JVM进阶 -- 浅谈即时编译

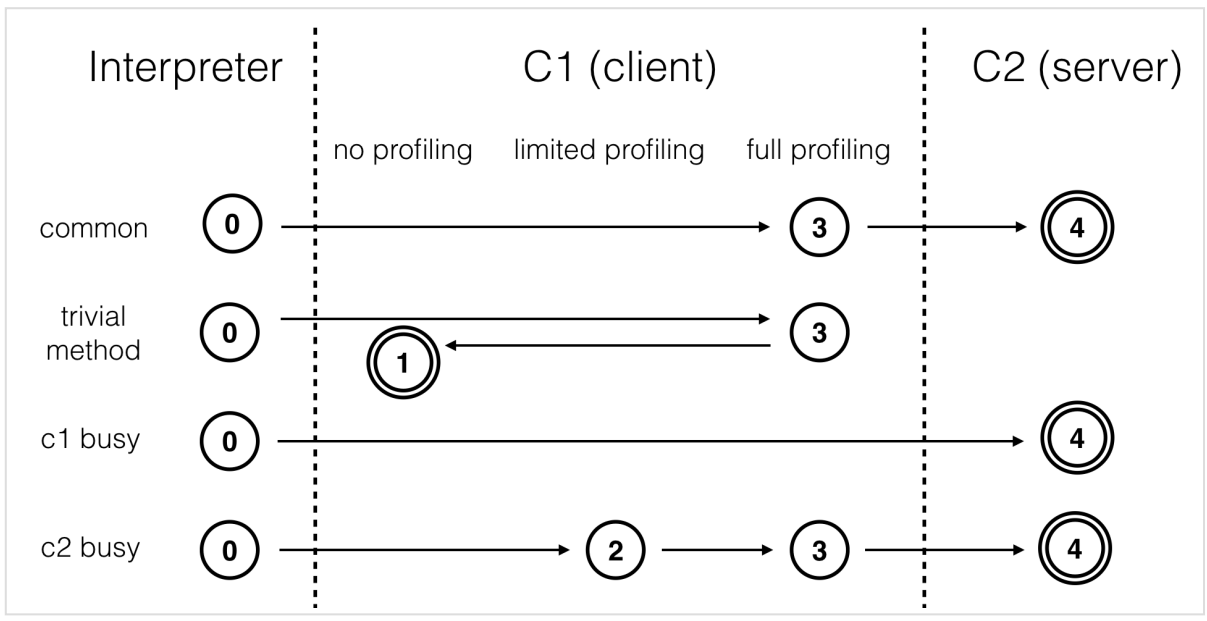
概念

1. 即时编译是用来**提升应用运行效率**的技术
2. 代码会先在JVM上**解释执行**，之后反复执行的**热点代码**会被**即时翻译成为机器码**，直接运行在**底层硬件**上

分层编译

1. HotSpot包含多个即时编译器：C1、C2和Graal（Java 10，实验性）
2. 在Java 7之前，需要根据程序的特性选择对应的**即时编译器**
    - 对于**执行时间较短**或**对启动性能有要求**的程序，采用**编译效率较快的C1**，对应参数：`-client`
    - 对于**执行时间较长**或**对峰值性能有要求**的程序，采用**生成代码执行效率较快的C2**，对应参数：`-server`
3. Java 7引入了**分层编译**（-XX:+TieredCompilation），综合了**C1的启动性能优势**和**C2的峰值性能优势**
4. 分层编译将**JVM的执行状态**分了5个层次
    - 0：解释执行（也会profiling）
    - 1：执行**不带profiling**的C1代码
    - 2：执行仅带**方法调用次数**和**循环回边执行次数**profiling的C1代码
    - 3：执行带**所有profiling**的C1代码
    - 4：执行C2代码
5. 通常情况下，**C2代码的执行效率比C1代码高出30%以上**
6. 对于C1代码的三种状态，按执行效率从高至低：1层 > 2层 > 3层
    - **1层的性能略高于2层，2层的性能比3层高出30%**
    - **profiling越多，额外的性能开销越大**
7. profiling：在程序执行过程中，收集能够反映程序执行状态的数据
    - profile：收集的数据
    - JDK附带的**hprof**（CPU+Heap）
    - JVM**内置profiling**
8. Java 8默认开启了分层编译，无论开启还是关闭分层编译，原本的`-client`和`-client`都是无效的
    - 如果**关闭分层编译**，JVM将直接采用**C2**
    - 如果只想用C1，在打开分层编译的同时，使用参数：-XX:TieredStopAtLevel=1

编译路径



1. 1层和4层是**终止状态**
    - 当一个**方法**被**终止状态**编译后，如果**编译后的代码没有失效**，那么JVM**不会再次发出该方法的编译请求**
2. 通常情况下，热点方法会被3层的C1编译，然后再被4层的C2编译
3. 如果方法的**字节码数目较少**（如getter/setter），并且**3层的profiling没有可收集的数据**
    - JVM会断定**该方法对于C1和C2的执行效率相同**
    - JVM会在3层的C1编译后，**直接选用1层的C1编译**
    - 由于1层是**终止状态**，JVM不会继续用4层的C2编译
4. 在C1忙碌的情况下，JVM在**解释执行过程**中对程序进行**profiling**，而后直接由4层的C2编译
5. 在C2忙碌的情况下，方法会被2层的C1编译，然后再被3层的C1编译，以减少方法在3层的执行时间

触发JIT的条件

1. JVM是依据**方法的调用次数**以及**循环回边的执行次数**来触发JIT的
2. JVM将在0层、2层和3层执行状态时进行profiling，其中包括方法的调用次数和循环回边的执行次数
    - 循环回边是一个控制流程图中的概念，在字节码中，可以简单理解为**往回跳**的指令
    - 在即时编译过程中，JVM会识别循环的头部和尾部，**循环尾部到循环头部的控制流就是真正意义上的循环回边**
    - C1将在**循环回边**插入**循环回边计数器**的代码
    - 解释执行和C1代码中增加循环回边计数的**位置**并不相同，但这不会对程序造成影响
    - JVM不会对这些**计数器**进行**同步**操作，因此收集到的执行次数也**不是精确值**
    - 只要该数值**足够大**，就能表示对应的方法包含热点代码
3. 在**不启动**分层编译时，当**方法的调用次数和循环回边的次数的和**超过–XX:CompileThreshold，便会触发JIT
    - 使用**C1**时，该值为**1500**
    - 使用**C2**时，该值为**10000**
4. 当**启用**分层编译时，阈值大小是**动态调整**的
    - **阈值 * 系数**

系数

```
系数的计算方法：
s = queue_size_X / (TierXLoadFeedback * compiler_count_X) + 1

其中X是执行层次，可取3或者4
queue_size_X：执行层次为X的待编译方法的数目
TierXLoadFeedback：预设好的参数，其中Tier3LoadFeedback为5，Tier4LoadFeedback为3
compiler_count_X：层次X的编译线程数目。
```

编译线程数

1. 在64位JVM中，默认情况下，编译线程的总数目是根据**处理器数量**来调整的
    - –XX:+CICompilerCountPerCPU=true，**编译线程数依赖于处理器数量**
    - –XX:+CICompilerCountPerCPU=false –XX:+CICompilerCount=N，**强制设定总编译线程数**
2. JVM会将这些编译线程按照1:2的比例分配给C1和C2（至少1个），对于4核CPU，总编译线程数为3

```
// –XX:+CICompilerCountPerCPU=true
n = log2(N) * log2(log2(N)) * 3 / 2
其中 N 为 CPU 核心数目，N >= 4
```

触发条件

当启用分层编译时，触发JIT的条件

```
i > TierXInvocationThreshold * s || (i > TierXMinInvocationThreshold * s  && i + b > TierXCompileThreshold * s)
其中i为方法调用次数，b为循环回边执行次数
```

Profiling

1. 在分层编译中的0层、2层和3层，都会进行profiling，最为基础的是**方法的调用次数**以及**循环回边的执行次数**
    - 主要拥有触发JIT
2. 此外，0层和3层还会收集用于4层C2编译的数据，例如
    - **branch profiling**
        - 分支跳转字节码，包括跳转次数和不跳转次数
    - **type profiling**
        - 非私有实例方法调用指令：**invokevirtual**
        - 强制类型转换指令：**checkcast**
        - 类型测试指令：**instanceof**
        - 引用类型数组存储指令：**aastore**
3. branch profiling和type profiling将给应用带来不少的**性能开销**

- 3层C1的性能比2层C1的性能低30%
- 通常情况下，我们不会在**解析执行**过程中进行branch profiling和type profiling
  - 只有在方法**触发C1编译后**，JVM认为该方法**有可能被C2编译**，才会在该方法的C1代码中收集这些profile
- 只有在**极端**情况下（如等待C1编译的方法数目太多），才会开始在**解释**执行过程中收集这些profile
- C2可以根据收集得到的数据进行**猜测和假设**，从而作出比较**激进的优化**

branch profiling

Java代码

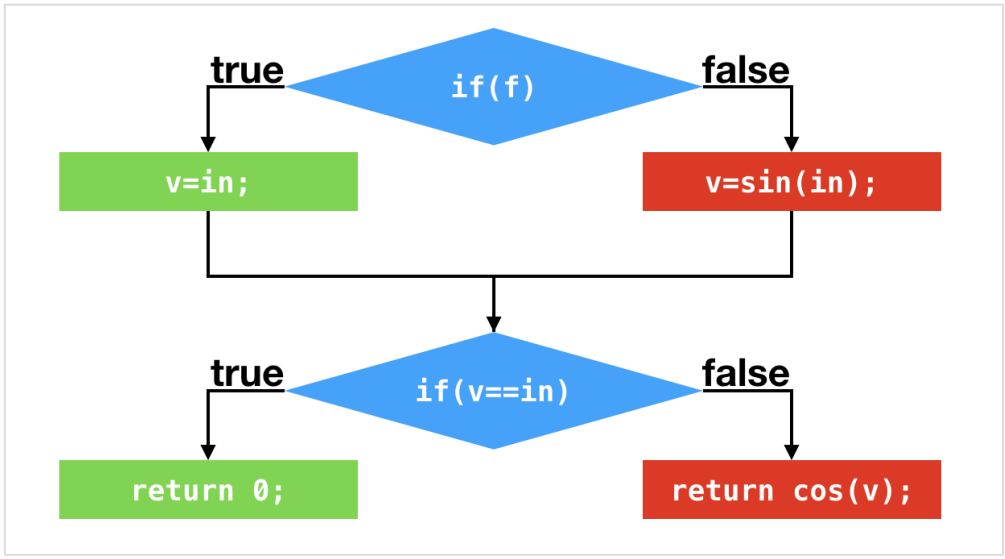
```
public static int foo(boolean f, int in) {
    int v;
    if (f) {
        v = in;
    } else {
        v = (int) Math.sin(in);
    }
    if (v == in) {
        return 0;
    } else {
        return (int) Math.cos(v);
    }
}
```

字节码

```
public static int foo(boolean, int);
  descriptor: (ZI)I
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=3, args_size=2
       0: iload_0
       1: ifeq          9       // false，跳转到偏移量为9的字节码
       4: iload_1
       5: istore_2
       6: goto          16
       9: iload_1
      10: i2d
      11: invokestatic          // Method java/lang/Math.sin:(D)D
      14: d2i
      15: istore_2
      16: iload_2
      17: iload_1
      18: if_icmpne     23      // 如果v!=in，跳转到偏移量为23的字节码
      21: iconst_0
      22: ireturn
      23: iload_2
      24: i2d
      25: invokestatic          // Method java/lang/Math.cos:(D)D
      28: d2i
      29: ireturn
```

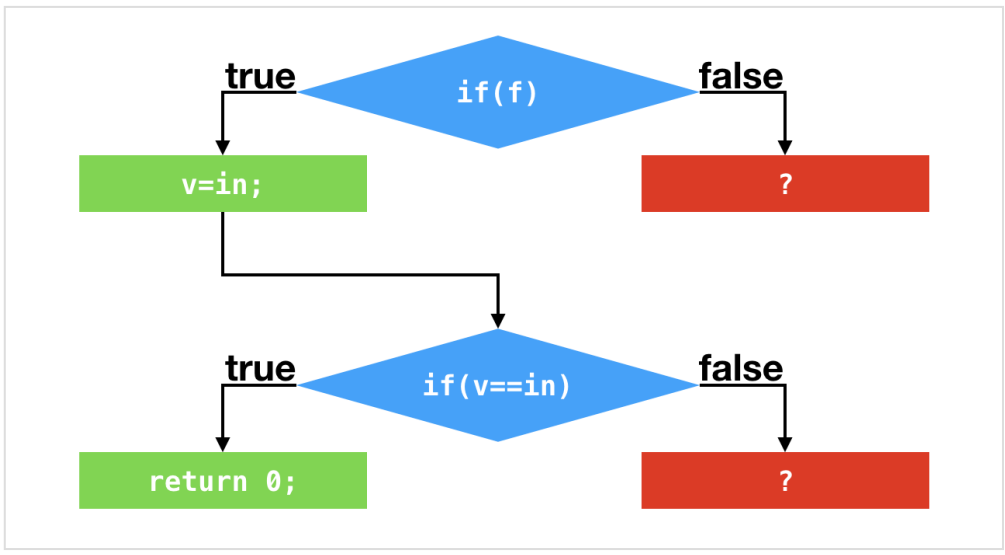
优化过程

正常分支



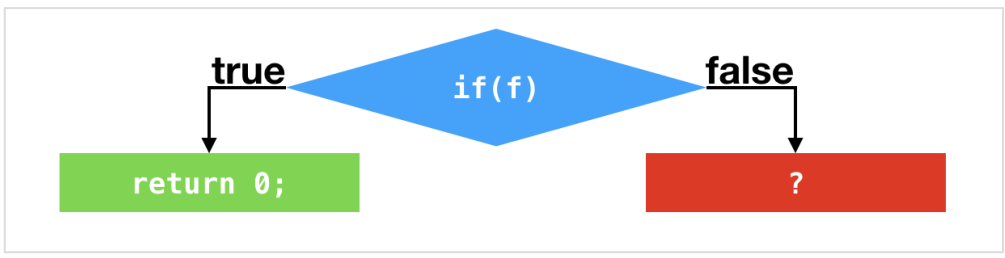
profiling

假设应用程序调用该方法，所传入的都是true，那么偏移量为1和偏移量为18的条件跳转指令所对应的分支profile中，其跳转的次数都是0。实际执行的分支如下：



剪枝

C2根据这两个分支profile作出假设，在后续的执行过程中，这两个条件跳转指令仍旧不会执行，基于这个假设，C2不会在编译这两个条件跳转语句所对应的false分支（剪枝）。最终的结果是在第一个条件跳转之后，C2代码直接返回0



小结

1. 根据条件跳转指令的分支profile，即时编译器可以将**从未执行过**的分支减掉
    - 避免编译这些不会用到的代码
    - 节省**编译时间**以及部署代码所要消耗的**内存空间**
2. 剪枝同时也能精简数据流，从而触发更多的优化
3. 现实中，分支profile出现仅跳转或者不跳转的情况并不常见
4. 即时编译器对分支profile的利用也不仅仅限于剪枝
    - 还可以依据分支profile，**计算每一条执行路径的概率**
    - 以便于某些编译器优化优先处理概率较高的路径

type profiling

Java代码

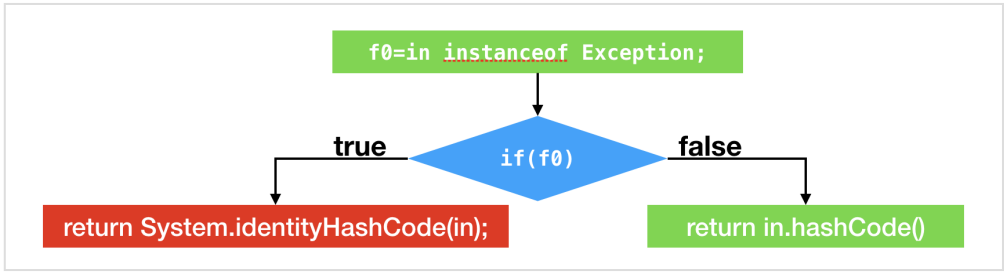
```
public static int hash(Object in) {
    if (in instanceof Exception) {
        return System.identityHashCode(in);
    } else {
        return in.hashCode();
    }
}
```

字节码

```
public static int hash(java.lang.Object);
  descriptor: (Ljava/lang/Object;)I
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
       0: aload_0
       1: instanceof        // class java/lang/Exception
       4: ifeq          12  // 不是Exception，跳转到偏移量为12的字节码
       7: aload_0
       8: invokestatic      // Method java/lang/System.identityHashCode:(Ljava/lang/Object;)I
      11: ireturn
      12: aload_0
      13: invokevirtual     // Method java/lang/Object.hashCode:()I
      16: ireturn
```

优化过程

正常分支



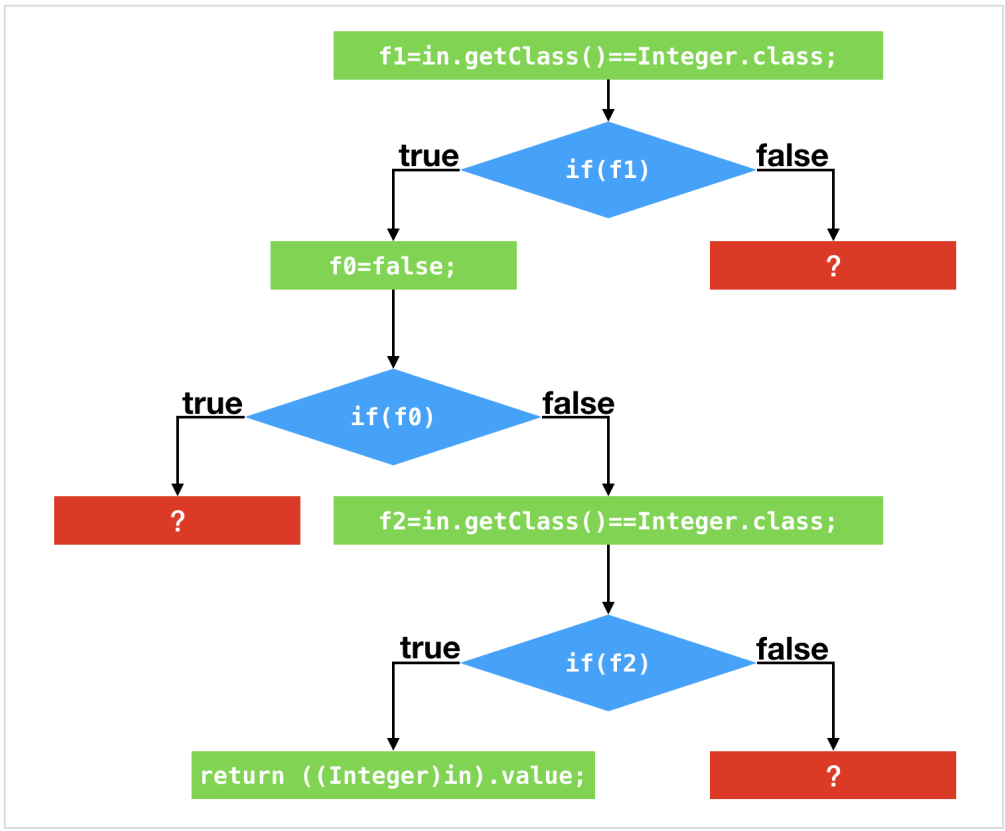
profiling+优化

1. 假设应用调用该方法时，所传入的Object皆为Integer实例
    - 偏移量为1的**instanceof**指令的**类型profile**仅包含Integer
    - 偏移量为4的分支跳转语句的**分支profile**不跳转次数为0
    - 偏移量为13的方法调用指令的**类型profile**仅包含Integer
2. 测试instanceof
    - 如果instanceof的**目标类型是final类型**，那么JVM仅需比较测试**对象的动态类型**是否为该final类型
    - 如果**目标类型不是final类型**，JVM需要依次按下列顺序测试是否与目标类型一致
        - 该类本身
        - 该类的父类、祖先类
        - 该类所直接实现或间接实现的接口
3. instanceof指令的类型profile仅包含Integer
    - JVM会假设在接下来的执行过程中，所输入的Object对象仍为Integer对象
    - 生成的代码将**直接测试所输入的动态类型是否为Integer**，如果是继续执行接下来的代码
4. 然后，即时编译器会采用**针对分支profile的优化**以及**对方法调用的条件去虚化内联**
    - 内联结果：生成的代码将测试所输入对象的动态类型是否为Integer，如果是，执行`Integer.hashCode()`方法的代码

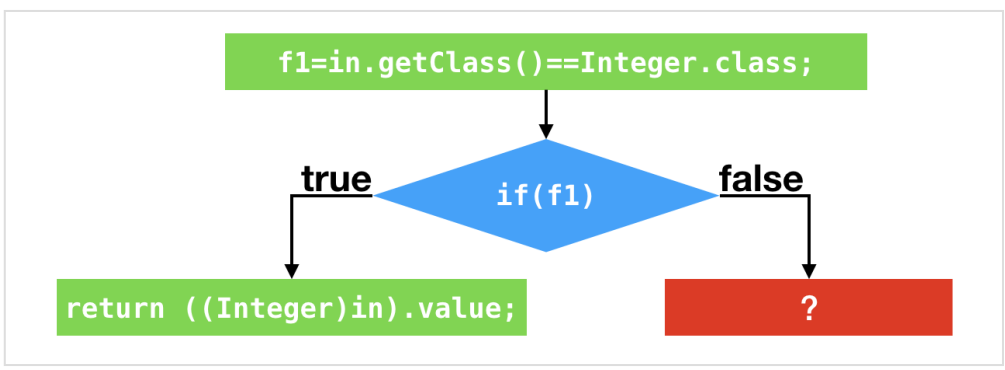
```
public final class Integer ... {
    @Override
    public int hashCode() {
        return Integer.hashCode(value);
    }

    public static int hashCode(int value) {
        return value;
    }
}
```

针对上面三个profile的分支图



进一步优化（剪枝）



小结

1. 和基于分支profile的优化一样，基于类型profile的优化同样也是作出假设，从而精简控制流以及数据流，两者的**核心是假设**
2. 对于**分支profile**，即时编译器假设**仅执行某一分支**
3. 对于**类型profile**，即时编译器假设的是**对象的动态类型仅为类型profile中的那几个**
4. 如果**假设失败**，将进入**去优化**

去优化

1. 去优化：从执行即时编译生成的机器码**切回解释执行**
2. 在生成的机器码中，即时编译器将在**假设失败**的位置插入一个**陷阱**（trap）
    - 陷阱实际上是一条**call指令**，调用至JVM**专门负责去优化的方法**
    - 上图红色方框的问号，便代表陷阱
3. 去优化的过程很复杂，由于即时编译器采用了许多优化方式，其生成的代码和原本字节码的差异非常大
4. 在去优化的过程中，**需要将当前机器码的执行状态切换至某一字节码之前的执行状态，并从该字节码开始执行**
    - 要求即时编译器在编译过程中记录好这两种执行状态的**映射**
5. 在调用JVM的去优化方法时，即时编译器生成的机器码可以根据**产生去优化的原因**决定**是否保留这份机器码**，以及**何时重新编译对应的Java代码**
    - 如果去优化的原因**与优化无关**
        - 即使重新编译也不会改变生成的机器码，那么生成的机器码可以在调用去优化代码时传入Action_None
        - **表示保留这一份机器码，在下次调用该方法时重新进入这一份机器码**
    - 如果去优化的原因**与静态分析的结果有关**，例如类层次分析
        - 那么生成的机器码可以在调用去优化方法时传入Action_Recompile
        - **表示不保留这一份机器码，但是可以不经过重新收集profile，直接重新编译**
    - 如果去优化的原因**与基于profile的激进优化有关**
        - 那么生成的机器码需要在调用去优化方法时传入Action_Reinterpret
        - **表示不保留这一份机器码，并且需要重新收集profile，再重新编译**
        - 因为之前收集到的profile已经**不能准确**反映程序的运行情况，需要重新收集