源码见GitHub：<https://github.com/DuanJiaNing/DesignPattern.git>

# 1 单例模式（Singleton）

单例模式**必须保证类只有一个实例存在**，使用者无法自由构建对象，避免产生过多对象消耗资源。

单例：构造器私有，通过静态方法获取唯一实例。

1. **饿汉单例** – 线程不安全 – 占用资源

声明唯一实例时立即初始化

1. **懒汉单例** – 线程不安全 – 延迟初始化

使用时才初始化

1. **静态内部类** – 线程安全 – 延迟初始化

将唯一实例定义成静态内部类的最终成员变量

1. **DCL单例** – 线程安全 – 延迟加载

使用双重检查和锁保证线程安全

1. **枚举单例** – 线程安全

反序列化时对象也不会重建

反序列化操作的钩子函数：readResolve()

## 单例失效

1. 通过反射（setAccessible）可以调用私有构造器创建第二实例，**如果要抵御这种攻击，可以修改构造器，让它在被要求创建第二实例的时候抛出异常。**
2. 除枚举单例外，其它方式在反序列化时都会重新创建对象（单例失效）

序列化重建对象的修复方法：

**声明所有实例都是瞬时（transient）的，并提供一个readResolve方法，以返回唯一实例。**

1. 多进程情况下也会失效
2. DCL失效问题：new不是一个原子操作，指令重排序的问题。

<http://www.jianshu.com/p/83f7958b0944>

instance = new Singleton(); //问题的根源产生

JVM中，上面的代码对应三个操作：

memory = allocate(); //#1为对象分配内存空间

init(memory); //#2初始化成员

instance = memory; //#3设置instance,将其指向刚分配的内存空间。

重排序后可能出现的情况：

memory = allocate(); //#1为对象分配内存空间

instance = memory; //#2设置instance,将其指向刚分配的内存空间。

init(memory); //#3初始化成员

实例并未初始化，但引用却赋给了对象。修复：**将唯一实例使用volatile关键字修饰。**

# 2 装饰模式（Decorator）

装饰模式和适配器模式都属于包装模式（Wrapper）

<https://www.cnblogs.com/java-my-life/archive/2012/04/20/2455726.html>

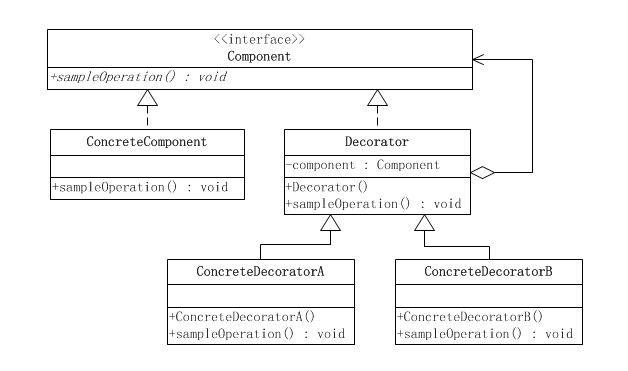
对客户透明的方式动态的**给一个对象附加上更多的责任**。装饰模式的典型应用是javaI/O流。

如：BufferReader:

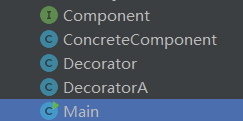
String str = "abcdefghijklmnopqrstuvwxyz";  
Reader reader = new BufferedReader(new StringReader(str));  
reader.readLine();

BufferReader和StringReader都实现了Reader接口，这里BufferReader为装饰者，StringReader为被装饰者，Reader为两者都实现的接口。

BufferReader为StringReader所实现的Reader接口的各个方法进行了**包装**，对使用者而言，使用者看似调用的Reader接口是BufferReader实现的，然而实际调用的是StringReader实现的，BufferReader**间接**调用StringReader的方法，并**附加了更多的责任**（BufferReader包装了更多的功能）。



1. 抽象构件(**Component**)角色：给出一个抽象接口，以规范准备接收附加责任的对象。
2. 具体构件(**ConcreteComponent**)角色：定义一个将要接收附加责任的类。
3. 装饰(**Decorator**)角色：持有一个构件(Component)对象的实例，并定义一个与抽象构件接口一致的接口。
4. 具体装饰(**ConcreteDecorator**)角色：负责给构件对象“贴上”附加的责任。



// 使用装饰  
 Component c = new DecoratorA(new ConcreteComponent());  
 // 不使用装饰  
 Component cc = new ConcreteComponent();  
 // 违反客户端透明性要求  
 //ConcreteComponent ccc = new ConcreteComponent();  
 //使用装饰及更多额外功能  
 Decorator d = new DecoratorA(new ConcreteComponent());  
 //使用装饰及更多额外功能,“半透明”的装饰模式  
 DecoratorA dd = new DecoratorA(new ConcreteComponent());  
 d.sameOperation();  
 System.*out*.println();  
 dd.sameOperation();  
 dd.a(); // 半透明  
 System.*out*.println();  
 cc.sameOperation();  
 System.*out*.println();  
 c.sameOperation();

**控制台输出：**

实际功能

抽象装饰

具体装饰

实际功能

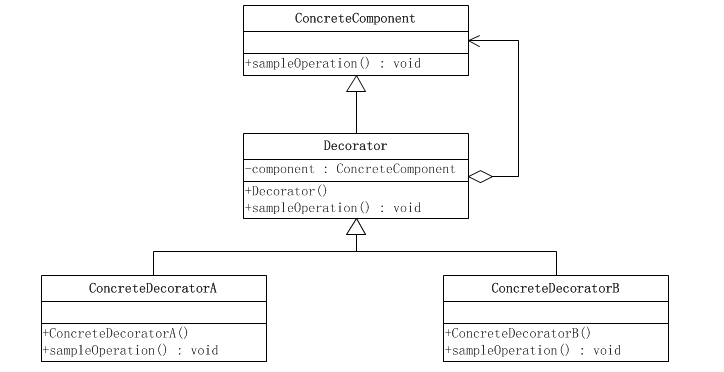
实际功能

抽象装饰

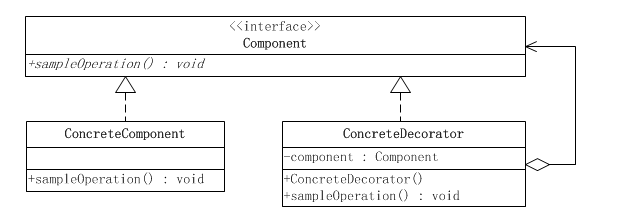
具体装饰

## 装饰模式的简化

省去抽象构件



省去抽象装饰



## 透明性的要求

**装饰模式对客户端的透明性要求程序不要声明一个ConcreteComponent类型的变量，而应当声明一个Component类型的变量。**

## 半透明的装饰模式（半装饰、半适配器模式。）

**装饰模式的用意是在不改变接口的前提下，增强所考虑的类的性能。在增强性能的时候，往往需要建立新的公开的方法。**

大多数的装饰模式的实现都是“半透明”的，而不是完全透明的。换言之，允许装饰模式改变接口，**增加新的方法**。这意味着客户端**可以声明ConcreteDecorator类型的变量，从而可以调用ConcreteDecorator类中才有的方法。**

## 装饰模式的优点

（1）装饰模式与继承关系的目的都是要复用和扩展对象的功能，但是装饰模式可以提供比继承更多的灵活性。装饰模式允许系统动态决定“贴上”一个需要的“装饰”，或者除掉一个不需要的“装饰”。继承关系则不同，继承关系是静态的，它在系统运行前就决定了。

（2）通过使用不同的具体装饰类以及这些装饰类的排列组合，设计师可以创造出很多不同行为的组合。

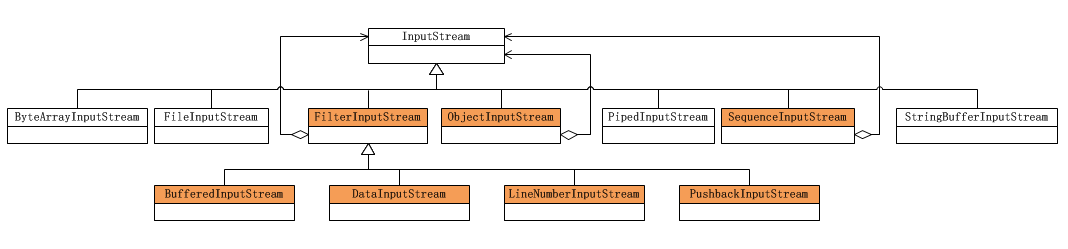
## 装饰模式的缺点

由于使用装饰模式，可以比使用继承关系需要较少数目的类。使用较少的类，当然使设计比较易于进行。但是，在另一方面，使用装饰模式会产生比使用继承关系更多的对象。更多的对象会使得查错变得困难，特别是这些对象看上去都很相像。

## 在JAVA I/O库中的应用

1. 为什么使用装饰模式

由于Java I/O库需要**很多性能的各种组合**，如果这些性能都是用继承的方法实现的，那么每一种组合都需要一个类，这样就会造成大量功能重复的类出现。而如果采用装饰模式，那么类的数目就会大大减少，**功能的重复**也可以减至最少。因此装饰模式是Java I/O库的基本模式。



根据上图可以看出：

**抽象构件(**Component)角色：由InputStream扮演。这是一个抽象类，为各种子类型提供统一的接口。

**具体构件**(ConcreteComponent)角色：由ByteArrayInputStream、FileInputStream、PipedInputStream、StringBufferInputStream等类扮演。它们继承了InputStream。

**抽象装饰**(Decorator)角色：由FilterInputStream，ObjectInputStream和SequenceInputStream扮演。它们同样继承了InputStream。

**具体装饰**(ConcreteDecorator)角色：由几个类扮演，如FilterInputStream的具体装饰有：BufferedInputStream、DataInputStream以及两个不常用到的类LineNumberInputStream、PushbackInputStream。

# 3 外观模式（Facade）

外观模式是为了解决类与类之间的依赖关系，像spring一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个Facade类中，间接调用，降低了类之间的耦合度。

通过外观的包装，使**应用程序只能看到外观现象，而不会看到具体的细节对象，这样会降低程序的复杂度，提高程序的可维护性。**

用户只需直接与外观角色交互，用户与子系统之间的复杂关系由外观角色充当，降低耦合度。

如jsp 9大内置对象之：**session、application、config**

1. session：org.apache.catalina.session.**StandardSessionFacade**

实现HttpSession，同时内部组合有正真的HttpSession。

2. application：org.apache.catalina.core.**ApplicationContextFacade**

实现ServletContext，内部组合ApplicationContext（实现ServletContext），

3. config：org.apache.catalina.core.**StandardWrapperFacade**

实现ServletConfig，内部组合StandardWrapper（实现ServletConfig）。

# 4 代理模式（Proxy）

<https://www.cnblogs.com/cenyu/p/6289209.html>

提供了对目标对象另外的访问方式：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，即扩展目标对象的功能。

如：艺人-**经纪人**-相关人士，房源-**中介**-购房者



代理模式的关键点是:代理对象与目标对象.代理对象是对目标对象的扩展,并会调用目标对象。

## 静态代理

静态代理在使用时,需要定义接口或者父类,被代理对象与代理对象一起实现相同的接口或者是继承相同父类.

**优点**：可以做到在不修改目标对象的功能前提下,对目标功能扩展.

**缺点**：如果有很多的代理类，而代理对象需要与目标对象实现一样的接口,那么一旦接口增加方法,目标对象与代理对象都要维护.

## 动态代理

**代理对象不需要实现接口**，代理对象的生成,是利用JDK的API,动态的在内存中构建**目标对象**(需要我们指定创建代理对象/目标对象**实现的接口的类型**)

动态代理也叫做:JDK代理,接口代理。

JDK提供的API：java.lang.reflect.Proxy

**public static Object newProxyInstance(ClassLoader loader,**

**Class<?>[] interfaces,**

**InvocationHandler h)**

创建代理只需要使用newProxyInstance方法。

final Artist artist = new ArtistTaylor(); // 要代理的目标对象  
//代理对象  
Artist proxy = (Artist) Proxy.*newProxyInstance*(artist.getClass().getClassLoader(),  
 artist.getClass().getInterfaces(),  
 (proxy1, method, args1) -> {  
 *p*.accept("before");  
 Object o = method.invoke(artist, args1); // 执行接口的方法  
 *p*.accept("after");  
 return o;  
 });  
proxy.meetFan();

输出：

before

taylor will going to meet her fans

after

## Cglib代理

上面的静态代理和动态代理模式**都要求目标对象是实现一个接口的目标对象**,**但是有时候目标对象只是一个单独的对象,并没有实现任何的接口,这个时候就可以使用以目标对象子类的方式实现代理,这种方法就叫做:Cglib代理**

* 也叫作子类代理,它是在内存中构建一个子类对象从而实现对目标对象功能的扩展.

Cglib是一个强大的高性能的代码生成包,它可以在运行期扩展java类与实现java接口.它广泛的被许多AOP的框架使用,例如Spring AOP和synaop,为他们提供方法的interception(拦截)

Cglib包的底层是通过使用一个小而快的字节码处理框架ASM来转换字节码并生成新的类.不鼓励直接使用ASM,因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉.

使用：

1 引用Cglib的jar包（spring-core-xxx.jar已经默认包含）

2 **代理类不能为final 的**

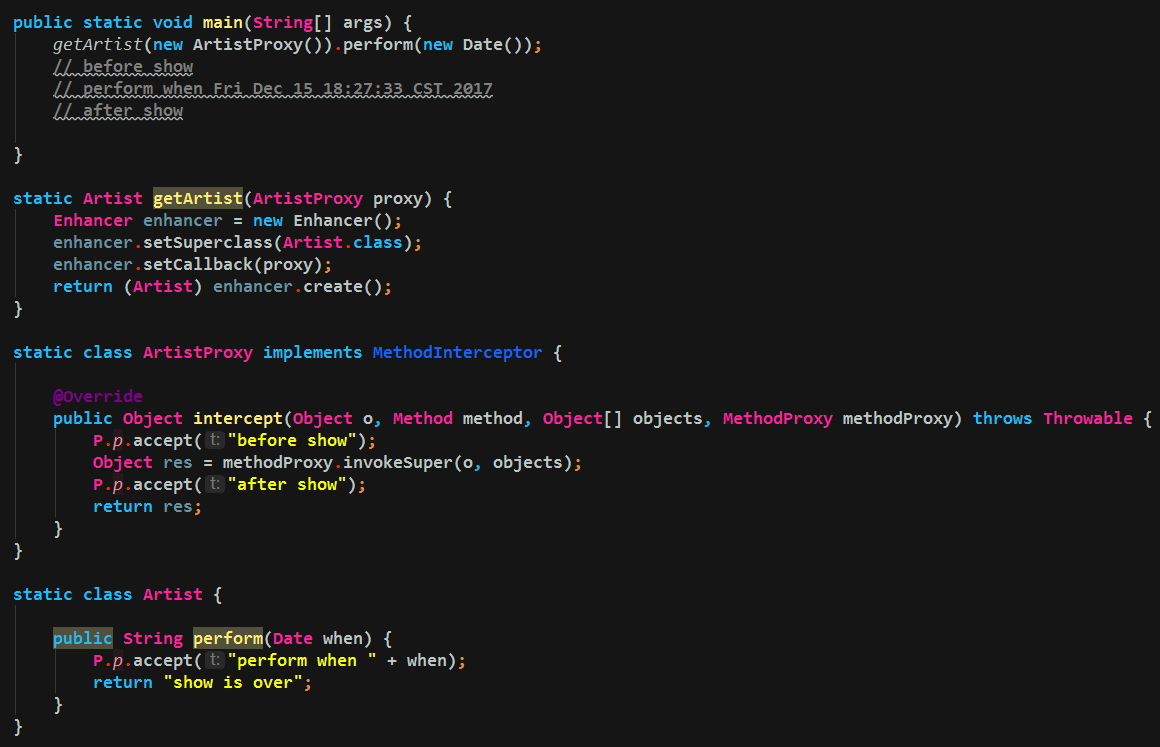
3 目标对象的方法如果为final/static（子类无法覆写，private方法也是final的）,那么就不会被拦截,即不会执行目标对象额外的业务方法.

在Spring的AOP编程中:

如果加入容器的目标**对象有实现接口,用JDK代理**

如果目标**对象没有实现接口,用Cglib代理**

cglib其实通过修改字节码，给没有实现接口的目标类（需要代理的类）生成了一个继承它的子类。



# 5 享元模式（Flyweight）- 结构性模式

复用内存中已存在的对象，降低系统创建对象实例的性能消耗。

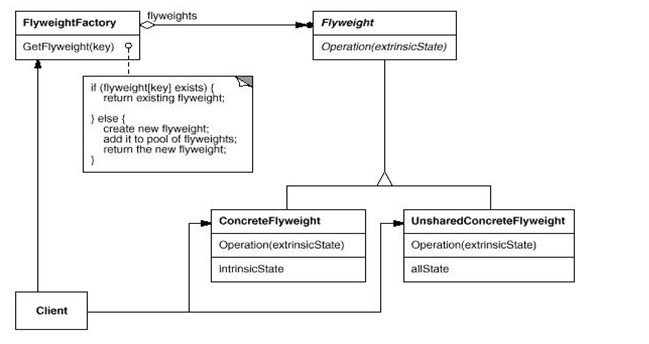
享元模式以共享的方式高效的支持大量细粒度对象的重用。

享元对象能做到共享的关键是区分了内部状态和外部状态。

1 **内部状态**：可以共享，不会随环境改变而改变。

2 **外部状态**：不可以共享，会随环境改变而改变。

有时我们可能需要创建很多的实例，通常来说，这些实例的差别不大，这个时候，我们可以考虑复用其中创建的某个（几个）实例，而不用去new这么多**相似的对象**，来完成这样的工作。



**FlyWeight抽象享元类**：通常是一个接口或抽象类，声明公共方法，这些方法可以向外界提供对象的内部状态，设置外部状态。

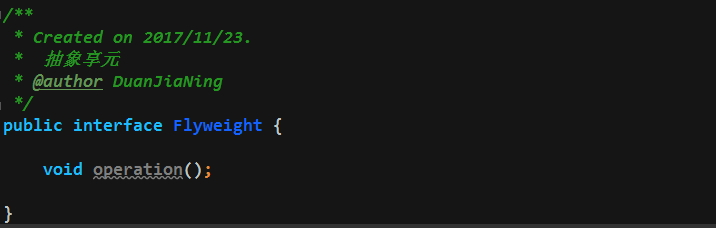
**UnsharedConcreteFlyWeight非共享享元类**：不能被共享的子类可以设计为非共享享元类。

**ConcreteFlyWeight具体享元类**：为内部状态提供成员变量进行存储。

**FlyWeightFactory享元工厂类**：创建并管理享元对象，享元池一般设计为键值对。

享元模式中，最关键的享元工厂。它将维护已创建的享元实例，并通过实例标记（一般用内部状态）去索引对应的实例。当目标对象未创建时，享元工厂负责创建实例并将其加入标记-对象映射。当目标对象已创建时，享元工厂直接返回已有实例，实现对象的复用。

享元模式中对象的复用完全依靠享元工厂。





## 优点

1 享元模式的外部状态相对独立，使得**对象可以在不同的环境中被复用**（共享对象可以适应不同的外部环境）

2 享元模式可共享相同或相似的细粒度对象，从而**减少了内存消耗，同时降低了对象创建与垃圾回收的开销**

## 缺点

1 外部状态由客户端保存，共享对象读取外部状态的开销可能比较大

2 享元模式要求**将内部状态与外部状态分离**，这使得程序的逻辑复杂化，同时也增加了状态维护成本

# 6 观察者模式（Observer）

<http://www.jianshu.com/p/1025f644f100>

观察者模式面向的需求是：A 对象（观察者）对 B 对象（被观察者）的某种变化高度敏感，需要在 B 变化的一瞬间做出反应。

举个例子，新闻里喜闻乐见的警察抓小偷，警察需要在小偷伸手作案的时候实施抓捕。在这个例子里，警察是观察者，小偷是被观察者，警察需要时刻盯着小偷的一举一动，才能保证不会漏过任何瞬间。

程序的观察者模式和这种真正的『观察』略有不同，观察者不需要时刻盯着被观察者（例如 A 不需要每过 2ms 就检查一次 B 的状态），而是采用注册(Register)或者称为订阅(Subscribe)的方式，告诉**被观察者**：我需要你的某某状态，你要在它变化的时候通知我。



**Observable**

即**被观察者**，也可以被叫做主题（Subject）是被观察的对象。通常有注册方法（register），取消注册方法(remove)和通知方法(notify)。

**Observer**

即**观察者**，可以接收到主题的更新。当对某个主题感兴趣的时候需要**注册自己**，在不需要接收更新时进行注销操作。

**观察者**调用**被观察者**的注册方法注册自己，观察者调用被观察者的取消注册方法取消注册。

## RxJava 的观察者模式

<http://gank.io/post/560e15be2dca930e00da1083>

RxJava 有四个基本概念：**Observable** (可观察者，即被观察者)、**Observer** (观察者)、**subscribe** 、(订阅)、**事件。**

Observable 和 Observer 通过 subscribe() 方法实现订阅关系，从而 Observable 可以在需要的时候发出事件来通知 Observer。

与传统观察者模式不同， RxJava 的事件回调方法除了普通事件 onNext() （相当于 onClick() / onEvent()）之外，还定义了两个特殊的事件：onCompleted() 和 onError()。

**onCompleted():** 事件队列完结。RxJava 不仅把每个事件单独处理，还会把它们看做一个队列。RxJava 规定，当不会再有新的 onNext() 发出时，需要触发 onCompleted() 方法作为标志。

**onError():** 事件队列异常。在事件处理过程中出异常时，onError() 会被触发，同时队列自动终止，不允许再有事件发出。

在一个正确运行的事件序列中, onCompleted() 和 onError() 有且只有一个，并且是事件序列中的最后一个。需要注意的是，onCompleted() 和 onError() 二者也是互斥的，即在队列中调用了其中一个，就不应该再调用另一个。



# 7 工厂模式（Factory）

<http://blog.csdn.net/jason0539/article/details/23020989>

在面向对象编程中, 通常使用new操作符来构造对象实例。但是在一些情况下, new操作符直接生成对象会带来一些问题。举例来说：**许多类型对象的创造需要一系列的步骤**:

你可能需要计算或取得对象的初始设置;

选择生成哪个子对象实例;

或在生成你需要的对象之前必须先生成一些辅助功能的对象。

在这些情况,**新对象的建立就是一个 “过程”，不仅是一个操作**，像一部大机器中的一个齿轮传动。

通过工厂模式**可以轻松方便的构造对象实例，而不必关心构造对象实例的细节和复杂过程**。

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。**

## 分类及区别

工厂模式可以分为三类：

1. 简单工厂模式（Simple Factory）

***一个具体工厂只能生产某一类具体的产品***。

BMWm20工厂 –> BMWm20产品

1. 工厂方法模式（Factory Method）

**一个**抽象产品类，可以派生出**多个**具体产品类。

***一个抽象工厂类，可以派生出多个具体工厂类。***

***每个具体工厂类只能创建一个具体产品类的实例*。**

BMWm系产品 **——1个产品** BMWm系产品 -> BMWm10产品、BMW20产品、…

BMWm系工厂 -> BMWm20工厂、BWm10工厂…

BMWm20工厂 -> BMWm20产品 **——1类产品**

1. 抽象工厂模式（Abstract Factory）

**多个**抽象产品类，每个抽象产品类可以派生出**多个**具体产品类。

***一个抽象工厂类，可以派生出多个具体工厂类。***

***每个具体工厂类可以创建多个具体产品类的实例。***

BMWm系产品、BMWs系产品 **——多个产品** BMWm系产品 -> BMWm10产品、BMWm20产品、…

BMW工厂 -> BMWm系工厂、BMWs系工厂…

BMWm系工厂 -> BMWm10产品，BMWm20产品 **——多类产品**

这三种模式从上到下逐步抽象，并且更具一般（泛化）性。

区别：

工厂方法模式只有一个抽象产品类，而抽象工厂模式有多个。

工厂方法模式的具体工厂类只能创建一个具体产品类的实例，而抽象工厂模式可以创建多个。

## 工厂模式的演进过程

1. **还没有工厂时代**：假如还没有工业革命，如果一个客户要一款宝马车,一般的做法是客户去创建（**new**）一款宝马车，然后拿来用。
2. **简单工厂模式**：后来出现工业革命。用户不用去创建宝马车。因为客户有一个工厂来帮他创建宝马.想要什么车，这个工厂就可以建。比如想要320i系列车。工厂就创建这个系列的车。即工厂可以创建产品。
3. **工厂方法模式**时代：为了满足客户，宝马车系列越来越多，如320i，523i,30li等系列，一个工厂无法创建所有的宝马系列。于是由单独分出来多个具体的工厂。每个具体工厂创建一种系列。即**具体工厂类只能创建一个具体产品**。但是宝马工厂还是个抽象。你需要指定某个具体的工厂才能生产车出来。
4. **抽象工厂模式**时代：随着客户的要求越来越高，宝马车必须配置空调。于是这个工厂开始生产宝马车和需要的空调。最终是客户只要对宝马的销售员说：我要**523i 空调车**，销售员就直接给他523i空调车了。

几种不同模式对于产品的抽象是通用的：

抽象产品：



具体产品：



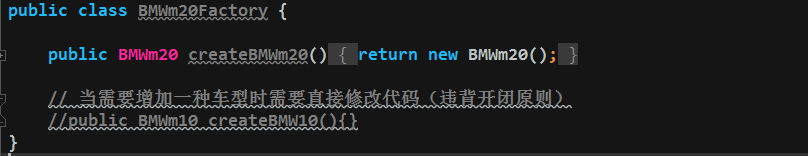
## 简单（静态）工厂模式

又称静态工厂模式，这里的静态指的是创建后无法修改，不具有动态改变的能力，要修改时只能通过修改源码。

OOP开闭原则考虑：

每增加一种新型车，都要在工厂类中增加相应的创建业务逻辑，这显然是**违背开闭原则**的。

1. **抽象产品**
2. **具体产品**
3. **具体工厂**：简单工厂模式核心

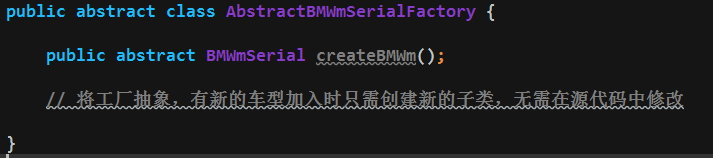


## 工厂方法模式

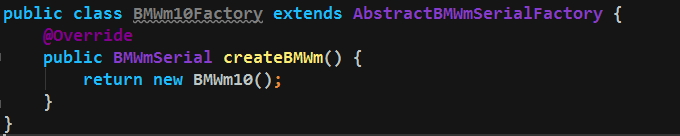
将工厂类抽象,而每新增的车种类型,就增加该车种类型对应工厂类的实现。

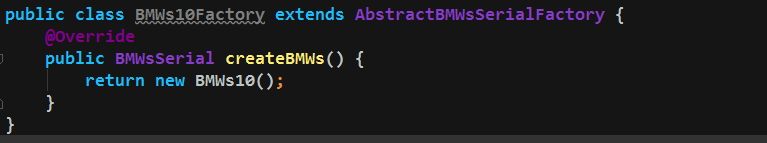
1. **抽象产品**
2. **具体产品**
3. **抽象工厂**：工厂方法模式的核心
4. **具体工厂**

抽象工厂：



具体工厂：





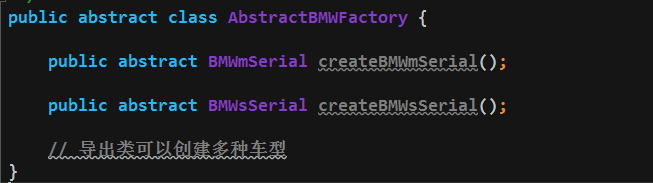
工厂方法模式仿佛已经很完美的对对象的创建进行了包装，使得客户程序中仅仅处理抽象产品角色提供的接口，但使得对象的数量成倍增长。**当产品种类非常多时，会出现大量的与之对应的工厂对象**，这不是我们所希望的。这个问题由抽象工厂模式解决。

## 抽象工厂模式

<http://blog.csdn.net/jason0539/article/details/44976775>

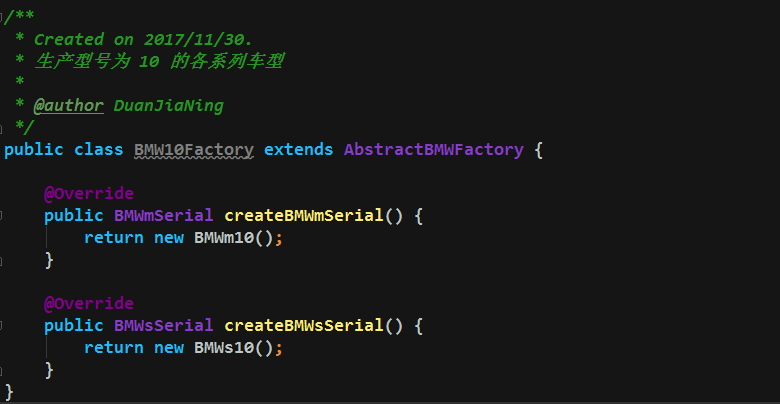
一个具体的工厂类**可以同时创建多个具体产品**

抽象工厂：



具体工厂：



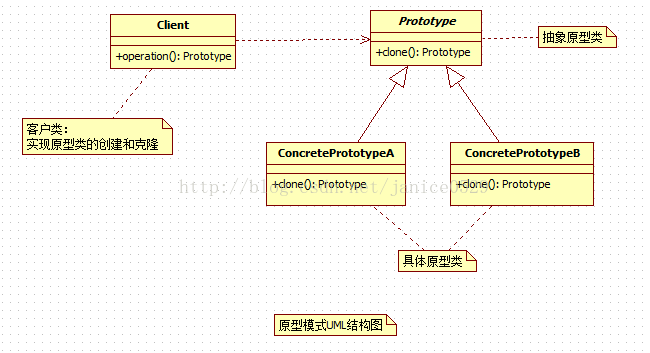


# 原型模式（Prototype）

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**，使用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。简单的说就是对象的拷贝生成新的对象（对象的克隆），原型模式是一种对象创建型模式。

**使用场景：**

创建新的对象可以通过对已有对象进行复制来获得，如果是相似对象，则只需对其成员变量稍作修改。



**Prototype**（抽象原型类）：它是声明克隆方法的接口，是所有具体原型类的公共父类，可以是抽象类也可以是接口，甚至还可以是具体实现类。

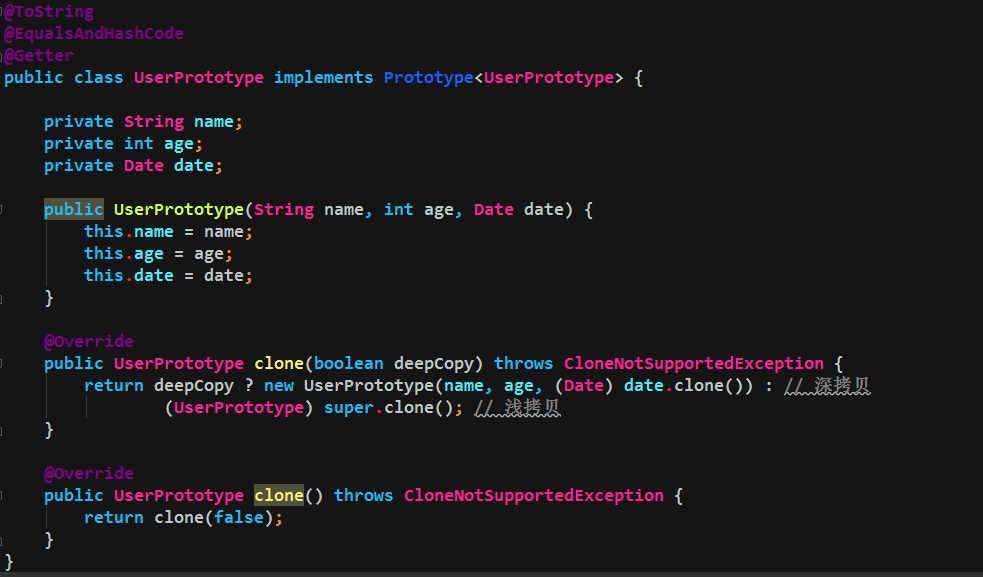
**ConcretePrototype**（具体原型类）：它实现在抽象原型类中声明的克隆方法，在克隆方法中返回自己的一个克隆对象。

**Client**（客户类）：让一个原型对象克隆自身从而创建一个全新的对象。

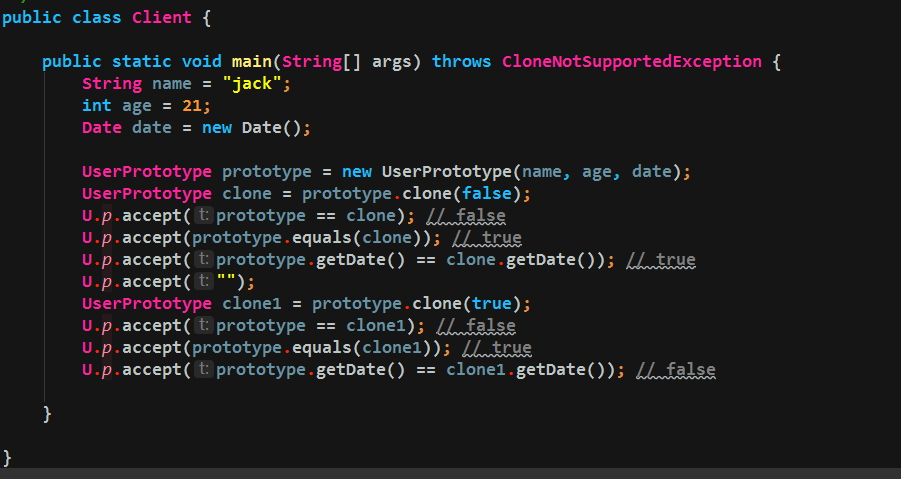
## 抽象原型类



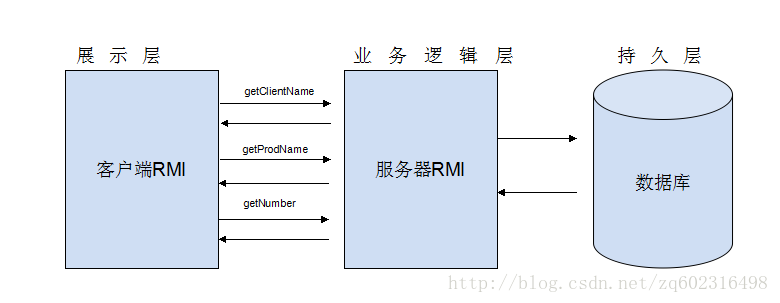
## 具体原型类



## 客户类（测试类）



# Value Object 模式



展示层主要负责数据的展示

业务路基层负责具体的业务逻辑处理

持久层负责数据库等持久化操作。

大型系统中，这些层次很有可能会被分离，部署在不同的服务器上。而在两个层次之间可能通过远程调用 RMI 等方式进行通信。

这种模式的问题：

1. 以面向对象设计的角度来看，这种方式不合理，繁琐且不具有较好的维护性。
2. 一个订单进行3次通信，性能太低。

为了解决这个问题：可以使用 Value Object 模式。**这种模式提倡将一个对象的各个属性进行封装，将封装后的对象在网络中传播，**从而使系统拥有更好的交互模型，并且减少网络通信数据，从而提高系统性能。

