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

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

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），多进程情况下也会失效。

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。

反序列化操作的钩子函数：readResolve()

# 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>

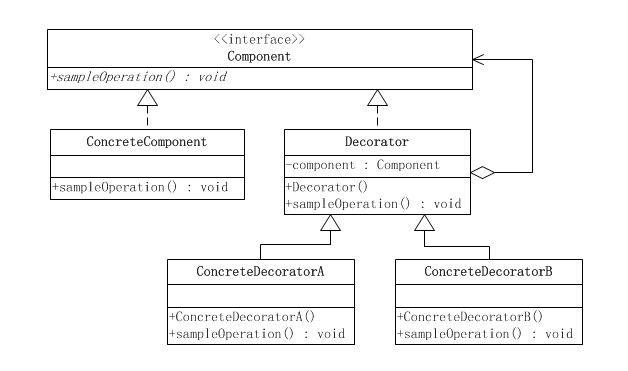
对客户透明的方式动态的**给一个对象附加上更多的责任**。装饰模式的典型应用是javaI/O流。

如：BufferReader:

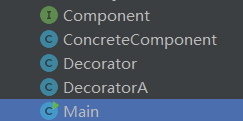
String str = "abcdefghijklmnopqrstuvwxyz";  
BufferedReader reader = new BufferedReader(new StringReader(str));  
reader.readLine();

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

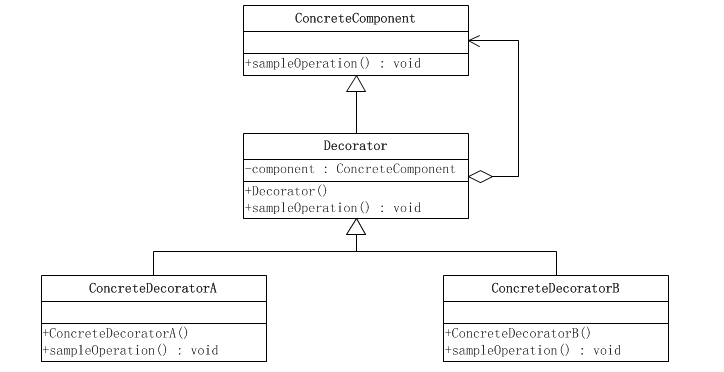
实际功能

抽象装饰

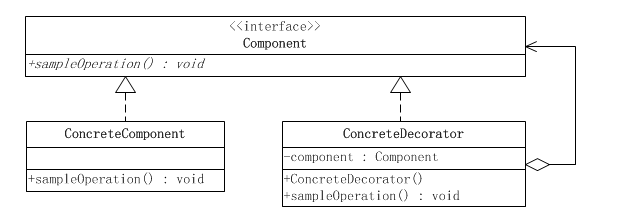
具体装饰

## 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。

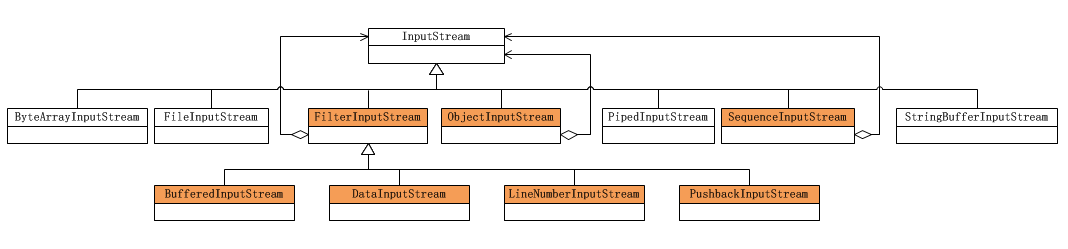
## 装饰模式的缺点

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，这样就会造成大量性能重复的类出现。而如果采用装饰模式，那么类的数目就会大大减少，**性能的重复**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。它们实现了抽象构件角色所规定的接口。

**抽象装饰**(Decorator)角色：由FilterInputStream扮演。它实现了InputStream所规定的接口。

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

# 3 外观模式（Facade）

# 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,增强额外的功能操作,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类实现代理,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块的字节码处理框架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,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

# 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。

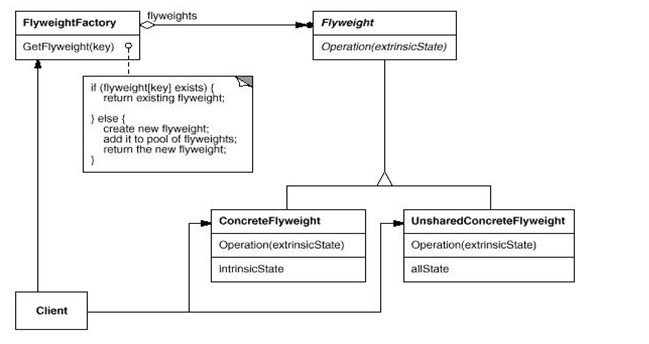
享元模式以共享的方式高效的支持大量细粒度对象的重用。

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