

05-理论二：封装、抽象、继承、多态分别可以解决哪些编程问题？

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。

```
public class Wallet {
    private String id;
    private long createTime;
    private BigDecimal balance;
    private long balanceLastModifiedTime;
    // ...省略其他属性...

    public Wallet() {
        this.id = IdGenerator.getInstance().generate();
        this.createTime = System.currentTimeMillis();
        this.balance = BigDecimal.ZERO;
        this.balanceLastModifiedTime = System.currentTimeMillis();
    }

    // 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
    public String getId() { return this.id; }
    public long getCreateTime() { return this.createTime; }
    public BigDecimal getBalance() { return this.balance; }
    public long getBalanceLastModifiedTime() { return this.balanceLastModifiedTime; }

    public void increaseBalance(BigDecimal increasedAmount) {
        if (increasedAmount.compareTo(BigDecimal.ZERO) < 0) {
            throw new InvalidAmountException("...");
        }
        this.balance.add(increasedAmount);
        this.balanceLastModifiedTime = System.currentTimeMillis();
    }

    public void decreaseBalance(BigDecimal decreasedAmount) {
        if (decreasedAmount.compareTo(BigDecimal.ZERO) < 0) {
            throw new InvalidAmountException("...");
        }
    }
}
```

```
    }  
    if (decreasedAmount.compareTo(this.balance) > 0) {  
        throw new InsufficientAmountException("...");  
    }  
    this.balance.subtract(decreasedAmount);  
    this.balanceLastModifiedTime = System.currentTimeMillis();  
}  
}
```

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，如果我们把属性封装起来，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。

```
public interface IPictureStorage {  
    void savePicture(Picture picture);  
    Image getPicture(String pictureId);  
    void deletePicture(String pictureId);  
    void modifyMetaInfo(String pictureId, PictureMetaInfo metaInfo);  
}  
  
public class PictureStorage implements IPictureStorage {  
    // ...省略其他属性...  
    @Override  
    public void savePicture(Picture picture) { ... }  
    @Override  
    public Image getPicture(String pictureId) { ... }  
    @Override  
    public void deletePicture(String pictureId) { ... }  
    @Override  
    public void modifyMetaInfo(String pictureId, PictureMetaInfo metaInfo) { ... }  
}
```

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。

归纳总结能力

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paraentheses()，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。

代码复用，结构紧凑

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。

```
public class DynamicArray {
    private static final int DEFAULT_CAPACITY = 10;
    protected int size = 0;
    protected int capacity = DEFAULT_CAPACITY;
    protected Integer[] elements = new Integer[DEFAULT_CAPACITY];

    public int size() { return this.size; }
    public Integer get(int index) { return elements[index]; }
    //...省略n多方法...

    public void add(Integer e) {
        ensureCapacity();
        elements[size++] = e;
    }

    protected void ensureCapacity() {
        //...如果数组满了就扩容...代码省略...
    }
}

public class SortedDynamicArray extends DynamicArray {
    @Override
    public void add(Integer e) {
        ensureCapacity();
        for (int i = size-1; i>=0; --i) { //保证数组中的数据有序
            if (elements[i] > e) {
                elements[i+1] = elements[i];
            } else {
                break;
            }
        }
    }
}
```

```

    }
    elements[i+1] = e;
    ++size;
}
}

public class Example {
    public static void test(DynamicArray dynamicArray) {
        dynamicArray.add(5);
        dynamicArray.add(1);
        dynamicArray.add(3);
        for (int i = 0; i < dynamicArray.size(); ++i) {
            System.out.println(dynamicArray[i]);
        }
    }

    public static void main(String args[]) {
        DynamicArray dynamicArray = new SortedDynamicArray();
        test(dynamicArray); // 打印结果: 1、3、5
    }
}

```

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。

```

public interface Iterator {
    String hasNext();
    String next();
    String remove();
}

public class Array implements Iterator {
    private String[] data;

    public String hasNext() { ... }
}

```



```

    public String next() { ... }
    public String remove() { ... }
    //...省略其他方法...
}

public class LinkedList implements Iterator {
    private LinkedListNode head;

    public String hasNext() { ... }
    public String next() { ... }
    public String remove() { ... }
    //...省略其他方法...
}

public class Demo {
    private static void print(Iterator iterator) {
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }

    public static void main(String[] args) {
        Iterator arrayIterator = new Array();
        print(arrayIterator);

        Iterator linkedListIterator = new LinkedList();
        print(linkedListIterator);
    }
}

```

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。

```

class Logger:
    def record(self):
        print("I write a log into file.")

class DB:
    def record(self):
        print("I insert data into db. ")

def test(recorder):
    recorder.record()

def demo():
    logger = Logger()
    db = DB()
    test(logger)

```

```
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，和同学们一起交流和分享。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：

- №修罗★幻影 2019-11-13 08:20:30

Java 不支持多重继承的原因

多重继承有副作用：钻石问题(菱形继承)。

假设类 B 和类 C 继承自类 A，且都重写了类 A 中的同一个方法，而类 D 同时继承了类 B 和类 C，那么此时类 D 会继承 B、C 的方法，那对于 B、C 重写的 A 中的方法，类 D 会继承哪一个呢？这里就会产生歧义。

考虑到这种二义性问题，Java 不支持多重继承。但是 Java 支持多接口实现，因为接口中的方法，是抽象的（从JDK1.8之后，接口中允许给出一些默认方法的实现，这里不考虑这个），就算一个类实现了多个接口，且这些接口中存在某个同名方法，但是我们在实现接口的时候，这个同名方法需要由我们这个实现类自己来实现，所以并不会出现二义性的问题。[71赞]

- Smallfly 2019-11-14 10:14:12

争哥对面向对象的总结完美符合 What/How/Why 模型，我按照模型作下梳理。

封装

What：隐藏信息，保护数据访问。

How：暴露有限接口和属性，需要编程语言提供访问控制的语法。

Why：提高代码可维护性；降低接口复杂度，提高类的易用性。

##抽象

What: 隐藏具体实现，使用者只需关心功能，无需关心实现。

How: 通过接口类或者抽象类实现，特殊语法机制非必须。

Why: 提高代码的扩展性、维护性；降低复杂度，减少细节负担。

##继承

What: 表示 is-a 关系，分为单继承和多继承。

How: 需要编程语言提供特殊语法机制。例如 Java 的 “extends”，C++ 的 “:”。

Why: 解决代码复用问题。

##多态

What: 子类替换父类，在运行时调用子类的实现。

How: 需要编程语言提供特殊的语法机制。比如继承、接口类、duck-typing。

Why: 提高代码扩展性和复用性。

3W 模型的关键在于 Why，没有 Why，其它两个就没有存在的意义。从四大特性可以看出，面向对象的终极目的只有一个：可维护性。易扩展、易复用，降低复杂度等等都属于可维护性的实现方式。

[19赞]

- 小白 2019-11-13 10:06:50
go语言的“隐藏式接口”算是多态中duck-typing的实现方式吧 [13赞]
 - 哥本 2019-11-13 16:50:38
我理解的四大特性
封装：加装备（添加盔甲）
继承：师傅掌对掌传输武功（毫无保留）
抽象：从道到术，柳叶能伤人
多态：奥特曼变身。 [8赞]
 - weiguozhihui 2019-11-13 12:14:30
c 语言通过结构体来实现封装，只是c 的结构体没有关键字来控制结构体内部成员的访问权限问题，属于一种比较粗的封装。另外C中通过void*+结构体+函数指针也是可以实现多态的。Linux内核代码好多都是用了面向对象编程思想。C++中引入public protected private 关键字来进行访问控制权管理。C++中没有Java中的interface 关键字来描述接口类，但也是可以通过虚函数基类来进行的Java中的接口类的。C++是直接支持多继承的，但这个特性也被好多人诟病。 [6赞]
 - 业余草 2019-11-13 08:27:04
是时候抛出这道难住99%的程序员的设计模式面试题了！
<https://mp.weixin.qq.com/s/9SBV9ZycAQY82BacYICY2w> [6赞]
 - 拉格朗日的忧桑 2019-11-13 23:35:34
这是迄今讲面向对象特性最深刻的，没有之一 [5赞]
- 作者回复2019-11-14 08:16:06
☺ 多谢认可
- 晨风破晓 2019-11-13 20:26:07
PHP不支持多继承，具体为什么还没了解过，四大特性都是有现有语法支持的；看完这堂课，貌似对多态还不是很理解 [5赞]
 - 秉心说 2019-11-13 07:58:37

多继承会带来菱形继承的问题。例如一个类的两个父类，都继承了同一个祖父类，两个父类都 override 了祖父类的方法，这时候孙子类就不知道如何调用了。

Java 8 的 interface 可以有方法默认实现，这应该可以算是曲线救国的多继承吧。 [5赞]

- DesertSnow 2019-11-13 10:47:57

我们使用Java已经很长时间了，我们有多少次因为缺少多重继承而面临困难呢？

我个人的经验是一次都没有。因为多重继承很少有机会被用到，所以更安全的做法是去掉它而保持简单性。

就算是碰到需要多重继承的情景，我们也可以找到替代方法。

我的观点是，去掉对多重继承的支持不是Java的缺陷，对开发者来说是件好事。 [4赞]

- 划时代 2019-11-13 11:17:16

话题一：

C++语言的多重继承，存在三层继承关系时，采用virtual继承方式，形成菱形继承。标准库中的iostream类就存在多重继承关系，见图<http://www.cplusplus.com/img/iostream.gif>。

话题二：

C++语言的多态实现方式比较丰富，除了类中的virtual函数实现运行期多态以外。还支持编译期多态（模板元编程），不仅能够动态改变对象执行的函数，还能动态改变对象的定义类型。

[3赞]

- 李湘河 2019-11-13 10:45:59

示例代码自己运行了麽，虽说是将设计思想、设计模式，但是代码很多错误呀，像多态示例代码，父类私有属性在子类继承中能直接用吗？ [3赞]

作者回复2019-11-15 12:03:42

已经修改成protected的了。代码自己改改，可以运行的。但代码的作用主要还是辅助解释理论，所以有所删减，也是考虑到文章篇幅的问题。

- 每天晒白牙 2019-11-13 10:35:21

专栏中有个思考题是 java 为何不支持类多继承？却支持接口的多继承？

而有些语言如python是支持多继承的？

首先java类支持多继承的话，两个父类有同名方法，在子类中调用的时候就不知道调用哪个了，出现决议(钻石问题或菱形问题)问题

而接口支持多继承因为接口定义的方法不能有方法体，所以不会出现决议问题。

而从jdk1.8开始，接口可以有默认方法(方法要用default标识)，必须要有方法体，这样在接口多继承上也不会有决议问题吗？其实你去试下就知道了，java发现这种情况，会通过让你强制在子接口中重写这个默认方法，这样就不会有决议问题了

python支持多继承因为它通过MRO解决的，pythoner应该懂，我是不太懂，感兴趣的可以去研究 [3赞]

- Paul Shan 2019-11-13 17:57:20

Java对于封装，抽象，继承，多态支持够用了。封装和抽象解决的是信息隐藏的问题，也就是说不同位置需要看到的信息不同。封装是以类为边界，两边需要的信息量不对等。抽象是以调用者和实现者的角度来区分，两者需要的信息量差异也很大。继承和多态是抽象思想的延续，当类被分为接口和实现的时候，所有针对接口实现的操作，也同样使用于实现类。当一个接口有多个实现类，而针对接口实现的操作适用于所有满足这个接口的实现类，这样就让针对接口的代码得到复用。 [2赞]

- 编程界的小学生 2019-11-13 09:51:51

JAVA前来报到，不支持多继承是因为考虑到钻石问题，比如A继承B和C，B和C继承了D，那么A类中覆盖

的D方法是B重写的还是C重写的呢？导致混乱。

有个问题就是: 某种意义上讲，JAVA是不是隐式支持多继承的？因为每个类都默认继承了Object类，都用有Object的toString等方法。 [2赞]

- allean 2019-11-13 07:48:23
打卡 [2赞]

- 秋惊蛰 2019-11-13 01:44:27
试着说一下Python吧

- 抽象：抽象是编程语言的共有特点，甚至是计算机科学的特点，从变量，函数，类，模块，包等概念都是不同层次的抽象。抽象和把大象装进冰箱分三步是一个道理，它给出了思路，指明了方向，省略了细节。我们用层层抽象来应对计算机系统的复杂性。Python主要的抽象工具是函数和类，模块和包也算吧，毕竟也是隐藏了细节。

- 封装：Python不支持严格意义上的封装，没有private, protected等访问修饰符，这样做是为了保证动态语言最大的灵活性，同时Python里很多理念都是约定大于定义的，私有的属性需要大家守约，不要去随意访问，这也是Python被吐槽的地方吧，大型项目约束力不够。

- 继承：Python支持多重继承，主要是因为它没有类似于Java的“接口类”的语法吧，用多重继承可以定义一些纯功能性的类，减少类的层级。

- 多态：Python的多态就是鸭子类型了，鸭子类型的背后是所谓“协议”，协议是非正式的接口，是一种特性，表现为一个或多个相关的方法，比如迭代器协议，序列协议。实现了迭代器协议就和Java中实现了Iterator接口一样。 [2赞]

- DesertSnow 2019-11-13 00:02:04
沙发 [2赞]

- Monday 2019-11-13 22:38:50

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？

java不支持多重继承。

原因：多重继承中若不同父类存在相同方法签名时，子类调用此方法时会不知道调用哪个父类的方法的问题。

如何避免：个人觉得可以根据所继承父类的先后顺序来确定优先级 [1赞]

- helloworld 2019-11-13 17:50:04

这节课虽然对于有编程经验的人来说很容易理解，但是读完后能感受到老师清晰的解读思路，赞！ [1赞]