

面向对象设计与设计模式

面向对象

什么是面向对象编程和面向对象编程语言

- 面向对象编程：OOP，全称是 Object Oriented Programming。
- 面向对象编程语言：OOPL，全称是 Object Oriented Programming Language。
- 面向对象编程是一种编程范式或编程风格。它以类或对象作为组织代码的基本单元，并将封装、抽象、继承、多态四个特性，作为代码设计和实现的基石。
- 面向对象编程语言是支持类或对象的语法机制，并有现成的语法机制，能方便地实现面向对象编程四大特性（封装、抽象、继承、多态）的编程语言。

-
- 面向对象编程中有两个非常重要、非常基础的概念，那就是类（class）和对象（object）。这两个概念最早出现在 1960 年，在 Simula 这种编程语言中第一次使用。而面向对象编程这个概念第一次被使用是在 Smalltalk 这种编程语言中。Smalltalk 被认为是第一个真正意义上的面向对象编程语言。1980 年左右，C++ 的出现，带动了面向对象编程的流行。

-
- 直到今天，如果不按照严格的定义来说，大部分编程语言都是面向对象编程语言，比如 Java、C++、Go、Python、C#、Ruby、JavaScript、Objective-C、Scala、PHP、Perl 等等。除此之外，大部分程序员在开发项目的时候，都是基于面向对象编程语言进行的面向对象编程。

-
- 面向对象分析英文缩写是 OOA，全称是 Object Oriented Analysis；
 - 面向对象设计的英文缩写是 OOD，全称是 Object Oriented Design。
 - OOA、OOD、OOP 三个连在一起就是面向对象分析、设计、编程（实现），正好是面向对象软件开发要经历的三个阶段。

封装、抽象、继承、多态 分别可以解决哪些编程问题

- 封装 (Encapsulation)
- 抽象 (Abstraction)
- 继承 (Inheritance)
- 多态 (Polymorphism)

封装 (Encapsulation)

- 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。


```
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; }
```



```
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 }
```


抽象 (Abstraction)

- 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。

继承 (Inheritance)

- 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。
 - 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。
- 继承最大的一个好处就是代码复用。但是过度使用继承，继承层次过深过复杂，就会导致代码可读性、可维护性变差。

多态 (Polymorphism)

- 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。
- 多态特性需要编程语言提供特殊的语法机制来实现，主要的实现方式：
 - 继承加方法重写
 - 利用接口类语法（一些语言，如 C++ 就不支持接口类语法）
 - duck-typing（只有一些动态语言才支持，比如 Python、JavaScript 等）
- 多态特性能提高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。

- 继承

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



```
22 public class SortedDynamicArray extends DynamicArray {
23     @Override
24     public void add(Integer e) {
25         ensureCapacity();
26         int i;
27         for (i = size-1; i>=0; --i) { //保证数组中的数据有序
28             if (elements[i] > e) {
29                 elements[i+1] = elements[i];
30             } else {
31                 break;
32             }
33         }
34         elements[i+1] = e;
35         ++size;
36     }
37 }
```



```
39 public class Example {
40     public static void test(DynamicArray dynamicArray) {
41         dynamicArray.add(5);
42         dynamicArray.add(1);
43         dynamicArray.add(3);
44         for (int i = 0; i < dynamicArray.size(); ++i) {
45             System.out.println(dynamicArray.get(i));
46         }
47     }
48
49     public static void main(String args[]) {
50         DynamicArray dynamicArray = new SortedDynamicArray();
51         test(dynamicArray); // 打印结果: 1、3、5
52     }
53 }
```


- 接口

```
1  public interface Iterator {
2      String hasNext();
3      String next();
4      String remove();
5  }
6  public class Array implements Iterator {
7      private String[] data;
8      public String hasNext() { ... }
9      public String next() { ... }
10     public String remove() { ... }
11     //...省略其他方法...
12 }
13 public class LinkedList implements Iterator {
14     private LinkedListNode head;
15     public String hasNext() { ... }
16     public String next() { ... }
17     public String remove() { ... }
18     //...省略其他方法...
19 }
```



```
23 public class Demo {
24     private static void print(Iterator iterator) {
25         while (iterator.hasNext()) {
26             System.out.println(iterator.next());
27         }
28     }
29
30     public static void main(String[] args) {
31         Iterator arrayIterator = new Array();
32         print(arrayIterator);
33
34         Iterator linkedListIterator = new LinkedList();
35         print(linkedListIterator);
36     }
37 }
```


- Python
- Duck-typing

```
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)
```


编程范式（编程风格）

- 面向过程编程
- 面向对象
- 函数式编程

面向对象编程相比面向过程编程的优势

- 对于大规模复杂程序的开发，程序的处理流程并非单一的一条主线，而是错综复杂的网状结构。面向对象编程比起面向过程编程，更能应对这种复杂类型的程序开发。
- 面向对象编程相比面向过程编程，具有更加丰富的特性（封装、抽象、继承、多态）。利用这些特性编写出来的代码，更加易扩展、易复用、易维护。
- 从编程语言跟机器打交道的方式的演进规律中，我们可以总结出：面向对象编程语言比起面向过程编程语言，更加人性化、更加高级、更加智能。

违反面向对象编程风格的典型代码设计

- 用OOPL就一定OOP了吗？以下列举三种违反面向对象编程风格的典型代码设计。
- 滥用 getter、setter 方法
 - setter 暴露了直接修改数据的方法，可能破坏数据的一致性
 - getter 就安全吗？返回一个引用？
 - 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。

违反面向对象编程风格的典型代码设计

- 滥用全局变量和全局方法、静态变量和静态方法
 - 不允许修改，只可以只读。常见 Constants 类和 Utils 类
 - 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。
 - 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。如果能够将这类中的属性和方法，划分归并到其他业务类中，能极大地提高类的内聚性和代码的可复用性。
 - 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。

违反面向对象编程风格的典型代码设计

- 定义数据和方法分离的类
 - 传统的 MVC 结构分为 Model 层、Controller 层、View 层这三层。
 - 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。
 - 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：贫血模型