**多态的实现**

下面从虚拟机运行时的角度来简要介绍多态的实现原理，这里以Java虚拟机（[**Java Virtual Machine**](http://en.wikipedia.org/wiki/Java_Virtual_Machine), JVM）规范的实现为例。

在JVM执行Java字节码时，**类型信息被存放在方法区中**，通常为了优化对象调用方法的速度，方法区的类型信息中增加一个指针，该指针指向一张记录该类方法入口的表（称为方法表），**表中的每一项都是指向相应方法的指针。**

方法表的构造如下：

由于Java的单继承机制，一个类只能继承一个父类，而所有的类又都继承自Object类。方法表中**最先存放的是Object类的方法，接下来是该类的父类的方法，最后是该类本身的方法。**这里关键的地方在于，**如果子类改写了父类的方法，那么子类和父类的那些同名方法共享一个方法表项，都被认作是父类的方法。**

注意这里只有非私有的实例方法才会出现，并且静态方法也不会出现在这里，原因很容易理解：静态方法跟对象无关，可以将方法地址直接引用，而不像实例方法需要间接引用。

更深入地讲，静态方法是由虚拟机指令invokestatic调用的，私有方法和构造函数则是由invokespecial指令调用，只有被invokevirtual和invokeinterface指令调用的方法才会在方法表中出现。

由于以上方法的排列特性（Object——父类——子类），使得**方法表的偏移量总是固定的**。例如，对于任何类来说，其方法表中equals方法的偏移量总是一个定值，所有继承某父类的子类的方法表中，其父类所定义的方法的偏移量也总是一个定值。

前面说过，方法表中的表项都是指向该类对应方法的指针，这里就开始了多态的实现：

假设Class A是Class B的子类，并且A改写了B的方法method()，那么在B的方法表中，method方法的指针指向的就是B的method方法入口。

而对于A来说，它的方法表中的method方法则会指向其自身的method方法而非其父类的（这在类加载器载入该类时已经保证，同时JVM会保证总是能从对象引用指向正确的类型信息）。

结合**方法指针偏移量是固定的**以及**指针总是指向实际类的方法域**，我们不难发现多态的机制就在这里：

在调用方法时，实际上必须首先完成实例方法的符号引用解析，结果是该符号引用被解析为方法表的偏移量。虚拟机通过对象引用得到方法区中类型信息的入口，查询类的方法表，当将子类对象声明为父类类型时，形式上调用的是父类方法，此时虚拟机会从实际类的方法表（虽然声明的是父类，但是实际上这里的类型信息中存放的是子类的信息）中查找该方法名对应的指针（这里用“查找”实际上是不合适的，前面提到过，方法的偏移量是固定的，所以只需根据偏移量就能获得指针），进而就能指向实际类的方法了。

我们的故事还没有结束，事实上上面的过程仅仅是利用继承实现多态的内部机制，多态的另外一种实现方式：实现接口相比而言就更加复杂，原因在于，**Java的单继承保证了类的线性关系，而接口可以同时实现多个，这样光凭偏移量就很难准确获得方法的指针。**所以在JVM中，多态的实例方法调用实际上有两种指令：

**invokevirtual指令用于调用声明为类的方法；**

**invokeinterface指令用于调用声明为接口的方法。**

当使用invokeinterface指令调用方法时，就不能采用固定偏移量的办法，只能老老实实挨个找了（当然实际实现并不一定如此，JVM规范并没有规定究竟如何实现这种查找，不同的JVM实现可以有不同的优化算法来提高搜索效率）。我们不难看出，**在性能上，调用接口引用的方法通常总是比调用类的引用的方法要慢。**这也告诉我们，在类和接口之间优先选择接口作为设计并不总是正确的，当然设计问题不在本文探讨的范围之内，但显然具体问题具体分析仍然不失为更好的选择。