1 单例模式

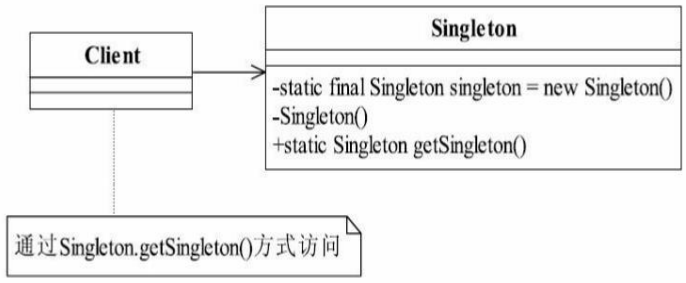
1.1 概念

单例模式（Singleton Pattern）：确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。

1.2 类图

饿汉	双重检查加锁，适用于 JDK1.5 及其以上

public class Singleton {
private static final Singleton singleton = new Singleton();
 	
private Singleton(){ //限制产生多个对象
}
//通过该方法获得实例对象
public static Singleton getSingleton(){
return singleton;
}
//其他方法，尽量是 static
public static void doSomething(){
}
}
	public class Singleton {
	private **volatile** static Singleton instance;
	private Singleton(){
	}
	public static Singleton getInstance(){
	if(instance == null){
	synchronized (Singleton.class) {
	if(instance == null)
	instance = new Singleton();
	}
	}
	return instance;
	}
	}



1.3 优点&缺点

● 减少了内存开支，特别是一个对象需要频繁地创建、销毁时，而且创建或销毁时性能又无法优化。

● 减少了系统的性能开销，当一个对象的产生需要比较多的资源时，则可以通过在应用启动时直接产生一个单例对象，然后用永久驻留内存的方式来解决。

● 单例模式可以避免对资源的多重占用，例如一个写文件动作，由于只有一个实例存在内存中，避免对同一个资源文件的同时写操作。

● 单例模式可以在系统设置全局的访问点，优化和共享资源访问，例如可以设计一个单例类，负责所有数据表的映射处理。

✧ 单例模式一般没有接口，扩展很困难，若要扩展，只能修改代码。因为接口对单例模式是没有任何意义的，它要求“自行实例化”，并且提供单一实例、接口或抽象类是不可能被实例化的。

✧ 单例模式对测试是不利的。在并行开发环境中，如果单例模式没有完成，是不能进行测试的。

✧ 单例模式与单一职责原则有冲突。一个类应该只实现一个逻辑，而不关心它是否是单例的，是不是要单例取决于环境。

1.4 最佳实践

如在 Spring 中，每个 Bean 默认就是单例的，这样做的优点是 Spring 容器可以管理这些 Bean 的生命期，决定什么时候创建出来，什么时候销毁，销毁的时候要如何处理，等等。

2 工厂方法模式

工厂模式主要是为创建对象提供过渡接口，以便将创建对象的具体过程屏蔽隔离起来，达到提高灵活性的目的。工厂模式可以分为三类：
1）简单工厂模式（Simple Factory） 2）工厂方法模式（Factory Method） 3）抽象工厂模式（Abstract Factory）

2.1 简单工厂模式

简单工程其实不是一个设计模式，反而比较像是一种变成习惯。**简单工厂模式的工厂类一般是使用静态方法**，通过接收的参数的不同来返回不同的对象实例。不修改代码的话，是无法扩展的。

产品类

```
public interface Human {
    public void getColor();
}

public class YellowHuman implements Human {
    public void getColor(){
        System.out.println("皮肤颜色是黄色的！");
    }
}
public class WhiteHuman implements Human {
    public void getColor(){
        System.out.println("皮肤颜色是白色的！");
    }
}
```

工厂类

```
public class HumanFactory {
    public static <T extends Human> T createHuman(Class<T> c){
        Human human=null;
        try {
            human = (Human)Class.forName(c.getName()).newInstance();
        } catch (Exception e) {
            System.out.println("人种生成错误！");
        }
        return (T)human;
    }
}
```

```
public class HumanFactory {
        public Human createHuman(String type){
                Human human=null;
                If(type.equals(“yellow”)){
                        human = new YellowHuman();
                } else if(type.equals(“white”)){
                        human = new WhiteHuman();
                }
                Return human;
}
```

2.2 工厂方法模式

产品类

```
public interface Human {
    public void getColor();
}
public class YellowHuman implements Human {
    public void getColor(){
        System.out.println("皮肤颜色是黄色的！");
    }
}
public class WhiteHuman implements Human {
    public void getColor(){
        System.out.println("皮肤颜色是白色的！");
    }
}
```

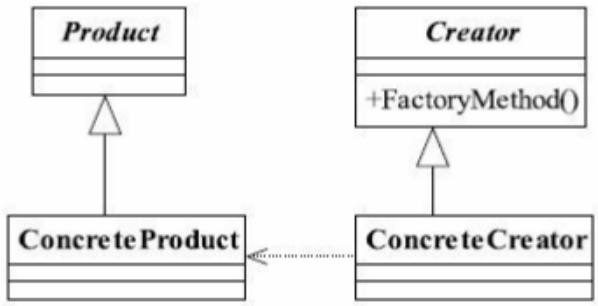
工厂类

```
public interface AbstractHumanFactory {
            Human    createHuman();
}
public class YellowHumanFactory implements AbstractHumanFactory {
    public Human createHuman(){
        return new YellowHuman();
    }
}
public class WhiteHumanFactory implements AbstractHumanFactory {
    public Human createHuman(){
        return new WhiteHuman();
    }
}
```

2.2.1 工厂方法模式的定义

定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法使一个类的实例化延迟到其子类。

2.2.2 UML 图



产品类	具体产品类

public abstract class Product {
  public void method1(){ //产品类的公共方法
  //业务逻辑处理
  }
 	
  public abstract void method2();//抽象方法
}
	public class ConcreteProduct1 extends Product {
	  public void method2() {
	  //业务逻辑处理
	  }
	}
	public class ConcreteProduct2 extends Product {
	  public void method2() {
	  //业务逻辑处理
	  }
	}

抽象工厂类	具体工厂类

public abstract class Creator {
/*
* 创建一个产品对象，其输入参数类型可以自行设置
* 通常为 String、Enum、Class 等，当然也可以为空
*/
public abstract <T extends Product> T createProduct(Class<T> c);
}
	public class ConcreteCreator extends Creator {
	  public <T extends Product> T createProduct(Class<T> c){
	  Product product=null;
	  try {
	product = (Product)Class.forName(c.getName()).newInstance();
	  } catch (Exception e) {
	  //异常处理
	  }
	  return (T)product;
	  }
	}

场景类：

```
public class Client {
    public static void main(String[] args) {
        Creator creator = new ConcreteCreator();
        Product product = creator.createProduct(ConcreteProduct1.class);
        //继续业务处理
    }
}
```

2.3.3 优点&缺点

- ✧ 首先，工厂方法模式是 new 一个对象的替代品，所以在所有需要生成对象的地方都可以使用，但是需要慎重地考虑是否要增加一个工厂类进行管理，增加代码的复杂度。
- ✧ 其次，需要灵活的、可扩展的框架时，可以考虑采用工厂方法模式。
- ✧ 再次，工厂方法模式可以用在异构项目中。
- ✧ 最后，可以使用在测试驱动开发的框架下。

2.3.4 扩展

负责生成单例的工厂类

```
public class SingletonFactory {
    private static Singleton singleton;
    static{
        try {
            Class cl= Class.forName(Singleton.class.getName());
            Constructor constructor=cl.getDeclaredConstructor();//获得无参构造
            constructor.setAccessible(true);   //设置无参构造是可访问的
            singleton = (Singleton)constructor.newInstance();   //产生一个实例对象
        } catch (Exception e) {
            //异常处理
        }
    }
    public static Singleton getSingleton(){
        return singleton;
    }
}
```

2.3 抽象工厂

```
public interface Human {
    public void getColor();//每个人种都有相应的颜色
    public void talk();   //人类会说话
    public void getSex();   //每个人都有性别
}
```

```
public abstract class AbstractWhiteHuman implements Human {
    public void getColor(){
        System.out.println("白色人种的皮肤颜色是白色的！");
    }
    public void talk() {
        System.out.println("白色人一般说的都是单字节。");
    }
}
。。。   。。。
```

```
public class FemaleYellowHuman extends AbstractYellowHuman {
    public void getSex() {
        System.out.println("黄人女性");
    }
}
public class MaleYellowHuman extends AbstractYellowHuman {
    public void getSex() {
        System.out.println("黄人男性");
    }
}
。。。   。。。
```

```
public interface HumanFactory {
    public Human createYellowHuman();
    public Human createWhiteHuman();
    public Human createBlackHuman();
}
```

```
public class FemaleFactory implements HumanFactory {
    public Human createBlackHuman() {
        return new FemaleBlackHuman();
    }
    public Human createWhiteHuman() {
        return new FemaleWhiteHuman();
    }
```

	public Human createYellowHuman() {
	return new FemaleYellowHuman();
	}
	}
	… …

2.3.1 抽象工厂定义

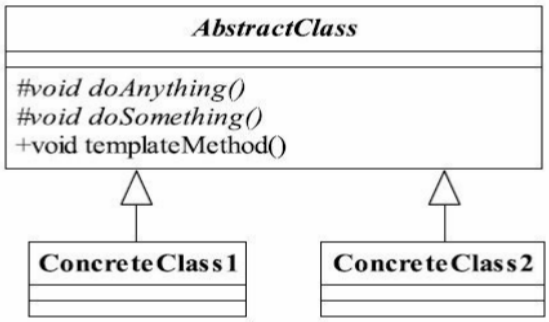
抽象工厂模式（Abstract Factory Pattern）是一种比较常用的模式，其定义如下：为创建一组相关或相互依赖的对象提供一个接口，而且无须指定它们的具体类。）

3 模板方法

3.1 定义

定义一个操作中的算法的框架，而将一些步骤延迟到子类中。使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

3.2 UML



基本方法也叫做基本操作，是由子类实现的方法，并且在模板方法被调用。
模板方法可以有一个或几个，一般是一个具体方法，也就是一个框架，实现对基本方法的调度，完成固定的逻辑。
※※**注意**：为了防止恶意的操作，**一般模板方法都加上 final 关键字，不允许被覆写**。

3.3 代码

```
public abstract class AbstractClass {
    protected abstract void doSomething();   //基本方法
    protected abstract void doAnything();   //基本方法
    public void templateMethod(){   //模板方法
        this.doAnything();
        this.doSomething();
    }
}
```

```
public class ConcreteClass1 extends AbstractClass {
    protected void doAnything() {
        //业务逻辑处理
    }
    protected void doSomething() {
        //业务逻辑处理
    }
}
public class ConcreteClass2 extends AbstractClass {
    protected void doAnything() {
        //业务逻辑处理
    }
    protected void doSomething() {
        //业务逻辑处理
    }
}
```

※※**注意**：抽象模板中的基本方法尽量设计为 protected 类型，符合迪米特法则，不需要暴露的属性或方法尽量不要设置为 protected 类型。

3.4 优点&缺点

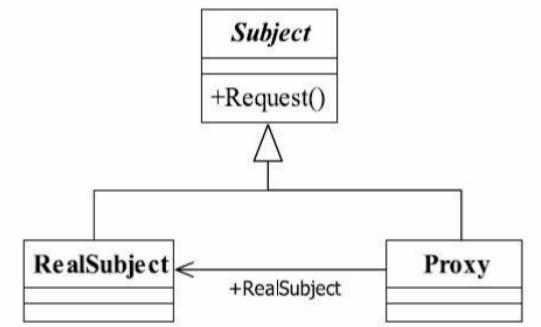
- ✧ 封装不变部分，扩展可变部分
- ✧ 提取公共部分代码，便于维护
- ✧ 行为由父类控制，子类实现

4 代理模式

4.1 定义

代理模式（Proxy Pattern）为其他对象提供一种代理以控制对这个对象的访问。即通过代理对象访问目标对象.这样做的好处是:可以在目标对象实现的基础上,增强额外的功能操作,即扩展目标对象的功能.

4.2 UML



4.3 代码

//抽象主题类

```
public interface Subject {
    public void request();   //定义一个方法
}
```

//真实主题类

```
public class RealSubject implements Subject {
    public void request() {
        //业务逻辑处理
    }
}
```

//代理类

```
public class Proxy implements Subject {
    private Subject subject = null;   //要代理哪个实现类
    public Proxy(){
        this.subject = new Proxy();
    }
    public Proxy(Object...objects ){   //传递代理者
    }
    public void request() {
        this.before();
        this.subject.request();
        this.after();
    }
    private void before(){… …}   //预处理
    private void after(){ … …}    //善后处理
}
```

4.4 静态代理

静态代理实现中，一个委托类对应一个代理类，代理类在编译期间就已经确定。

```
//抽象主题类
interface Subject {
    void request();
}
```

```
//真实主题类
class RealSubject implements Subject {
    public void request(){
        System.out.println("RealSubject");
    }
}
```

```
//代理类
class Proxy implements Subject {
    private Subject subject;
    public Proxy(Subject subject){
        this.subject = subject;
    }
    public void request(){
        System.out.println("begin");
        subject.request();
        System.out.println("end");
    }
}
```

```
//测试代码
public class ProxyTest {
    public static void main(String args[]) {
        RealSubject subject = new RealSubject();
        Proxy p = new Proxy(subject);
        p.request();
    }
}
```

4.5 动态代理类

动态代理有以下特点:①代理对象,不需要实现接口；②代理对象的生成,是利用 JDK 的 API,动态的在内存中构建代理对象(需要我们指定创建代理对象/目标对象实现的接口的类型)；③动态代理也叫做:JDK 代理,接口代理。

```
public interface Service {
    //目标方法
    public abstract void add();
}
```

```
public class UserServiceImpl implements Service {
    public void add() {
        System.out.println("This is add service");
    }
}
```

```
class MyInvocatioHandler implements InvocationHandler {      //定义代理类
    private Object target;
    public MyInvocatioHandler(Object target) {
        this.target = target;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("-----before-----");
        Object result = method.invoke(target, args);
        System.out.println("-----end-----");
        return result;
    }
    public Object getProxy() {     // 生成代理对象
        ClassLoader loader = Thread.currentThread().getContextClassLoader();
        Class<?>[] interfaces = target.getClass().getInterfaces();
        return Proxy.newProxyInstance(loader, interfaces, this);
    }
}
```

测试代码：

```
public class ProxyTest {
    public static void main(String[] args) {
        Service service = new UserServiceImpl();
        MyInvocatioHandler handler = new MyInvocatioHandler(service);
        Service serviceProxy = (Service)handler.getProxy();
        serviceProxy.add();
    }
}
```

4.6 CGLIB 代理

```
public class UserDao {     // 目标对象,没有实现任何接口
    public void save() {
        System.out.println("----已经保存数据!----");
    }
}
```

```
public class ProxyFactory implements MethodInterceptor{
    private Object target;
    public ProxyFactory(Object target) {
        this.target = target;
    }
    public Object getProxyInstance(){    //给目标对象创建一个代理对象
        Enhancer en = new Enhancer();     //1.工具类
        en.setSuperclass(target.getClass());    //2.设置父类
        en.setCallback(this);    //3.设置回调函数
        return en.create();    //4.创建子类(代理对象)
    }
    @Override
    public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {
        System.out.println("开始事务...");
        Object returnValue = method.invoke(target, args);     //执行目标对象的方法
        System.out.println("提交事务...");
        return returnValue;
    }
}
```

测试代码：

```
public class App {
    public void test(){
        UserDao target = new UserDao();    //目标对象
        UserDao proxy = (UserDao)new ProxyFactory(target).getProxyInstance();     //代理对象
        proxy.save();    //执行代理对象的方法
    }
}
```

4.7 优点 & 缺点

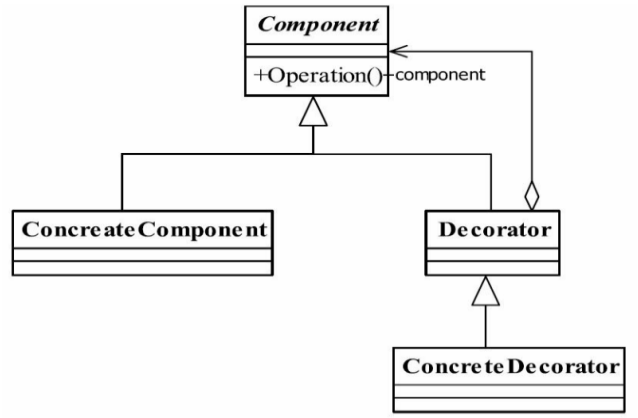
- ✧ **职责清晰：**真实的角色就是实现实际的业务逻辑，不用关心其他非本职责的事务，通过后期的代理完成一件事务，附带的结果就是编程简洁清晰。
- ✧ **高扩展性：**具体主题角色是随时都会发生变化的，只要它实现了接口，甭管它如何变化，都 OK。

5 装饰模式

5.1 定义

装饰模式（Decorator Pattern）：动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式相比生成子类更为灵活。

5.2 UML



在类图中，有四个角色需要说明：

- ✧ Component 抽象构件：是一个接口或者是抽象类，定义我们最核心的对象，也就是最原始的对象。
  注意 在装饰模式中，必然有一个最基本、最核心、最原始的接口或抽象类充当 Component 抽象构件。
- ✧ ConcreteComponent 具体构件：是最核心、最原始、最基本的接口或抽象类的实现，你要装饰的就是它。
- ✧ Decorator 装饰角色：一般是一个抽象类，实现接口或者抽象方法，它里面可不一定有抽象的方法呀，在它的属性里必然有一个 private 变量指向 Component 抽象构件。
- ✧ 具体装饰角色： ConcreteDecoratorA 和 ConcreteDecoratorB 是两个具体的装饰类，你要把你最核心的、最原始的、最基本的东西装饰成其他东西。

5.3 代码

```
public abstract class Component {
        //抽象的方法
        public abstract void operate();
}
```

```
public class ConcreteComponent extends Component {
        @Override     //具体实现
        public void operate() {
                System.out.println("do Something");
        }
}
```

```
public abstract class Decorator extends Component
{
        private Component component = null;
        //通过构造函数传递被修饰者
        public Decorator(Component _component){
                this.component = _component;
        }
        @Override     //委托给被修饰者执行
        public void operate() {
                this.component.operate();
        }
}
```

```
public class ConcreteDecorator1 extends Decorator {
        public ConcreteDecorator1(Component _component){
                super(_component);
        }
        private void method1(){     //定义自己的修饰方法
                System.out.println("method1 修饰");
        }
        public void operate(){     //重写父类的 Operation 方法
                this.method1();
                super.operate();
        }
}
```

	public class ConcreteDecorator2 extends Decorator {
	public ConcreteDecorator2(Component _component){
	super(_component);
	}
	private void method2(){ //定义自己的修饰方法
	System.out.println("method2 修饰");
	}
	public void operate(){ //重写父类的 Operation 方法
	super.operate();
	this.method2();
	}
	}

测试代码：

```
public class Client {
    public static void main(String[] args) {
        Component component = new ConcreteComponent();
        component = new ConcreteDecorator1(component);      //第一次修饰
        component = new ConcreteDecorator2(component);     //第二次修饰
        component.operate();     //修饰后运行
    }
}
```

5.4 优点&缺点

- ✧ 装饰类和被装饰类可以独立发展，而不会相互耦合。换句话说，Component 类无须知道 Decorator 类，Decorator 类是从外部来扩展 Component 类的功能，而 Decorator 也不用知道具体的构件。
- ✧ 装饰模式是继承关系的一个替代方案。我们看装饰类 Decorator，不管装饰多少层，返回的对象还是 Component，实现的还是 is-a 的关系。
- ✧ 装饰模式可以动态地扩展一个实现类的功能。

- ➢ **多层的装饰是比较复杂的**。为什么会复杂呢？就像剥洋葱一样，你剥到了最后才发现是最里层的装饰出现了问题，想象一下工作量吧，因此，尽量减少装饰类的数量，以便降低系统的复杂度。

5.5 最佳应用

装饰模式是对继承的有力补充。装饰模式可以替代继承，解决我们类膨胀的问题。同时，你还要知道继承是静态地给类增加功能，而装饰模式则是动态地增加功能。**装饰模式还有一个非常好的优点：扩展性非常好**。

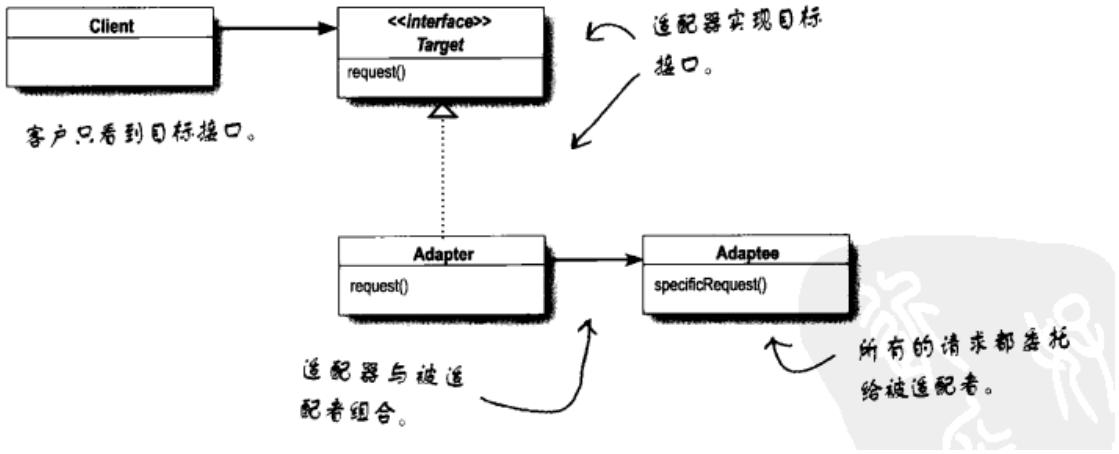
简单点说，三个继承关系 Father、**Son**、GrandSon 三个类，我要在 Son 类上增强一些功能怎么办？我想你会坚决地顶回去！不允许，对了，为什么呢？你增强的功能是修改 Son 类中的方法吗？增加方法吗？对 GrandSon 的影响呢？特别是 GrandSon 有多个的情况，你会怎么办？这个评估的工作量就够你受的，所以这是不允许的，那还是要解决问题的呀，怎么办？**通过建立 SonDecorator 类来修饰 Son**，相当于创建了一个新的类，这个对原有程序没有变更，通过扩展很好地完成了这次变更。

6 适配器模式

6.1 定义

适配器模式（Adapter Pattern）：将一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。适配器模式又叫做变压器模式，也叫做包装模式（Wrapper）。

6.2 对象适配器 UML



先来看看适配器模式的三个角色。

- ✧ **Target 目标角色：**期望接口，目标角色是一个已经在正式运行的角色，你不可能去修改角色中的方法，你能做的就是如何去实现接口中的方法，而且通常情况下，**目标角色是一个接口或者是抽象类**。
- ✧ **Adaptee 源角色：**你想把谁转换成目标角色，这个“谁”就是源角色，它是已经存在的、运行良好的类或对象，经过适配器角色的包装，它会成为一个崭新、靓丽的角色。
- ✧ **Adapter 适配器角色：**适配器模式的核心角色，其他两个角色都是已经存在的角色，而**适配器角色是需要新建立的，它的职责非常简单：把源角色转换为目标角色，怎么转换？通过继承或是类关联的方式。**

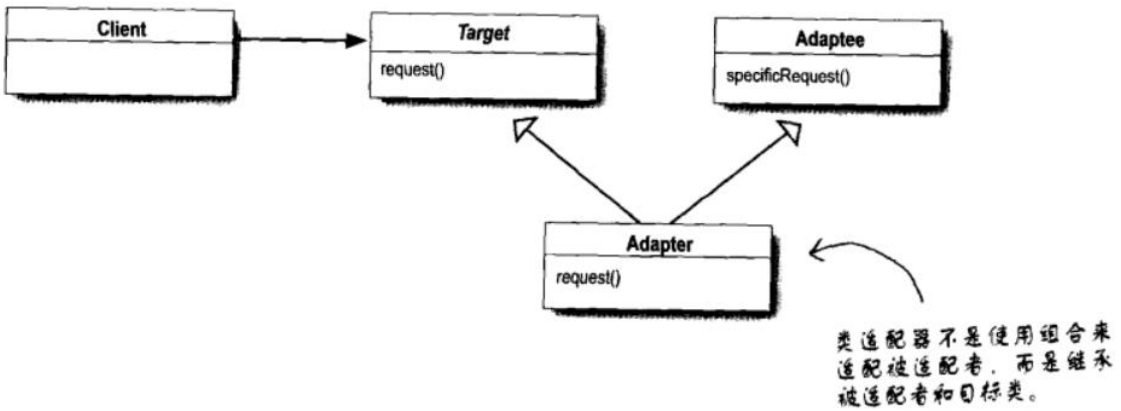
6.3 对象适配器代码

//目标角色
public interface Target {
　　public void request();　//目标角色有自己的方法
}
	public class ConcreteTarget implements Target {
	　　public void request() {
	　　System.out.println("ConcreteTarget!"); }
	}

//源角色
public class Adaptee {
　　public void doSomething(){　//原有的业务逻辑
　　System.out.println("Adaptee");
　　}
}
	//适配器角色
	public class Adapter **extends Adaptee implements Target** {
	　　public void request() {
	　　super.doSomething();
	　　}
	}

6.4 类的适配器

“类”适配器需要多重继承才能够实现它，在 Java 中是不可能的。适配器继承 target 和 adaptee。

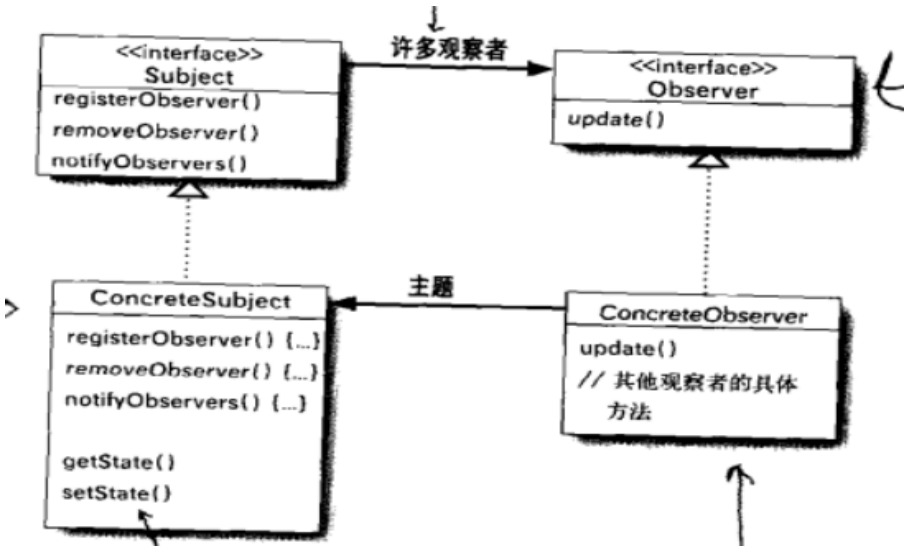


7 观察者模式

7.1 定义

观察者模式（Observer Pattern）也叫做发布订阅模式（Publish/subscribe）,其定义如下：定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。

7.2 UML 图



7.3 代码

```
//被观察者
public interface Subject {
    public void addObserver(Observer o);
    public void delObserver(Observer o);
    public void notifyObservers(String str);
}
```

```
//具体被观察者
public class ConcreteSubject implements  Subject {
    // 存放观察者
    private Vector<Observer> obsVector = new Vector<Observer>();
    @Override
    public void addObserver (Observer o) {
        obsVector.add(o);
    }
    @Override
    public void delObserver(Observer o){
        this.obsVector.remove(o);
    }
    @Override
    public void notifyObservers (String str){
        for (Watcher watcher : list){
            watcher.update(str);
        }
    }
```

```
// 观察者
public interface Observer {
    //更新方法
    public void update();
}
```

```
//具体观察者
public class ConcreteObserver implements Observer {
    public void update(String str) {
        System.out.println(str);
    }
}
```

测试代码：

```
public class Client {
    public static void main(String[] args) {
        Subject subject = new ConcreteSubject();     //创建一个被观察者
        Observer obs= new ConcreteObserver();   //定义一个观察者
        subject.addObserver(obs);     //观察者观察被观察者
        subject. notifyObservers (“XXXX”);   //观察者开始活动了
    }
}
```

8 生产者消费者

8.1 方式三：BlockingQueue 实现

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
public class BlockingQueueConsumerProducer {
    public static void main(String[] args) {
        Resource3 resource = new Resource3();
        ProducerThread3 p = new ProducerThread3(resource); //生产者线程
        ConsumerThread3 c1 = new ConsumerThread3(resource); //多个消费者
        ConsumerThread3 c2 = new ConsumerThread3(resource);
        ConsumerThread3 c3 = new ConsumerThread3(resource);
        p.start();
        c1.start();
        c2.start();
        c3.start();
    }
}

//消费者线程
class ConsumerThread3 extends Thread {
    private Resource3 resource3;
    public ConsumerThread3(Resource3 resource) {
        this.resource3 = resource;
    }
    public void run() {
        while (true) {
            try {
                Thread.sleep((long) (1000 * Math.random()));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            resource3.remove();
        }
    }
}

//生产者线程
```

```
class ProducerThread3 extends Thread {
    private Resource3 resource3;
    public ProducerThread3(Resource3 resource) {
        this.resource3 = resource;
    }
    public void run() {
        while (true) {
            try {
                Thread.sleep((long) (1000 * Math.random()));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            resource3.add();
        }
    }
}

class Resource3 {
    private BlockingQueue resourceQueue = new LinkedBlockingQueue(10);
    public void add() {     //向资源池中添加资源
        try {
            resourceQueue.put(1);
            System.out.println("生产者" + Thread.currentThread().getName()
                    + "生产一件资源," + "当前资源池有" + resourceQueue.size() + "个资源");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    //向资源池中移除资源
    public void remove() {
        try {
            resourceQueue.take();
            System.out.println("消费者" + Thread.currentThread().getName() +
                    "消耗一件资源," + "当前资源池有" + resourceQueue.size()+ "个资源");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

8.2 方式二：synchronized、wait 和 notify

```
public class ProducerConsumerWithWaitNofity {
    public static void main(String[] args) {
        Resource resource = new Resource();
        ProducerThread p1 = new ProducerThread(resource); //生产者线程
        ConsumerThread c1 = new ConsumerThread(resource); //消费者线程
        p1.start();   c1.start();
   }
```

```
}
//公共资源
class Resource {
    private int num = 0;        //当前资源数量
    private int size = 10;      //资源池中允许存放的资源数目
    //从资源池中取走资源
    public synchronized void remove() {
        if (num > 0) {
            num--;
            System.out.println("消费者" + Thread.currentThread().getName() + "消耗一件资源");
            notifyAll();//通知生产者生产资源
        } else {
            try {
                wait();  //如果没有资源，则消费者进入等待状态
                System.out.println("消费者" + Thread.currentThread().getName() + "进入等待");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    //向资源池中添加资源
    public synchronized void add() {
        if (num < size) {
            num++;
            System.out.println(Thread.currentThread().getName() + "生产一件资源");
            notifyAll();  //通知等待的消费者
        } else {0
            try {
                wait();//生产者进入等待状态，并释放锁
                System.out.println(Thread.currentThread().getName() + "线程进入等待");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

//消费者线程
class ConsumerThread extends Thread {
    private Resource resource;
    public ConsumerThread(Resource resource) {
        this.resource = resource;
    }
    @Override
    public void run() {
        while (true) {
            try {
```

```
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            resource.remove();
        }
    }
}

//生产者线程
class ProducerThread extends Thread {
    private Resource resource;
    public ProducerThread(Resource resource) {
        this.resource = resource;
    }
    @Override
    public void run() {
        while (true) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            resource.add();
        }
    }
}
```

8.3 lock 和 condition 的 await、signalAll

```
//使用 Lock 和 Condition 解决生产者消费者问题
public class LockCondition {
    public static void main(String[] args) {
        Lock lock = new ReentrantLock();
        Condition producerCondition = lock.newCondition();
        Condition consumerCondition = lock.newCondition();
        Resource2 resource = new Resource2(lock, producerCondition, consumerCondition);
        ProducerThread2 producer1 = new ProducerThread2(resource);  //生产者线程
        ConsumerThread2 consumer1 = new ConsumerThread2(resource); //消费者线程
        ConsumerThread2 consumer2 = new ConsumerThread2(resource);
        ConsumerThread2 consumer3 = new ConsumerThread2(resource);
        producer1.start();
        consumer1.start();
        consumer2.start();
        consumer3.start();
    }
}

//消费者线程
```

```
class ConsumerThread2 extends Thread {
    private Resource2 resource;
    public ConsumerThread2(Resource2 resource) {
        this.resource = resource;
    }
    public void run() {
        while (true) {
            try {
                Thread.sleep((long) (1000 * Math.random()));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            resource.remove();
        }
    }
}

//生产者线程
class ProducerThread2 extends Thread {
    private Resource2 resource;
    public ProducerThread2(Resource2 resource) {
        this.resource = resource;
    }
    public void run() {
        while (true) {
            try {
                Thread.sleep((long) (1000 * Math.random()));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            resource.add();
        }
    }
}

//公共资源类
class Resource2 {
    private int num = 0;//当前资源数量
    private int size = 10;//资源池中允许存放的资源数目
    private Lock lock;
    private Condition producerCondition;
    private Condition consumerCondition;
    public Resource2(Lock lock, Condition producerCondition, Condition consumerCondition) {
        this.lock = lock;
        this.producerCondition = producerCondition;
        this.consumerCondition = consumerCondition;
    }
```

```
    //向资源池中添加资源
    public void add() {
        lock.lock();
        try {
            if (num < size) {
                num++;
                System.out.println(Thread.currentThread().getName() +"生产一件资源");
                consumerCondition.signalAll();  //唤醒等待的消费者
            } else {
                try {
                    producerCondition.await();
                    System.out.println(Thread.currentThread().getName() + "线程进入等待");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        } finally {
            lock.unlock();
        }
    }

    //从资源池中取走资源
    public void remove() {
        lock.lock();
        try {
            if (num > 0) {
                num--;
                System.out.println("消费者"+Thread.currentThread().getName()+ "消耗一件资源");
                producerCondition.signalAll();//唤醒等待的生产者
            } else {
                try {
                    consumerCondition.await();
                    System.out.println(Thread.currentThread().getName() + "线程进入等待");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        } finally {
            lock.unlock();
        }
    }
}
```