原 Java中为什么静态方法不能被重写？为什么静态方法不能隐藏实例方法？

2017年07月04日 21:16:28 Special__Yang 阅读数：6304 标签： java 继承 class 静态方法 实例方法 更多

问题描述

Java中为什么静态方法不能被重写？为什么静态方法不能隐藏实例方法？诸如此类。

前期准备

首先理解重写的意思，重写就是子类中对父类的实例方法进行重新定义功能，且返回类型、方法名以及参数列表保持一致，且对重写的调用主型。实际类型如果实现了该方法则直接调用该方法，如果没有实现，则在继承关系中从低到高搜索有无实现。那么问题又来了，为什么只能对实例方法头好晕，这两个问题在这互相推脱责任。

理解三个概念：静态类型，实际类型，方法接受者。

```
1  Person student= new Student();
2  student.work();
```

> 静态类型就是编译器编译期间认为对象所属的类型，这个主要根据声明类型决定，所以上述Person就是静态类型
> 实际类型就是解释器在执行时根据引用实际指向的对象所决定的，所以Student就是实际类型。
> 方法接受者就是动态绑定所找到执行此方法的对象，比如student。

还要理解类编译的class文件中字节码的方法调用指令。

（1）invokestatic：调用静态方法

（2）invokespecial：调用实例构造器方法，私有方法。

（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：先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。

> 非虚方法：不能被重写或者说覆盖的方法，指的是构造方法、静态方法、私有方法和final 修饰的方法。
> 虚方法：则是能被重写的方法，一般指的是实例方法。

例子

```
1  package com.learn.pra06;
2  class Demo01{
3      public void method1(){
4          System.out.println("This is father non-static");
5      }
6      public static void method2(){
7          System.out.println("This is father static");
8      }
9  }
10 public class Demo02 extends Demo01{
11     public void method1(){
12         System.out.println("This is son non-static");
13     }
14     public static void method2(){
15         System.out.println("This is son static");
16     }
17
18     public static void main(String[] args){
19         Demo01 d1= new Demo01();
20         Demo02 d2= new Demo02();
21         Demo01 d3= new Demo02(); //父类引用指向子类对象
22         d1.method1();
23         d1.method2();
24         d2.method1();
25         d2.method2();
```

```
26          d3.method1();
27          d3.method2();
28      }
29  }
```

运行结果：

```
This is father non-static
This is father static
This is son non-static
This is son static
This is son non-static
This is father static
```

对于这样的运行结果前5行应该没什么疑问，用常规的思维就能理解，可是最后一条，what?说好的动态绑定呢，Are you kidding me？No,这里没有定了，问题又来了，为什么静态方法不发生动态绑定？动态绑定到底发生了什么？简直是头脑风暴，说实话我也是蒙蒙的，接下来可能正确也可能不正不离十。

分析

首先看看上面main 方法的字节码：

```
1   // access flags 0x9
2    public static main([Ljava/lang/String;)V
3     L0
4      LINENUMBER 19 L0
5      NEW com/learn/pra06/Demo01
6      DUP
7      INVOKESPECIAL com/learn/pra06/Demo01.<init> ()V
8      ASTORE 1
9     L1
10     LINENUMBER 20 L1
11     NEW com/learn/pra06/Demo02
12     DUP
13     INVOKESPECIAL com/learn/pra06/Demo02.<init> ()V
14     ASTORE 2
15    L2
16     LINENUMBER 21 L2
17     NEW com/learn/pra06/Demo02
18     DUP
19     INVOKESPECIAL com/learn/pra06/Demo02.<init> ()V
20     ASTORE 3
21    L3
22     LINENUMBER 22 L3
23     ALOAD 1
24     INVOKEVIRTUAL com/learn/pra06/Demo01.method1 ()V
25    L4
26     LINENUMBER 23 L4
27     INVOKESTATIC com/learn/pra06/Demo01.method2 ()V
28    L5
29     LINENUMBER 24 L5
30     ALOAD 2
31     INVOKEVIRTUAL com/learn/pra06/Demo02.method1 ()V
32    L6
33     LINENUMBER 25 L6
34     INVOKESTATIC com/learn/pra06/Demo02.method2 ()V
35    L7
36     LINENUMBER 26 L7
37     ALOAD 3
38     INVOKEVIRTUAL com/learn/pra06/Demo01.method1 ()V
39    L8
40     LINENUMBER 27 L8
41     INVOKESTATIC com/learn/pra06/Demo01.method2 ()V
42    L9
43     LINENUMBER 28 L9
44     RETURN
45    L10
```

L+number 对应就是main方法体中每一行，我们可以清晰的看见代码执行的指令，简直大爱，有种相见恨晚的赶脚。

`Demo01 d1= new Demo01();` 这个语句将会在运行期发生什么呢？结合我们前期准备学的那几个指令集。查看以上的字节码发现： `INVOKESPECIA com/learn/pra06/Demo01.<init> ()V` 请问将会调用Demo01的构造函数，这个毋庸置疑。同理L1也是如此。

`Demo01 d3= new Demo02();` 虽然声明类型为父类，但实际new的时候是子类，同样字节码也对应如此。 `INVOKESPECIAL com/learn/pra06/Demo ()V`

`d1.method1();` 这个语句是应该是对象调用其实例方法，字节码也很好说明了这一点： `INVOKEVIRTUAL com/learn/pra06/Demo01.method1 ()V` INVOKEVIRTUAL，则代表调用虚方法，并且此方法的引用存在方法表中（这个待会再说），**只用INVOKEVIRTUAL指令会去方法表寻找要调用方法的**

`d1.method2();` 这句是对象调用静态方法，字节码为： `INVOKESTATIC com/learn/pra06/Demo01.method2 ()V` 此方法则是直接调用方法区中静态 经过方法表，这也就解释了静态方法的执行只看静态类型，而与实际类型无关，又因为重写的方法调用看的是实际类型，所以静态方法不能被重写。d2 调用解释与d1相同。

重点是d3方法的调用过程， `d3.method1();` 字节码： `INVOKEVIRTUAL com/learn/pra06/Demo01.method1 ()V` 运用了INVOKEVIRTUAL指令，该 会到方法表中去调用真实指向的方法，因为method01可能被重写，所以编译器期间标明运行时应调用method1所在的方法表中位置存的真正方法的引用 method01方法被Demo02重写，所以方法表中原先存父类method01方法的引用被改写成子类的method01方法的引用，所以在运行时根据INVOKEVIRT 到的method01的方法是子类的。

那么 `d3.method2();` 通过查看字节码发现： `INVOKESTATIC com/learn/pra06/Demo01.method2 ()V` 用到INVOKESTATIC ，不能访问方法表，而 父类的method2的，所以运行时调用就是父类的静态方法。

> 编译时把对象的静态类型（声明类型）作为该方法的接受者。运行时则根据指令集再进行更改。

INVOKEVIRTUAL指令流程

```
1  package com.learn.pra06;
2  public class ClassReference {
3      static class Person {
4          @Override
5          public String toString(){
6              return "I'm a person.";
7          }
8          public void eat(){
9              System.out.println("Person eat");
10         }
11         public void speak(){
12             System.out.println("Person speak");
13         }
14
15     }
16     static class Boy extends Person{
17         @Override
18         public String toString(){
19             return "I'm a boy";
20         }
21         @Override
22         public void speak(){
23             System.out.println("Boy speak");
24         }
25         public void fight(){
26             System.out.println("Boy fight");
27         }
28     }
29     static class Girl extends Person{
30         @Override
31         public String toString(){
32             return "I'm a girl";
33         }
34         @Override
35         public void speak(){
36             System.out.println("Girl speak");
37         }
38         public void sing(){
39             System.out.println("Girl sing");
40         }
41     }
42     public static void main(String[] args) {
43         Person boy = new Boy();
44         Person girl = new Girl();
45         System.out.println(boy);
```

```
46          boy.eat();
47          boy.speak();
48          System.out.println(girl);
49          girl.eat();
50          girl.speak();
51      }
52  }
```

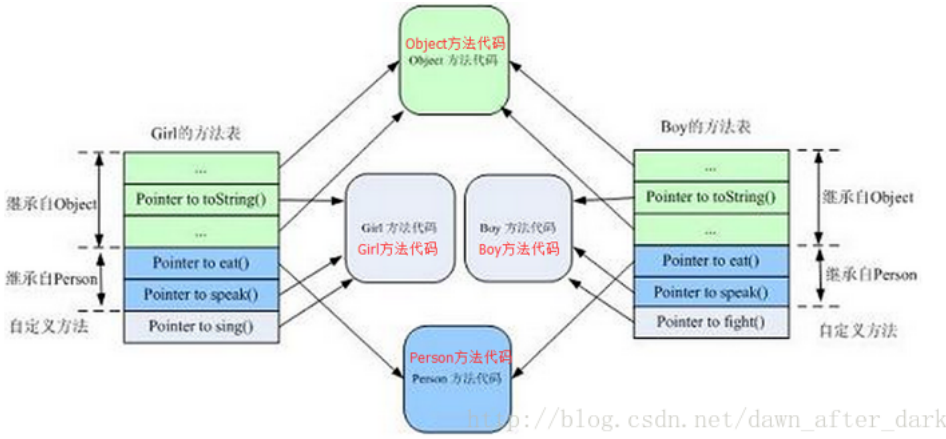
执行结果：

```
I'm a boy
Person eat
Boy speak
I'm a girl
Person eat
Girl speak
```

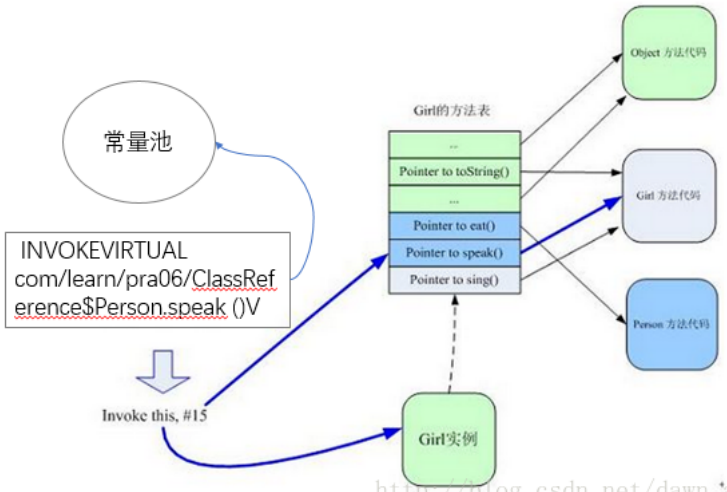
由于Boy 和Girl 没有重写父类Person eat方法，所以会调用父类的eat方法。
字节码：

```
1    public static main([Ljava/lang/String;)V
2     L0
3      LINENUMBER 47 L0
4      NEW com/learn/pra06/ClassReference$Boy
5      DUP
6      INVOKESPECIAL com/learn/pra06/ClassReference$Boy.<init> ()V
7      ASTORE 1
8     L1
9      LINENUMBER 48 L1
10     NEW com/learn/pra06/ClassReference$Girl
11     DUP
12     INVOKESPECIAL com/learn/pra06/ClassReference$Girl.<init> ()V
13     ASTORE 2
14    L2
15     LINENUMBER 49 L2
16     GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
17     ALOAD 1
18     INVOKEVIRTUAL java/io/PrintStream.println (Ljava/lang/Object;)V
19    L3
20     LINENUMBER 50 L3
21     ALOAD 1
22     INVOKEVIRTUAL com/learn/pra06/ClassReference$Person.eat ()V
23    L4
24     LINENUMBER 51 L4
25     ALOAD 1
26     INVOKEVIRTUAL com/learn/pra06/ClassReference$Person.speak ()V
27    L5
28     LINENUMBER 53 L5
29     GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
30     ALOAD 2
31     INVOKEVIRTUAL java/io/PrintStream.println (Ljava/lang/Object;)V
32    L6
33     LINENUMBER 54 L6
34     ALOAD 2
35     INVOKEVIRTUAL com/learn/pra06/ClassReference$Person.eat ()V
36    L7
37     LINENUMBER 55 L7
38     ALOAD 2
39     INVOKEVIRTUAL com/learn/pra06/ClassReference$Person.speak ()V
40    L8
41     LINENUMBER 57 L8
42     RETURN
43    L9
```

很明显在L2处，编译器会将根类Object的toString 方法的引用写入class文件，说明编译器会将祖先的方法引用写入，而非近亲。
L3,L4,L6,L7都用到了INVOKEVIRTUAL，到底流程是怎么样的呢？
首先看看方法表在内存的模型：



通过看Girl和Boy方法表可以看出继承的方法从头到尾开始排列，并且方法引用在子类的中都有固定索引，即都有相同的偏移量；若子类重写父类某个方
子类方法表原先存父类的方法引用变成重写后方法的引用，到这就应该理解为什么可以根据对象类型而调用到正确的方法，关键就在于**方法表。**
下面以 `girl.speak` 为例，看看INVOKEVIRTUAL指令流程



解说图：
1. 首先INVOKEVIRTUAL com/learn/pra06/ClassReference$Person.speak()V$中，根据$com/learn/pra06/ClassReference$Person.speak ()V
到该方法的偏移量
2. 查看Person的方法表，得到speak方法在该方法表的偏移量（假设为15），这样就得到该方法的直接引用。
3. 根据this判断出该引用指的是Girl实例
4. 然后去找Girl实例的方法表，根据上面的偏移量在方法表中找到该方法引用，因为该方法引用的值在类加载根据是否重写了方法已经确定了正确的方
我们这里就可以直接调用该方法。

为什么静态方法不能隐藏实例方法？

静态方法的调用的是通过在编译器静态绑定的，而实例方法的调用是在运行时动态绑定的，2者的调用的方式不同，所以二者只能存在其一，否则会存

为什么静态方法能隐藏静态方法？

因为调用方式一致，不会像上面造成歧义，虽然父类和子类都定义了同样的函数，但是编译器会根据对象的静态类型激活对应的静态方法的引用，造成
象，实则不是重写！

总结

总体流程就是：编译器将类编译成class文件，其中方法会根据静态类型从而将对应的方法引用写入class中，运行时，JVM会根据INVOKEVIRTUAL 所
用在常量池找到该方法的偏移量，再根据this找到引用类型真实指向的对象，访问这个对象类型的方法表，根据偏移量找出存放目标方法引用的位置，
用，调用这个引用实际指向的方法，完成多态！