链滴

深入理解 Java 即时编译器（上）

作者：Lord-X
原文链接：https://ld246.com/article/1564804822825
来源网站：链滴

□rose□rose 如果您觉得我的文章对您有帮助的话，记得在GitHub上star一波哈□ose□rose

□rose□rose GitHub_awesome-it-blog □rose□rose

本文会先介绍Java的执行过程，进而引出对即时编译器的探讨，下篇会介绍分层编译的机制，最后介即时编译器对应用启动性能的影响。

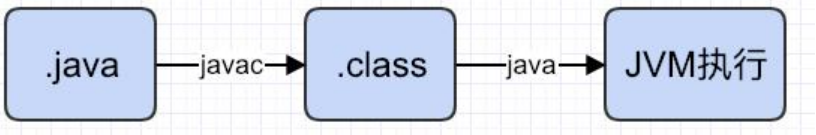
本文内容基于HotSpot虚拟机，设计Java版本的地方会在文中说明。

0 Java程序的执行过程

Java面试中，有一道面试题是这样问的：Java程序是解释执行还是编译执行？

在我们刚学习Java时，大概会认为Java是编译执行的。其实，Java既有解释执行，也有编译执行。

Java程序通常的执行过程如下：



源码.java文件通过javac命令编译成.class的字节码，再通过java命令执行。

需要说明的是，在编译原理中，通常将编译分为前端和后端。其中前端会对程序进行词法分析、语法析、语义分析，然后生成一个中间表达形式（称为IR：Intermediate Representation）。后端再讲个中间表达形式进行优化，最终生成目标机器码。

在Java中，javac之后生成的就是中间表达形式（.class），举个栗子

```
public class JITDemo2 {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

上述代码通过javap反编译后如下：

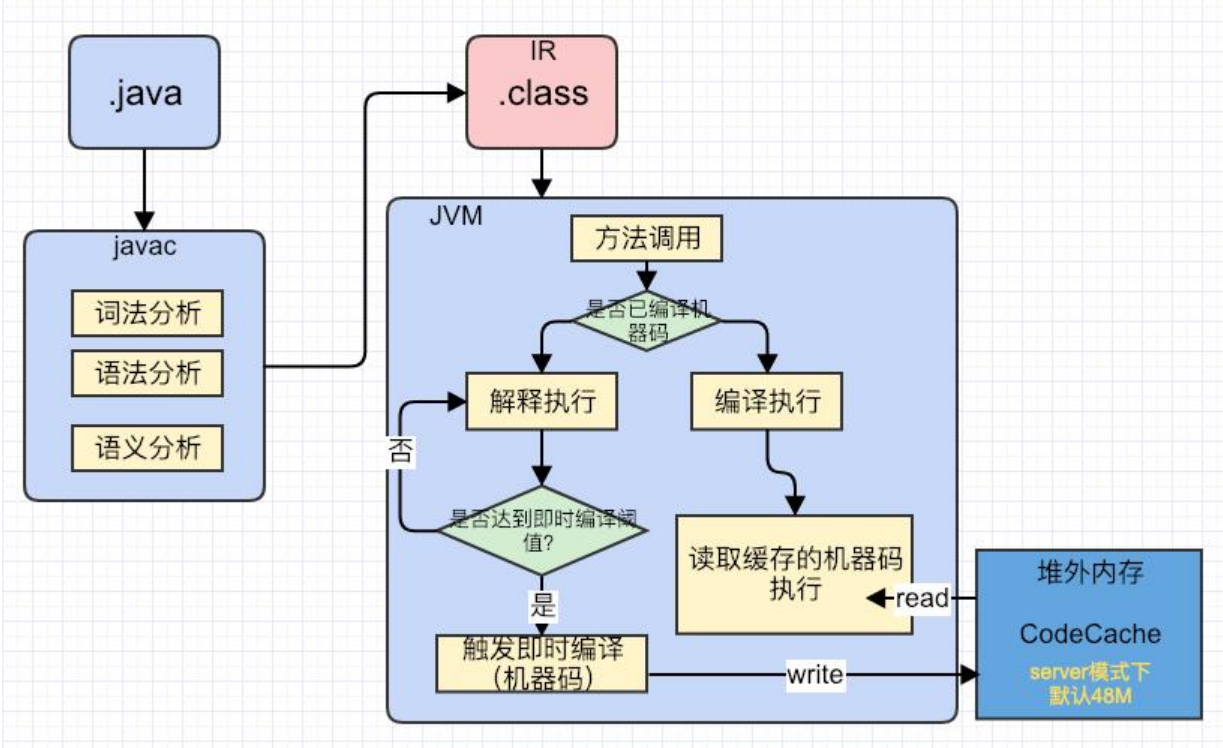
```
// javap -c JITDemo2.class

Compiled from "JITDemo2.java"
public class com.example.demo.jitdemo.JITDemo2 {
  public com.example.demo.jitdemo.JITDemo2();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return


  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3                  // String Hello World
       5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

JVM在执行时，首先会逐条读取IR的指令来执行，这个过程就是解释执行的过程。当某一方法调用次达到即时编译定义的阈值时，就会触发即时编译，这时即时编译器会将IR进行优化，并生成这个方法机器码，后面再调用这个方法，就会直接调用机器码执行，这个就是编译执行的过程。

所以，从.java文件到最终的执行，其过程大致如下：



原文链接：深入理解 Java 即时编译器（上）

（CodeCache会在下文中介绍）

那么，何时出发即时编译？即时编译的过程又是怎样的？我们继续往下研究。

1 Java即时编译器初探

HotSpot虚拟机有两个编译器，称为C1和C2编译器（Java10以后新增了一个编译器Graal）。

C1编译器对应参数-client，对于执行时间较短，对启动性能有要求的程序，可以选择C1。

C2编译器对应参数-server，对峰值性能有要求的程序，可以选择C2。

但无论是-client还是-server，C1和C2都是有参与编译工作的。这种方式成为混合模式（mixed），是默认的方式，可以通过java -version看出：

```
C:\Users\Lord_X_>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)
```

最后一行的mixed mode说明了这一点。

我们也可以通过-Xint参数强行指定只使用解释模式，此时即时编译器完全不参与工作，java -versio的最后一行会显示interpreted mode。

可以通过参数-Xcomp强行指定只使用编译模式，此时程序启动后就会直接对所有代码进行编译，这方式会拖慢启动时间，但启动后由于省去了解释执行和C1、C2的编译时间，代码执行效率会提升很。此时java -version的最后一行会显示compiled mode。

下面通过一段代码来对比一下三种模式的执行效率（一个简陋的性能）：

```
public class JITDemo2 {

    private static Random random = new Random();

    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        int count = 0;
        int i = 0;
        while (i++ < 99999999){
            count += plus();
        }
        System.out.println("time cost : " + (System.currentTimeMillis() - start));
    }

    private static int plus() {
        return random.nextInt(10);
    }
}
```

- 首先是纯解释执行模式

添加虚拟机参数：-Xint -XX:+PrintCompilation（打印编译信息）

执行结果：

```
JITDemo2
E:\developer\jdk8\bin\java -XX:+PrintCompilation -Xint "
time cost : 21160

Process finished with exit code 0
```

编译信息没有打印出来，侧面证明了即时编译器没有参与工作。

- 然后是纯编译执行模式

添加虚拟机参数：-Xcomp -XX:+PrintCompilation

执行结果：

```
JITDemo2
  2563 1211  s b  3       java.io.BufferedOutputStream::flush (12 bytes)
  2563 1212  s b  4       java.io.BufferedOutputStream::flush (12 bytes)
  2564 1211  s    3       java.io.BufferedOutputStream::flush (12 bytes)   made not entra
  2564 1213    b  3       java.io.FileOutputStream::write (12 bytes)
  2564 1214    b  4       java.io.FileOutputStream::write (12 bytes)
  2565 1213       3       java.io.FileOutputStream::write (12 bytes)   made not entrant
time cost : 1653  2565 1215    b  3       java.io.OutputStream::flush (1 bytes)
  2565 1216    b  1       java.io.OutputStream::flush (1 bytes)
  2565 1215       3       java.io.OutputStream::flush (1 bytes)   made not entrant
  2566 1217   !b  3       java.io.PrintStream::newLine (73 bytes)
  2566 1218   !b  4       java.io.PrintStream::newLine (73 bytes)
```

会产生大量的编译信息

- 最后是混合模式

添加虚拟机参数：-XX:+PrintCompilation

执行结果：

```
JITDemo2
   156   44      4       java.util.Random::next (47 bytes)
   156   40      3       java.util.Random::nextInt (74 bytes)   made not entrant
   156   45      4       com.example.demo.gcdemo.JITDemo2::plus (9 bytes)
   157   41      3       java.util.Random::next (47 bytes)   made not entrant
   157   38      3       com.example.demo.gcdemo.JITDemo2::plus (9 bytes)   made not entrant
   158   46 %    3       com.example.demo.gcdemo.JITDemo2::main @ 9 (58 bytes)
   159   47      3       com.example.demo.gcdemo.JITDemo2::main (58 bytes)
   159   48 %    4       com.example.demo.gcdemo.JITDemo2::main @ 9 (58 bytes)
   161   46 %    3       com.example.demo.gcdemo.JITDemo2::main @ -2 (58 bytes)   made not entrant
  1456   48 %    4       com.example.demo.gcdemo.JITDemo2::main @ -2 (58 bytes)   made not entrant
time cost : 1310

Process finished with exit code 0
```

结论：耗时由大到小排序为：纯解释模式 > 纯编译模式 > 混合模式

但这里只是一个很简短的程序，如果是长时间运行的程序，不知纯编译模式的执行效率会否高于混合式，而且这个测试方式并不严格，最好的方式应该是在严格的基准测试下测试。

2 何时触发即时编译

即时编译器触发的根据有两个方面：

- 方法的调用次数
- 循环回边的执行次数

JVM在调用一个方法时，会在计数器上+1，如果方法里面有循环体，每次循环，计数器也会+1。

在不启用分层编译时（下篇会介绍），当某一方法的计数器达到由参数-XX:CompileThreshold指定值时（C1为1500，C2为10000），就会触发即时编译。

下面做个关闭分层编译时，即时编译触发的实验：

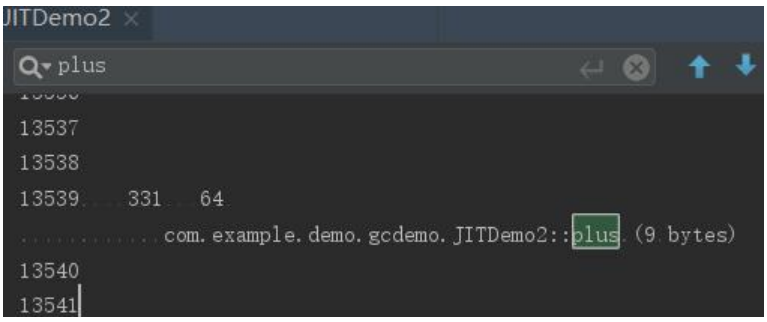
- 首先是根据方法调用触发（不涉及循环）

```
// 参数：-XX:+PrintCompilation -XX:-TieredCompilation（关闭分层编译）
public class JITDemo2 {
    private static Random random = new Random();

    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        int count = 0;
        int i = 0;
        while (i++ < 15000){
            System.out.println(i);
            count += plus();
        }
        System.out.println("time cost : " + (System.currentTimeMillis() - start));
    }

    // 调用时，编译器计数器+1
    private static int plus() {
        return random.nextInt(10);
    }
}
```

执行结果如下：



由于解释执行时的计数工作并没有严格与编译器同步，所以并不会是严格的10000，其实只要调用次足够大，就可以视为热点代码，没必要做到严格同步。

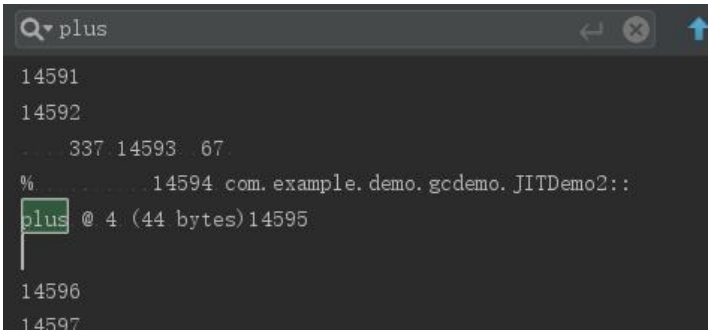
- 根据循环回边

```
public class JITDemo2 {
    private static Random random = new Random();

    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        plus();
        System.out.println("time cost : " + (System.currentTimeMillis() - start));
    }

    // 调用时，编译器计数器+1
    private static int plus() {
        int count = 0;
        // 每次循环，编译器计数器+1
        for (int i = 0; i < 15000; i++) {
            System.out.println(i);
            count += random.nextInt(10);
        }
        return random.nextInt(10);
    }
}
```

执行结果：



- 根据方法调用和循环回边

PS：每次方法调用中有10次循环，所以每次方法调用计数器应该+11，所以应该会在差不多大于1000/11=909次调用时触发即时编译。

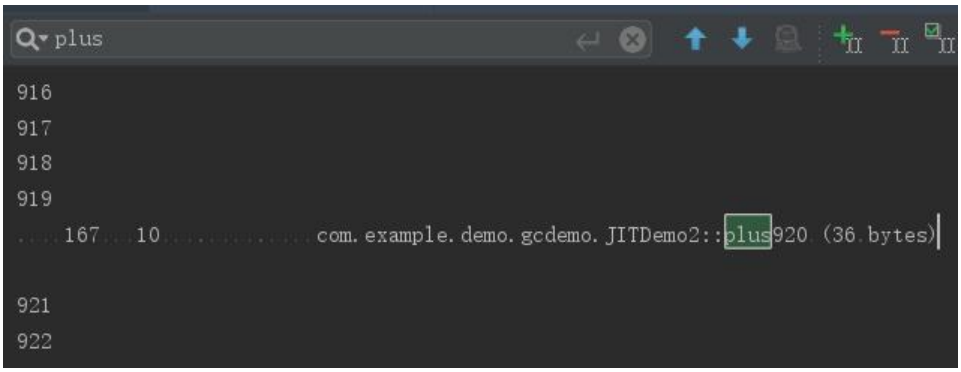
```
public class JITDemo2 {
    private static Random random = new Random();

    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        int count = 0;
        int i = 0;
        while (i++ < 15000) {
            System.out.println(i);
            count += plus();
        }
```

```
        System.out.println("time cost : " + (System.currentTimeMillis() - start));
    }

    // 调用时，编译器计数器+1
    private static int plus() {
        int count = 0;
        // 每次循环，编译器计数器+1
        for (int i = 0; i < 10; i++) {
            count += random.nextInt(10);
        }
        return random.nextInt(10);
    }
}
```

执行结果：



3 CodeCache

CodeCache是热点代码的暂存区，经过即时编译器编译的代码会放在这里，它存在于堆外内存。

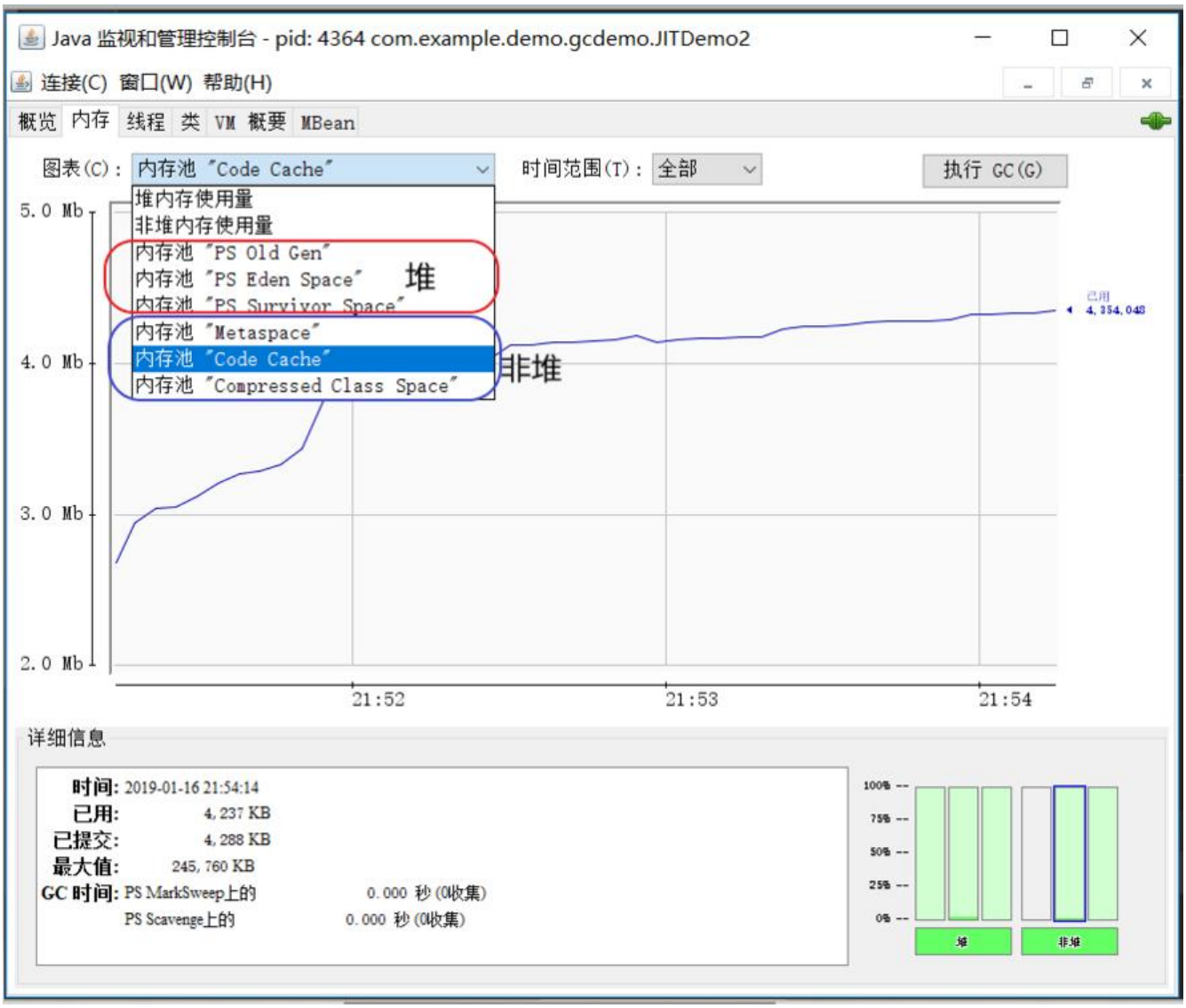
-XX:InitialCodeCacheSize和-XX:ReservedCodeCacheSize参数指定了CodeCache的内存大小。

- -XX:InitialCodeCacheSize：CodeCache初始内存大小，默认2496K
- -XX:ReservedCodeCacheSize：CodeCache预留内存大小，默认48M

PS：可以通过-XX:+PrintFlagsFinal打印出所有参数的默认值。

3.1 通过jconsole监控CodeCache

可以通过JDK自带的jconsole工具看到CodeCache在内存中所处的位置，例如



从图中曲线图可以看出CodeCache已经使用了4M多。

3.2 CodeCache满了会怎样

平时我们为一个应用分配内存时往往会忽略CodeCache，CodeCache虽然占用的内存空间不大，而他也有GC，往往不会被填满。但如果CodeCache一旦被填满，那对于一个QPS高的、对性能有高要的应用来说，可以说是灾难性的。

通过上文的介绍，我们知道JVM内部会先尝试解释执行Java字节码，当方法调用或循环回边达到一定数时，会触发即时编译，将Java字节码编译成本地机器码以提高执行效率。这个编译的本地机器码是存在CodeCache中的，如果有大量的代码触发了即时编译，而且没有及时GC的话，CodeCache就会填满。

一旦CodeCache被填满，已经被编译的代码还会以本地代码方式执行，但后面没有编译的代码只能解释执行的方式运行。

通过第2小节的比较，可以清晰看出解释执行和编译执行的性能差异。所以对于大多数应用来说，这情况的出现是灾难性的。

CodeCache被填满时，JVM会打印一条日志：

```
...29 common frames omitted
Caused by: java.lang.NoSuchMethodError: java.lang.invoke.MethodHandle.linkToSpecial(Ljava/lang/Object;Ljava/lang/
    at java.lang.invoke.MethodHandleNatives.resolve(Native Method) ~[na:1.8.0_121]
    at java.lang.invoke.MemberName$Factory.resolve(MemberName.java:975) ~[na:1.8.0_121]
    at java.lang.invoke.MemberName$Factory.resolveOrFail(MemberName.java:1000) ~[na:1.8.0_121]
...30 common frames omitted
Caused by: java.lang.VirtualMachineError: out of space in CodeCache for method handle intrinsic
...33 common frames omitted

CodeCache: size=5120Kb used=4618Kb max_used=4618Kb free=501Kb
 bounds [0x0000000002a70000, 0x0000000002f70000, 0x0000000002f70000]
 total_blobs=2505 nmethods=2093 adapters=322
 compilation: disabled (not enough contiguous free space left)
Java HotSpot(TM) 64-Bit Server VM warning: CodeCache is full. Compiler has been disabled.
Java HotSpot(TM) 64-Bit Server VM warning: Try increasing the code cache size using -XX:ReservedCodeCacheSize=
```

JVM针对CodeCache提供了GC方式： -XX:+UseCodeCacheFlushing。在JDK1.7.0_4之后这个参数认开启，当CodeCache即将填满时会尝试回收。JDK7在这方面的回收做的不是很少，GC收益较低，JDK8有了很大的改善，所以可以通过升级到JDK8来直接提升这方面的性能。

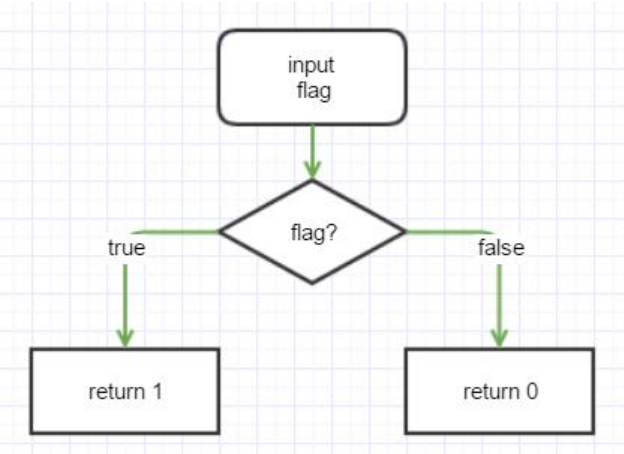
3.3 CodeCache的回收

那么什么时候CodeCache中被编译的代码是可以回收的呢？

这要从编译器的编译方式说起。举个例子，下面这段代码：

```
public int method(boolean flag) {
    if (flag) {
        return 1;
    } else {
        return 0;
    }
}
```

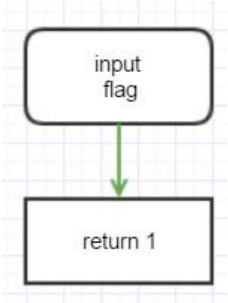
从解释执行的角度来看，他的执行过程如下：



但经过即时编译器编译后的代码不一定是这样，即时编译器在编译前会收集大量的执行信息，例如，果这段代码之前输入的flag值都为true，那么即时编译器可能会将他变异成下面这样：

```
public int method(boolean flag) {
    return 1;
}
```

即下图这样



但可能后面不总是flag=true，一旦flag传了false，这个错了，此时编译器就会将他“去优化”，变编译执行方式，在日志中的表现是made not entrant：

```
12569 3048 1 org.springframework.core.annotation.AnnotationUtils::isInJavaLangAnnotationPackage (19 bytes) ma
12569 3046 1 java.util.LinkedHashMap$LinkedValues::size (8 bytes) made not entrant
12569 3044 1 java.util.AbstractList$Itr::<init> (6 bytes) made not entrant
12569 3045 1 java.util.AbstractList$Itr::<init> (31 bytes) made not entrant
12569 3043 1 java.util.ArrayList::rangeCheckForAdd (26 bytes) made not entrant
12569 3041 1 com.fasterxml.jackson.databind.type.ClassKey::hashCode (5 bytes) made not entrant
12569 3049 1 org.apache.catalina.LifecycleState::isAvailable (5 bytes) made not entrant
12569 3050 1 java.beans.Introspector::makeQualifiedMethodName (50 bytes) made not entrant
12569 3053 1 org.springframework.core.SerializableTypeWrapper::forTypeProvider (173 bytes) made not entrant
12569 3052 1 org.springframework.beans.GenericTypeAwarePropertyDescriptor::<init> (328 bytes) made zombie
12569 3058 1 java.util.Collections$EmptySet::toArray (11 bytes) made not entrant
```

此时该方法不能再进入，当JVM检测到所有线程都退出该编译后的made not entrant，会将该方法记为：made zombie，此时 这块代码占用的内存就是可回收的了。可以通过编译日志看出：

```
23364 2856 1 java.util.HashMap::<init> (98 bytes) made zombie
23364 2853 1 java.util.jar.Attributes::putValue (17 bytes) made zombie
23365 2876 ! 1 java.util.zip.Inflater::setInput (74 bytes) made zombie
23365 2874 1 java.util.jar.JarInputStream::read (48 bytes) made zombie
23365 2875 1 java.io.InputStream::<init> (5 bytes) made zombie
23365 2873 1 java.io.PushbackInputStream::read (145 bytes) made zombie
23365 2865 1 java.io.PushbackInputStream::ensureOpen (18 bytes) made zombie
23365 2858 1 java.util.ArrayList::<init> (12 bytes) made zombie
23365 2854 1 java.io.BufferedInputStream::read1 (108 bytes) made zombie
23365 2852 1 java.util.jar.Attributes$Name::<init> (48 bytes) made zombie
23365 2864 1 java.util.Dictionary::<init> (5 bytes) made zombie
23365 2868 ! 1 java.lang.ref.Finalizer::add (41 bytes) made zombie
23365 2863 1 java.util.Hashtable::<init> (114 bytes) made zombie
```

3.4 CodeCache的调优

在Java8中提供了一个JVM启动参数：-XX:+PrintCodeCache，他可以在JVM停止时打印CodeCach的使用情况，可以在每次停止应用时观察一下这个值，慢慢调整为一个最合适的大小。

以一个SpringBoot的Demo说明一下：

```
// 启动参数：-XX:ReservedCodeCacheSize=256M -XX:+PrintCodeCache
@RestController
@SpringBootApplication
public class DemoApplication {
  // ... other code ...

  public static void main(String[] args) {
    SpringApplication.run(DemoApplication.class, args);
    System.out.println("start....");
    System.exit(1);
  }
}
```

这里我将CodeCache定义为256M，并在JVM退出时打印了CodeCache使用情况，日志如下：

```
2019-01-16 23:19:06.476  INFO 46364 --- [          main] o.s.b.w.embed
2019-01-16 23:19:06.479  INFO 46364 --- [          main] com.example.d
start....
2019-01-16 23:19:06.481  INFO 46364 --- [      Thread-7] ConfigServlet
2019-01-16 23:19:06.483  INFO 46364 --- [      Thread-7] o.s.j.e.a.Ann
CodeCache: size=262144Kb used=6721Kb max_used=6721Kb free=255422Kb
 bounds [0x0000000003790000, 0x0000000003e30000, 0x0000000013790000]
 total_blobs=3566 nmethods=3146 adapters=332
 compilation: enabled

Process finished with exit code 1
```

最多只使用了6721K（max_used），浪费了大量的内存，此时就可以尝试将-XX:ReservedCodeCaceSize=256M调小，将多余的内存分配给别的地方。

4 参考文档

[1] https://blog.csdn.net/yandaonan/article/details/50844806

[2] 深入理解Java虚拟机 周志明 第11章

[3] 极客时间《深入拆解Java虚拟机》 郑雨迪