栈帧(Stack frame)

栈帧是一种帮助虚拟机方法调用与方法执行的数据结构。

栈帧本身是一种数据结构，封装了方法的局部变量表，动态链接信息，方法的返回地址以及操作数帧等信息。

可以通过slot 存储局部变量，slot 可复用的。

符号引用，直接引用

有些符号引用是在类加载阶段或是第一次使用就会转换为直接引用，这种转换叫做静态解析；

另一些符号则是在每次运行期转换为直接引用，这种转换叫做动态链接，这体现为Java的多态性。

invokeinterface:调用接口中的方法,实际上是在运行期决定的,决定到底调用实现接口的哪个对象的特定方法

invokestatic:调用静态方法

invokespecial:调用自己的私有方法，构造方法(<init>)以及父类的方法

invokevirtual:调用虚方法，运行期动态查找的方法

invokedynamic：动态调用方法

静态解析的4种情形:

1. 静态方法
2. 构造方法
3. 私有方法
4. 父类方法

以上四种方法被称为非虚方法,他们是在类加载阶段就可以将符号引用转换为直接引用的。

要点:

1. 静态分派:多态(静态类型，实际类型)，方法重载
2. 动态分派:虚方法表，接口方法表

方法的静态分派:

Grandpa g1 = new Father();

以上代码， g1的静态类型是Grandpa，而g1的实际类型(真正指向的类型)是Father.

我们可以得出这样一个结论:变量的静态类型是不会发生变化的，而变量的实际类型则是可以发生变 化的(多态的一种体现）实际变量是在运行期方可确定。而在字节码层面都是使用的变量的静态类型。

重载，是一个静态行为，在编译器就能确定

```
package main.java.Test1;
/**
 * Created By poplar on 2019/11/10
 * 静态解析的四种场：静态方法、父类方法、构造方法、私有方法。
 * 以上四种方法称为非虚方法，在类加载阶段将符号引用转换为直接引用。
 */

/**
 * 方法的静态分派。
 * Grandpa g1 = new Father();
 * 以上代码, g1的静态类型是Grandpa,而g1的实际类型(真正指向的类型)是Father.
 * 我们可以得出这样一个结论:变量的静态类型是不会发生变化的,而变量的实际类型则是可以发生变化的(多态的一种体现)
 * 实际变量是在运行期方可确定
 */
public class Invoke {

    public static void test(){
        System.out.println("invokestatic");
    }

    //方法重载,是一种静态行为,在编译器就能确定
    public static void test(Grandpa g){
        System.out.println("Grandpa");
    }

    public static void test(Father f){
        System.out.println("Father");
    }

    public static void test(Son s){
        System.out.println("Son");
    }

    public static void main(String[] args) {
        test();
```

```
        Grandpa g1=new Father();
        Grandpa g2=new Son();

        Invoke i=new Invoke();
        i.test(g1);  Grandpa
        i.test(g2);  Grandpa
    }
}

class Grandpa{
}

class Father extends Grandpa{
}

class Son extends Father{
}
```

方法的动态分派:

方法的动态分派涉及到一个重要概念:方法接收者。

invokevirtual字节码指令的多态查找流程

比较方法重载(overload)与方法重写(overwrite) ,我们可以得到这样一个结论:

方法重载是静态的,是编译期行为;

方法重写是动态的,是运行期行为;

尽管在字节码层面，调用的静态类型的方法。但是此刻并没有将符号引用化为直接

引用，而是在运行期进行动态解析，实际调用的实际类型的方法。

针对于方法调用动态分派的过程，虚拟机会在，类的方法区建立一各虚方法表的数据结构

针对invokeinterface指令来说，虚拟机会建立一个叫做接口方法表的数据结构

```
package main.java.Test1;
```

```
public class Invoke2 {
    public static void main(String[] args) {
        Fruit apple=new Apple();
        Fruit orange=new Orange();
        apple.test();  Apple
        orange.test();  Apple

        apple=new Orange();
        apple.test();  Orange
    }
}

class Fruit{
    public void test(){
        System.out.println("Fruit");
    }
}

class Apple extends  Fruit{
    @Override
    public void test() {
        System.out.println("Apple");
    }
}

class Orange extends Fruit{
    @Override
    public void test() {
        System.out.println("Orange");
    }
}
```

```
 0: new           #2                  // class main/java/Test1/Apple
         3: dup
         4: invokespecial #3                 // Method main/java/Test1/Apple."<init>":()V
         7: astore_1
         8: new           #4                 // class main/java/Test1/Orange
        11: dup
        12: invokespecial #5                 // Method main/java/Test1/Orange."<init>":()V
        15: astore_2
        16: aload_1
        17: invokevirtual #6                 // Method main/java/Test1/Fruit.test:()V
        20: aload_2
        21: invokevirtual #6                 // Method main/java/Test1/Fruit.test:()V
        24: new           #4                 // class main/java/Test1/Orange
        27: dup
        28: invokespecial #5                 // Method main/java/Test1/Orange."<init>":()V
```

31: astore_1
32: aload_1
33: invokevirtual #6 // Method main/java/Test1/Fruit.test:()V
36: return

指令集与寄存器的指令

现代JVM在执行Java代码的时候，通过都会讲解释执行与编译执行二者结合起来运行。

解释执行：通过解释器来读取字节码，遇到相应的指令就去执行该指令。

编译执行：就是通过即时编译器（Just In time，JIT）将字节码转换为本地机器码来执

行；现代JVM会根据代码热点来生成响应的本地机器码.

基于栈的指令集与基于寄存器的指令集之间的关系；

a. JVM 执行指令时所采取的方式是基于栈的指令集.

b. 基于栈的指令集主要是操作有入栈与出栈两种.

c. 基于栈的指令集的优势在于它可以在不同平台之间转移，而基于寄存器的指令

集就是与硬件架构紧密关联，无法做到可移植。

d. 基于栈的指令集的缺点在于完成相同的操作，指令数量通常要比基于寄存器的

指令集数量要多，基于栈的指令集是在内存中完成的操作的。 而基于寄存器

的指令集是直接由CPU来执行的，它是在高速缓冲区执行的， 速度要快很多，

虽然虚拟机可以采用一些优化手段，但是整体来说基于栈的指令集的执行速

度要慢一些。

例子:2-1

1. iconst_1 //将减数1压入栈顶.
2. iconst_2 //将减数2压入栈顶.

3. isub //将栈顶以及下面的弹出对于响应的数字执行减法2-1,然后将结果压入栈顶.
4. istore_0 //将1放入局部变量表0的位置上.