第 3 章 执行引擎（上）

不少人在初次接触计算机系统原理时，可能会因为各种各样的寄存器用途和加减指令而感到乏味，很难将这些晦涩枯燥的概念和显示器中色彩缤纷的画面联系起来。其实现实世界和计算机系统有几分相似，微观粒子的种类是有限的，而世界万物都是由这些类型有限的微观粒子排列组合而成。如今我们正在建造的 Java 虚拟机，就像一台可以运行万物的机器，这台机器只需要 200 多条指令就可以实现一切功能，而指令的作用，仅仅是改变某一个可寻址空间的比特序列。

早在上个世纪 40 年代，冯·诺依曼就给现代计算机划定了一个流水线结构，直到现在依然没有对这个模型做出本质改变。Java 虚拟机的执行引擎只不过使用了软件方法代替了硬件 CPU，取指令和执行指令由集成电路的电信号传递变成了高级编程语言的代码逻辑。

执行引擎也不完全只是一个软件 CPU 抽象。在没有操作系统的时代，程序直接面向硬件运行，多个用户只能在物理空间中排队依次用纸带打孔机输入代码。在计算机硬件管理器的逐步完善过程中，更为人熟知的操作系统诞生了，并抽象出了如进程、文件等概念支持多用户程序执行。Java 虚拟机并不像操作系统一样复杂，但麻雀虽小，五脏俱全，虚拟机内部的执行引擎同样需要提供线程调度功能。

本章的主要目标是实现虚拟机执行引擎部分功能，在没有 JIT 优化时，指令是解释执行的，所以执行引擎也叫解释器。尽管解释执行的性能较低，但由于其简单易懂的特点，广泛运用于各种语言的虚拟机中。

3.1

Java 虚拟机（除 Android 外）是一种栈虚拟机，这种类型的虚拟机通常只有一个 PC 寄存器，而且许多指令都没有操作数。有汇编语言经验的程序员可能不会忘记被各种专用寄存器和寻址方式支配的恐惧，要想阅读一段汇编代码，就不可避免地需要熟悉目标硬件体系中各个寄存器的作用和寻址模式。

在 Java 虚拟机中，复杂的寄存器和寻址模式都得到了极大的简化，计算被抽象成了统一而简单的过程：通过唯一的 PC 寄存器寻找方法区的指令，操作数总是通过操作数栈（Operand Stacks）传递，同时栈帧中还包含一个局部变量表（Local Variables）来存放方法调用参数和临时变量。

由于 Java 语言“一切皆对象”的指导原则，JVM 中的寻址其实就约等于寻找对象。对象在栈中总是以引用的形式存在（不考虑逃逸分析的情况），引用的值实际上就是对象的地址。

不考虑调度功能的执行引擎通常都拥有类似下列伪代码的结构：

```
while code has next:
    match code[pc]:
        execute(code[pc][,code[pc..]])
        pc = pc + n
```

对于 Java 虚拟机而言，除了基础的取指令执行指令循环之外，还需要实现诸如类加载、异常处理、线程间通信等一系列功能，整个执行引擎的讨论会分为 2 章完成，其结构会逐渐完善。

下面用白纸画图的方式来模拟 Java 虚拟机执行一段简单的 Java 字节码，通过这个过程，我们可以对执行引擎的基本流程建立一个感性认识，后续转化为代码逻辑就容易

多了。

首先准备一段简单的 Java 程序并编译和反编译，如代码清单 3-1 所示。

代码清单 3-1 四则运算的字节码

```
public class Simple {
    public static void test(int a, long b, float c, double d, boolean f) {
        int aa = a + 1;
        long bb = b - 1;
        float cc = c * 1.0f;
        if (f) {
            double dd = d / 1.0d;
        }
    }
}
```

使用 javap 查看字节码，如下所示：

```
$javap -v Simple
  MD5 checksum 60e0aae1bb059127cb558068da0ce387
  Compiled from "Simple.java"
public class Simple
  // 省略常量池
{
  public Simple();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method
java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 1: 0

  public static void test(int, long, float, double, boolean);
    descriptor: (IJFDZ)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=4, locals=13, args_size=5
         0: iload_0
         1: iconst_1
         2: iadd
         3: istore       7
         5: lload_1
         6: lconst_1
         7: lsub
```

```
          8: lstore       8
         10: fload_3
         11: fconst_1
         12: fmul
         13: fstore       10
         15: iload        6
         17: ifeq         26
         20: dload        4
         22: dconst_1
         23: ddiv
         24: dstore       11
         26: return
       LineNumberTable:
         line 4: 0
         line 5: 5
         line 6: 10
         line 7: 15
         line 8: 20
         line 10: 26
       StackMapTable: number_of_entries = 1
         frame_type = 254 /* append */
           offset_delta = 26
           locals = [ int, long, float ]
}
SourceFile: "Simple.java"
```

字节码结构已经很熟悉了，这里重点关注 test 方法部分。与 2.2 节解析的结果一样，test 方法的描述符是(IJFDZ)V，拥有 5 个参数和 void 返回类型，访问修饰符 flags 为 ACC_PUBLIC|ACC_STATIC。Code 属性表 stack=4，locals=13，它指示虚拟机创建操作数栈容量为 4、局部变量表容量为 13 的栈帧来执行本方法。

接下来，方法指令由一行一行类似汇编的语句构成。冒号左边的数字代表指令偏移地址，也就是 PC 寄存器存储的值；冒号右边是指令和操作数，每条指令有 0 个或多个参数，具体取决于指令逻辑。

紧接着的 LineNumberTable 是 Code 属性表的一个属性表，因为它并不是必须的，所以在 2.2.5 节没有刻意讨论。它的作用在于，当程序抛出异常时，虚拟机可以在栈帧信息中显示异常对应的代码行数，该信息由编译器添加，可以在编译时通过参数-g:none 禁用。

最后是 StackMapTable 属性表，同样在第 2 章中被忽略了，它的主要作用是用于类加载器快速验证字节码，感兴趣的你可以参考 JVM 规范。

现在尝试在白纸上运行代码，首先在白纸上画出包含一个栈帧的栈，栈帧的局部变量表长度为 13，操作数栈长度为 4。局部变量表可以通过索引访问，操作数栈只能使用后入先出的顺序访问。如图 3-1 所示。

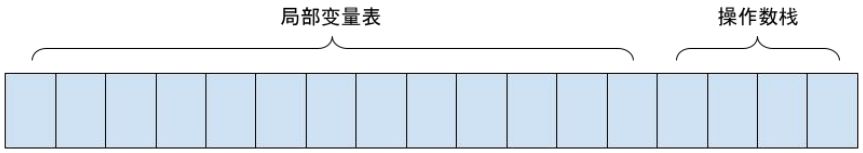

图 3-1 初始化栈帧

Code 属性表第 1 条指令 iload_0，在 JVM 规范中查找其含义，如图 3-2 所示。

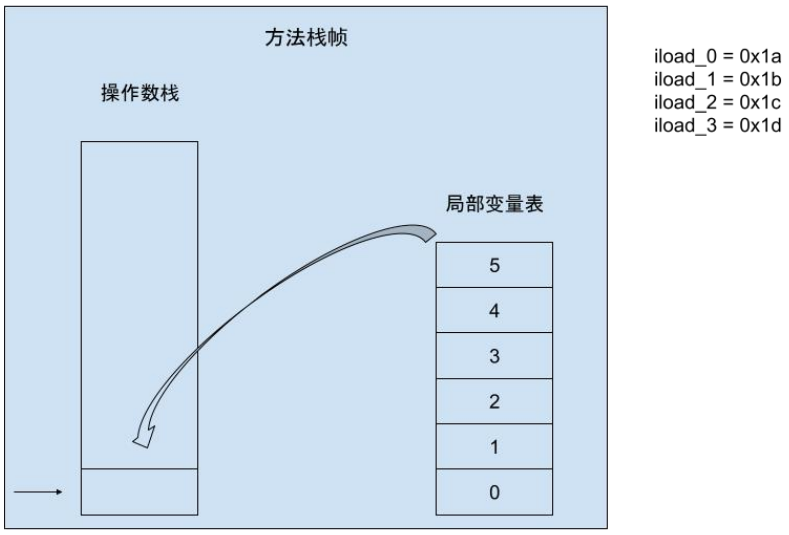

图 3-2 iload_n 指令

iload_<n>是一系列逻辑相同的指令，0x1a~0x1d 分别代表从局部变量表将第 0~3 个元素推入栈顶，这个元素必须是整数类型。这里要执行的是 iload_0，那么局部变量表的第 0 个元素是什么呢？虽然还没有开始讨论方法调用，但是指令总是包裹在方法中，为

了避免过多的上下文，这里无伤大雅地先说结论：局部变量表的前 n 个槽位在进入方法时就装载了所有参数，n 的值取决于所有参数的存储长度，通过解析方法描述符获得。

test 方法的 locals[0]存放第 0 个参数 int，locals[1]是第 1 个参数 long，locals[3]是第 2 个参数 float，以此类推。注意槽位的定义，由于 long 和 double 类型占据 2 个 slot，因此 locals[2]不是一个合法的索引，而是 locals[1]的一部分。

假设调用 test 方法的语句为 Simple.test(1,2L,3.0f,4.0d,true)，在开始执行第 1 条指令前，执行引擎就已经将所有参数装载，更新局部变量表的状态，如图 3-3 所示。

图 3-3 初始化局部变量表

现在再来执行第 0 条指令 iload_0。将局部变量表 locals[0]复制到操作数栈顶，复制之后，操作数栈指针应该向右移动。接下来的 10 条指令分别是：lload、lstore、fload、fstore、dload、dstore、iadd、lsub、fmul 和 ddiv。Java 虚拟机指令的命名通常有规律可寻，结合编译之前的 Java 源代码可推测，指令第一个字母表示操作的数据类型，后面的单词表示具体逻辑。load 和 store 分别代表从局部变量表加载数据到操作数栈顶以及从操作数栈顶弹出数据保存到局部变量表，add、sub、mul 和 div 分别表示数学加减乘除。

你可能注意到对于 char 和 boolean 类型的操作指令也是以 i 开头，事实上，由于 Java 虚拟机的指令只有 1 字节定长，所以，为了尽可能节约指令编码空间，short、boolean、char 等数据类型都使用 int 类型的操作指令而不作区分。关于此类简单指令的描述细节，

你可参考 JVM 规范中的指令集部分，此处不再占据篇幅一一讨论。

对照指令表，PC=0~3 的栈帧变化如图 3-4 所示。

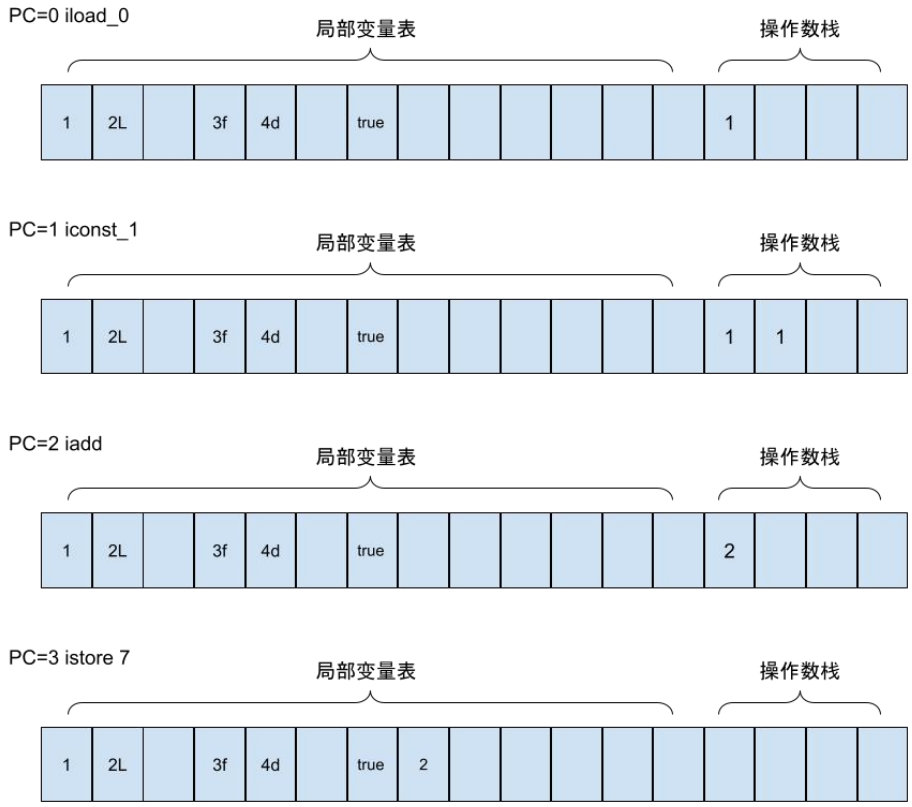

图 3-4 指令 0~3

如图 3-4 所示，编译器准确地计算出栈帧中每个位置的变化，指示应该将数字 2 存储到 locals[7]而不是覆盖前面的变量表，对应到 Java 代码中，就是第 1 行代码。

后续过程一直到指令 17 都大同小异，你可对照指令逻辑自行完成。指令 17 对应的 ifeq 稍微复杂一点，它是 if-equals 的简写，指令逻辑为：将操作数栈弹出一个 int 并与 0 比较，当 int 等于 0 时，则将 PC 置为 ifeq 后面的 2 字节所构成的地址。指令 15~17 连起来的过程为：将 locals[6]推入栈顶，判断当前栈顶是否为 0，如果是，则跳转到 26，否则继续执行。指令 26 正好是 return 指令，指示从当前方法返回，与 test 方法的逻辑

完美对应。值得注意的是，指令 15 将布尔值变量推入栈顶作为 ifeq 指令操作数，Java 虚拟机也遵循 C 语言的传统：true=1, false=0, 所以参数为 true 时不执行跳转。指令 17~26 的栈帧变化如图 3-5 和图 3-6 所示。

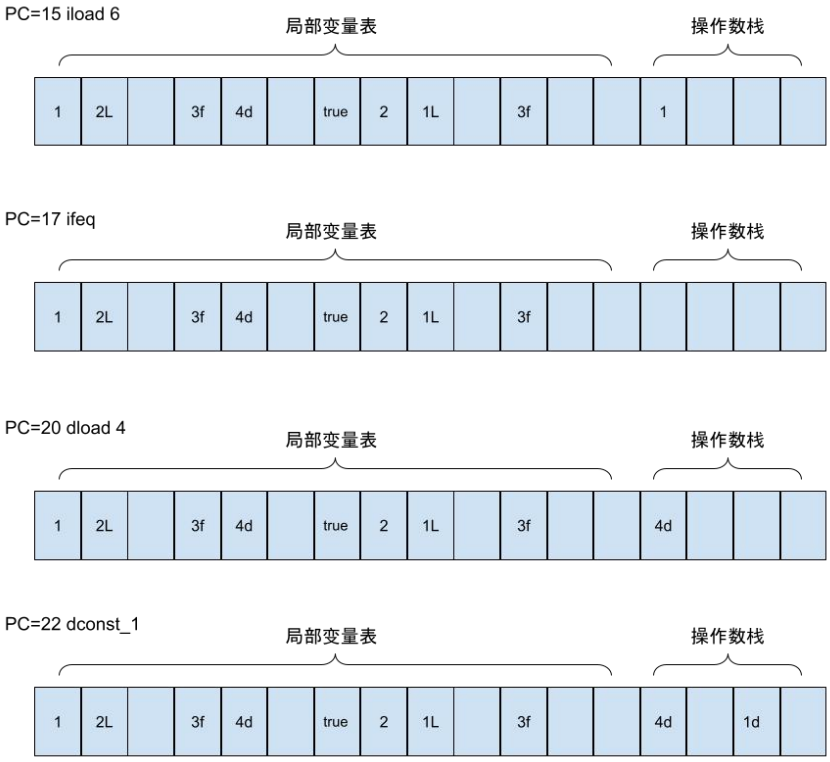

图 3-5 指令 17~22

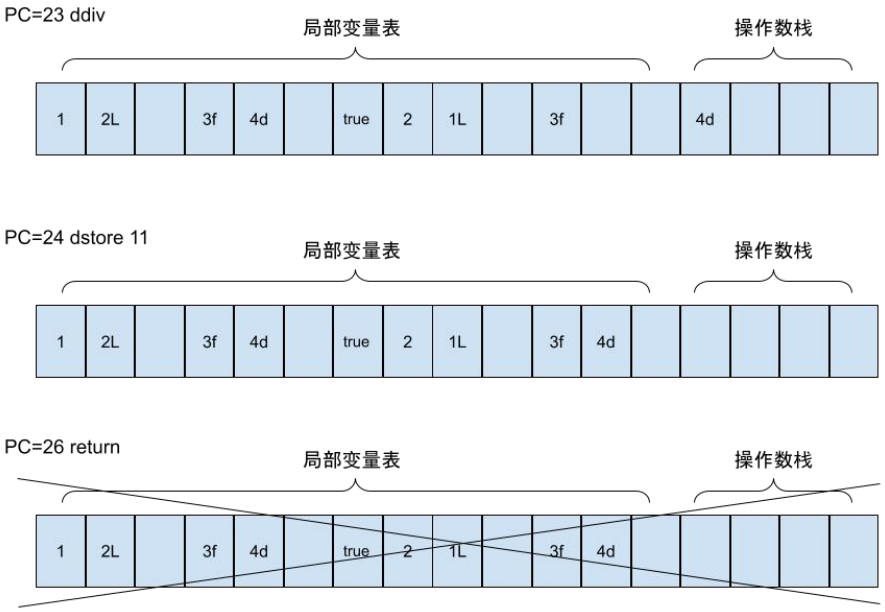

图 3-6 指令 23~26

最后一条指令 return 结束当前方法，栈帧资源释放，且没有返回值。

经过逐条指令的解释执行，我们成功地模拟 Java 虚拟机运行了一段代码。计算机程序的本质是简单的指令组合，而这些指令所做的工作仅仅是在正确的可寻址位置正确地改变比特序列，虽然目前只涉及一些简单指令，但仍不失为一个理解栈虚拟机模型的良好开端。本章后续小节讨论如何将这一过程转化为 Rust 代码，着重介绍整体结构而避免每条指令的细节，最终目标是真正地启动 Java 虚拟机并运行一段简单的 Java 程序。

3.2

前一节忽略了很多教条式的知识，现在给出 Java 虚拟机关于栈和栈帧比较明确的定义：栈是线程的私有内存空间，由栈帧组成，每个栈帧对应一次方法调用所需的空间，在方法调用开始时，新的栈帧被推入栈，结束时栈帧弹出。栈帧内部则由局部变量表、操作数栈、当前类指针和当前方法指针组成，用于存储运行时方法数据。

如果一开始就从以上定义出发，未免显得太过抽象，幸好已经经历了白纸执行字节的过程，现在可以反过来思考栈帧结构是如何设计的。首先，栈已经是程序员非常熟悉的一种数据结构了，无论在何种编程语言实现中，栈总是被用于实现方法或函数调用，因为应该被“遗忘”的数据或不应该被“遗忘”的数据，栈都能通过移动指针轻易处理。

Java 虚拟机中的栈帧存储了当前类和当前方法指针，这个容易理解，因为方法执行必然需要上下文，尤其需要指令序列，执行引擎在执行方法时只需从顶层栈帧的方法指针读取指令即可。

局部变量表和操作数栈则充当了临时数据存储的角色，其中，局部变量表可以随机访问，在新栈帧初始化时，执行引擎就方法参数存入其中，而操作数栈只能以先进后出的顺序访问，初始化时是一片空白，执行引擎只需要忠实地按照指令改变二者的数据即可，在之前已经生动地体会过这一过程，此处不再赘述。

值得赞叹的是，这一切正确执行的基础都在于 Java 编译器准确无误地将代码编译为了字节码，就像未卜先知一样知道栈帧中的空间何时可以被覆盖，何时需要被暂存。不过 Java 编译器没有进一步优化字节码，而是将大量的优化任务交给了虚拟机。代码清单 3-1 就是典型的例子，方法中出现的局部变量在无后续访问地情况下，依然会严格出现在编译后的字节码中，哪怕确定不会再被访问，依然会有指令指示将该值存储到即将被释放的局部变量表中，如果按部就班地执行，只是做无用功。

现代编译理论早已可以优化处理大量类似的情形，例如，集成开发环境就可以使用控制流分析或数据流分析识别出“多余”的代码，Java 作为一种高性能语言，选择将此类优化放在虚拟机层面，自有兼容性的考量，本书虽不涉及 JIT 优化等高级课题，但有必要讨论实现中的可提升空间。

回到正题中，根据以上讨论，方法栈的大致结构已经清楚，初步尝试定义栈的数据结构，如代码清单 3-4 所示。

代码清单 3-4 src/mem/stack.rs (不好的定义)

```
pub struct JavaFrame {
    locals: Vec<u8>,
    operands: Vec<u8>,
    klass: Arc<Klass>,
    method: Arc<Method>,
}
pub struct JavaStack {
    frames: Vec<JavaFrame>,
}
```

代码清单 3-4 看上去是一种自然的定义，然而常见的 Java 虚拟机并不采用这样的方式，因为操作数栈和局部变量表被定义为单独的字节序列对象，在内存空间中不连续，有违栈的概念。从字面含义上理解，栈帧应该是数据的区间划分，而不是实际的数据存储单元。尽管 JVM 规范并没有明确要求方法栈的实现必须在内存中连续存储，但为了更高效和方便的使用，还是将栈的定义做一些修改，将操作数栈和局部变量表放在统一的字节数组中，如代码清单 3-5 所示。

代码清单 3-5 src/mem/stack.rs(更合理的定义)

```
pub struct JavaStack {
    data: Vec<u8>,
    frames: Vec<JavaFrame>,
    max_stack_size: usize,
}
pub struct JavaFrame {
    locals: *mut u8,
    operands: *mut u8,
    class: *const Class,
    method: *const Method,
    pc: usize,
    max_locals: usize,
}
```

方法栈定义的另外一项修改是将当前类和当前方法定义为裸指针，这在 Rust 中并不常见。通常情况下，更鼓励使用引用或智能指针代替裸指针，因为 Rust 编译器可以

保证在使用引用或智能指针时，不会出现垂悬指针（Dangling Pointer），参见第 1 章的相关内容。但考虑到栈帧的使用场景中，方法区才是类和方法的所有者，方法区的生命周期与整个 Java 虚拟机的生命周期相同，栈帧必然不晚于方法区结束自己的生命周期，所以将当前类和当前方法的指针定义为裸指针也不会出现问题。唯一需要注意的是，类和方法在方法区中均以引用计数器的形式存在，从引用计数器转换为裸指针是一个尚处于不稳定特性的阶段，只在 nightly 版本的 Rust 编译器中才允许，且需要添加额外的编译标记。

有了栈帧的定义，可以方便地为 JavaStack 添加一些必要的操作方法，初始化栈时即向操作系统申请固定大小的内存空间（可以实现为虚拟机的启动参数），大多数操作方法都只需移动指针即可。因为该部分代码量较大，只列举了一些操作方法作为参考，如代码清单 3-6 所示。

代码清单 3-6 src/mem/stack.rs

```
    impl JavaStack {
        // 获取最顶层栈帧
        pub fn mut_frame(&mut self) -> &mut JavaFrame {
            self.frames.last_mut().expect("empty_stack")
        }
        // 获取当前栈帧的操作数栈
        pub fn operands(&self) -> *mut u8 {
            self.frame().operands
        }
        // 根据 PC 获取当前指令
        pub fn code_at(&self, pc: usize) -> u8 {
            self.method().get_code().unwrap().2[pc]
        }
        // 对应 load 系列指令
        pub fn load(&mut self, offset: usize, count: usize) {
            unsafe {
                self.operands()
                    .copy_from(self.locals().add(offset * PTR_SIZE), count *
PTR_SIZE);
                self.update(self.operands().add(count * PTR_SIZE));
            }
        }
        // 对应 store 系列指令
```

```
    pub fn store(&mut self, offset: usize, count: usize) {
        unsafe {
            self.update(self.operands().sub(count * PTR_SIZE));
            self.locals()
                .add(offset * PTR_SIZE)
                .copy_from(self.operands(), count * PTR_SIZE);
        }
    }
}
```

准备工作完成，现在可以正式开始执行引擎的建造了。这部分代码存放于 interperter 模块，它的核心是一个巨大的 loop-match 结构，就像一条流水线不停的根据 PC 的值取当前方法中的指令执行。考虑到书中贴大段 loop-match 代码影响阅读，这里使用 Rust 和伪代码结合的方式演示执行引擎的核心结构，每个匹配代表一组指令，如代码清单 3-7 所示。

代码清单 3-7 src/interpreter/mod.rs

```
pub fn execute(stack: &mut JavaStack) {
    let mut pc: usize = 0;
    while stack.has_next(pc) {
        let instruction = stack.code_at(pc);
        match instruction {
            // nop
            0x00 => {
                pc = pc + 1;
            }
            // aconst_null
            0x01 => {
                stack.push(&NULL);
                pc = pc + 1;
            }
            …
        }
    }
}
```

执行引擎只有一个核心函数 execute，参数就是当前线程栈，可以理解为每一个在虚拟机上运行的 Java 线程背后都在执行这段逻辑。这是对前述解释器普适模型伪代码的 Rust 实现，每个分支都会根据指令逻辑改变某个地址的状态，同时 PC 按照参数长度递增指向下一条指令。配合 JavaStack 提供的辅助方法，尝试实现前一小节中介绍的一些

指令，如代码清单 3-8 所示。

代码清单 3-8 src/interperter/mod.rs

```
match instruction {
    ...
    // iload/fload
    0x15 | 0x17 => {
        let opr = stack.code_at(pc + 1) as usize;
        stack.load(opr, 1);
        pc = pc + 2;
    }
    ...
}
```

在 loop-match 中，指令分支应尽可能地合并，除了减少重复代码外，还能提高运行时性能。Rust 借鉴了许多函数式语言的特性，提供了强大的模式匹配功能，条件分支可以使用逻辑运算符和范围表达式，结合 JavaStack 结构体实现的公共方法，可以大幅减少指令分支的数量。代码清单 3-8 中使用范围表达式合并了一些简单传输指令分支，因为无论局部变量表还是操作数栈，都没有限制存储空间的类型，只要数据类型的大小相同，不同数据类型的同类操作就可以合并为一个分支，如 iload 和 fload 指令。

容易忽略的是 PC 值的更新，代码清单 3-7 将 PC 实现为 execute 方法的局部变量（后续将会修改），每一轮循环只会根据 PC 取当前方法的指令偏移地址，所以每条指令都需要在执行完毕时根据自身参数数量更新 PC 值以正确地指向下一条指令，xload_n 没有参数，则 PC 值只需要自增 1。当然也存在指令逻辑本身就是修改 PC 值的情况，如已经遇到过的 ifeq，以及下一节将要讨论的方法调用指令。

由于虚拟机指令繁多，本书的重点是介绍虚拟机的整体实现过程，避免陷入规范细节的汪洋大海，因此在开发过程中只例举一些关键指令的实现，具体指令的实现请参考 JVM 指令集。

3.3

在 2.3 节中，因为需要设计 Klass 的结构，我们曾经简要介绍过 Java 虚拟机中的方法调用，静态方法是其中最简单的一种，无需涉及对象实例和运行时虚函数表查询，仅需要编译期确定的方法指针即可完成。本节将详细讨论静态方法的调用过程和栈帧变化，掌握这个过程之后，可以轻易扩展到其他方法调用指令。

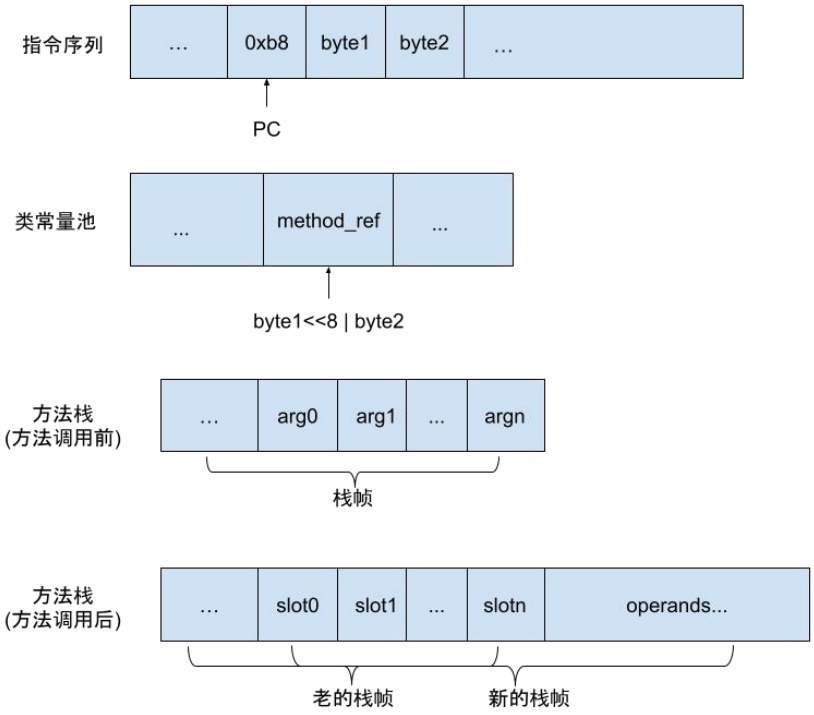

图 3-7 静态方法调用过程

图 3-7 描述了静态方法的调用过程以及调用前后栈帧的变化，实际上，所有的方法调用指令都具有类似的过程，不同之处仅在于从方法区获取方法指针的步骤，方法调用共分为如下 4 个步骤。

- 确定将要调用的方法指针。
- 为新方法分配栈帧空间，并将方法参数存入新栈帧。
- 从新方法的第 0 条指令开始执行。

- 方法结束时恢复上下文。

在第 1 步中，当执行引擎遇到 invokestatic 指令时（code[pc]=0xb8），将紧接着的 2 个字节 code[pc+1]和 code[pc+2]作为指令参数，以 byte1<<8|byte2 的值作为指向当前类中常量池的索引，该索引指向一个方法句柄，包含类名、方法名和方法描述符三元组。获取到方法句柄之后，通过 2.3 节方法区提供的方法查找类和方法，获取相应指针。例如，调用 Thread.sleep 方法时，操作数指向当前类常量池中的方法句柄为：

```
[java/lang/Thread, sleep, ()V]
```

在第 2 步中，方法的 Code 属性表包含局部变量表和操作数栈的容量信息（2.2.5 节解析 Code 属性表得到的 max_locals 和 max_stacks），使用该值为新栈帧分配空间。在白纸模拟执行程序时曾提到，开始执行第 0 条指令之前，局部变量表就已经装载好了所有参数，这正是 invokestatic 和其他方法调用指令的工作。

如果即将调用的方法有参数，当前栈帧的方法指令会包含若干 load 指令，将指定数据推入栈顶作为参数，执行 invokestatic 时，根据方法描述符，将所有需要的参数从原栈帧的操作数栈顶弹出，在此基础上建立新的栈帧，将参数存入新栈帧的局部变量表，并设置类和方法指针的指向。

在第 3 步中，保存当前 PC 的值，然后将 PC 置为 0，结束 invokestatic 的逻辑。当下一轮指令循环开始时，解释器就会从新方法的第 0 条指令开始执行。

在第 4 步中，虽然这不是方法调用指令的逻辑，但属于整个方法调用过程的一步。当遇到方法结束指令时，比如 return 和 athrow，释放当前栈帧。如果方法有返回值或者有异常抛出，将返回值或异常实例推入原栈帧的操作数栈顶，并还原 PC。

Hotspot 虚拟机关于方法调用有一些精巧的设计细节值得品味。回到第 1 个步骤，

在 invokestatic 指令之前，代码段中还有若干条 load 指令（数量取决于将要调用的方法参数数量）用于方法参数入栈，这里的入栈顺序是从左至右，例如方法：

```
public static void test(int a, long b);
```

入栈顺序是先将参数 a 入操作数栈，再将参数 b 入操作数栈。开始执行新方法逻辑之前，这些参数同样需要以从 0 到 n 的顺序放入新栈帧的局部变量表中，栈是一个先进后出的数据结构，从左至右的入栈顺序似乎增加了额外的麻烦。

这是因为，在 Hotspot 虚拟机实现中，栈帧划分与逻辑上的概念有些区别，两个栈帧之间在空间上并非毫无关系。仔细观察图 3-6 会发现，执行 invokestatic 指令前后，原栈帧的顶部和新栈帧的底部是重叠的，原栈帧的操作数栈最后几个存储单元刚好就是新栈帧的局部变量表最前面的几个存储单元。Hotspot 虚拟机在调用方法时，实际上并没有发生数据拷贝，即参数并没有出栈动作，而是简单地移动指针，将原栈帧中操作数栈的一部分视为新栈帧局部变量表的一部分。

这并不是一种新颖的设计，在栈虚拟机中早已广泛使用，而基于寄存器的模型中参数传递过程并没有类似的连续内存空间，所以通常会将参数传递设计为从右至左的顺序。

对照 3.2 节中的栈帧结构设计，代码清单 3-9 初步实现了方法调用的栈帧变化过程，除了 invokestatic 指令之外，同样适用于其他 3 条 invoke 指令，主要区别来自于方法指针和类指针的定位（第 5 章将会讨论动态派发和静态派发）。随着后续讨论的深入，代码还会不断完善，逐步加入线程上下文、本地方法调用、类加载中断和栈帧活动引用收集等功能。

代码清单 3-9 src/mem/stack.rs

```
impl JavaStack {
```

```
    pub fn invoke(
        &mut self,
        class: *const Class,
        method: *const Method,
        pc: usize,
        locals: usize,
    ) -> usize {
        let m = unsafe { &*method };
        let (_, desc, af) = m.get_name_and_descriptor();
        let (params, slots, ret) = bytecode::resolve_method(desc, af);
        // 本地方法暂未实现，直接返回 pc，对执行引擎而言就像已经执行完毕
        if m.is_native() {
            let ret = ret.size();
            self.downward(slots - ret);
            return pc;
        }
        // 移动指针
        let locals = unsafe {
            if self.is_empty() {
                self.data.as_mut_ptr().add(locals * PTR_SIZE)
            } else {
                self.operands().sub(locals * PTR_SIZE)
            }
        };
        let method_ref = unsafe { &*method };
        match method_ref.get_code() {
            None => panic!("AbstractMethod"),
            // 根据方法的描述符和 Code 属性表推入新栈帧，PC 为返回点
            Some((_, max_locals, _, _, _)) => self.frames.push(JavaFrame {
                locals: locals,
                operands: unsafe { locals.add(max_locals * PTR_SIZE) },
                class: class,
                method: method,
                pc: pc,
                max_locals: max_locals as usize,
            }),
        }
        // 返回值为准备执行方法的指令，PC=0
        0
    }
}
```

与 invoke_x 指令相对应的，方法正常结束时，会执行 x_return 指令，该指令会将当前栈帧弹出，并把返回值推入前一个栈帧的操作数栈顶。方法正常返回的代码如代码清单 3-10 所示。

代码清单 3-10 src/mem/stack.rs

```
impl JavaStack {
    pub fn return_normal(&mut self) -> usize {
        let frame = self.frames.pop().expect("empty_stack");
        if !self.is_empty() {
```

```
                let (_, descriptor, af) = &self.method().get_name_type();
                let (_, _, ret) = bytecode::resolve_method(descriptor, af);
                // 弹出栈帧，并移动指针只保留返回值
                let slots = ref.size();
                unsafe {
                    let val = frame.operands.sub(slots * PTR_SIZE);
                    self.operands().copy_from(val, slots * PTR_SIZE);
                    self.update(self.operands().add(slots * PTR_SIZE));
                }
            }
            // 执行引擎跳转到方法返回点
            frame.pc
        }
    }
```

invoke 方法的调用时机是在执行引擎模块中的 for-match 中，所以方法返回值设计为 usize，即下一条指令地址，如果没有异常发生，下一条指令应该是新栈帧对应方法的第一条指令，也就是返回 0。

此外，还需考虑本地方法调用的情况，如果方法修饰符包含 native，则需要进入本地方法的执行过程，由于本地方法目前还未实现，执行引擎可以将其视为一个黑盒子，直接返回原栈帧的下一条指令地址即可。invoke 和 backtrack 方法中都包含了方法描述符的解析，该部分代码只需要处理字符串，所以不再赘述，只需注意 double 和 long 类型的参数占 2 个 slot 即可。

为了保证同时适用于其他 invoke 指令，需要显示判断方法是否存在 static 修饰符，对于不是 static 的方法，第 1 个参数为对象实例 this。现在可以利用栈帧的变更操作在指令分支中实现 invokestatic 和 return 的逻辑，如代码清单 3-11 所示。

代码清单 3-11 src/interpreter/mod.rs

```
    // invokestatic
    0xb8 => {
        // invokestaic 指令之后紧跟 2 字节指向常量池的方法符号引用
        let method_idx = stack.code_at(pc + 1) << 8 | stack.code_at(pc + 2);
        // 栈帧的当前类
        let klass = ...;
        // 方法符号引用三元组
        let (c, (m, t)) = klass.constant_pool.get_javaref(method_idx);
```

```
        // 尝试查找符号 c 对应的类，暂不考虑类加载器
        let klass = ClassArena::load_class(c, Classloader::ROOT).unwrap();
        // 查找方法，根据 2.3 节的讨论，静态方法直接从字节码声明方法中查找
        let method = klass.bytecode.as_ref().unwrap().get_method(m, t);
        let (_, desc, af) = method.get_name_and_type();
        let (_, slots, _) = bytecode::resolve_method(desc, af);
        let class = Arc::as_ptr(&klass.bytecode.as_ref().unwrap());
        let method = Arc::as_ptr(&method);
        // 更新 PC 值
        pc = stack.invoke(class, method, pc + 3, slots);
    }
    …
    // return
    0xb1 => {
        pc = stack.return_normal();
    }
```

由于还没有实现异常处理，代码中凡是遇见错误的情况，都直接粗暴地结束整个虚拟机进程，认真实现异常处理还需要仔细参考 JVM 规范，从已实现的指令看，目前异常只会由非法的类文件引起，执行一定范围内的指令不会导致程序崩溃。

现在到了振奋人心的时刻了，目前虚拟机已经支持了一些简单的指令和静态方法调用，而 Java 程序的入口点正好是一个静态方法，所以可以首次尝试执行一段简单的 Java 代码了！当然，现在还不是遵照传统运行 HelloWorld 程序的时候，虚拟机还只能执行一些简单的算数运算任务，并且没有标准输出，为了观察虚拟机运行过程，可以在指令循环其实位置添加输出语句。现在为整个工程添加可执行部分，如代码清单 3-12 所示。

代码清单 3-12 src/main.rs

```
pub fn main() {
    // 省略命令行参数解析
}

fn resolve_system_classpath(java_home: &str) -> Vec<String> {
    // 省略解析 JAVA_HOME
}

fn resolve_user_classpath(user_classpath: &str) -> Vec<String> {
    // 省略解析用户类路径
}

fn start_vm(class_name: &str, user_classpath: &str, java_home: &str) {
```

```
        let system_paths = resolve_system_classpath(java_home);
        let user_paths = resolve_user_classpath(user_classpath);
        ClassArena::init(user_paths, system_paths);
        let mut main_thread_stack = mem::stack::JavaStack::new();
        let klass = match ClassArena::load_class(c, Classloader::ROOT) {
            Err(class) => panic!(format!("ClassNotFoundException: {}", class)),
            Ok(class) => class,
        };
        let method = klass
            .bytecode
            .get_method("main", "([Ljava/lang/String;)V")
            .expect("Main method not found");
        let class = Arc::as_ptr(&klass.bytecode.as_ref().unwrap());
        let method = Arc::as_ptr(&method);
        main_thread_stack.invoke(class, method, 0, 1);
        interpreter::execute(&mut main_thread_stack);
    }
```

代码清单 3-12 比预期要复杂一些。虚拟机是一个命令行程序，所以加入了一个第三方命令行参数库用于参数解析，main 函数大部分代码都由命令行参数解析逻辑构成。

回忆 2.1.1 小节中的 Java 启动命令，参数-cp 指定用户类路径，使用 linux 风格分割，最后一项是入口类名，系统类路径通过环境变量 JAVA_HOME 获取，复用了系统中预装的 jdk。resolve_system_classpath 和 resolve_user_classpath 函数的作用是将 classpath 转换为 Entry 列表，start_vm 函数首先使用解析后的 classpath 初始化方法区，然后从方法区加载主类，从主类中查找方法名为 main，描述符为([Ljava/lang/String;)V 的方法，最后，分配栈空间作为虚拟机主线程的上下文，以 main 方法为入口开始执行。

现在需要一段简单的 Java 代码作为测试用例。将代码清单 3-1 改为包含 main 方法的类，同时仍旧要注意避免一些较为复杂的指令，或者暂时将指令简化，例如测试字节码中出现的 ldc 指令应该包含 load int/float 常数、字符串常量和 MethodHandle/MethodType3 种情况，在实现堆内存之前，可以只处理 int/float 常数。

新的 Simple.java 文件和对应的字节码如代码清单 3-13 所示。

代码清单 3-13 包含 main 方法的四则运算

```
public class Simple {

    public static void test(int a, long b, float c, double d, boolean f) {
        int aa = a + 1;
        long bb = b - 1;
        float cc = c * 1.0f;
        if (f) {
            double dd = d / 1.0d;
        }
    }

    public static void main(String[] args) {
        test(1, 1L, 1.5f, 0.5d, true);
    }
}
```

本着指令按需实现的目标，根据 Simple.class 字节码反编译的结果，需要实现大约 20 条指令才能运行，除前一节已实现的 xload 指令，还需要实现 iconst、iadd 等指令。

为了观察虚拟机执行过程，在 JavaFrame 增加一个 dump 方法用于观察栈帧状态变化，在每条语句执行前调用即可。如代码清单 3-14 所示。

代码清单 3-14 src/mem/stack.rs

```
impl JavaFrame {
    pub fn dump(&self, pc: usize) {
        let (name, descriptor, _) = self.method.get_name_and_descriptor();
        println!("current class: {:?}", self.klass.bytecode.get_name());
        println!("current method: {:?} {:?}", name, descriptor);
        println!("locals: {:02x?}", self.locals);
        println!("stacks: {:02x?}", self.operands);
        println!("pc: {:?}", pc);
        println!("instructions: {:02x?}\n",
self.method.get_code().expect("").2);
    }
}
```

编译 Rust 代码，启动虚拟机执行 Simple.class，运行结果如图 3-8 所示。

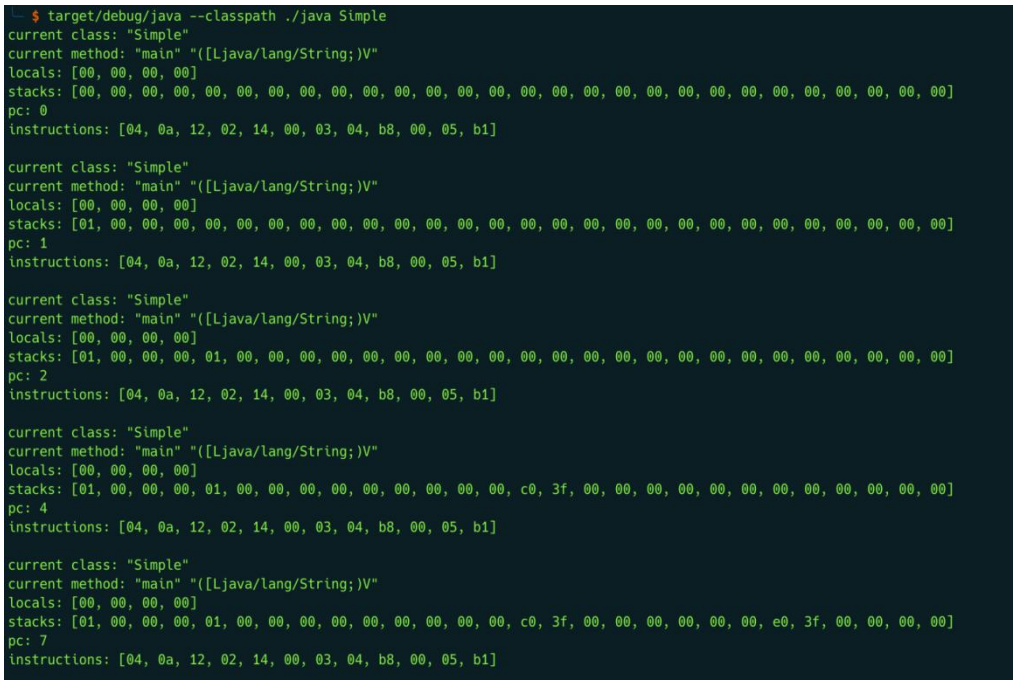

图 3-8 Simple.class 运行结果

3.4

单线程程序是枯燥的，Java 从很早以前就将线程内置在语言标准库中，屏蔽了操作系统的差异性。现在，内置多线程几乎是新兴编程语言的必备特性。但是，Java 语言中的线程只是一个普通类，其中的代码除了 native 方法之外，看上去与普通类并无二致，虚拟机用户所能感知的多线程带来的并发效果实际是由虚拟机完成的。

3.4.1 广义的线程概念

根据教科书的说法，线程是操作系统时间片的基本调度单位。作为程序员再熟悉不过的概念之一，线程的准确定义却是模糊的，因为“线程”的名称是各系统中线程概念的公约数，不同平台的实现不尽相同，比如在 Linux 内核中，线程和进程的实现并没有明确区分。

在计算机世界中，一切都讲究抽象。咬文嚼字地说，线程存在的意义并不是为了单纯的并发执行，因为 CPU 本身并不关心当前执行的是哪一段代码，如果仅仅是需要同时执行多个程序，完全可以每个程序都执行若干指令就切换。所以，确切地讲，线程是为了支持程序自身控制多个执行流而抽象出的可管理的计算时间资源。之所以如此咬文嚼字，是因为要实现虚拟机，必然无法绕过线程，我们无法直接管理 CPU 时间片，却能管理底层操作系统提供的抽象实体。

如果将 Java 虚拟机也视为一个操作系统，那么可以说虚拟机线程是为了支持运行于虚拟机的程序并发执行控制而存在的概念，具体的执行流控制机制对用户程序透明，但却有着和底层操作系统提供的并发原语类似的控制机制，其实本质就是相同的，因为几乎所有 Java 虚拟机的并发实现都委托给了操作系统。

提到实现方式，不得不提到并发控制实现方式不同的 Go 语言。Go 语言的并发模型广受欢迎，内置的 goroutine 对应绿色线程或者用户线程的概念。即不使用操作系统提供的并发支持，而是在运行时中包含了一个调度器，允许程序在不直接访问操作系统任务调度功能的情况下实现并发，使得执行流切换（这里为了避免误导，不使用线程一词，尽管其也符合广义的时间片资源概念）更加廉价。

操作系统线程并不是一个严谨的说法，更深层次的，线程可以一级一级往下追溯，一直到实际执行硬件指令的时间片。每个层级可以复用下一层级提供的时间片资源，向上提供封装后的并发支持。从上往下，线程映射关系依次是：运行于虚拟机的用户程序线程（封装的 Thread 类）→虚拟机线程（虚拟机封装操作系统线程或自行实现用户态调度器）→操作系统用户态线程（系统调用暴露的线程支持）→操作系统内核线程（内核调度器的基本单位）→硬件线程（拥有寄存器级线程上下文，执行硬件指令的时间片）。

3.4.2 映射用户态线程

虚拟机线程和操作系统用户态线程使用 1：1 的映射关系是最简单的实现方式。严格说来，实现虚拟机的宿主语言同样和用户态线程也存在映射关系，只不过在 Rust 或 C++实现的虚拟机中，这种映射关系也是 1：1 的，通常也就省略这一层映射关系。

线程的每一层映射可以有自己的线程上下文概念，同时，也可以向上暴露出一些上下文信息，比如 Java 程序可以获取当前线程的 ID，该 ID 就是虚拟机通过运行时支持所暴露的信息。虽然直接映射操作系统线程，但是虚拟机并没有将虚拟机线程 ID 设置为操作系统的 PID，至少 Hotspot 是如此。如果在 Linux 上使用 htop 工具查看运行的多线程 Java 程序，会发现多条名字相同的记录，由于 Linux 不明确区分进程和线程，所以每条记录实际上是操作系统的一个线程，该线程 ID 和使用 jcmd 等工具查看的虚拟机线程 ID 不同。

在前一节实现的静态方法调用中，我们直接用宿主语言的主线程来执行字节码，相当于使用本地线程映射虚拟机线程，但是作为线程，自然包含一些上下文信息。结合已知的执行引擎的相关内容，虚拟机的线程主要对应方法栈和 PC 寄存器两部分，所以方法栈和 PC 寄存器可以移动到线程上下文中作为成员。

在暂不考虑向 Java 层暴露相应接口的情况下，虚拟机内部也需要对线程实体进行操作，就如同内核中使用 task 结构体代表进程概念进而映射硬件时间片。例如线程状态和线程 ID 等信息也可以参考内核中的设计加入到上下文中。

除此之外，Java 中还有线程上下文类加载器的概念，作为默认情况下当前线程加载的类的类加载器，由于还未实现对象和堆，暂时使用可拷贝的枚举表示类加载器实例。所谓可拷贝，在 Rust 中是指，使用等号赋值或函数参数传递时，深拷贝一份对象值进

行赋值或传递参数，不会改变原对象的状态，参见第 1 章 Rust 语法概览。

综上讨论，初步尝试定义线程上下文，如代码清单 3-15 所示。

代码清单 3-15 src/interpreter/thread.rs

```
pub struct ThreadContext {
    // 当前线程的 PC 寄存器值
    pub pc: usize,
    // 当前线程的方法栈
    pub stack: JavaStack,
    // 尚不明确的线程状态
    pub status: AtomicU32,
    // 线程 id
    pub id: u32,
    // 线程上下文类加载器
    pub classloader: Classloader,
}
```

线程上下文存在的目的是为了映射下一级线程资源的实体，最终需要为执行引擎提供相关的信息。立刻能修改的正是刚实现的 main 方法调用，因为方法栈和 PC 已经被移动到线程上下文中，execute 方法不再接受&mut JavaStack 参数，取而代之的是&mut ThreadContext。加入线程上下文的概念之后，需要在指令执行过程中实现中断的场景（如类加载、异常处理）不必再笨拙地传入和返回 PC 值了，直接将 ThreadContext 的可变引用作为参数传递，修改 PC 即可。为 ThreadContext 添加新建线程的方法，如代码清单 3-16 所示。

代码清单 3-16 src/interpreter/thread.rs

```
impl ThreadContext {
    pub fn new_thread(id: u32,
                      classloader: Classloader,
                      class_name: &str,
                      method_name: &str,
                      method_descriptor: &str) {
        let context = ThreadContext::new(id, classloader);
        let class = match ClassArena::load_class(class_name, &mut context) {
            Err(class) => panic!(format!("ClassNotFound: {}", class)),
            Ok((class, _)) => class,
        };
        let method = class
            .bytecode
```

```
                .as_ref()
                .unwrap()
                .get_method(method_name, method_descriptor)
                .expect("Method not found");
            context.stack.invoke(
                Arc::as_ptr(&class.bytecode.as_ref().unwrap()),
                Arc::as_ptr(&method),
                0,
                1,
            );
            interpreter::execute(&mut context);
        }
    }
```

上述代码提供了新建线程的方法，但没有 Rust 多线程特性的影子。这是因为启动虚拟机时，执行用户程序 main 方法可以复用上述代码，直接使用虚拟机进程的主线程，如果需要启动新线程，需要将当前线程创建的对象所有权移交给新线程。

Rust 遵循“可变不共享，共享不可变”的原则，线程上下文属于线程私有数据，无需在多个线程间共享，只有移交所有权之后，线程内的局部变量才能绑定对象。

3.4.3 自定义虚拟机指令 invokeasync

本书开发 Java 虚拟机的目的是为了更好地学习虚拟机原理，而学习的最好效果就是能过灵活运用所学知识。在离原原本本实现 Java 虚拟机线程还很遥远的情况下（因为依赖本地方法调用），本节介绍如何扩展一条自定义虚拟机指令 invokeasync，实现原生指令达到程序并发执行的效果。

Java 语言的多线程支持以标准库的形式提供，在一些第三方框架中，也可以简洁地为方法添加指定注解，框架就会以异步的方式调用该方法，比如广泛运用于企业级开发领域的 spring 框架。这类框架通常是以运行时反射或者字节码增强技术实现此类功能，Go 语言则更进一步，直接在语法层面支持执行异步方法。现在，作为虚拟机实现者，我们可以以更为底层的方式实现类似 Go 语言的并发效果。

首先分析虚拟机在正常情况下如何启动线程。用户程序需要继承 Thread 类或者实现 Runnable 接口，再调用 start 方法启动线程。start 方法是一个本地方法，理论上本地方法可以不受限制地实现任何功能，尽管还没有介绍本地方法的调用过程，现在仍然可以猜测在 start 方法内，一定有新建线程上下文，并且使用操作系统线程执行 run 方法的逻辑。

这个过程和虚拟机启动时执行用户程序 main 方法一致，只不过需要将指令流水线映射到新的操作系统线程中。既然前一节已经介绍了从头建立一个线程的实现方式，现在可以通过添加指令绕过复杂的本地方法调用过程达到新建线程的目的。

选择一个未被使用的指令编码，比如 0xd3，定义为 invokeasync 指令，为了简化实现，规定 invokeasync 只允许调用静态方法，指令的参数与 invokestatic 一致，操作数也是指向常量池的方法句柄索引。如代码清单 3-17 所示。

代码清单 3-17 定制指令 invokeasync

```
// 自定义指令，不存在于 JVM 规范中
0xd3 => {
    // 和 invokestatic 指令一致的参数处理，注意此处已经将方法栈移动到了线程上下文中
    let method_idx = context.stack.code_at(context.pc + 1) << 8
        | context.stack.code_at(context.pc + 2);
    let class = unsafe { context.stack.class_ptr().as_ref() }.unwrap();
    let (c, (m, t)) = class.constant_pool.get_javaref(method_idx);
    let ctx_classloader = context.classloader;
    std::thread::spawn(move || {
        ThreadContext::new_thread(ctx_classloader, c, m, t);
    });
    context.pc = context.pc + 3;
 }
```

根据设想，当执行引擎执行到 0xd3 指令时，虚拟机必然已存在至少一个线程。使用宿主语言启动一个新线程，并且将所需的参数传入建立虚拟机线程开始执行。原虚拟机线程的 PC 值更新至下一条指令。

自定义的指令需要对应的编译器支持，例如添加方法修饰符 async，在 Java 语法层面上规定必须和 static 关键字一起连用。关于编译器的知识，已经超出了本书的讨论范围，这里并没有修改 Java 编译器的打算，而是通过修改编译后的二进制文件模拟测试 invokeasync 指令。首先准备一个签名为 public static void (int, int)的方法，如代码清单 3-18 所示。

代码清单 3-18 InvokeAsync.java

```
public class InvokeAsync {
    public static void main(String[] args) {
        invokeasync(1, 1);
        invokestatic(1, 1);
    }

    public static void invokestatic(int a, int b) {
        int c = a + b;
    }

    public static /**async**/ void invokeasync(int a, int b) {
        int c = a + b;
    }
}
```

使用二进制编辑器打开编译出的 InvokeAsync.class 文件，如 vim 的二进制编辑模式。找到第一条 invokestatic 指令（0xb8），将其替换为 invokeasync（0xd3），如图 3-9 所示。然后使用虚拟机测试执行 InvokeAsync，运行结果如图 3-10 所示。

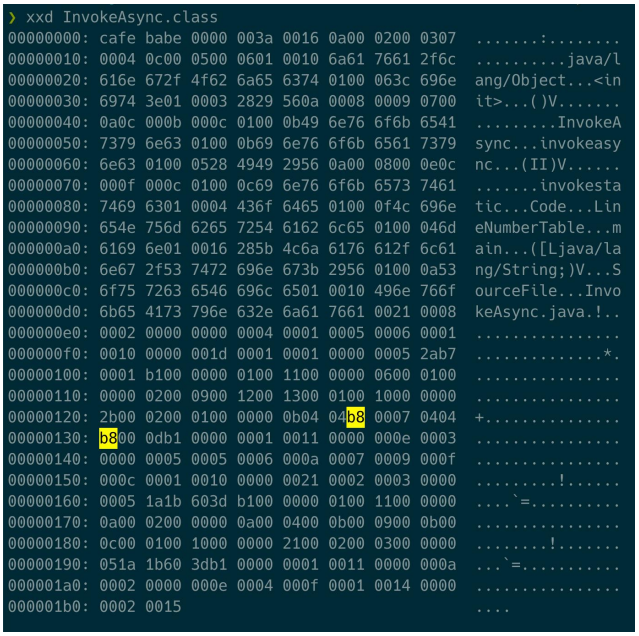

图 3-9 手动修改字节码

```
method layer: 0
current class: "InvokeAsync"
current method: "invokeasync" "(II)V"
method layer: 1
current class: "InvokeAsync"
current method: "invokestatic" "(II)V"
locals: [01, 00, 00, 00, 01, 00, 00, 00, 00, 00, 00, 00]
operands: [01, 00, 00, 00, 01, 00, 00, 00]
pc: 2
locals: [00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00]
operands: [00, 00, 00, 00]
pc: 1
[pc]: 60
code: [1a, 1b, 60, 3d, b1]

[pc]: 1b
code: [1a, 1b, 60, 3d, b1]

method layer: 1
current class: "InvokeAsync"
current method: "invokestatic" "(II)V"
locals: [01, 00, 00, 00, 01, 00, 00, 00, 00, 00, 00, 00]
operands: [02, 00, 00, 00]
pc: 3
```

图 3-10 invokeasync 指令执行结果

从运行结果可以看出，输出变得无序，说明指令已经在并发执行。由于 invokeasync 使用本地线程的方式实现并发，开销较大，如果更进一步，可以将多条 invokeasync 的流水线映射到一组操作系统线程中，实现 M：N 的绿色线程。

在 Java 中加入类似 Go 语言的语法级并发支持是一件看上去很酷的事情，虽然现实

中的 Java 虚拟机不大可能实现这样的特性，但也许有朝一日你实现自己的编程语言时，可以考虑加入类似的特性。

3.5

在第 2 章末尾记录的几个方法区遗留问题中，类初始化是一项非常重要的任务。在执行引擎模块中，当使用某一个类的代码时，必须保证其初始化完成。由于类初始化可能包含指令逻辑，且正好也是以静态方法的形式存在，现在正是完善该功能的时机。

在 Java 虚拟机看来，类初始化方法也是一个普通的静态方法，只不过用户程序无法显示调用，由编译器赋予了一个特殊的名字<clinit>，对应 Java 源代码中的 static 代码块。后续还会在 Java 标准库中遇到许多由虚拟机主动调用的特殊方法，这些方法虽然不是语言的一部分，但是需要虚拟机做特殊处理，从某种程度上讲，虚拟机和语言的边界也因此变得模糊。

类初始化方法的描述符恒定为()v，使用方法名和描述符，通过方法区提供的类查找功能，我们能轻易获取到方法的指令序列。下面通过一个带有静态代码块的 Java 程序测试类初始化代码，如代码清单 3-19 所示。

代码清单 3-19 InitialOnLoad.java

```
public class InitialOnLoad {
    static {
        System.out.println("Class initialized.");
    }

    public static int load = 0;

    public static void test() {}
}
```

使用 javap 查看代码清单 3-19 对应的字节码，会发现确实有一个名为<clinit>的静态

方法，不过 javap 已经将<clinit>自动翻译为了 static{}，如图 3-11 所示。

```
static {};
  descriptor: ()V
  flags: ACC_STATIC
  Code:
    stack=2, locals=0, args_size=0
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3                  // String Class initialized.
       5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: iconst_0
       9: putstatic     #5                  // Field load:I
      12: return
    LineNumberTable:
      line 3: 0
      line 6: 8
```

图 3-11 类初始化方法

那么虚拟机在什么情况下才会触发类初始化呢？类初始化是类加载的一个步骤，当执行引擎尝试在方法区中查找一个类时，如果类没有被加载过，就需要经历类加载过程，也就自然需要执行类初始化方法。

从现在的视角看来，类初始化相关功能放在第 2 章 Klass 部分显然更优雅，但现实情况是，需要将线程上下文作为参数传递给方法区模块，才能选择正确的栈空间执行类初始化方法，这对于目前的进展依然是超前的，而且这无疑也是一个对方法区代码的巨大改动。在陌生的系统中经常就会面临这样的问题，就像登山时一路顺着上山的方向走，有时也会遇到因为路线选择错误而折返的情况。

考虑类初始化方法也像一个普通静态方法一样需要在执行引擎中执行，只是触发时机需要由类加载模块确定，一种比较笨拙的解决办法是为 Klass 增加 initialized 字段，解释器尝试查找类时，如果发现类未被初始化，则提前结束当前正在执行的指令，转而调用类初始化方法。由于整个解释器只是一个指令循环，如果修改了某些状态再结束当前指令，则会出现指令执行一半的情况。所以，类初始化的检测必须放在指令即将改变任何内存状态之前。类加载过程中的初始化就是耳熟能详的中断执行过程。

根据规范，类加载过程必须是线程安全的，虽然类容器已经实现为线程安全的容器，

但为了避免同一个类被多次执行类初始化方法，虚拟机需要为 Klass 的加载状态上锁。

此外，JVM 规范还规定父类必须先于子类加载，所以类加载同时是一个典型的递归过程。程序员遇到典型的递归问题时往往会偏向于直接使用递归调用实现，但执行引擎本身就是基于栈结构进行方法调用，此类问题应该优先使用中断的方式实现。具体做法是，保持原有的指令循环结构不变，在某条指令执行过程中需要加载类时，按照静态方法调用的过程将类初始化方法对应的栈帧推入栈顶，并提前结束当前指令（在循环中使用 continue 语句），在新一轮的循环中，执行引擎发现新的栈顶帧已经变为了类初始化方法，继续逐条指令执行即可。

基于以上讨论，修改方法区类加载部分，如代码清单 3-20 所示。

代码清单 3-20 src/mem/metaspace.rs

```
impl ClassArena {
    // 递归方法，返回值添加一个标志位用于告诉执行引擎知否需要中断
    pub fn load_class(
        class_name: &str,
        context: &mut ThreadContext,
        classloader: Option<Classloader>,
    ) -> Result<(Arc<Klass>, bool), String> {
        // 省略部分未修改代码
        match classarena!().get(&class_name) {
            Some(klass) => {
                // 省略类加载器的比较逻辑
            },
            None => {
                // 保证类加载的线程安全
                let _ = class_arena!().mutex.lock().unwrap();
                // 双锁检查
                if let Some(loaded) = classarena!().get(&class_name) {
                    // 省略类加载器的比较逻辑
                }
                let class = match Self::parse_class(&class_name)
                    .map(|c| Arc::new(c))
                    .ok_or(Err(class_name.to_owned()))?;
                // 推入类初始化方法作为新栈帧，先于父类和接口入栈，最后执行
                initialize_class(&class, context);
                let mut interfaces: Vec<Arc<Klass>> = vec![];
                for interface in class.get_interfaces() {
                    interfaces.push(
```

```
                        Self::load_class(interface, context, cl)?.0
                    );
                }
                let superclass = if !class.get_super().is_empty() {
                    Some(
                      Self::load_class(class.get_super(), context, cl)?.0
                    )
                } else {
                    None
                };
                let klass = Arc::new(Klass::new(class,
                                                cl,
                                                superclass,
                                                interfaces));
                classarena!().insert(class_name, klass.clone());
                Ok((klass, false))
            }
        }
    }

    fn initialize_class(class: &Arc<Class>, context: &mut ThreadContext) {
        match class.get_method("<clinit>", "()V") {
            // 仅仅将类初始化方法推入方法栈，引导执行引擎中断执行
            Some(clinit) => {
                context.pc = context
                    .stack
                    .invoke(
                        Arc::as_ptr(&class),
                        Arc::as_ptr(&clinit),
                        context.pc,
                        0
                    );
            }
            None => {}
        }
    }
}
```

上述代码重新实现了类加载的核心逻辑，并且考虑类加载器的因素。解释器查找类时，需要将当前线程上下文传入，用于类加载器调用类初始化方法。代码中考虑了父类、接口和对应数组类的加载情形，initialize_class 并不会实际执行类加载方法，只是将准备好的类初始化方法栈帧推入方法栈，只有当 load_class 方法返回时，执行引擎才会重新获得执行权，所以子类的 initialize_class 放在父类之前。

已实现的静态方法调用只需稍作修改。在改变任意状态之前，尝试获取所需的类，

如果类加载方法返回了中断标志，则中断当前执行（当前指令循环 continue），因为 load_class 已经将类初始化的栈帧推入了栈顶，执行引擎无需做其它任何额外处理。

main 方法是一个特例，因为该方法是整个程序的入口，也是主线程栈的最底层帧，如果直接调用 main 方法，入口类必定没有被加载。所以，在虚拟机开始执行 main 方法前，应该先调用入口类的初始化方法，递归地执行完继承链的类初始化方法之后，线程栈变为空，再调用 main 方法。

对应的修改在创建线程部分，为 new_thread 函数添加一个 init 参数，当线程底帧的类没有加载时，先进行类初始化，代码中只需增加一个额外的判断即可。启动虚拟机时传递 init 参数为 false 即可保证入口类正确初始化，流程如图 3-12 所示。

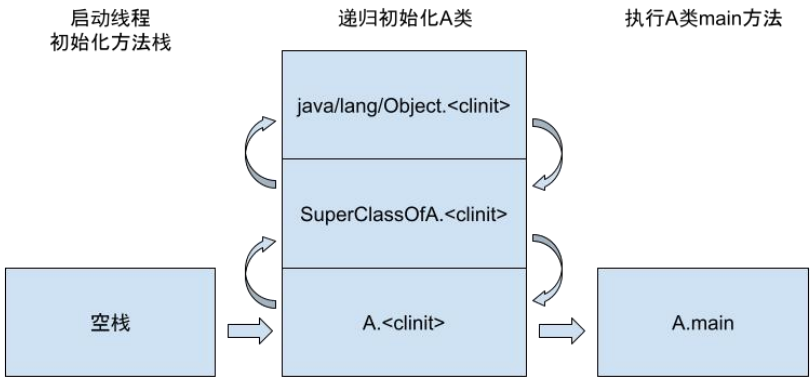

图 3-12 保证类加载的 main 方法执行过程

图 3-12 只是为了表达继承链的加载顺序，除了执行 main 方法之外，java/lang/Object 还有更优先的机会被加载。

3.6

本章的主题围绕着执行引擎展开，实现了执行引擎的基础功能部分，使目前的虚拟机能执行一些简单的 Java 程序，向着虚拟机雏形迈出了阶段性的一步。

本章首先以白纸画图的方式，通过人工阅读字节码执行了一段简单的 Java 程序，建立对栈虚拟机执行过程的感性认知，并大致了解执行引擎所需的依赖模块，建议你自行重复此过程以加深对栈虚拟机模型的理解。

第 2 节主要讨论如何合理地定义方法栈和栈帧的数据结构，然后给出了执行引擎的简单流水线结构，将第 1 节手工执行的指令翻译为 Rust 程序。Java 虚拟机存在的主要目的就是为了执行用户程序，一个合理且易于扩展的执行引擎结构就显得格外重要，尽管 Rust 通常情况下不推荐使用裸指针类型，但对于实现较为底层的系统，裸指针依然是综合考量下的最佳选择，第 4 节和第 5 节对执行引擎的完善也充分展示了使用裸指针实现方法栈的便捷。

第 3 节讨论了 Java 虚拟机的方法调用过程，这是执行引擎除了单条指令执行外最基本的功能需求。到第 3 节为止，虚拟机仅实现了多种方法调用指令中的静态方法调用，不涉及堆内存的使用和方法的动态绑定，将范围限定在执行引擎内部，重点关注了方法调用前后对应的栈帧变化过程。正确地实现静态方法调用之后，立即将该功能运用于 main 方法的执行，也因此赋予了虚拟机正式执行 Java 程序的能力。

第 4 节讨论了线程的基础概念和 Java 虚拟机中的线程支持。虚拟机线程并不完全等价于操作系统线程，在谈论编程语言的线程支持时，多数情况下是在谈论程序的并发执行能力。本书最后选用了以 1：1 映射操作系统线程的方式实现线程，同时也修改了方法栈的结构以定义虚拟机的线程上下文。由于暂未实现本地调用，第 4 节最后另辟蹊径，以扩展字节码指令的方式完成了多线程特性的测试。

第 5 节重新回到了类加载器的话题，得益于第 3 节静态方法调用的支持和第 4 节线程上下文的定义，类加载过程中的初始化步骤得以实现。类加载器现在可以通过传递线

程上下文，以中断的方式引导执行引擎执行类初始化方法，反过来看，执行引擎也完善了中断执行流程，允许在指令执行中途转而执行类加载方法，再恢复当前指令的执行。在第 5 章会看到，异常处理也需要使用中断执行的结构实现。