

常用的设计模式汇总，超详细！

Java团长 3月1日

- 1.单例模式
- 2.观察者模式
- 3.装饰者模式
- 4.适配器模式
- 5.工厂模式
- 6.代理模式



来源：cnblogs.com/ILoke-Yang/p/8054466.html

单例模式

简单点说，就是一个应用程序中，某个类的实例对象只有一个，你没有办法去new，因为构造器是被private修饰的，一般通过getInstance()的方法来获取它们的实例。

getInstance()的返回值是一个对象的引用，并不是一个新的实例，所以不要错误的理解成多个对象。单例模式实现起来也很容易，直接看demo吧

```
public class Singleton {  
  
    private static Singleton singleton;  
  
    private Singleton() {  
    }  
}
```

```
public static Singleton getInstance() {  
    if (singleton == null) {  
        singleton = new Singleton();  
    }  
    return singleton;  
}
```

按照我的习惯，我恨不得写满注释，怕你们看不懂，但是这个代码实在太简单了，所以我没写任何注释，如果这几行代码你都看不明白的话，那你可以洗洗睡了，等你睡醒了再来看我的博客说不定能看懂。

上面的是最基本的写法，也叫懒汉写法（线程不安全）下面我再公布几种单例模式的写法：

懒汉式写法（线程安全）

```
public class Singleton {  
    private static Singleton instance;  
    private Singleton () {}  
    public static synchronized Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

饿汉式写法

```
public class Singleton {  
    private static Singleton instance = new Singleton();  
    private Singleton () {}  
    public static Singleton getInstance() {  
        return instance;  
    }  
}
```

静态内部类

```
public class Singleton {  
    private static class SingletonHolder {  
        private static final Singleton INSTANCE = new Singleton();  
    }  
    private Singleton () {}  
    public static final Singleton getInstance() {  
        return SingletonHolder.INSTANCE;  
    }  
}
```

枚举

```
public enum Singleton {  
    INSTANCE;  
    public void whateverMethod() {  
    }  
}
```

这种方式是Effective Java作者Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象，可谓是很坚强的壁垒啊，不过，个人认为由于1.5中才加入enum特性，用这种方式写不免让人感觉生疏。

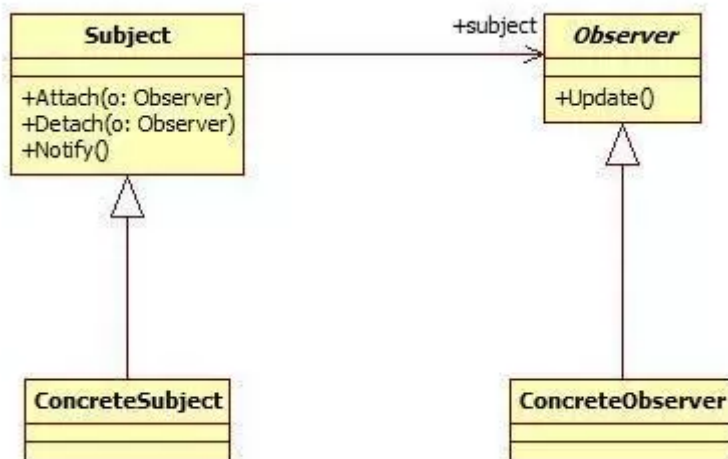
双重校验锁

```
public class Singleton {  
    private volatile static Singleton singleton;  
    private Singleton () {}  
    public static Singleton getSingleton() {  
        if (singleton == null) {  
            synchronized (Singleton.class) {  
                if (singleton == null) {  
                    singleton = new Singleton();  
                }  
            }  
        }  
        return singleton;  
    }  
}
```

总结：我个人比较喜欢静态内部类写法和饿汉式写法，其实这两种写法能够应付绝大多数情况了。其他写法也可以选择，主要还是看业务需求吧。

观察者模式

对象间一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。



观察者模式UML图

看不懂图的人端着小板凳到这里来，给你举个栗子：假设有三个人，小美（女，22），小王和小李。小美很漂亮，小王和小李是两个程序猿，时刻关注着小美的一举一动。有一天，小美说了一句：“谁来陪我打游戏啊。”这句话被小王和小李听到了，结果乐坏了，蹭蹭蹭，没一会儿，小王就冲到小美家门口了，在这里，小美是被观察者，小王和小李是观察者，被观察者发出一条信息，然后观察者们进行相应的处理，看代码：

```
public interface Person {
    //小王和小李通过这个接口可以接收到小美发过来的消息
    void getMessage(String s);
}
```

这个接口相当于小王和小李的电话号码，小美发送通知的时候就会拨打getMessage这个电话，拨打电话就是调用接口，看不懂没关系，先往下看

```
public class LaoWang implements Person {

    private String name = "小王";

    public LaoWang() {
    }
}
```

```
@Override
public void getMessage(String s) {
    System.out.println(name + "接到了小美打过来的电话，电话内容是：" + s);
}

}

public class LaoLi implements Person {

    private String name = "小李";

    public LaoLi() {
    }

    @Override
    public void getMessage(String s) {
        System.out.println(name + "接到了小美打过来的电话，电话内容是：->" + s);
    }

}
```

代码很简单，我们再看看小美的代码：

```
public class XiaoMei {
    List<Person> list = new ArrayList<Person>();
    public XiaoMei() {
    }

    public void addPerson(Person person) {
        list.add(person);
    }

    //遍历list，把自己的通知发送给所有暗恋自己的人
    public void notifyPerson() {
        for(Person person:list) {
            person.getMessage("你们过来吧，谁先过来谁就能陪我一起玩儿游戏!");
        }
    }

}
```

我们写一个测试类来看一下结果对不对

```
public class Test {  
    public static void main(String[] args) {  
  
        XiaoMei xiao_mei = new XiaoMei();  
        LaoWang lao_wang = new LaoWang();  
        LaoLi lao_li = new LaoLi();  
  
        //小王和小李在小美那里都注册了一下  
        xiao_mei.addPerson(lao_wang);  
        xiao_mei.addPerson(lao_li);  
  
        //小美向小王和小李发送通知  
        xiao_mei.notifyPerson();  
    }  
}
```

完美~

装饰者模式

对已有的业务逻辑进一步的封装，使其增加额外的功能，如Java中的IO流就使用了装饰者模式，用户在使用的时候，可以任意组装，达到自己想要的效果。举个栗子，我想吃三明治，首先我需要一根大大的香肠，我喜欢吃奶油，在香肠上面加一点奶油，再放一点蔬菜，最后再用两片面包夹一下，很丰盛的一顿午饭，营养又健康。（ps：不知道上海哪里有卖好吃的三明治的，求推荐~）那我们应该怎么来写代码呢？首先，我们需要写一个Food类，让其他所有食物都来继承这个类，看代码：

```
public class Food {  
  
    private String food_name;  
  
    public Food() {  
    }  
  
    public Food(String food_name) {  
        this.food_name = food_name;  
    }  
  
    public String make() {  
        return food_name;  
    }  
}
```

```
};  
}
```

代码很简单，我就不解释了，然后我们写几个子类继承它：

//面包类

```
public class Bread extends Food {  
  
    private Food basic_food;  
  
    public Bread(Food basic_food) {  
        this.basic_food = basic_food;  
    }  
  
    public String make() {  
        return basic_food.make()+"面包";  
    }  
}
```

//奶油类

```
public class Cream extends Food {  
  
    private Food basic_food;  
  
    public Cream(Food basic_food) {  
        this.basic_food = basic_food;  
    }  
  
    public String make() {  
        return basic_food.make()+"奶油";  
    }  
}
```

//蔬菜类

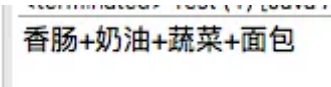
```
public class Vegetable extends Food {  
  
    private Food basic_food;  
  
    public Vegetable(Food basic_food) {  
        this.basic_food = basic_food;  
    }  
  
    public String make() {  
        return basic_food.make()+"蔬菜";  
    }  
}
```

```
}  
  
}
```

这几个类都是差不多的，构造方法传入一个Food类型的参数，然后在make方法中加入一些自己的逻辑，如果你还是看不懂为什么这么写，不急，你看看我的Test类是怎么写的，一看你就明白了

```
public class Test {  
    public static void main(String[] args) {  
        Food food = new Bread(new Vegetable(new Cream(new Food("香肠"))));  
        System.out.println(food.make());  
    }  
}
```

看到没有，一层一层封装，我们从里往外看：最里面我new了一个香肠，在香肠的外面我包裹了一层奶油，在奶油的外面我又加了一层蔬菜，最外面我放的是面包，是不是很形象，哈哈~ 这个设计模式简直跟现实生活中一摸一样，看懂了吗？ 我们看看运行结果吧



运行结果

一个三明治就做好了~

适配器模式

将两种完全不同的事物联系到一起，就像现实生活中的变压器。假设一个手机充电器需要的电压是20V，但是正常的电压是220V，这时候就需要一个变压器，将220V的电压转换成20V的电压，这样，变压器就将20V的电压和手机联系起来了。

```
public class Test {  
    public static void main(String[] args) {  
        Phone phone = new Phone();  
        VoltageAdapter adapter = new VoltageAdapter();  
        phone.setAdapter(adapter);  
        phone.charge();  
    }  
}
```



```
// 手机类
class Phone {

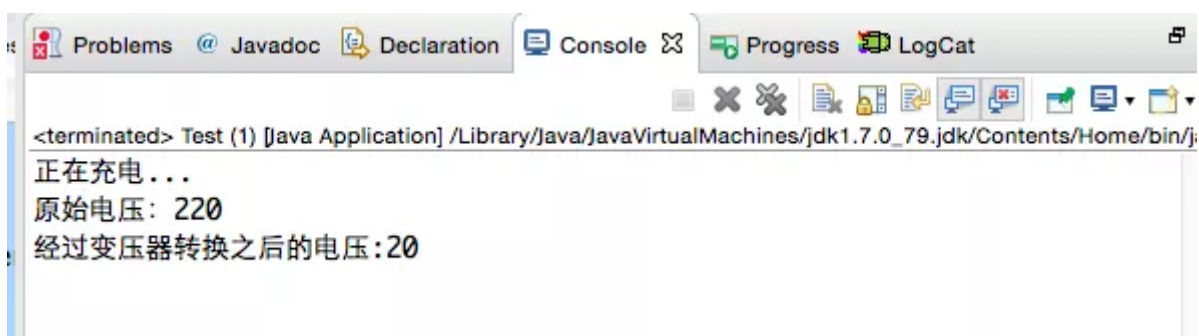
    public static final int V = 220; // 正常电压220v, 是一个常量

    private VoltageAdapter adapter;

    // 充电
    public void charge() {
        adapter.changeVoltage();
    }

    public void setAdapter(VoltageAdapter adapter) {
        this.adapter = adapter;
    }
}

// 变压器
class VoltageAdapter {
    // 改变电压的功能
    public void changeVoltage() {
        System.out.println("正在充电...");
        System.out.println("原始电压: " + Phone.V + "V");
        System.out.println("经过变压器转换之后的电压:" + (Phone.V - 200) + "V");
    }
}
```



工厂模式

简单工厂模式：一个抽象的接口，多个抽象接口的实现类，一个工厂类，用来实例化抽象的接口

// 抽象产品类

```
abstract class Car {  
    public void run();  
  
    public void stop();  
}
```

// 具体实现类

```
class Benz implements Car {  
    public void run() {  
        System.out.println("Benz开始启动了。。。。");  
    }  
  
    public void stop() {  
        System.out.println("Benz停车了。。。。");  
    }  
}
```

```
class Ford implements Car {  
    public void run() {  
        System.out.println("Ford开始启动了。。。");  
    }  
  
    public void stop() {  
        System.out.println("Ford停车了。。。。");  
    }  
}
```

// 工厂类

```
class Factory {  
    public static Car getCarInstance(String type) {  
        Car c = null;  
        if ("Benz".equals(type)) {  
            c = new Benz();  
        }  
        if ("Ford".equals(type)) {  
            c = new Ford();  
        }  
        return c;  
    }  
}
```

```
public class Test {  
  
    public static void main(String[] args) {
```

```
        Car c = Factory.getCarInstance("Benz");
        if (c != null) {
            c.run();
            c.stop();
        } else {
            System.out.println("造不了这种汽车。。。");
        }
    }
}
```

工厂方法模式：有四个角色，抽象工厂模式，具体工厂模式，抽象产品模式，具体产品模式。不再是由一个工厂类去实例化具体的产品，而是由抽象工厂的子类去实例化产品

```
// 抽象产品角色
public interface Moveable {
    void run();
}

// 具体产品角色
public class Plane implements Moveable {
    @Override
    public void run() {
        System.out.println("plane....");
    }
}

public class Broom implements Moveable {
    @Override
    public void run() {
        System.out.println("broom.....");
    }
}

// 抽象工厂
public abstract class VehicleFactory {
    abstract Moveable create();
}

// 具体工厂
public class PlaneFactory extends VehicleFactory {
    public Moveable create() {
        return new Plane();
    }
}
```

```

    }
}

public class BroomFactory extends VehicleFactory {
    public Moveable create() {
        return new Broom();
    }
}

// 测试类
public class Test {
    public static void main(String[] args) {
        VehicleFactory factory = new BroomFactory();
        Moveable m = factory.create();
        m.run();
    }
}

```

抽象工厂模式：与工厂方法模式不同的是，工厂方法模式中的工厂只生产单一的产品，而抽象工厂模式中的工厂生产多个产品

```

/抽象工厂类
public abstract class AbstractFactory {
    public abstract Vehicle createVehicle();
    public abstract Weapon createWeapon();
    public abstract Food createFood();
}

//具体工厂类，其中Food, Vehicle, Weapon是抽象类，
public class DefaultFactory extends AbstractFactory{
    @Override
    public Food createFood() {
        return new Apple();
    }
    @Override
    public Vehicle createVehicle() {
        return new Car();
    }
    @Override
    public Weapon createWeapon() {
        return new AK47();
    }
}

//测试类
public class Test {

```

```
public static void main(String[] args) {  
    AbstractFactory f = new DefaultFactory();  
    Vehicle v = f.createVehicle();  
    v.run();  
    Weapon w = f.createWeapon();  
    w.shoot();  
    Food a = f.createFood();  
    a.printName();  
}  
}
```

代理模式 (proxy)

有两种，静态代理和动态代理。先说静态代理，很多理论性的东西我不讲，我就算讲了，你们也看不懂。什么真实角色，抽象角色，代理角色，委托角色。。。乱七八糟的，我是看不懂。之前学代理模式的时候，去网上翻一下，资料一大堆，打开链接一看，基本上都是给你分析有什么什么角色，理论一大堆，看起来很费劲，不信的话你们可以去看看，我是看不懂他们在说什么。咱不来虚的，直接用生活中的例子说话。（注意：我这里并不是否定理论知识，我只是觉得有时候理论知识晦涩难懂，喜欢挑刺的人一边去，你是来学习知识的，不是来挑刺的）

到了一定的年龄，我们就要结婚，结婚是一件很麻烦的事情，（包括那些被父母催婚的）。有钱的家庭可能会找司仪来主持婚礼，显得热闹，洋气~好了，现在婚庆公司的生意来了，我们只需要给钱，婚庆公司就会帮我们安排一整套结婚的流程。整个流程大概是这样的：家里人催婚->男女双方家庭商定结婚的黄道吉日->找一家靠谱的婚庆公司->在约定的时间举行结婚仪式->结婚完毕

婚庆公司打算怎么安排婚礼的节目，在婚礼完毕以后婚庆公司会做什么，我们一概不知。。。别担心，不是黑中介，我们只要把钱给人家，人家会把事情给我们做好。所以，这里的婚庆公司相当于代理角色，现在明白什么是代理角色了吧。

代码实现请看：

```
//代理接口  
public interface ProxyInterface {  
    //需要代理的是结婚这件事，如果还有其他事情需要代理，比如吃饭睡觉上厕所，也可以写  
    void marry();  
    //代理吃饭(自己的饭，让别人吃去吧)  
    //void eat();  
    //代理拉屎，自己的屎，让别人拉去吧
```

```
//void shit();  
}
```

文明社会，代理吃饭，代理拉屎什么的我就不写了，有伤社会风化~~~能明白就好

好了，我们看看婚庆公司的代码：

```
public class WeddingCompany implements ProxyInterface {  
  
    private ProxyInterface proxyInterface;  
  
    public WeddingCompany(ProxyInterface proxyInterface) {  
        this.proxyInterface = proxyInterface;  
    }  
  
    @Override  
    public void marry() {  
        System.out.println("我们是婚庆公司的");  
        System.out.println("我们在做结婚前的准备工作");  
        System.out.println("节目彩排...");  
        System.out.println("礼物购买...");  
        System.out.println("工作人员分工...");  
        System.out.println("可以开始结婚了");  
        proxyInterface.marry();  
        System.out.println("结婚完毕，我们需要做后续处理，你们可以回家了，其余的事情我们公司来做");  
    }  
  
}
```

看到没有，婚庆公司需要做的事情很多，我们再看看结婚家庭的代码：

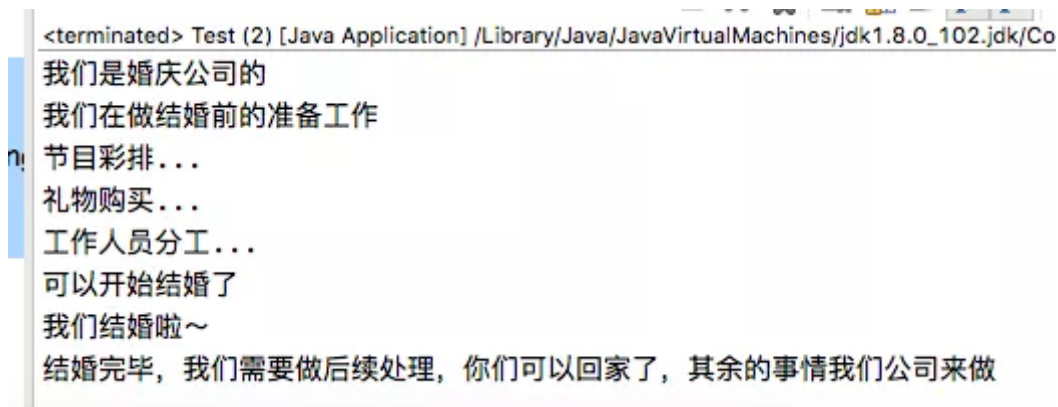
```
public class NormalHome implements ProxyInterface{  
  
    @Override  
    public void marry() {  
        System.out.println("我们结婚啦~");  
    }  
  
}
```

这个已经很明显了，结婚家庭只需要结婚，而婚庆公司要包揽一切，前前后后的事情都是婚庆公司来做，听说现在婚庆公司很赚钱的，这就是原因，干的活多，能不赚钱吗？

来看看测试类代码：

```
public class Test {  
    public static void main(String[] args) {  
        ProxyInterface proxyInterface = new WeddingCompany(new NormalHome());  
        proxyInterface.marry();  
    }  
}
```

运行结果如下：



```
<terminated> Test (2) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_102.jdk/Contents/Home/bin/java -Djava.class.path=.  
我们是婚庆公司的  
我们在做结婚前的准备工作  
节目彩排...  
礼物购买...  
工作人员分工...  
可以开始结婚了  
我们结婚啦~  
结婚完毕，我们需要做后续处理，你们可以回家了，其余的事情我们公司来做
```

在我们预料中，结果正确，这就是静态代理，动态代理我就不想说了，跟java反射有关系，网上资料很多，我以后有时间再更新吧。

PS：如果觉得我的分享不错，欢迎大家随手点“好看”、转发。

(完)

看完视频，
若3个月内工资没有翻翻，
立马赔给你**1000元**！

全行业最新最全视频首次曝光！

看完直接内推到百度、腾讯、阿里等一线互联网公司！

原价：~~¥6280.00~~

本公众号粉丝

免费！ 免费！ 免费！

内容包括：

- 1、Web前端全套
- 2、Java全套
- 3、Python全套
- 4、大数据全套....

(部分截图如下)

总价值超6万！

01.html和css

02.京东项目

03.Javascript 基础

04.webapi

05.javascript高级

06.jquery

07.php基础

14.nodejs

15.vue.js

16.reactjs

17.项目实战(4大项目)

18.赠送-ES6零基础-彩票项目

19.赠送-Angular

20.赠送-Bootstrap

完整的开发工具

01 EditPlus 3.51中文版.zip










02 windows下vim编辑器.rar


03 Adobe Dreamweaver CS5 简体中文版


04 sublime中文版+使用方法.zip


05 firebug汉化版.rar


06 HttpWatch Pro V7.0.23 汉化版.zip


 08.Ajax	 21.前端开发常用工具	 07 Aptana-Studio.rar
 09.阿里百秀项目实战	 【赠】100IT 名企前端面试真题.docx	 08 FastStone Capture 7.4 简体中文绿色免
 10.html5和css3	 【赠】不限速工具使用说明.txt	 09 Dreamweaver CS6绿色版.rar


☐  01-JavaSE知识(学习27天)


☐  02-Web前端知识(学习5天)


☐  03-MySql数据库与JDBC(学习3天)


☐  04-JavaWeb知识(学习12天)


☐  05-JavaWeb企业实战项目(学习6天)


☐  06-Hibernate框架(学习4天)


☐  07-Struts2框架(学习4天)


☐  08-Spring框架(学习4天)


☐  09-SSH企业案例_CRM-客户管理系统(6天)


☐  10-Oracle数据库(学习4天)


☐  11-Maven(学习2天)

☐  12-SSH企业案例2_ERP_项目整合(学习15天)

☐  13-Mybatis(学习2天)

☐  14-SpringMVC(学习2天)

☐  15-SSM企业案例-客户管理系统(学习1天)

☐  16-SSM分布式案例-电商商城(学习13天)

-

-

-

-

-

-

-

-

-

-

-

-

-

-

-

-

2018-07-15 22:34

2018-07-15 22:34

2018-07-15 22:34

2018-07-15 22:34

2018-07-15 22:34

2018-07-15 22:34

2018-07-15 22:34

2018-07-15 22:34

2018-07-15 22:34

2018-07-15 22:34

2018-07-15 22:34

2018-07-15 22:34

2018-07-15 22:34

2018-07-15 22:34

2018-07-15 22:34

2018-07-15 22:34

下图是我的私人微信
微信添加时
记得备注暗号:1
如果没有备注数字1，一律不通过好友！



已有1139人领取

每天前十位，
下面1000本程序员必读书（仅展示部分）

任选一本**免费送!**

（每日限前20名）



Flask Web开发：基于Python
[美]米格尔·格...
安道 译



深度学习入门：基于Python的理论与应用
斋藤康毅
陆宇杰 译



第一行代码——Android
(第2版)
周志明 著



Angular权威教程
Ari Lerner, Feli...
Nice Angular社... 译



用数学的语言看世界
大栗博司
尤斌斌 译



前端架构设计
[美]Micah God...
潘泰焱 张鹏 ...



图解Java多线程设计模式
结城浩
侯振龙 杨文... 译



程序员的算法趣题
增井敏克
绝云 译



算法图解
Aditya Bhargav...
袁国忠 译



Web开发权威指南
Chris Aquino, T...
奇舞团 译



React设计模式与最佳实践
米凯莱·贝尔托...
林昱 译



JavaScript测试驱动开发
Venkat Subram...
王姓委 译



同构JavaScript应用开发
Jason Strimpel ...
张俊达 译



React Native应用开发实例解析
Alexander McL...
林昱 译



学习JavaScript数据结构与算法
[巴西] Loiane G...
邓钢 孙晓楠 译



Head First
JavaScript程序
Eric T. Freeman...
袁国忠 译



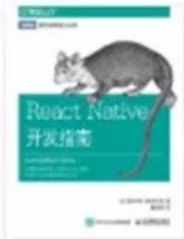
JavaScript编程
精粹
Ved Antani
门佳 译



精通
JavaScript (第2
John Resig Jo...
门佳 李伟 译



你不知道的
JavaScript (中
Kyle Simpson
单业 姜南 译



React Native开
发指南
[美]Bonnie Eise...
黄为伟 译

Java团长

专注于Java干货分享



扫描上方二维码获取更多Java干货