本文链接： http://www.cnblogs.com/yyyyy5101/archive/2011/08/02/2125324.html

动态绑定意味着：目前正在调用的方法正是最适用于要操作对象的那个方法。然而它并不意味着对所有的参数都执行最佳匹配。  
在Java中，一个方法的参数在编译阶段常被静态地绑定。  
  
一个例子：

1. class Base{
2. public void foo(Base x){
3. System.out.println("Base.Base");
4. }
6. public void foo(Derived x){
7. System.out.println("Base.Derived");
8. }
9. }

1. class Derived extends Base{
2. public void foo(Base x){
3. System.out.println("Derived.Base");
4. }
6. public void foo(Derived x){
7. System.out.println("Derived.Derived");
8. }
9. }

1. class Main{
2. public static void whichFoo(Base arg1, Base arg2){
3. arg1.foo(arg2);
4. }
6. public static void main(String[] args)}{
7. Base b = new Base();
8. Derived d = new Derived();
10. whichFoo(b,b);
11. whichFoo(b,d);
12. whichFoo(d,b);
13. whichFoo(d,d);
14. }
15. }

因为参数通常在编译阶段被匹配，在whichFoo方法中，形式参数arg2的类型是Base, 因此不管arg2实际引用的是什么类型，arg1.foo(arg2)匹配的foo都将是：  
public void foo(Base x)  
  
惟一的问题在于用Base还是Derived版本中的foo(Base x)函数？当知道arg1引用的对象时，这是在运行阶段要决定的。  
  
精确使用的方法是编译器绑定，在编译阶段，最佳方法名依赖于参数的静态和控制引用的静态类型所适合的方法。在这一点上，设置方法的名称，这一步叫静态重载。  
决定方法是哪一个类的版本，这通过由虚拟机推断出这个对象的运行时类型来完成，一旦知道运行时类型，虚拟机就唤起继承机制，寻找方法的最终版本。这叫做动态绑定。  
  
在方法whichFoor的调用arg1.foo(arg2)，将根据arg1的运行时类型是Base还是Derived来调用Base类或者Derived类中的foo(Base x)版本函数。  
  
由此理解方法的覆盖和重载。重载函数的实际调用版本由编译器绑定决定，而覆盖函数的实际调用版本由动态绑定决定。  
Example: Overloading Ambiguity   
Consider the example:   
class Point { int x, y; }   
class ColoredPoint extends Point { int color; }   
  
class Test {   
static void test(ColoredPoint p, Point q) {   
System.out.println("(ColoredPoint, Point)");   
}   
static void test(Point p, ColoredPoint q) {   
System.out.println("(Point, ColoredPoint)");   
}   
public static void main(String[] args) {   
ColoredPoint cp = new ColoredPoint();   
test(cp, cp);                                                                                   // compile-time error   
}   
}   
This example produces an error at compile time. The problem is that there are two declarations of test that are applicable and accessible, and neither is more specific than the other. Therefore, the method invocation is ambiguous.   
If a third definition of test were added:   
static void test(ColoredPoint p, ColoredPoint q) {   
System.out.println("(ColoredPoint, ColoredPoint)");   
}   
then it would be more specific than the other two, and the method invocation would no longer be ambiguous.

问题的由来：  
首先是方法的参数是父类对象，传入子类对象是否可行  
然后引出Parent p = new Children();  
这句代码不是很理解，google的过程中引出向上转型  
要理解向上转型又引出了动态绑定  
从动态绑定又引出了静态绑定

程序**绑定**的概念：  
绑定指的是一个方法的调用与方法所在的类(方法主体)关联起来。对java来说，绑定分为静态绑定和动态绑定；或者叫做前期绑定和后期绑定

**静态绑定**：  
在程序执行前方法已经被绑定，此时由编译器或其它连接程序实现。例如：C。  
针对java简单的可以理解为程序编译期的绑定；这里特别说明一点，java当中的方法只有final，static，private和构造方法是前期绑定

**动态绑定**：  
后期绑定：在运行时根据具体对象的类型进行绑定。  
若一种语言实现了后期绑定，同时必 须提供一些机制，可在运行期间判断对象的类型，并分别调用适当的方法。也就是说，编译器此时依然不知道对象的类型，但方法调用机制能自己去调查，找到正确 的方法主体。不同的语言对后期绑定的实现方法是有所区别的。但我们至少可以这样认为：它们都要在对象中安插某些特殊类型的信息。

**动态绑定的过程**：  
虚拟机提取对象的实际类型的方法表；  
虚拟机搜索方法签名；  
调用方法。

**关于绑定相关的总结**：  
在了解了三者的概念之后，很明显我们发现java属于后期绑定。在java中，几 乎所有的方法都是后期绑定的，在运行时动态绑定方法属于子类还是基类。但是也有特殊，针对static方法和final方法由于不能被继承，因此在编译时 就可以确定他们的值，他们是属于前期绑定的。特别说明的一点是，private声明的方法和成员变量不能被子类继承，所有的private方法都被隐式的 指定为final的(由此我们也可以知道：将方法声明为final类型的一是为了防止方法被覆盖，二是为了有效的关闭java中的动态绑定)。java中 的后期绑定是有JVM来实现的，我们不用去显式的声明它，而C++则不同,必须明确的声明某个方法具备后期绑定。

**java当中的向上转型或者说多态是借助于动态绑定实现的，所以理解了动态绑定，也就搞定了向上转型和多态**。  
前面已经说了对于java当中的方法而言，除了final，static，private和构造方法是前期绑定外，其他的方法全部为动态绑定。而动态绑定的典型发生在父类和子类的转换声明之下：  
比如：Parent p = new Children();  
其具体过程细节如下：  
1：编译器检查对象的声明类型和方法名。假设我们调用x.f(args)方法，并且x已经被声明为C类的对象，那么编译器会列举出C类中所有的名称为f的方法和从C类的超类继承过来的f方法  
2：接下来编译器检查方法调用中提供的参数类型。如果在所有名称为f 的方法中有一个参数类型和调用提供的参数类型最为匹配，那么就调用这个方法，这个过程叫做“重载解析”    
3：当程序运行并且使用动态绑定调用方法时，虚拟机必须调用同x所指向的对象的实际类型相匹配的方法版本。假设实际类型为D(C的子类)，如果D类定义了f(String)那么该方法被调用，否则就在D的超类中搜寻方法f(String),依次类推

上面是理论，下面看几个示例（示例来自网络）：

1. view plaincopy to clipboardprint?
2. public class Father {
3. public void method() {
4. System.out.println("父类方法，对象类型：" + this.getClass());
5. }
6. }
8. public class Son extends Father {
9. public static void main(String[] args) {
10. Father sample = new Son();//向上转型
11. sample.method();
12. }
13. }

声明的是父类的引用，但是执行的过程中调用的是子类的对象，程序首先寻找子类对象的method方法，但是没有找到，于是向上转型去父类寻找

1. public class Son extends Father {
2. public void method() {
3. System.out.println("子类方法，对象类型：" + this.getClass());
4. }
6. public static void main(String[] args) {
7. Father sample = new Son();//向上转型
8. sample.method();
9. }
10. }

由于子类重写了父类的method方法，根据上面的理论知道会去调用子类的method方法去执行，因为子类对象有method方法而没有向上转型去寻找

前面的理论当中已经提到了java的绑定规则，由此可知，在处理java类中的成员变量时，并不是采用运行时绑定，而是一般意义上的静态绑定。所以在向上转型的情况下，对象的方法可以找到子类，而对象的属性还是父类的属性。

代码如下：

1. public class Father {
3. protected String name="父亲属性";
5. public void method() {
6. System.out.println("父类方法，对象类型：" + this.getClass());
7. }
8. }
10. public class Son extends Father {
11. protected String name="儿子属性";
13. public void method() {
14. System.out.println("子类方法，对象类型：" + this.getClass());
15. }
17. public static void main(String[] args) {
18. Father sample = new Son();//向上转型
19. System.out.println("调用的成员："+sample.name);
20. }
21. }

 结论，调用的成员为父亲的属性。

这个结果表明，子类的对象(由父类的引用handle)调用到的是父类的成员变量。所以必须明确，运行时（动态）绑定针对的范畴只是对象的方法。

现在试图调用子类的成员变量name，该怎么做？最简单的办法是将该成员变量封装成方法getter形式。

代码如下：

1. public class Father {
2. protected String name = "父亲属性";
3. public String getName() {
4. return name;
5. }
6. public void method() {
7. System.out.println("父类方法，对象类型：" + this.getClass());
8. }
9. }
11. public class Son extends Father {
12. protected String name="儿子属性";
14. public String getName() {
15. return name;
16. }
18. public void method() {
19. System.out.println("子类方法，对象类型：" + this.getClass());
20. }
22. public static void main(String[] args) {
23. Father sample = new Son();//向上转型
24. System.out.println("调用的成员："+sample.getName());
25. }
26. }

结果：调用的是儿子的属性