面向对象六大设计原则

开闭原则（Open Closed Principle）

- 核心思想是：对扩展开放，对修改关闭。
- 也就是说，对已经使用的类的改动是通过增加代码进行的，而不是修改现有代码。

单一职责原则（Single Responsiblity Principle）

- 其实就是开发人员经常说的”高内聚，低耦合”
- 也就是说，每个类应该只有一个职责，对外只能提供一种功能，而引起类变化的原因应该只有一个。
- 在设计模式中，所有的设计模式都遵循这一原则。

里式替换原则（Liskov Substitution Principle）

- 核心思想：在任何父类出现的地方都可以用它的子类来替代。
- 或者理解为任何出现父类或者接口的地方，都可以使用子类的实现去替代。
- 也就是说，同一个继承体系中的对象应该有共同的行为特征。

依赖倒转原则（Dependency Inversion Principle）

- 核心思想：要依赖于抽象和接口，不要依赖于具体实现。
- 其实就是说：在应用程序中，所有的类如果使用或依赖于其他的类，则应该依赖这些其他类的抽象类或者接口，而不是直接依赖这些其他类的具体类。
- 为了实现这一原则，就要求我们在编程的时候针对抽象类或者接口编程，而不是针对具体实现编程。

接口分离原则（Interface Segregation Principle）

- 核心思想：不应该强迫程序依赖它们不需要使用的方法。
- 其实就是说：一个接口不需要提供太多的行为，一个接口应该只提供一种对外的功能，不应该把所有的操作都封装到一个接口中。

迪米特原则（最少认知原则，Principle of Least Knowledge）

- 核心思想：一个对象应当对其他对象尽可能少的了解
- 其实就是说：降低各个对象之间的耦合，提高系统的可维护性。在模块之间应该只通过接口编程，而不理会模块的内部工作原理，它可以使各个模块耦合度降到最低，促进软件的复用。

23种设计模式

设计模式概述

- 设计模式（Design pattern）代表了最佳的实践，通常被有经验的面向对象的软件开发人员所采用。
- 设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。
- 设计模式是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。
- 设计模式不是一种方法和技术，而是一种思想。
- 设计模式和具体的语言无关，学习设计模式就是要建立面向对象的思想，尽可能的面向接口编程，低耦合，高内聚，使设计的程序可复用。
- 学习设计模式能够促进对面向对象思想的理解，反之亦然。它们相辅相成。

设计模式的类型

总体来说，设计模式分为三类23种：

- 创建型（5种）：`工厂模式、抽象工厂模式、单例模式、原型模式、构建者模式`
- 结构型（7种）：`适配器模式、装饰模式、代理模式`、外观模式、桥接模式、组合模式、享元模式
- 行为型（11种）：`模板方法模式、策略模式`、观察者模式、中介者模式、状态模式、责任链模式、命令模式、迭代器模式、访问者模式、解释器模式、备忘录模式

Spring源码专题中遇到了哪些设计模式

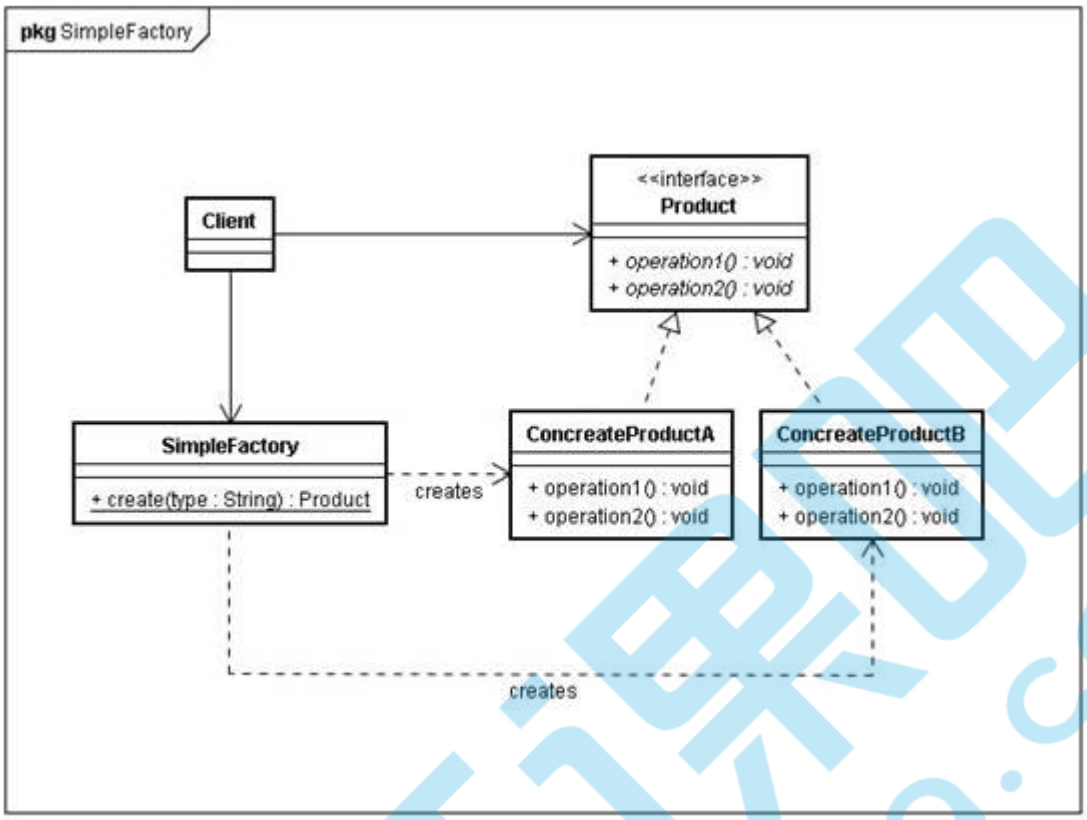
创建型设计模式

简单工厂模式

介绍

工厂类拥有一个工厂方法（`create`），接受了一个参数，通过不同的参数实例化不同的产品类。

图示



优缺点

- 优点：
  - 很明显，简单工厂的特点就是“简单粗暴”，通过一个含参的工厂方法，我们可以实例化任何产品类，上至飞机火箭，下至土豆面条，无所不能。
  - 所以简单工厂有一个别名：上帝类。
- 缺点：
  - 任何”东西“的子类都可以被生产，负担太重。当所要生产产品种类非常多时，工厂方法的代码量可能会很庞大。
  - 在遵循开闭原则（对拓展开放，对修改关闭）的条件下，简单工厂对于增加新的产品，无能为力。因为增加新产品只能通过修改工厂方法来实现。

工厂方法正好可以解决简单工厂的这两个缺点。

示例

- 普通-简单工厂类：

```
public class AnimalFactory {

    //简单工厂设计模式（负担太重、不符合开闭原则）
    public static Animal createAnimal(String name){
        if ("cat".equals(name)) {
            return new Cat();
        }else if ("dog".equals(name)) {
            return new Dog();
        }else if ("cow".equals(name)) {
            return new Dog();
```

```
        }else{
            return null;
        }
    }
}
```

- 静态方法工厂

```
//该简单工厂，也称为静态方法工厂
public class AnimalFactory2 {

    public static Dog createDog(){
        return new Dog();
    }

    public static Cat createCat(){
        return new Cat();
    }
}
```

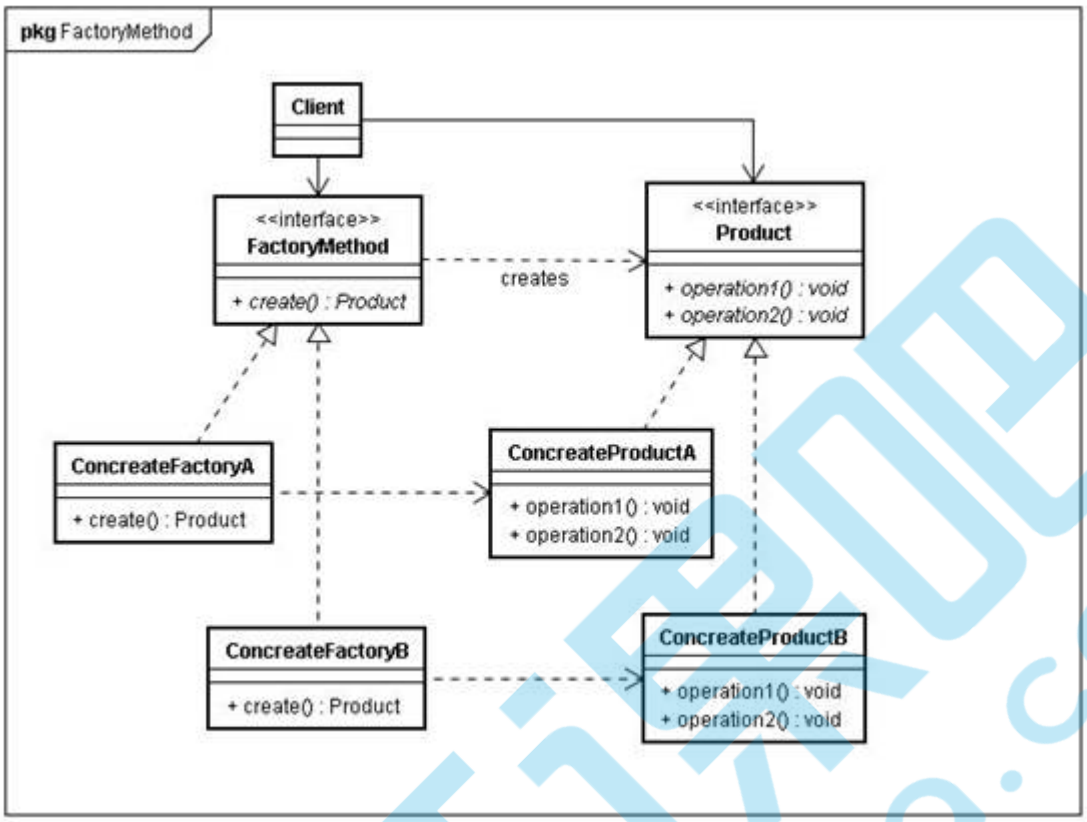
工厂方法模式

介绍

工厂方法是针对每一种产品提供一个工厂类。

通过不同的工厂实例来创建不同的产品实例。

图示



优缺点

- 优点：
  - 工厂方法模式就很好的减轻了工厂类的负担，把某一类/某一种东西交由一个工厂生产；（对应简单工厂的缺点1）
  - 同时增加某一类”东西“并不需要修改工厂类，只需要添加生产这类”东西“的工厂即可，使得工厂类符合开放-封闭原则。
- 缺点：
  - 对于某些可以形成产品族的情况处理比较复杂。

示例

- 抽象出来的工厂对象

```
// 抽象出来的动物工厂----它只负责生产一种产品
public abstract class AnimalFactory {
    // 工厂方法
    public abstract Animal createAnimal();
}
```

- 具体的工厂对象1

```java
// 具体的工厂实现类
public class CatFactory extends AnimalFactory {

    @Override
    public Animal createAnimal() {
        return new Cat();
    }
}
```

- 具体的工厂对象2

```java
//具体的工厂实现类
public class DogFactory extends AnimalFactory {

    @Override
    public Animal createAnimal() {
        return new Dog();
    }
}
```

抽象工厂模式

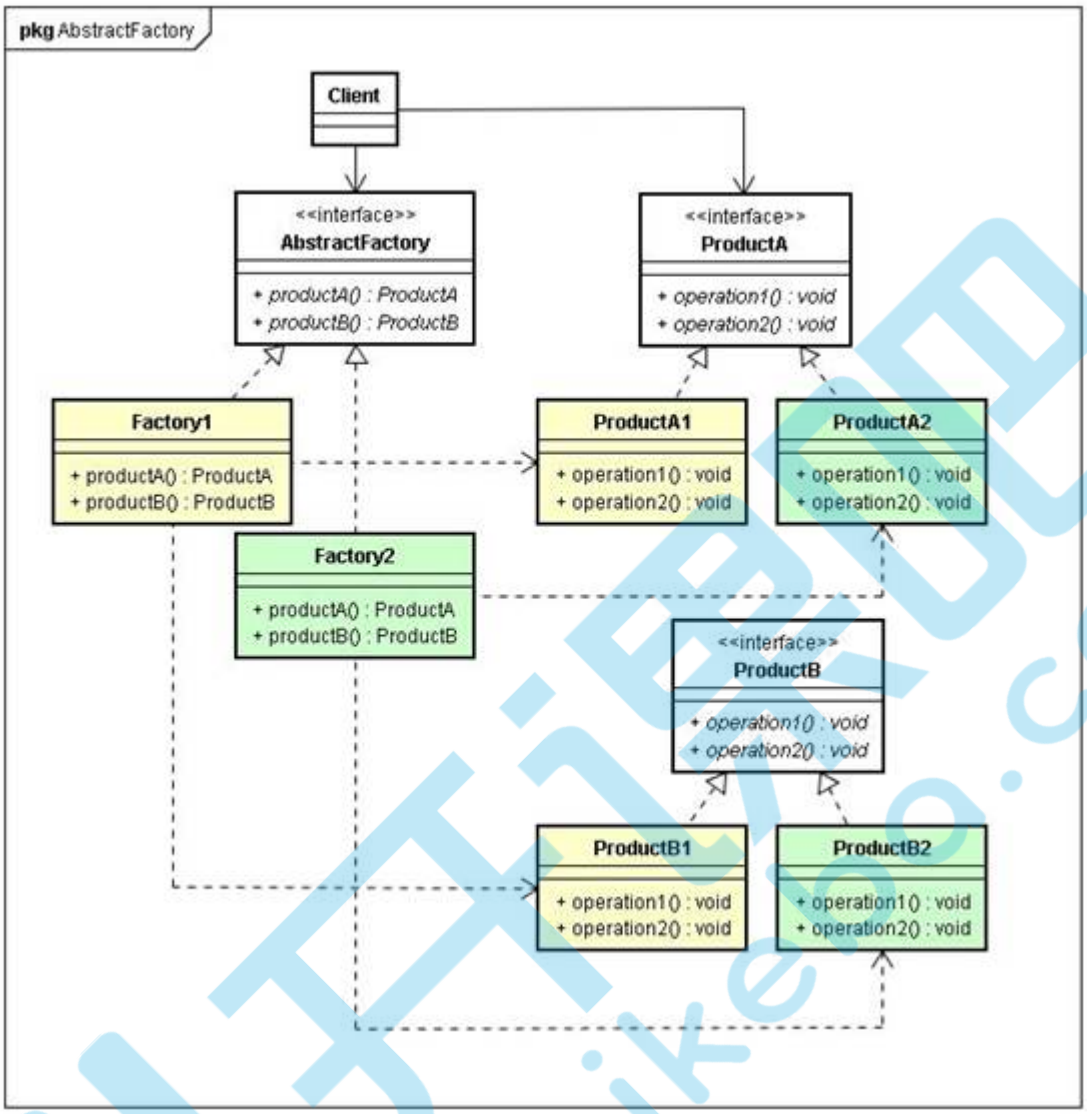
介绍

- 抽象工厂是应对产品族概念的。

  > 例如，汽车可以分为轿车、SUV、MPV等，也分为奔驰、宝马等。我们可以将奔驰的所有车看作是一个产品族，而将宝马的所有车看作是另一个产品族。分别对应两个工厂，一个是奔驰的工厂，另一个是宝马的工厂。与工厂方法不同，奔驰的工厂不只是生产具体的某一个产品，而是一族产品（奔驰轿车、奔驰SUV、奔驰MPV）。“抽象工厂”的“抽象”指的是就是这个意思。

- 上边的工厂方法模式是一种极端情况的抽象工厂模式（即只生产一种产品的抽象工厂模式），而抽象工厂模式可以看成是工厂方法模式的一种推广。

图示



工厂模式区别

- 简单工厂 ： 用来生产同一等级结构中的任意产品。（不支持拓展增加产品）
- 工厂方法 ： 用来生产同一等级结构中的固定产品。（支持拓展增加产品）
- 抽象工厂 ： 用来生产不同产品族的全部产品。（不支持拓展增加产品；支持增加产品族）

示例

待补充

单例模式

介绍

单例对象（Singleton）是一种常用的设计模式。在Java应用中，单例对象能保证在一个JVM中，该对象只有一个实例存在。这样的模式有几个好处：

1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。

2、省去了new操作符，降低了系统内存的使用频率，减轻GC压力。

示例

- 饿汉式单例

```
public class Student1 {
    // 2：成员变量初始化本身对象
    private static Student1 student = new Student1();

    // 1：构造私有
    private Student1() {
    }

    // 3：对外提供公共方法获取对象
    public static Student1 getSingletonInstance() {
        return student;
    }
}
```

- 懒汉式单例

```
public class Student5 {

    private Student5() {
    }
    /*
     * 此处使用一个内部类来维护单例 JVM在类加载的时候，是互斥的，所以可以由此保证线程安全问题
     */
    private static class SingletonFactory {
        private static Student5 student = new Student5();
    }
    /* 获取实例 */
    public static Student5 getSingletonInstance() {
        return SingletonFactory.student;
    }
}
```

原型模式

介绍

> 原型模式虽然是创建型的模式，但是与工厂模式没有关系，从名字即可看出，该模式的思想就是将一个对象作为原型，对其进行复制、克隆，产生一个和原对象类似的新对象。

示例

- 先创建一个原型类：

```java
public class Prototype implements Cloneable {

    public Object clone() throws CloneNotSupportedException {
        Prototype proto = (Prototype) super.clone();
        return proto;
    }
}
```

很简单，一个原型类，只需要实现Cloneable接口，覆写clone方法，此处clone方法可以改成任意的名称，因为Cloneable接口是个空接口，你可以任意定义实现类的方法名，如cloneA或者cloneB，因为此处的重点是super.clone()这句话，super.clone()调用的是Object的clone()方法，而在Object类中，clone()是native的，具体怎么实现，此处不再深究。

在这儿，我将结合对象的浅复制和深复制来说一下，首先需要了解对象深、浅复制的概念：

- 浅复制：将一个对象复制后，基本数据类型的变量都会重新创建，而引用类型，指向的还是原对象所指向的。
- 深复制：将一个对象复制后，不论是基本数据类型还有引用类型，都是重新创建的。简单来说，就是深复制进行了完全彻底的复制，而浅复制不彻底。

- 写一个深浅复制的例子

```java
public class Prototype implements Cloneable, Serializable {

    private static final long serialVersionUID = 1L;
    private String string;

    private SerializableObject obj;

    /* 浅复制 */
    public Object clone() throws CloneNotSupportedException {
        Prototype proto = (Prototype) super.clone();
        return proto;
    }

    /* 深复制 */
    public Object deepClone() throws IOException, ClassNotFoundException {

        /* 写入当前对象的二进制流 */
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(bos);
        oos.writeObject(this);

        /* 读出二进制流产生的新对象 */
        ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());
        ObjectInputStream ois = new ObjectInputStream(bis);
        return ois.readObject();
    }

    public String getString() {
        return string;
```

```
30      }
31
32      public void setString(String string) {
33          this.string = string;
34      }
35
36      public SerializableObject getObj() {
37          return obj;
38      }
39
40      public void setObj(SerializableObject obj) {
41          this.obj = obj;
42      }
43
44  }
45
46  class SerializableObject implements Serializable {
47      private static final long serialVersionUID = 1L;
48  }
```

构建者模式

介绍

建造者模式的定义是：将一个复杂对象的构造与它的表示分离，使同样的构建过程可以创建不同的表示，这样的设计模式被称为建造者模式。

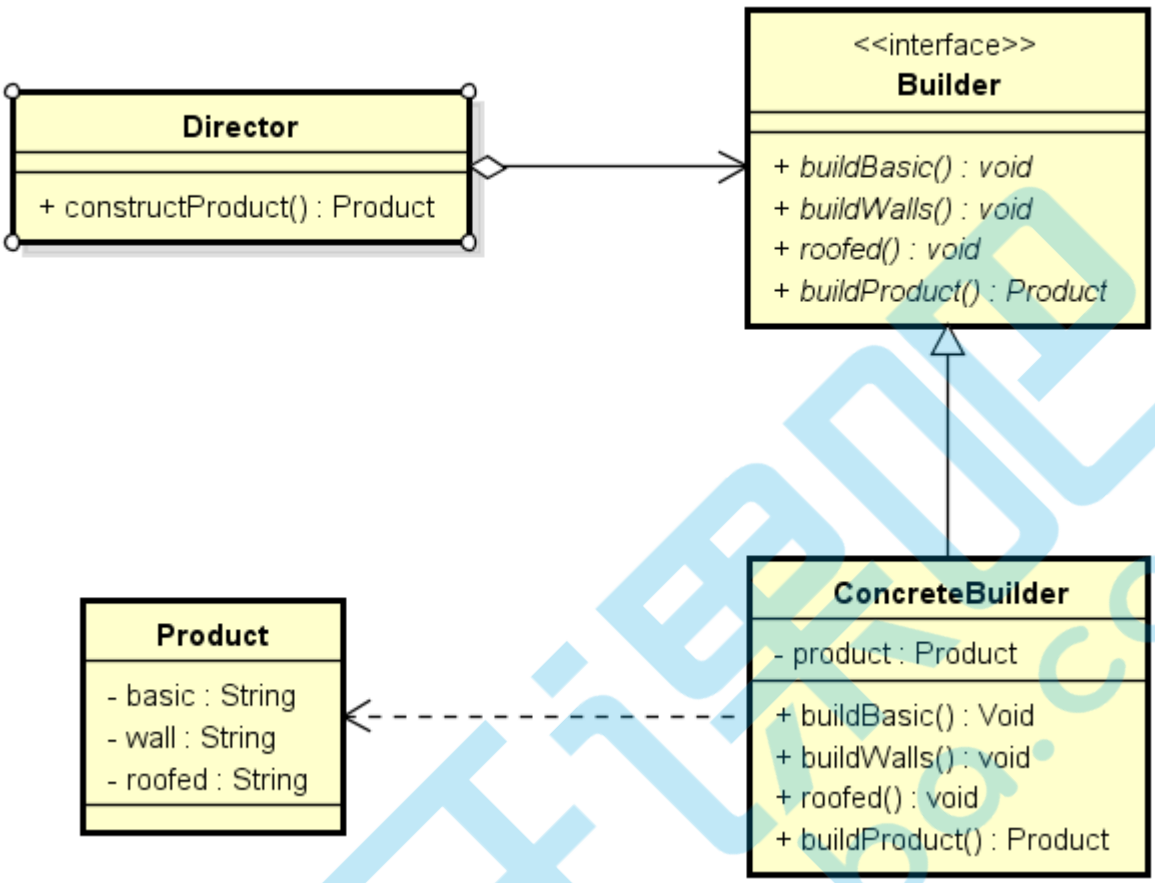
建造者模式的角色定义，在建造者模式中存在以下4个角色：

1. builder:为创建一个产品对象的各个部件指定抽象接口。
2. ConcreteBuilder:实现Builder的接口以构造和装配该产品的各个部件，定义并明确它所创建的表示，并提供一个检索产品的接口。
3. Director:构造一个使用Builder接口的对象。
4. Product:表示被构造的复杂对象。ConcreteBuilder创建该产品的内部表示并定义它的装配过程，包含定义组成部件的类，包括将这些部件装配成最终产品的接口。

工厂模式和构建者模式的区别

构建者模式和工厂模式很类似，区别在于构建者模式是一种个性化产品的创建。而工厂模式是一种标准化的产品创建。

图示



- 导演类：按照一定的顺序或者一定的需求去组装一个产品。
- 构造者类：提供对产品的不同个性化定制，最终创建出产品。
- 产品类：最终的产品

示例

- 构建者

```
// 构建器
public class StudentBuilder {

    // 需要构建的对象
    private Student student = new Student();

    public StudentBuilder id(int id) {
        student.setId(id);
        return this;
    }

    public StudentBuilder name(String name) {
        student.setName(name);
        return this;
    }

    public StudentBuilder age(int age) {
        student.setAge(age);
```

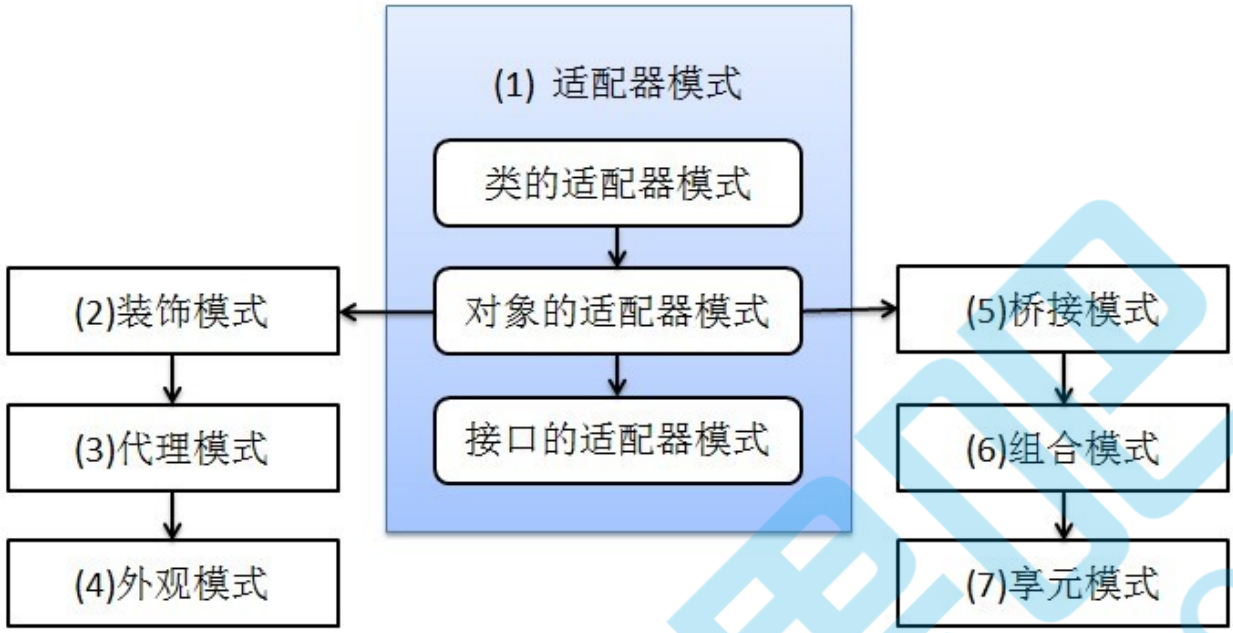
```
        return this;
    }

    public StudentBuilder father(String fatherName) {
        Father father = new Father();
        father.setName(fatherName);
        student.setFather(father);
        return this;
    }

    // 构建对象
    public Student build() {
        return student;
    }
}
```

- 导演类

```
// 导演类/测试类
public class BuildDemo {

    public static void main(String[] args) {

        StudentBuilder builder = new StudentBuilder();
        // 决定如何创建一个Student
        Student student = builder.age(1).name("zhangsan").father("zhaosi").build();
        System.out.println(student);

    }
}
```

结构型设计模式

我们接着讨论设计模式，上篇文章我讲完了5种创建型模式，这章开始，我将讲下7种结构型模式：**适配器模式、装饰模式、代理模式、外观模式、桥接模式、组合模式、享元模式。**其中对象的适配器模式是各种模式的起源，我们看下面的图：

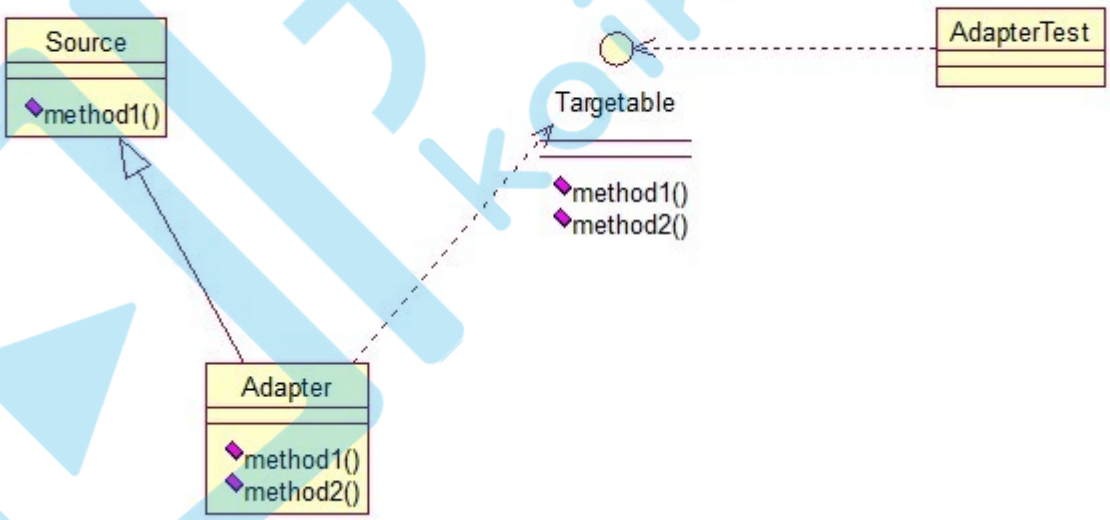


6、适配器模式（Adapter）

适配器模式

适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。

类的适配器模式



核心思想就是：有一个 `Source` 类，拥有一个方法，待适配，目标接口是 `Targetable` ，通过 `Adapter` 类，将 `Source` 的功能扩展到 `Targetable` 里，看代码：

```java
public class Source {
    public void method1() {
        System.out.println("this is original method!");
    }
}
```

```java
public interface Targetable {
    /* 与原类中的方法相同 */
    public void method1();
    /* 新类的方法 */
    public void method2();
}
```

```java
public class Adapter extends Source implements Targetable {
    @Override
    public void method2() {
        System.out.println("this is the targetable method!");
    }
}
```

`Adapter`类继承`Source`类，实现`Targetable`接口，下面是测试类：

```java
public class AdapterTest {
    public static void main(String[] args) {
        Targetable target = new Adapter();
        target.method1();
        target.method2();
    }
}
```

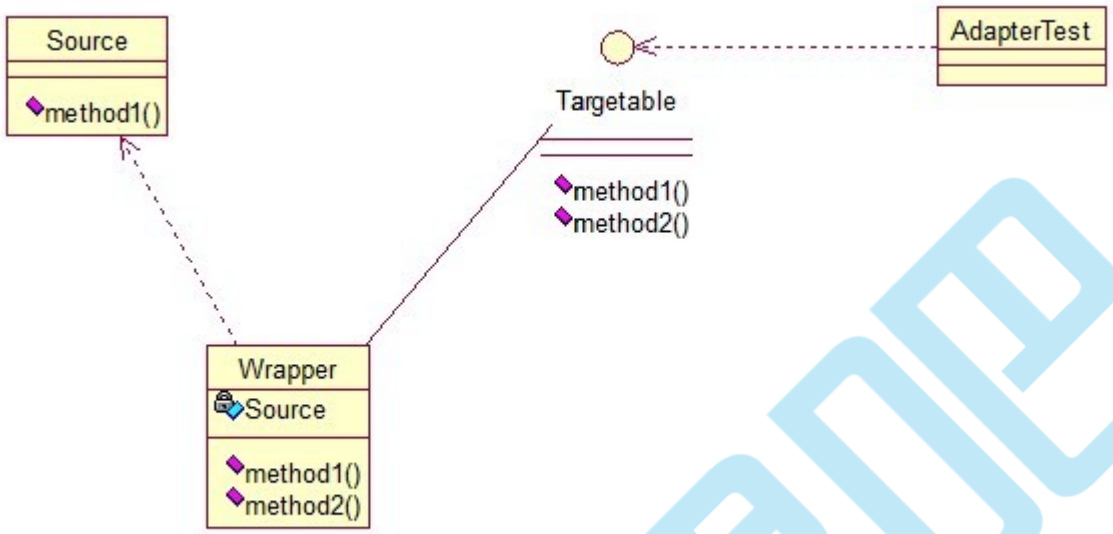
输出：

```
this is original method!
this is the targetable method!
```

这样`Targetable`接口的实现类就具有了`Source`类的功能。

对象的适配器模式

基本思路和类的适配器模式相同，只是将`Adapter`类作修改，这次不继承`Source`类，而是持有`Source`类的实例，以达到解决兼容性的问题。看图：



只需要修改`Adapter`类的源码即可：

```
public class Wrapper implements Targetable {
    private Source source;
    public Wrapper(Source source){
        super();
        this.source = source;
    }
    @Override
     public void method2() {
         System.out.println("this is the targetable method!");
     }
     @Override
     public void method1() {
         source.method1();
     }
 }
```

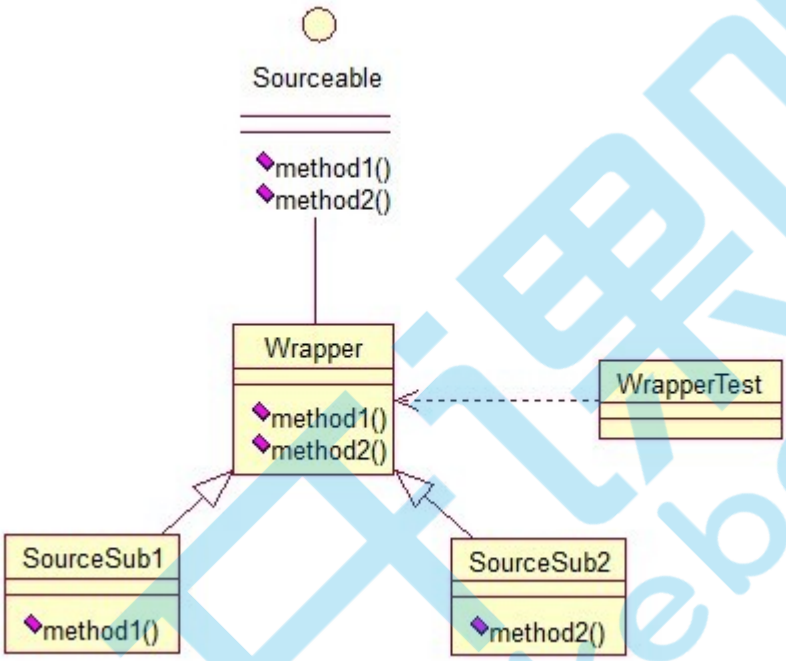
测试类：

```
public class AdapterTest {
    public static void main(String[] args) {
        Source source = new Source();
        Targetable target = new Wrapper(source);
        target.method1();
        target.method2();
    }
}
```

输出与第一种一样，只是适配的方法不同而已。

接口的适配器模式

第三种适配器模式是**接口的适配器模式**，接口的适配器是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。看一下类图：



这个很好理解，在实际开发中，我们也常会遇到这种接口中定义了太多的方法，以致于有时我们在一些实现类中并不是都需要。

看代码：

```
public interface Sourceable {
    public void method1();
    public void method2();
}
```

抽象类Wrapper2：

```
public abstract class Wrapper2 implements Sourceable{

    public void method1(){}
    public void method2(){}
}
```

```
public class SourceSub1 extends Wrapper2 {
    public void method1(){
        System.out.println("the sourceable interface's first Sub1!");
    }
}
```

```
public class SourceSub2 extends Wrapper2 {
    public void method2(){
        System.out.println("the sourceable interface's second Sub2!");
    }
}
```

测试类：

```
public class WrapperTest {

    public static void main(String[] args) {
        Sourceable source1 = new SourceSub1();
        Sourceable source2 = new SourceSub2();

        source1.method1();
        source1.method2();
        source2.method1();
        source2.method2();
    }
}
```

测试输出：

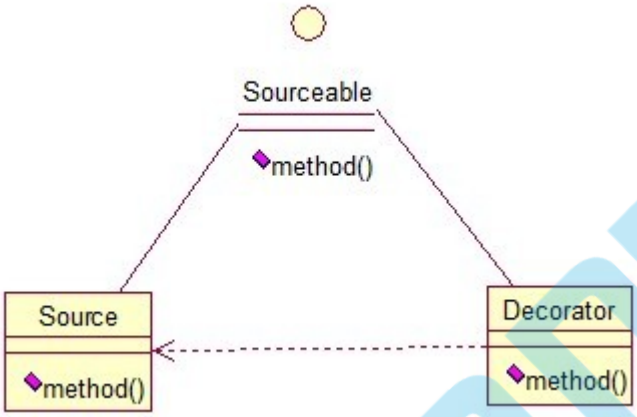
```
the sourceable interface's first Sub1!
the sourceable interface's second Sub2!
```

达到了我们的效果！

> 讲了这么多，总结一下三种适配器模式的应用场景：
>
> - 类的适配器模式：当希望将一个类转换成满足另一个新接口的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。
> - 对象的适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个Wrapper类，持有原类的一个实例，在Wrapper类的方法中，调用实例的方法就行。
> - 接口的适配器模式：当不希望实现一个接口中所有的方法时，可以创建一个抽象类Wrapper，实现所有方法，我们写别的类的时候，继承抽象类即可。

装饰模式

顾名思义，装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例，关系图如下：



Source类是被装饰类，Decorator类是一个装饰类，可以为Source类动态的添加一些功能，代码如下：

```
public interface Sourceable {
    public void method();
}
```

```
public class Source implements Sourceable {
    @Override
    public void method() {
        System.out.println("the original method!");
    }
}
```

```
 public class Decorator implements Sourceable {
     private Sourceable source;
     public Decorator(Sourceable source){
         super();
         this.source = source;
     }
     @Override
     public void method() {
         System.out.println("before decorator!");
         source.method();
         System.out.println("after decorator!");
     }
 }
```

测试类：

```
public class DecoratorTest {
    public static void main(String[] args) {
        Sourceable source = new Source();
        Sourceable obj = new Decorator(source);
        obj.method();
    }
}
```

输出：

```
before decorator!
the original method!
after decorator!
```

装饰器模式的应用场景：

1. 需要扩展一个类的功能。
2. 动态的为一个对象增加功能，而且还能动态撤销。（继承不能做到这一点，继承的功能是静态的，不能动态增删。）

缺点：

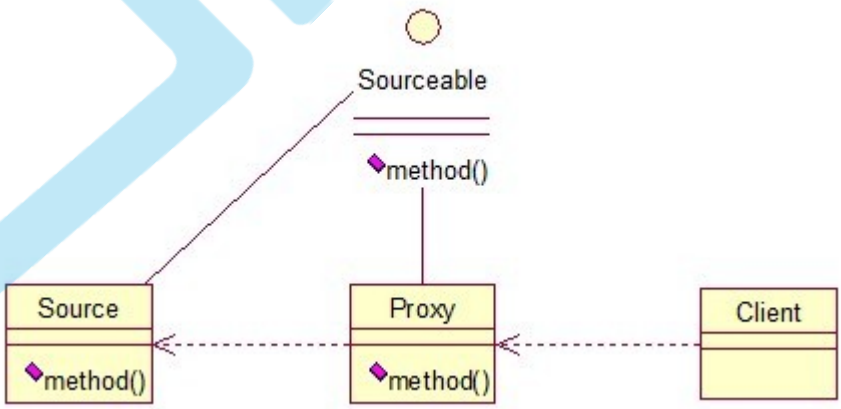
产生过多相似的对象，不易排错！

代理模式

其实每个模式名称就表明了该模式的作用，代理模式就是多一个代理类出来，替原对象进行一些操作。代理又分为动态代理和静态代理

静态代理

比如我们在租房子的时候回去找中介，为什么呢？因为你对该地区房屋的信息掌握的不够全面，希望找一个更熟悉的人去帮你做，此处的代理就是这个意思。再如我们有的时候打官司，我们需要请律师，因为律师在法律方面有专长，可以替我们进行操作，表达我们的想法。先来看看关系图：



根据上文的阐述，代理模式就比较容易的理解了，我们看下代码：

```java
public interface Sourceable {
    public void method();
}
```

```java
public class Source implements Sourceable {
    @Override
    public void method() {
        System.out.println("the original method!");
    }
}
```

```java
public class Proxy implements Sourceable {
    private Source source;
    public Proxy(){
        super();
        this.source = new Source();
    }
    @Override
    public void method() {
        before();
        source.method();
        atfer();
    }
    private void atfer() {
        System.out.println("after proxy!");
    }
    private void before() {
        System.out.println("before proxy!");
    }
}
```

测试类：

```java
public class ProxyTest {
    public static void main(String[] args) {
        Sourceable source = new Proxy();
        source.method();
    }
}
```

输出：

```
before proxy!
the original method!
after proxy!
```

代理模式的应用场景：

如果已有的方法在使用的时候需要对原有的方法进行改进，此时有两种办法：

1. 修改原有的方法来适应。这样违反了“对扩展开放，对修改关闭”的原则。
2. 就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式。

使用代理模式，可以将功能划分的更加清晰，有助于后期维护！

动态代理

JDK动态代理

```
public class JDKProxyFactory implements InvocationHandler {

    // 目标对象的引用
    private Object target;

    // 通过构造方法将目标对象注入到代理对象中
    public JDKProxyFactory(Object target) {
        super();
        this.target = target;
    }

    /**
     * @return
     */
    public Object getProxy() {

        // 如何生成一个代理类呢？
        // 1、编写源文件
        // 2、编译源文件为class文件
        // 3、将class文件加载到JVM中(ClassLoader)
        // 4、将class文件对应的对象进行实例化（反射）

        // Proxy是JDK中的API类
        // 第一个参数：目标对象的类加载器
        // 第二个参数：目标对象的接口
        // 第二个参数：代理对象的执行处理器
        Object object = Proxy.newProxyInstance(target.getClass().getClassLoader(),
target.getClass().getInterfaces(),
                this);

        return object;
    }

    /**
     * 代理对象会执行的方法
```

```
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
{
        Method method2 = target.getClass().getMethod("saveUser", null);
        Method method3 = Class.forName("com.sun.proxy.$Proxy4").getMethod("saveUser",
null);
        System.out.println("目标对象的方法:" + method2.toString());
        System.out.println("目标接口的方法:" + method.toString());
        System.out.println("代理对象的方法:" + method3.toString());
        System.out.println("这是jdk的代理方法");
        // 下面的代码，是反射中的API用法
        // 该行代码，实际调用的是[目标对象]的方法
        // 利用反射，调用[目标对象]的方法
        Object returnValue = method.invoke(target, args);

        return returnValue;
    }
}
```

CGLib动态代理

```
public class CgLibProxyFactory implements MethodInterceptor {

    /**
     * @param clazz
     * @return
     */
    public Object getProxyByCgLib(Class clazz) {
        // 创建增强器
        Enhancer enhancer = new Enhancer();
        // 设置需要增强的类的类对象
        enhancer.setSuperclass(clazz);
        // 设置回调函数
        enhancer.setCallback(this);
        // 获取增强之后的代理对象
        return enhancer.create();
    }

    /***
     * Object proxy:这是代理对象，也就是[目标对象]的子类
     * Method method:[目标对象]的方法
     * Object[] arg:参数
     * MethodProxy methodProxy：代理对象的方法
     */
    @Override
    public Object intercept(Object proxy, Method method, Object[] arg, MethodProxy
methodProxy) throws Throwable {
        // 因为代理对象是目标对象的子类
        // 该行代码，实际调用的是父类目标对象的方法
        System.out.println("这是cglib的代理方法");
```

```
29
30          // 通过调用子类[代理类]的invokeSuper方法，去实际调用[目标对象]的方法
31          Object returnValue = methodProxy.invokeSuper(proxy, arg);
32          // 代理对象调用代理对象的invokeSuper方法，而invokeSuper方法会去调用目标类的invoke方法完
    成目标对象的调用
33
34          return returnValue;
35      }
36  }
```

行为型设计模式

其他设计模式

MVC设计模式

委托设计模式