重载、重写在 JVM层面的原理【静态分派与动态分派】

前言

1、方法调用概念

方法调用阶段**唯一的任务**就是**确定被调用方法的版本 （即调用哪一个方法）**，暂时还未涉及方法内部的具体运行过程，因此并不等同于方法中的代码被执行。

所有方法调用的目标方法在Class文件里面都是一个常量池中的符号引用。

2、方法调用分类

- **解析**

发生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。

解析的前提：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 可改变的。

即：调用目标在程序代码写好、编译器进行编译那一刻就已经确定下来。这类方法 的调用被称为解析（Resolution）

发生在解析阶段的对象

主要有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，这两种方法各自的特点决定了它们都不可能通 过继承或别的方式重写出其他版本，因此它们都适合在类加载阶段进行解析。

或者说

“非虚方法”（Non-Virtual Method）：在类加载的时候就可以把符号引 用解析为该方法的直接引用。有**静态方法、私有方法、实例构造器、父类方法4种，再加上被final 修饰的方法**（尽管它使用invokevirtual指令调用），**这5种方法**都可以在解析阶段中确定唯一的调用版本。

- **分派**

分派调用过程将会揭示多态性特征的一些最基本的体现，分为两种：

1. **静态分派**
2. **动态分派**

编译阶段中编译器的选择过程，也就是静态分派的过程；

运行阶段中虚拟机的选择，也就是动态分派的过程。

重载——静态分派 & 重写——动态分派

这里所谓的**分派**指的是在Java中对方法的调用。Java中有三大特性：封装、继承和多态。**分派是多态性的体现**，Java虚拟机底层提供了我们开发中“重写”和“重载”的底层实现。其中重载属于静态分派，而重写则是动态分派的过程。

1、重载——静态分派

静态分派只会涉及重载，而重载是在编译期间确定的，那么静态分派自然是一个静态的过程（因为还没有涉及到Java虚拟机）。静态分派的最直接的解释是**在重载的时候是通过参数的静态类型而不是实际类型作为判断依据的。**

代码清单8-6　方法静态分派演示

```
package org.fenixsoft.polymorphic;

/**
 * 方法静态分派演示
 * @author zzm
 */
public class StaticDispatch {

    static abstract class Human {
    }

    static class Man extends Human {
    }

    static class Woman extends Human {
    }

    public void sayHello(Human guy) {
        System.out.println("hello,guy!");
    }

    public void sayHello(Man guy) {
        System.out.println("hello,gentleman!");
    }

    public void sayHello(Woman guy) {
        System.out.println("hello,lady!");
    }

    public static void main(String[] args) {
        Human man = new Man();
        Human woman = new Woman();
        StaticDispatch sr = new StaticDispatch();
        sr.sayHello(man);
        sr.sayHello(woman);
    }
```

运行结果：

```
hello,guy!
hello,guy!
```

> **静态类型与实际类型**

> 上面代码中的“Human”称为变量的“静态类型”（Static Type），或者叫“外观类 型”（Apparent Type），后面的“Man”则被称为变量的“实际类型”（Actual Type）或者叫“运行时类 型”（Runtime Type）。
>
> **区别**
>
> 1、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被改变，并且最终的静态类型是在编译期可知的；
>
> 2、实际类 型变化的结果在运行期才可确定，编译器在编译程序的时候并不知道一个对象的实际类型是什么。

```
// 实际类型变化
Human human = (new Random()).nextBoolean() ? new Man() : new Woman();

// 静态类型变化
sr.sayHello((Man) human)
sr.sayHello((Woman) human)
```

> 对象human的实际类型是可变的，编译期间它完全是个“薛定谔的人”，到底是Man还是Woman，必须等到程序运行到这行的时候才能确定。而在编译期，human的静态类型是Human

代码中故意定义了两个静态类型相同，而实际类型不同的变量，但虚拟机（或者准确地说是编译器）在重载时是通过参数的静态类型而不是实际类型作为 判定依据的。由于静态类型在编译期可知，所以在编译阶段，Javac编译器就根据参数的静态类型决定 了会使用哪个重载版本，因此选择了sayHello(Human)作为调用目标。

2、重写——动态分派

Java虚拟机是如何在程序运行期间确定方法的执行版本的呢?

牵涉到 invokevirtual 指令本身，

根据《Java虚拟机规范》，invokevirtual指令的运行时解析过程大致分为以下几步：

1）找到操作数栈顶的第一个元素所指向的对象的实际类型，记作C。

2）如果在类型C中找到与常量中的描述符和简单名称都相符的方法，则进行访问权限校验，如果 通过则返回这个方法的直接引用，查找过程结束；不通过则返回java.lang.IllegalAccessError异常【非法访问异常】。

3）否则，按照继承关系从下往上依次对C的各个父类进行第二步的搜索和验证过程。

4）如果始终没有找到合适的方法，则抛出java.lang.AbstractMethodError异常【抽象方法错误的异常】。

代码清单8-8　方法动态分派演示

```
package org.fenixsoft.polymorphic;

/**
 * 方法动态分派演示
 * @author zzm
 */
public class DynamicDispatch {

    static abstract class Human {
        protected abstract void sayHello();
    }

    static class Man extends Human {
        @Override
        protected void sayHello() {
            System.out.println("man say hello");
        }
    }

    static class Woman extends Human {
        @Override
        protected void sayHello() {
            System.out.println("woman say hello");
        }
    }

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

运行结果：

```
man say hello
woman say hello
woman say hello
```

3.虚拟机动态分派的实现

由于动态分派是非常频繁的操作，实际实现中不可能真正如此实现。Java虚拟机是通过"稳定优化"的手段——**在方法区中建立一个虚方法表（Virtual Method Table）**，通过使用方法表的索引来代替元数据查找以提高性能。虚方法表中存放着各个方法的实际入口地址（由于Java虚拟机自己建立并维护的方法表，所以没有必要使用符号引用，那不是跟自己过不去嘛），如果子类没有覆盖父类的方法，那么子类的虚方法表里面的地址入口与父类是一致的；如果重写父类的方法，那么子类的方法表的地址将会替换为子类实现版本的地址。

方法表是在类加载的连接阶段（验证、准备、解析）进行初始化，准备了子类的初始化值后，虚拟机会把该类的虚方法表也进行初始化。

参考：

《深入理解Java虚拟机_第三版》 第8.3章节

(https://developer.aliyun.com/article/14517)