前面我们看到过几个关于方法调用的指令了。比如上篇文章有讲到的对象实例初始化<init>函数由 invokespecial 调用。这篇文章我们将介绍关于方法调用的五个指令：

- invokestatic：用于调用静态方法
- invokespecial：用于调用私有实例方法、构造器，以及使用 super 关键字调用父类的实例方法或构造器，和所实现接口的默认方法
- invokevirtual：用于调用非私有实例方法
- invokeinterface：用于调用接口方法
- invokedynamic：用于调用动态方法

0x01 方法的静态绑定与动态绑定

要理解为什么需要上面 5 种方法调用，需要先弄清楚 Java 的两种方法绑定方式：静态绑定与动态绑定。
在编译时时能确定目标方法叫做**静态绑定**，相反地，需要在运行时根据调用者的类型动态识别的叫**动态绑定**。

invokestatic 和 invokespecial 这两个指令对应的方法是静态绑定的，invokestatic 调用的是类的静态方法，在编译期间确定，运行期不会修改。剩下的三个都属于动态绑定，下面进行一一介绍。

0x02 invokestatic

invokestatic 用来调用静态方法，即使用 static 关键字修饰的方法。

它要调用的方法在编译期间确定，运行期不会修改，属于静态绑定。它也是所有方法调用指令里面最快的。比如 Integer.valueOf("42")对应字节码

```
0: ldc           #2                  // String 42
2: invokestatic  #3                  // Method
java/lang/Integer.valueOf:
(Ljava/lang/String;)Ljava/lang/Integer;
5: pop
```

0x03 invokevirtual vs invokespecial 既生瑜何生亮

- invokevirtual：用来调用 public、protected、package 访问级别的方法
- invokespecial：顾名思义，它是「特殊」的方法，包括实例构造方法、私有方法（private 修饰的方法）和父类方法（即 super 关键字调用的方法）。很明显明显，这些「特殊」的方法可以直接确定实际执行的方法的实现，与 invokestatic 一样，也属于静态绑定

在 JDK 1.0.2 之前，invokespecial 指令曾被命名为 invokenonvirtual，以区别于 invokevirtual

看到这里，你有没有想过为什么有了 invokevirtual 还需要 invokespecial 的存在呢?

其实 java 虚拟机规范里面有比较详细的介绍

> The difference between the invokespecial and the invokevirtual instructions is that invokevirtual invokes a method based on the class of the object. The invokespecial instruction is used to invoke instance initialization methods as well as private methods and methods of a superclass of the current class.

- invokespecial 用在在类加载时就能确定需要调用的具体方法，而不需要等到运行时去根据实际的对象值去调用该对象的方法。private 方法不会因为继承被覆写的，所以 private 方法归为了 invokespecial 这一类。
- invokevirtual 用在方法要根据对象类型不同动态选择的情况，在编译期不确定。

举一个实际的例子

```
public class Color {
    public void printColorName()  {
        System.out.println("Color name from
parent");
    }
}
public class Red extends Color {
    @Override
    public void printColorName() {
        System.out.println("Color name is Red");
    }
}
public class Yellow extends Color {
    @Override
    public void printColorName() {
        System.out.println("Color name is
Yellow");
    }
}
public class InvokeVirtualTest {
    private static Color yellowColor = new
Yellow();
    private static Color redColor = new Red();
    public static void main(String[] args) {
        yellowColor.printColorName();
        redColor.printColorName();
    }
}

输出
Color name is Yellow
Color name is Red
```

我们来看一下 main 函数的字节码

```
0: getstatic     #2                  // Field
yellowColor:LColor;
3: invokevirtual #3                  // Method
Color.printColorName:()V
6: getstatic     #4                  // Field
redColor:LColor;
9: invokevirtual #3                  // Method
Color.printColorName:()V
```

可以看到 3 和 9 行指令完全一样，都是Color.printColorName，并没有被编译器改写为Yellow.printColorName和Red.printColorName。它们最终调用的目标方法却不同，invokevirtual 会根据对象的实际类型进行分派（虚方法分派），在编译期间不能确定最终会调用子类还是父类的方法。

0x04 invokeinterface vs invokevirtual 孪生兄弟大不同

invokeinterface 用于调用接口方法，在运行时再确定一个实现此接口的对象。
那它跟 invokevirtual 有什么区别呢？为什么不用 invokevirtual 来实现接口方法的调用？其实也不是不可以，只是为了效率上的考量。

invokestatic 指令需要调用的方法只属于某个特定的类，在编译期唯一确定，不会运行时动态变化，是最快的
invokespecial 指令可能调用的方法也在编译期确定，且只有少数几个需要处理的方法，查找也非常快
invokevirtual 和 invokeinterface 的关系就比较微妙了，区别没有

那么明显，我们用一个实际的例子来说明，可以这么认为，每个类文件都关联着一个「虚方法表」（virtual method table），这个表中包含了父类的方法和自己扩展的方法。比如

```
class A {
    public void method1() { }
    public void method2() { }
    public void method3() { }
}

class B extends A {
    public void method2() { } // overridden from
BaseClass
    public void method4() { }
}
```

对应的虚方法表如下：

现在 B 类的虚方法表保留了父类 A 中方法的顺序，只是覆盖了 method2() 指向的函数链接和新增了method4()。
假设这时需要调用 method2 方法，invokevirtual 只需要直接去找虚方法表位置为 2 的地方的函数引用就可以了

如果是用 invokeinterface，这样的优化是没法起作用的，比如，我们改一下让 B 实现 X 接口

```
interface X {
    void methodX()
}
class B extends A implements X {
    public void method2() { } // overridden from
BaseClass
    public void method4() { }
    public void methodX() { }
}
Class C implements X {
    public void methodC() { }
    public void methodX() { }
}
```

这样的情况下虚方法表如下

这种情况下，B 类的 methodX 在位置 5 的地方，C 类的 methodX 在位置 2 的地方，如果要用 invokevirtual 调用 methodX 就不能直接从固定的虚方法表索引位置拿到对应的方法链接。
invokeinterface 不得不搜索整个虚方法表来找到对应方法，效率上远不如 invokevirtual

0x05 动态方法调用秘密武器 invokedynamic

invokedynamic 是五种 invoke 里面最复杂的，下一篇文章将专门介绍 invokedynamic 的概念

0x06 小结

这篇文章讲解了方法调用的 5 个指令，一起来回顾一下要点：

- 第一，介绍了动态绑定和静态绑定的区别。
- 第二，介绍了 invokestatic、invokevirtual、invokespecial、invokeinterface 四个指令背后深层次效率上的考量。

0x07 思考

最后，给你留两道思考题

1. invokestatic、invokevirtual、invokespecial、invokeinterface 这四个指令调用效率的排序是怎么样的?
2. JDK8 的 lambda 表达式为什么采用 invokedynamic 来实现？跟匿名内部类的方式相比有哪些优点?

欢迎你在留言区留言，和我一起讨论。