方法调用：不等同于方法执行，方法调用阶段的唯一目标就是确定被调用方法的版本（即调用哪一个方法），暂时还不设计到方法内部的具体运行过程。一切方法调用在Class文件中都存储的都是符号引用，而不是方法在实际运行时内存布局中的入口地址。

解析：
所有方法调用中的目标方法在Class文件中都是一个常量池的符号引用，在类加载的解析阶段，会将其中的一部分符号引用转化为直接引用，这种解析的前提是：**方法在程序真正运行之前就有一个可确定的调用版本，并且这个方法的调用版本在运行期是不可变的。**在java语言中符号“编译器可知，运行期不可变”这个要求，主要包括静态方法和私有方法两大类，前者与类型直接关联，后者在外部不可被访问，这两个方法各自的特点决定了他们都不可能通过集成或者别的方式重写其他版本。因此他们适合在类加载阶段进行解析。

与之对应的，在java虚拟机中提供了5条方法调用字节码指令：
1.invokestatic：调用静态方法
2.invokespecial：调用实例构造器<init>方法、私有方法和父类方法
3.invokevirtual：调用所有的虚方法
4.invokeinterface：调用接口方法、会在运行时再确定一个实现此接口的对象
5.invokedynamic：现在运行时动态解析出调用点限定符所引用的方法，然后再执行该方法。前四条调用指令，分派逻辑都固话在java虚拟机内部的，而invokedynamic指令的分派逻辑是由用户所设定的引导方法决定的。

只有被Invokestatic和invokeSpecial指令调用的方法，都可以在解析阶段中确定唯一 的调用版本，符合这个条件的有静态方法、私有方法、实例构造器、父类方法四种，它们在类加载的时候就会把符号引用解析为该方法的直接引用。这些方法可以成为非虚方法，与之相反的是虚方法（除去final方法）。java中非虚方法除了invokestatic和invokespecial之外还有一种方式就是被final修饰的方法，因为final方法没法被覆盖。

解析调用是一个静态的过程，在编译期间完全确定，在类装载的解析阶段中就会把涉及到符号引用全部转变称为可确定的直接引用，不会延迟到运行期再去完成。而分派调用则可能是动态的，可能是静态的，根据分派依据可以分为单分派

和多分派，这两种分派方式两两组合构成了静态单分派，静态多分派，动态单分派，动态多分派四种情况。

分派：

1.静态分派

所有依赖静态类型来定位方法执行版本的分派动作，都称为静态分派，静态分派的最典型应用就是多太中的方法重载。静态分派发生在编译阶段，因此静态分配的动作实际上不是有虚拟机来执行的。

```
class Human {}

class Man extends Human {}

class Woman extends Human {}

class Invoker {

    public void sayHello(Human human) {
        System.out.println("Hello. Human");
    }

    public void sayHello(Man man) {
        System.out.println("Hello. Man");
    }

    public void sayHello(Woman woman) {
        System.out.println("Hello. Woman");
    }

    public static void main(String[] args) {
        Human man = new Man();
        Human woman = new Woman();

        Invoker invoker = new Invoker();
```

```
        invoker.sayHello(man);
        invoker.sayHello(woman);

        invoker.sayHello((Man) man);

        invoker.sayHello((Woman) woman);
    }
}
```

执行结果:
Hello. Human
Hello. Human
Hello. Man
Hello. Woman
我们把上面的代码中的Human称为变量的静态类型，Man称为变量的实际类型。静态类型和实际类型在程序中都可以发生一些变化，区别是静态类型的变化仅仅在使用时发生，变量本身的静态类型不会改变，并且最终的静态类型是编译期克制的，而实际类型变化的结果在运行期才可确定。

2.动态分派
动态分派与多态性的另一重要体现----方法覆写有这很紧密的关系，向上转型后调用子类覆写的方法便是一个很好说明动态分配的例子。java动态分派主要体现在“重写”上。所以我们把这种在运行期根据实际类型确定方法执行版本的分派称为动态分派

虚拟机动态分派的实现：
为类在方法区中建立一个虚方法表（Virtual method table, 也称为Vtable，与之对应的在invokeinterface执行时会用到接口方法表-Interface method table,简称itable），使用虚方法表索引来代替元数据查找已提高性能。虚方法表中存放了各个方法的实际入口地址，如果某方法在子类中没有被重写，那么子类的虚方法表中的地址入口和父类相同方法的地址入口是一致的。如果重写了该方法，子类方法表中的地址将会替换为指向子类版本的入口地址。