

设计模式

设计模式的 6 大设计原则

设计模式的三大分类

常见的设计模式有哪几种

1. 单例模式：保证一个类仅有一个实例，并提供一个访问它的全局访问点。（连接池）

1. 饿汉式

2. 懒汉式

3. 双重检测

2. 工厂模式

3. 观察者模式

● 推模型

● 拉模型

4. 装饰模式

5. 建造者模式

6. 代理模式

7. 策略模式

设计模式的 6 大设计原则

1. 单一职责原则：就一个类而言，应该仅有一个引起它变化的原因。

2. 开放封闭原则：软件实体可以扩展，但是不可修改。即面对需求，对程序的改动可以通过增加代码来完成，但是不能改动现有的代码。

3. 里氏代换原则：一个软件实体如果使用的是一个基类，那么一定适用于其派生类。即在软件中，把基类替换成派生类，程序的行为没有变化。

4. 依赖倒转原则：抽象不应该依赖细节，细节应该依赖抽象。即针对接口编程，不要针对实现编程。

5. 迪米特原则：如果两个类不直接通信，那么这两个类就不应当发生直接的相互作用。如果一个类需要调用另一个类的某个方法的话，可以通过第三个类转发这个调用。

6. 接口隔离原则：每个接口中不存在派生类用不到却必须实现的方法，如果不然，就要将接口拆分，使用多个隔离的接口。

设计模式的三大分类

1. **创造型模式**：单例模式、工厂模式、建造者模式、原型模式 （4）
2. 结构型模式：适配器模式、桥接模式、外观模式、组合模式、装饰模式、享元模式、代理模式 （7）
3. **行为型模式**：责任链模式、命令模式、解释器模式、迭代器模式、中介者模式、备忘录模式、观察者模式、状态模式、策略模式、模板方法模式、访问者模式 （12）

注意：简单工厂模式 违背了六大原则中的开发-封闭原则，故而不属于23种GOF设计模式之一
也叫静态工厂方法模式

常见的设计模式有哪几种

1. 单例模式：保证一个类仅有一个实例，并提供一个访问它的全局访问点。（连接池）

单例模式的实现需要三个必要的条件：

1. 单例类的构造函数必须是私有的，这样才能将类的创建权控制在类的内部，从而使得类的外部不能创建类的实例。
2. 单例类通过一个私有的静态变量来存储其唯一实例。
3. 单例类通过提供一个公开的静态方法，使得外部使用者可以访问类的唯一实例。

另外，实现单例类时，还需要考虑三个问题：

1. 创建单例对象时，是否线程安全。
2. 单例对象的创建，是否延时加载。
3. 获取单例对象时，是否需要加锁（锁会导致低性能）。

下面介绍五种实现单例模式的方式。

1. 饿汉式

饿汉式的单例实现比较简单，其在类加载的时候，静态实例 `instance` 就已创建并初始化好了。

```
1 public class Singleton {
2     private static final Singleton instance = new Singleton();
3
4     private Singleton () {}
5
6     public static Singleton getInstance() {
7         return instance;
8     }
9 }
10
```

饿汉式单例优缺点：

- 优点：
 - 单例对象的创建是线程安全的；
 - 获取单例对象时不需要加锁。
- 缺点：单例对象的创建，不是延时加载。

一般认为延时加载可以节省内存资源。但是延时加载是不是真正的好，要看实际的应用场景，而不一定所有的应用场景都需要延时加载。

2. 懒汉式

与饿汉式对应的是懒汉式，懒汉式为了支持延时加载，将对象的创建延迟到了获取对象的时候，但为了线程安全，不得不为获取对象的操作加锁，这就导致了低性能。

```
1 public class Singleton {
2     private static final Singleton instance;
3
4     private Singleton () {}
5
6     public static synchronized Singleton getInstance() {
7         if (instance == null) {
8             instance = new Singleton();
9         }
10
11         return instance;
12     }
13 }
14
```

懒汉式单例优缺点：

- 优点：
 - 对象的创建是线程安全的。
 - 支持延时加载。
- 缺点：获取对象的操作被加上了锁，影响了并发度。
 - 如果单例对象需要频繁使用，那这个缺点就是无法接受的。
 - 如果单例对象不需要频繁使用，那这个缺点也无伤大雅。

3. 双重检测

饿汉式和懒汉式的单例都有缺点，双重检测的实现方式解决了这两者的缺点。

双重检测将懒汉式中的 `synchronized` 方法改成了 `synchronized` 代码块。

Java

复制代码

```
1 public class Singleton {
2     private static Singleton instance;
3
4     private Singleton () {}
5
6     public static Singleton getInstance() {
7         if (instance == null) {
8             synchronized(Singleton.class) { // 注意这里是类级别的锁
9                 if (instance == null) { // 这里的检测避免多线程并发时多次创建对象
10                     instance = new Singleton();
11                 }
12             }
13         }
14         return instance;
15     }
16 }
17
```

双重检测单例优点：


- 对象的创建是线程安全的。
- 支持延时加载。
- 获取对象时不需要加锁。

2. 工厂模式

包括简单工厂模式、抽象工厂模式、工厂方法模式

- a. 简单工厂模式：主要用于创建对象。用一个工厂来根据输入的条件产生不同的类，然后根据不同类的虚函数得到不同的结果。

Java

 复制代码

```
1 public class SimpleFactory {
2     public static Product createProduct(String type) {
3         if (type.equals("A")) {
4             return new ProductA();
5         } else if (type.equals("B")) {
6             return new ProductB();
7         } else {
8             return null;
9         }
10    }
11 }
12
13 public interface Product {
14     void use();
15 }
16
17 public class ProductA implements Product {
18     @Override
19     public void use() {
20         System.out.println("Using Product A");
21     }
22 }
23
24 public class ProductB implements Product {
25     @Override
26     public void use() {
27         System.out.println("Using Product B");
28     }
29 }
30
```

- b. 工厂方法模式：修正了简单工厂模式中不遵守开放封闭原则。把选择判断移到了客户端去实现，如果想添加新功能就不用修改原来的类，直接修改客户端即可。

```
1 ▼ public interface Factory {  
2     Product createProduct();  
3 }  
4  
5 ▼ public class ProductAFactory implements Factory {  
6     @Override  
7     public Product createProduct() {  
8         return new ProductA();  
9     }  
10 }  
11  
12 ▼ public class ProductBFactory implements Factory {  
13     @Override  
14     public Product createProduct() {  
15         return new ProductB();  
16     }  
17 }
```

c. 抽象工厂模式：定义了一个创建一系列相关或相互依赖的接口，而无需指定他们的具体类。

```
1 public interface AbstractFactory {
2     Product createProductA();
3     Product createProductB();
4 }
5
6 ---
7
8 public class AbstractProductFactory implements AbstractFactory {
9     @Override
10    public Product createProductA() {
11        return new ProductAFactory().createProduct();
12    }
13
14    @Override
15    public Product createProductB() {
16        return new ProductBFactory().createProduct();
17    }
18 }
19 public class Client {
20    public static void main(String[] args) {
21        AbstractFactory factory = new AbstractProductFactory();
22        Product productA = factory.createProductA();
23        Product productB = factory.createProductB();
24        productA.use();
25        productB.use();
26    }
27 }
28
29
```

3. 观察者模式

定义了一种一对多的关系，让多个观察对象同时监听一个主题对象，主题对象发生变化时，会通知所有的观察者，使他们能够更新自己。（微信朋友圈动态通知、消息通知、邮件通知、广播通知、桌面程序的事件响应）

在观察者模式中，又分为推模型和拉模型两种方式。

● 推模型

主题对象向观察者推送主题的详细信息，不管观察者是否需要，推送的信息通常是主题对象的全部或部分数据。

● 拉模型

主题对象在通知观察者的时候，只传递少量信息。如果观察者需要更具体的信息，由观察者主动到主题对象中获取，相当于是观察者从主题对象中拉数据。一般这种模型的实现中，会把主题对象自身通过 update() 方法传递给观察者，这样在观察者需要获取数据的时候，就可以通过这个引用来获取了。

推

```
public interface Observer {
    void update(Object obj);
}

public interface Subject {
    void attach(Observer observer);
    void detach(Observer observer);
    void notifyObservers(Object obj);
}

public class ConcreteSubject implements Subject {
    private List<Observer> observers = new ArrayList<>();

    @Override
    public void attach(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void detach(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers(Object obj) {
        for (Observer observer : observers) {
            observer.update(obj);
        }
    }
}

public class ConcreteObserver implements Observer {
    @Override
    public void update(Object obj) {
        System.out.println("Observer is notified with object: " + obj);
    }
}

public class Client {
    public static void main(String[] args) {
        Subject subject = new ConcreteSubject();
        Observer observer = new ConcreteObserver();
        subject.attach(observer);
        subject.notifyObservers("Hello World");
        subject.detach(observer);
    }
}
```

直接把变化信息全部推给观察者

拉

```
public interface Observer {
    void update(Object obj);
}

public interface Subject {
    void attach(Observer observer);
    void detach(Observer observer);
    void notifyObservers();
}

public class ConcreteSubject implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private Object obj;

    @Override
    public void attach(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void detach(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(obj);
        }
    }

    public void setObj(Object obj) {
        this.obj = obj;
        notifyObservers();
    }
}

public class ConcreteObserver implements Observer {
    @Override
    public void update(Object obj) {
        System.out.println("Observer is notified with object: " + obj);
    }
}

public class Client {
    public static void main(String[] args) {
        Subject subject = new ConcreteSubject();
        Observer observer = new ConcreteObserver();
        subject.attach(observer);
        ((ConcreteSubject) subject).setObj("Hello World");
        subject.detach(observer);
    }
}
```

这里也可以把整个主题传给观察者，由观察者决定拉取什么信息

4. 装饰模式

动态地给一个对象添加一些额外的职责，就增加功能来说，装饰模式比生成派生类更为灵活。（输入输出流）

文件流 -> 输入输出流 -> 缓冲池流 （层层包装，扩展功能）


```
1   BufferedReader in1 = new BufferedReader(new InputStreamReader(new FileInput
    tStream(file)));//字符流
2   DataInputStream in2 = new DataInputStream(new BufferedInputStream(new File
    InputStream(file)));//字节流
3       // DataInputStream-从数据流读取字节，并将它们装换为正确的基本类型值或字符串
4       // BufferedInputStream-可以通过减少读写次数来提高输入和输出的速度
```

5. 建造者模式

建造者模式的目的是**为了分离对象的属性与创建过程**。

建造者模式是构造方法的一种替代方案，为什么需要建造者模式，我们可以想，假设有一个对象里面有20个属性：

- 1 ● 属性1
- 2 ● 属性2
- 3 ● ...
- 4 ● 属性20

对开发者来说这不是疯了，也就是说我要去使用这个对象，我得去了解每个属性的含义，然后在构造函数或者Setter中一个一个去指定。更加复杂的场景是，这些属性之间是有关联的，比如属性1=A，那么属性2只能等于B/C/D，这样对于开发者来说更是增加了学习成本，开源产品这样的对象相信不会有太多开发者去使用。

为了解决以上的痛点，建造者模式应运而生，对象中属性多，但是通常重要的只有几个，因此建造者模式会**让开发者指定一些比较重要的属性**或者让开发者**指定某几个对象类型**，然后让建造者去实现复杂的构建对象的过程，这就是对象的属性与创建分离。这样对于开发者而言隐藏了复杂的对象构建细节，降低了学习成本，同时提升了代码的可复用性。

```
1  @Data
2  ▼ public class CarBuilder {
3      // 车型
4      private String type;
5
6      // 动力
7      private String power;
8
9  ▼  public Car build() {
10      Assert.assertNotNull(type);
11      Assert.assertNotNull(power);
12      return new Car(this);
13  }
14
15
16  ▼  public CarBuilder type(String type) {
17      this.type = type;
18      return this;
19  }
20
21  ▼  public CarBuilder power(String power) {
22      this.power = power;
23      return this;
24  }
25
26  }
```

```
1  @Test
2  ▼ public void test() {
3      Car car = new CarBuilder()
4          .power("动力一般")
5          .type("紧凑型车")
6          .build();
7
8      System.out.println(JSON.toJSONString(car));
9  }
```

6. 代理模式

- 优点：代理可以协调调用方与被调用方，降低了系统的耦合度。根据代理类型和场景的不同，可以起到控制安全性、减小系统开销等作用。
- 缺点：增加了一层代理处理，增加了系统的复杂度，同时可能会降低系统的相应速度。

Aop 就是使用代理模式来实现的。

```
1 public interface Subject {
2     void request();
3 }
4
5 public class RealSubject implements Subject {
6     @Override
7     public void request() {
8         System.out.println("RealSubject handles the request");
9     }
10 }
11
12
13 ---
14
15 public class Proxy implements Subject {
16     private RealSubject realSubject;
17
18     @Override
19     public void request() {
20         if (realSubject == null) {
21             realSubject = new RealSubject();
22         }
23         System.out.println("Proxy handles the request");
24         // before aop
25         realSubject.request();
26
27         // post aop
28     }
29 }
30
31 public class Client {
32     public static void main(String[] args) {
33         Proxy proxy = new Proxy();
34         proxy.request();
35     }
36 }
37
38
```

7. 策略模式

优缺点

- 优点：策略模式提供了对“开闭原则”的完美支持，用户可以在不修改原有系统的基础上选择算法或行为。干掉复杂难看的if-else。
- 缺点：调用时，必须提前知道都有哪些策略模式类，才能自行决定当前场景该使用何种策略。

```
1  public interface Strategy {
2      void execute();
3  }
4
5  public class ConcreteStrategyA implements Strategy {
6      @Override
7      public void execute() {
8          System.out.println("Executing strategy A");
9      }
10 }
11
12 public class ConcreteStrategyB implements Strategy {
13     @Override
14     public void execute() {
15         System.out.println("Executing strategy B");
16     }
17 }
18
19 public class Context {
20     private Strategy strategy;
21
22     public Context(Strategy strategy) {
23         this.strategy = strategy;
24     }
25
26     public void executeStrategy() {
27         strategy.execute();
28     }
29 }
30
31 public class Client {
32     public static void main(String[] args) {
33         Strategy strategyA = new ConcreteStrategyA();
34         Strategy strategyB = new ConcreteStrategyB();
35
36         Context context = new Context(strategyA);
37         context.executeStrategy();
38
39         context = new Context(strategyB);
40         context.executeStrategy();
41     }
42 }
```