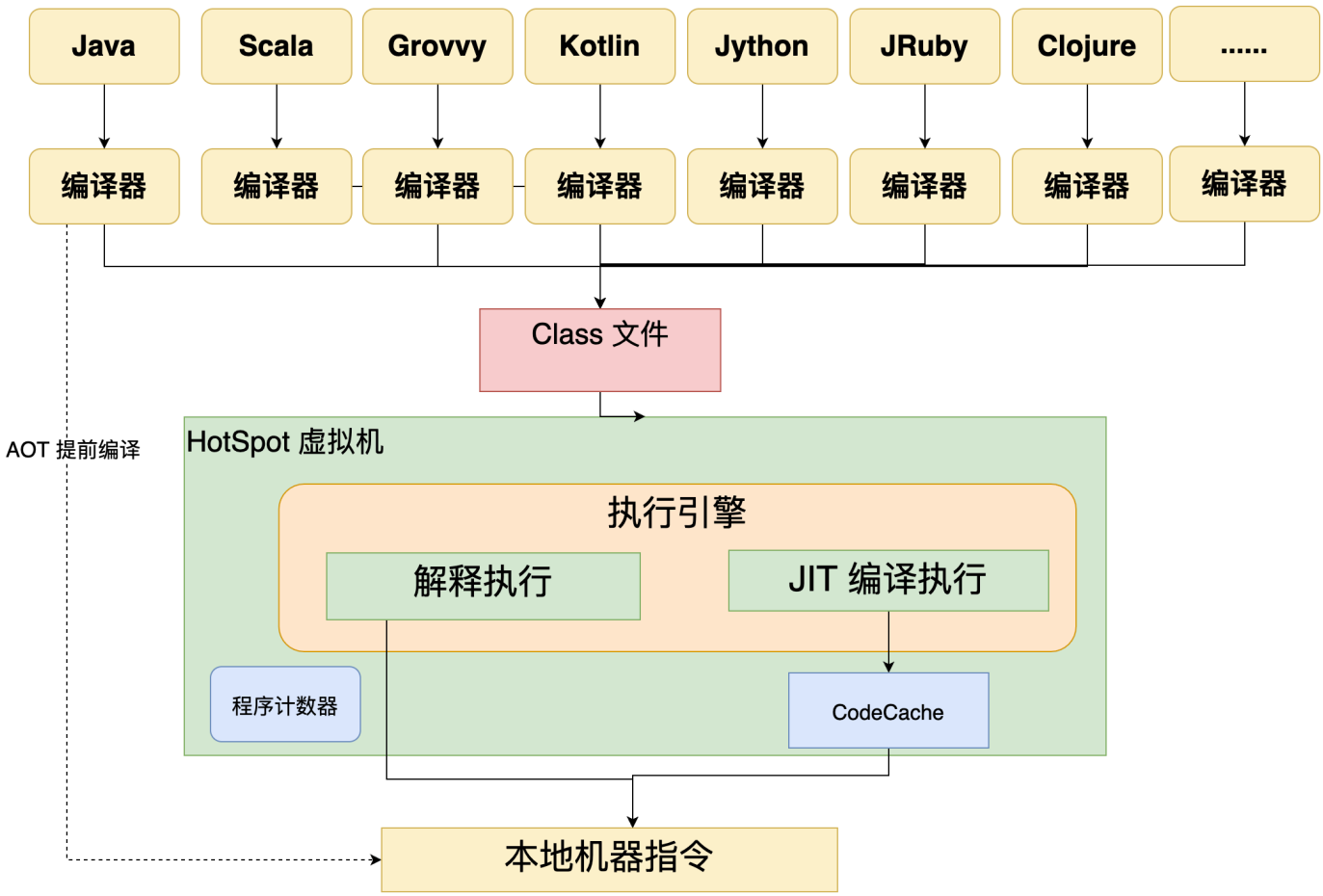
深入理解 JVM 执行引擎

-- 楼兰

从计算机程序出现的第一天起，对执行效率的追求就是程序员天生的坚定信仰。这个过程犹如一场没有终点、永不停歇的车赛。程序员是车手，而技术平台则是赛道上飞驰的赛车。今天我们就来看看，JVM 这辆赛车是如何提升执行性能的。

前端编译与后端编译

在前面部分我们已经知道，Java 程序的编译过程是分两个部分的。一个部分是从java文件编译成为class文件，这一部分也称为前端编译。另一个部分则是这些class文件，需要进入到 JVM 虚拟机，将这些字节码指令编译成操作系统识别的具体机器指令。这一部分也称为后端编译。

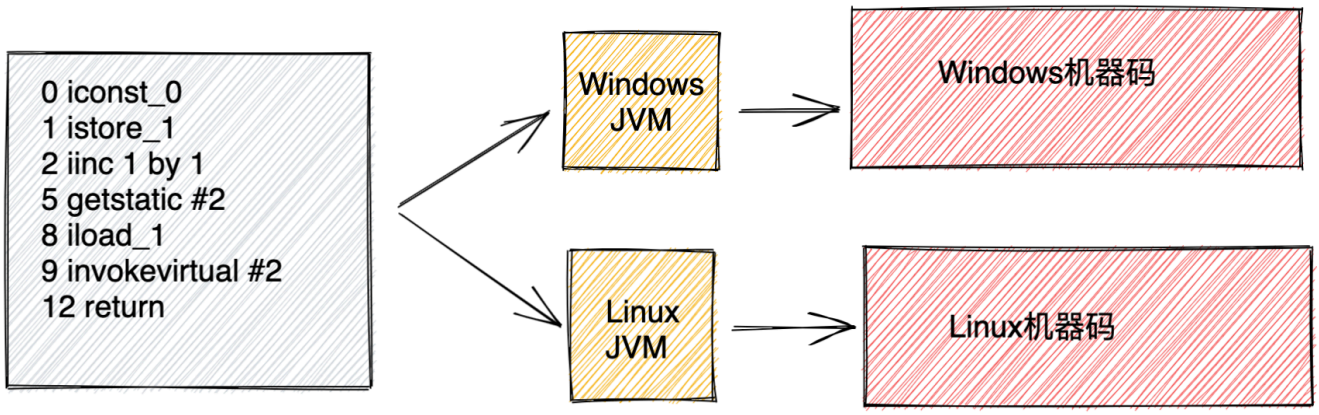


其中前端编译是在 JVM 虚拟机之外执行，所以与 JVM 虚拟机没有太大的关系。任何编程语言，只要能够编译出满足 JVM 规范的 Class 文件，就可以提交到 JVM 虚拟机执行。至于编译的过程，如果你不是想要专门去研究语言，那么就没有必要太过深入的去了解了。这里就暂时略过。我们更关注JVM 在后端编译过程中如何提升执行的效率。

字节码指令是如何执行的

解释执行与编译执行

在之前已经介绍过，Class 文件当中就已经保留了每一行 Java 代码对应的字节码指令，也就是说，执行引擎要如何执行一段 Java 代码，其实早在 Class 文件当中就已经确定了。执行引擎要做的事情，其实就是将这些Class文件中的字节码指令翻译成对应操作系统的机器码，然后扔给服务器执行就行了。 本质上，就相当于是一个翻译。



那么怎么做这个翻译工作呢？最简单的方式，当然就是来一个指令就翻译一次。就像是一个无脑的翻译机器，不用管合不合理，按字翻译就是了。没错，早期的JVM执行引擎其实就是这么做的，这种执行方式，就称为**解释执行**。

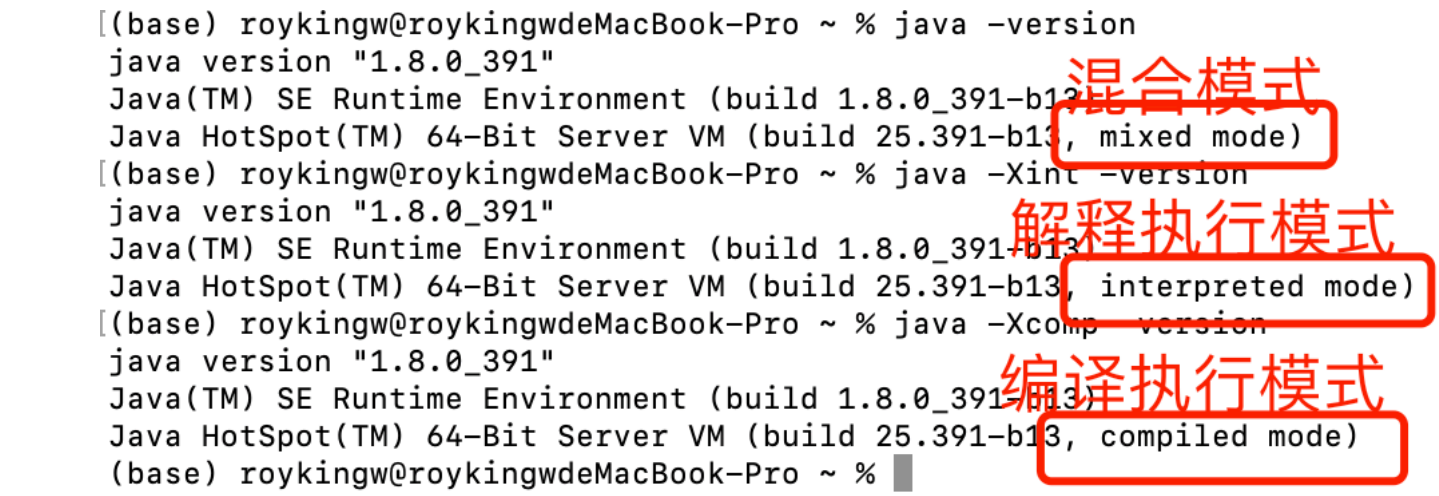
但是这种方式需要在上层语言和机器码之间经过中间一层JVM字节码的转换，显然执行效率是比不上 C 和 C++那些直接面向本地机器指令编程的语言的，这也是长久以前，Java 被 C 和 C++开发者吐槽执行速度慢的根源。

那么要怎么提升JAVA的执行效率呢?

JAVA的基本思想就是维护一个缓存，CodeCache，将那些字节码指令，提前编译出来，放到缓存里。到执行的时候，直接从缓存中查出来就好了。这种先编译，后执行的方式，就称为**编译执行**。

但是JAVA官方也不知道程序员会写出什么样稀奇古怪的代码，所以，自然没办法提前维护出一个完整的字节码缓存。那么就只能退而求其次，将那些运行频率最高的热点代码提前编译出来，放到缓存里。这样，至少最常用的那些方法的调用效率能够提高。完成这个任务的编译器，称为**即时编译器 JIT**(Just In Time Compiler)。

使用java -version就可以看到当前使用的是哪种执行模式。



从这里可以看到，HotSpot虚拟机并没有直接选择执行效率更快的编译执行，而是默认采用的一种混合执行的方式。

为什么JVM不直接采用性能明显更高的编译执行模式呢？这是因为虽然编译执行可以将越来越多的代码编译成本地代码，这样可以减少解释器的中间损耗，获得更高的执行效率。但是，这也意味着对内存有更多的资源限制，在很多资源比较紧张的场景，比如客户端应用，嵌入式系统等，使用解释执行就能更节约内存。

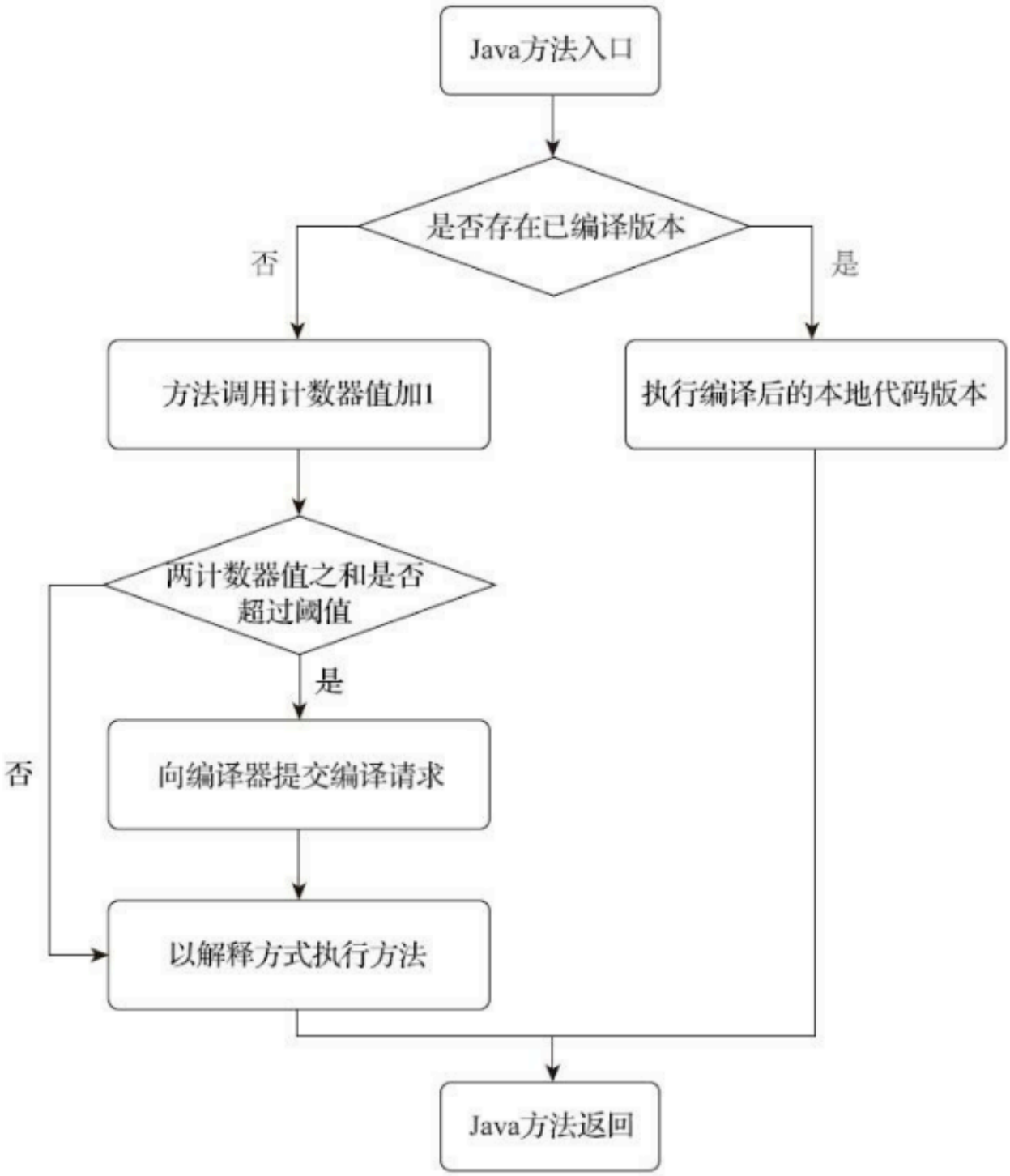
另外，编译执行需要较长的预热过程。在 CodeCache 中的代码缓存维护好之前，编译执行相比解释执行需要额外的性能消耗，用来识别热点代码，维护 CodeCache 。同时，编译执行在识别热点代码的过程中，还需要解释执行来帮助提供一些信息支持。在 HotSpot 中，会默认使用混合执行模式，而不是单纯的使用其中一种模式。

热点代码识别

使用 JIT 实时编译的前提就是需要识别出热点代码。要知道某段代码是不是热点代码，是不是需要触发即时编译，这个行为称为“**热点探测**”(Hot Spot Code Detection)。热点探测有很多种实现思路，而在 HotSpot 虚拟机中采用的是一种基于计数器的热点探测方法。HotSpot 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)。当虚拟机运行参数确定的前提下，这两个计数器都有一个明确的阈值，计数器阈值一旦溢出，就会触发即时编译。

首先来看看方法调用计数器。顾名思义，这个计数器就是用于统计方法被调用的次数。每次调用一个方法时，就记录一次这个方法的执行次数。当他的执行次数非常多，超过了某一个阈值，那么这个方法就可以认为是热点方法。这个方法对应的代码，自然也就是热点代码了。这时就可以向JIT提交一个针对该方法的代码编译请求了。

方法调用技术器的默认阈值是10000次，这个阈值可以通过虚拟机参数**-XX:CompileThreshold**来设定。当一个方法被调用时，虚拟机会先检查该方法是否存在被即时编译过的版本，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，则将该方法的调用计数器值加一，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阈值。一旦已超过阈值的话，将会向即时编译器提交一个该方法的代码编译请求。整体流程如下图：



比如这个方法计数器的默认阈值，就可以使用 java -XX:+PrintFlagsInitial -version 指令查询。

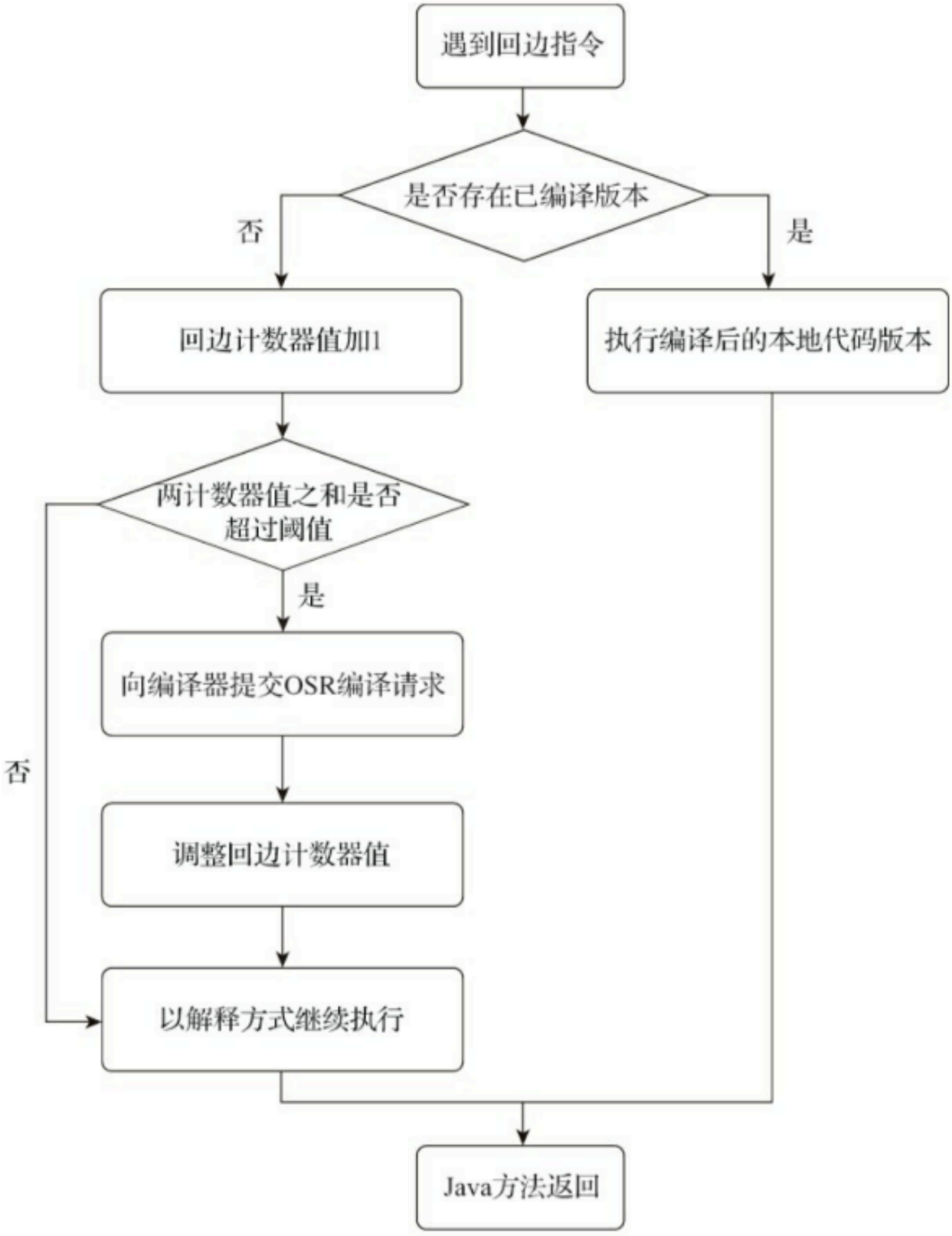
```
ccstrlist CompileCommand                      =                                     {product}
    ccstr CompileCommandFile                  =                                     {product}
ccstrlist CompileOnly                         =                                     {product}
     intx CompileThreshold                    = 10000                               {pd product}
     bool CompilerThreadHintNoPreempt         = true                                {product}
     intx CompilerThreadPriority              = -1                                  {product}
```

方法计数器以方法为维度，自然是不够精细的。所以，要更精细的识别热点代码，还需要配合接下来的**回边计数器**

回边计数器，它的作用是统计一个方法中循环体代码执行的次数。在字节码中遇到控制流向后跳转的指令就称为“回边(Back Edge)”，很显然建立回边计数器统计的目的是为了发现一个方法内部频繁的循环调用。回边计数器在服务端模式下默认的阈值是 10700

> 阈值计算公式(有兴趣可以了解一下)：回边计数器阈值 =方法调用计数器阈值（CompileThreshold）×（OSR比率（OnStackReplacePercentage）-解释器监控比率（InterpreterProfilePercentage）/100
>
> 其中OnStackReplacePercentage默认值为140，InterpreterProfilePercentage默认值为33，如果都取默认值，那Server模式虚拟机回边计数器的阈值为10700。回边计数器阈值 =10000×（140-33）=10700

当解释器遇到一条回边指令时，会先查找将要执行的代码片段是否有已经编译好的版本，如果有的话，它将会优先执行已编译的代码，否则就把回边计数器的值加一，然后判断方法调用计数器与回边计数器值之和是否超过回边计数器的阈值。当超过阈值的时候，将会提交一个编译请求，并且把回边计数器的值稍微降低一些，以便继续在解释器中执行循环，等待编译器输出编译结果，整个执行过程如下图。



客户端编译器与服务端编译器

既然识别出热点代码了，自然是要进行编译优化。在HotSpot虚拟机中，热点代码在被JIT实时编译的过程中，JIT编译器会运用很多经典的编译优化技术来实现对字节码指令的优化，让编译出来的记过运行效率更高。而这是所有成熟产品都需要考虑的问题。

HotSpot虚拟机中内置了两个即时编译器，其中前两个编译器存在已久，分别被称为“客户端编译器”（Client Compiler）和“服务端编译器”（Server Compiler），简称为**C1编译器**和**C2编译器**（部分资料和JDK源码中C2也叫Opto编译器）。

C1就相当于是一个初级翻译。编译过程中，C1会对字节码进行简单和可靠的优化，耗时短，以达到更快的编译速度。启动快，占用内存小。但是翻译出来的机器码优化程度不太高。比较适合于一些小巧的桌面应用，因此也称为**客户端编译器**

C2就相当于是一个高级翻译。编译过程中，C2会对字节码进行更激进的优化，优化后的俗代码执行效率更高。但是相应的，工作量也变得更大了。C2的启动更慢，占用内存也更多。进行耗时较长的优化，以及激进优化，但优化的代码执行效率更高。启动慢，占用内存多，执行效率高。比较适合于一些资源充裕的服务级应用，因此也称为**服务端编译器**。

与之前介绍的解释执行和编译执行一样，C2也并不是一定就比C1更好。实际上，这两个编译器并不是互相取代的关系，而是相互协作的关系。

由于即时编译器编译本地代码需要占用程序运行时间，通常要编译出优化程度越高的代码，所花费的时间便会越长；而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，这对解释执行阶段的速度也有所影响。为了在程序启动响应速度与运行效率之间达到最佳平衡，HotSpot虚拟机在编译子系统中加入了分层编译的功能，分层编译根据编译器编译、优化的规模与耗时，划分出不同的编译层次。

等级	描述	性能
0	程序纯解释执行，并且解释器不开启性能监控功能(Profiling)	1
1	使用C1编译器将字节码编译为本地代码来运行，进行简单可靠的稳定优化。不开启性能监控功能。	4
2	仍然使用C1编译器执行，仅开启方法及回边次数统计等有限的性能监控功能。	3
3	仍然使用C1编译器执行，开启全部性能监控，除了第2层的统计信息外，还会收集如分支跳转、虚方法调用版本等全部的统计信息。	2
4	使用C2编译器将字节码编译为本地代码，相比起C1编译器，C2编译器会启用更多编译耗时更长的优化，还会根据性能监控信息进行一些不可靠的激进优化。	5

JDK8 中提供了参数 -XX:TieredStopAtLevel=1 可以指定使用哪一层编译模型。

例如下面这个小案例，你可以分别使用 -Xint , -Xcomp, -XX:TieredStopAtLevel=1,-XX:TieredStopAtLevel=5，进行一下比较。

> 这几个参数的作用之前都解释过。

```
public class JitDemo {

    private int add(int x){
        return x+1;
    }

    public static void main(String[] args) {
        int a = 0;
```

```
        JitDemo demo = new JitDemo();
        long l = System.currentTimeMillis();
        for (int i = 0; i < 10000000; i++) {
            a = demo.add(a);
        }
        System.out.println("a= "+a);
        System.out.println(">>>>>>>>>"+(System.currentTimeMillis()-l));
    }
}
```

从这个过程可以看到，虽然C2的优化效果最好，但是我们也不可能一上来就使用C2进行编译。使用C2进行编译之前，往往需要通过C1编译器收集很多额外的数据。而C2编译器的最终优化效果，也需要跟C1进行比较才能确定。某些特定情况下，C2的执行效果有可能还不如C1，这时，就需要重新优化，重新编译。所以，这两个编译器并不是互相取代的关系，而是相互协作的关系。

后端编译优化技术

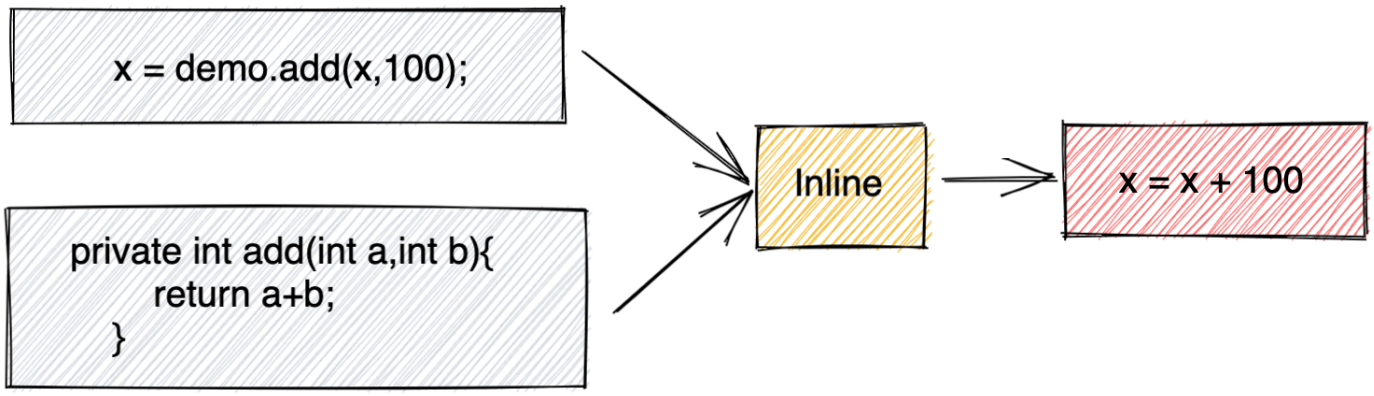
之前解释过，解释编译时，JVM 只负责依次将字节码指令转换成为机器指令。而在转换过程中，JVM会对代码做一些优化，从而保证在大部分情况下，即使程序员写出很烂的代码，JVM也能保持一个不错的执行效率。这就是编译优化技术。

> 具体的优化策略可以参看OpenJDK 的一个 WIKI 上的博客： https://wiki.openjdk.java.net/display/HotSpot/PerformanceTacticIndex。

这些代码优化技术实现起来确实有非常大的难度，因为大部分具体实现都需要在汇编层面进行。但是大部分的优化机制理解起来其实并不是太困难，所以接下来我会挑几个比较容易理解的优化机制，带你有个大致的了解。

方法内联 Inline

方法内联的优化行为就是把目标方法的代码复制到发起调用的方法之中，避免发生真实的方法调用。这样就可以减少频繁创建栈帧的性能开销。



比如下面的示例代码：

```
public class CompDemo {
    private int add1(int x1,int x2,int x3,int x4){
        return add2(x1,x2)+ add2(x3,x4);
    }
    private int add2(int x1, int x2){
```

```
        return x1+x2;
    }
    //内联优化
    private int add(int x1,int x2,int x3,int x4){
        return x1+x2+x3+x4;
    }

    public static void main(String[] args) {
        CompDemo compDemo = new CompDemo();
        //超过方法调用计数器的阈值 100000 次，才会进入 JIT 实时编译，进行内联优化。
        for (int i = 0; i < 1000000; i++) {
            compDemo.add1(1,2,3,4);
        }
    }
}
```

加入 JVM 参数：-XX:+PrintCompilation -XX:+UnlockDiagnosticVMOptions -XX:+PrintInlining -XX:+PrintCompilation -XX:+UnlockDiagnosticVMOptions 后可以看到以下的执行日志

```
  37   31 %     4       com.roy.jit.CompDemo::main @ 10 (32 bytes)
                         @ 21   com.roy.jit.CompDemo::add1 (15 bytes)   inline (hot)
                           @ 3   com.roy.jit.CompDemo::add2 (4 bytes)   inline (hot)
                           @ 10   com.roy.jit.CompDemo::add2 (4 bytes)   inline (hot)
```

当然，发生方法内联的前提是要让这个方法循环足够的次数，成为热点代码。比如，如果将循环次数减少，就看不到方法内联了。

方法内联的优化行为理解起来并不困难，就是把目标方法的代码原封不动的“复制”到发起调用的方法之中，避免发生真实的方法调用。但是，实际上，Java 虚拟机中的内联过程却远没有想象中那么容易。而且，方法内联往往还是很多后续优化手段的基础。

例如，对于下面的这个案例：

```
public class InlineDemo {
    public static void foo(Object obj){
        if(obj !=null){
            System.out.println("do something");
        }
    }
    //方法内联之后会继续进行无用代码消除
    public static void testInline(){
        Object obj= null;
        foo(obj);
    }
    public static void main(String[] args) {
        long l = System.currentTimeMillis();
        for (int i = 0; i < 10000000; i++) {
            testInline();
        }
        System.out.println(">>>>>>>>>"+(System.currentTimeMillis()-l));
    }
```

```
}
```

单独来看，testInline和里面的foo是两个独立的方法，但是，将foo方法内联到testInline方法后，就可以发现这就是一段无用的死代码"Dead Code"。接下来，JVM 虚拟机就可以进行 dead code elimination 死代码抹除的优化。

在JDK8中，提供了多个跟Inline内联相关的参数，可以用来干预内联行为。

- -XX:+Inline 启用方法内联。默认开启。
- -XX:InlineSmallCode=size 用来判断是否需要对方法进行内联优化。如果一个方法编译后的字节码大小大于这个值，就无法进行内联。默认值是1000bytes。
- -XX:MaxInlineSize=size 设定内联方法的最大字节数。如果一个方法编译后的字节码大于这个值，则无法进行内联。默认值是35bytes。
- -XX:FreqInlineSize=size 设定热点方法进行内联的最大字节数。如果一个热点方法编译后的字节码大于这个值，则无法进行内联。默认值是325bytes。
- -XX:MaxTrivialSize=size 设定要进行内联的琐碎方法的最大字节数(Trivial Method：通常指那些只包含一两行语句，并且逻辑非常简单的方法。比如像这样的方法

  ```
  public int getTrivialValue() {
      return 42;
  }
  ```

)。默认值是6bytes。
- -XX:+PrintInlining 打印内联决策，通过这个指令可以看到哪些方法进行了内联。默认是关闭的。另外，这个参数需要配合-XX:+UnlockDiagnosticVMOptions 参数使用。

从这几个相关参数可以看到，我们可以通过以下一些方法提高内联发生的概率。

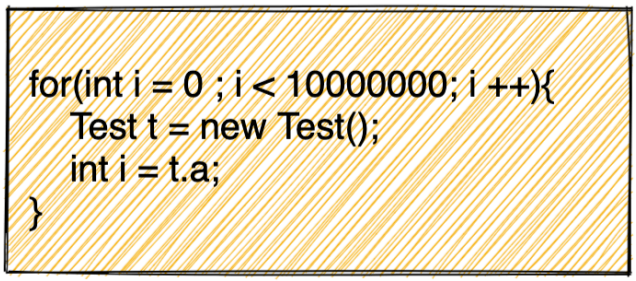
1 、在编程中，尽量多写小方法，避免写大方法。方法太大不光会导致方法无法内联，另外，成为热点方法后，还会占用更多的CodeCache。

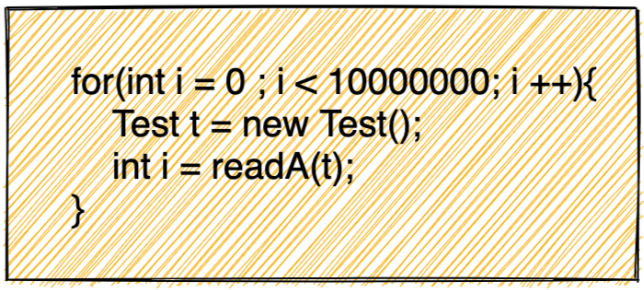
2 、在内存不紧张的情况下，可以通过调整JVM参数，减少热点阈值或增加方法体阈值，让更多的方法可以进行内联。

3 、尽量使用final, private,static关键字修饰方法。方法如果需要继承(也就是需要使用invokevirtual指令调用)，那么具体调用的方法，就只能在运行这一行代码时才能确定，编译器很难在编译时得出绝对正确的结论，也就加大了编译执行的难度。

逃逸分析 Escape Analysis

当一个对象在方法里面被定义后，它可能被外部方法所引用，例如作为调用参数传递到其他方法中，这种称为方法逃逸；甚至还有可能被外部线程访问到，譬如赋值给可以在其他线程中访问的实例变量，这种称为线程逃逸；从不逃逸、方法逃逸到线程逃逸，称为对象由低到高的不同逃逸程度。



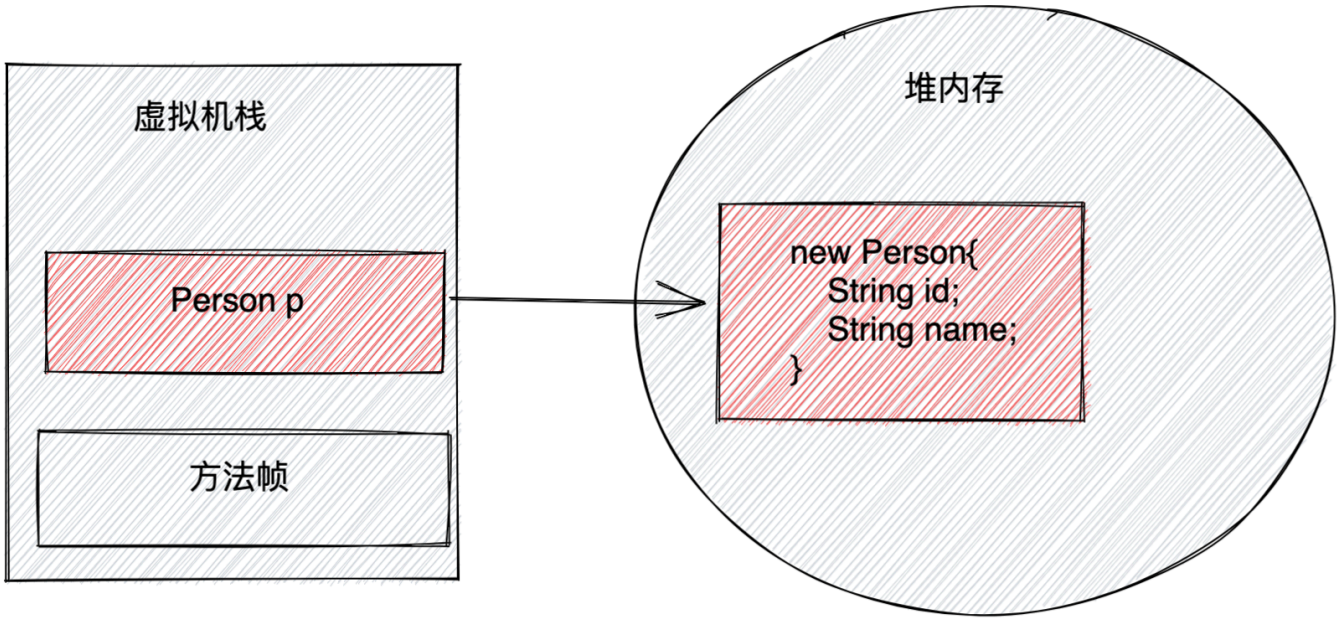


左侧的代码中，t对象，不会被外部引用，只会在方法中使用，所以不会发生逃逸。而右侧的代码中，t对象就很明显被其他方法使用了，这就会产生逃逸。JDK8 中默认开启了逃逸分析，可以添加参数 -XX:-DoEscapeAnalysis 主动关闭逃逸分析。

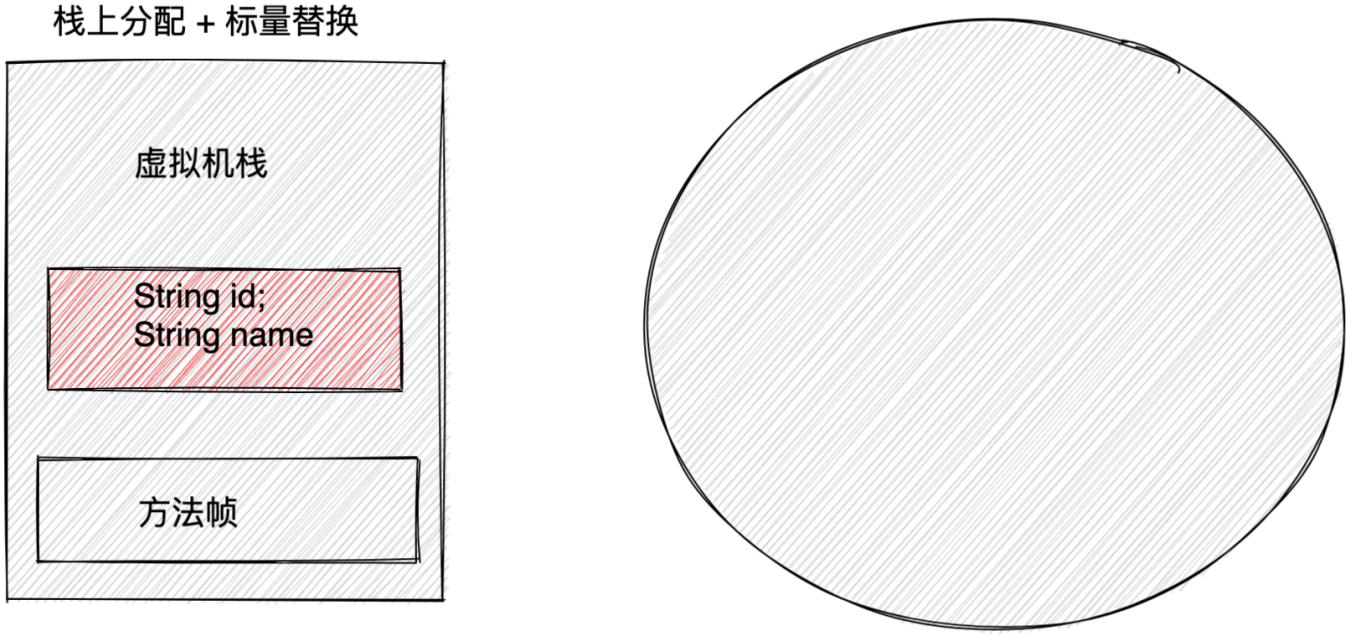
如果能证明一个对象不会逃逸到方法或线程之外，那么 JIT 就可以为这个对象实例采取后续一系列的优化措施。

第一个是**标量替换**(Scalar Replacement)。若一个数据已经无法再分解成更小的数据来表示了，Java虚拟机中的原始数据类型（int、long等数值类型及reference类型等）都不能再进一步分解了，那么这些数据就可以被称为标量。如果把一个Java对象拆散，根据程序访问的情况，将其用到的成员变量恢复为原始类型来访问，这个过程就称为标量替换。假如逃逸分析能够证明一个对象不会被方法外部访问，并且这个对象可以被拆散，那么程序真正执行的时候将可能不去创建这个对象，而改为直接创建它的若干个被这个方法使用的成员变量来代替。将对象拆分后，除了可以让对象的成员变量在栈上分配和读写之外，还可以为后续进一步的优化手段创建条件。标量替换对逃逸程度的要求更高，它不允许对象逃逸出方法范围内。JDK8 中默认开启了标量替换，可以通过添加参数 -XX:-EliminateAllocations 主动关闭标量替换。

第二个是**栈上分配**(Stack Allocations)。正常情况下，JVM 中所有对象都应该创建在堆上，并由 GC 线程进行回收。如果确定一个对象不会逃逸出线程之外，那让这个对象在栈上分配内存将会是一个很不错的主意，对象所占用的内存空间就可以随栈帧出栈而销毁。在一般应用中，完全不会逃逸的局部对象和不会逃逸出线程的对象所占的比例是很大的，如果能使用栈上分配，那大量的对象就会随着方法的结束而自动销毁了，垃圾收集子系统的压力将会下降很多。栈上分配可以支持方法逃逸，但不能支持线程逃逸。



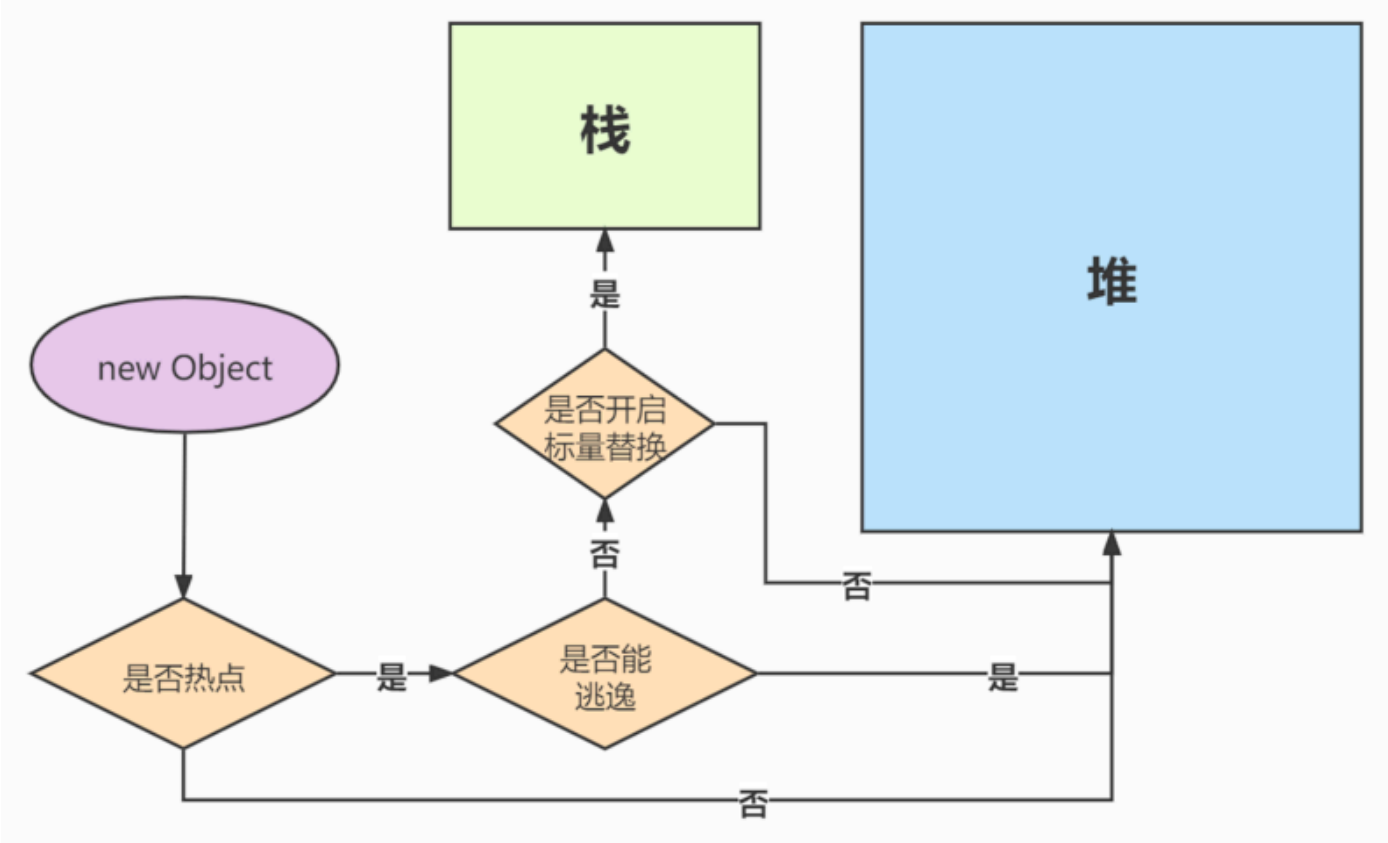
栈上分配 + 标量替换



这三种优化措施中，逃逸分析是基础。因为虚拟机栈是对应一个线程的，而堆内存是对应整个Java进程的。如果发生了线程逃逸，那么堆中的同一个对象，可能隶属于多个线程，这时要将堆中的对象挪到虚拟机栈中，那就必须扫描所有的虚拟机栈，看看在这个虚拟机栈对应的线程中是否引用了这个对象。这个性能开销是难以接受的。

而栈是一个非常小的内存结构，他也不可能像堆中那么豪横的使用内存空间，所以，也必须要对对象进行最大程度的瘦身，才能放到栈中。而瘦身的方式，就是去掉对象的mark标志位中的补充信息，拆分成最精简的标量。所以，要开启栈上分配，标量替换也是不可或缺的。

他们三者的关系如下图：



可以用下面示例方法进行一下验证：

```
public class EscapeAnalysisTest {
    public static void main(String[] args) throws InterruptedException {
        long start = System.currentTimeMillis();
        for (int i = 0; i < 10000000; i++) {
            allocate();
        }
        System.out.println("运行耗时: "+(System.currentTimeMillis()-start));
        Thread.sleep(6000000);
    }

    static void allocate(){
        MyObject myObject = new MyObject(2024,2024.6);
    }

    static class MyObject {
        int a;
        double b;

        MyObject(int a,double b){
            this.a = a;
            this.b = b;
        }
    }
}
```

以我的测试环境来看，默认情况下， 运行时间大概 2 毫秒，而关闭逃逸分析或者关闭标量替换后，运行时间就扩大到了 44毫秒左右。

锁消除 lock elision

这也是经过逃逸分析后可以直接进行的优化措施。

这个优化措施主要是针对 synchronized 关键字。当 JVM 检测到一个锁的代码不存在多线程竞争时，会对这个对象的锁进行锁消除。

> 多线程并发资源竞争是一个很复杂的场景，所以通常要检测是否存在多线程竞争是非常麻烦的。
>
> 但是有一种情况很简单，如果一个方法没有发生逃逸，那么他内部的锁都是不存在竞争的。

比如下面的示例代码：

```
public class LockElisionDemo {
    public static String BufferString(String s1,String s2){
        StringBuffer sb = new StringBuffer();
        sb.append(s1);
        sb.append(s2);
        return sb.toString();
    }

    public static String BuilderString(String s1, String s2){
        StringBuilder sb = new StringBuilder();
        sb.append(s1);
        sb.append(s2);
        return sb.toString();
    }

    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 100000000; i++) {
            BufferString("aaaaa","bbbbbb");
        }
        System.out.println("StringBuffer耗时: "+(System.currentTimeMillis()-startTime));

        long startTime2 = System.currentTimeMillis();
        for (int i = 0; i < 100000000; i++) {
            BuilderString("aaaaa","bbbbbb");
        }
        System.out.println("StringBuilder耗时: "+(System.currentTimeMillis()-startTime2));
    }
}
```

其中分别测试了 StringBuffer 和 StringBuilder 的字符串构建方法。这两个方法功能上没有什么区别，最大的区别在于，StringBuffer 是线程安全的，他的append和toString都是加了 synchronized 同步锁的，而 StringBuilder 则没有加。之前介绍过，synchronized 关键字其实是在Class文件中添加了monitorenter和monitorexit两个字节码指令的，所以，StringBuffer显然要比 StringBuilder 更慢。

在当前代码中，BufferString 方法只是在main这一个线程里调用，不存在线程竞争，所有这个synchronized 同步锁是没有作用的，因此，在触发了 JIT 后，JVM 会在编译时就会将这个无用的锁消除掉。这样，两个方法的耗时是差不多的。

```
StringBuffer耗时：1521
StringBuilder耗时：1039
```

与之形成对比，如果给这个示例代码添加一个JVM 参数： -XX:-EliminateLocks 主动关闭锁清除后，再执行这个案例，两个方法的耗时差距就明显更大了。

```
StringBuffer耗时：2461
StringBuilder耗时：1049
```

> 有道云笔记链接：https://note.youdao.com/s/KTlGtULQ