13丨软件设计的里氏替换原则：正方形可以继承长方形吗？

2019-12-20 李智慧

讲述：李智慧
时长 11:04 大小 10.15M

我们都知道，面向对象编程语言有三大特性：封装、继承、多态。这几个特性也许可以很快就学会，但是如果想要用好，可能要花非常多的时间。

通俗地说，接口（抽象类）的多个实现就是多态。多态可以让程序在编程时面向接口进行编程，在运行期绑定具体类，从而使得类之间不需要直接耦合，就可以关联组合，构成一个更强大的整体对外服务。绝大多数设计模式其实都是利用多态的特性玩的把戏，前面两篇学习的开闭原则和依赖倒置原则也是利用多态的特性。正是多态使得编程有时候像变魔术，如果能用好多态，可以说掌握了大多数的面向对象编程技巧。

封装是面向对象语言提供的特性，将属性和方法封装在类里面。用好封装的关键是，知道应该将哪些属性和方法封装在某个类里。一个方法应该封装进 A 类里，还是 B 类里？这个问

题其实就是如何进行对象的设计。深入研究进去，里面也有大量的学问。

继承似乎比多态和封装要简单一些，但实践中，继承的误用也很常见。

里氏替换原则

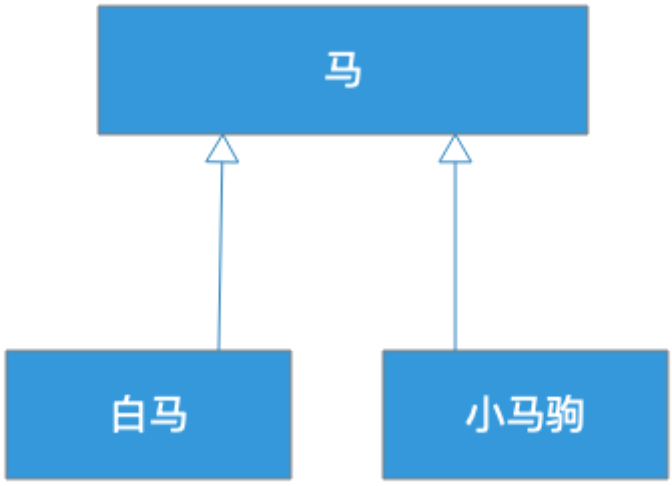
关于如何设计类的继承关系，怎样使继承不违反开闭原则，实际上有一个关于继承的设计原则，叫里氏替换原则。这个原则说：若对每个类型 T1 的对象 o1，都存在一个类型 T2 的对象 o2，使得在所有针对 T2 编写的程序 P 中，用 o1 替换 o2 后，程序 P 的行为功能不变，则 T1 是 T2 的子类型。

上面这句话比较学术，通俗地说就是：**子类型必须能够替换掉它们的基类型。**

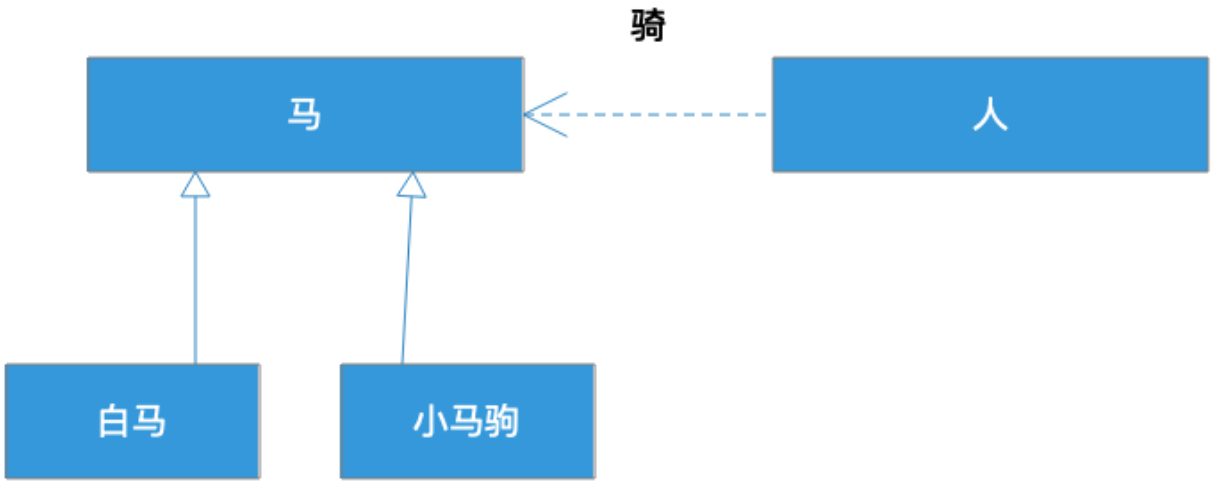
再稍微详细点说，就是：程序中，所有使用基类的地方，都应该可以用子类代替。

语法上，任何类都可以被继承。但是一个继承是否合理，从继承关系本身是看不出来的，需要把继承放在应用场景的上下文中去判断，使用基类的地方，是否可以用子类代替？

这里有一个马的继承设计：



白马和小马驹都是马，所以都继承了马。这样的继承是不是合理呢？我们需要放到应用场景中：



在这个场景中，是人骑马。根据这里的关系，继承了马的白马和小马驹，应该都可以代替马。白马代替马当然没有问题，人可以骑白马，但是小马驹代替马可能就不合适了，因为小马驹还没长好，无法被人骑。

那么很显然，作为子类的白马可以替换掉基类马，但是小马不能替换马，因此小马继承马就不太合适了，违反了里氏替换原则。

一个违反里氏替换规则的例子

我们再看这样一段代码：

复制代码

```
void drawShape(Shape shape) {
    if (shape.type == Shape.Circle ) {
        drawCircle((Circle) shape);
    } else if (shape.type == Shape.Square) {
        drawSquare((Square) shape);
    } else {
        ……
    }
}
```

这里 Circle 和 Square 继承了基类 Shape，然后在应用的方法中，根据输入 Shape 对象类型进行判断，根据对象类型选择不同的绘图函数将图形画出来。这种写法的代码既常见又糟糕，它同时违反了开闭原则和里氏替换原则。

首先看到这样的 if/else 代码，就可以判断违反了开闭原则：当增加新的 Shape 类型的时候，必须修改这个方法，增加 else if 代码。

其次也因为同样的原因违反了里氏替换原则：当增加新的 Shape 类型的时候，如果没有修改这个方法，没有增加 else if 代码，那么这个新类型就无法替换基类 Shape。

要解决这个问题其实也很简单，只需要在基类 Shape 中定义 draw 方法，所有 Shape 的子类，Circle、Square 都实现这个方法就可以了：

复制代码

```
public abstract Shape{
  public abstract void draw();
}
```

上面那段 drawShape() 代码也就可以变得更简单：

复制代码

```
void drawShape(Shape shape) {
  shape.draw();
}
```

这段代码既满足开闭原则：增加新的类型不需要修改任何代码。也满足里氏替换原则：在使用基类的这个方法中，可以用子类替换，程序正常运行。

正方形可以继承长方形吗?

一个继承设计是否违反里氏替换原则，需要在具体场景中考察。我们再看一个例子，假设我们现在有一个长方形的类，类定义如下：

复制代码

```
public class Rectangle {
    private double width;
    private double height;
    public void setWidth(double w) { width = w; }
    public void setHeight(double h) { height = h; }
    public double getWidth() { return width; }
    public double getHeight() { return height; }
    public double calculateArea() {return width * height;}
```

```
9 }
```

这个类满足我们的应用场景，在程序中多个地方被使用，一切良好。但是现在，我们有个新需求，我们还需要一个正方形。

通常，我们判断一个继承是否合理，会使用“IS A”进行判断，类 B 可以继承类 A，我们就说类 B IS A 类 A，比如白马 IS A 马，轿车 IS A 车。

那正方形是不是 IS A 长方形呢？通常我们会说，正方形是一种特殊的长方形，是长和宽相等的长方形，从这个角度讲，那么正方形 IS A 长方形，也就是可以继承长方形。

具体实现上，我们只需要在设置长方形的长或宽的时候，同时设置长和宽就可以了，如下：

复制代码

```
1 public class Square extends Rectangle {
2     public void setWidth(double w) {
3         width = height = w;
4     }
5     public void setHeight(double h) {
6         height = width = w;
7     }
8 }
```

这个正方形类设计看起来很正常，用起来似乎也没有问题。但是，真的没有问题吗？

继承是否合理我们需要用里氏替换原则来判断。之前也说过，是否合理并不是从继承的设计本身看，而是从应用场景的角度看。如果在应用场景中，也就是在程序中，子类可以替换父类，那么继承就是合理的，如果不能替换，那么继承就是不合理的。

这个长方形的使用场景是什么样的呢，我们看使用代码：

复制代码

```
1 void testArea(Rectangle rect) {
2     rect.setWidth(3);
3     rect.setHeight(4);
4     assert 12 == rect.calculateArea();
5 }
```

显然，在这个场景中，如果用子类 Square 替换父类 Rectangle，计算面积 calculateArea 将返回 16，而不是 12，程序是不能正确运行的，这样的继承不满足里氏替换原则，是不合适的继承。

子类不能比父类更严格

类的公有方法其实是对使用者的一个契约，使用者按照这个契约使用类，并期望类按照契约运行，返回合理的值。

当子类继承父类的时候，根据里氏替换原则，使用者可以在使用父类的地方使用子类替换，那么从契约的角度，子类的契约就不能比父类更严格，否则使用者在用子类替换父类的时候，就会因为更严格的契约而失败。

在上面这个例子中，正方形继承了长方形，但是正方形有比长方形更严格的契约，即正方形要求长和宽是一样的。因为正方形有比长方形更严格的契约，那么在使用长方形的地方，正方形因为更严格的契约而无法替换长方形。

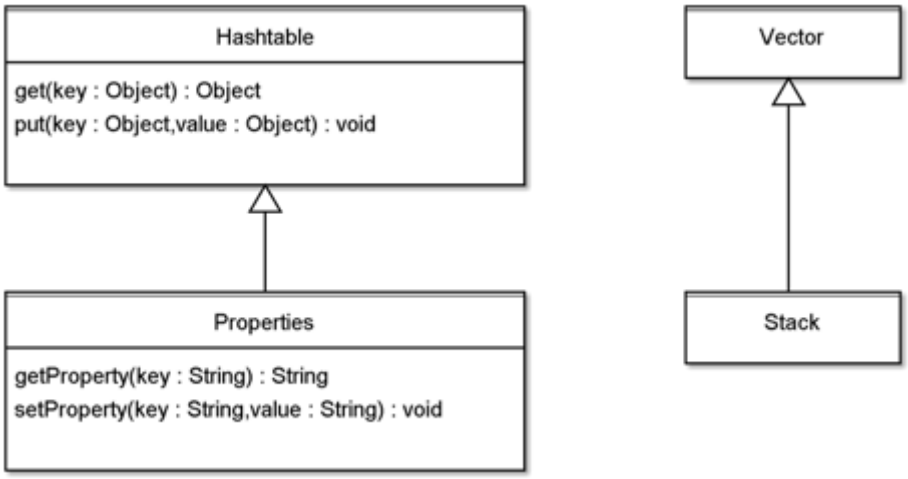
我们开头小马继承马的例子也是如此，小马比马有更严格的要求，即不能骑，那么小马继承马就是不合适的。

在类的继承中，如果父类方法的访问控制是 protected，那么子类 override 这个方法的时候，可以改成是 public，但是不能改成 private。因为 private 的访问控制比 protected 更严格，能使用父类 protected 方法的地方，不能用子类的 private 方法替换，否则就是违反里氏替换原则的。相反，如果子类方法的访问控制改成 public 就没问题，即子类可以有比父类更宽松的契约。同样，子类 override 父类方法的时候，不能将父类的 public 方法改成 protected，否则会出现编译错误。

通常说来，子类比父类的契约更严格，都是违反里氏替换原则的。

子类不应该比父类更严格，这个原则看起来既合理又简单，但是在实际中，如果你不严谨地审视自己的设计，是很可能违背里氏替换原则的。

在 JDK 中，类 Properties 继承自类 Hashtable，类 Stack 继承自 Vector。



这样的设计，其实是违反里氏替换原则的。Properties 要求处理的数据类型是 String，而它的父类 Hashtable 要求处理的数据类型是 Object，子类比父类的契约更严格；Stack 是一个栈数据结构，数据只能先进先出，而它的父类 Vector 是一个线性表，子类比父类的契约更严格。

这两个类都是从 JDK1 就已经存在的，我想，如果能够重新再来，JDK 的工程师一定不会这样设计。这也从另一个方面说明，不恰当的继承是很容易就发生的，设计继承的时候，需要更严谨的审视。

小结

实践中，当你继承一个父类仅仅是为了复用父类中的方法的时候，那么很有可能你离错误的继承已经不远了。一个类如果不是为了被继承而设计，那么最好就不要继承它。粗暴一点地说，如果不是抽象类或者接口，最好不要继承它。

如果你确实需要使用一个类的方法，最好的办法是组合这个类而不是继承这个类，这就是人们通常说的**组合优于继承**。比如这样：

复制代码

```
Class A{
 public Element query(int id){...}
 public void modify(Element e){...}
}

Class B{
  private A a;
  public Element select(int id){
    a.query(id);
  }
```

```
    public void modify(Element e){
      a.modify(e);
    }
}
```

如果类 B 需要使用类 A 的方法，这时候不要去继承类 A，而是去组合类 A，也能达到使用类 A 方法的效果。这其实就是**对象适配器模式**了，使用这个模式的话，类 B 不需要继承类 A，一样可以拥有类 A 的方法，同时还有更大的灵活性，比如可以改变方法的名称以适应应用接口的需要。

当然，继承接口或者抽象类也并不保证你的继承设计就是正确的，最好的方法还是用里氏替换原则检查一下你的设计：使用父类的地方是不是可以用子类替换？

违反里氏替换原则不仅仅发生在设计继承的地方，也可能发生在使用父类和子类的地方，错误的使用方法，也可能导致程序违反里氏替换原则，使子类无法替换父类。

思考题

下面给你留一道思考题吧。

父类中有抽象方法 f，抛出异常 AException：

复制代码

```
public abstract void f() throws AException;
```

子类 override 父类这个方法后，想要将抛出的异常改为 BException，那么 BException 应该是 AException 的父类还是子类？

为什么呢？请你用里氏替换原则说明，并在评论区写下你的思考，我会和你一起交流，也欢迎你把这篇文章分享给你的朋友或者同事，一起交流一下。

上一篇　12 | 软件设计的依赖倒置原则：如何不依赖代码却可以复用它的功能?

下一篇　14 | 软件设计的单一职责原则：为什么说一个类文件打开最好不要超过一屏?

精选留言 (10)

写留言

观弈道人
2019-12-20

BException应该是AAexception的子类

展开

1　5

俊杰
2019-12-20

BException应该是AException的子类，否则当使用子类替换父类后，抛出的BException无法被catch(AException e)语句捕获

展开

作者回复: √

3

苏志辉
2019-12-20

BExpection应该是AExpection的父类，子类不能比父类抛的更广，否则，使用父类的地方没法处理

1 3

陈小龙 Cheney
2019-12-20

BException应该是AException的子类。
因为子类必须能够替换掉父类，因此子类抛出的异常，原先处理父类的代码必须能够处理。那么子类抛出的BEception就应当是AEception的子类，才能被处理父类异常的代码正确处理。

展开

2

Citizen Z
2019-12-22

假如 AException extends BException

```
Father f = new Child();
try {
    father.f(); // throws BException...
```

展开

扬帆丶启航
2019-12-22

按照按照里氏替换的原则，BException是AException的父类，这样才能满足在各个情况下都能替换抽象类的方法，如果是AException的子类，可能在某些情况，不满足BException

一步
2019-12-21

里氏替换原则 要求子类可以无缝的替换父类，比父类更松。

但是在实际的开发中，往往是子类比父类更加严格，细化到适合使用在某一应用场景下，目的性越来越明确

...

展开

一步

2019-12-21

LSP(里氏替换原则) 应该还要求参数和返回值也要一致的，要不然子类没办法替换父类

lifuz

2019-12-20

BException 应该是AException 的父类，用集合的概念，BException 是大集合，包含AException这个小集合，再按照里式替换原则子类不能比父类更严格，即只能用更大的集合，不能用它的子集

Geek_robert

2019-12-20

课程越来越好了

展开

1