Lab5 Instructions

！！！注意看这里！！！

1. 用JDK8（只能用8！）
2. 用oracle的jdk
3. 正确设置JAVA_HOME
4. 前几次实验没问题**不代表**上面三个配置正确了！
5. 在实现对long/double的操作时，请把**低4字节先入栈**/储存在局部变量表中**较低**的slot里
6. 各种类型转换指令可以直接使用Java语言提供的强制类型转换，例如 `int result = (int)toConvert;`

简介

Java程序是以字节码的形式储存在classfile中的，而JVM能够直接执行的指令也正是Java字节码。 在前面几次实验中，我们已经正确读取了classfile的内容，为指令的执行做好了准备， 而这次大作业的内容主要是实现几条JVM指令，**对应手册的第六章。** 在实现指令之前，为了加深理解，我们在JVM的"运行时环境"这部分框架代码中留了几个填空（已用TODO标记）， 这部分主要对应手册的2.5节。 在完成这几个填空之后，就可以尝试着实现JVM指令了！

jvm的运行时环境

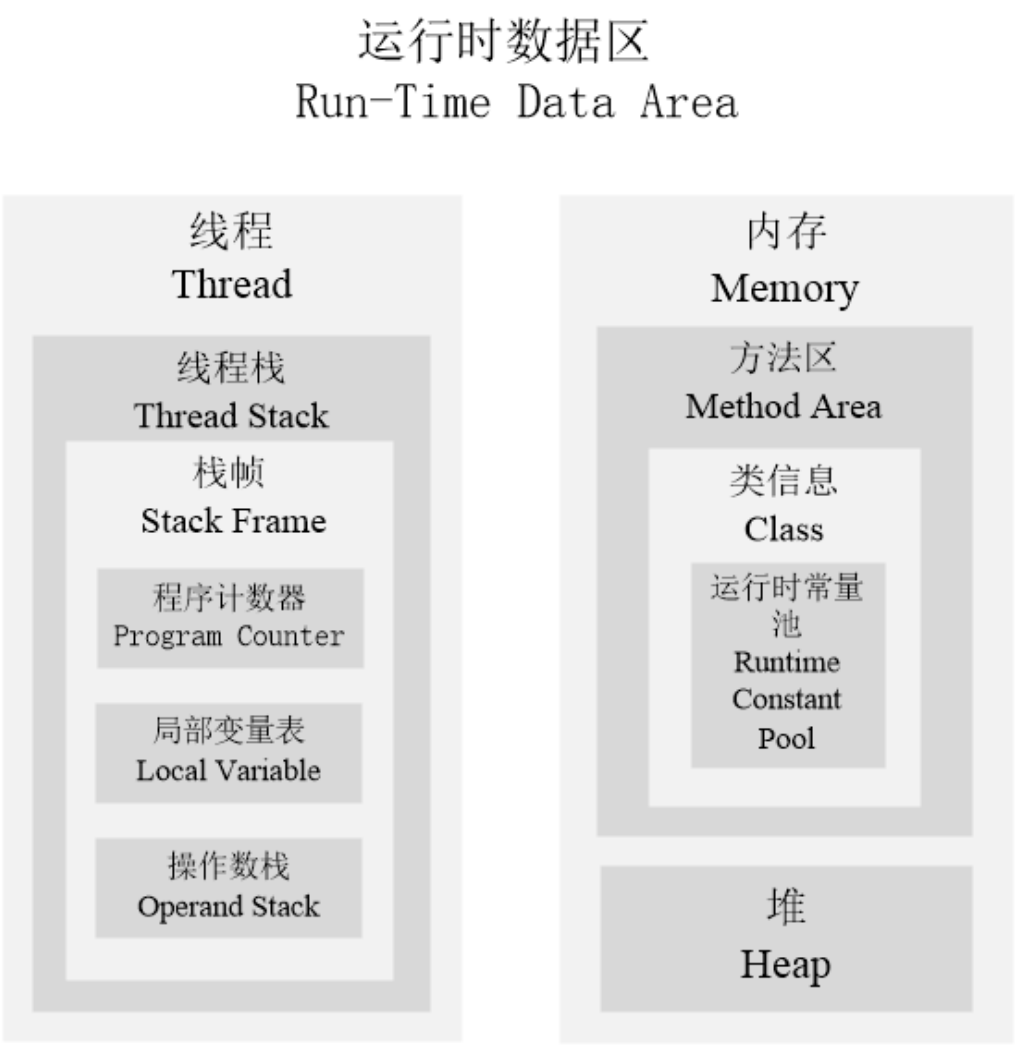
"运行时环境"这个词指的是JVM为执行指令所提供的计算模型。 例如之前计基课上学到的32位MIPS机器，它有32个通用寄存器，能够访问0 ~ 2^32−1范围内的内存。

再比如我们平常使用的x86-64架构的CPU有16个通用寄存器，能够按照一定的规则（具体规则比较复杂，不要在意这些细节～）来访问内存。

而JVM与它们的不同之处在于，JVM的指令采取了"**栈式指令风格**"，使用操作数栈来实现复杂的计算功能。 以一条加法指令为例，在MIPS机器上可以表示为`ADDI R1,R1,$1`，在x86-64机器上可以表示为`addl %eax, $1`， 而在JVM中可能通过这样的指令序列来表示（把操作数栈栈顶元素加1）：

```
ICONST_1
IADD
```

下面我们具体来看JVM的运行时环境。



JVM执行某一方法的流程

- 读取此方法的代码
- 初始化方法的栈帧
  - 初始化操作数栈
  - 初始化局部变量表
  - 初始化PC为0
- 将栈帧push到 Thread Stack中
- **while**(下一条指令不是`return`指令){

  指令 := 从**PC**处读取一条指令

  PC := 下一条指令的起始地址

  指令.执行()

  }
- 从Thread Stack中pop 出此栈帧

啥是PC

PC 是 指令 在 当前正在执行的方法的 code中的 偏移量。 在之前对classfile解析的过程中大家可以发现，有一种attribute叫做“code”，code本身是一个字节的序列，它和方法是一一对应的。一个方法只有一段code，而一段code也只能对应一个方法。（一个例外是，有的方法，例如抽象方法，并没有对应的code。）当这个方法开始执行时，PC被设置为0，每读取一条指令之后，我们首先增加PC，使它指向下一条指令，然后再执行这条指令。这里值得关注的是，有一些指令需要修改PC的值，例如跳转/分支指令，**在实现这些指令的时候，请务必记住，PC的值已经指向下一条指令了。**

啥是局部变量表和操作数栈

局部变量表储存方法的局部变量，而操作数栈储存下一步操作要用到的操作数。对于习惯x86-64指令集的同学们来说，这个概念可能比较令人困惑。我们先来看一个简单的例子： 对于表达式`result = a+b+c`，一种在x86-64指令集下的计算方式是：

```
movl %eax, a
movl %ecx, b
addl %eax, %ecx
movl %ecx, c
addl %eax, %ecx
movl result, %eax
```

其中a, b, c, result均指这些符号对应的实际的内存地址。 在这里，我们先将a和b读入寄存器；然后进行相加；之后将c读入寄存器；然后和a+b的中间结果相加；然后将最终结果存入result. 而在JVM的栈式指令中，这段计算看起来可能是这样：

```
; operandStack:[]

iLOAD [index of a]

; operandStack:[a]

iLOAD [index of b]

; operandStack:[a, b]

iADD

; operandStack:[a+b]

iLOAD [index of c]

; operandStack:[a+b, c]

iADD

; operandStack:[a+b+c]

iSTORE [index of result]

; operandStack:[]
```

其中`index of SYMBOL`代表`SYMBOL`在局部变量表中的下标。分号开头的行是注释，里面标记了当前操作数栈里的内容。LOAD指令将变量从局部变量表读取到操作数栈中，STORE指令将操作数栈中的数据储存到对应的局部变量表中，ADD指令将两个操作数从操作数栈读出，然后把它们相加的结果写回操作数栈顶。

> 【和实验无关内容，可以跳过】操作数栈的存在一定程度上是为了提高运行效率：因为寄存器的读写速度要比内存快得多，所以将频繁访问的操作数放在寄存器里可以增加指令执行的效率。而不同平台的寄存器数量不一样，栈式指令集提供了很好的跨平台支持--无论在什么平台上，都可以尽可能多地将操作数栈映射到物理机器的寄存器上，从而提高指令执行的效率。

框架代码解析

关于runtime

OperandStack中，我们使用一个Slot的数组来模拟栈结构，其中成员变量`top`用来表示当前栈顶的空闲位置。这个top并不会指向一个具体的元素而是总是指向一个空位。

关于指令

对于任何一条指令，我们都实现了下面这样的接口：

```
public abstract class Instruction {
    public abstract void execute(StackFrame frame);

    public abstract void fetchOperands(ByteBuffer reader);
}
```

其中，`fetchOperands`方法是为了读取构造这条指令所需要的参数：例如LDC这条指令（参见英文版手册第八版第538页），这条指令用于将某个运行时常量池中的值读入操作数栈。它的第一个字节是操作数，之后紧跟着一个字节是`index`，它是运行时常量池对应项的下标。

`execute`方法是执行这条指令的接口，其中frame就是当前方法的栈帧。

系统中的解释器会用一个这样的循环来执行指令

```
while(true){
  //读取一条指令
  instruction = getInstruction(PC);
  //取指令的构造参数
  instruction.fetchOperands(reader);
  //更新PC的值
  updatePC();
  //执行指令
  instruction.execute(frame);
}
```

下面我们以FLOAD指令为例来说明如何从手册实现代码。

fload ***fload***

Operation Load `float` from local variable 指令的功能概述

Format

fload
index

指令的格式

Forms *fload* = 23 (0x17) 指令 opcode 的具体值

Operand Stack ... → ..., *value* 指令执行前后操作数栈的变化

Description The *index* is an unsigned byte that must be an index into the local variable array of the current frame (§2.6). The local variable at *index* must contain a `float`. The *value* of the local variable at *index* is pushed onto the operand stack. 指令的详细规格

Notes The *fload* opcode can be used in conjunction with the *wide* instruction (§*wide*) to access a local variable using a two-byte unsigned index. 其它内容

首先我们从指令格式中知道了这条指令包括一个字节的操作数和一个字节的`index`， **在实验的框架代码中，我们已经将`**index**`读取好了，它保存在指令的叫做`**index**`的成员中。** 然后从执行前后操作数栈的变化可以看出，这条指令向操作数栈push了一个新的元素。从详细规格中 可以看出，这条指令从局部变量表中读出第`index`个变量，然后把它的值压入操作数栈。 至此，我们可以得到下面这样的实现：

```
public void execute(StackFrame frame) {
    //从局部变量表读取对应元素
    float val = frame.getLocalVars().getFloat(this.index);
    //将这个元素压入操作数栈
    frame.getOperandStack().pushFloat(val);
}
```

太长不看版：

在这个实验中，所有需要实现的部分都已经用`**TODO**`标记出来了。其中包括对运行时环境的个别接口的实现和一些指令的实现。大家在实验之前首先通过各种渠道理解一下JVM的运行时环境，然后参照手册实现对应指令即可。

测试用例

本实验包括三个测试用例和一个加分项。加分项主要涉及了类型转换指令。 我们首先将几个简单的Java程序编译成了class文件，框架代码读取了这些class文件，并且找到它们的main方法开始执行。在执行的过程中，解释器会（像上面那个循环所写的一样）依次构造、执行指令。你可以使用javap命令来读取测试的class文件中的指令以便debug。

值得一提的是，测试用例中涉及到了几个`TestUtil`类的方法，这几个方法是在框架代码中通过某种方式来实现的。`TestUtil.fail()`会在被执行到时抛出`RuntimeException()`，`TestUtil.equalInt(a,b)`会判断a和b是否相等，如果不相等，则抛出`RuntimeException()`，`TestUtil.equalFloat(a,b)`同理，只是它是用来判断float型变量的。

在解释器运行时框架代码输出的信息仅供参考，不保证正确，请酌情使用。

testJmp

涉及指令

ifeq ifne goto

```
public static void testJmp(boolean a, boolean b, boolean c) {
    //ifeq
    if (a) {
        TestUtil.fail();
    } else {
        //ifne ifeq
        if (b || c) {
            //ifeq
            if (c) {
                //goto
                //return
            }else {
                TestUtil.fail();
            }
        } else {
            TestUtil.fail();
        }
    }

}


public static void main(String[] args) {
    testJmp(false, false, true);
}
```

branchTest

涉及指令

dcmpg dcmpl dload goto iadd iconst_1 iconst_3 if_icmpeq if_icmpge if_icmpgt if_icmple if_icmplt if_icmpne ifge ifle iload

```
public class ConditionTest {
```

```
    public static void test(int small, int big, long smallL, long bigL, float
smallF, float bigF,
                            double smallD, double bigD) {
        if (small == 3) {
            if (small < big && smallL < bigL && smallF < bigF && smallD < bigD) {

            } else {
                TestUtil.fail();
            }
            big++;
            if (big > small && bigL > smallL && bigF > smallF && bigD > smallD) {

            } else {
                TestUtil.fail();
            }
        } else {
            TestUtil.fail();
        }


        if (small <= big) {
            if (big > small) {

            } else {
                TestUtil.fail();
            }
            if (big + 1 >= small) {
                if (big == small) {
                    TestUtil.fail();
                }
                if (big != small) {

                } else {
                    TestUtil.fail();
                }
            }
        }
    }

    public static void main(String[] args) {
        test(3, 4, 5, 6, 7f, 8f, 9, 10);
    }
}
```

mathTest

涉及指令

goto iadd iconst_1 iconst_5 idiv if_icmpne iload istore imul isub

```
public class MathTest {

    public static void test(int a, int b) {
        int c = a + b;
```

```
        if (c == 11 && a - b == 1 && a * b == 30 && a / b == 1) {
            TestUtil.equalInt(a, 6);
            TestUtil.equalInt(b, 5);
        } else {
            TestUtil.fail();
        }

    }

    public static void main(String[] args) {
        test(6, 5);
    }
}
```

ConversionTest（加分项）

涉及指令：各种conversion

（bipush, d2f, d2i, d2l, dadd, dconst_0, dload_1, dload_3, f2i, f2l, fadd, fconst_0, fload, fload_0, fstore, goto, i2b, i2c, i2d, i2l, i2s, iconst_3, if_icmpeq, ifeq, iload, ineg, invokestatic, ishl, istore, l2d, l2f, l2i, ladd, lcmp, ldc, ldc2_w, lload, lstore, pop, return, sipush）看起来很多，但是别害怕，因为其中并不是所有指令都要求在本次作业中实现了，有不少指令框架代码已经实现好了(´▽`)/

```
public class ConversionTest {
    public static void test(float flt, double db, double bigDB, float bigFLT) {
        //d2f
        float a = (float) db;
        float b = flt;
        TestUtil.equalFloat(a, b);
        //d2i
        int c = (int) db;
        //f2i
        int d = (int) flt;
        TestUtil.equalInt(c, d);
        TestUtil.equalInt(c, 3);
        //d2i
        int max = (int) bigDB;
        TestUtil.equalInt(max, Integer.MAX_VALUE);
        //d2L
        long maxL = (long) bigDB;
        if (maxL != 2147483648L) {
            TestUtil.fail();
        }
        //f2i
        if (max != (int) bigFLT) {
            TestUtil.fail();
        }
        //f2l
        if (maxL != (long) bigFLT) {
            TestUtil.fail();
        }
        int toB;
        if (TestUtil.equalInt(c, c)) {
            toB = 128;
        } else {
```

```
            toB = 128;
        }
        //i2b
        byte bt = (byte) toB;
        TestUtil.equalInt(bt, -128);
        //i2c
        char ch = (char) toB;
        TestUtil.equalInt(ch, 128);
        //i2d
        TestUtil.equalInt((int) ((double) toB + 0.0), toB);
        //i2l & l2i
        TestUtil.equalInt((int) ((long) toB + ch + bt), toB);
        //l2d
        TestUtil.equalInt((int) ((double) ((long) toB + ch + bt) + 0.0), toB);
        //l2f
        TestUtil.equalInt((int) ((float) ((long) toB + ch + bt) + 0.0f), toB);
        toB <<= 8;
        //i2s
        short sh = (short) toB;
        TestUtil.equalInt(sh, -toB);
    }

    public static void main(String[] args) {
        test(3.99f, 3.99, 2147483648.0, 2147483648.0f);
    }
}
```