11 | 即时编译：高性能JVM的核心秘密

2021-11-17 海纳

《编程高手必学的内存知识》 课程介绍 >

讲述：海纳

时长 19:59 大小 18.31M

你好，我是海纳。

在前面的课程里，我们讲解了进程内部的具体布局，以及每一个部分的功能和作用。你会发现，所有的例子都是用 C/C++ 写的，我相信你在学习的过程中，心里可能会产生这的疑问：**那 Java 和 Python 语言是怎么运行起来的呢？**

有这个疑问非常合理。我曾经讲过 C/C++ 编译的结果，它在 linux 上是 ELF 文件，在 windows 上是 exe 文件，这两种文件都可以直接被操作系统加载运行的二进制文件。另外，C/C++ 源代码也可以被编译成动态链接库文件。

而在 Java 语言里，程序员都知道 Java 源代码被 javac 编译以后，生成的是字节码文件，也就是 class 文件，而且不管编译所使用的操作系统是什么，相同的 Java 源码必然得到相同的 class 文件。class 文件显然与上面 C/C++ 编译的二进制文件都不相同，因为它与编译的平台无关。

这节课，我们就围绕着 Java 是怎么运行起来的这个问题逐层展开，在这个过程中，我会教你如何阅读和分析字节码，以及猜测它的 JIT 结果。所以通过这节课的学习，你不仅能了解到 Java 字节码的核心知识、JVM 中的解释器和 JIT 编译器的原理，而且，还能进一步理解 JVM 虚拟机。在这个基础上，你就能写出更高效、对编译器更友好的程序，而且碰到桥接方法这一类 Java 中非常抽象和难以理解的概念时，也能着手分析。

既然 Java 源代码都被编译成了字节码文件，那我们就从 Java 字节码讲起吧。

Java 字节码

我们先在某个目录下创建一个名为 Main.java 的 Java 源文件：

复制代码

```
import java.lang.Math;

public class Main {
    public static double dist(double x1, double y1, double x2, double y2) {
        double dist_x = x1 - x2;
        double dist_y = y1 - y2;
        return Math.sqrt(dist_x * dist_x + dist_y * dist_y);
    }

    public static void main(String[] args) {
        System.out.println(dist(0.0f, 0.0f, 1.0f, 1.0f));
    }
}
```

然后，在这个目录下执行：

复制代码

```
$ javac Main.java
```

这时，我们会观察到目录下多了一个名为 Main.class 的文件，我们知道这个文件就是 Java 的字节码文件。只要选择的 javac 版本相同，无论这个实验是在 windows 系统上，还是在 linux 系统上，也无论它运行在 x86 平台，还是鲲鹏平台，得到的字节码文件都是相同的。

比如，我们可以使用 java 命令来执行这个字节码文件：

复制代码

```
$ java Main
1.4142135623730951
```

也可以使用 javap 命令来观察它的字节码：

复制代码

```
$ javap -c Main.class
  public static double dist(double, double, double, double);
    Code:
       0: dload_0
       1: dload         4
       3: dsub
       4: dstore        8
       6: dload_2
       7: dload         6
       9: dsub
      10: dstore        10
      12: dload         8
      14: dload         8
      16: dmul
      17: dload         10
      19: dload         10
      21: dmul
      22: dadd
      23: invokestatic  #2                  // Method java/lang/Math.sqrt:(D)D
      26: dreturn
```

这一段代码显示的是 dist 方法编译以后的字节码。字节码的格式是：

复制代码

```
op_code(1 byte), operand1(1 byte, optional)
```

每条字节码都有自己的 op_code，然后带有 0 个或者 1 个参数。每个 op_code 在 class 文件中都是一个无符号 byte 类型的整数，刚好占据一个字节，这也是 Java 虚拟机指令被称为字节码的原因。javap 命令为了方便人的阅读，会将 op_code 翻译成文字助记符。

dist 方法的字节码从第 4 行开始，第 4 行开头的数字 0，代表的是这一条字节码的偏移量。dload_0 具体是什么含义，我们这节课后面的内容中解释。这里我们只注意这条字节码占据的宽度是 1，所以第二条字节码的偏移就是 1。

第 5 行字节码开头的数字 1，同样表示该字节码的偏移值，这一条字节码不同于上一条，它带有一个参数 4。字节码的操作符和参数各占 1 个字节，所以这条字节码的长度就是 2。可以推算下一条字节码的偏移值是当前字节码偏移值加当前字节码长度，也就是 1+2=3。所以第 6 行的开头数字是 3。

那么这些字节码究竟是怎么执行的呢？这就不得不提 Java 语言虚拟机了。**通常，Java 语言虚拟机包含两大核心模块：执行器和内存管理器，这里的执行器就是专门用来执行字节码的**。目前使用最广泛的虚拟机是 Hotspot，它的执行器包括了解释器和 JIT 编译器，接下来，我就以 Hotspot 为例，分别对它们加以介绍。

简单的解释器实现

通常来讲，解释器是对字节码进行解释执行的。而在 Hotspot 里，**解释器主要分为两大类：cpp 解释器和模板解释器**。其中，cpp 解释器结构最为简单，其他解释器都是在它的基础上做了不同的改进，但核心原理基本相同。因此，这里我就以 Hotspot 中的 cpp 解释器为例，带你来分析它的实现原理。

原理理解起来并不复杂，**最简单的解释器就是按照规范，模拟一个栈，然后按照字节码的语义逐个执行**。这种解释器的策略就是取一条字节码，然后按照这条字节码的语义对栈进行操作。我这里还是以上面的例子来进行说明。

每一个 Java 方法的栈里面都有一个模拟栈和一个变量表。在刚开始执行 dist 方法的时候，栈里的模拟栈是空的，变量表中的值只有 4 个入参，如下图所示：

模拟栈

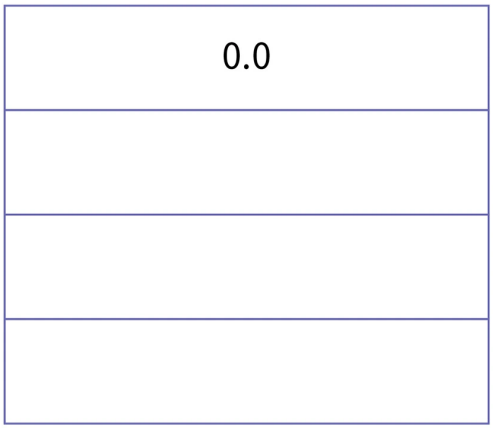
变量表

位置	值
0	0.0
2	0.0
4	1.0
6	1.0

极客时间

因为参数类型都是双精度数，所以变量表里每个参数都占据两个存储位。第一条字节码是d_load0，这条字节码的语义是将变量表位置为 0 的那个值加载到栈上，执行完这条字节码以后的栈是这样的：

模拟栈



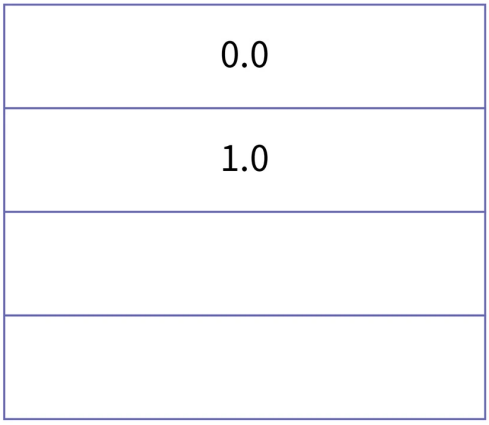
变量表

位置	值
0	0.0
2	0.0
4	1.0
6	1.0

极客时间

接下来，解释器会执行下一条字节码“dload 4"，这条字节码的语义是将位置为 4 的那个值加载到栈上，执行完以后的栈是这样的：

模拟栈

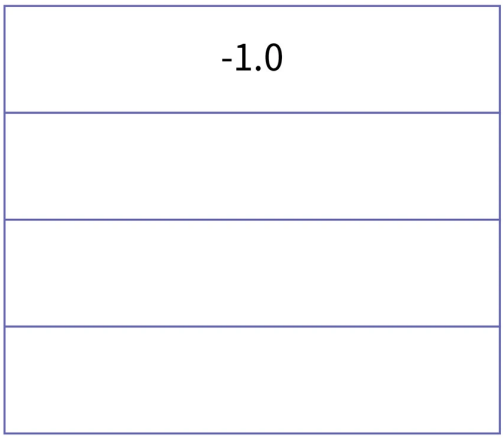


变量表

位置	值
0	0.0
2	0.0
4	1.0
6	1.0

再下一条字节码是 dsub，这条字节码的语义是将栈顶元素 1.0 取出做为减数，然后再取出新的栈顶元素 0.0 做为被减数，执行减法，得到差 -1.0，再把这个数放到栈顶，经过这步操作以后，栈里的情况是这样的：

模拟栈



变量表

位置	值
0	0.0
2	0.0
4	1.0
6	1.0

接下来是 dstore 8，这条字节码的语义是将栈顶元素取出，并存储到变量表里，位置为 8 的地方。所以，经过这一条字节码以后的栈的情况如下所示：

模拟栈

变量表

位置	值
0	0.0
2	0.0
4	1.0
6	1.0
8	-1.0

极客时间

因为分析的情况类似，所以剩余的字节码我就不再分析了，你可以自己练习一下。

讲到这里，你就能理解，**每一条字节码的语义都是由 Java 语言规范规定的，不管在什么平台上，模拟栈和变量表这两个数据结构都是相同的。本质上，字节码就是对模拟栈和变量表不断地进行操作。这种逐条取出字节码，逐条执行的方式被称为解释执行。对字节码进行解释执行的执行器叫做解释器。**

在理解了解释器以后，你肯定会想到，解释器的运行效率肯定很差吧。举个例子，对于加法操作，C++ 的加法语句会被翻译成加法指令，只需要一条就够了。但是 Java 的加法语句却要经历两次出栈操作、一次加法操作和一次入栈操作。

如果 Java 语言虚拟机只有解释器一种执行策略的话，性能肯定无法支撑 Java 获得今天的地位。实际上，Java 语言的运行效率是非常高的，这是怎么做到的呢？它的核心秘密就在于即时编译 (Just In Time, JIT)。我们接下来一起打开 JIT 编译器的神秘面纱吧。

JIT 编译器

JVM 在运行之初将 class 文件加载进内存，然后就开始解释执行。如果一个函数被执行多次，JVM 就会认为这个函数是一个热点 (hotspot) 函数，然后就将它翻译成机器码执行。在第 6 节课至第 8 节课中所讲的 C++ 静态编译相比，JIT 最大的特点是在程序运行时进行编译。

这种编译方式相对解释器，性能得到了巨大的提升。它与静态编译相比又具有怎样的特点呢？我们接下来从原理入手，抽丝剥茧来回答这个问题。我们先来了解 JIT 编译器能成功运行所依赖的两大核心机制，分别是：

1. **申请可写可执行的内存区域，确保在运行器可以生成可执行的机器码；**
2. **基于性能采样的编译优化 (Profiling Guided Optimization, PGO)，可以使得 JIT 编译器获得超过静态编译器的运行性能。**

我们先来看看 JIT 编译的第一个核心机制：可写可执行的内存区域。

可写可执行的内存区域

具体来讲，这个机制是申请一块既有写权限又有执行权限的内存，然后把你要编译的 Java 方法，翻译成机器码，写入到这块内存里。当再需要调用原来的 Java 方法时，就转向调用这块内存。你可以看下面例子：

复制代码

```
#include<stdio.h>

int inc(int a) {
    return a + 1;
}

int main() {
    printf("%d\n", inc(3));
    return 0;
}
```

上面这个例子很简单，就是把 3 加 1，然后打印出来，我们通过以下命令，查看一下它的机器码：

复制代码

```
$ gcc -o inc inc.c
$ objdump -d inc
```

然后在这一堆输出中，可以找到 inc 方法最终被翻译成了这样的机器码：

复制代码

```
1   40052d:  55                    push   %rbp
2   40052e:  48 89 e5              mov    %rsp,%rbp
3   400531:  89 7d fc              mov    %edi,-0x4(%rbp)
4   400534:  8b 45 fc              mov    -0x4(%rbp),%eax
5   400537:  83 c0 01              add    $0x1,%eax
6   40053a:  5d                    pop    %rbp
7   40053b:  c3                    retq
```

我们先来分析一个这块机器码，你可以看到，它首先会保存上一个栈帧的基址，并把当前的栈指针赋给栈基址寄存器（第 1 行），这是进入一个函数的常规操作。这个过程我们在第 4 节课有详细介绍，你可以去看一下。

然后把 edi 存到栈上（第 3 行）。在 X86 64 位 Linux 系统上，前 6 个参数都是使用寄存器传参的。第一个参数会使用 rdi，第二个参数使用 rsi，等等。所以 edi 里存的其实就是第一个参数，也就是整数 3，为什么使用 rdi 的低 32 位，也就是 edi 呢？因为我们的入参 a 是 int 型，你可以试试换成 long 型，会有什么样的效果。

接着，把上一步存到栈上的那个整数再存进 eax 中（第 4 行）。最后，把 eax 加上 1， 然后就退栈，返回（第 5 行往后）。按照二进制接口 (Application Binary Interface, ABI) 的规定，返回值通过 eax 传递。

通过上面的分析，你会发现，其实上面编译代码的第 3 行和第 4 行根本没有存在的必要，gcc 默认情况下，生成的机器码有点傻，它总要把入参放到栈上，但其实，我们是可以直接把参数从 rdi 中放入到 rax 中的。

如果你还是觉得这段代码太复杂了，那我们可以自己改一下，让它更精简一点。怎么做呢？答案就是运行时修改 inc 的逻辑，修改后的代码如下所示：

复制代码

```
1  #include<stdio.h>
2  #include<memory.h>
3  #include<sys/mman.h>
4
5  typedef int (* inc_func)(int a);
6
7  int main() {
8      char code[] = {
9          0x55,             // push rbp
```

```
        0x48, 0x89, 0xe5, // mov rsp, rbp
        0x89, 0xf8,       // mov edi, eax
        0x83, 0xc0, 0x01, // add $1, eax
        0x5d,             // pop rbp
        0xc3              // ret
    };

    void * temp = mmap(NULL, sizeof(code), PROT_WRITE | PROT_EXEC,
            MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);

    memcpy(temp, code, sizeof(code));
    inc_func p_inc = (inc_func)temp;
    printf("%d\n", p_inc(7));

    return 0;
}
```

在这个例子中，我们使用了 mmap 来申请了一块有写权限和执行权限的内存，然后把我们手写的机器码拷进去，然后使用一个函数指针指向这块内存，并且调用它。通过这种方式我们就可以执行这一段手写的机器码了。我们来运行一下看看：

复制代码

```
$ gcc -o inc inc.c
$ ./inc
8
```

为了生成更精简的机器码，我们可以引入编译器优化手段，例如全局值编码、死代码消除、标量展开、公共子表达式消除和常量传播等等。这样生成出来的机器码会更加优化。所以这种编译方式就被称为即时编译。

我们搞清楚了 JIT 编译的第一个核心机制后，再来看它的第二个核心机制：基于采样的编译优化。

基于采样的编译优化和退优化

我们知道，架构师的一个核心职责是对比各种技术方案的优劣，我至今还能听到有的架构师说 Java 性能不好，理由有很多，比如 Java 是解释执行、Java 里所有函数都是虚函数、每次调用都需要查询虚表等等。

这些说法在老版本的 JVM 中可能是对的，但随着 JVM 中的 JIT 编译器的演进和优化，上面所讲的 Java 语言的性能缺陷都在逐渐被克服。

在第 6 节课我们已经学习过了编译器的作用就是把高级语言翻译成性能最好的机器码。不管是静态编译还是 JIT 编译，它们的功能都是一样的，但是 JIT 编译往往可以做得更好。我们通过一个编译优化的常量传播例子来说明。

这个例子的代码如下：

复制代码

```
1 public static int test() {
2     int b = 3;
3     int c = 4;
4     return b + c;
5 }
```

在这块代码中，由于第 2 行对 b 赋值一个常量后，后面的语句没有再改过 b 的值，我们就可以把后面所有出现 b 的地方都改为 3，同理所有出现 c 的地方都改为 4。经过这种优化，代码的第 4 行就可以改写成：

复制代码

```
1   return 3 + 4;
```

接着，编译器再做一轮分析，将运算符两边都是常量的情况，直接进行计算，也就是把上面的代码再优化成：

复制代码

```
1   return 7;
```

这种优化被称为**常量折叠**。（这里也请你思考一下，如果这行代码是 C 语言的，在最优化的情况下，gcc 会生成什么样的机器码？）

通过上面的例子，我们讲了编译器优化的一个简单思路。其实，编译器的优化还有很多其他的思路和方法，这里我就不再一一列举了，如果你对编译器设计特别感兴趣的话，我推

荐你去看看《编译原理 》和《高级编译器设计与实现 》等书。

接下来，我们再看第二个例子，这是一个 C 语言编译器没有办法优化，但是 JIT 编译却能进一步优化的例子。在这个例子中，你会发现常量传播不再起作用了：

复制代码

```
1  public static int test(boolean flag) {
2      int b = 0;
3      if (flag) {
4          b = 3;
5      }
6      else {
7          b = 2;
8      }
9      return b + 4;
10 }
```

和第一个例子相比较，这个例子增加了第 3 行到第 8 行的条件判断。所以编译器无法知道在第 9 行 b 的真实取值是什么。只能严格按照这个函数的逻辑去生成比较，跳转，赋值等等，那么这个例子就比可以常量折叠那个例子复杂多了。

一般情况下，虽然这个例子中的 test 函数没有优化空间了，但是 JVM 的 JIT 技术还是在这里找到了最优化的机会。假如存在一种情况，每一次 test 方法被调用的时候，传的参数 flag 都是 true 或者都是 false，也就是说，flag 的取值固定。那么 JIT 编译器就可以认为另外一个分支是不存在的，可以不编译。

JIT 编译器在开始之前，test 方法是由解释器执行的。解释器一边执行，一边会统计 flag 的取值，这种统计就叫做性能采样（Profiling)。当 JIT 编译器发现，test 方法被调用了 500 次（这个阈值可以以 JVM 参数指定），每一次 flag 的值都是 true，那它就可以合理地猜测，下一次可能还是 true，它就会把 test 方法优化成这个样子：

复制代码

```
1  public static int test(boolean flag) {
2      return 7;
3  }
```

但是，这种做法，相信你也都看出问题来了，如果恰好 test 方法的下一次调用就是 false 呢？所以 JVM 必须在 test 方法里留一个哨兵，当参数 flag 的值为 false 的时候，可以再退回到解释器执行。这个过程就是**退优化（Deoptimization）**。这个过程相当于，JVM 的 JIT 编译器生成的机器码等效于以下代码：

复制代码

```
public static int test(boolean flag) {
    if (!flag)
        deoptimize()
    return 7;
}
```

在这个代码中，deoptimize 方法是 JVM 提供的内建方法，它的作用是由 JIT 编译器退回到解释器进行执行。这个过程涉及栈帧（第 5 节课的核心概念）的运行时切换，无疑是非常精巧和复杂的，但我们做为 Java 语言的使用者，并不需要完全理解 JIT 背后的每一个技术细节。但通过这个例子，我们可以掌握了如何写程序，才能让 JIT 编译器帮我们生成最高效的机器码。

让 JIT 编译器运行得好，我们只需要遵守一条原则：**让程序行为可预测**。因为 JIT 编译优化的基本假设是过去和未来，程序的运行规律基本一致，所以它基于过去的行为测试未来。如果它预测的未来和真实情况不一致，就会发生退优化。退优化的情况会对性能带来巨大的伤害，所以 JIT 有时也可能是一把双刃剑。

到此为止，我们就把 JIT 编译器的基本原理介绍完了。**可写可执行内存和基于采样的编译优化这两大机制保障了 JIT 编译器的实现，而 JIT 编译器又是 JVM 高效的核心秘密**。如果你还想了解更多关于 JIT 的知识，你可以思考一下，JIT 还有哪些优化方式呢？这里我就不再展开了，你可以查阅相关资料来学习，欢迎你在留言区与我交流。

总结

在今天这节课里，我们学习了 Java 字节码的基本原理，了解到 Java 字节码是一种基于栈的中间格式。机器码是由 CPU 的设计风格和指令集决定的，所以在不同的架构上，机器码都是不同的，但是 Java 字节码则与机器码不同，它在所有的平台和操作系统上，都是一样的，这就屏蔽掉了平台差异。

字节码文件是由 Java 语言虚拟机加载执行的。只要遵守 Java 语言规范，任何人都可以实现自己的 Java 语言虚拟机，例如 IBM 的 J9，开源的 Hotspot 虚拟机等等。虚拟机有自动内存管理模块和执行器两大核心组成。自动内存管理是我们这个专栏的核心内容，我们会在后面的内容中加以介绍。

虚拟机中用于执行字节码的模块是执行器，最简单的执行器是解释器，但是解释器的性能比较差。为了提高性能，JVM 中引入了 JIT 编译器。JIT 编译器依赖两个核心机制：

1. **申请可写可执行的内存区域，以便于运行时生成机器码；**
2. **基于性能采样的编译优化 (Profiling Guided Optimization, PGO)，可以使得 JIT 编译器获得最佳性能。**

当 JIT 编译器在做性能优化的时候，我们根据统计做了很多假设，比如某个分支语句，一直只选择其中一个分支，另一个分支就不会走到；再比如某个对象的类型永远都是父类，而不是子类等等。

这种根据过去的规律去预测未来的做法，并不能保证完全正确，所以 JIT 编译器往往又会配合退优化机制一起工作。当预测正确时，此时的性能是最好的，但是如果预测不正确，就可以退回到解释器进行执行，从 JIT 编译器退回解释器的过程就是退优化。

思考题

我们在第 3 节课讲过了进程的内存布局，但当时是以 C/C++ 的进程来举例的。考虑以下 Java 程序：

复制代码

```
1   public static int test() {
2         Random r = new Random();
3         int a = r.nextInt();
4         return a + 1;
5     }
```

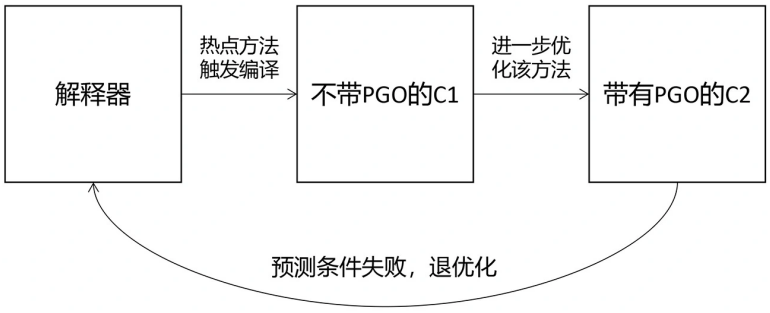
请你分析，变量 r 和 a 分别创建在进程的什么位置？欢迎你在留言区分享你的想法和收获，我在留言区等你。

吊打面试官

- 请讲一下hotspot中的执行器?

hotspot中的执行器是用于执行字节码的，Java字节码是基于栈设计的，它的解释器就是对栈操作的模拟，实现简单，但性能比较差。

为了解决性能问题，hotspot引入了两个即时编译器，分别是C1和C2，在有些文档中，C1也被称为client compiler，C2被称为server compiler。C1编译速度快，但优化效果稍差一些，C2的优化效果好，但编译速度慢一些。JIT编译器都是基于Profiling的，它只会把调用最频繁的方法进行编译。如下图所示：



高频面试真题

好啦，这节课到这就结束啦。欢迎你把这节课分享给更多对计算机内存感兴趣的朋友。我是海纳，我们下节课再见！

上一篇 10 | 页中断：fork、mmap背后的保护神

下一篇 12 | 内存虚拟化：云原生时代的奠基者

精选留言 (3)

写留言

送过快递的码农
2021-11-18

终于到Java了 ，发现还是有许多看不懂。我猜应该还是在堆里面吧。虽然从Java的角度来说，临时变量是应该在栈里面。但是我猜想对于C进程来说，在解释之前，进程是不知道的啊，只有执行的时候，才知道你是在Java栈还是Java堆里面，而且老师也说过，Java是模拟的栈还有变量表，那模拟栈应该是对应C的结构体吧，应该是放堆存着先？

展开

kylin
2021-11-17

a是局部变量，存放在虚拟机的模拟栈上，但JVM会将模拟栈创建在进程虚拟内存哪里呢？猜测是栈
r是Java对象，分配到Java堆中，JVM应该提前使用mmap创建一大块内存，应该是在内存映射区吧

一子三木
2021-11-17

r是在堆上，a在虚拟机栈上？

展开

共 1 条评论 >