解析

1.invokeinterface//调用接口方法
2.invokestatic//调用静态方法
3.invokespecial//调用自己的私有方法、构造方法<init>、父类的方法
4.invokevirtual//调用虚方法，运行期动态查找
5.invokedynamic//动态调用方法

能被 invokestatic 和 invokespecial 调用的方法，在解析时就能确定唯一的调用版本。符合这个条件的有：

1. 父类方法
2. 静态方法
3. 实例构造器
4. 私有方法

以上4类方法称为非虚方法。在类加载阶段，会把符号引用转化为直接引用。
Java中的非虚方法除了使用invokestatic 和 invokespecial调用的方法之外，还有被final修饰的方法也是非虚方法。

类加载过程的解析阶段，就是把能在解析阶段确定唯一调用版本的方法的符号引用转为直接引用。

静态分派

依赖静态类型来定位方法执行版面的分派动作成为静态分派。典型应用就是方法重载。静态分派的动作发生在编译阶段。静态分派发生在编译阶段。

```
Human man = new Man();
```

Human是静态类型，Man是时机类型。
静态类型在编译期可知，实际类型在运行期可知。
变量的静态类型是不会发生改变的，而变量实际类型是可以改变的。
重载是通过参数的静态类型作为判断依据的，静态类型决定使用哪个重载版本。

```
//Grandpa 为 p1、p2 的静态类型
Grandpa p1 = new Father();
Grandpa p2 = new Son();

obj.test(p1);//Grandpa
obj.test(p2);//Grandpa

public class Test{
    public void test(Grandpa p){
```

```
        sout("Grandpa");
    }
    public void test(Father f){
        sout("Father");
    }
    public void test(Son s){
        sout("Son");
    }
}
```

动态分派

方法重写时动态的，是运行期行为。

```
class Apple extends Fruit {...}
class Orange extends Fruit {...}

Fruit apple = new Apple();
Fruit orange = new Orange();

//字节码 invokevirtual
apple.test();//apple
orange.test();//orange
```

invokevirtual 执行步骤

1. 找到操作数栈顶的第一个元素。
2. 根据这个元素的实际类型，去找调用的方法。
3. 如果找不到这个方法，按继承的层次关系往上查找。
4. 如果一直都找不到，则会抛异常。

```
16 aload_1
17 invokevirtual #6 <Fruit.test> 对应 apple.test();
20 aload_2
21 invokevirtual #6 <Fruit.test> 对应 orange.test();
```

可以看出，即便是字节码中相同的符号引用，在运行期也会被解析为不同的直接引用。这也是Java中方法多态性的重要表现。

虚方法表

是运行期的概念，标识的是方法的实际入口调用地址。

子类没有重写的来自父类的方法，会存入到虚方法表中。比如 Object 类中的 hashCode、notify 等方法。

1. 子类的虚方法表中对应的方法会直接指向父类方法的地址

2. 虚方法表中的方法在子类和父类中的索引都是一样的，这样也会提高查找速度。

单分派多分派

```
public class Dispatcher {
    public static class Father {
        public void hardChoice(QQ arg) {
            System.out.println("father choose QQ");
        }
        public void hardChoice(_360 arg) {
            System.out.println("father choose _360");
        }
    }
    public static class Son extends Father {
        @Override
        public void hardChoice(QQ arg) {
            System.out.println("son choose QQ");
        }
        @Override
        public void hardChoice(_360 arg) {
            System.out.println("son choose 360");
        }
    }

    public static void main(String[] args) {
        Father father = new Father();
        Father son = new Son();
        father.hardChoice(new _360());
        son.hardChoice(new QQ());
    }
}

father.hardChoice(new _360());
son.hardChoice(new QQ());

24:  invokevirtual   #8; //Method Dispatcher$Father.hardChoice:
(LDispatcher$_360;)V
35:  invokevirtual   #11; //Method Dispatcher$Father.hardChoice:
(LDispatcher$QQ;)V
```

首先确定方法的接收者，发现两个对象变量的静态类型都是Father类型的，因此在class文件中写的Father类中方法的符号引用。再者，对于方法参数，一个是_360对象，一个是QQ对象，按照静态类型匹配的原则，自然找到各自的方法。

上面的两步都是在编译器中做出的，属于静态分派，在选择目标方法时根据了两个宗量，是多分派的。因此，静态分派属于多分派类型。

当java执行时，当执行到son.hardChoice(new QQ())；时，发现son的实际类型是Son，因此会调用Son类中的方法。在执行father.hardChoice(new _360())；时也有这个过程，只不过father的实际类型就是Father而已。发现，在目标选择时只依据了一个宗量，是单分派的。因此，动态分派属于单

基于栈的指令集

```
public int test1(){
    int a = 1;
    int b = 2;
    int c = 3;
    int d = 4;
    int num = (a + b - c) * d;
    return num;
}
  0 iconst_1
 1 istore_1
 2 iconst_2
 3 istore_2
 4 iconst_3
 5 istore_3
 6 iconst_4
 7 istore 4
 9 iload_1
10 iload_2
11 iadd
12 iload_3
13 isub
14 iload 4
16 imul
17 istore 5
19 iload 5
21 ireturn
```

- **iconst：push到 operand stack**

- **istore：弹出 operand stack top 元素，并把这个值设置到局部变量表的对应位置**
- **iadd： 弹出 operand stack 顶的两个元素，相加后的结果重新push到operand stack**
- **iload：The value of the local variable at index is pushed onto the operand stack.把局部变量表对应位置处的value压到操作数栈中。**
- **ireturn: 弹出当前操作数栈的值，并压入调用者的操作数栈中。当前方法的操作数栈中的值会被丢弃掉。**

语法糖

==运算符在不遇到算术运算的情况下不会自动拆箱。

```
public class JVMTEST {
    public static void main(String[] args) {
        Integer a = 1;
        Integer b = 2;
        Integer c = 3;
        Integer d = 3;
        Integer e = 321;
        Integer f = 321;
        Long g = 3L;

        System.out.println(c == d);//t
        System.out.println(e == f);//f
        System.out.println(c == (a+b));//t
        System.out.println(c.equals(a+b));//t
        System.out.println(g == (a+b));//t
        System.out.println(g.equals(a+b));//f
    }
}
```

反编译之后可以看到

```
public class JVMTEST
{
  public static void main(String[] args) {
    Integer a = Integer.valueOf(1);
    Integer b = Integer.valueOf(2);
    Integer c = Integer.valueOf(3);
    Integer d = Integer.valueOf(3);
    Integer e = Integer.valueOf(321);
    Integer f = Integer.valueOf(321);
    Long g = Long.valueOf(3L);

    System.out.println((c == d));
```

```
    System.out.println((e == f));
    System.out.println((c.intValue() == a.intValue() + b.intValue()));
    System.out.println(c.equals(Integer.valueOf(a.intValue() + b.intValue())));
    System.out.println((g.longValue() == (a.intValue() + b.intValue())));
    System.out.println(g.equals(Integer.valueOf(a.intValue() + b.intValue())));
  }
}
```