文章已收录我的仓库：Java学习笔记与免费书籍分享

JIT 即时编译及优化技术

前言

我相信很多人都知道 Java 是一门解释性语言，不同与 C/C++，Javac编译器将 .java 文件编译成 .class 文件，然后由 JVM 运行时读取 .class 指令，一条一条的解释指令，将 .class 指令逐条的翻译成机器码运行，这有有点也有缺点，让我们来具体探讨一下此方面的点点滴滴。

即时编译

编译与解释是不同的概念，上面我们说解释是将字节码在运行时**逐条**地翻译成机器码执行，而编译是提前**一次性**地翻译成机器码，这样就无需运行时去慢慢的解释了，虽然增加了编译的时间，但运行效率却大大提高了。

而且，编译后的机器代码是完全可以复用的，通俗点讲，经过编译后的机器码是多次运行的，而只需编译一次；而对于解释型语言来说，无论运行多少次重复代码，都需要一条一条的去解释执行。这就是JVM编译优化的思考方向：提前编译热点代码。

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称为即时编译器（Just In Time Compiler），简称 JIT 编译器。

热点代码探测

什么会成为热点代码？

- 被多次调用的方法。一个方法被调用得多了，方法体内代码执行的次数自然就多，成为“热点代码”是理所当然的。
- 被多次执行的循环体。一个方法只被调用过一次或少量的几次，但是方法体内部存在循环次数较多的循环体，这样循环体的代码也被重复执行多次，因此这些代码也应该认为是“热点代码”。

如何检测热点代码？

- 基于采样的热点探测：采用这种方法的虚拟机会周期性地检查各个线程的栈顶如果发现某个（或某些）方法经常出现在栈顶，那这个方法就是“热点方法”
    - 优点：实现简单高效，容易获取方法调用关系（将调用堆栈展开即可）
    - 缺点：不精确，容易因为因为受到线程阻塞或别的外界因素的影响而扰乱热点探测
- 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（甚至是代码块）建立计数器，统计方法的执行次数，如果次数超过一定的阈值就认为它是“热点方法”
    - 优点：统计结果精确严谨
    - 缺点：实现麻烦，需要为每个方法建立并维护计数器，不能直接获取到方法的调用关系

大多数虚拟机往往采用第二种方法，既然有两种代码(方法和循环体)可能会成为热点代码，那么也会存在两种计数器：

- 方法调用计数器：这个计数器用于统计方法被调用的次数。默认阈值在 Client 模式下是 1500 次，在 Server 模式下是 10000 次。
- 回边计数器：统计一个方法中循环体代码执行的次数。具体的，在遇到控制流向后跳转的指令会被记为回边。

解释一下`控制流向后跳转的指令会被记为回边`这句话，这里的向后指的是时间向后，考虑下面循环代码：

```
flag = xxx;
while (flag) {
    do...somthing
}
```

事实上这种循环代码如果编译后大概是这个样子:

```
flag = xxx;
do:
        do...somthing
if (flag) {
```

```
    goto do;
}
```

可以看到循环体编译后会存在一个 goto 控制流跳转，回边检测就是检测类似的控制流。

当计数器超过阈值时，如果已经存在编译的版本，则直接运行编译的版本；否则，JVM就会提交一个 OSR 编译请求，请求即时编译器编译热点代码，然后将计数器的值降低至阈值之下，等待编译完成。

编译优化技术

我们都知道C语言中可以通过开启 `-o1 -o2` 等参数开启编译优化，编译优化会消耗较长的时间，但优化后的代码质量高，运行效率高，Java官方也一直在不停的提高即时编译器的优化效果。

Java中存在两种即时编译器，即C1、C2编译器，C1是客户端下的编译器，此类编译器通常不会进行过多的优化，只是正常的编译代码，编译速度快，但运行效率一般；C2是服务器下默认的编译器，此类编译器往往会进行大量的优化操作，编译速度缓慢，但编译后运行效率高。

语言无关的经典优化技术之一：公共子表达式消除

如果一个表达式 E 已经计算过了，并且从先前的计算到现在 E 中所有变量的值都没有发生变化，那么 E 的这次出现就成为了公共子表达式。对于这种表达式，没必要花时间再对它进行计算，只需要直接使用前面计算过的表达式结果代替 E 就可以了。

例子：`int d = (c * b) * 12 + a + (a + b * c)`

可能会被替换为：`int d = E * 12 + a + (a + E)`，此时 b * c 只会被计算一次。

语言相关的经典优化技术之一：数组范围检查消除

Java语言是相对安全的语言，这是因为 Java 语言作为很多隐式的判断，例如在访问数组时 JVM 会先进行判断，判断下标是否合法；又例如进行除法前，Java 会先判断除数是否为0...如果不合法，Java会友善的抛出异常提醒程序员，而程序员可以捕捉这次异常。但对于 C/C++ 开发者就没那么幸运了，一旦发生异常，C/C++并不会进行这些检查，如果数组真的越界了，那么就非法操作了其他内存的数据，程序却不会停止，除非访问了操作系统禁止访问的段，这是相当糟糕的。

但 Java 多了许多的隐式判断，这也导致了“Java天生就慢人一等”，一个优化技术是，如果在编译器就确保操作是合法的，则可以去掉这些隐式的判断。

例如:

```
int[] a = new int[3];
a[2] = 1;
```

像上面这种代码编译器就能确定是合法的，则可直接去掉判断这一类字节码不编译。

最重要的优化技术之一：方法内联

方法内联是编译优化技术之母，即将方法体原封不动的“复制”到发起调用的方法之中，从而避免真实的方法调用，为其他优化技术打下基础。例如：

```
static void foo(Object o) {
    if (o != null) {
        do....
    }
}

//测试的代码
static void test() {
    Object o = null;
    foo(o);
}

//内联的 test 代码
static void test() {
    Object o = null;
    if (o != null) {
        do....
    }
}
//通过方法内联后，编译器可以消除无用代码，即不会执行 test() 代码
```

但是Java中的方法内联是非常困难的，原因在于Java中几乎所有的方法都是虚方法，可能在父类或者子类中有不同的实现，Java中只有对构造器、静态方法可能会展开内联。C 语言中不存在虚方法，因此 C 语言可以达到极高的优化效率。

但在 JIT 技术越来越成熟的今天，JIT 可以在运行时确定甚至是预测虚方法，然后利用 JIT 即时编译进行方法内联，由于 JIT 是 JVM 运行时编译，即使预测出错也可以 “变量回归（归零）”，重新开始运行，这是纯编译器无法做到的。

函数内联的有点很明显：避免过程调用（参数保存等工作，内核态切换），将过程间分析转为过程内分析。

函数内联也是有缺点的：会使函数变得庞大，原本几行的函数内联后可能变得几百行，占用内存（递归函数将无限大），并且存放 CPU 指令缓存时可能会导致更多的不命中。

最前沿的优化技术之一：逃逸分析

Java语言所有对象都在堆上分配，这引起不少人的诟病，有时候我们仅仅只是想简单的使用一个对象中的一个变量或者一个方法，却不得不去 Java 堆中分配，但要知道 Java 垃圾回收是需要耗费大量时间的，甚至是 “STW” 级别的，如果一个对象在方法栈帧中不会

逃逸，即不会被其他方法或其他线程访问，那么该对象被认为是不共享的，可以在栈上进行分配，减轻垃圾回收的压力。

如果能证明一个对象不会逃逸到方法或线程之外，也就是别的方法或线程无法通过任何途径访问到这个对象，则可以为这个变量进行一些高效的优化：

- **栈上分配**：将不会逃逸的局部对象分配到栈上，那对象就会随着方法的结束而自动销毁，减少垃圾收集系统的压力。
- **同步消除**：如果该变量不会发生线程逃逸，也就是无法被其他线程访问，那么对这个变量的读写就不存在竞争，可以将同步措施消除掉。
- **标量替换**：标量是指无法在分解的数据类型，比如原始数据类型以及reference类型。而聚合量就是可继续分解的，比如 Java 中的对象。标量替换如果一个对象不会被外部访问，并且对象可以被拆散的话，真正执行时可能不创建这个对象，而是直接创建它的若干个被这个方法使用到的成员变量来代替。这种方式不仅可以让对象的成员变量在栈上分配和读写，还可以为后后续进一步的优化手段创建条件。

  例如`P p = new P(); int x = p.x; return x;`这段代码，如果确定p不会逃逸，那么可能会直接进行标量替换，JVM不会分配P对象，而是直接分配 x 字段基本类型。

逃逸分析的论文很早就有了，尽管如此，逃逸分析仍然是项复杂的技术，例如下面代码：

```
class Test {
    private Strint str;

    public String getStr(){return str;}

    public void test() {
        String s = new Str();
        str = s;
        System.out.println(str);
    }
}
```

可以看到即使 test() 方法内 s 对象看起来好像未发送逃逸，但 s 将自己赋值给了可能发送逃逸的 str，因此 s 仍然有可能会发生逃逸。

Java中使用相对简单的逃逸分析，一旦对象存在发送逃逸的可能性，JVM均认为该对象逃逸，则避免优化。

要怎么判断呢，一个简单的算法是，变量 P：

- P 是否传递给其他函数
- P 是否传递给全局变量
- P 是否传递给已经逃逸的对象

如果是，则认为逃逸，否则认为非逃逸。

如果存在函数内联，逃逸的概率将大大降低。

为什么不直接编译所有字节码

看了这么多你可能会想，既然编译这么牛逼，为什么JVM不直接把所有字节码一次性全部编译呢？这有如下几点原因，要知道，解释器并非一无是处。

- 全部编译编译时间长，用户等待时间长，通常客户端模式下不太可取。
- 解释器是逐条翻译的，这意味着解释器可以可以很好的根据当前记录的信息进行分支预测、虚方法调用预测、循环可能会执行多少次。即使当代处理器也具有分支预测的功能，但和解释器这类动态预测比起来确实天差地别。
- 正如刚刚所说，解释器可以对虚方法调用进行预测，如果可以预测到虚方法，那么提前编译时就可以进行方法的内联，极大的提高编译优化效果。即使错误的预测了，也可以回归解释器解释，纯编译器可不能如此的任性。

相信未来的很长一段时间解释器和编译器仍然是Java后端编译的共同技术。