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？

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。

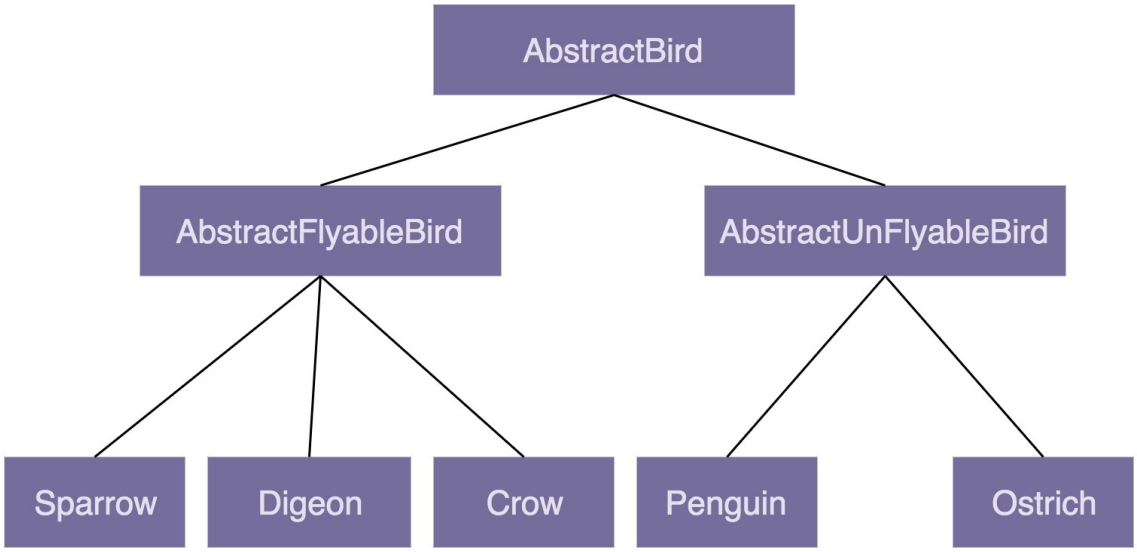
我们知道，大部分鸟都会飞，那我们可不可以在AbstractBird抽象类中，定义一个fly()方法呢？答案是否定的。尽管大部分鸟都会飞，但也有特例，比如鸵鸟就不会飞。鸵鸟继承具有fly()方法的父类，那鸵鸟就具有“飞”这样的行为，这显然不符合我们对现实世界中事物的认识。当然，你可能会说，我在鸵鸟这个子类中重写（override）fly()方法，让它抛出UnSupportedMethodException异常不就可以了吗？具体的代码实现如下所示：

```
public class AbstractBird {
  //...省略其他属性和方法...
  public void fly() { //... }
}

public class Ostrich extends AbstractBird { //鸵鸟
  //...省略其他属性和方法...
  public void fly() {
    throw new UnSupportedMethodException("I can't fly.'");
  }
}
```

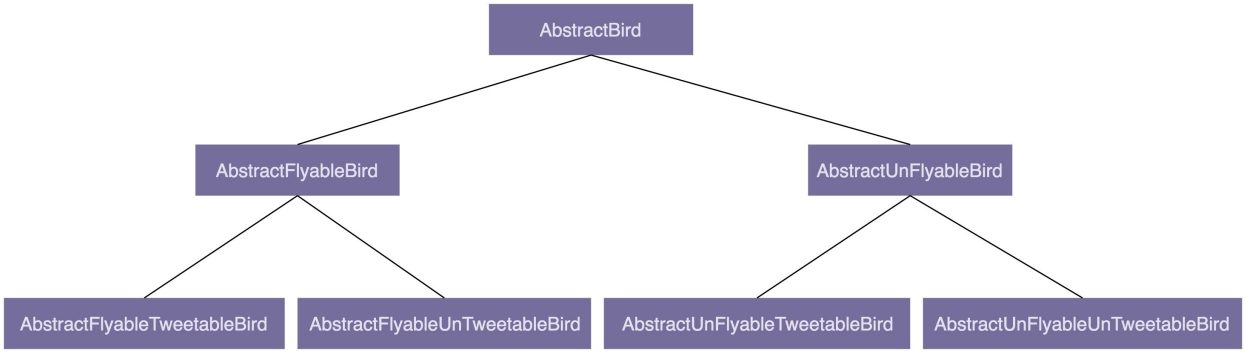
这种设计思路虽然可以解决问题，但不够优美。因为除了鸵鸟之外，不会飞的鸟还有很多，比如企鹅。对于这些不会飞的鸟来说，我们都需要重写fly()方法，抛出异常。这样的设计，一方面，徒增了编码的工作量；另一方面，也违背了我们之后要讲的最小知识原则（Least Knowledge Principle，也叫最少知识原则或者迪米特法则），暴露不该暴露的接口给外部，增加了类使用过程中被误用的概率。

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：

```
public interface Flyable {
  void fly();
}
public interface Tweetable {
  void tweet();
}
public interface EggLayable {
  void layEgg();
}
public class Ostrich implements Tweetable, EggLayable {//鸵鸟
  //... 省略其他属性和方法...
  @Override
  public void tweet() { //... }
  @Override
  public void layEgg() { //... }
}
public class Sparrow impelents Flayable, Tweetable, EggLayable {//麻雀
  //... 省略其他属性和方法...
  @Override
  public void fly() { //... }
  @Override
  public void tweet() { //... }
  @Override
  public void layEgg() { //... }
}
```

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：

```
public interface Flyable {
  void fly();
}
public class FlyAbility implements Flyable {
  @Override
  public void fly() { //... }
}
//省略Tweetable/TweetAbility/EggLayable/EggLayAbility

public class Ostrich implements Tweetable, EggLayable {//鸵鸟
  private TweetAbility tweetAbility = new TweetAbility(); //组合
  private EggLayAbility eggLayAbility = new EggLayAbility(); //组合
  //... 省略其他属性和方法...
  @Override
  public void tweet() {
```

```
    tweetAbility.tweet(); // 委托
  }
  @Override
  public void layEgg() {
    eggLayAbility.layEgg(); // 委托
  }
}
```

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：

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

public class Crawler {
  private Url url; // 组合
  public Crawler() {
    this.url = new Url();
  }
  //...
}

public class PageAnalyzer {
  private Url url; // 组合
```

```
  public PageAnalyzer() {
    this.url = new Url();
  }
  //..
}
```

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，但是我们希望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。

```
public class FeignClient { // feighn client框架代码
  //...省略其他代码...
  public void encode(String url) { //... }
}

public void demofunction(FeignClient feignClient) {
  //...
  feignClient.encode(url);
  //...
}

public class CustomizedFeignClient extends FeignClient {
  @Override
  public void encode(String url) { //...重写encode的实现...}
}

// 调用
FeignClient client = new CustomizedFeignClient();
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。

精选留言：

- 探索无止境 2019-11-25 01:22:55
  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课聊聊上节课留下的思考题，您的处理方式？ [67赞]

- Paul Shan 2019-11-25 05:09:57
  我的观点比较极端，用接口，组合和委托代替继承。原因如下：
  1. 人无法预知未来，现在比较稳定的类继承关系将来未必稳定。
  2.两种设计之间的选择耗费资源，每次都要为这个问题拿捏一下，甚至争论一下，不如把争论放在业务逻辑的实现上。
  3.相对于接口+组合+委托增加的复杂度，代码统一成接口+组合+委托带来的好处更多，利于阅读和交流，毕竟读代码的次数大于写的次数，读一种类型的代码的难度远低于读两种类型。
  4.新的编程语言让接口+组合+委托变得容易，例如Kotlin就有专门的语法糖支持，消除了很多模板代码。
  5.接口+组合+委托符合矢量化思想，那就是将物体特征分成不同的维度，每个维度独立变化。继承则是将物体分类，抽取共性，处理共性，操作的灵活性大打折扣，毕竟现实中的物体特征多，共性少。 [37赞]

- 守拙 2019-11-25 16:39:41
  课堂讨论answer:
  Entity， Bo， Vo三者之间，显然并不存在 is-a关系，首先排除使用继承。

  其次三者间也并非是严格的has-a关系，half measure之一是考虑使用组合（composition） + 委托（delegation）的方式解决代码重复的问题，但并不是我心中的最佳答案.

  我的答案是不解决三者间的代码重复问题。Value Class就只是Value Class, 代码重复并不是业务上的代码重复,那就让它重复吧.

  [10赞]

- Geek 2019-11-25 01:16:56
  打卡✔

看完之后有种感觉，我们平常写的spring的依赖注入这种形式，是不是就是跟组合，委托这种模式啊 [10赞]

- Shanks 2019-11-25 22:11:29
  希望，评论区能增加一个可选开关，“只看作者回复”的评论「(*/ω＼*)」 [4赞]

- 睡觉zzz 2019-11-25 10:47:11
  GO完全摒弃了继承，在语法上只有组合，接口之间也可以组合(这也是官方鼓励的做法)。 [3赞]

- 沧月丶下 2019-11-25 10:09:17
  public class FeignClient { // feighn client框架代码

  feighn -> feign 勘误~ [3赞]

- 花儿少年 2019-11-26 00:33:59
  VO,BO,DO表示什么前面都说过了，我觉得得换一个思路去看待这种模型转换的问题。
  这里我们将BO看做ddd里面的核心域中的实体。那么这个对象的变化应该对VO或者DO隐藏起来，VO是对外的模型，为什么需要感知到内部业务的变化，DO是具体的存储方式，这是由实现决定的，在业务逻辑中也不应该关心。重要的是隔离，让这三者独立变化。
  所以我的结论是，既不应该用继承，也不应该使用组合，使用防腐层，模型转换层隔离这种变化才是最好的。
  但是实际上在很多业务中BO和DO是差不多的，于是就混用了，在业务不复杂的时候，也没太大关系。业务运行的很好，也不难理解。

  追求完美，却不可能处处完美。 [2赞]

- Jxin 2019-11-25 13:24:46
  1.bo vo和entity三个命名在现在面向服务而非页面的后端编程，并不合适。
  2.这里最好用组合。entity是最小的实体单元，bo可能面对多个entity聚合，vo可能面对多个bo聚合，这种场景下，显然组合更适合。虽然也存在entity和bo一对一的场景，或者bo中只有一个主entity的场景，这种场景用继承倒也不为过。但是，为了套路单一，减少阅读思考，统一组合便是，没必要再引入继承。

  3.老项目里面，代码已经高度耦合，而且不是面相接口写的代码，那么整体改动成本会很大。这种情况下用继承实现多态我觉得挺合适。

  4.java1.8提供接口的方法默认实现后，我觉得继承的处境真的挺尴尬，新项目反正能用继承实现的用组合也可以。所以除非父子关系特别明显（继承不深其实比组合直观），不然没什么必要用继承了。 [2赞]

- tt 2019-11-25 10:30:11
  谈谈对下面一段话的理解：

  “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。”

  理解或总结如下：

1、“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”，我的理解为：is-a意味着has-multi-a's或者has-all-needed-a's。故而需要实现多个接口，而接口抽象的是操作或者方法而非数据（数据和方法的抽象由抽象类来完成），所以具体的操作要由被组合进来的类对象来完成，站在类间关系的角度来看，外部类和被组合类之间的关系被称为“委托”。

2、这里面，被组合类的代码被抽象到了接口中，或者反过来说接口的具体操作下沉到了被组合类中，这就是“代码复用我们可以通过组合和委托来实现”的含义，代码被不同的被组合类“分门（类）别类”的复用了。

3、“多态特性我们可以利用接口来实现”，因为接口代表了某种契约，而多态就是用子类代替父类。只要实现了某种接口，按照契约，自然就可以在某些方面或某种程度上代替父类。所以我觉得接口是“更细粒度更多控制的更有节制的继承”。

回到本课的问题。

之前的课说到VO，BO，Entity是典型的面向过程的编码，里面基本都是数据，没有方法。那么自然不可以用接口来减少代码的重复，只能用继承了。

但是MVC的结构，我理解它是一种分层客户端服务器架构，Layered Client-Server，每一层为其之上的层服务，并使用其之下的层所提供的服务。为了减少层之间的耦合，必要的重复是可以的。 [2赞]

- 傲慢与偏执， 2019-11-25 08:43:28
  我只有在该类需要更细化详情信息的时候会组合详情类的list 看了这节课后 受益匪浅 [2赞]

- 李湘河 2019-11-25 08:37:40
  现代军事武器中的开发都在追求模块化开发，这样装备之间通用性更强，战损时随时可以替换掉损坏的模块，这样又可以重新作战，当要增强坦克某一部分的性能时，仅改进对应的模块就行，感觉很像组合的思想。就像文中说的，对于结构稳定，层次浅的地方完全可以用继承，或者说可以局部用继承，比如VO层，对于用户检验，分页等都可以抽象出来 [2赞]

- 辣么大 2019-11-25 07:35:24
  Entity，也称为DO （Data Object），与数据库表结构一一对应，到过DAO层向上传输对象，应独立为一个类。

  BO，VO 可以采用继承或者组合的方式，复用DO的代码。

  谨慎使用继承方式来进行扩展，不得已使用的话，必须符合里氏替换原则，父类能够出现的地方子类一定能够出现。 [2赞]

- Sinderlar 2019-11-25 00:45:14
  我的个人感觉,等待高手更好的回答//
  Entity在VO,BO中基本上都是一模一样的,使用组合把Entity引用进来,然后在BO,VO中创建各自独特的属性/ [2赞]

- 黄林晴 2019-11-25 00:19:36
  打卡✔
  老师好，今天刚用继承优化了代码臃肿的问题，但是感觉好奇怪，请老师指导:
  所有的消息都会先到一个A类中，在A类中，根据消息类型，比如类型1 2 3 4去处理不同的业务，每一类的业务都需要处理对应数据，原本随着消息类型的增加不断往这个A类中扩展代码，导致不好维护，所以我对每个业务模型建对应的类继承这个A类，在A类中将消息转给对应类去处理，其实new一个类 将所有参

数传过去也可以，但是因为参数太多太多不美观，所以使用了继承，但回过头来想，我的继承只是被动使用的，好像和继承的原理相违背 [2赞]

- Monday 2019-11-25 23:29:10
  1、我们现在的项目只有Entity与VO，而且两者之间互不相关，确实有太多的属性和方法重复了。
  2、思考题：学习了今天内容，个人觉得可以使用委托和组合的方法，将Entity与VO共有属性抽象出来一个类，然后Entity与VO来组合此类。 [1赞]

- 啦啦啦 2019-11-25 16:42:46

```
<?php
//叫
interface jiaoable
{
public function jiao();
}
//飞
interface flyable
{
public function fly();
}
//下单
interface eggable
{
public function egg();
}

//叫的实现类
class jiaoablity implements jiaoable
{
public function jiao()
{
echo 'jiao';
}
}

//鸵鸟类
class tuoniao implements jiaoable,eggable
{
private $jiaoobj;

public function __construct()
{
$this->jiaoobj = new jiaoablity();
}

public function jiao()
{
$this->jiaoobj->jiao();
}
```

```
public function egg()
{
echo '我是鸵鸟我会下单';
}
}

//麻雀类
class maque implements jiaoable, flyable
{
private $jiaoobj;

public function __construct()
{
$this->jiaoobj = new jiaoablity();
}

public function jiao()
{
$this->jiaoobj->jiao();
}

public function fly()
{
echo '我是麻雀我会飞';
}
}
```
[1赞]

- 辣么大 2019-11-25 09:35:52
  很多同学提出复用Entity（DO），我有不同意见：若修改DO，可能会影响到BO和VO。
  我们都知道DO对应数据表，如PersonDO类有id，age，name。
  若现在需求改变，age要从政府系统获取，原有的Person表要删除age字段，相应的DO类就要修改，UI仍然显示person.age。BO、VO有如果使用了DO就会受到影响。
  为了降低影响，BO,VO考虑使用PersonDTO。
  上面的例子中DTO中保留Person.age属性，在Service层中将DO转换为DTO，转换时PersonDTO.age从其他系统获取。
  这样虽然增加了代码量，对DAO层的修改影响降到最低。 [1赞]

- 小晏子 2019-11-25 08:46:20
  因为Entity， BO， VO都是描述对象的，只不过是用于不同目的的对象，这些对象之间会有很多重复的元素定义，针对这样的重复定义，我倒是感觉可以使用继承，将重复的那些元素都提取出来作为父类，然后entity，VO，BO去继承这个父类，在实现自己独有的元素。而接口主要是针对不同能能的复用，用在E，B，V身上并不合适。 [1赞]

- 未设置 2019-11-25 08:14:58
  希望作者能在课程末尾梳理下上一节课程的课后习题，或者集中点评下大家的留言。感谢 [1赞]