

## 10 | 理论七：为何说要多用组合少用继承？如何决定该用组合还是继承？

王争 · 设计模式之美



在面向对象编程中，有一条非常经典的设计原则，那就是：组合优于继承，多用组合少用继承。为什么不推荐使用继承？组合相比继承有哪些优势？如何判断该用组合还是继承？今天，我们就围绕着这三个问题，来详细讲解一下这条设计原则。


话不多说，让我们正式开始今天的学习吧！

### 为什么不推荐使用继承？

继承是面向对象的四大特性之一，用来表示类之间的 is-a 关系，可以解决代码复用的问题。虽然继承有诸多作用，但继承层次过深、过复杂，也会影响到代码的可维护性。所以，对于是否应该在项目中使用继承，网上有很多争议。很多人觉得继承是一种反模式，应该尽量少用，甚至不用。为什么会有这样的争议？我们通过一个例子来解释一下。

假设我们要设计一个关于鸟的类。我们将“鸟类”这样一个抽象的事物概念，定义为一个抽象类 `AbstractBird`。所有更细分的鸟，比如麻雀、鸽子、乌鸦等，都继承这个抽象类。

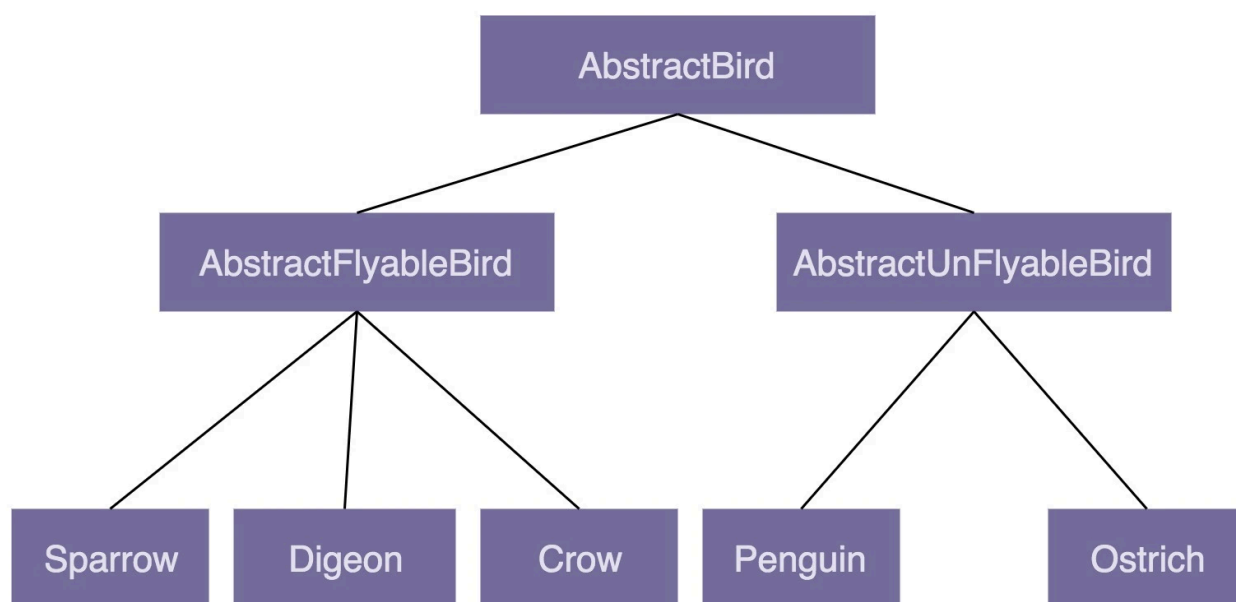
我们知道，大部分鸟都会飞，那我们可不可以在 `AbstractBird` 抽象类中，定义一个 `fly()` 方法呢？答案是否定的。尽管大部分鸟都会飞，但也有特例，比如鸵鸟就不会飞。鸵鸟继承具有 `fly()` 方法的父类，那鸵鸟就具有“飞”这样的行为，这显然不符合我们对现实世界中事物的认识。当然，你可能会说，我在鸵鸟这个子类中重写（`override`）`fly()` 方法，让它抛出 `UnsupportedMethodException` 异常不就可以了吗？具体的代码实现如下所示：

 复制代码

```
1 public class AbstractBird {
2     //...省略其他属性和方法...
3     public void fly() { //... }
4 }
5
6 public class Ostrich extends AbstractBird { //鸵鸟
7     //...省略其他属性和方法...
8     public void fly() {
9         throw new UnsupportedOperationException("I can't fly.");
10    }
11 }
```

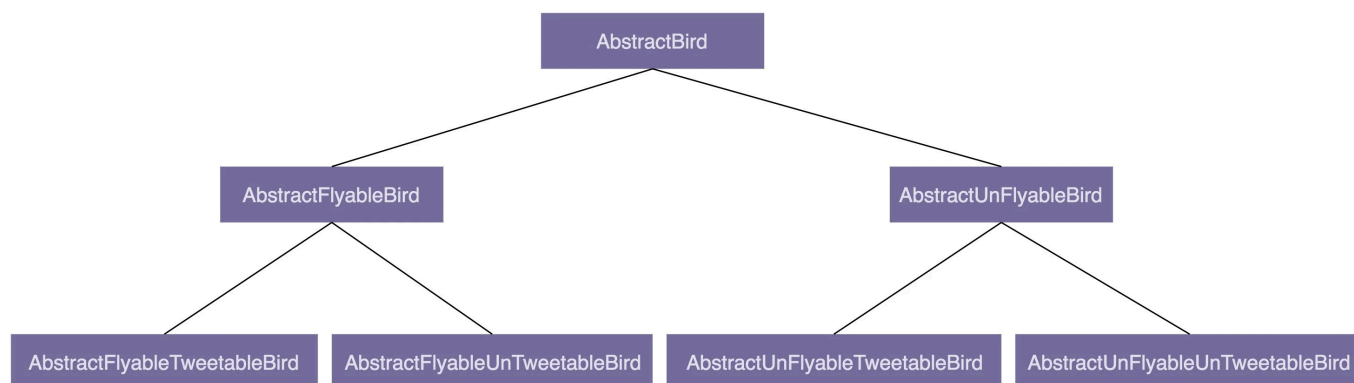
这种设计思路虽然可以解决问题，但不够优美。因为除了鸵鸟之外，不会飞的鸟还有很多，比如企鹅。对于这些不会飞的鸟来说，我们都需要重写 `fly()` 方法，抛出异常。这样的设计，一方面，徒增了编码的工作量；另一方面，也违背了我们之后要讲的最小知识原则（Least Knowledge Principle，也叫最少知识原则或者迪米特法则），暴露不该暴露的接口给外部，增加了类使用过程中被误用的概率。

你可能又会说，那我们再通过 `AbstractBird` 类派生出两个更加细分的抽象类：会飞的鸟类 `AbstractFlyableBird` 和不会飞的鸟类 `AbstractUnFlyableBird`，让麻雀、乌鸦这些会飞的鸟都继承 `AbstractFlyableBird`，让鸵鸟、企鹅这些不会飞的鸟，都继承 `AbstractUnFlyableBird` 类，不就可以了吗？具体的继承关系如下图所示：



从图中我们可以看出，继承关系变成了三层。不过，整体上来讲，目前的继承关系还比较简单，层次比较浅，也算是一种可以接受的设计思路。我们再继续加点难度。在刚刚这个场景中，我们只关注“鸟会不会飞”，但如果我们还关注“鸟会不会叫”，那这个时候，我们又该如何设计类之间的继承关系呢？

是否会飞？是否会叫？两个行为搭配起来会产生四种情况：会飞会叫、不会飞会叫、会飞不会叫、不会飞不会叫。如果我们继续沿用刚才的设计思路，那就需要再定义四个抽象类（AbstractFlyableTweetableBird、AbstractFlyableUnTweetableBird、AbstractUnFlyableTweetableBird、AbstractUnFlyableUnTweetableBird）。



如果我们还需要考虑“是否会下蛋”这样一个行为，那估计就要组合爆炸了。类的继承层次会越来越深、继承关系会越来越复杂。而这种层次很深、很复杂的继承关系，一方面，会导致代码的可读性变差。因为我们要搞清楚某个类具有哪些方法、属性，必须阅读父类的代码、父类的父类的代码……一直追溯到最顶层父类的代码。另一方面，这也破坏了类的封装特性，将父类的实现细节暴露给了子类。子类的实现依赖父类的实现，两者高度耦合，一旦父类代码修改，就会影响所有子类的逻辑。

总之，继承最大的问题就在于：继承层次过深、继承关系过于复杂会影响到代码的可读性和可维护性。这也是为什么我们不推荐使用继承。那刚刚例子中继承存在的问题，我们又该如何来解决呢？你可以先自己思考一下，再听我下面的讲解。

## 组合相比继承有哪些优势？

实际上，我们可以利用组合（composition）、接口、委托（delegation）三个技术手段，一块儿来解决刚刚继承存在的问题。

我们前面讲到接口的时候说过，接口表示具有某种行为特性。针对“会飞”这样一个行为特性，我们可以定义一个 Flyable 接口，只让会飞的鸟去实现这个接口。对于会叫、会下蛋这些行为特性，我们可以类似地定义 Tweetable 接口、EggLayable 接口。我们将这个设计思路翻译成 Java 代码的话，就是下面这个样子：

 复制代码

```
1 public interface Flyable {
2     void fly();
3 }
4 public interface Tweetable {
5     void tweet();
6 }
7 public interface EggLayable {
8     void layEgg();
9 }
10 public class Ostrich implements Tweetable, EggLayable { //鸵鸟
11     //... 省略其他属性和方法...
12     @Override
13     public void tweet() { //... }
14     @Override
15     public void layEgg() { //... }
16 }
```


```

17 public class Sparrow implements Flyable, Tweetable, EggLayable { //麻雀
18     //... 省略其他属性和方法...
19     @Override
20     public void fly() { //... }
21     @Override
22     public void tweet() { //... }
23     @Override
24     public void layEgg() { //... }
25 }

```

不过，我们知道，接口只声明方法，不定义实现。也就是说，每个会下蛋的鸟都要实现一遍 layEgg() 方法，并且实现逻辑是一样的，这就会导致代码重复的问题。那这个问题又该如何解决呢？

我们可以针对三个接口再定义三个实现类，它们分别是：实现了 fly() 方法的 FlyAbility 类、实现了 tweet() 方法的 TweetAbility 类、实现了 layEgg() 方法的 EggLayAbility 类。然后，通过组合和委托技术来消除代码重复。具体的代码实现如下所示：

 复制代码

```

1 public interface Flyable {
2     void fly();
3 }
4 public class FlyAbility implements Flyable {
5     @Override
6     public void fly() { //... }
7 }
8 //省略Tweetable/TweetAbility/EggLayable/EggLayAbility
9
10 public class Ostrich implements Tweetable, EggLayable { //鸵鸟
11     private TweetAbility tweetAbility = new TweetAbility(); //组合
12     private EggLayAbility eggLayAbility = new EggLayAbility(); //组合
13     //... 省略其他属性和方法...
14     @Override
15     public void tweet() {
16         tweetAbility.tweet(); // 委托
17     }
18     @Override
19     public void layEgg() {
20         eggLayAbility.layEgg(); // 委托
21     }
22 }

```



我们知道继承主要有三个作用：表示 is-a 关系，支持多态特性，代码复用。而这三个作用都可以通过其他技术手段来达成。比如 is-a 关系，我们可以通过组合和接口的 has-a 关系来替代；多态特性我们可以利用接口来实现；代码复用我们可以通过组合和委托来实现。所以，从理论上讲，通过组合、接口、委托三个技术手段，我们完全可以替换掉继承，在项目中不用或者少用继承关系，特别是一些复杂的继承关系。

## 如何判断该用组合还是继承？

尽管我们鼓励多用组合少用继承，但组合也并不是完美的，继承也并非一无是处。从上面的例子来看，继承改写成组合意味着要做更细粒度的类的拆分。这也就意味着，我们要定义更多的类和接口。类和接口的增多也就或多或少地增加代码的复杂程度和维护成本。所以，在实际的项目开发中，我们还是要根据具体的情况，来具体选择该用继承还是组合。

如果类之间的继承结构稳定（不会轻易改变），继承层次比较浅（比如，最多有两层继承关系），继承关系不复杂，我们就可以大胆地使用继承。反之，系统越不稳定，继承层次很深，继承关系复杂，我们就尽量使用组合来替代继承。

除此之外，还有一些设计模式会固定使用继承或者组合。比如，装饰者模式（decorator pattern）、策略模式（strategy pattern）、组合模式（composite pattern）等都使用了组合关系，而模板模式（template pattern）使用了继承关系。

前面我们讲到继承可以实现代码复用。利用继承特性，我们把相同的属性和方法，抽取出来，定义到父类中。子类复用父类中的属性和方法，达到代码复用的目的。但是，有的时候，从业务含义上，A 类和 B 类并不一定具有继承关系。比如，Crawler 类和 PageAnalyzer 类，它们都用到了 URL 拼接和分割的功能，但并不具有继承关系（既不是父子关系，也不是兄弟关系）。仅仅为了代码复用，生硬地抽象出一个父类出来，会影响到代码的可读性。如果不熟悉背后设计思路的同事，发现 Crawler 类和 PageAnalyzer 类继承同一个父类，而父类中定义的却只是 URL 相关的操作，会觉得这个代码写得莫名其妙，理解不了。这个时候，使用组合就更加合理、更加灵活。具体的代码实现如下所示：

 复制代码


```
1 public class Url {  
2     //...省略属性和方法  
3 }
```

```

4
5 public class Crawler {
6     private Url url; // 组合
7     public Crawler() {
8         this.url = new Url();
9     }
10    //...
11 }
12
13 public class PageAnalyzer {
14     private Url url; // 组合
15     public PageAnalyzer() {
16         this.url = new Url();
17     }
18    //..
19 }

```

还有一些特殊的场景要求我们必须使用继承。如果你不能改变一个函数的入参类型，而入参又非接口，为了支持多态，只能采用继承来实现。比如下面这样一段代码，其中 FeignClient 是一个外部类，我们没有权限去修改这部分代码，但是我们能重写这个类在运行时执行的 encode() 函数。这个时候，我们只能采用继承来实现了。

 复制代码

```

1 public class FeignClient { // Feign Client框架代码
2     //...省略其他代码...
3     public void encode(String url) { //... }
4 }
5
6 public void demofunction(FeignClient feignClient) {
7     //...
8     feignClient.encode(url);
9     //...
10 }
11
12 public class CustomizedFeignClient extends FeignClient {
13     @Override
14     public void encode(String url) { //...重写encode的实现...}
15 }
16
17 // 调用
18 FeignClient client = new CustomizedFeignClient();
19 demofunction(client);

```

尽管有些人说，要杜绝继承，100% 用组合代替继承，但是我的观点没那么极端！之所以“多用组合少用继承”这个口号喊得这么响，只是因为，长期以来，我们过度使用继承。还是那句话，组合并不完美，继承也不是一无是处。只要我们控制好它们的副作用、发挥它们各自的优势，在不同的场合下，恰当地选择使用继承还是组合，这才是我们所追求的境界。

## 重点回顾

到此，今天的内容就讲完了。我们一块儿来回顾一下，你需要重点掌握的知识点。

### 1. 为什么不推荐使用继承？

继承是面向对象的四大特性之一，用来表示类之间的 is-a 关系，可以解决代码复用的问题。虽然继承有诸多作用，但继承层次过深、过复杂，也会影响到代码的可维护性。在这种情况下，我们应该尽量少用，甚至不用继承。

### 2. 组合相比继承有哪些优势？

继承主要有三个作用：表示 is-a 关系，支持多态特性，代码复用。而这三个作用都可以通过组合、接口、委托三个技术手段来达成。除此之外，利用组合还能解决层次过深、过复杂的继承关系影响代码可维护性的问题。

### 3. 如何判断该用组合还是继承？

尽管我们鼓励多用组合少用继承，但组合也并不是完美的，继承也并非一无是处。在实际的项目开发中，我们还是要根据具体的情况，来选择该用继承还是组合。如果类之间的继承结构稳定，层次比较浅，关系不复杂，我们就可以大胆地使用继承。反之，我们就尽量使用组合来替代继承。除此之外，还有一些设计模式、特殊的应用场景，会固定使用继承或者组合。

## 课堂讨论

我们在基于 MVC 架构开发 Web 应用的时候，经常会在数据库层定义 Entity，在 Service 业务层定义 BO（Business Object），在 Controller 接口层定义 VO（View Object）。大部分



情况下，Entity、BO、VO 三者之间的代码有很大重复，但又不完全相同。我们该如何处理 Entity、BO、VO 代码重复的问题呢？

欢迎在留言区写下你的答案，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

### AI智能总结

面向对象编程中，组合优于继承的设计原则是为了解决继承层次过深、过复杂导致的可维护性和可读性问题。文章通过举例说明了继承在表示is-a关系时可能存在的问题，以及通过组合、接口、委托三个技术手段来替代继承的优势。通过组合和接口的has-a关系来替代is-a关系，利用接口实现多态特性，通过组合和委托实现代码复用，完全可以替换掉继承。因此，在项目中应该尽量避免过深、过复杂的继承关系，特别是一些复杂的继承关系。文章通过具体的代码实现和实际案例，详细解释了为何要多用组合少用继承，以及如何决定该用组合还是继承。

文章还提到了如何判断该用组合还是继承，指出在实际项目开发中，应根据具体情况选择使用继承还是组合。稳定的继承结构、浅层次、不复杂的关系可以使用继承，而不稳定、深层次、复杂的关系则应尽量使用组合。此外，特殊的设计模式和应用场景也会固定使用继承或组合。最后，总结了为什么不推荐使用继承、组合相比继承的优势以及如何处理代码重复的问题。

总的来说，本文深入浅出地解释了组合优于继承的设计原则，通过具体案例和技术手段的对比，使读者能够清晰地理解何时该使用组合，何时该使用继承，以及它们各自的优势和劣势。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

## 全部留言 (228)

最新 精选



探索无止境

2019-11-25

我个人感觉VO和BO都会采用组合entity的方式，老师是否可以在下一节课聊聊上节课留下的思考题，您的处理方式？

作者回复：抽空集中答疑一下吧

共 30 条评论 >

168



Cy23

2019-11-25

打卡✓

继承的层次过深带来的缺点看明白了，组合和委托不太理解，回头又好好看了看代码，视乎理解了，希望不要忘记了。也不知道是否理解对了？

记下自己的体会：

组合——类中包含其他实现类，感觉就是把大的功能分成多个类来实现，然后再根据需要组合进来使用。

委托——类中实现接口方法的时，把具体的实现方法调用其他类中的方法处理，也就是委托给别人（被委托者）帮他写好具体的实现。

作者回复: 理解的没问题！

共 2 条评论 >

👍 32



CC

2019-11-25

希望作者能在课程末尾梳理下上一节课的课后习题，或者集中点评下大家的留言。感谢

作者回复: 可以的...

共 2 条评论 >

👍 23



Hua100

2020-07-26

请教一下，java8之后接口可以有default，那是不是就可以不需要使用组合+接口+委托了呢？是不是只需要接口+default方法就可以了？类似这样：

```

public interface Fly {
    default void fly(){
        // 具体操作
    }
}

// Tweetable, Eggable同理
public class Ostrich implements Tweetable, Eggable {
    public void egg(){
        Eggable.super.egg();
    }
    // tweet同理

```

}

不知道我的理解对不对，我感觉这样就没必要用三个技术结合的方式。求大佬解答。

作者回复: 是的，你理解的没错。但是，并不是所有的语言中，接口都支持默认实现这个特性。

共 2 条评论 >

👍 14



**ANYI**

2019-11-26

这个课堂讨论，争哥啥时候可以给大家讲解下，这个貌似都是大家比较关注的点， 😊

作者回复: 恩恩，好的，抽空答疑一下

共 3 条评论 >

👍 3



**Miaoze**

2019-11-27

老师，针对你举鸟的例子：使用组合 + 接口 + 委托 代替继承的例子。如果在Spring中，怎么通过接口（自动注入）调用Ostrich鸵鸟的实现方法（下蛋和叫）？Tweetable和EggLayable接口各定义一个接口变量由鸵鸟初始化？这样感觉雍余了。

请解答一下？

作者回复: 你的意思是通过接口调用两个方法对吧，如果实在有这个需求，可以重新定义一个接口，extends另外两个接口。不过，你说的spring注入的情况，完全可以不依赖接口呀，直接依赖实现类也可以的。



👍 2



**小先生**

2019-11-25

请问老师，好多类想要拥有相同的一个属性，考虑到代码复用性，是否只能继承啦

作者回复: 也可以用组合的，组合也能解决复用的问题



👍 2



**欠债太多**

2019-11-25

我们现在采用entity实现，VO和BO都去继承它，减少代码重复。看了专栏后，我认为可以通过讲VO和BO组合成Entity实现，不知道这样做，是不是合适

作者回复: 好像不合适，怎么通过vo和bo组合成entity呢？



Jessica

2019-12-29

老师能不能加餐集中解答一周的问题，有些老师提问的刚好我们在项目中也思考过，就是没找到很好的解决方案

作者回复: 可以的 等后面有时间了吧 年底比较忙

共 2 条评论 >



西电

2020-07-15

请教一下，如果是按照功能分成多个基类，然后再按需继承。

比如将大的基类分成，Fly类，Tweet类，Egg类三个类。

鸵鸟就只继承Tweet类，Egg类

这样也可以实现接口，组合，委托的效果啊，请问一下这样有什么坏处呢？

作者回复: 有些语言不支持多重继承，而且从语义上也不对，什么是Fly类啊，飞是行为，更适合用接口来表达，不适合作为类吧

