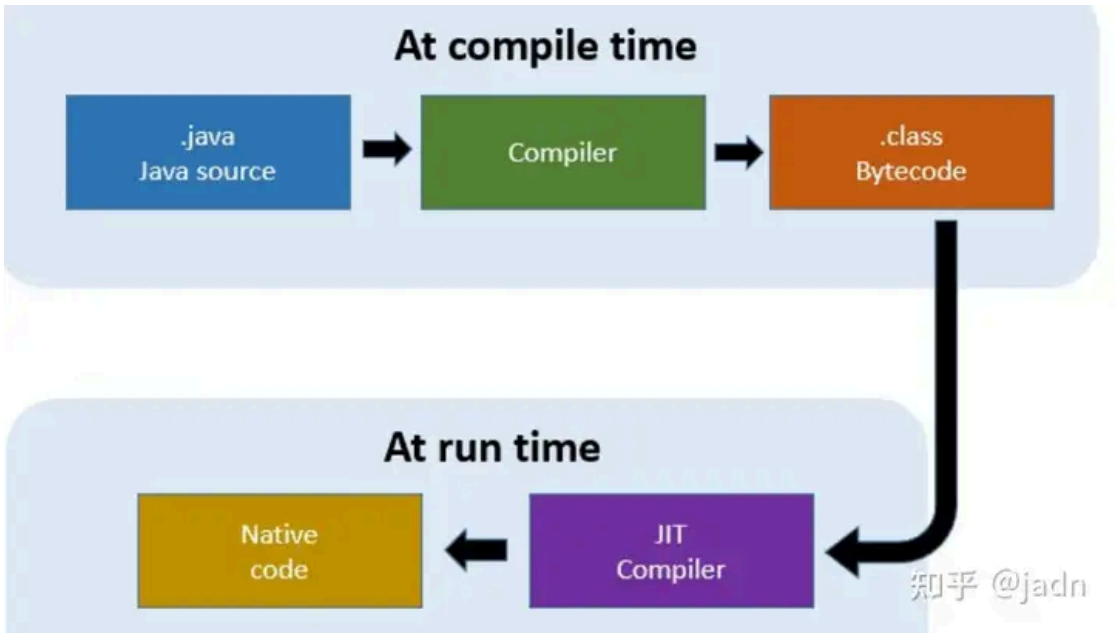
```
java -Xint -version
java version "1.8.0_121"
```

```
java -Xcomp -version
java version "1.8.0_121"
Java(TM) SE Runtime Environment (build 1.8.0_121-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, compiled mode)
```

Java 功能“一次编译，到处运行”的关键是 **bytecode**。字节码转换为应用程序的机器指令的方式对应用程序的速度有很大的影响。这些字节码可以被解释，编译为本地代码，或者直接在指令集架构中符合字节码规范的处理器上执行。

- 解释字节码的是 **Java 虚拟机（JVM）** 的标准实现，这会使程序的执行速度变慢。为了提高性能，**JIT 编译器**在运行时与 **JVM** 交互，并将适当的字节码序列编译为本地机器代码。
- 使用 **JIT 编译器**时，硬件可以执行本机代码，而不是让 **JVM** 重复解释相同的字节码序列，并导致翻译过程相对冗长。这样可以提高执行速度，除非方法执行频率较低。
- **JIT 编译器**编译字节码所花费的时间被添加到总体执行时间中，并且如果不频繁调用 **JIT 编译**的方法，则可能导致执行时间比用于执行字节码的解释器更长。
- 当将字节码编译为本地代码时，**JIT 编译器**会执行某些优化。
- 由于 **JIT 编译器**将一系列字节码转换为本机指令，因此它可以执行一些简单的优化。
- **JIT 编译器**执行的一些常见优化操作包括数据分析，从堆栈操作到寄存器操作的转换，通过寄存器分配减少内存访问，消除常见子表达式等。
- **JIT 编译器**进行的优化程度越高，在执行阶段花费的时间越多。

因此，**JIT 编译器**无法承担所有静态编译器所做的优化，这不仅是因为增加了执行时间的开销，而且还因为它只对程序进行了限制。



- **JIT 编译器**默认情况下处于启用状态，并在调用 **Java** 方法时被激活。
- **JIT 编译器**将该方法的字节码编译为本地机器代码，“即时”编译以运行。
- 编译方法后，**JVM** 会直接调用该方法的已编译代码，而不是对其进行解释。

从理论上讲，如果编译不需要处理器时间和内存使用量，则编译每种方法都可以使 **Java** 程序的速度接近本机应用程序的速度。

JIT 编译确实需要处理器时间和内存使用率。**JVM** 首次启动时，将调用数千种方法。即使程序最终达到了非常好的峰值性能，编译所有这些方法也会严重影响启动时间。

不同应用程序的不同编译器

JIT 编译器有两种形式，并且选择使用哪个编译器通常是运行应用程序时唯一需要进行的编译器调整。实际上，即使在安装 **Java** 之前，也要考虑知道要选择哪个编译器，因为不同的 **Java** 二进制文件包含不同的编译器。

客户端编译器

著名的优化编译器是 **C1**，它是通过 **-clientJVM** 启动选项启用的编译器。顾名思义，**C1** 是客户端编译器。它是为客户端应用程序设计的，这些客户端应用程序具有较少的可用资源，并且在

服务器端编译器

对于长时间运行的应用程序（例如服务器端企业 Java 应用程序），客户端编译器可能不够。可以使用类似 C2 的服务器端编译器。通常通过将 JVM 启动选项添加-server 到启动命令行来启用 C2 。由于大多数服务器端程序预计将运行很长时间，因此启用 C2 意味着您将能够比使用运行时间短的轻量级客户端应用程序收集更多的性能分析数据。因此，您将能够应用更高级的优化技术和算法。

分层编译

为什么要进行分层编译

这是由于编译器编译本机代码须要占用程序运行时间，要编译出优化程度更高的代码锁花费的时间可能更长，并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。这对解释运行的速度也有影响。为了在程序启动响应速度和运行效率之间寻找平衡点。因此採用分层编译的策略。

- 分层编译结合了客户端和服务端编译。分层编译利用了 JVM 中客户端和服务端编译器的优势。
- 客户端编译器在应用程序启动期间最活跃，并处理由较低的性能计数器阈值触发的优化。
- 客户端编译器还会插入性能计数器，并为更高级的优化准备指令集，服务端编译器将在稍后阶段解决这些问题。

分层编译是一种非常节省资源的性能分析方法，因为编译器能够在影响较小的编译器活动期间收集数据，以后可以将其用于更高级的优化。与仅使用解释的代码配置文件计数器所获得的信息相比，这种方法还可以产生更多的信息。

分层策略例如以下所看到的：

- 第 0 层：程序解释运行。解释器不开启性能监控功能，可触发第 1 层编译。
- 第 1 层：即 C1 编译。将字节码编译为本地代码。进行简单和可靠的优化，如有必要将增加性能监控的逻辑。
- 第 2 层：即 C2 编译，将字节码编译为本地代码，同一时候启用一些编译耗时较长的优化，甚至会依据性能监控信息进行一些不可靠的激进优化。

代码优化

- 当选择一种方法进行编译时，JVM 会将其字节码提供给即时编译器（JIT）。JIT 必须先了解字节的语义和语法，然后才能正确编译该方法。
- 为了帮助 JIT 编译器分析该方法，首先将其字节码重新格式化为称为 trees，它比字节码更类似于机器代码。
- 然后对方法的树进行分析和优化。
- 最后，将树转换为本地代码。
- JIT 编译器可以使用多个编译线程来执行 JIT 编译任务，使用多个线程可以潜在地帮助 Java 应用程序更快地启动。

编译线程的默认数量由 JVM 标识，并且取决于系统配置。如果生成的线程数不是最佳的，则可以使用该 XcompilationThreads 选项覆盖 JVM 决策。

编译包括以下阶段：

内联

内联是将较小方法的树合并或“内联”到其调用者的树中的过程。这样可以加速频繁执行的方法调用。

局部优化

局部优化可以一次分析和改进一小部分代码。许多本地优化实现了经典静态编译器中使用的久经考验的技术。

控制流优化

控制流优化分析方法（或方法的特定部分）内部的控制流，并重新排列代码路径以提高其效率。

全局优化可一次对整个方法起作用。它们更加“昂贵”，需要大量的编译时间，但可以大大提高性能。

本机代码生成

本机代码生成过程因平台架构而异。通常，在编译的此阶段，将方法的树转换为机器代码指令；根据架构特征执行一些小的优化。

编译对象

编译对象即为会被编译优化的热点代码。有下面两类：

- 被多次调用的方法
- 被多次运行的循环体

触发条件

这就牵扯到触发条件这个概念，推断一段代码是否是热点代码。是否须要触发即时编译，这样的行为成为热点探测（Spot Detection）。

热点探测有两种手段：

基于采样的热点探测（Sample Based Hot Spot Detection）

虚拟机会周期性的检查各个线程的栈顶，假设发现某些方法常常性的出如今栈顶，那么这种方法就是热点方法。

基于计数器的热点探测（Counter Based Hot Spot Detection）

虚拟机会为每一个方法或代码块建立计数器，统计方法的运行次数。假设运行次数超过一定的阈值就觉得他是热点方法。

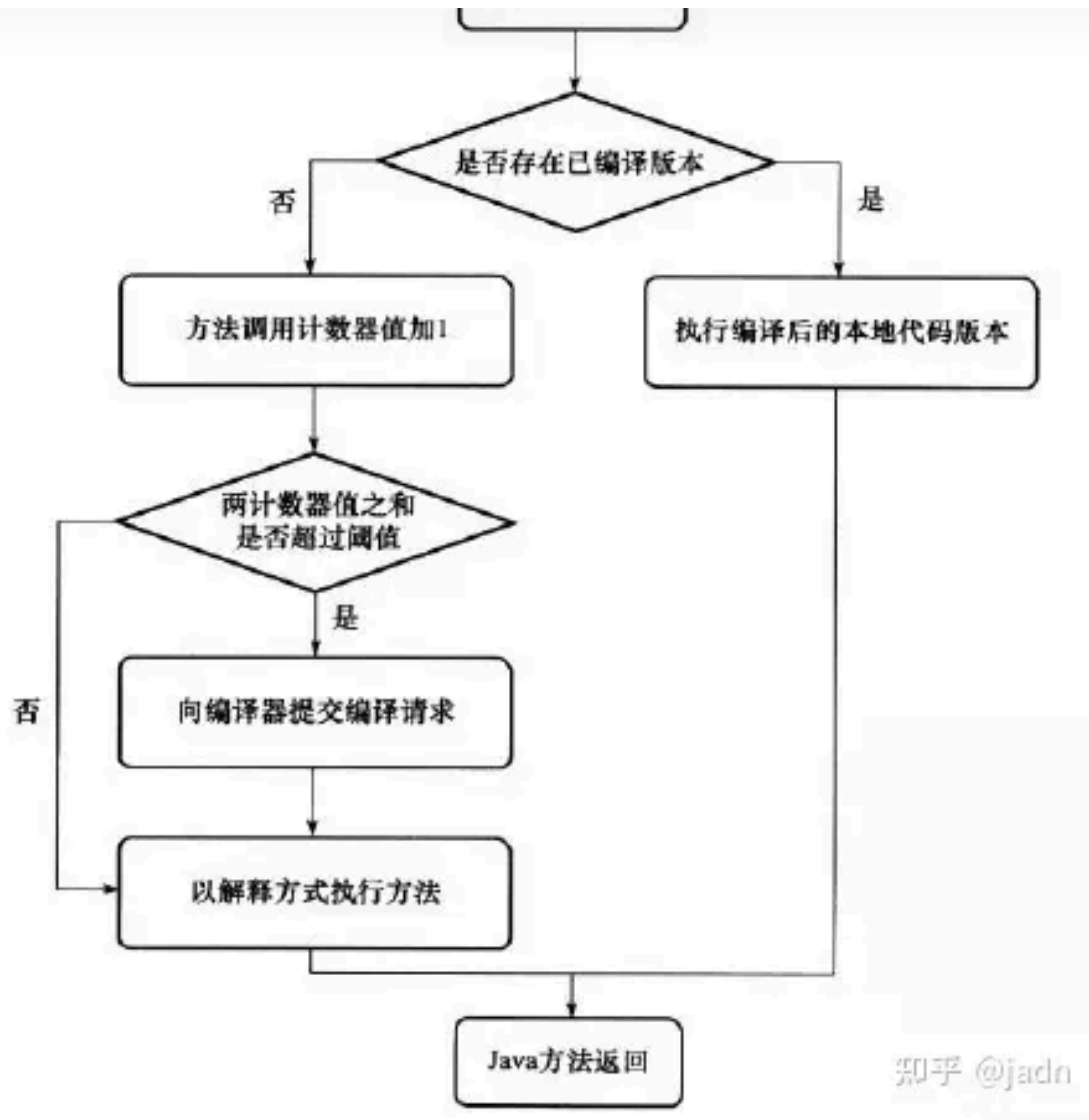
HotSpot JVM 使用另外一种方法基于计数器的热点探测方法。它为每一个方法准备了两类计数器：

方法调用计数器

这个阈值在 Client 模式下是 1500 次。在 Server 模式下是 10000 此，这个阈值能够通过参数 `-XX:CompileThreshold` 来人为设定。

- 方法调用次数统计的并非方法被调用的绝对次数，而是相对的运行频率，即一段时间内方法被调用的次数，当超过一定时间限度，假设方法的调用次数仍然不足以让它提交给即时编译器编译，那这种方法的调用计数器会被降低一半，这个过程被称为方法调用计数器的热度衰减（Counter Decay）。
- 而这段时间就称为此方法统计的半衰周期（Counter Half Life Time）。相同也能够使用参数 `-XX:-UseCounterDecay` 来关闭热度衰减。

方法调用计数器触发即时编译的整个流程例如以下图所看到的：



回边计数器

什么是回边？

在字节码遇到控制流向后跳转的指令称为回边（Back Edge）。

- 回边计数器是用来统计一个方法中循环体代码运行的次数，回边计数器的阈值能够通过参数 -XX: OnStackReplacePercentage 来调整。

虚拟机运行在 Client 模式下，回边计数器阈值计算公式为：

方法调用计数器闭值(CompileThreshold) xOSR比率(OnStackReplacePercentage) / 100

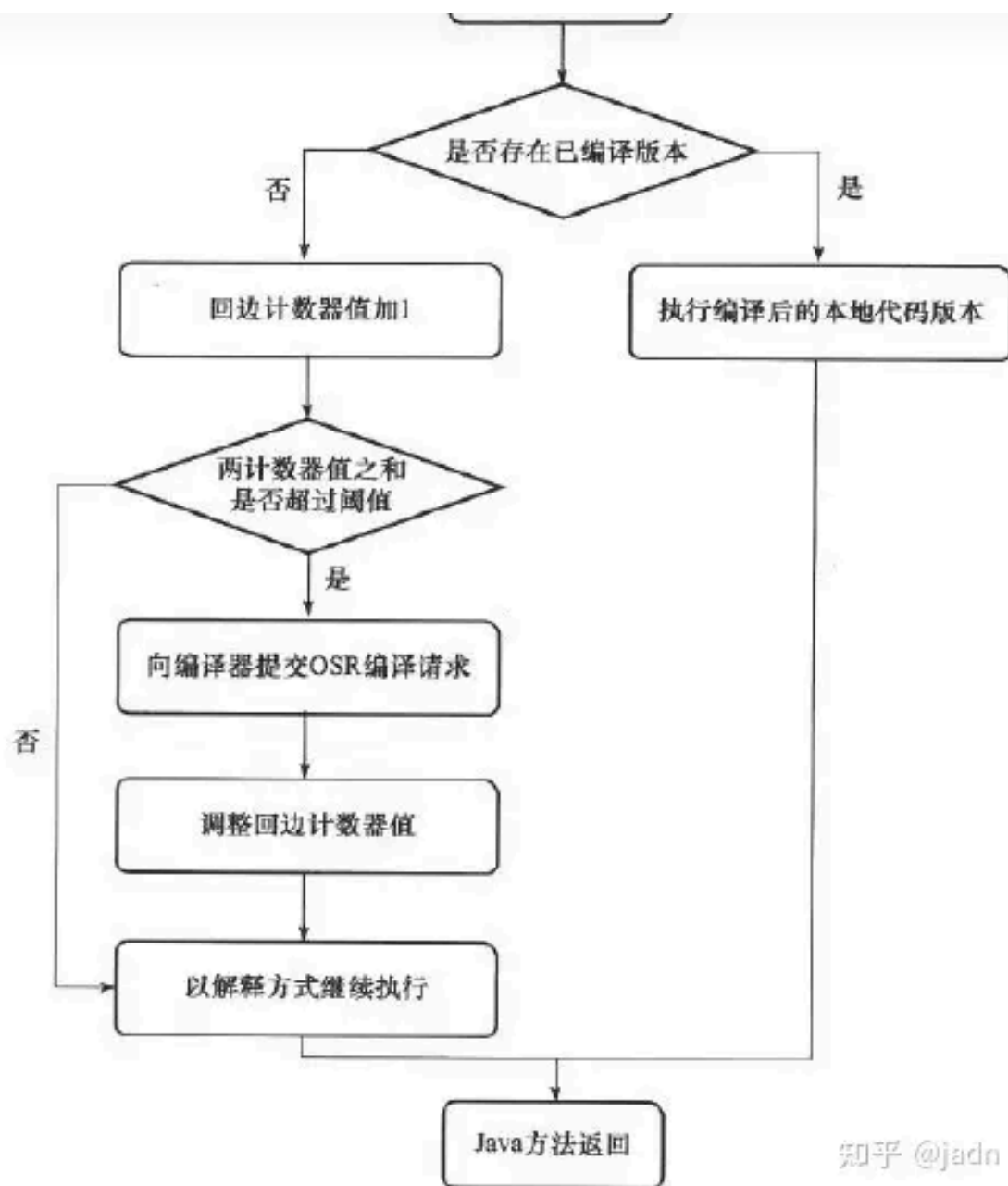
当中 OnSlackReplacePercentage 默认值为 933，假设都取默认值，那 Client 模式虚拟机的回边计数器的阈值为 13995。

虚拟机运行在 Server 模式下，回边计数器阈值的 itm 公式为：

方法调用计数器阈值(CompileThreshold) x (OSR比率(OnStackReplacePercentage) - 解释器监控比率(InterpreterProffePercentage) / 100

- 当中 OnSlackReplacePercentage 默认值为 140。InterpreterProffePercentage 默认值为 33.
- 假设都取默认值。BF Server 模式虚拟机回边计数器的阈值为 10700。

回边计数器触发即时编译的流程例如以下图所看到的：



知乎 @jadr

回边计数器与方法调用计数器不同的是，回边计数器没有热度衰减，因此这个计数器统计的就是循环运行的绝对次数。

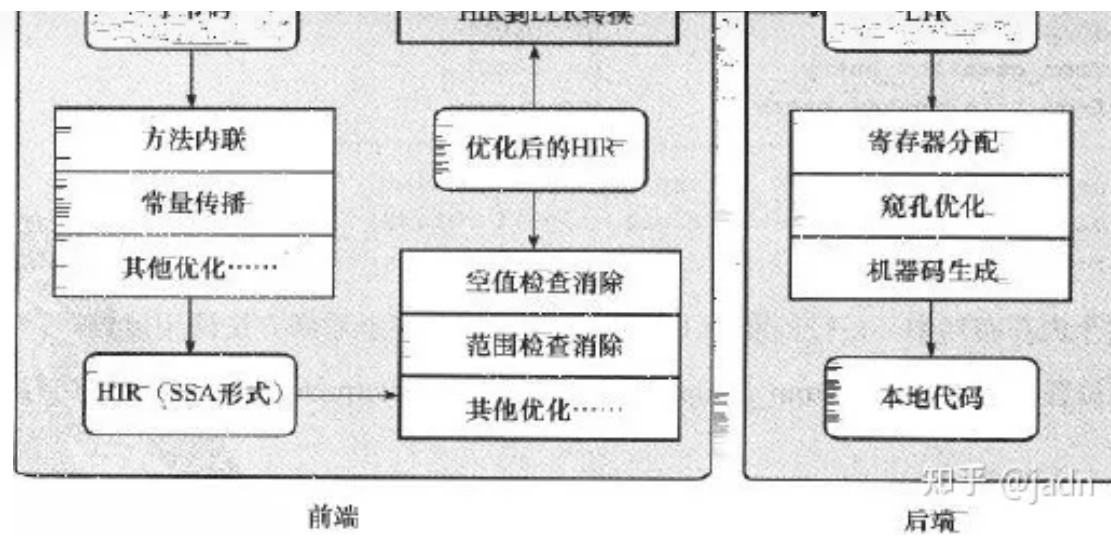
编译流程

在默认设置下，不管是方法调用产生的即时编译请求，还是 OSR 编译请求，虚拟机在代码编译器还未完毕之前，都仍然依照解释方式继续进行，而编译动作则在后台的编译线程中继续进行。也能够使用-XX:-BackgroundCompilation 来禁止后台编译，则此时一旦遇到 JIT 编译，运行线程向虚拟机提交请求后会一直等待，直到编译完后再开始运行编译器输出的本地代码。

那么在后台编译过程中，编译器做了什么事呢？

Client Compiler 编译流程

- 第一阶段：一个平台独立的前端将字节码构造成为一种高级中间码表示（High Level Intermediate Representation），HIR 使用静态单分配的形式来表示代码值，这能够使得一些的构造过程之中和之后进行的优化动作更 easy 实现，在此之前编译器会在字节码上完毕一部分基础优化，如方法内联、常量传播等。
- 第二阶段：一个平台相关的后端从 HIR 中产生低级中间代码表示（Low Level Intermediate Representation），而在此之前会在 HIR 上完毕还有一些优化。如空值检查消除、范围检查消除等。以便让 HIR 达到更高效的代码表示形式。
- 第三阶段：在平台相关的后端使用线性扫描算法（Linear Scan Register Allocation）在 LIR 上分配寄存器,并在 LIR 上做窥孔优化（Peephole）优化，然后产生机器码。



发布于 2021-10-24 15:27

解释器和JIT编译器的区别

解释器是一行一行解释执行代码，意味着下次执行还得原来流程执行一遍，如果执行的代码频次高效率就会比较低，JIT就是第一次执行的时候就翻译成了机器码，下次再运行时，直接运行机器码即可，如果代码调用频次高，那么首次翻译成机器码那点时间基本可忽略，效率自然提升。

注意字节码和机器码的区别：机器码是直接可以被计算机执行的语言，而字节码只是java编译后的文件不能被计算机直接执行

深入理解执行引擎，解释器、JIT即时编译器

原创

huisheng_qaq

已于 2024-05-10 09:23:22 修改

版权

阅读量8.6k

收藏 28

点赞数 23

分类专栏：

jvm

文章标签：


jvm

JIT即时编译器

解释器

热点探测

执行引擎



jvm 专栏收录该内容

9 订阅

13 篇文章

已订阅

JVM系列整体栏目

内容	链接地址
【一】初识虚拟机与java虚拟机	https://blog.csdn.net/zhenghuishengq/article/details/129544460
【二】jvm的类加载子系统以及jclasslib的基本使用	https://blog.csdn.net/zhenghuishengq/article/details/129610963
【三】运行时私有区域之虚拟机栈、程序计数器、本地方法栈	https://blog.csdn.net/zhenghuishengq/article/details/129684076
【四】运行时数据区共享区域之堆、逃逸分析	https://blog.csdn.net/zhenghuishengq/article/details/129796509
【五】运行时数据区共享区域之方法区、常量池	https://blog.csdn.net/zhenghuishengq/article/details/129958466
【六】对象实例化、内存布局和访问定位	https://blog.csdn.net/zhenghuishengq/article/details/130057210
【七】执行引擎，解释器、JIT即时编译器	https://blog.csdn.net/zhenghuishengq/article/details/130088553

深入理解执行引擎，解释器、JIT即时编译器

一，深入理解执行引擎

- 1，执行引擎的概述
- 2，Java代码编译和执行的过程

2.1，解释器和编译器

2.2，机器码、指令、汇编语言、高级语言

2.3，解释器和编译器工作机制（重点）

2.4，JIT编译器的热点代码和热点探测

2.5，方法调用计数器和回边计数器

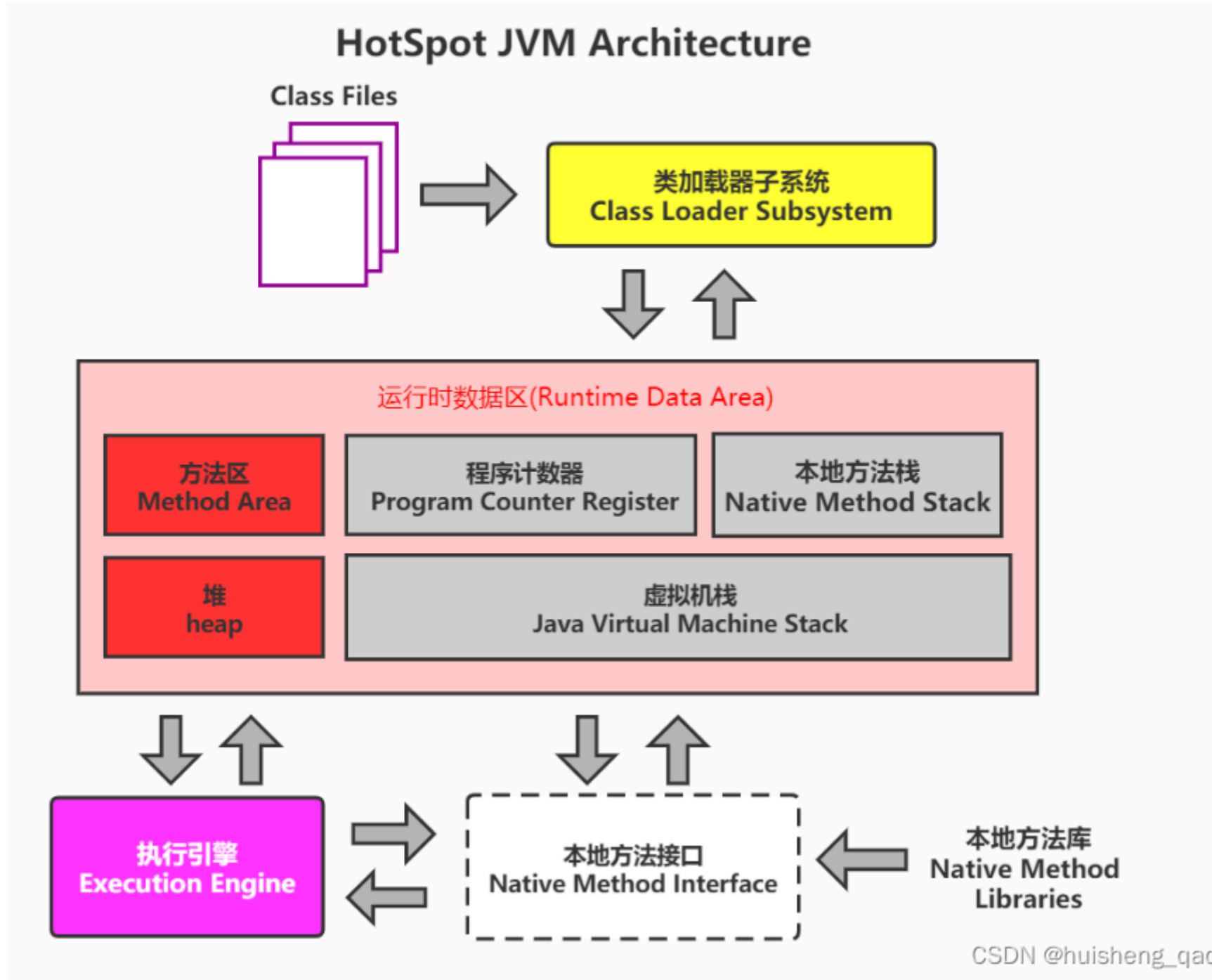
2.5.1，方法调用计数器

2.5.2，回边计数器
- 2.6，编译器和解释器设置

一，深入理解执行引擎

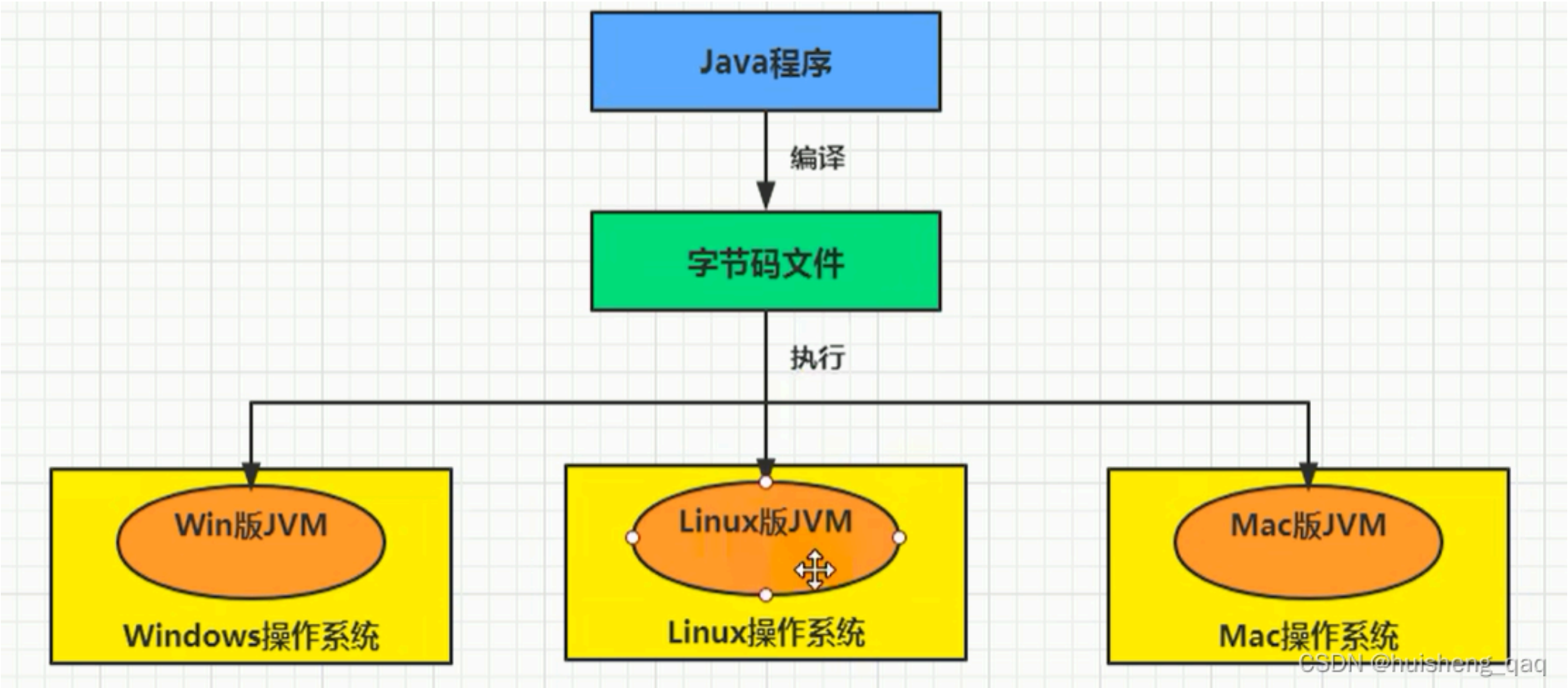
1，执行引擎的概述

在JVM整个体系中，执行引擎属于第三层，主要用来执行具体的字节码文件。本文主要探讨的就是这个执行引擎。



执行引擎是 **Java虚拟机** 核心组成的一部分，“虚拟机”是一个相对于“物理机”的一个概念，这两种机器都有执行代码的能力，其区别是**物理机的执行引擎是直接建立在处理器、缓存、指令集和操作系统的层面上的，而虚拟机的执行引擎是由软件自行实现的**，因此可以不受物理条件制约的指令集与执行引擎的结构体系，能够执行那些不被硬件直接支持的格式。java虚拟机可以理解成一个抽象的计算机，相较于真正的物理机而言，java虚拟机的执行效率会略慢于物理机。

JVM的主要任务是负责装载字节码到其内部，但字节码并不能够直接运行在操作系统上面，因为字节码指令并非等价于本地机器指令，他内部包含的仅仅是一些能够被JVM识别的字节码指令等信息。如下图所示，这些字节码指令不能直接在操作系统上解释执行，而是需要现通过 **jvm虚拟机** 来执行这些字节码指令。

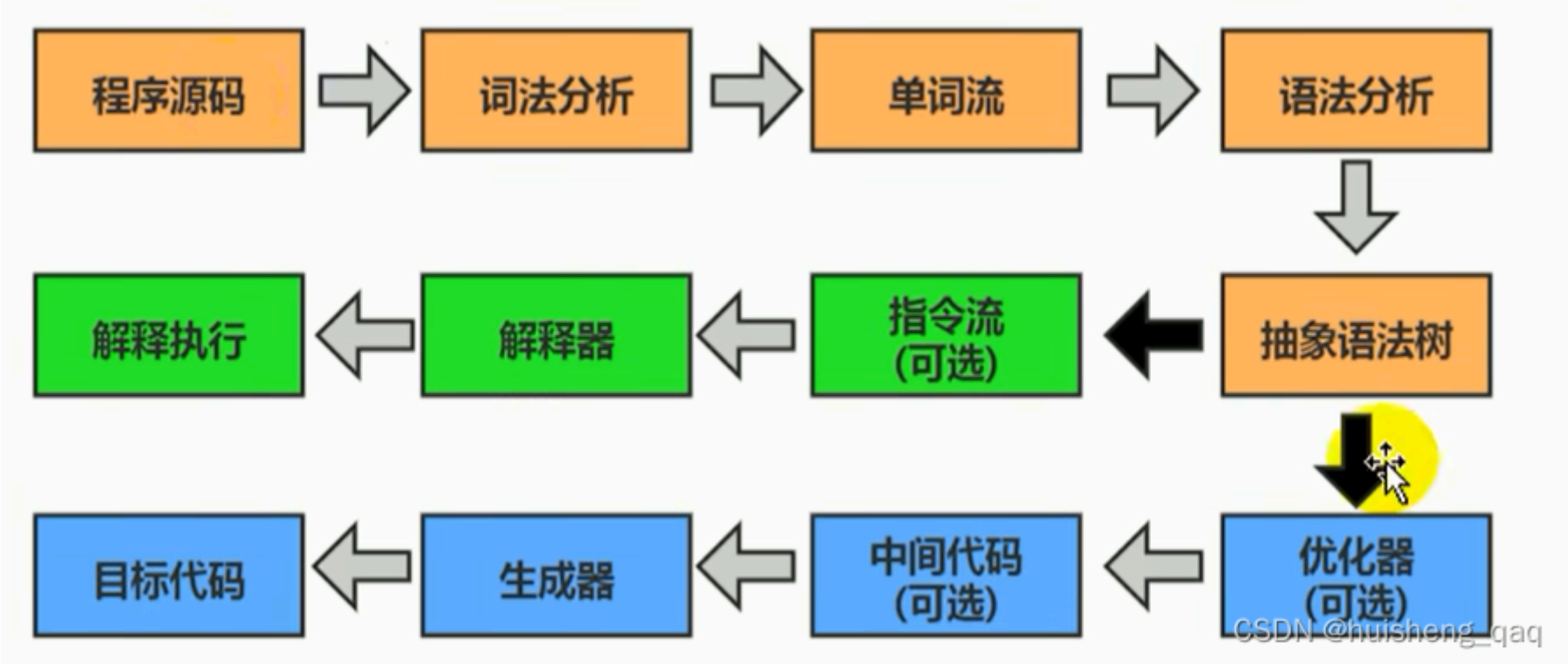


因此，执行引擎的主要作用就是：**将字节码指令解释成或者编译成对应平台上面的本地机器指令**，简单的来说，JVM中的执行引擎充当了**将高级语言翻译成机器语言的翻译者**。

执行引擎在执行过程中，其需要的具体的字节码指令**完全依赖于程序计数器**，每当完成一项操作指令之后，程序计数器就会更新下一条需要被执行的指令地址。在方法的执行全过程中，执行引擎有可能会通过存储在局部变量表的对象引用准确的获取存储在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目标对象的类型信息。

2，Java代码编译和执行的过程

大部分的程序代码在转换成物理机的目标代码或者虚拟机能执行的指令集之前，都需要经历过几下几个步骤



前面的黄线流程代表的就是将 .java 文件编译成 .class 文件，属于是前端编译；

绿色部分属于解释器解释执行的过程，即逐行翻译、解释、执行的过程；

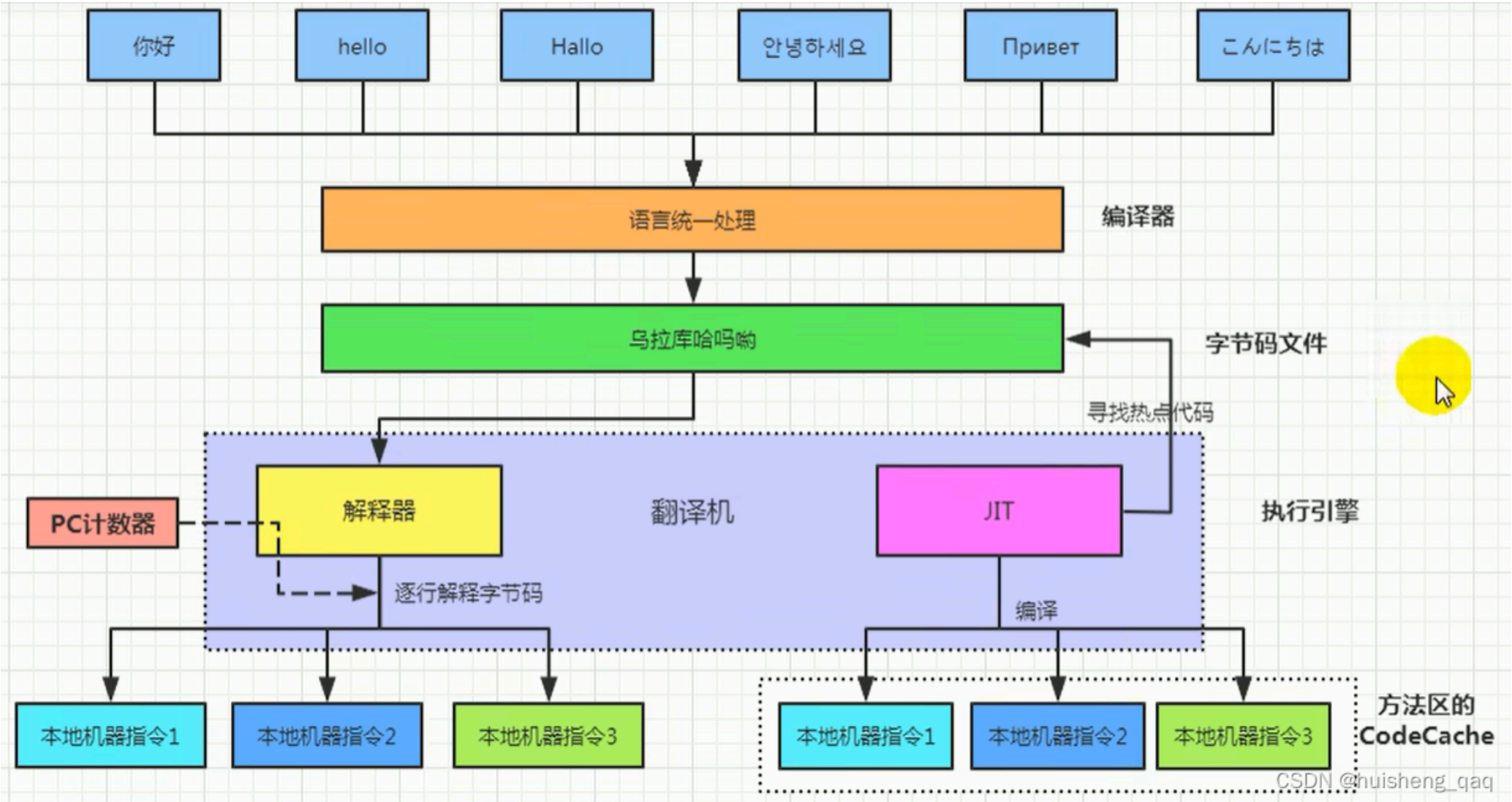
蓝色部分属于是JIT即时编译器编译性阶段，属于是后端编译。

2.1，解释器和编译器

解释器：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，将每条字节码文件中的内容翻译成对应平台的本地机器指令

JIT编译器：jit，又名*Just In Time Compiler*，就是直接将源代码编译成和本地平台相关的机器语言。

在java语言中，是既可以通过解释器来执行代码，也可以通过编译器来执行代码的，这二者都可以达到相同的目的，并且这二者以**合作的方式相辅相成，取长补短**，以最合适的方法让Java内部执行的效率更高。JVM虚拟机不仅仅是针对于Java语言，只要遵循Jvm虚拟机规范的语言，都可以使用JVM虚拟机解释执行。



如上图，将不同的语言通过统一处理，生成对应的字节码文件，然后通过虚拟机中的解释器或者JIT即时编译器对这些字节码进行解释执行，然后翻译成对应的字节码指令，最后将这些指令全部存储在方法区的CodeCache中。

2.2，机器码、指令、汇编语言、高级语言

1，机器码

各种用二进制编码方式表示的指令，叫做 **机器指令码**，如通过 `01010101` 这种二进制的方式进行编码，最开始人们就用它编写程序，这就是 **机器语言**。机器语言虽然可以被计算机接收，但是和人们的语言差别太大，不易被人家理解和记忆，用它变成也容易出错。用它编写的程序，一经输入计算机，CPU直接读取运行，因此和其他语言的程序，执行速度最快。机器指令和CPU紧密相关，因此不同类型的CPU所对应的机器指令也就不同。

2，指令

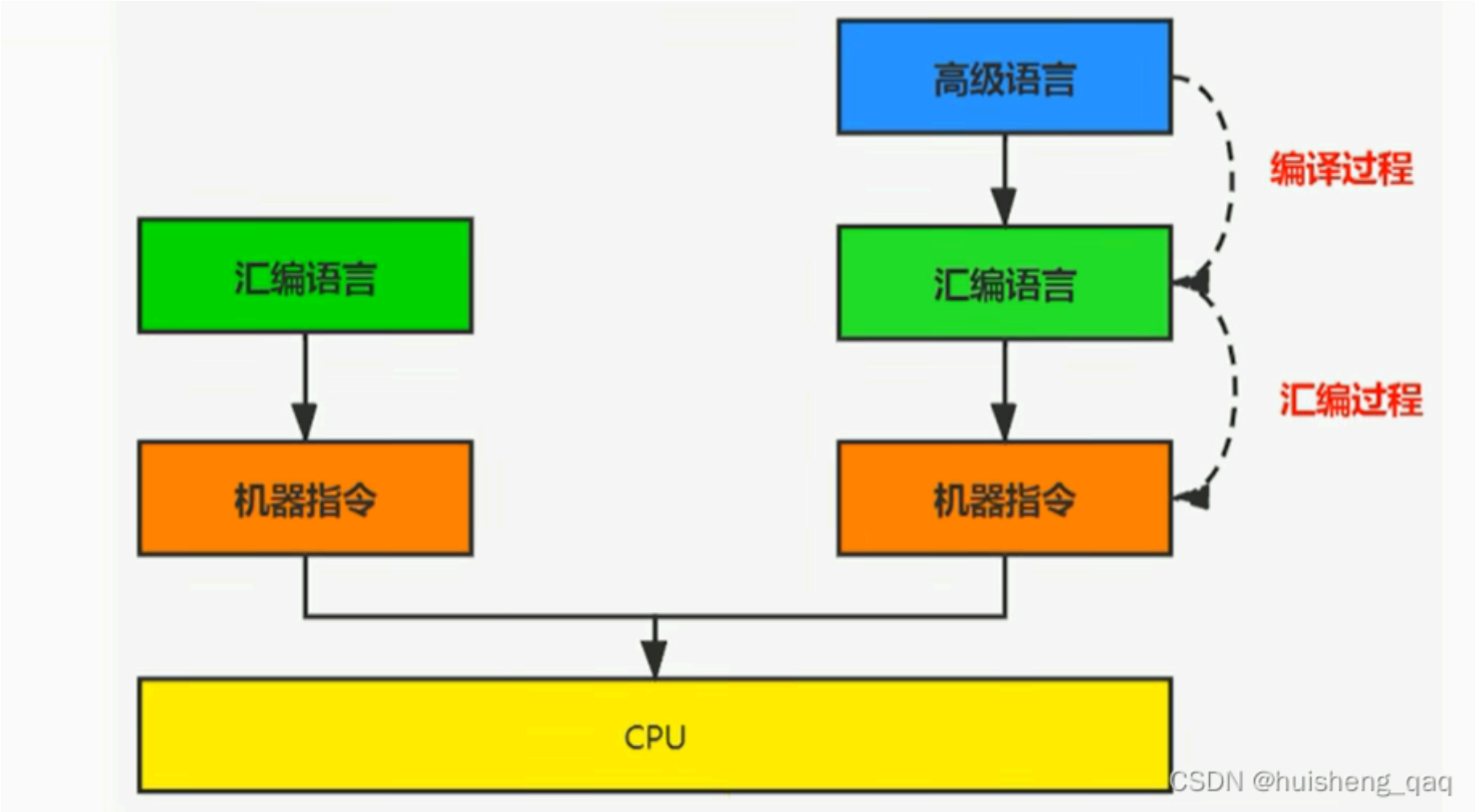
由于机器码是由0和1的二进制组成，可读性实在是太差，于是人们发明了指令。指令就是把机器码特定的 0和1 序列，简化成了对应的指令，如 `mov`和`inc`等，可读性好。但是由于不同的硬件平台，执行同一个操作，其对应的字节码可能会不同，所以不同硬件平台的同一种指令，对应的机器码也可能不同。在不同的硬件平台，各自支持各自的指令，每个平台所支持的指令总和，称之为对应平台的 **指令集**。

3，汇编语言

又由于指令的可读性差，于是又发明了这个汇编语言。在汇编语言中，用助记符代替机器指令的操作码，用地址符号代替指令或者操作数的地址。在不同的硬件平台，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。

4，高级语言

高级语言比上述语言接近人的语言，如当今流行的c或者c++，当计算机执行高级语言的时候，仍然需要把程序解释或者编译成机器指令码，完成这个过程的程序就叫做解释程序或者编译程序。因此不管是汇编语言还是这个高级语言，都需要最终生成这个机器指令，然后将这个机器指令放在CPU上面操作，最终解释执行。



字节码属于是一种中间状态的二进制代码，他比机器码更加抽象，需要直译器转译后才能成为机器码，与硬件环境无关，可以直接通过编译器或者虚拟机器，将源码编译成字节码。

2.3，解释器和编译器工作机制（重点）

解释器真正意义上所承担的角色就是一个“运行时的翻译者”，**就是将字节码中的内容翻译成对应平台的本地机器指令执行**。每当一条字节指令被解释执行完成后，接着再根据 **程序计数器** 中记录的下一条需要被执行的字节码指令执行解释操作。

在JVM平台中，也对解释器进行了优化，采用了一种**JIT** 的即时编译的技术，目的是避免函数被解释执行，而是将整个函数体编译成机器码，每次函数执行时，只执行编译后的编译码即可，这种方式大大的提升了执行效率。

在hotspot虚拟机中，JIT即时编译器的速度远快于解释器，并且将字节码指令直接生成机器指令，存储在这个方法区的CodeCache中缓存起来，**比这个解释器逐行翻译的效率**高很多。因此在今天，Java程序的运行性能早以脱胎换骨，已经可以达到和c/c++程序一较高下的地步。

但是即使这个jit即时编译器的速度很快，在HotSpot虚拟机中，依旧保留了这个解释器，原因是JIT即时编译器虽然效率很高，但是需要一定的时间编译成机器码，才能继续工作。但是这个编译器在程序启动之后，可以立马进行工作，省去编译的时间，立即执行。

所以综上所述，在程序启动的时候JIT需要编译，那么就由解释器来执行程序，待JIT即时编译器编译成机器码之后，再由这个JIT即时编译器来完成，这样就能让整个执行引擎发挥最大的效率。因此二者合作共存才能让效率最大化。

2.4, JIT编译器的热点代码和热点探测

Java语言的编译器其实是一段不太确定的操作过程，因为他可能是一指前端编译器(编译器的前端，.java文件编译成.class文件)的过程，也可能是指后端的编译器(JIT编译器，将字节码转换成机器码)的过程，还有可能是指静态提前编译器，直接把.java文件编译成本地机器代码的过程。

在使用这个JIT编译器的时候，需要判断代码被调用执行的频率，对于需要被编译为本地代码的字节码，被称为**热点代码**，JIT编译器在运行时对那些频繁被调用的热点代码会做出深度优化，**将其直接编译为对应平台的本地机器指令**，以提升Java程序的执行性能。

热点代码：指的是一个被多次调用的方法，或者是一个方法体内部循环次数较多的循环体，都可以被称为"热点代码"。因此可以通过JIT编译器译为本地机器指令，由于这种编译方式发生在方法的执行过程中，因此也被称为栈上替换。

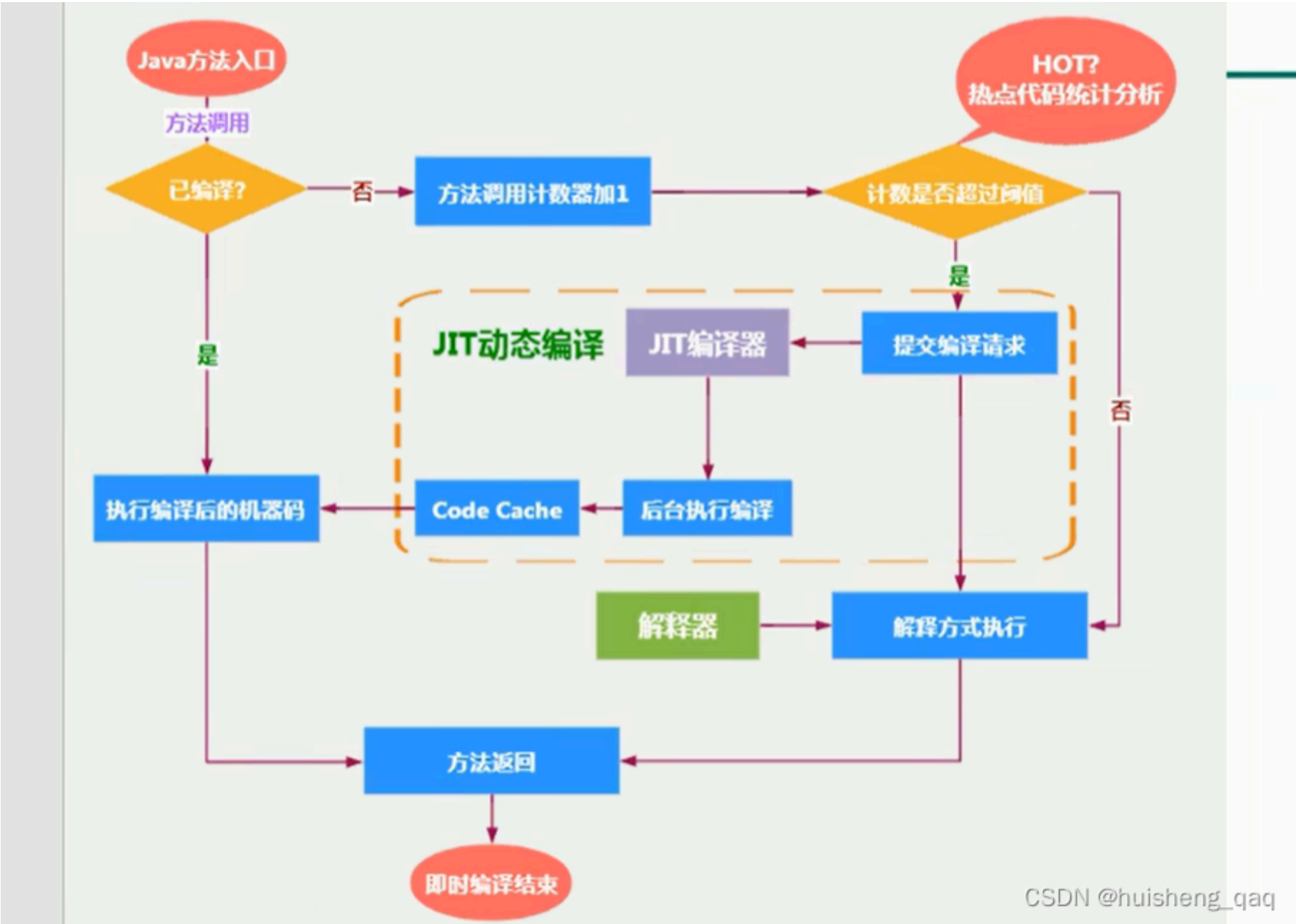
热点探测方式：而是否可以成为这个热点代码，主要是依靠这个热点探测功能，HotSpot虚拟机主要采用的热点探测方式是**基于计数器的热点探测**。HotSpot虚拟机又将每个方法建立两个不同类型的计数器，分别是**方法调用计数器**和**回边计数器**，方法调用计数器用于统计方法的调用次数，回边计数器用于统计循环体的执行次数。

2.5, 方法调用计数器和回边计数器

在JIT的热点探测中，主要是通过计数器的方式来实现对代码的探测，计数器主要分为方法调用计数器和回边计数器。

2.5.1, 方法调用计数器

这个计数器主要用于统计方法被调用的次数，它的默认阈值在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。这个阈值也可以通过虚拟机参数 `-XX:CompileThreshold` 进行设置。当一个方法被调用的时候，会先检查这个方法是否存在被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，则将此方法的调用计数器值加1，然后判断 **方法调用计数器和回边计数器** 值的和是否超过方法调用计数器的阈值，如果已经超过阈值，那么将会向即时编译器提交一个该方法的代码编译请求。



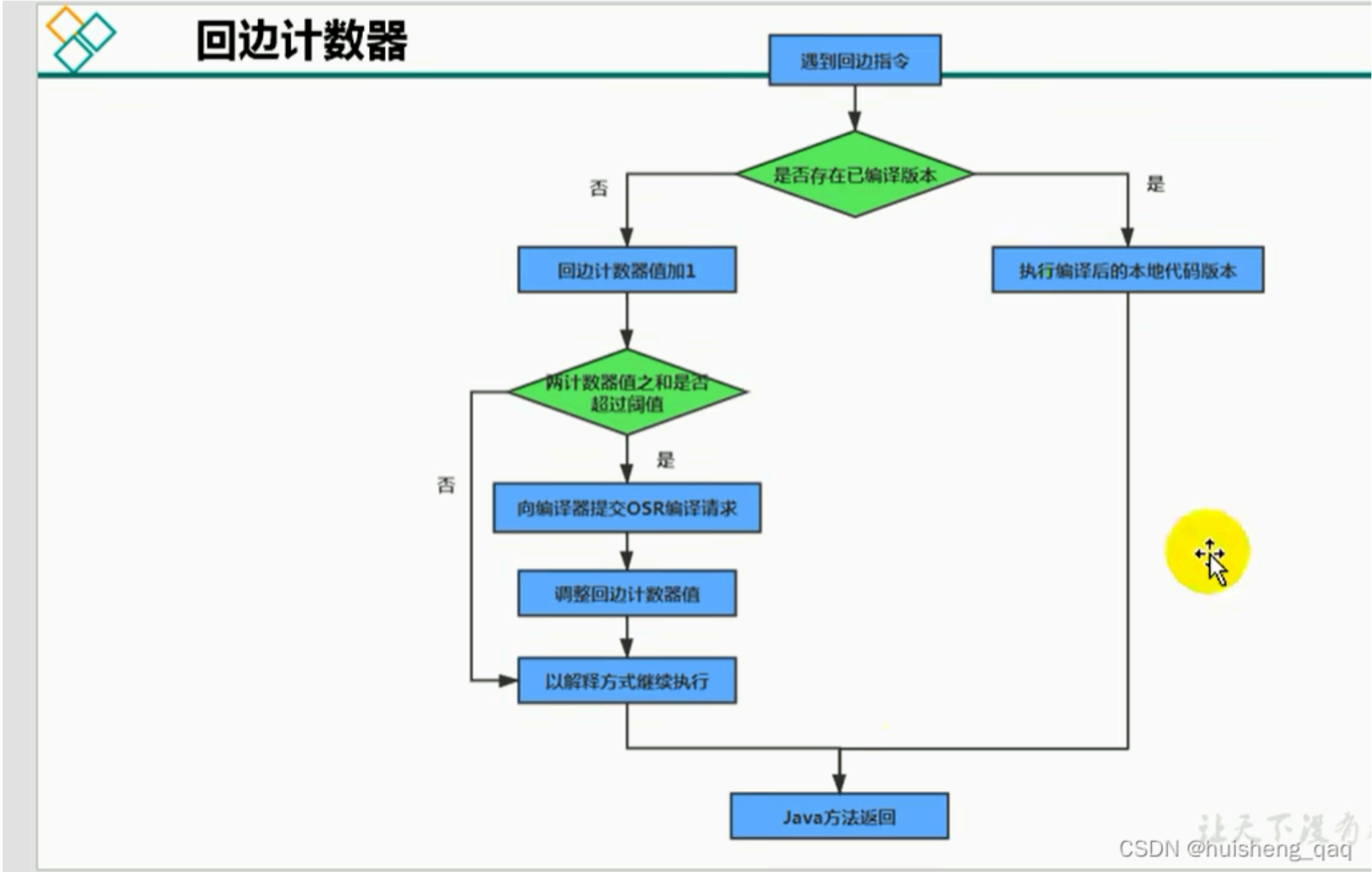
如上图所示，在调用方法时，会先判断该代码是否已经编译，如果已经编译，则直接通过这个JIT即时编译器将机器码生成对应的本地机器码指令；如果未编译，则将方法调用计数器加1，随后回去判断是否超过阈值，如果超过阈值，则会提交编译请求，通过JIT即时编译器进行动态编译，然后将编译后的机器指令缓存在CodeCache中，如果未超过阈值，那么继续通过解释器解释执行。

在JVM内部对调用的次数也做了一定的限制，并不是说一直对调用的次数进行类加，而是在一段时间内记录方法调用的次数，当超过一定的时间限度，如果方法调用的次数依旧没有达到这个阈值，那么方法的调用计数器就会进行一个 **衰减** 的过程，每次衰减一半，这段衰减的过程被称为方法统计的 **半衰周期**

进行衰减的动作是虚拟机在垃圾收集的时候顺便进行的，可以使用虚拟机参数 `-XX:-UseCounterDecay` 来关闭或者开启热度衰减，因此只要系统运行的时间足够长，那么绝大多数的方法都会编译成本地代码。同时也可以通过参数 `-XX:CounterHalfLifeTime` 设置半衰周期的时间，单位是s

2.5.2, 回边计数器

主要是统计一个方法中的循环体的执行次数，在字节码中遇到流控流向后跳转的指令称为“回边”。



和方法调用计数器一样，会先判断一下该代码是否已经编译，如果未编译，则回边计数器的值加1，然后去判断将当前累加的值和方法调用计数器的值进行累加是否超过阈值，如果超过，则使用JIT编译器，否则依旧使用解释器执行。

2.6, 编译器和解释器设置

上述可知在HotSpot虚拟机中存在解释器和编译器，如通过以下命令可以得知，当前虚拟机采用的是一种混合的方式共同执行程序。

```
1 | java -version
```

```
C:\Users\p'v>java -version
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_144-b01)
Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, inter
```

除了这种之外，也可以通过显式的命令为Java虚拟机指定只由其中一种执行程序，如可以通过以下这个命令设置只使用解释器执行程序

```
1 | java -Xint -version
```

```
C:\Users\p'v>java -Xint -version
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_144-b01)
Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, inter
```

或者可以通过以下命令只设置使用编译器来执行程序，但是如果编译出现问题，解释器会接入执行

```
1 | java -Xcomp -version
```

```
C:\Users\p'v>java -Xcomp -version
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_144-b01)
Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)
```

当然上面这两种需要在特殊的场景下使用，需要变回混合使用

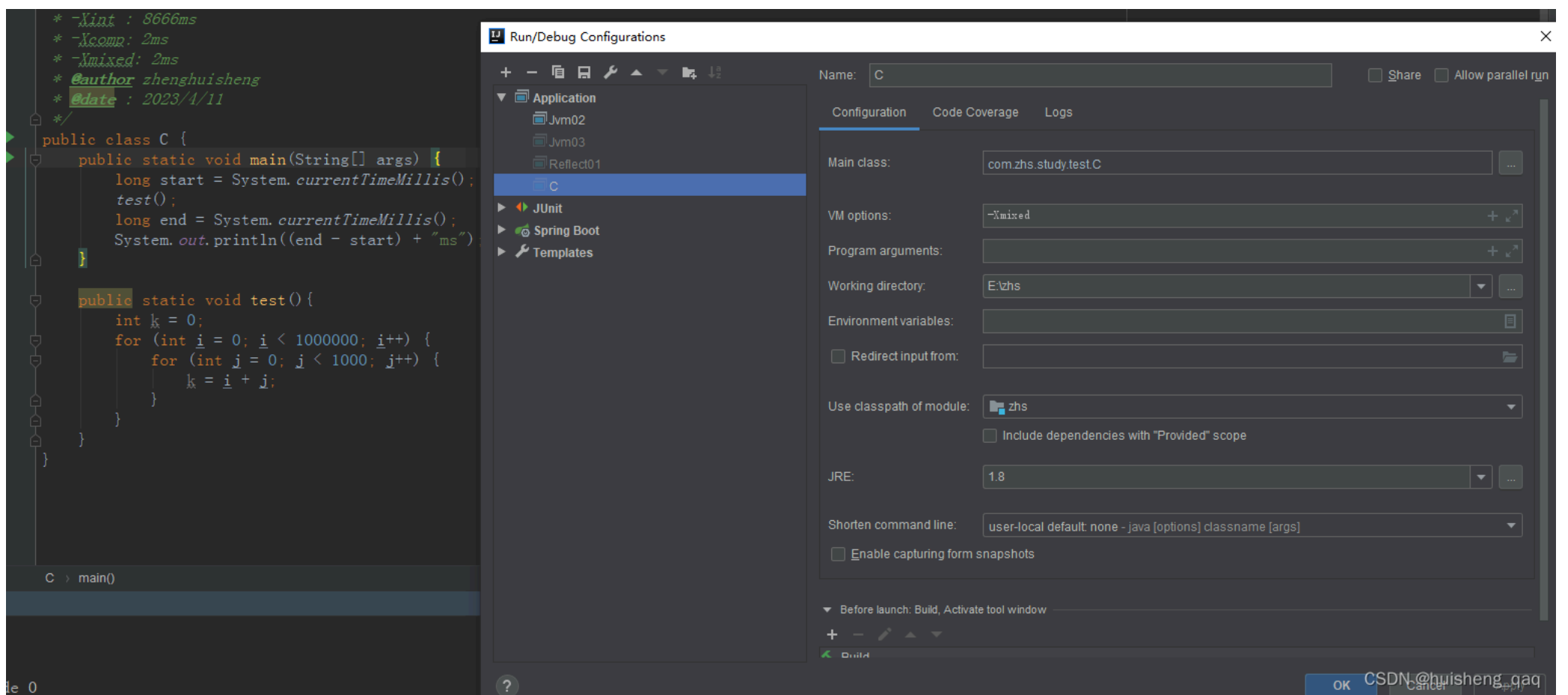
```
1 | java -Xmixed -version
```

```
C:\Users\p'v>java -Xmixed -version
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_144-b01)
Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)
```

除了可以通过这个命令行设置之外，也可以通过这个虚拟机参数就行设置，其代码如下，通过虚拟机的不同参数设置，可以得到以下答案，纯解释需要花8666ms，纯编译只需要花2ms，混合使用也是1-2ms，因此选择这个混合是最佳的，同时也可以知道使用这个纯编译器的时间远远小于这个纯解释型。

```
1  /**
2  *
3  * -Xint : 8666ms
4  * -Xcomp: 2ms
5  * -Xmixed: 2ms
6  * @author zhenghuisheng
7  * @date : 2023/4/11
8  */
9  public class C {
10     public static void main(String[] args) {
11         long start = System.currentTimeMillis();
12         test();
13     }
14 }
```

在虚拟机设置那里修改对应的参数即可。



而在HotSpot虚拟机中内嵌有两个JIT的编译器，分别是Client Compiler和Server Compiler，但是在绝大多数的情况下，这两个编译器被称为C1编译器和C2编译器。

🧠 -client：运行在Client模式下，对字节码进行可靠和简单的优化，耗时短

🧠 -server：运行在Server模式下，对字节码进行耗时长优化、激进优化，效率更高

C1编译器的优化策略主要有：**方法内联、去虚拟化、冗余消除**
C2编译器的优化策略主要有：**标量替换、栈上分配、同步消除**
