Java指令确定方法入口的细致分析

引言

当我们日常交流中，讨论到Java执行哪个方法的时候，实际上有两层含义，第一层含义是指确定方法的常量池引用。举个例子来说：

```
    public void test(Human human){
        System.out.println("this is a human");
    }

    public void test(Man man){
        System.out.println("this is a man");
    }

```

当我们如下调用的时候：

```
   MethodResolve methodResolve=new MethodResolve
   Human human=new Man();
   methodResolve.test(human);

```

我们很容易就确认，这里面调用将会是

```
    public void test(Human human){
        System.out.println("this is a human");
    }

```

这就是所谓的确定方法调用的常量池引用。这个过程是在编译期确定的。这里举的例子也是比较特殊的，它恰好是一个重载的例子。很多的文章教程会过分强调重载的特性，但是如果从“编译器是根据方法签名唯一确定一个Java方法”，那么重载就是便于人类认知的一种分类而已。在编译的时候，重载的方法和普通的方法并没有什么不同。
第二层含义是确定方法的入口。举个例子来说：

```
public class MethodResolveFather {
    public void test(Human human){
        System.out.println("this is a human fro
    }
}

public class MethodResolve extends MethodResolv
    public void test(Human human){
        System.out.println("this is a human fro
    }
}

```

如果调用是这样的：

```
MethodResolveFather methodResolve=new MethodReso
Human human=new Man();
methodResolve.test(human);

```

我们可以确定的是，最终的输出结果是”this is a human from MethodResolve”。
这篇文章要讨论的就是第二种含义下，Java虚拟机如何确定一个方法的“版本”，或者说，如何确定一个方法的入口。众所周知的是，在解析方法的常量池引用的时机有两个，第一个是在加载期，第二个时机是在指令执行的时候。因此本文探讨的主题也可以是“什么决定了一个方法的常量池引用会在加载期间被解析？”。或者说在虚拟机的层面上，重写

究竟是怎样的一种机制，为何有些方法能够被重写，而有些方法不能够被重写。也可以理解成，虚拟机究竟是如何支持重写的？

不同指令的比较

这里主要讨论Java方法调用指令中的四种：

- invokevirtual: 调用实例方法
- invokeinterface: 调用接口方法
- invokespecial: 调用一些特殊方法，如实例初始化方法，私有方法和父类方法
- invokestatic: 调用静态方法

这里先从invokespecial开始。invokespecial指令在Java虚拟机规范（java8的）里面描述该指令用于调用初始化的方法，如<init>和<cinit>方法，还有私有方法。前面的两个方法<init>和&ltcinit&gt都是编译器编译的时候生成的，不受Java语言开发者的控制（这里是指你无法写出叫这两个名字的方法，并且能够通过编译）。而私有方法，有一个显著的特性，它不能被重写。这些方法的符号引用能够在加载期被解析是很容易想通的，这个方法只可能有这么一个“版本”。我们要额外考虑一下调用父类方法的情况。例如：

```
1  public class MethodResolve extends MethodResolve
2
3      public void test(Human human){
4          super.test(human);
5          System.out.println("this is a human from
6      }
7  }
8
```

显然，当我们执行子类的test方法的时候，执行到super.test的时候，它调用的就会是父类的test方法——即便子类其实已经重写了test方法。这和super的语义是一致的，当使用super关键字调用的时候，其实就明确告诉了虚拟机：“在确定这一次方法调用的时候，只需要沿着继承树向上找就可以了”。而依据Java的加载模型，一个类的祖先类会先于该类被加载——这意味着父类中的方法的入口是确定下来的了。所以当使用super调用的时候，被调用的方法的符号引用也会在加载时候完成解析。
这里有一个有意思的话题，就是为什么使用this关键字调用的方法却不具备这样的特性？抽象地来说，这是因为this含义不明。举个例子：

```
1  public class MethodResolveFather {
2
3      public void test(Human human){
4          System.out.println("this is a human fro
5      }
6
7      public void testThis(){
8          System.out.println("ready……");
9          this.test(new Human());
10     }
11 }
12
```

假如我们的调用是这样的:

```
1      public void testLLL(MethodResolveFather fath
2          father.testThis();
3      }
4
```

这个时候，稍微注意一下就会发现，虽然方法接收的参数类型是MethodResolveFather，但是实际上，并不知道它的实际类型是什么。试想一下，如果加载MethodResolveFather类的时候要解析这个testThis里面的this.test的符号引用，如果这个时候完成解析，也就是说把这个test方法的版本指向了MethodResolveFather的版本，那么很显然，并不符号Java语

言的规范。
因为如果有这样一个子类

```
public class MethodResolve extends MethodResolve

    public void test(Human human){
        System.out.println("this is a human from
    }
}

```

该子类覆盖了父类中的test方法。而如果实际上testLLL这个方法被调用的时候传入的参数的实际类型是MethodResolve，那么按照Java语言的规范，此刻执行testLLL方法里面的testThis的时候，执行到this.test一句时候，应该调用子类的test方法，也就是应该输出”this is a human from MethodResolve”。
从上可以看出，一个方法的this语义是指实例的实际类型，而不是这个this关键字出现在代码中的那个类型。this引用调用的方法存在重写的可能，所以不可能在加载时候完成符号解析。

这里我们就可以提出一个初步的结论：：“如果一个方法，解析符号引用的时候不需要考虑实际类型，那么就可以在加载时期完成符号解析”。

我们先从invokestatic指令来验证一下这个结论。这很容易。invokestatic指令用于调用静态方法。虽然子类能够继承父类的方法，但是它有一个特殊性——就是在调用一个静态方法的时候，并没有实际类型的概念。举例来说：

```
public class MethodResolveFather {
    public static void staticTest(){
        System.out.println("this is a static me
    }
}
public class MethodResolve extends MethodResolv
    public static void staticTest(){
        System.out.println("this is a static me
    }
}

```

当我们调用：

```
        MethodResolveFather.staticTest();
        MethodResolve.staticTest();

```

很显然的是，它会输出

```
this is a static method from MethodResolveFather
this is a static method from MethodResolve

```

如果子类里面并没有这个staticTest方法:

```
public class MethodResolve extends MethodResolve
}

```

那么前面的调用输出的是：

```
this is a static method from MethodResolveFather
this is a static method from MethodResolveFather

```

奇怪的是，为什么静态方法明确有继承和重写的概念，但是却能够在加载期完成解析。答案就在于静态方法的调用参数里面，没有this引用。从指令的角度来说，就是invokestatic指令执行的时候，并不需要从操作数栈里面取出一个对象来。
对于静态方法而言，你使用父类来调用，那么调用的就是父类的版本；你使用的是子类来调用，那么调用的就是子类

的版本。即便用具体实例来调用静态方法，依旧遵循这条原则。
下面用一个奇诡的例子来说明这条论断的合理性，可以加深理解：

```
        MethodResolveFather father=new MethodRes
        father.staticTest();
        MethodResolve son=new MethodResolve();
        son.staticTest();
        MethodResolveFather methodResolve=new Me
        methodResolve.staticTest();

```

输出是：

```
this is a static method from MethodResolveFather
this is a static method from MethodResolve
this is a static method from MethodResolveFather

```

还记得我在引言部分里面说过，编译的时候，编译器是不在意你的实际类型的，也就是说:

```
methodResolve.staticTest();

```

编译之后的字节码是这样的：

```
36: invokestatic  #9                  // Method

```

下面我们再来考虑一个更加特殊的类型，final方法。final方法，如果没有被声明成static的话，也不是private方法，那么在编译之后的字节码里面，它将被invokevirtual指令调用。即便如此，final方法的符号引用依旧会在加载期就被解析完成。这里再次强调一下我们的论断：“如果一个方法，解析符号引用的时候不需要考虑实际类型，那么就可以在加载时期完成符号解析”。这些final方法完全符号我们所描述的。
一个方法如果被声明成了final，那么就意味着它无法被子类重写，也就是意味着，不论在什么时候解析这个方法的符号引用，只需要从声明类型出发，沿着继承树向上找就可以了。而一个类的祖先类，在加载的时候就是完全确定的。所以一个final方法完全可以在加载的时候完成解析。

由此我们可以肯定我们的结论是正确的：如果一个方法，解析符号引用的时候不需要考虑实际类型，那么就可以在加载时期完成符号解析。

invokesvirtual和invokeinterface确定方法的过程

根据前面讨论的结果，这两条指令执行的方法，除了被声明为final以外，其余的情况下，都需要在运行期完成符号解析。
Java虚拟机规范里面说，对于这两条指令，都是从实例的实际类型出发，沿着继承树向上查找。但是并没有规定具体如何实现。实际上，在hotspot里面，这两条指令的方法查找过程是有很大不同的。
hotspot里面采用的是vtable的东西。vtable是一张表，记录了该类里面的所有方法的位置。

> *注：这里所有的方法，我其实不太确定是否含有私有方法和静态方法，理论上来说，不记录可能更好*

例如：

```
public class VtableBaseClass {

    public void method1(){

    }

    public void method2(){

    }

    public void method3(){

    }

}
```

VtableBaseClass/method1
VtableBaseClass/method2
VtableBaseClass/method3

vtable还有一个十分重要的特性，就是一个方法在子类的vtable的位置和在父类中的位置是一样的，如：

```
public class VtableExtClass extends VtableBaseClass{
    @Override
    public void method2(){

    }

    public void method4(){

    }
}
```

VtableBaseClass/method1
VtableExtClass/method2
VtableBaseClass/method3
VtableExtClass/method4

这会带来一个极大的便利，就是在方法第一次被调用的情况下，查找vtable得到了该方法对应的表项的位置，那么下次就不再需要遍历这张表了，而只需要用记住的位置来访问对应的表项。
但是接口方法则不太一样，举例来说：

```
public class VtabelInterfaceImpl1 implements VtableInterface{
    public void ifaceMethod1() {

    }

    public void method2(){

    }

    public void method3(){

    }
}
```

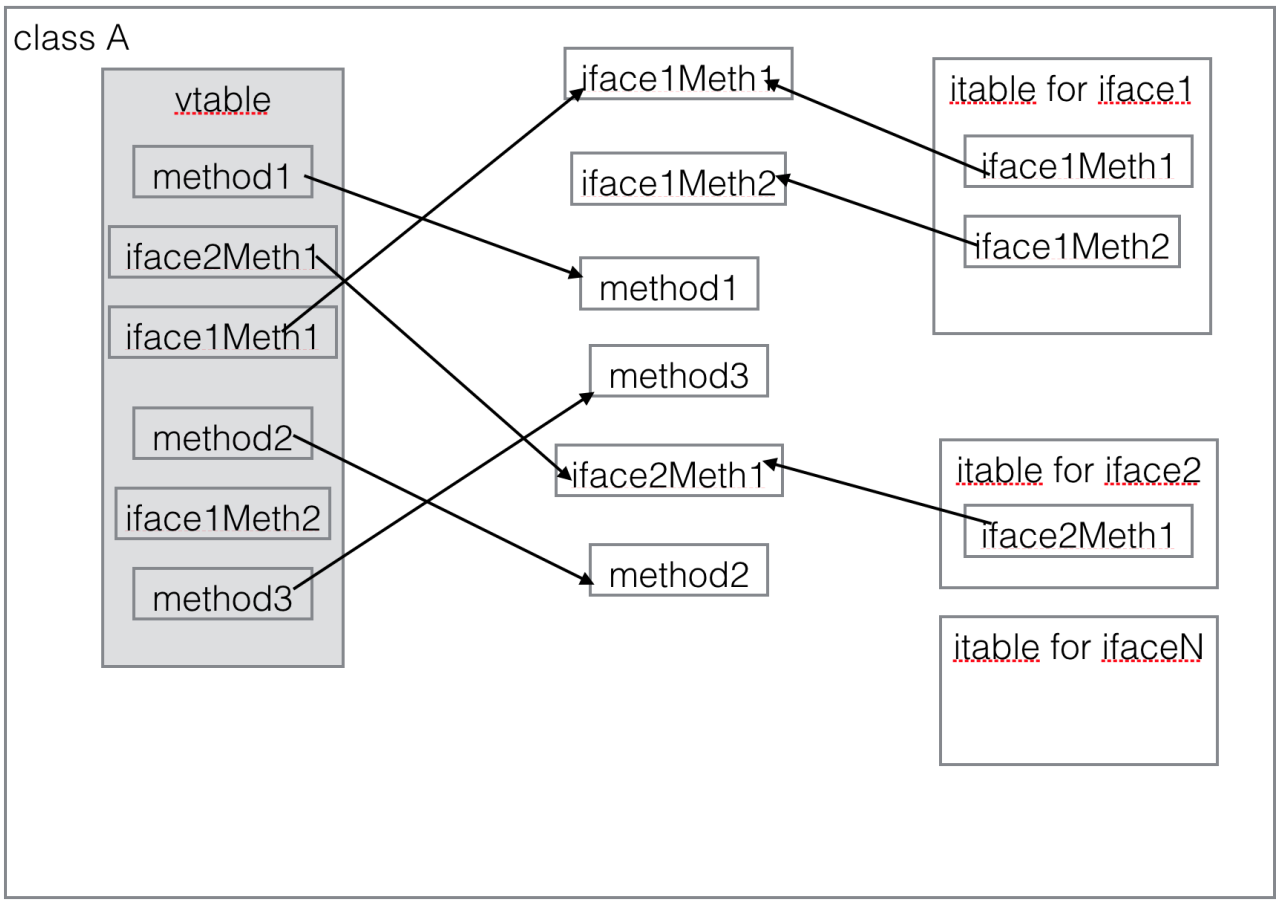
VtabelInterfaceImpl1/ifaceMethod1
VtabelInterfaceImpl1/method2
VtabelInterfaceImpl1/method3

```
public class VtableInterfaceImpl2 implements VtableInterface {

    public void methodA(){

    }

    public void methodB(){

    }

    public void ifaceMethod1() {

    }
}
```

VtabelInterfaceImpl2/methodA
VtabelInterfaceImpl2/methodB
VtabelInterfaceImpl2/ifaceMethod1

虽然两个类都继承同一个接口，但是接口方法在两个类的vtable中的位置是不同的。这就造成，使用接口调用接口方法，每一次都需要遍历整个vtable。
实际上，接口方法的调用，其实现方式在虚拟机hotspot里面是很大不同的。接口方法使用的是itable，它如同vtable一般，不过记录的是接口名/方法。

下图可以看出vtable和itable的区别和联系：



对于一个类来说，itable可能会存在多个。因此在解析一个接口方法的符号引用的时候，第一步是找到这个接口对应的itable，第二步才是从itable里面找到对应的方法

> *注：我认为这样的查找过程比直接查找vtable要快。也就是说，在符号解析的过程中，使用vtable来解析接口方法的符号引用不如使用itable快。这个论断我不太有信心，可能是错的*

性能分析

从前面的分析，可以看出来，通常意义上说，invokestatic和invokespecial比较快是有道理的，它们能够在加载期完成方法的符号解析；而invokevirtual中的final方法也可以达到和invokestatic, invokespecial同样的效果；而在invokevirtual和invokeinterface之间，invokeinterface的确慢，但是现代的编译器和虚拟机能够通过许多种的努力，使它的执行效率能够接近invokevirtual的效率。

总体上来说，将一个方法写成final是有益的；使用抽象类来调用方法比使用接口来调用方法也是有益的。但是我觉得通过这种手段来提高运行效率的效果是比较小的。