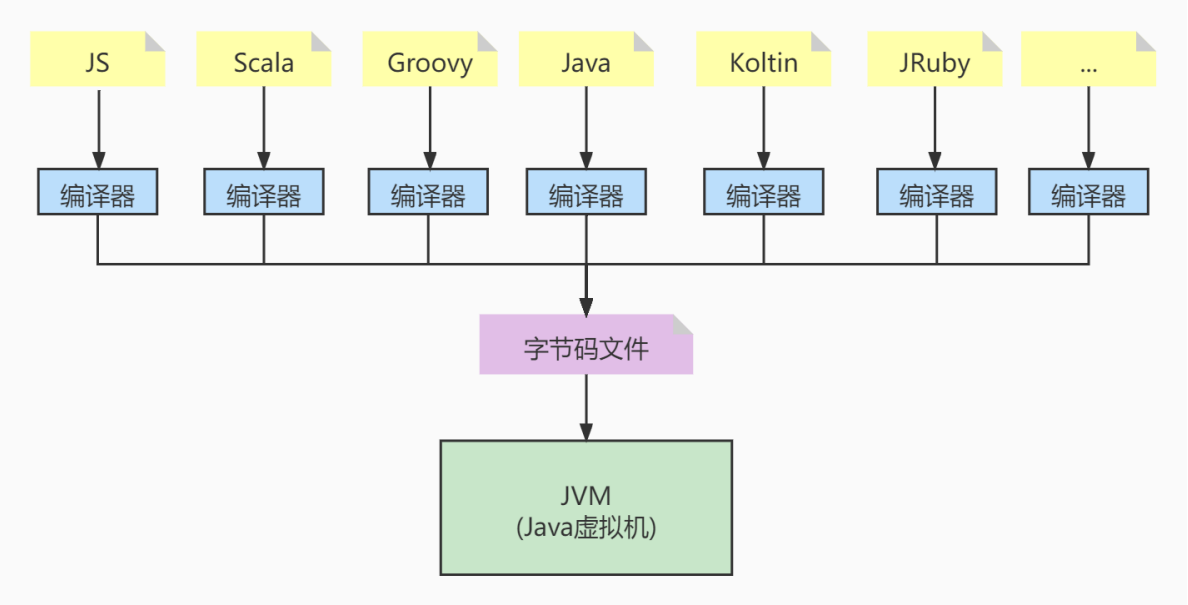
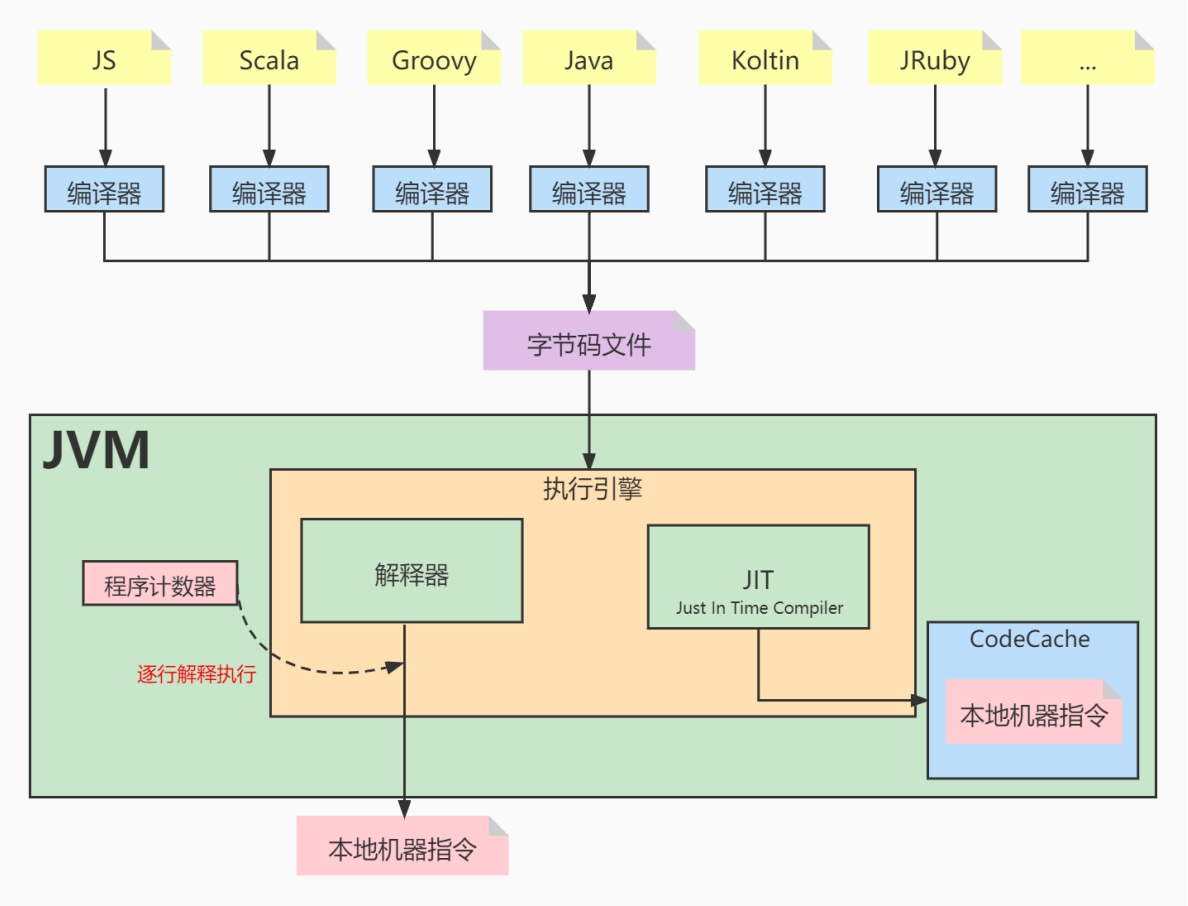
JVM的语言无关性

有道云链接：http://note.youdao.com/noteshare?id=03a80a9c9aaddaed7ddf899fb02527bc&sub=6EE37BF7ECA04B9297079E3C4C54C92B

跨语言（语言无关性）：JVM只识别字节码，所以JVM其实跟语言是解耦的，也就是没有直接关联，JVM运行不是翻译Java文件，而是识别class文件，这个一般称之为字节码。还有像Groovy 、Kotlin、Scala等等语言，它们其实也是编译成字节码，所以它们也可以在JVM上面跑，这个就是JVM的跨语言特征。Java的跨语言性一定程度上奠定了非常强大的java语言生态圈。



解释执行与JIT



Java程序在运行的时候，主要就是执行字节码指令，一般这些指令会按照顺序解释执行，这种就是解释执行。

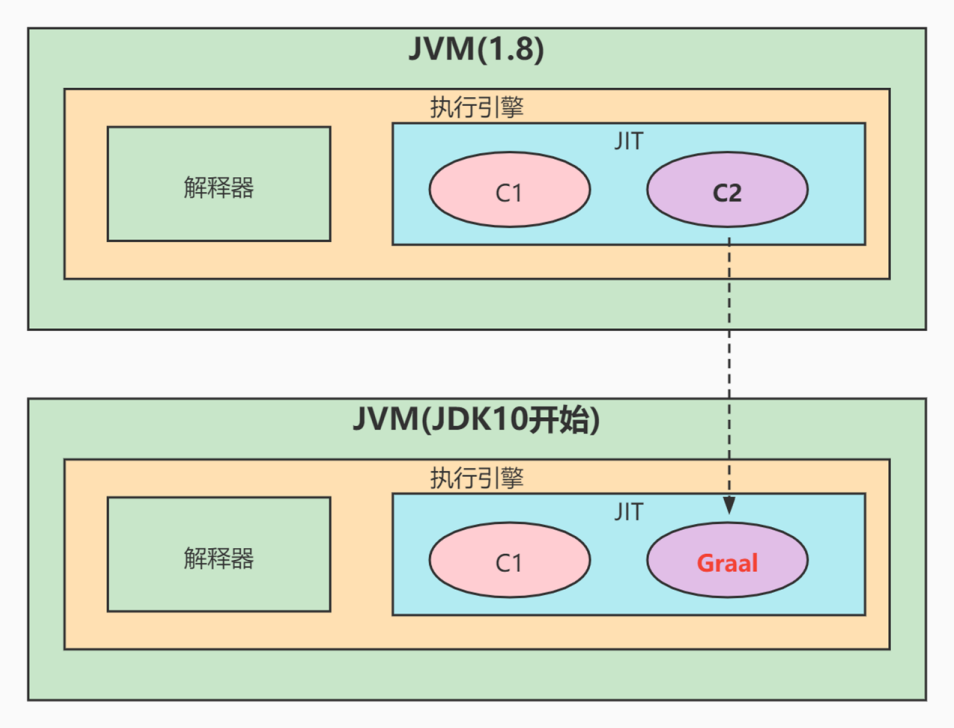
```
//解释执行
public void Compile(String str) {
    if(str.equals("iload")){//遇到具体字节码指令，翻译成把机器码
        //0101010101010 机器指令
    }
    if(str.equals("bipush")){
        //111010111010110 机器指令
    }
    if(str.equals("istore")){
        //001010111011110 机器指令
    }
}
```

但是那些被频繁调用的代码，比如调用次数很高或者在 for 循环里的那些代码,如果按照解释执行，效率是非常低的。（这个就是Java以前被C、C++开发者吐槽慢的原因）

以上的这些代码称为热点代码。所以，为了提高热点代码的执行效率，在运行时，虚拟机将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，就称为即时编译器（Just In Time Compiler），简称 JIT 编译器。

C1、C2与Graal编译器



在JDK1.8中 HotSpot 虚拟机中，内置了两个 JIT，分别为 C1 编译器和 C2 编译器。

C1编译器

C1 编译器是一个简单快速的编译器，主要的关注点在于局部性的优化，适用于执行时间较短或对启动性能有要求的程序，例如，GUI 应用对界面启动速度就有一定要求，C1也被称为 Client Compiler。

C1编译器几乎不会对代码进行优化

C2编译器

C2 编译器是为长期运行的服务器端应用程序做性能调优的编译器，适用于执行时间较长或对峰值性能有要求的程序。根据各自的适配性，这种即时编译也被称为Server Compiler。

但是C2代码已超级复杂，无人能维护！所以才会开发Java编写的Graal编译器取代C2(JDK10开始)

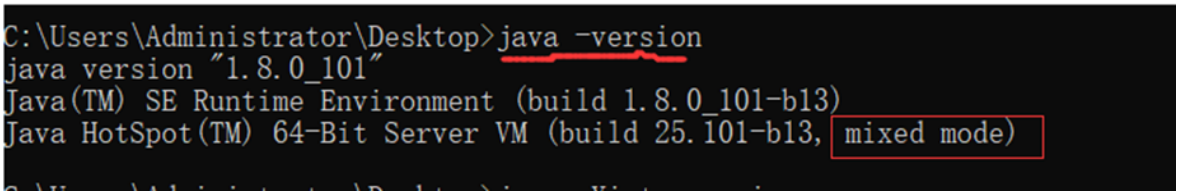
分层编译

在 Java7之前，需要根据程序的特性来选择对应的 JIT，虚拟机默认采用解释器和其中一个编译器配合工作。

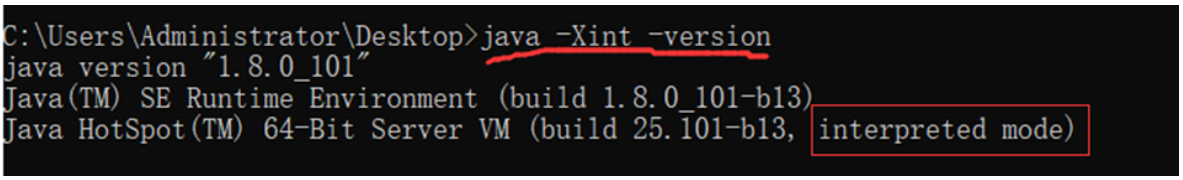
Java7及以后引入了分层编译，这种方式综合了 C1 的启动性能优势和 C2 的峰值性能优势，当然我们也可以通过参数强制指定虚拟机的即时编译模式。

在 Java8 中，默认开启分层编译。

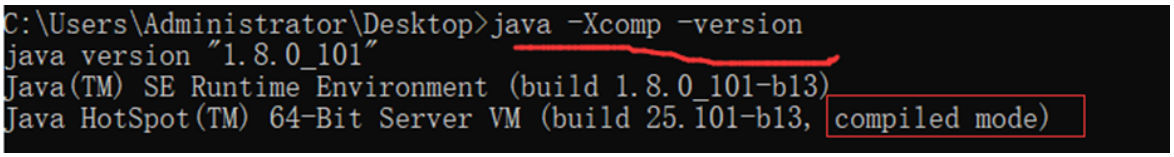
通过 **java -version** 命令行可以直接查看到当前系统使用的编译模式**(**默认分层编译**)**



使用“**-Xint**”参数强制虚拟机运行于只有解释器的编译模式



使用“**-Xcomp**”强制虚拟机运行于只有 **JIT** 的编译模式下



JVM 的执行状态分为了 5 个层次：(不重要、了解即可)

Ø 第 0 层：程序解释执行，默认开启性能监控功能（Profiling），如果不开启，可触发第二层编译；

Ø 第 1 层：可称为 C1 编译，将字节码编译为本地代码，进行简单、可靠的优化，不开启 Profiling；

Ø 第 2 层：也称为 C1 编译，开启Profiling，仅执行带方法调用次数和循环回边执行次数 profiling 的 C1 编译；

Ø 第 3 层：也称为 C1 编译，执行所有带 Profiling 的 C1 编译；

Ø 第 4 层：可称为 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。

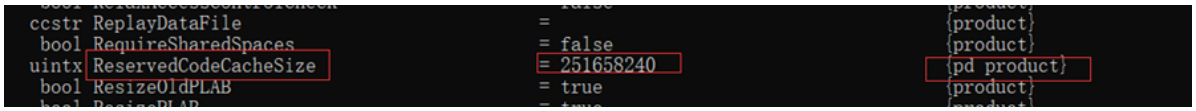
热点代码

热点代码，就是那些被频繁调用的代码，比如调用次数很高或者在 for 循环里的那些代码。这些再次编译后的机器码会被缓存起来，以备下次使用，但对于那些执行次数很少的代码来说，这种编译动作就纯属浪费。

JVM提供了一个参数“-XX:ReservedCodeCacheSize”，用来限制 CodeCache 的大小。也就是说，JIT 编译后的代码都会放在 CodeCache 里。

如果这个空间不足，JIT 就无法继续编译，编译执行会变成解释执行，性能会降低一个数量级。同时，JIT 编译器会一直尝试去优化代码，从而造成了 CPU 占用上升。

通过 java -XX:+PrintFlagsFinal –version查询:



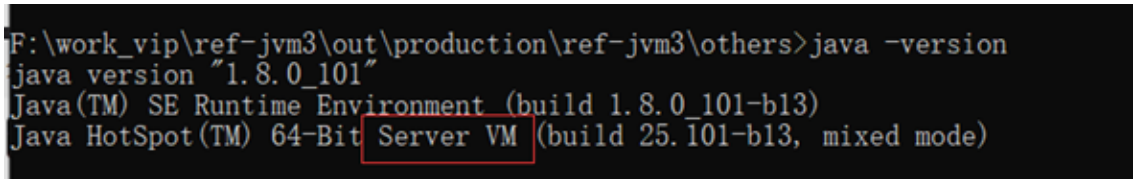
热点探测

在 HotSpot 虚拟机中的热点探测是 JIT 优化的条件，热点探测是基于计数器的热点探测，采用这种方法的虚拟机会为每个方法建立计数器统计方法的执行次数，如果执行次数超过一定的阈值就认为它是“热点方法”

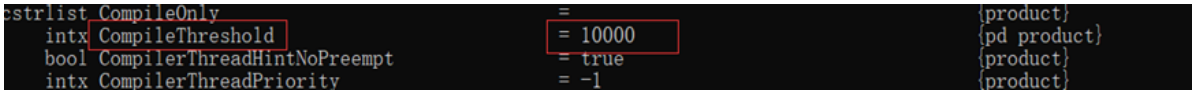
虚拟机为**每个方法**准备了**两类计数器**：方法调用计数器（Invocation Counter）和回边计数器（Back Edge Counter）。在确定虚拟机运行参数的前提下，这两个计数器都有一个确定的阈值，当计数器超过阈值溢出了，就会触发 JIT 编译。

方法调用计数器

用于统计方法被调用的次数，方法调用计数器的默认阈值在客户端模式下是 1500 次，在服务端模式下是 10000 次(我们用的都是服务端，java –version查询)，可通过 -XX: CompileThreshold 来设定



通过 java -XX:+PrintFlagsFinal –version查询



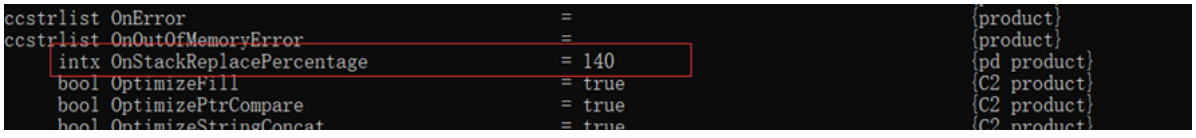
回边计数器

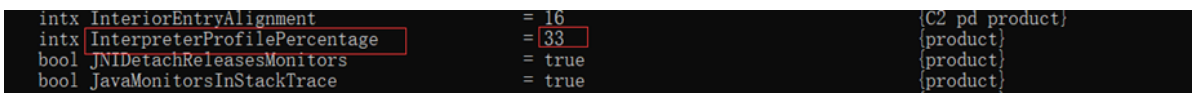
用于统计一个方法中循环体代码执行的次数，在字节码中遇到控制流向后跳转的指令称为“回边”（Back Edge），该值用于计算是否触发 C1 编译的阈值，在不开启分层编译的情况下，在服务端模式下是**10700。**

怎么算的呢！参考以下公式（有兴趣可了解）：

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

通过 java -XX:+PrintFlagsFinal –version查询先关参数:





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

编译优化技术

JIT 编译运用了一些经典的编译优化技术来实现代码的优化，即通过一些例行检查优化，可以智能地编译出运行时的最优性能代码.

方法内联

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。

例如以下方法：

```
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;
}
```

最终会被优化为：

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

JVM 会自动识别热点方法，并对它们使用方法内联进行优化。

我们可以通过 -XX:CompileThreshold 来设置热点方法的阈值。

但要强调一点，热点方法不一定会被 JVM 做内联优化，如果这个方法体太大了，JVM 将不执行内联操作。

而方法体的大小阈值，我们也可以通过参数设置来优化：

经常执行的方法，默认情况下，方法体大小小于 325 字节的都会进行内联，我们可以通过 -XX:FreqInlineSize=N 来设置大小值；

```
 intx FreqInlineSize                = 325                                 {pd product}
double G1ConcMarkStepDurationMillis = 10.000000                           {product}
```

不是经常执行的方法，默认情况下，方法大小小于 35 字节才会进行内联，我们也可以通过 -XX:MaxInlineSize=N 来重置大小值。

```
intx MaxInlineLevel                = 9                                   {product}
intx MaxInlineSize                 = 35                                  {product}
intx MaxJNILocalCapacity           = 65536                               {product}
```

代码演示

```
public static void main(String[] args) {
    CompDemo compDemo = new CompDemo();
    //方法调用计数器的默认阈值10000次，我们循环遍历超过需要阈值
    for(int i=0; i<1000000; i++) {
        compDemo.add1(1,2,3,4);
    }

}
```

设置 VM 参数：-XX:+PrintCompilation -XX:+UnlockDiagnosticVMOptions
-XX:+PrintInlining
-XX:+PrintCompilation　//在控制台打印编译过程信息
-XX:+UnlockDiagnosticVMOptions //解锁对JVM进行诊断的选项参数。默认是关闭的，开启后支持一些特定参数对JVM进行诊断
-XX:+PrintInlining //将内联方法打印出来

```
                          @ 10   ex16.CompDemo::add2 (4 bytes)
  137   38 %     4       ex16.CompDemo::main @ 10 (32 bytes)
                          @ 21   ex16.CompDemo::add1 (15 bytes)   inline (hot)
                            @ 3   ex16.CompDemo::add2 (4 bytes)   inline (hot)
                            @ 10   ex16.CompDemo::add2 (4 bytes)   inline (hot)
  138   36 %     3       ex16.CompDemo::main @ -2 (32 bytes)   made not entrant
  139   38 %     4       ex16.CompDemo::main @ -2 (32 bytes)   made not entrant
```

如果循环太少，则不会触发方法内联

```
//方法调用计数器的默认阈值10000次，我们循环遍历超过需要阈值
for(int i=0; i<1000; i++) {
    compDemo.add1(1,2,3,4);
}
```

```
                                  @ 14   java.lang.System::arraycopy
  236   34       3       ex16.CompDemo::add2 (4 bytes)
  237   35       3       ex16.CompDemo::add1 (15 bytes)
                             @ 3   ex16.CompDemo::add2 (4 bytes)
                             @ 10   ex16.CompDemo::add2 (4 bytes)
```

热点方法的优化可以有效提高系统性能，一般我们可以通过以下几种方式来提高方法内联：

- 通过设置 JVM 参数来减小热点阈值或增加方法体阈值，以便更多的方法可以进行内联，但这种方法意味着需要占用更多地内存；
- 在编程中，避免在一个方法中写大量代码，习惯使用小方法体；
- 尽量使用 final、private、static 关键字修饰方法，编码方法因为继承，会需要额外的类型检查。

锁消除

在非线程安全的情况下，尽量不要使用线程安全容器，比如 StringBuffer。由于 StringBuffer 中的 append 方法被 Synchronized 关键字修饰，会使用到锁，从而导致性能下降。

```
@Override
public synchronized StringBuffer append(String str) {
    toStringCache = null;
    super.append(str);
    return this;
}
```

但实际上，在以下代码测试中，StringBuffer 和 StringBuilder 的性能基本没什么区别。这是因为在局部方法中创建的对象只能被当前线程访问，无法被其它线程访问，这个变量的读写肯定

不会有竞争，这个时候 JIT 编译会对这个对象的方法锁进行锁消除。

下代码测试中，StringBuffer 和 StringBuilder 的性能基本没什么区别。这是因为在局部方法中创建的对象只能被当前线程访问，无法被其它线程访问，这个变量的读写肯定不会有竞争，这个时候 JIT 编译会对这个对象的方法锁进行锁消除。

```
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 sd = new StringBuilder();
    sd.append(s1);
    sd.append(s2);
    return sd.toString();
}
```

```
StringBuffer花费的时间536
StringBuilder花费的时间298
```

我们把锁消除关闭---测试发现性能差别有点大

-XX:+EliminateLocks开启锁消除（jdk1.8默认开启，其它版本未测试）

-XX:-EliminateLocks 关闭锁消除

```
StringBuffer花费的时间948
StringBuilder花费的时间279
```

标量替换

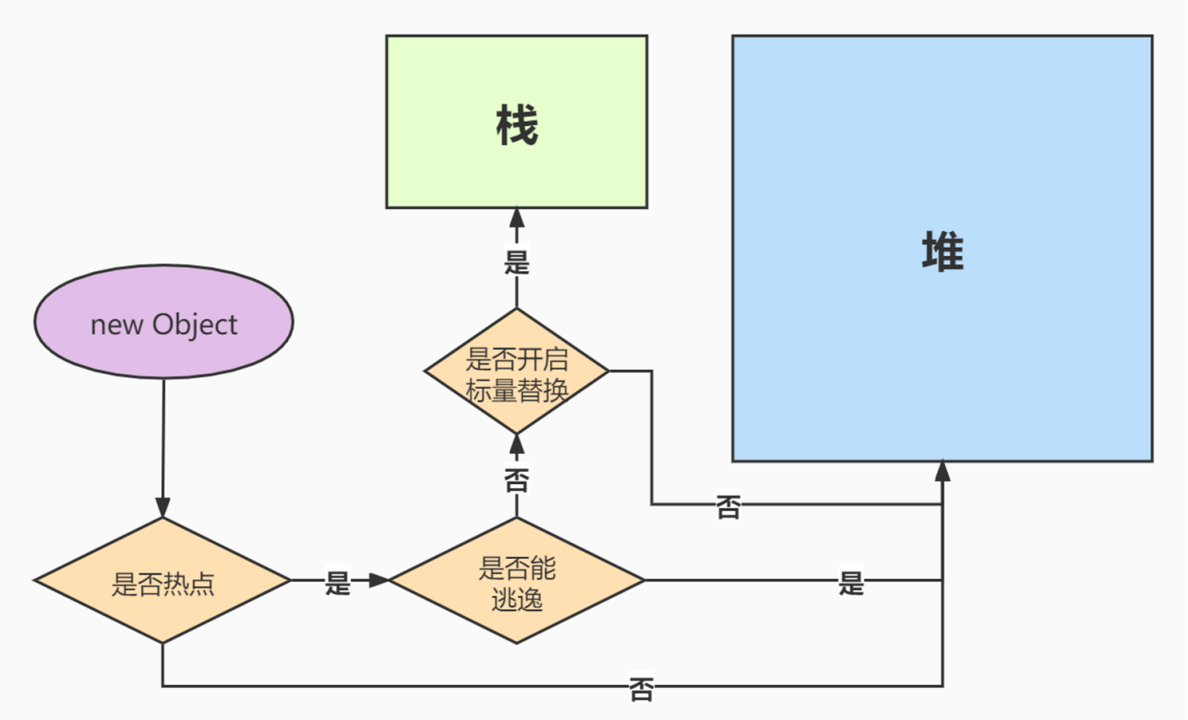
逃逸分析证明一个对象不会被外部访问，如果这个对象可以被拆分的话，当程序真正执行的时候可能不创建这个对象，而直接创建它的成员变量来代替。将对象拆分后，可以分配对象的成员变量在栈或寄存器上，原本的对象就无需分配内存空间了。这种编译优化就叫做标量替换（前提是需要开启逃逸分析）。

```java
/**
 * @author  King老师
 * 标量替换
 */
public class VariableDemo {

    public void foo() {
        Teacher teacher = new Teacher();
        teacher.name = "king";
        teacher.age = 18;
        //to do something
    }

    public void foo1() {
        String name = "king";
        int age = 18;
        //to do something
    }

}
```

-XX:+DoEscapeAnalysis开启逃逸分析（jdk1.8默认开启）
-XX:-DoEscapeAnalysis 关闭逃逸分析

-XX:+EliminateAllocations开启标量替换（jdk1.8默认开启）
-XX:-EliminateAllocations 关闭标量替换

逃逸分析技术



逃逸分析的原理：分析对象动态作用域，当一个对象在方法中定义后，它可能被外部方法所引用。

比如：调用参数传递到其他方法中，这种称之为方法逃逸。甚至还有可能被外部线程访问到，例如：赋值给其他线程中访问的变量，这个称之为线程逃逸。

从不逃逸到方法逃逸到线程逃逸，称之为对象由低到高的不同逃逸程度。

如果确定一个对象不会逃逸出线程之外，那么让对象在栈上分配内存可以提高JVM的效率。

当然逃逸分析技术属于JIT的优化技术，所以必须要符合热点代码，JIT才会优化，另外对象如果要分配到栈上，需要将对象拆分，这种编译优化就叫做标量替换技术。

如下图中foo方法如果使用标量替换的话，那么最后执行的话就是foo1方法的效果。

```java
/**
 * @author  King老师
 * 标量替换
 */
public class VariableDemo {

    public void foo() {
        Teacher teacher = new Teacher();
        teacher.name = "king";
        teacher.age = 18;
        //to do something
    }

    public void foo1() {
        String name = "king";
        int age = 18;
        //to do something
    }

}
```

逃逸分析代码示例

EscapeAnalysisTest.java

```java
/*
 * 逃逸分析-栈上分配
 * -XX:-DoEscapeAnalysis
 */
public class EscapeAnalysisTest {
    public static void main(String[] args) throws Exception {
        long start = System.currentTimeMillis();
        for (int i = 0; i < 50000000; i++) {
            allocate();
        }
        System.out.println((System.currentTimeMillis() - start) + " ms");
        Thread.sleep( millis: 600000);
    }

    static void allocate() { MyObject myObject = new MyObject( a: 2020, b: 2020.6); }

    static class MyObject {
        int a;
        double b;

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

这段代码在调用的过程中Mybojject这个对象属于不可逃逸，JVM可以做栈上分配，所以运行速度非常快！

JVM默认会做逃逸分析、会进行标量替换，会进行栈上分配。

```
EscapeAnalysisTest
"C:\Program Files\Java\jdk1.8.0_101\bin\java.exe" ...
4 ms
```

然后关闭逃逸分析

```
-XX:-DoEscapeAnalysis
```

```
EscapeAnalysisTest
"C:\Program Files\Java\jdk1.8.0_101\bin\java.exe" ...
278 ms
```

然后关闭逃逸分析

```
-XX:-EliminateAllocations
```

```
EscapeAnalysisTest
"C:\Program Files\Java\jdk1.8.0_101\bin\java.exe" ...
278 ms
```

测试结果可见，开启逃逸分析对代码的执行性能有很大的影响！那为什么有这个影响？

逃逸分析

如果是逃逸分析出来的对象可以在栈上分配的话，那么该对象的生命周期就跟随线程了，就不需要垃圾回收，如果是频繁的调用此方法则可以得到很大的性能提高。

采用了逃逸分析后，满足逃逸的对象在栈上分配

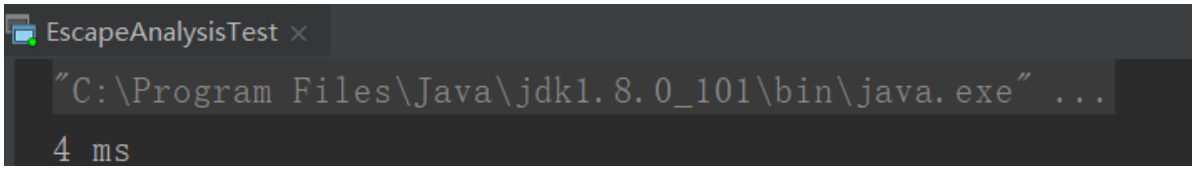
没有开启逃逸分析，对象都在堆上分配，会频繁触发垃圾回收（垃圾回收会影响系统性能），导致代码运行慢

代码验证

开启GC打印日志

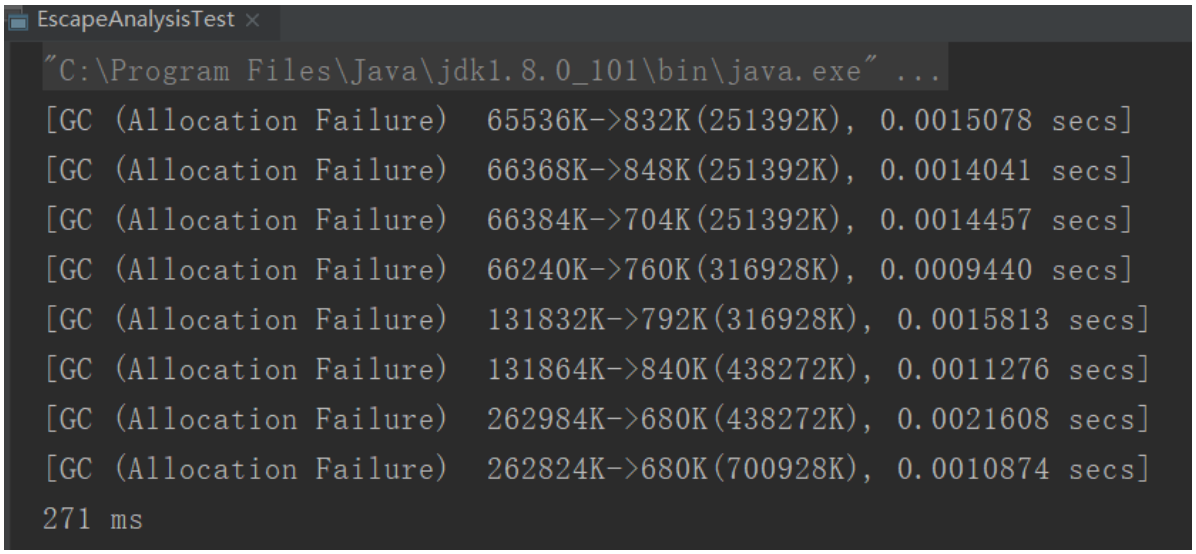
```
-XX:+PrintGC
```

开启逃逸分析



可以看到没有GC日志（因为进行了栈上分配）

关闭逃逸分析



可以看到关闭了逃逸分析，**JVM**在频繁的进行垃圾回收（**GC**），正是这一块的操作导致性能有较大的差别。