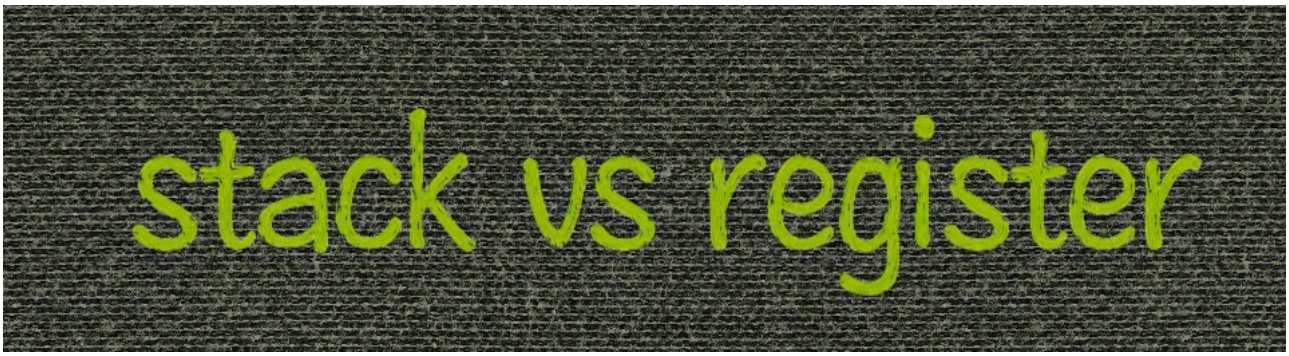
字节码是运行在 JVM 上的，为了能弄懂字节码，需要对 JVM 的运行原理有所了解。这篇文章将以栈帧为切入点理解字节码在 JVM 上执行的细节。

0x01 虚拟机：stack based vs register based



虚拟机常见的实现方式有两种：Stack based 的和 Register based。比如基于 Stack 的虚拟机有Hotspot JVM、.net CLR，这种基于 Stack 实现虚拟机是一种广泛的实现方法。而基于 Register 的虚拟机有 Lua 语言虚拟机 LuaVM 和 Google 开发的安卓虚拟机 DalvikVM。

两者有什么不同呢？举一个计算两数相加的例子：`c = a + b`
基于 HotSpot JVM 的源码和字节码如下

```
源码
void bar(int a, int b) {
    int c =  a + b;
}

对应字节码
0: iload_1 // 将 a 压入操作数栈
1: iload_2 // 将 b 压入操作数栈
2: iadd    // 将栈顶两个值出栈，相加，然后将结果放回栈顶
3: istore_3 // 将栈顶值存入局部变量表中第 3 个 slot 中
```

基于寄存器的 LuaVM 的 lua 源码和字节码如下，查看字节码使用luac -l -l -v -s test.lua 命令

```
源码
local function my_add(a, b)
    return a + b;
end

对应字节码
1    [3] ADD           2 0 1
```

基于寄存器的 add 指令直接把寄存器 R0 和 R1 相加，结果保存在寄存器 R2 中。

基于栈和基于寄存器的过程对比如下

基于栈和寄存器的指令集各有优缺点，基于栈的指令集移植性更好，代码更加紧凑、编译器实现更加简单，但完成相同功能所需的指令数一般比寄存器架构多，需要频繁的入栈出栈，栈架构指令集的执行速度会相对而言慢一些。

为了理解字节码的细节，我们需要详细了解字节码的执行过程。众所周知，Hotspot JVM 是一个基于栈的虚拟机，每个线程都有一个虚拟机栈，存储了「栈帧」。每次方法调用都伴随着栈帧的创建销毁。

0x02 栈帧

栈帧（Stack Frame）是用于支持虚拟机进行方法调用和方法执行的数据结构
栈帧随着方法调用而创建，随着方法结束而销毁，栈帧的存储空间分配在 Java 虚拟机栈中，每个栈帧拥有自己的**局部变量表（Local Variables）**、**操作数栈（Operand Stack）** 和 **指向运行时常量池的引用**

如下图所示

局部变量表

每个栈帧内部都包含一组称为局部变量表（Local Variables）的变量列表，局部变量表的大小在编译期间就已经确定。Java 虚拟机使用局部变量表来完成方法调用时的参数传递，当一个方法被调用时，它的参数会被传递到从 0 开始的连续局部变量列表位置上。当一个实例方法（非静态方法）被调用时，第 0 个局部变量是调用这个实例方法的对象的引用（也就是我们所说的 this ）

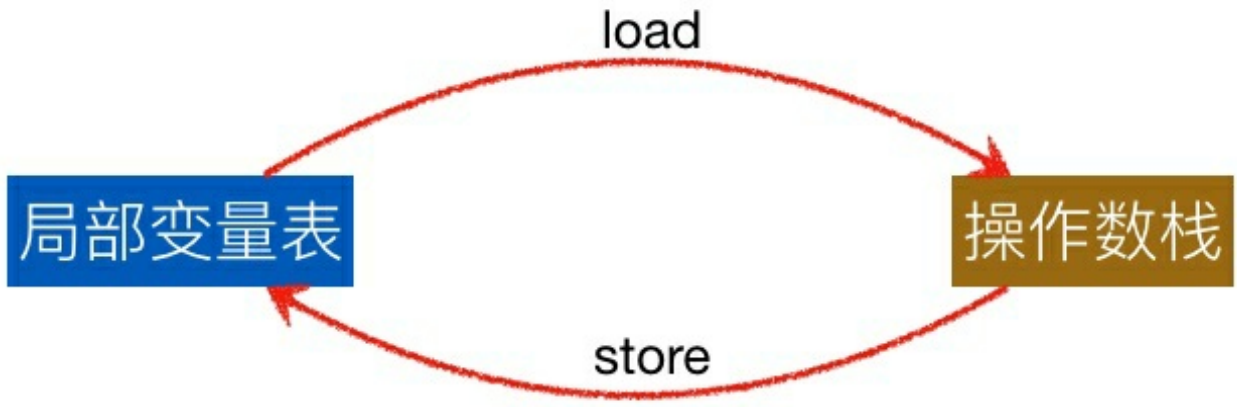
操作数栈

每个栈帧内部都包含了一个称为操作数栈的后进先出（LIFO）栈，栈的大小同样也是在编译期间确定。Java 虚拟机提供的一些字节码指令用来从局部变量表或者对象实例的字段中复制常量或者变量到操作数栈，也有一些指令用于从操作数栈取走数据、操作数据和把操作结果重新入栈。在方法调用时，操作数栈也用来准备调用方法的参数和接收方法返回的结果。

比如 iadd 指令用来将两个 int 类型的数值相加，它要求执行之前操作数栈已经存在两个由前面其它指令放入的 int 型数值，在 iadd 指令执行时，两个 int 值从操作数栈中出栈，相加求和，然后将求和的结果重新入栈。

比如 1 + 2 这样的指令执行过程如下

整个 JVM 指令执行的过程就是局部变量表与操作数栈之间不断 load、store 的过程



我们再来看一个稍微复杂一点的例子

```
public class ScoreCalculator {
    public void record(double score) {
    }

    public double getAverage() {
        return 0;
    }
}
public static void main(String[] args) {
    ScoreCalculator calculator = new
ScoreCalculator();

    int score1 = 1;
    int score2 = 2;

    calculator.record(score1);
    calculator.record(score2);

    double avg = calculator.getAverage();
}
```

javap 查看字节码输出如下

```
public static void main(java.lang.String[]);
descriptor: ([Ljava/lang/String;)V
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=3, locals=6, args_size=1
     0: new           #2                  //
class ScoreCalculator
     3: dup
     4: invokespecial #3                  //
```

```
Method ScoreCalculator."<init>":()V
     7: astore_1

     8: iconst_1
     9: istore_2

    10: iconst_2
    11: istore_3

    12: aload_1
    13: iload_2
    14: i2d
    15: invokevirtual #4                  //
Method ScoreCalculator.record:(D)V

    18: aload_1
    19: iload_3
    20: i2d
    21: invokevirtual #4                  //
Method ScoreCalculator.record:(D)V

    24: aload_1
    25: invokevirtual #5                  //
Method ScoreCalculator.getAverage:()D
    28: dstore        4

    30: return
```

- 0 ~ 7：新建了一个 ScoreCalculator 对象，使用 astore_1 存储在局部变量 calculator 中：astore_1 的含义是把栈顶的值存储到局部变量表下标为 1 的位置上，这里为什么会有一个 dup，我们后面会讲到
- 8 ~ 11：iconst_1 和 iconst_2 用来将整数 1 和 2 加载到栈

顶，istore_2 和 istore_3 用来将栈顶的元素存储到局部变量表 2 和 3 的位置上

- 12 ~ 15：可以看到 *store* 指令会把栈顶元素移除，所以下次我们要用到这些局部变量时，需要使用 load 命令重新把它加载到栈顶。比如我们要执行calculator.record(score1)，对应的字节码如下

```
12: aload_1
13: iload_2
14: i2d
15: invokevirtual #4 // Method
ScoreCalculator.record:(D)V
```

可以看到 aload_1 先从局部变量表中 1 的位置加载 calculator 对象，iload_2 从 局部变量表中 2 的位置加载一个整型值，i2d 这个指令用来将整型值转为 double 并将新的值重新入栈，到目前为止参数全部就绪，可以用 invokevirtual 执行方法调用了

- 24 ~ 28：同样是一个普通的方法调用，流程还是先 aload_1 加载 calculator 对象，invokevirtual 调用 getAverage 方法，并将 栈顶元素存储到局部变量表下标为 4 的位置上

有一点需要注意的是 javap 输出的locals=6，但是我们目前看到的局部变量只有args、calculator、score1、score2、avg这 5 个，为什么这里等于 6 呢？这是因为 avg 为 double 型变量，需要两个槽位（slot）
整个过程局部变量表如下图所示

0	1	2	3	4	5
args	calculator	score1	score2	avg	

局部变量表

其实局部变量表可以通过 javap 用 -l 参数直接输出，但是我们用 javap -v -p -l MyLocalVariableTest 并没有输出任何局部变量表相关的信息。这是因为默认情况下局部变量表属于调试级别的信息，javac 编译的时候并没有编译进字节码，我们可以加上 javac -g 生成字节码的时候同时生成所有的调试信息，如下所示

```
javac -g  MyLocalVariableTest.java
javap  -v -p -l   MyLocalVariableTest
LocalVariableTable:
Start  Length  Slot  Name   Signature
    0      31     0  args   [Ljava/lang/String;
    8      23     1 calculator
LScoreCalculator;
   10      21     2 score1   I
   12      19     3 score2   I
   30       1     4   avg   D
```

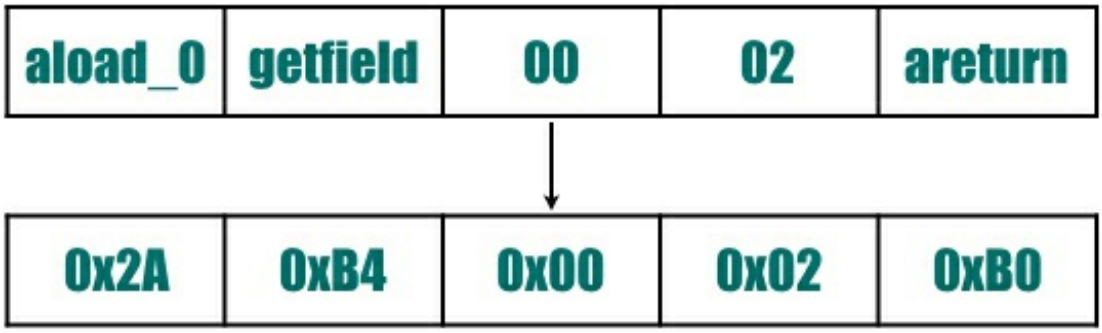
0x03 从二进制看 class 文件和字节码

```
public class Get {
    String name;

    public String getName() {
        return name;
    }
}

javap 查看字节码如下

public java.lang.String getName();
descriptor: ()Ljava/lang/String;
flags: ACC_PUBLIC
Code:
  stack=1, locals=1, args_size=1
     0: aload_0
     1: getfield      #2 / Field
name:Ljava/lang/String;
     4: areturn
```



直接从二进制来看下这个 class 文件 xxd Get.class

我们可以手动用 16 进制编辑器去修改这些字节码文件，只是比较容易出错，所以产生了一些字节码操作的工具，最出名的莫过于 ASM 和 Javassist。我们后面讲到软件破解的时候，会介绍直接修改字节码和通过 ASM 动态修改字节码这两种方式

0x04 小结

一起来回顾一下这篇文章的要点：

- 第一，基于栈和基于寄存器指令集的优劣势；
- 第二，讲解了 JVM 栈帧的构成（局部变量表、操作数栈、指向运行时常量池的引用），顺带讲解了 `javap -l` 参数和其在局部变量表中的应用；
- 第三，从类文件二进制角度看字节码的实现，并引出 ASM 字节码改写技术。

0x05 思考

最后，给你留一道作业，用 Java 写一个简单的 class 文件解析工具支持输出函数列表。

欢迎你在留言区留言，和我一起讨论。