

## 05 | 理论二：封装、抽象、继承、多态分别可以解决哪些编程问题？

王争 · 设计模式之美



上一节课，我简单介绍了面向对象的一些基本概念和知识点，比如，什么是面向对象编程，什么是面向对象编程语言等等。其中，我们还提到，理解面向对象编程及面向对象编程语言的关键就是理解其四大特性：封装、抽象、继承、多态。不过，对于这四大特性，光知道它们的定义是不够的，我们还要知道每个特性存在的意义和目的，以及它们能解决哪些编程问题。所以，今天我就花一节课的时间，针对每种特性，结合实际的代码，带你将这些问题搞清楚。


这里我要强调一下，对于这四大特性，尽管大部分面向对象编程语言都提供了相应的语法机制来支持，但不同的编程语言实现这四大特性的语法机制可能会有所不同。所以，今天，我们在讲解四大特性的时候，并不与具体某种编程语言的特定语法相挂钩，同时，也希望你不要局限在你自己熟悉的编程语言的语法思维框架里。

### 封装（Encapsulation）

首先，我们来看封装特性。封装也叫作信息隐藏或者数据访问保护。类通过暴露有限的访问接口，授权外部仅能通过类提供的方式（或者叫函数）来访问内部信息或者数据。这句话怎么理

解呢？我们通过一个简单的例子来解释一下。

下面这段代码是金融系统中一个简化版的虚拟钱包的代码实现。在金融系统中，我们会给每个用户创建一个虚拟钱包，用来记录用户在我们的系统中的虚拟货币量。对于虚拟钱包的业务背景，这里你只需要简单了解一下即可。在面向对象的实战篇中，我们会有单独两节课，利用 OOP 的设计思想来详细介绍虚拟钱包的设计实现。

 复制代码

```
1 public class Wallet {
2     private String id;
3     private long createTime;
4     private BigDecimal balance;
5     private long balanceLastModifiedTime;
6     // ...省略其他属性...
7
8     public Wallet() {
9         this.id = IdGenerator.getInstance().generate();
10        this.createTime = System.currentTimeMillis();
11        this.balance = BigDecimal.ZERO;
12        this.balanceLastModifiedTime = System.currentTimeMillis();
13    }
14
15    // 注意：下面对get方法做了代码折叠，是为了减少代码所占文章的篇幅
16    public String getId() { return this.id; }
17    public long getCreateTime() { return this.createTime; }
18    public BigDecimal getBalance() { return this.balance; }
19    public long getBalanceLastModifiedTime() { return this.balanceLastModifiedTime; }
20
21    public void increaseBalance(BigDecimal increasedAmount) {
22        if (increasedAmount.compareTo(BigDecimal.ZERO) < 0) {
23            throw new InvalidAmountException("...");
24        }
25        this.balance.add(increasedAmount);
26        this.balanceLastModifiedTime = System.currentTimeMillis();
27    }
28
29    public void decreaseBalance(BigDecimal decreasedAmount) {
30        if (decreasedAmount.compareTo(BigDecimal.ZERO) < 0) {
31            throw new InvalidAmountException("...");
32        }
33        if (decreasedAmount.compareTo(this.balance) > 0) {
34            throw new InsufficientAmountException("...");
35        }
36        this.balance.subtract(decreasedAmount);
37        this.balanceLastModifiedTime = System.currentTimeMillis();
38    }
39 }
```

```
38     }  
39 }
```

从代码中，我们可以发现，Wallet 类主要有四个属性（也可以叫作成员变量），也就是我们前面定义中提到的信息或者数据。其中，id 表示钱包的唯一编号，createTime 表示钱包创建的时间，balance 表示钱包中的余额，balanceLastModifiedTime 表示上次钱包余额变更的时间。

我们参照封装特性，对钱包的这四个属性的访问方式进行了限制。调用者只允许通过下面这六个方法来访问或者修改钱包里的数据。

```
String getId()  
  
long getCreateTime()  
  
BigDecimal getBalance()  
  
long getBalanceLastModifiedTime()  
  
void increaseBalance(BigDecimal increasedAmount)  
  
void decreaseBalance(BigDecimal decreasedAmount)
```

之所以这样设计，是因为从业务的角度来说，id、createTime 在创建钱包的时候就确定好了，之后不应该再被改动，所以，我们并没有在 Wallet 类中，暴露 id、createTime 这两个属性的任何修改方法，比如 set 方法。而且，这两个属性的初始化设置，对于 Wallet 类的调用者来说，也应该是透明的，所以，我们在 Wallet 类的构造函数内部将其初始化设置好，而不是通过构造函数的参数来外部赋值。

对于钱包余额 balance 这个属性，从业务的角度来说，只能增或者减，不会被重新设置。所以，我们在 Wallet 类中，只暴露了 increaseBalance() 和 decreaseBalance() 方法，并没有暴露 set 方法。对于 balanceLastModifiedTime 这个属性，它完全是跟 balance 这个属性的修改操作绑定在一起的。只有在 balance 修改的时候，这个属性才会被修改。所以，我们把 balanceLastModifiedTime 这个属性的修改操作完全封装在了 increaseBalance() 和

decreaseBalance() 两个方法中，不对外暴露任何修改这个属性的方法和业务细节。这样也可以保证 balance 和 balanceLastModifiedTime 两个数据的一致性。

对于封装这个特性，我们需要编程语言本身提供一定的语法机制来支持。这个语法机制就是**访问权限控制**。例子中的 private、public 等关键字就是 Java 语言中的访问权限控制语法。private 关键字修饰的属性只能类本身访问，可以保护其不被类之外的代码直接访问。如果 Java 语言没有提供访问权限控制语法，所有的属性默认都是 public 的，那任意外部代码都可以通过类似 wallet.id=123; 这样的方式直接访问、修改属性，也就没办法达到隐藏信息和保护数据的目的了，也就无法支持封装特性了。

**封装特性的定义讲完了，我们再来看一下，封装的意义是什么？它能解决什么编程问题？**

如果我们对类中属性的访问不做限制，那任何代码都可以访问、修改类中的属性，虽然这样看起来更加灵活，但从另一方面来说，过度灵活也意味着不可控，属性可以随意被以各种奇葩的方式修改，而且修改逻辑可能散落在代码中的各个角落，势必影响代码的可读性、可维护性。比如某个同事在不了解业务逻辑的情况下，在某段代码中“偷偷地”重设了 wallet 中的 balanceLastModifiedTime 属性，这就会导致 balance 和 balanceLastModifiedTime 两个数据不一致。

除此之外，类仅仅通过有限的方法暴露必要的操作，也能提高类的易用性。如果我们把类属性都暴露给类的调用者，调用者想要正确地操作这些属性，就势必要对业务细节有足够的了解。而这对于调用者来说也是一种负担。相反，如果我们将属性封装起来，暴露少许的几个必要的方法给调用者使用，调用者就不需要了解太多背后的业务细节，用错的概率就减少很多。这就好比，如果一个冰箱有很多按钮，你就要研究很长时间，还不一定能操作正确。相反，如果只有几个必要的按钮，比如开、停、调节温度，你一眼就能知道该如何来操作，而且操作出错的概率也会降低很多。

## 抽象 (Abstraction)

讲完了封装特性，我们再来看抽象特性。封装主要讲的是如何隐藏信息、保护数据，而抽象讲的是如何隐藏方法的具体实现，让调用者只需要关心方法提供了哪些功能，并不需要知道这些功能是如何实现的。

在面向对象编程中，我们常借助编程语言提供的接口类（比如 Java 中的 interface 关键字语法）或者抽象类（比如 Java 中的 abstract 关键字语法）这两种语法机制，来实现抽象这一特性。

这里我稍微说明一下，在专栏中，我们把编程语言提供的接口语法叫作“接口类”而不是“接口”。之所以这么做，是因为“接口”这个词太泛化，可以指好多概念，比如 API 接口等，所以，我们用“接口类”特指编程语言提供的接口语法。

对于抽象这个特性，我举一个例子来进一步解释一下。

 复制代码

```
1 public interface IPictureStorage {
2     void savePicture(Picture picture);
3     Image getPicture(String pictureId);
4     void deletePicture(String pictureId);
5     void modifyMetaInfo(String pictureId, PictureMetaInfo metaInfo);
6 }
7
8 public class PictureStorage implements IPictureStorage {
9     // ...省略其他属性...
10    @Override
11    public void savePicture(Picture picture) { ... }
12    @Override
13    public Image getPicture(String pictureId) { ... }
14    @Override
15    public void deletePicture(String pictureId) { ... }
16    @Override
17    public void modifyMetaInfo(String pictureId, PictureMetaInfo metaInfo) { ... }
18 }
```

在上面的这段代码中，我们利用 Java 中的 interface 接口语法来实现抽象特性。调用者在使用图片存储功能的时候，只需要了解 IPictureStorage 这个接口类暴露了哪些方法就可以了，不需要去查看 PictureStorage 类里的具体实现逻辑。

实际上，抽象这个特性是很容易实现的，并不需要非得依靠接口类或者抽象类这些特殊语法机制来支持。换句话说，并不是说一定要为实现类（PictureStorage）抽象出接口类

(IPictureStorage)，才叫作抽象。即便不编写 IPictureStorage 接口类，单纯的 PictureStorage 类本身就满足抽象特性。

之所以这么说，那是因为，类的方法是通过编程语言中的“函数”这一语法机制来实现的。通过函数包裹具体的实现逻辑，这本身就是一种抽象。调用者在使用函数的时候，并不需要去研究函数内部的实现逻辑，只需要通过函数的命名、注释或者文档，了解其提供了什么功能，就可以直接使用了。比如，我们在使用 C 语言的 malloc() 函数的时候，并不需要了解它的底层代码是怎么实现的。

除此之外，在上一节课中，我们还提到，抽象有时候会被排除在面向对象的四大特性之外，当时我卖了一个关子，现在我就来解释一下为什么。

抽象这个概念是一个非常通用的设计思想，并不单单用在面向对象编程中，也可以用来指导架构设计等。而且这个特性也并不需要编程语言提供特殊的语法机制来支持，只需要提供“函数”这一非常基础的语法机制，就可以实现抽象特性、所以，它没有很强的“特异性”，有时候并不被看作面向对象编程的特性之一。

**抽象特性的定义讲完了，我们再来看一下，抽象的意义是什么？它能解决什么编程问题？**

实际上，如果上升一个思考层面的话，抽象及其前面讲到的封装都是人类处理复杂性的有效手段。在面对复杂系统的时候，人脑能承受的信息复杂程度是有限的，所以我们必须忽略掉一些非关键性的实现细节。而抽象作为一种只关注功能点不关注实现的设计思路，正好帮我们的大脑过滤掉许多非必要的信息。

除此之外，抽象作为一个非常宽泛的设计思想，在代码设计中，起到非常重要的指导作用。很多设计原则都体现了抽象这种设计思想，比如基于接口而非实现编程、开闭原则（对扩展开放、对修改关闭）、代码解耦（降低代码的耦合性）等。我们在讲到后面的内容的时候，会具体来解释。

换一个角度来考虑，我们在定义（或者叫命名）类的方法的时候，也要有抽象思维，不要在方法定义中，暴露太多的实现细节，以保证在某个时间点需要改变方法的实现逻辑的时候，不用去修改其定义。举个简单例子，比如 getAliyunPictureUrl() 就不是一个具有抽象思维的命



名，因为某一天如果我们不再把图片存储在阿里云上，而是存储在私有云上，那这个命名也要随之被修改。相反，如果我们定义一个比较抽象的函数，比如叫作 `getPictureUrl()`，那即便内部存储方式修改了，我们也不需要修改命名。

## 继承 (Inheritance)

学习完了封装和抽象两个特性，我们再来看继承特性。如果你熟悉的是类似 Java、C++ 这样的面向对象的编程语言，那你对继承这一特性，应该不陌生了。继承是用来表示类之间的 is-a 关系，比如猫是一种哺乳动物。从继承关系上来讲，继承可以分为两种模式，单继承和多继承。单继承表示一个子类只继承一个父类，多继承表示一个子类可以继承多个父类，比如猫既是哺乳动物，又是爬行动物。

为了实现继承这个特性，编程语言需要提供特殊的语法机制来支持，比如 Java 使用 `extends` 关键字来实现继承，C++ 使用冒号 (`class B : public A`)，Python 使用 parentheses (`()`)，Ruby 使用 `<`。不过，有些编程语言只支持单继承，不支持多重继承，比如 Java、PHP、C#、Ruby 等，而有些编程语言既支持单重继承，也支持多重继承，比如 C++、Python、Perl 等。

为什么有些语言支持多重继承，有些语言不支持呢？这个问题留给你自己去研究，你可以针对你熟悉的编程语言，在留言区写一写具体的原因。

**继承特性的定义讲完了，我们再来看，继承存在的意义是什么？它能解决什么编程问题？**

继承最大的一个好处就是代码复用。假如两个类有一些相同的属性和方法，我们就可以将这些相同的部分，抽取到父类中，让两个子类继承父类。这样，两个子类就可以重用父类中的代码，避免代码重复写多遍。不过，这一点也并不是继承所独有的，我们也可以通过其他方式来解决这个代码复用的问题，比如利用组合关系而不是继承关系。


如果我们再上升一个思维层面，去思考继承这一特性，可以这么理解：我们代码中有一个猫类，有一个哺乳动物类。猫属于哺乳动物，从人类认知的角度上来说，是一种 is-a 关系。我们通过继承来关联两个类，反应真实世界中的这种关系，非常符合人类的认知，而且，从设计的角度来说，也有一种结构美感。

继承的概念很好理解，也很容易使用。不过，过度使用继承，继承层次过深过复杂，就会导致代码可读性、可维护性变差。为了了解一个类的功能，我们不仅需要查看这个类的代码，还需要按照继承关系一层一层地往上查看“父类、父类的父类.....”的代码。还有，子类和父类高度耦合，修改父类的代码，会直接影响到子类。

所以，继承这个特性也是一个非常有争议的特性。很多人觉得继承是一种反模式。我们应该尽量少用，甚至不用。关于这个问题，在后面讲到“多用组合少用继承”这种设计思想的时候，我会非常详细地再讲解，这里暂时就不展开讲解了。

## 多态 (Polymorphism)

学习完了封装、抽象、继承之后，我们再来看面向对象编程的最后一个特性，多态。多态是指，子类可以替换父类，在实际的代码运行过程中，调用子类的方法实现。对于多态这种特性，纯文字解释不好理解，我们还是看一个具体的例子。

 复制代码

```
1 public class DynamicArray {
2     private static final int DEFAULT_CAPACITY = 10;
3     protected int size = 0;
4     protected int capacity = DEFAULT_CAPACITY;
5     protected Integer[] elements = new Integer[DEFAULT_CAPACITY];
6
7     public int size() { return this.size; }
8     public Integer get(int index) { return elements[index]; }
9     //...省略n多方法...
10
11     public void add(Integer e) {
12         ensureCapacity();
13         elements[size++] = e;
14     }
15
16     protected void ensureCapacity() {
17         //...如果数组满了就扩容...代码省略...
18     }
19 }
20
21 public class SortedDynamicArray extends DynamicArray {
22     @Override
23     public void add(Integer e) {
24         ensureCapacity();
25         int i;
```



```

26     for (i = size-1; i>=0; --i) { //保证数组中的数据有序
27         if (elements[i] > e) {
28             elements[i+1] = elements[i];
29         } else {
30             break;
31         }
32     }
33     elements[i+1] = e;
34     ++size;
35 }
36 }
37
38 public class Example {
39     public static void test(DynamicArray dynamicArray) {
40         dynamicArray.add(5);
41         dynamicArray.add(1);
42         dynamicArray.add(3);
43         for (int i = 0; i < dynamicArray.size(); ++i) {
44             System.out.println(dynamicArray.get(i));
45         }
46     }
47
48     public static void main(String args[]) {
49         DynamicArray dynamicArray = new SortedDynamicArray();
50         test(dynamicArray); // 打印结果: 1、3、5
51     }
52 }

```

多态这种特性也需要编程语言提供特殊的语法机制来实现。在上面的例子中，我们用到了三个语法机制来实现多态。

第一个语法机制是编程语言要支持父类对象可以引用子类对象，也就是可以将 SortedDynamicArray 传递给 DynamicArray。

第二个语法机制是编程语言要支持继承，也就是 SortedDynamicArray 继承了 DynamicArray，才能将 SortedDynamicArray 传递给 DynamicArray。


第三个语法机制是编程语言要支持子类可以重写（override）父类中的方法，也就是 SortedDynamicArray 重写了 DynamicArray 中的 add() 方法。

通过这三种语法机制配合在一起，我们就实现了在 test() 方法中，子类 SortedDynamicArray 替换父类 DynamicArray，执行子类 SortedDynamicArray 的 add() 方法，也就是实现了多态

特性。

对于多态特性的实现方式，除了利用“继承加方法重写”这种实现方式之外，我们还有其他两种比较常见的的实现方式，一个是利用接口类语法，另一个是利用 duck-typing 语法。不过，并不是每种编程语言都支持接口类或者 duck-typing 这两种语法机制，比如 C++ 就不支持接口类语法，而 duck-typing 只有一些动态语言才支持，比如 Python、JavaScript 等。

接下来，我们先来看如何利用接口类来实现多态特性。我们还是先来看一段代码。

 复制代码


```
1 public interface Iterator {
2     boolean hasNext();
3     String next();
4     String remove();
5 }
6
7 public class Array implements Iterator {
8     private String[] data;
9
10    public boolean hasNext() { ... }
11    public String next() { ... }
12    public String remove() { ... }
13    //...省略其他方法...
14 }
15
16 public class LinkedList implements Iterator {
17     private LinkedListNode head;
18
19    public boolean hasNext() { ... }
20    public String next() { ... }
21    public String remove() { ... }
22    //...省略其他方法...
23 }
24
25 public class Demo {
26     private static void print(Iterator iterator) {
27         while (iterator.hasNext()) {
28             System.out.println(iterator.next());
29         }
30     }
31
32     public static void main(String[] args) {
33         Iterator arrayIterator = new Array();
34         print(arrayIterator);
35     }
36 }
```

```
35
36     Iterator linkedListIterator = new LinkedList();
37     print(linkedListIterator);
38 }
39 }
```

在这段代码中，Iterator 是一个接口类，定义了一个可以遍历集合数据的迭代器。Array 和 LinkedList 都实现了接口类 Iterator。我们通过传递不同类型的实现类（Array、LinkedList）到 print(Iterator iterator) 函数中，支持动态的调用不同的 next()、hasNext() 实现。

具体点讲就是，当我们往 print(Iterator iterator) 函数传递 Array 类型的对象的时候，print(Iterator iterator) 函数就会调用 Array 的 next()、hasNext() 的实现逻辑；当我们往 print(Iterator iterator) 函数传递 LinkedList 类型的对象的时候，print(Iterator iterator) 函数就会调用 LinkedList 的 next()、hasNext() 的实现逻辑。

刚刚讲的是用接口类来实现多态特性。现在，我们再来看下，如何用 duck-typing 来实现多态特性。我们还是先来看一段代码。这是一段 Python 代码。

 复制代码

```
1 class Logger:
2     def record(self):
3         print("I write a log into file.")
4
5 class DB:
6     def record(self):
7         print("I insert data into db. ")
8
9 def test(recorder):
10     recorder.record()
11
12 def demo():
13     logger = Logger()
14     db = DB()
15     test(logger)
16     test(db)
```

从这段代码中，我们发现，duck-typing 实现多态的方式非常灵活。Logger 和 DB 两个类没有任何关系，既不是继承关系，也不是接口和实现的关系，但是只要它们都有定义了 record() 方法，就可以被传递到 test() 方法中，在实际运行的时候，执行对应的 record() 方法。

也就是说，只要两个类具有相同的方法，就可以实现多态，并不要求两个类之间有任何关系，这就是所谓的 duck-typing，是一些动态语言所特有的语法机制。而像 Java 这样的静态语言，通过继承实现多态特性，必须要求两个类之间有继承关系，通过接口实现多态特性，类必须实现对应的接口。

**多态特性讲完了，我们再来看，多态特性存在的意义是什么？它能解决什么编程问题？**

多态特性能提高代码的可扩展性和复用性。为什么这么说呢？我们回过头去看讲解多态特性的时候，举的第二个代码实例（Iterator 的例子）。

在那个例子中，我们利用多态的特性，仅用一个 print() 函数就可以实现遍历打印不同类型（Array、LinkedList）集合的数据。当再增加一种要遍历打印的类型的时候，比如 HashMap，我们只需让 HashMap 实现 Iterator 接口，重新实现自己的 hasNext()、next() 等方法就可以了，完全不需要改动 print() 函数的代码。所以说，多态提高了代码的可扩展性。

如果我们不使用多态特性，我们就无法将不同的集合类型（Array、LinkedList）传递给相同的函数（print(Iterator iterator) 函数）。我们需要针对每种要遍历打印的集合，分别实现不同的 print() 函数，比如针对 Array，我们要实现 print(Array array) 函数，针对 LinkedList，我们要实现 print(LinkedList linkedList) 函数。而利用多态特性，我们只需要实现一个 print() 函数的打印逻辑，就能应对各种集合数据的打印操作，这显然提高了代码的复用性。

除此之外，多态也是很多设计模式、设计原则、编程技巧的代码实现基础，比如策略模式、基于接口而非实现编程、依赖倒置原则、里式替换原则、利用多态去掉冗长的 if-else 语句等等。关于这点，在学习后面的章节中，你慢慢会有更深的体会。

## 重点回顾

今天的内容就讲完了，我们来一起总结回顾一下，你需要重点掌握的几个知识点。

## 1. 关于封装特性

封装也叫作信息隐藏或者数据访问保护。类通过暴露有限的访问接口，授权外部仅能通过类提供的方式来访问内部信息或者数据。它需要编程语言提供权限访问控制语法来支持，例如 Java 中的 `private`、`protected`、`public` 关键字。封装特性存在的意义，一方面是保护数据不被随意修改，提高代码的可维护性；另一方面是仅暴露有限的必要接口，提高类的易用性。

## 2. 关于抽象特性

封装主要讲如何隐藏信息、保护数据，那抽象就是讲如何隐藏方法的具体实现，让使用者只需要关心方法提供了哪些功能，不需要知道这些功能是如何实现的。抽象可以通过接口类或者抽象类来实现，但也并不需要特殊的语法机制来支持。抽象存在的意义，一方面是提高代码的可扩展性、维护性，修改实现不需要改变定义，减少代码的改动范围；另一方面，它也是处理复杂系统的有效手段，能有效地过滤掉不必要关注的信息。

## 3. 关于继承特性

继承是用来表示类之间的 `is-a` 关系，分为两种模式：单继承和多继承。单继承表示一个子类只继承一个父类，多继承表示一个子类可以继承多个父类。为了实现继承这个特性，编程语言需要提供特殊的语法机制来支持。继承主要是用来解决代码复用的问题。

## 4. 关于多态特性

多态是指子类可以替换父类，在实际的代码运行过程中，调用子类的方法实现。多态这种特性也需要编程语言提供特殊的语法机制来实现，比如继承、接口类、`duck-typing`。多态可以提高代码的扩展性和复用性，是很多设计模式、设计原则、编程技巧的代码实现基础。

## 课堂讨论

今天我们要讨论的话题有如下两个。

1. 你熟悉的编程语言是否支持多重继承？如果不支持，请说一下为什么不支持。如果支持，请说一下它是如何避免多重继承的副作用的。

2. 你熟悉的编程语言对于四大特性是否都有现成的语法支持？对于支持的特性，是通过什么语法机制实现的？对于不支持的特性，又是基于什么原因做的取舍？

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

面向对象编程的四大特性：封装、抽象、继承、多态是面向对象编程的核心概念。本文重点介绍了封装特性，通过限制对类内部信息或数据的访问方式，提供了访问权限控制，从而保护数据不被随意修改，提高类的易用性和可维护性。文章通过一个虚拟钱包的代码实现示例，说明了封装特性的意义和目的。抽象特性通过隐藏方法的具体实现，让调用者只需要关心方法提供了哪些功能，并不需要知道这些功能是如何实现的。继承特性用来表示类之间的is-a关系，实现代码复用，但过度使用继承可能导致代码可读性、可维护性变差。文章还提到了抽象和继承的意义及存在的问题。多态特性允许子类替换父类，在实际的代码运行过程中，调用子类的方法实现。多态提高了代码的可扩展性和复用性，是很多设计模式、设计原则、编程技巧的代码实现基础。文章还介绍了多态的实现方式，包括继承加方法重写、接口类和duck-typing。多态特性的存在意义在于提高代码的可扩展性和复用性。整体而言，本文深入浅出地介绍了面向对象编程的四大特性，以及它们的意义和实现方式，对于想要深入理解面向对象编程的读者来说，是一篇很有价值的文章。

© 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 (358)

最新 精选



拉格朗日的忧桑

2019-11-13

这是迄今讲面向对象特性最深刻的，没有之一

作者回复: 😊 多谢认可

共 2 条评论 >

👍 148



码农Kevin亮

2019-11-17

请问老师，有些教科书上会把重载也理解为多态，老师怎么理解呢



作者回复: 是的 有人把它叫做静态多态



👍 9



6点无痛早起学习的和...

2019-12-22

多态是否又分为编译时多态和运行时多态?

作者回复: 是的



👍 7



昨日火冷啊

2019-11-17

老师好,  
请问c++的template算不算 duck-typing? 比如定义一个模版,  $A<T>$ , 用到函数 $T.Load()$ 。然后有类B 和C都有成员函数 $Load()$ 。那么都可以通过使用 $A<B>$  和 $A<C>$  来使用模版, 而B, C之间不需要继承关系。

作者回复: 从你的描述来看, 有点类似的。

共 3 条评论 >

👍 6



安静

2019-11-13

duck-typing 实现多态的方式确实灵活, 这个维护成功也会比较高吗?

作者回复: 我个人觉得duck typing过于灵活了



👍 6



辣么大

2019-11-13

老师, 我还有一个问题:  
Java中支持反射机制。利用反射机制, 我们可以访问类的私有变量和私有方法。请问这是否有违“封装”的初衷呢? 哪些场合我们要必须使用“反射”访问私有变量或者私有方法呢?

作者回复: 理论上讲这是一种hack的方法 不推荐使用

共 3 条评论 >

👍 5



学好习，做好人

2019-11-14

钱包那个例子，个人感觉应该把钱包余额的加减操作独立出来，比如WalletAction，并且WalletAction作为wallet的成员变量会不会更好一些，因为id,createTime,modifyTime可以算作钱包的一些固有属性，但是余额增加更偏重一些动态的计算行为，用组合的方式会不会更好些呢？

作者回复: 也可以，不过看起来就有点过度设计的意思了，多了一个类，维护成本也提高了。



👍 4



ThinkingQuest

2020-08-27

口语中的“爬行动物”指的是：动物界 脊索动物门 脊椎动物亚门 爬行纲。  
猫属于： 动物界 脊索动物门 脊椎动物亚门 哺乳纲 食肉目 猫科 猫属

所以，猫不是“爬行动物”。

你要说猫属于“会爬行的动物”，那也对。

作者回复: 🤔🤔🤔🤔🤔，谢谢姐

共 2 条评论 >

👍 2



小妖

2019-11-21

我觉得文中对多肽的定义有问题，多肽不仅是只子类替换父类（父类对象引用子类对象），也包括父类（代理类）可以在某些时候代替子类作为参数传递（继承的方式实现参数代理），更直观的表现是实现类替换接口（接口引用接口的实现比如：`List<String> list=new ArrayList<>();`），这听起来有点像是抽象，实际上抽象的很多情况下是依赖多肽的，比如，方法接口接口作为参数，而不必接收具体的类这体现了抽象，但更体现了多肽.....求交流

作者回复: 你举的例子不还是子类传递给父类吗



wanghai

2019-11-19

我最近在做一个功能，而且我也是有一些迷茫

我用到了继承，在父类定义了很多方法，然后我有很多子类，每个子类自己来实现一些需要重写的方法。

我也想到了组合，但是组合好像无法重写方法，导致变的更麻烦，希望大神帮我解惑～～

作者回复: 后面章节会讲到 别急

