07-理论四：哪些代码设计看似是面向对象，实际是面向过程的？

上一节课，我们提到，常见的编程范式或者说编程风格有三种，面向过程编程、面向对象编程、函数式编程，而面向对象编程又是这其中最主流的编程范式。现如今，大部分编程语言都是面向对象编程语言，大部分软件都是基于面向对象编程这种编程范式来开发的。

不过，在实际的开发工作中，很多同学对面向对象编程都有误解，总以为把所有代码都塞到类里，自然就是在进行面向对象编程了。实际上，这样的认识是不正确的。有时候，从表面上看似是面向对象编程风格的代码，从本质上看却是面向过程编程风格的。

所以，今天，我结合具体的代码实例来讲一讲，有哪些看似是面向对象，实际上是面向过程编程风格的代码，并且分析一下，为什么我们很容易写出这样的代码。最后，我们再一起辩证思考一下，面向过程编程是否就真的无用武之地了呢？是否有必要杜绝在面向对象编程中写面向过程风格的代码呢？

好了，现在，让我们正式开始今天的学习吧！

哪些代码设计看似是面向对象，实际是面向过程的？

在用面向对象编程语言进行软件开发的时候，我们有时候会写出面向过程风格的代码。有些是有意为之，并无不妥；而有些是无意为之，会影响到代码的质量。下面我就通过三个典型的代码案例，给你展示一下，什么样的代码看似是面向对象风格，实际上是面向过程风格的。我也希望你通过对这三个典型例子的学习，能够做到举一反三，在平时的开发中，多留心一下自己编写的代码是否满足面向对象风格。

1.滥用getter、setter方法

在之前参与的项目开发中，我经常看到，有同事定义完类的属性之后，就顺手把这些属性的getter、setter方法都定义上。有些同事更加省事，直接用IDE或者Lombok插件（如果是Java项目的话）自动生成所有属性的getter、setter方法。

当我问起，为什么要给每个属性都定义getter、setter方法的时候，他们的理由一般是，为了以后可能会用到，现在事先定义好，类用起来就更加方便，而且即便用不到这些getter、setter方法，定义上它们也无伤大雅。

实际上，这样的做法我是非常不推荐的。它违反了面向对象编程的封装特性，相当于将面向对象编程风格退化成了面向过程编程风格。我通过下面这个例子来给你解释一下这句话。

```
public class ShoppingCart {
  private int itemsCount;
  private double totalPrice;
  private List<ShoppingCartItem> items = new ArrayList<>();

  public int getItemsCount() {
    return this.itemsCount;
  }

  public void setItemsCount(int itemsCount) {
    this.itemsCount = itemsCount;
  }

  public double getTotalPrice() {
    return this.totalPrice;
```

```
  }

  public void setTotalPrice(double totalPrice) {
    this.totalPrice = totalPrice;
  }

  public List<ShoppingCartItem> getItems() {
    return this.items;
  }

  public void addItem(ShoppingCartItem item) {
    items.add(item);
    itemsCount++;
    totalPrice += item.getPrice();
  }
  // ...省略其他方法...
}
```

在这段代码中，ShoppingCart是一个简化后的购物车类，有三个私有（private）属性：itemsCount、totalPrice、items。对于itemsCount、totalPrice两个属性，我们定义了它们的getter、setter方法。对于items属性，我们定义了它的getter方法和addItem()方法。代码很简单，理解起来不难。那你有没有发现，这段代码有什么问题呢？

我们先来看前两个属性，itemsCount和totalPrice。虽然我们将它们定义成private私有属性，但是提供了public的getter、setter方法，这就跟将这两个属性定义为public公有属性，没有什么两样了。外部可以通过setter方法随意地修改这两个属性的值。除此之外，任何代码都可以随意调用setter方法，来重新设置itemsCount、totalPrice属性的值，这也会导致其跟items属性的值不一致。

而面向对象封装的定义是：通过访问权限控制，隐藏内部数据，外部仅能通过类提供的有限的接口访问、修改内部数据。所以，暴露不应该暴露的setter方法，明显违反了面向对象的封装特性。数据没有访问权限控制，任何代码都可以随意修改它，代码就退化成了面向过程编程风格的了。

看完了前两个属性，我们再来看items这个属性。对于items这个属性，我们定义了它的getter方法和addItem()方法，并没有定义它的setter方法。这样的设计貌似看起来没有什么问题，但实际上并不是。

对于itemsCount和totalPrice这两个属性来说，定义一个public的getter方法，确实无伤大雅，毕竟getter方法不会修改数据。但是，对于items属性就不一样了，这是因为items属性的getter方法，返回的是一个List集合容器。外部调用者在拿到这个容器之后，是可以操作容器内部数据的，也就是说，外部代码还是能修改items中的数据。比如像下面这样：

```
ShoppingCart cart = new ShoppCart();
...
cart.getItems().clear(); // 清空购物车
```

你可能会说，清空购物车这样的功能需求看起来合情合理啊，上面的代码没有什么不妥啊。你说得没错，需求是合理的，但是这样的代码写法，会导致itemsCount、totalPrice、items三者数据不一致。我们不应该将清空购物车的业务逻辑暴露给上层代码。正确的做法应该是，在ShoppingCart类中定义一个clear()方法，将清空购物车的业务逻辑封装在里面，透明地给调用者使用。ShoppingCart类的clear()方法的具体代

码实现如下：

```
public class ShoppingCart {
  // ...省略其他代码...
  public void clear() {
    items.clear();
    itemsCount = 0;
    totalPrice = 0.0;
  }
}
```

你可能还会说，我有一个需求，需要查看购物车中都买了啥，那这个时候，ShoppingCart类不得不提供items属性的getter方法了，那又该怎么办才好呢？

如果你熟悉Java语言，那解决这个问题的方法还是挺简单的。我们可以通过Java提供的Collections.unmodifiableList()方法，让getter方法返回一个不可被修改的UnmodifiableList集合容器，而这个容器类重写了List容器中跟修改数据相关的方法，比如add()、clear()等方法。一旦我们调用这些修改数据的方法，代码就会抛出UnsupportedOperationException异常，这样就避免了容器中的数据被修改。具体的代码实现如下所示。

```
public class ShoppingCart {
  // ...省略其他代码...
  public List<ShoppingCartItem> getItems() {
    return Collections.unmodifiableList(this.items);
  }
}

public class UnmodifiableList<E> extends UnmodifiableCollection<E>
                         implements List<E> {
  public boolean add(E e) {
    throw new UnsupportedOperationException();
  }
  public void clear() {
    throw new UnsupportedOperationException();
  }
  // ...省略其他代码...
}

ShoppingCart cart = new ShoppingCart();
List<ShoppingCartItem> items = cart.getItems();
items.clear();//抛出UnsupportedOperationException异常
```

不过，这样的实现思路还是有点问题。因为当调用者通过ShoppingCart的getItems()获取到items之后，虽然我们没法修改容器中的数据，但我们仍然可以修改容器中每个对象（ShoppingCartItem）的数据。听起来有点绕，看看下面这几行代码你就明白了。

```
ShoppingCart cart = new ShoppingCart();
cart.add(new ShoppingCartItem(...));
List<ShoppingCartItem> items = cart.getItems();
```

```
ShoppingCartItem item = items.get(0);
item.setPrice(19.0); // 这里修改了item的价格属性
```

这个问题该如何解决呢？我今天就不展开来讲了。在后面讲到设计模式的时候，我还会详细地讲到。当然，你也可以在留言区留言或者把问题分享给你的朋友，和他一起讨论解决方案。

getter、setter问题我们就讲完了，我稍微总结一下，在设计实现类的时候，除非真的需要，否则，尽量不要给属性定义setter方法。除此之外，尽管getter方法相对setter方法要安全些，但是如果返回的是集合容器（比如例子中的List容器），也要防范集合内部数据被修改的危险。

2.滥用全局变量和全局方法

我们再来看，另外一个违反面向对象编程风格的例子，那就是滥用全局变量和全局方法。首先，我们先来看，什么是全局变量和全局方法？

如果你是用类似C语言这样的面向过程的编程语言来做开发，那对全局变量、全局方法肯定不陌生，甚至可以说，在代码中到处可见。但如果你是用类似Java这样的面向对象的编程语言来做开发，全局变量和全局方法就不是很多见了。

在面向对象编程中，常见的全局变量有单例类对象、静态成员变量、常量等，常见的全局方法有静态方法。单例类对象在全局代码中只有一份，所以，它相当于一个全局变量。静态成员变量归属于类上的数据，被所有的实例化对象所共享，也相当于一定程度上的全局变量。而常量是一种非常常见的全局变量，比如一些代码中的配置参数，一般都设置为常量，放到一个Constants类中。静态方法一般用来操作静态变量或者外部数据。你可以联想一下我们常用的各种Utils类，里面的方法一般都会定义成静态方法，可以在不用创建对象的情况下，直接拿来使用。静态方法将方法与数据分离，破坏了封装特性，是典型的面向过程风格。

在刚刚介绍的这些全局变量和全局方法中，Constants类和Utils类最常用到。现在，我们就结合这两个几乎在每个软件开发中都会用到的类，来深入探讨一下全局变量和全局方法的利与弊。

我们先来看一下，在我过去参与的项目中，一种常见的Constants类的定义方法。

```
public class Constants {
  public static final String MYSQL_ADDR_KEY = "mysql_addr";
  public static final String MYSQL_DB_NAME_KEY = "db_name";
  public static final String MYSQL_USERNAME_KEY = "mysql_username";
  public static final String MYSQL_PASSWORD_KEY = "mysql_password";

  public static final String REDIS_DEFAULT_ADDR = "192.168.7.2:7234";
  public static final int REDIS_DEFAULT_MAX_TOTAL = 50;
  public static final int REDIS_DEFAULT_MAX_IDLE = 50;
  public static final int REDIS_DEFAULT_MIN_IDLE = 20;
  public static final String REDIS_DEFAULT_KEY_PREFIX = "rt:";

  // ...省略更多的常量定义...
}
```

在这段代码中，我们把程序中所有用到的常量，都集中地放到这个Constants类中。不过，定义一个如此大

而全的Constants类，并不是一种很好的设计思路。为什么这么说呢？原因主要有以下几点。

首先，这样的设计会影响代码的可维护性。

如果参与开发同一个项目的工程师有很多，在开发过程中，可能都要涉及修改这个类，比如往这个类里添加常量，那这个类就会变得越来越大，成百上千行都有可能，查找修改某个常量也会变得比较费时，而且还会增加提交代码冲突的概率。

其次，这样的设计还会增加代码的编译时间。

当Constants类中包含很多常量定义的时候，依赖这个类的代码就会很多。那每次修改Constants类，都会导致依赖它的类文件重新编译，因此会浪费很多不必要的编译时间。不要小看编译花费的时间，对于一个非常大的工程项目来说，编译一次项目花费的时间可能是几分钟，甚至几十分钟。而我们在开发过程中，每次运行单元测试，都会触发一次编译的过程，这个编译时间就有可能会影响到我们的开发效率。

最后，这样的设计还会影响代码的复用性。

如果我们要在另一个项目中，复用本项目开发的某个类，而这个类又依赖Constants类。即便这个类只依赖Constants类中的一小部分常量，我们仍然需要把整个Constants类也一并引入，也就引入了很多无关的常量到新的项目中。

那如何改进Constants类的设计呢？我这里有两种思路可以借鉴。

第一种是将Constants类拆解为功能更加单一的多个类，比如跟MySQL配置相关的常量，我们放到MysqlConstants类中；跟Redis配置相关的常量，我们放到RedisConstants类中。当然，还有一种我个人觉得更好的设计思路，那就是并不单独地设计Constants常量类，而是哪个类用到了某个常量，我们就把这个常量定义到这个类中。比如，RedisConfig类用到了Redis配置相关的常量，那我们就直接将这些常量定义在RedisConfig中，这样也提高了类设计的内聚性和代码的复用性。

讲完了Constants类，我们再来讨论一下Utils类。首先，我想问你这样一个问题，我们为什么需要Utils类？Utils类存在的意义是什么？希望你先思考一下，然后再来看我下面的讲解。

实际上，Utils类的出现是基于这样一个问题背景：如果我们有两个类A和B，它们要用到一块相同的功能逻辑，为了避免代码重复，我们不应该在两个类中，将这个相同的功能逻辑，重复地实现两遍。这个时候我们该怎么办呢？

我们在讲面向对象特性的时候，讲过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，而父类中定义的却是URL相关的操作，会觉得这个代码写得莫名其妙，理解不了。

既然继承不能解决这个问题，我们可以定义一个新的类，实现URL拼接和分割的方法。而拼接和分割两个方法，不需要共享任何数据，所以新的类不需要定义任何属性，这个时候，我们就可以把它定义为只包含静态方法的Utils类了。

实际上，只包含静态方法不包含任何属性的Utils类，是彻彻底底的面向过程的编程风格。但这并不是说，我们就要杜绝使用Utils类了。实际上，从刚刚讲的Utils类存在的目的来看，它在软件开发中还是挺有用的，能解决代码复用问题。所以，这里并不是说完全不能用Utils类，而是说，要尽量避免滥用，不要不加思考地随意去定义Utils类。

在定义Utils类之前，你要问一下自己，你真的需要单独定义这样一个Utils类吗？是否可以把Utils类中的某些方法定义到其他类中呢？如果在回答完这些问题之后，你还是觉得确实有必要去定义这样一个Utils类，那就大胆地去定义它吧。因为即便在面向对象编程中，我们也并不是完全排斥面向过程风格的代码。只要它能为我们写出好的代码贡献力量，我们就可以适度地去使用。

除此之外，类比Constants类的设计，我们设计Utils类的时候，最好也能细化一下，针对不同的功能，设计不同的Utils类，比如FileUtils、IOUtils、StringUtils、UrlUtils等，不要设计一个过于大而全的Utils类。

3.定义数据和方法分离的类

我们再来看最后一种面向对象编程过程中，常见的面向过程风格的代码。那就是，数据定义在一个类中，方法定义在另一个类中。你可能会觉得，这么明显的面向过程风格的代码，谁会这么写呢？实际上，如果你是基于MVC三层结构做Web方面的后端开发，这样的代码你可能天天都在写。

传统的MVC结构分为Model层、Controller层、View层这三层。不过，在做前后端分离之后，三层结构在后端开发中，会稍微有些调整，被分为Controller层、Service层、Repository层。Controller层负责暴露接口给前端调用，Service层负责核心业务逻辑，Repository层负责数据读写。而在每一层中，我们又会定义相应的VO（View Object）、BO（Business Object）、Entity。一般情况下，VO、BO、Entity中只会定义数据，不会定义方法，所有操作这些数据的业务逻辑都定义在对应的Controller类、Service类、Repository类中。这就是典型的面向过程的编程风格。

实际上，这种开发模式叫作基于贫血模型的开发模式，也是我们现在非常常用的一种Web项目的开发模式。看到这里，你内心里应该有很多疑惑吧？既然这种开发模式明显违背面向对象的编程风格，为什么大部分Web项目都是基于这种开发模式来开发呢？

关于这个问题，我今天不打算展开讲解。因为它跟我们平时的项目开发结合得非常紧密，所以，更加细致、全面的讲解，我把它安排在面向对象实战环节里了，希望用两节课的时间，把这个问题给你讲透彻。

在面向对象编程中，为什么容易写出面向过程风格的代码？

我们在进行面向对象编程的时候，很容易不由自主地就写出面向过程风格的代码，或者说感觉面向过程风格的代码更容易写。这是为什么呢？

你可以联想一下，在生活中，你去完成一个任务，你一般都会思考，应该先做什么、后做什么，如何一步一步地顺序执行一系列操作，最后完成整个任务。面向过程编程风格恰恰符合人的这种流程化思维方式。而面向对象编程风格正好相反。它是一种自底向上的思考方式。它不是先去按照执行流程来分解任务，而是将任务翻译成一个一个的小的模块（也就是类），设计类之间的交互，最后按照流程将类组装起来，完成整个任务。我们在上一节课讲到了，这样的思考路径比较适合复杂程序的开发，但并不是特别符合人类的思考习惯。

除此之外，面向对象编程要比面向过程编程难一些。在面向对象编程中，类的设计还是挺需要技巧，挺需要一定设计经验的。你要去思考如何封装合适的数据和方法到一个类里，如何设计类之间的关系，如何设计类

之间的交互等等诸多设计问题。

所以，基于这两点原因，很多工程师在开发的过程，更倾向于用不太需要动脑子的方式去实现需求，也就不由自主地就将代码写成面向过程风格的了。

面向过程编程及面向过程编程语言就真的无用武之地了吗？

前面我们讲了面向对象编程相比面向过程编程的各种优势，又讲了哪些代码看起来像面向对象风格，而实际上是面向过程编程风格的。那是不是面向过程编程风格就过时了被淘汰了呢？是不是在面向对象编程开发中，我们就要杜绝写面向过程风格的代码呢？

前面我们有讲到，如果我们开发的是微小程序，或者是一个数据处理相关的代码，以算法为主，数据为辅，那脚本式的面向过程的编程风格就更适合一些。当然，面向过程编程的用武之地还不止这些。实际上，面向过程编程是面向对象编程的基础，面向对象编程离不开基础的面向过程编程。为什么这么说？我们仔细想想，类中每个方法的实现逻辑，不就是面向过程风格的代码吗？

除此之外，面向对象和面向过程两种编程风格，也并不是非黑即白、完全对立的。在用面向对象编程语言开发的软件中，面向过程风格的代码并不少见，甚至在一些标准的开发库（比如JDK、Apache Commons、Google Guava）中，也有很多面向过程风格的代码。

不管使用面向过程还是面向对象哪种风格来写代码，我们最终的目的还是写出易维护、易读、易复用、易扩展的高质量代码。只要我们能避免面向过程编程风格的一些弊端，控制好它的副作用，在掌控范围内为我们所用，我们就大可不用避讳在面向对象编程中写面向过程风格的代码。

重点回顾

今天的内容讲完了。让我们一块回顾一下，你应该掌握的重点内容。今天你要掌握的重点内容是三种违反面向对象编程风格的典型代码设计。

1.滥用getter、setter方法

在设计实现类的时候，除非真的需要，否则尽量不要给属性定义setter方法。除此之外，尽管getter方法相对setter方法要安全些，但是如果返回的是集合容器，那也要防范集合内部数据被修改的风险。

2.Constants类、Utils类的设计问题

对于这两种类的设计，我们尽量能做到职责单一，定义一些细化的小类，比如RedisConstants、FileUtils，而不是定义一个大而全的Constants类、Utils类。除此之外，如果能将这些类中的属性和方法，划分归并到其他业务类中，那是最好不过的了，能极大地提高类的内聚性和代码的可复用性。

3.基于贫血模型的开发模式

关于这一部分，我们只讲了为什么这种开发模式是彻彻底底的面向过程编程风格的。这是因为数据和操作是分开定义在VO/BO/Entity和Controler/Service/Repository中的。今天，你只需要掌握这一点就可以了。为什么这种开发模式如此流行？如何规避面向过程编程的弊端？有没有更好的可替代的开发模式？相关的更多问题，我们在面向对象实战篇中会一一讲解。

课堂讨论

今天课堂讨论的话题有两个，你可以选择一个熟悉的来发表观点。

1.今天我们讲到，用面向对象编程语言写出来的代码，不一定是面向对象编程风格的，有可能是面向过程编程风格的。相反，用面向过程编程语言照样也可以写出面向对象编程风格的代码。尽管面向过程编程语言可能没有现成的语法来支持面向对象的四大特性，但可以通过其他方式来模拟，比如在C语言中，我们可以利用函数指针来模拟多态。如果你熟悉一门面向过程的编程语言，你能聊一聊如何用它来模拟面向对象的四大特性吗？

2.看似是面向对象实际上是面向过程编程风格的代码有很多，除了今天我讲到的这三个，在你工作中，你还遇到过哪些其他情况吗？

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：

- 辣么大 2019-11-18 08:20:12
  不想往下看的请看第一句就好：贫血模型流行的原因：实现简单和上手快。

  具体解释慢慢看：

  ”贫血模型“的开发模式为什么会流行？

  1、实现简单。Object仅仅作为传递数据的媒介，不用考虑过多的设计方面，将核心业务逻辑放到service层，用Hibernate之类的框架一套，完美解决任务。
  2、上手快。使用贫血模式开发的web项目，新来的程序员看看代码就能“照猫画虎”干活了，不需要多高的技术水平。所以很多程序员干了几年，仅仅就会写CURD。
  3、一些技术鼓励使用贫血模型。例如J2EE Entity Beans，Hibernate等。

  总结：各种模型的好坏讨论一直不断，企业需要的是使用合适的技术把任务完成，从这个角度来说当下管用模型就是好模型。当然我们也要持开放的心态接受新的技术和思想，并结合业务的实际需要选择合适的技术。

  概念解释：

  贫血模型（Anemic Domain Model由
  MatinFowler提出）又称为失血模型，是指domain object仅有属性的getter/setter方法的纯数据类，将所有类的行为放到service层。原文他是这么说的“By pulling all the behavior out into services, however, you essentially end up with Transaction Scripts, and thus lose the advantages that the domain model can bring.”他的原文我放上来了，英文好的同学可以看看：https://martinfowler.com/bliki/AnemicDomainModel.html 。 我觉得他有点学者气太重，这篇博客他都不知道为啥贫血模型会流行（I don't know why this anti-pattern is so common）。 [26赞]

- 黄林晴 2019-11-18 00:36:16
  打卡
  看了今天的内容，发现自己三点都占了，😳
  遇到json数据使用Gsonformat转一下，默认生成所有get set方法，遇到统一使用的就会毫不犹豫定义工具类……，我有点怀疑自己是不是从未写过面相对象风格的代码 [17赞]

- 猫切切切切切 2019-11-18 06:07:07
  总的来说，使用面向对象风格编写的面向过程代码有如下特点：

  1. 使用了类，但是
  2. 要么完全没有封装（比如数据和操作分离的贫血模式）
  3. 要么破坏了封装（比如滥用 getter 或 setter）
  4. 要么完全没有抽象（大而全的 Constants 或 Utils 类）
  5. 要么封装或抽象不完全（比如类实例化后，或者子类继承后，需要自己管理其内部某些属性或状态）

  平时应该多留心代码是否存在上述特征。

  文中没有举封装或抽象不完全的例子，这里举一个。

  比如一个实现某种业务需求（如与某种类型设备通讯的应用协议）的 tcp 或 udp 服务器；

  实例化后还需要自己管理其协议相关的就绪状态（ready属性）；使你不得不对其再封装一层，并抽象其连接、断开等方法使其自动进行就绪状态的管理；

  每个继承都这么封装一遍，就会有大量重复的代码，而且其实类的实例化者或继承者并不需要也不应该关心就绪状态的管理，所以没有达到就绪状态管理的封装。

  这就是一种不完全的封装。 [14赞]

- 熊斌 2019-11-18 07:10:50
  我们的项目三点都占，造成这个局面的原因我认为有以下几点：
  1、习惯用IDE的代码生成插件
  2、团队整体设计水平有限
  3、基于mvc模式开发的 [8赞]

- 中年男子 2019-11-18 15:24:52
  先说问题2：看似面向对象实际面向过程的例子真是数不胜数了， 工作语言C/C++，90%是C++,大体上老师在文中已经提到了，其他的我暂时也没想起来，但是滥用面向对象继承特性的代码我真是看到了太多
  问题1：C中可以用struct 来实现class，只是访问控制权限都是public。类中的成员函数可以通过指向操作结构体的函数指针来实现，实现封装，需要绑定数据、函数、函数指针。可以创建函数指针表，构造函数设置函数指针指向正确的操作函数，函数指针表作为对象访问函数的接口。操作结构体的这些函数（成员函数）不像C++中能直接访问数据成员，需要显示的传递操作对象给成员函数。
  继承：在派生类中维护一个基类对象的指针。这样派生类可以访问基类对象的数据。
  多态：在基类中维护一个派生类对象的指针。这样基类可以访问派生类对象的数据。
  C++中的多态，有一个对象销毁的问题。基类的析构函数必须是虚函数
  在C中，这可以通过使基类的删除函数指针指向派生类的删除函数，因为派生类的删除函数清楚派生类的数据和基类的数据 [6赞]

- Jeff.Smile 2019-11-18 09:17:35
  有种上帝视角看自己的感觉！ [5赞]

- Daiver 2019-11-18 01:24:00
  啊，写了这么久的MVC，竟然是面向过程编程。 [5赞]

- 编程界的小学生 2019-11-18 10:05:55

1.get set 这个很好理解，但是我有很多疑问，比如有的属性理论上来讲不该添加set方法，那我怎么对他进行属性拷贝？比如两个vo进行拷贝属性值，还有作为接口参数，spring又怎么给他赋值？
2.看完贫血模式那个知识点后，我懵了，我甚至不知道怎么才能写出面向对象的代码了，如果数据和业务不分离的话，那比如我多个业务接口需要同一份数据，难道要定义多份吗？我有点懵了😟😟😟 [3赞]

作者回复2019-11-19 10:10:16
1. 并没有说一定不能定义set方法，文章中说不要滥定义用不上的set方法
2. 多个业务接口需要同一份数据？这个怎么理解呢？

- Monday 2019-11-18 21:48:56
我去去去，自从知道lombok后，@Data注解每个实体类必用。。。 [2赞]

- 嘉一 2019-11-18 20:28:30
个人觉得，MVC这种框架模式本质上与面向对象并不冲突。当我们在讨论面向对象的时候，我们究竟应该怎样去定义一个对象，究竟什么才能被我们看成是对象，是不是只有像某种物体，比如说一只鸟或者一只狗我们才能去把他定义为对象？我认为，MVC里面的三个部分Model 、Controller 、View 我们都能把他们单独的看成一个对象，比如说Model，本来它是数据单元，但是如果我们把他看做一个对象的话，里面存储的数据不就是我们对象里的属性么，而对于数据的二次加工处理等等操作不就是对象里的方法么？同理，对于View而言，里面小的view组件或者是其他的view不就是我们对象里面的属性，而对于不同的view组件或其他view的组合或者其他的处理操作不就是对象里面的方法么？所以说，不必死抠定义，数据就一定要和业务逻辑组成一个类云云。。。我们最后写出来的代码的目的就是，1.要解决问题；2.代码有可扩展性，可读性；3，代码解耦； [2赞]

作者回复2019-11-19 10:02:17
说的没错，MVC跟贫血模型没直接关系。我后面在实战篇会讲到的。你的观点我基本都赞同。

- 守拙 2019-11-18 16:10:42
今日的课堂讨论不会回答,尝试总结一下重点回顾的3个问题:

1.getter, setter问题的本质类的可变性问题.<Effective Java>中明确提到,除非有必要,否则类应该设计为不可变(Immutable)的.

2. Constants 和 Util类的问题本质是静态成员和静态方法问题.
静态成员和静态方法违背面向对象设计(OOP)原则,但从整体项目角度讲,静态成员和静态方法的好处大于其坏处,所以它们确实有存在的意义.

3.我对贫血模型的看法: 我是一名Android开发,日常使用的是MVC的变种MVP && MVVM模型.
MVP和MVVM相比MVC要更靠近OOP思想，但面向过程思想的设计仍包含于其中。
无论面向对象或面向过程，写出层次清晰，易扩展，易维护的代码才是目的。 [2赞]

- 中年男子 2019-11-18 15:25:18
接上条评论，代码实现如下，欢迎讨论
typedef struct _Base Base;
typedef void (*fptrDisplay)(Base*);
typedef void (*fptrDelete)(Base*);
void DisplayBase(Base*);
void DeleteBase(Base*);
typedef struct _Base
{
void* pDeriveObj;

```
int a;
int b;
fptrDisplay Display;
fptrDelete Delete;
}Base;
Base* new_base(int a, int b)
{
Base* pObj = NULL;
pObj = (Base*)malloc(sizeof(Base));
if (pObj == NULL)
{
return NULL;
}
// 当创建基类对象时指向自己
pObj->pDeriveObj = pObj;
pObj->a = a;
pObj->b = b;
pObj->Display = DisplayBase;
pObj->Delete = DeleteBase;
return pObj;
}
void DisplayBase(Base* base)
{
printf("member: a:%d\t b:%d\n", base->a, base->b);
}
void DeleteBase(Base* base)
{
printf("base destructor!\n");
free(base);
}
typedef struct _Derive
{
Base* pBaseObj;
int c;
int d;
}Derive;
void DeriveDisplay(Base* base);
void DeriveDelete(Base* base);
Base* new_Derive(int a, int b, int c, int d)
{
Derive* pObj = NULL;
Base* pBaseObj = new_base(a, b);
pObj = malloc(sizeof(Derive));
if (!pObj)
{
pBaseObj->Delete(pBaseObj);
return NULL;
}
pBaseObj->pDeriveObj = pObj;
pObj->pBaseObj = pBaseObj;
```

```
pObj->c = c;
pObj->d = d;
pBaseObj->Display = DeriveDisplay;
pBaseObj->Delete = DeriveDelete;
return pBaseObj;
}
void DeriveDisplay(Base* base)
{
Derive* pDeriveObj = (Derive*)(base->pDeriveObj);
printf("member:base:a:%d\t base:b:%d\nderive:c:%d\t derive:d:%d\n", base->a, base->b, pDeriveO
bj->c, pDeriveObj->d);
}
void DeriveDelete(Base* base)
{
printf("derive destructor!\n");
free(base->pDeriveObj);
free(base);
}
int main()
{
Base* pBase = new_base(1, 2);
Base* pDerive = new_Derive(3, 4, 5, 6);
pBase->Display(pBase);
pDerive->Display(pDerive);
pBase->Delete(pBase);
pDerive->Delete(pDerive);
} [2赞]
```

- Jxin 2019-11-18 13:16:30

  1.用shell实现自动化脚本做的服务编排，一般都是面向过程，一步一步的。而k8s的编排却是面向对象的，因为它为这个顺序流抽象出了很多角色，将原本一步一步的顺序操作转变成了多个角色间的轮转和交互。

  2.从接触ddd才走出javaer举面向对象旗，干面向过程勾当的局面。所谓为什么“充血模型”不流行，我认为不外呼两个。一，规范的领域模型对于底层基础架构来说并不友好（缺少setget），所以会导致规范的领域模型与现有基础架构不贴合，切很难开发出完全贴合的基础架构，进而引深出，合理的业务封装却阻碍关于复用通用抽象的矛盾。二，合理的业务封装，需要在战略上对业务先做合理的归类分割和抽象。而这个前置条件很少也不好达成。进而缺少前置设计封装出来的“充血模型”会有种四不像的味道，反而加剧了业务的复杂性，还不如“贫血模型”来得实用。事实上快节奏下，前置战略设计往往都是不足的，所以想构建优秀的“充血模型”架构，除了要对业务领域和领域设计有足够的认知，在重构手法和重构意愿上还要有一定讲究和追求，这样才能让项目以“充血模型”持续且良性的迭代。

  3.“充血模型”相对于“贫血模型”有什么好处？从我的经验来看，可读性其实可能“贫血模型”还好一点，这也可能有思维惯性的原因在里面。但从灵活和扩展性来说“充血模型”会优秀很多，因为好的“充血模型”往往意味着边界清晰（耦合低），功能内敛（高内聚）。这一块老师怎么看？ [2赞]

  作者回复2019-11-19 10:19:30

  说的非常好👍

- 饭太司替可 2019-11-18 11:48:41

C语言在A结构体和B结构体里面加一个指针指向结构体，就可以说A和B继承于C了。 [2赞]

- 青青子衿 2019-11-18 10:06:17
  有时候写get,set也是无奈之举，比如集成spring的时候，框架要求属性提供get,set [2赞]

- DebugDog 2019-11-18 10:06:12
  我全占了，今天才知道自己学的Java，天天在写面向过程😲 [2赞]

- 傲慢与偏执， 2019-11-18 08:46:09
  期待实战讲解 [2赞]

- 梦倚栏杆 2019-11-18 07:01:15
  1.现在因为使用封装好的框架，没有提供set方法，类的序列化会成为一个问题
  2.从理论上来说，数据和逻辑应该放在一起，但是数据的赋值往往可能依赖其他的service提供的数据，如果这样的话数据属性和纯粹依赖的service就会导致一个类的成员属性特别的多。
  3.对于一个具体现实对象而言，不同场景下可能关心的字段稍微有些不一样，对于此又该怎么处理呢？多个小对象，他们之间有无相关关系，有的话如何阐述，还是全部赋值完毕，都完整返回。当然可能具体场景具体分析，那是否有一个稍微通用的指导纲领
  期待老师关于面向对象的实战流程 [2赞]

  作者回复2019-11-18 08:31:05
  😳你说的后面都基本上有讲到

- Paul Shan 2019-11-18 04:53:14
  添加对象公有方法方法要注意两点：
  1.避免过多的访问内部数据的方法，只有在确定需要的时候再增加。
  2.避免返回内部引用（数组，对象，方法等），这些引用会无意间暴露修改内部状态的方法，导致数据不一致，也违反封装原则。这种情况如果数据量不大，返回一个拷贝对象即可。如果有效率考量，就要具体问题具体分析。 [2赞]

- 花儿少年 2019-11-18 00:40:48
  封装就是基本的函数
  继承是一种is-a的关系，属性的继承是父类的属性在子类之前按照顺序排列，接口的话是使用虚函数表，通过查表来确定父类有哪些接口
  多态的实现是通过函数指针，根据具体的子类找到子类继承的接口的入口地址，然后去执行就好了
  大致是c/c++的实现
  工作现在就是在用面对对象语言写着面向过程业务，😂😂 [2赞]