# **设计模式**

目录

[设计模式 1](#_Toc2117)

[第一章 概述 5](#_Toc14645)

[1.1. 设计模式的六大原则 5](#_Toc11354)

[1.1.1. 开闭原则（Open Close Principle） 5](#_Toc28476)

[1.1.2. 里氏代换原则（Liskov Substitution Principle） 5](#_Toc1818)

[1.1.3. 依赖倒转原则（Dependence Inversion Principle） 5](#_Toc15780)

[1.1.4. 接口隔离原则（Interface Segregation Principle） 6](#_Toc139)

[1.1.5. 迪米特法则（最少知道原则）（Demeter Principle） 6](#_Toc8563)

[1.1.6. 合成复用原则（Composite Reuse Principle） 6](#_Toc27213)

[1.2. 泛化、实现、依赖、关联、聚合、组合 6](#_Toc5050)

[1.2.1. 泛化Generalization【extends is-a】 6](#_Toc14197)

[1.2.2. 实现Realization【implements is-a】 7](#_Toc10351)

[1.2.3. 依赖Dependency【use】 7](#_Toc19566)

[1.2.4. 关联Association【has】 8](#_Toc6494)

[1.2.5. 聚合Aggregation【has-a】 8](#_Toc28343)

[1.2.6. 组合Composition【contains-a】 9](#_Toc17761)

[1.2.7. 总结 10](#_Toc23477)

[1.3. 设计模式的分类 11](#_Toc11892)

[第二章 单例模式 12](#_Toc1781)

[2.1. 简介 12](#_Toc5483)

[2.2. 懒汉式单例 12](#_Toc17291)

[2.2.1. 线程不安全的懒汉式单例 12](#_Toc2129)

[2.2.2. 在getInstance方法上加同步 13](#_Toc31100)

[2.2.3. 双重校验锁 13](#_Toc8373)

[2.2.4. 静态内部类 14](#_Toc20300)

[2.2.5. 总结 15](#_Toc22898)

[2.3. 饿汉式单例 15](#_Toc12585)

[2.4. 枚举 15](#_Toc24509)

[2.5. 总结 16](#_Toc10828)

[第三章 工厂模式 16](#_Toc8558)

[3.1. 简介 16](#_Toc27271)

[3.2. 工厂方法模式 16](#_Toc10990)

[3.3. 抽象工厂模式 17](#_Toc24856)

[3.4. Spring中的工厂模式 18](#_Toc14634)

[第四章 建造者模式 19](#_Toc24375)

[第五章 原型模式 20](#_Toc18549)

[第六章 代理模式 21](#_Toc2046)

[6.1. 简介 21](#_Toc28492)

[6.2. 静态代理 22](#_Toc16533)

[6.3. 动态代理 22](#_Toc3152)

[6.3.1. JDK动态代理 22](#_Toc10959)

[6.3.2. CGLIB动态代理 24](#_Toc18674)

[第七章 装饰者模式 27](#_Toc17816)

[7.1. 简介 27](#_Toc16039)

[7.2. 装饰器模式的应用场景 28](#_Toc27107)

[第八章 适配器模式 30](#_Toc15011)

[8.1. 简介 30](#_Toc20474)

[8.2. 类的适配 31](#_Toc15456)

[8.3. 对象的适配 32](#_Toc26718)

[8.4. 接口的适配 32](#_Toc28038)

[8.5. 三种适配的区别 33](#_Toc27801)

[第九章 外观模式 33](#_Toc1518)

[第十章 桥接模式 34](#_Toc28594)

[第十一章 组合模式 34](#_Toc25809)

[第十二章 享元模式 34](#_Toc13086)

[第十三章 策略模式 34](#_Toc12393)

[第十四章 管道模式 34](#_Toc27749)

[第十五章 模板方法 34](#_Toc6021)

[第十六章 观察者模式 34](#_Toc18652)

[16.1. 简介 34](#_Toc27876)

[16.2. 针对的问题 34](#_Toc10261)

[16.3. 角色组成 35](#_Toc21081)

[16.4. 举例说明 36](#_Toc2811)

[16.5. 优点 40](#_Toc6651)

[16.6. 缺点 41](#_Toc25785)

[第十七章 迭代子模式 42](#_Toc30195)

[第十八章 责任链模式 42](#_Toc9173)

[第十九章 命令模式 42](#_Toc5035)

[第二十章 备忘录模式 42](#_Toc15657)

[第二十一章 状态模式 42](#_Toc25950)

[第二十二章 访问者模式 42](#_Toc4000)

[第二十三章 中介者模式 42](#_Toc13671)

[第二十四章 解释器模式 42](#_Toc28385)

[第二十五章 并发型模式 42](#_Toc14521)

[第二十六章 线程池模式 42](#_Toc21582)

[第二十七章 Spring中的设计模式 42](#_Toc16999)

[27.1. 单例模式 42](#_Toc28929)

[27.2. 工厂模式 46](#_Toc2092)

[27.3. 适配器模式 46](#_Toc28849)

[27.4. 代理模式 49](#_Toc10308)

[27.5. 装饰者模式 49](#_Toc31897)

[27.6. 策略模式 50](#_Toc25377)

[27.6.1. 定义 50](#_Toc18147)

[27.6.2. Spring中策略模式的应用 50](#_Toc20418)

[27.7. 观察者模式 52](#_Toc27172)

[27.7.1. ApplicationEvent 52](#_Toc18352)

[27.7.2. ApplicationListener 53](#_Toc1509)

[27.7.3. ApplicationContext 53](#_Toc28652)

[27.7.4. ApplicationEventMulticaster 55](#_Toc2130)

[27.7.5. 举例说明 57](#_Toc11866)

[第二十八章 总结 60](#_Toc22150)

[28.1. 装饰者模式和代理模式区别 60](#_Toc5991)

[28.2. 适配器、装饰者、代理模式的区别 61](#_Toc20491)

[第二十九章 Spring源码学习 61](#_Toc32251)

[29.1. 基础知识 61](#_Toc20358)

[29.2. 源码学习 62](#_Toc24576)

[29.2.1. Bean加载流程 62](#_Toc6576)

# 概述

## 设计模式的六大原则

### 开闭原则（Open Close Principle）

开闭原则就是说对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

### 里氏代换原则（Liskov Substitution Principle）

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

### 依赖倒转原则（Dependence Inversion Principle）

这个是开闭原则的基础，具体内容：针对接口编程，依赖于抽象而不依赖于具体。

按理说，高层组件应该依赖于低层组件，低层组件为高层组件提供一些最基础的服务，但是工厂模式倒置了这一依赖现象，让低层组件反而要依赖于统一的抽象接口。

工厂模式让高层组件（WebCrawler）和低层组件（ATypeClient|BTypeClient|……）都依赖于共同的接口（HttpClient），这倒置了原本的依赖模型，解除了高层组件和低层组件之间的强依赖关系。

### 接口隔离原则（Interface Segregation Principle）

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

### 迪米特法则（最少知道原则）（Demeter Principle）

为什么叫最少知道原则，就是说：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。

### 合成复用原则（Composite Reuse Principle）

原则是尽量使用合成/聚合的方式，而不是使用继承。

## 泛化、实现、依赖、关联、聚合、组合

### 泛化Generalization【extends is-a】

泛化是一种继承关系，如果一个类A的所有属性和操作能被另一个类B所继承，则类B不仅可以包含自己独有的属性，而且可以包含类A的属性和操作。继承是类与类或者类与接口之间最常见的关系。

父类 父类实例=new 子类()；

泛化中子类直接继承了父类的属性和操作所以它的关系最强。

class Animal{}     
class Tiger extends Animal{}     
public class Test     
{     
    public void test()     
     {     
         Animal a=new Tiger();     
     }     
}

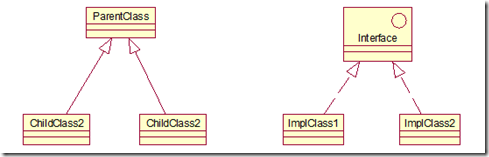
指的是一个类（称为子类、子接口）继承另外的一个类（称为父类、父接口）的功能，并可以增加它自己的新功能的能力，继承是类与类或者接口与接口之间最常见的关系。

UML中用带空心箭头的实线表示，箭头指向一般个体（父类）。

### 实现****Realization【implements is-a】****

实现关系和泛化是同等的强度。

指的是一个class类实现interface接口（可以是多个）的功能；实现是类与接口之间最常见的关系。



UML中用空心箭头和虚线表示，箭头指向定义约定的元素(一般指接口)。

### 依赖Dependency【use】

对于两个相对独立的对象，当一个对象负责构造另一个对象的实例时，或者依赖另一个对象的服务时，这两个对象之间主要体现为依赖关系

  与关联关系不同的是，依赖关系是以参数变量的形式传入到依赖类中。

  依赖是一种弱关联，只要一个类用到另一个类，但是和另一个类的关系不太明显的时候（可以说是“use”了那个类），可以把这种关系看成是依赖。

IMG_256

依赖是单向的。

    依赖关系表现在局部变量，方法的参数，以及对静态方法的调用。

public class Person{     
    /\*\* 拧螺丝 \*/    
    public void screw(Screwdriver screwdriver){     
         screwdriver.screw();     
     }     
}

UML中用带箭头的虚线表示，箭头指向被依赖元素。

### 关联Association【has】

对于两个相对独立的对象，当一个对象的实例与另一个对象的一些特定实例存在固定的对应关系时，这两个对象之间为关联关系。

IMG_256

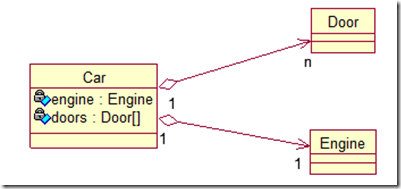
关联关系时使用实例变量来实现的。

public class Company{     
    private Employee employee;     
    public Employee getEmployee(){     
        return employee;     
     }     
    public void setEmployee(Employee employee){     
        this.employee=employee;     
     }     
    //公司运作     
    public void run(){     
         employee.startWorking();     
     }     
}

UML中用实线表示Association关系，箭头指向被依赖元素。

### 聚合Aggregation【has-a】

当对象A被加入到对象B中，成为对象B的组成部分时，对象B和A之间为聚集关系。聚合是关联关系的一种，是较强的关联关系，强调整体与部分之间的关系。



聚合关系时使用实例变量来实现的。

public class Computer{     
    private CPU cpu;     
    public CPU getCPU(){     
        return cpu;     
     }     
    public void setCPU(CPU cpu){     
        this.cpu=cpu;     
     }     
    //开启电脑     
    public void start(){     
        //cpu运作     
         cpu.run();     
     }     
}

UML中用带空心菱形头的实线表示，菱形头指向整体。

### 组合Composition【contains-a】

  组合关系也是聚合关系的一种，是比聚合关系更强的关系。组合关系是不能共享的。例如人有四肢、头等

   表示类之间整体和部分的关系，组合中部分和整体具有统一的生存周期。一旦整体对象不存在，部分对象也将不存在。部分对象和整体对象之间具有共生死的感觉。

IMG_256

组合关系时使用实例变量来实现的。

class Leg{};

class Arm{};

class Person

{

Leg mLeg;

Arm mArm;

}

UML中用带实心菱形头的实线表示Composition关系，菱形头指向整体。

### 总结

 实现关系和泛化是同等的强度。

#### 关联和依赖的区别

（1）关联关系中，体现的是两个类、或者类与接口之间语义级别的一种强依赖关系，比如我和我的朋友；这种关系比依赖更强、不存在依赖关系的偶然性、关系也不是临时性的，一般是长期性的，而且双方的关系一般是平等的。

（2）依赖关系中，可以简单的理解，就是一个类A使用到了另一个类B，而这种使用关系是具有偶然性的、临时性的、非常弱的，但是B类的变化会影响到A。

#### 关联与聚合的区别

   在语法上是难以区分的，在语义上才能更好的区分两者的区别。

    区别：关联关系所涉及的两个对象是处在同一个层次上的。比如人和自行车就是一种关联关系，而不是聚合关系，因为人不是自行车的组成部分。

         聚合关系涉及的两个对象处于不平等的层次上，一个代表整体，一个代表部分。比如：电脑和它的显示器、键盘、主板和内存就是聚集关系。

#### 聚合和组合的区别

1. 聚合和组合都是一种结合关系，只是额外具有整体部分的含义
2. 部件的生命周期不同

     聚合关系中，整体不会拥有部件的生命周期，所以整体删除时，部件不会被删除。再者，多个整体可以共享同一个部件。

     组合关系中，整体拥有部分的生命周期，所以整体删除时，部件一定会跟着删除。而且，多个整体不可以同时间共享一个部件。

1. 聚合关系是“has-a”关系，组合关系是“contain-a”关系

所以它们的关系是：组合>聚合>关联。

综上所述它们之间的耦合度是：泛化=实现>组合>聚合>关联>依赖。

## 设计模式的分类

总体来说设计模式分为三大类：

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

其实还有两类：并发型模式和线程池模式。

**要想正确理解设计模式，首先必须明确它是为了解决什么问题而提出来的。**

# 单例模式

## 简介

单例模式具有以下特点：   
1、单例类只能有一个实例。   
2、单例类必须自己创建自己的唯一实例。   
3、单例类必须给所有其他对象提供这一实例。

单例模式确保某个类只有一个实例，而且自行实例化并向整个系统提供这个实例。在计算机系统中，线程池、缓存、日志对象、对话框、打印机、显卡的驱动程序对象常被设计成单例。这些应用都或多或少具有资源管理器的功能。

单例模式的写法有好几种，这里主要介绍三种：懒汉式单例、饿汉式单例、静态内部类、枚举和双重校验锁。

懒汉式单例模式：在类加载时不初始化。

饿汉式单例模式：在类加载时就完成了初始化，所以类加载比较慢，但获取对象的速度快。

## 懒汉式单例

### 线程不安全的懒汉式单例

//*懒汉式单例类.在第一次调用的时候实例化自己*public class Singleton {  
 private Singleton() {  
 }  
 private static Singleton *singleton*;  
 //静态工厂方法  
 public static Singleton getInstance() {  
 if (*singleton* == null) {  
 *singleton* = new Singleton();  
 }  
 return *singleton*;  
 }  
}

Singleton通过将构造方法限定为private避免了类在外部被实例化，在同一个虚拟机范围内，Singleton的唯一实例只能通过getInstance()方法访问。

（事实上，通过Java反射机制是能够实例化构造方法为private的类的，那基本上会使所有的Java单例实现失效。此问题在此处不做讨论。）

但是以上懒汉式单例的实现没有考虑线程安全问题，它是线程不安全的，并发环境下很可能出现多个Singleton实例，要实现线程安全，有以下三种方式，都是对getInstance这个方法改造，保证了懒汉式单例的线程安全。

特点：延时加载；线程不安全，多线程下不能正常工作。

### 在getInstance方法上加同步

//静态工厂方法  
public static synchronized Singleton getInstance() {  
 if (*singleton* == null) {  
 *singleton* = new Singleton();  
 }  
 return *singleton*;  
}

这种写法在getInstance()方法中加入了synchronized锁。能够在多线程中很好的工作，而且看起来它也具备很好的lazy loading，但是效率很低（因为锁），并且大多数情况下不需要同步。

### 双重校验锁

public static Singleton getInstance() {  
 if (*singleton* == null) {  
 synchronized (Singleton.class) {  
 if (*singleton* == null) {  
 *singleton* = new Singleton();  
 }  
 }  
 }  
 return *singleton*;  
}

### 静态内部类

public class Singleton {

*/\*\*  
 \* 类级的内部类，也就是静态的成员式内部类，该内部类的实例与外部类的实例没有绑定关系，而且只有被调用到时才会装载(装在过程是由jvm保证线程安全)  
 \* ，从而实现了延迟加载  
 \*/*  
 private static class LazyHolder {

*/\*\*  
 \* 静态初始化器，由JVM来保证线程安全  
 \*/*  
 private static final Singleton *INSTANCE* = new Singleton();  
 }  
 private Singleton (){}

*/\*\*  
 这个模式的优势在于：getInstance方法并没有被同步，并且只是执行一个域的访问，因此延迟初始化并没有增加任何访问成本   
 \*/*  
 public static final Singleton getInstance() {  
 return LazyHolder.*INSTANCE*;  
 }  
}

这种方式同样利用了classloder的机制来保证初始化instance时只有一个线程，它跟饿汉式单例不同的是（很细微的差别）：饿汉式单例是只要Singleton类被装载了，那么instance就会被实例化（没有达到lazy loading效果），而这种方式是Singleton类被装载了，instance不一定被初始化。因为LazyHolder类没有被主动使用，只有显示通过调用getInstance方法时，才会显示装载LazyHolder类，从而实例化instance。想象一下，如果实例化instance很消耗资源，我想让他延迟加载，另外一方面，我不希望在Singleton类加载时就实例化，因为我不能确保Singleton类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化instance显然是不合适的。这个时候，这种方式相比饿汉式方法就显得更合理。

### 总结

第2.2.2种：加synchronized锁，在方法调用上加了同步，虽然线程安全了，但是每次都要同步，会影响性能，毕竟99%的情况下是不需要同步的，

第2.2.3种：双重校验锁，在getInstance中做了两次null检查，确保了只有第一次调用单例的时候才会做同步，这样也是线程安全的，同时避免了每次都同步的性能损耗。不过不推荐这种方式。

第2.2.4种：静态内部类，利用了classloader的机制来保证初始化instance时只有一个线程，所以也是线程安全的，同时没有性能损耗，所以一般我倾向于使用这一种。

## 饿汉式单例

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

这种方式基于classloder机制避免了多线程的同步问题，不过，instance在类装载时就实例化，这时候初始化instance显然没有达到lazy loading的效果。

以下为饿汉式单例的变种，和上面的一样，都是在类装载时实例化。

饿汉式单例天生就是线程安全的。

特点：可以通过反射机制攻击；线程安全（多个类加载器除外）。

private static Singleton *instance* = null;  
static {  
 *instance* = new Singleton();  
}

## 枚举

public enum Singleton implements Serializable{  
 *INSTANCE*,  
 *INSTANCE2*}

Singleton singleton=Singleton.*INSTANCE*;

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。

枚举实现线程安全的单例模式特点：

JVM会保证enum不能被反射并且构造器方法只执行一次。

## 总结

饿汉式在类创建的同时就实例化一个静态对象出来，不管之后会不会使用这个单例，都会占据一定的内存，但是相应的，在第一次调用时速度也会更快，因为其资源已经初始化完成，

而懒汉式顾名思义，会延迟加载，在第一次使用该单例的时候才会实例化对象出来，第一次调用时要做初始化，如果要做的工作比较多，性能上会有些延迟，之后就和饿汉式一样了。

什么是线程安全？

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。

# 工厂模式

## 简介

****工厂模式：父类定义了创建对象的接口，但是由子类来具体实现，工厂方法让类把实例化的动作推迟到了子类当中。****

也就是说，父类知道什么时候该去创建这个对象，也知道拿到这个对象之后应该对这个对象做什么事情，但是不知道如何去创建这个对象，对象的创建由子类来完成。

## 工厂方法模式

****工厂模式无法解决产品族和产品等级结构的问题。****

public interface IFoodFactory {  
 IFood produceFood();  
}

public class NoodleFactory implements IFoodFactory {  
 @Override  
 public IFood produceFood() {  
 System.*out*.println("---面条厂---");  
 return new Noodle();  
 }  
}

## 抽象工厂模式

抽象工厂模式中，一个工厂生产多个产品，它们是一个产品族，不同的产品族的产品派生于不同的抽象产品（或产品接口）

****工厂模式中，一个工厂生产一个产品，所有产品派生于同一个抽象产品（或产品接口）；而抽象工厂模式中，一个工厂生产多个产品，它们是一个产品族，不同的产品族的产品派生于不同的抽象产品（或产品接口）。****

public interface IDiningRoomFactory {  
 IFood produceFood();  
 IDrink produceDrink();  
}

public class DiningRoomFactoryA implements IDiningRoomFactory {  
 @Override  
 public IFood produceFood() {  
 System.*out*.println("---食堂A---");  
 return new Noodle();  
 }  
  
 @Override  
 public IDrink produceDrink() {  
 System.*out*.println("---食堂A---");  
 return new Water();  
 }  
}

## Spring中的工厂模式

// AbstractFactoryBean.java  
// 继承了FactoryBean，工厂Bean的主要作用是为了实现getObject()返回Bean实例  
public abstract class AbstractFactoryBean<T> implements FactoryBean<T>, BeanClassLoaderAware, BeanFactoryAware, InitializingBean, DisposableBean {  
  
 // 定义了获取对象的前置判断工作，创建对象的工作则交给了一个抽象方法  
// 这里判断了Bean是不是单例并且是否已经被加载过了（未初始化但加载过了，这个问题涉及到Spring处理循环依赖，以后会讨论到）  
 public final T getObject() throws Exception {  
 return this.isSingleton()?(this.initialized?this.singletonInstance:this.getEarlySingletonInstance()):this.createInstance();  
 }  
 // 由子类负责具体创建对象  
 protected abstract T createInstance() throws Exception;  
}

之所以这么写是因为这种写法带来了两个好处:

（1） 保证了创建Bean的方式的多样性   
 Bean工厂有很多种，它们负责创建各种各样不同的Bean，比如Map类型的Bean，List类型的Bean，Web服务Bean，子类们不需要关心单例或非单例情况下是否需要额外操作，只需要关心如何创建Bean，并且创建出来的Bean是多种多样的。

（2） 严格规定了Bean创建前后的其它动作   
 虽然子类可以自由的去创建Bean，但是创建Bean之前的准备工作以及创建Bean之后对Bean的处理工作是AbstractFactoryBean设定好了的，子类不需要关心，也没权力关心，在这个例子中父类只负责一些前置判断工作。

工厂方法模式非常的有趣，它给了子类创建实例的自由，又严格的规定了实例创建前后的业务流程。

# 建造者模式

# 原型模式

# 代理模式

## 简介

代理模式是常用的java设计模式，他的特征是代理类与委托类有同样的接口，代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等。代理类与委托类之间通常会存在关联关系，一个代理类的对象与一个委托类的对象关联，代理类的对象本身并不真正实现服务，而是通过调用委托类的对象的相关方法，来提供特定的服务。   
　　按照代理的创建时期，代理类可以分为两种。   
　　静态代理：由程序员创建或特定工具自动生成源代码，再对其编译。在程序运行前，代理类的.class文件就已经存在了。   
　　动态代理：在程序运行时，运用反射机制动态创建而成。

　　为什么使用动态代理？因为动态代理可以对请求进行任何处理。  
　　哪些地方需要动态代理？不允许直接访问某些类；对访问要做特殊处理等。

public interface IHello {  
 void sayHello(String name);  
}

public class Hello implements IHello {  
 @Override  
 public void sayHello(String name) {  
 System.*out*.println("你好："+name);  
 }  
}

## 静态代理

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-04-25 16:25  
 \* 与核心业务有关的辅助性操作  
 \*/*public class StaticProxy implements IHello {  
 private Logger logger = LoggerFactory.*getLogger*(this.getClass());  
 private IHello iHello;  
  
 public StaticProxy(IHello iHello){  
 this.iHello=iHello;  
 }  
  
 @Override  
 public void sayHello(String name) {  
 logger.info(System.*currentTimeMillis*()+"：开始！");  
 iHello.sayHello(name);  
 logger.info(System.*currentTimeMillis*()+"：结束！");  
 }  
}

## 动态代理

### JDK动态代理

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-04-25 16:24  
 \* \* 只要你是采用面向接口编程,那么,你的任何对象的方法执行之前要加上记录日志的操作都是可以的.  
 \* 他(DynaPoxyHello)自动去代理执行被代理对象(Hello)中的每一个方法,  
 \* 一个java.lang.reflect.InvocationHandler接口就把我们的代理对象和被代理对象解藕了.  
 \*  
 \*InvocationHandler是JDK动态代理的核心，生成的代理对象的方法调用都会委托到InvocationHandler.invoke()方法  
 \* JdkDynamicAopProxy是通过接口实现动态代理类，主要方法是getProxy(ClassLoader classLoader)，代理类生成之后再调用目标方法时就会调用invoke方法。  
 \*  
 \* 实现InvocationHandler接口，该接口定义了一个 invoke(Object proxy, Method method, Object[] args)的方法，  
 \* 1,proxy是最终生成的代理实例，一般不会用到；  
 \* 2,method是被代理目标实例的某个具体方法，通过它可以发起目标实例方法的反射调用；  
 \* 3,args是通过被代理实例某一个方法的入参，在方法反射调用时使用。  
 \*/*public class JDKDynamicProxy implements InvocationHandler {  
 private Logger logger = LoggerFactory.*getLogger*(this.getClass());  
  
 //被代理类的对象  
 private Object delegate;  
  
 //绑定被代理对象  
 public Object bind(Object delegate) {  
 this.delegate = delegate;  
 //返回实现了被代理类所实现的所有接口的Object对象，即动态代理，需要强制转型  
 //创建代理对象，注意这里被代理的对象类必须实现至少一个接口  
 return Proxy.*newProxyInstance*(delegate.getClass().getClassLoader(), delegate.getClass().getInterfaces(), this);  
 }  
  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 logger.info(System.*currentTimeMillis*() + "：开始！");

// 调用原始对象的方法  
 Object object = method.invoke(this.delegate, args);  
 if (method.getName().matches("say[a-zA-Z0-9]+")) {  
 logger.info("这是一个以say开头的方法！");  
 }  
 logger.info(System.*currentTimeMillis*() + "：结束！");  
 return object;  
 }  
}

JDK自从1.3版本开始，就引入了动态代理，并且经常被用来动态地创建代理。JDK的动态代理用起来非常简单，但它有一个限制，就是使用动态代理的对象必须实现一个或多个接口。比如上面的Hello类，实现了IHello接口，所以可以用JDK的动态代理。如果想代理没有实现接口的继承的类，该怎么办？ CGLIB就是最好的选择。

### CGLIB动态代理

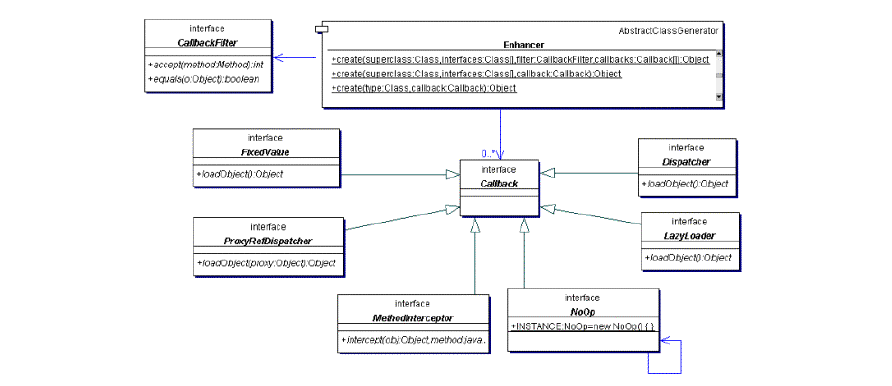
CGlib是一个强大的，高性能，高质量的Code生成类库。它可以在运行期扩展Java类与实现Java接口。其底层是通过小而快的字节码处理框架ASM（http://forge.ow2.org/projects/asm，使用BSD License）来转换字节码并生成新的类。大部分功能实际上是asm所提供的，CGlib只是封装了asm，简化了asm的操作，实现了在运行期动态生成新的class。

　　CGlib被许多AOP的框架使用，例如Spring AOP和dynaop，为他们提供方法的interception（拦截）；最流行的OR Mapping工具hibernate也使用CGLIB来代理单端single-ended（多对一和一对一）关联（对集合的延迟抓取，是采用其他机制实现的）；EasyMock和jMock是通过使用模仿（moke）对象来测试java代码的包，它们都通过使用CGLIB来为那些没有接口的类创建模仿（moke）对象。

　　CGLIB包的基本代码很少，但学起来有一定的困难，主要是缺少文档，API描述过于简单，这也是开源软件的一个不足之处。目前CGLIB的版本是cglib-2.2.jar，主要由一下部分组成：  
　　（1）net.sf.cglib.core：底层字节码处理类，他们大部分与ASM有关系。  
　　（2）net.sf.cglib.transform：编译期或运行期类和类文件的转换。  
　　（3）net.sf.cglib.proxy ：实现创建代理和方法拦截器的类。  
　　（4）net.sf.cglib.reflect ：实现快速反射和C#风格代理的类。  
　　（5）net.sf.cglib.util：集合排序工具类。  
　　（6）net.sf.cglib.beans：JavaBean相关的工具类。

CGLIB包是在ASM之上的一个高级别的层。对代理那些没有实现接口的类非常有用。本质上，它是通过动态的生成一个子类去覆盖所要代理类的不是final的方法，并设置好callback，则原有类的每个方法调用就会转变成调用用户定义的拦截方法（interceptors），这比JDK动态代理方法快多了。可见，Cglib的原理是对指定的目标类动态生成一个子类，并覆盖其中方法实现增强，但因为采用的是继承，所以不能对final修饰的类和final方法进行代理。

下图表示Cglib常用到的几类。



创建一个具体类的代理时，通常要用到的CGLIB包的APIs：

**net.sf.cglib.proxy.Callback**接口：在CGLIB包中是一个很关键的接口，所有被net.sf.cglib.proxy.Enhancer类调用的回调（callback）接口都要继承这个接口。

**net.sf.cglib.proxy.MethodInterceptor**接口：是最通用的回调（callback）类型，它经常被AOP用来实现拦截（intercept）方法的调用。能够满足任何的拦截（interception ）需要，当对有些情况下可能过度。这个接口只定义了一个方法。

代理类的所有方法经常会用到回调（callback），当然你也可以使用net.sf.cglib.proxy.CallbackFilter 有选择的对一些方法使用回调（callback），这种考虑周详的控制特性在JDK的动态代理中是没有的。CallbackFilter可以实现不同的方法使用不同的回调方法。所以CallbackFilter称为"回调选择器"更合适一些。  
CallbackFilter中的accept方法，根据不同的method返回不同的值i，这个值是在callbacks中callback对象的序号，就是调用了callbacks[i]。

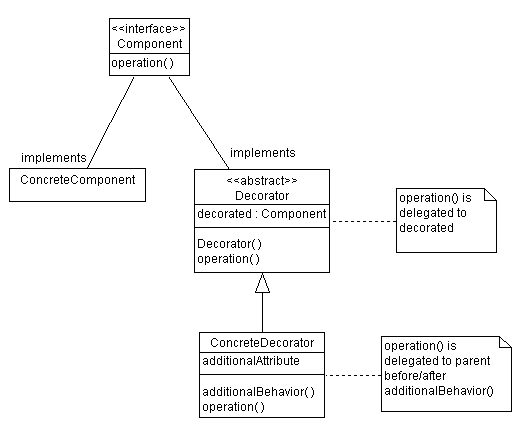
在JDK代理中，对 java.lang.reflect.InvocationHandler方法的调用对代理类的所有方法都有效。

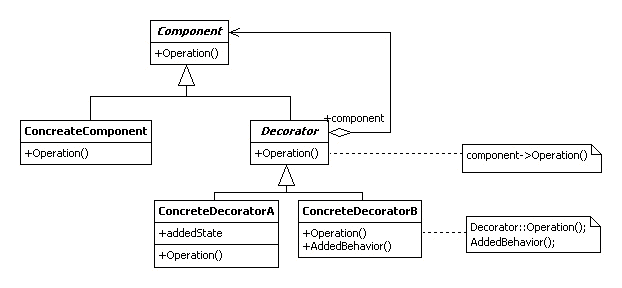
*/\*\*  
 \** ***@author*** *esther  
 \* 2018-04-25 16:24  
 \* Cglib的原理是对指定的目标类动态生成一个子类，并覆盖其中方法实现增强，但因为采用的是继承，所以不能对final修饰的类和final方法进行代理。  
 \*  
 \* 拦截器：实现MethodInterceptor接口的类，在intercept方法中实现对代理目标类的方法拦截。  
 \* 但同时Cglib为简化和提高性能提供了一些专门的回调类型如FixedValue（可以在实现的方法loadObject中指定返回固定值，而不调用目标类函数）、  
 \* NoOp（把对回调方法的调用直接委派到这个方法的父类，即不进行拦截）  
 \*/*public class CGLIBDynamicProxy implements MethodInterceptor {  
 private Logger logger = LoggerFactory.*getLogger*(this.getClass());  
  
 private Enhancer enhancer = new Enhancer();  
  
 //　要注意的是，target类是作为产生的代理的父类传进来的。不同于JDK的动态代理，它不能在创建代理时传target对象，target对象必须被CGLIB包来创建。  
 public Object getProxy(Class targetClass) {  
 enhancer.setSuperclass(targetClass); //① 设置需要创建子类的类  
 // 设置单一回调对象，在回调中拦截对目标方法的调用  
 enhancer.setCallback(this);  
 return enhancer.create(); //②通过字节码技术动态创建子类实例  
  
 }  
  
 //③拦截父类所有方法的调用  
  
 */\*\*  
 \* \* 回调方法:在代理实例上拦截并处理目标方法的调用，返回结果  
 \*  
 \** ***@param*** *obj 代理类  
 \** ***@param*** *method 被代理的方法  
 \** ***@param*** *args 该方法的参数数组  
 \** ***@param*** *proxy  
 \** ***@return*** *\** ***@throws*** *Throwable  
 \*/* @Override  
 public Object intercept(Object obj, Method method, Object[] args,  
 MethodProxy proxy) throws Throwable {  
 PerformanceMonitor.*begin*(obj.getClass().getName() + "." + method.getName());//③-1  
 // 调用目标方法，用methodProxy,  
 // 而不是原始的method，以提高性能  
 Object result = proxy.invokeSuper(obj, args); //③-2 使用Cglib代理调用  
 PerformanceMonitor.*end*();//③-1通过代理类调用父类中的方法  
 return result;  
 }  
}

# 装饰者模式

## 简介

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。





## 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。）

优点：比静态继承更具灵活性，**Avoids feature-laden classes high up in the hierarchy，**装饰对象和被装饰对象不一样。

缺点：产生过多相似的对象，不易排错！

public interface IProgrammer {  
 void coding();  
}

public class JavaProgrammer implements IProgrammer {  
 @Override  
 public void coding() {  
 System.*out*.println("Java程序员编码。");  
 }  
}

public abstract class Decorator implements IProgrammer {  
 private IProgrammer programmer;  
 public Decorator(IProgrammer programmer){  
 super();  
 this.programmer=programmer;  
 }  
 @Override  
 public void coding() {  
 programmer.coding();  
 }  
}

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-05-07 16:46  
 \* 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  
 \*/*public class DesignDecorator extends Decorator {  
 public DesignDecorator(IProgrammer programmer) {  
 super(programmer);  
 }  
  
 @Override  
 public void coding() {  
 System.*out*.println("编码前先设计。");  
 super.coding();  
 }  
}

# 适配器模式

## 简介

将一个类的接口转换成客户希望的另外一个接口。Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一下工作。--Gang of Four

主要分为三类：类的适配、对象的适配、接口的适配。

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-05-04 10:52  
 \* 介绍：src类: 我们有的220V电压  
 \*/*public class Voltage220 {  
 public int output220V() {  
 int src = 220;  
 System.*out*.println("我是" + src + "V");  
 return src;  
 }  
}

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-05-04 11:16  
 \* 介绍：src接口  
 \*/*public interface IVoltage220 {  
 int output220V();  
}

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-05-04 10:54  
 \* 介绍：dst接口：客户需要的5V电压  
 \*/*public interface IVoltage5 {  
 int output5V();  
}

## 类的适配

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-05-04 10:57  
 \* 介绍：Adapter类：完成220V-5V的转变  
 \* 通过继承src类，实现 dst 类接口，完成src->dst的适配。  
 \*/  
  
/\*\*  
 \* 类适配器需要继承src类这一点算是一个缺点，  
 因为这要求dst必须是接口，有一定局限性;  
 且src类的方法在Adapter中都会暴露出来，也增加了使用的成本。  
 但同样由于其继承了src类，所以它可以根据需求重写src类的方法，使得Adapter的灵活性增强了。  
 \*/*public class VoltageAdapter4Class extends Voltage220 implements IVoltage5 {  
  
 @Override  
 public int output5V() {  
 int src = output220V();  
 System.*out*.println("适配器工作开始适配电压");  
 int dst = src / 44;  
 System.*out*.println("适配完成后输出电压：" + dst);  
 return dst;  
 }  
}

## 对象的适配

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-05-04 10:56  
 \* 介绍：对象适配器模式：  
 \* 持有 src类，实现 dst 类接口，完成src->dst的适配。 。以达到解决兼容性的问题。  
 \*/*public class VoltageAdapter4Object implements IVoltage5 {  
 private Voltage220 voltage220;  
  
 public VoltageAdapter4Object(Voltage220 voltage220) {  
 this.voltage220 = voltage220;  
 }  
  
 @Override  
 public int output5V() {  
 int dst = 0;  
 if (Objects.*nonNull*(voltage220)) {  
 int src = voltage220.output220V();  
 System.*out*.println("对象适配器工作，开始适配电压");  
 dst = src / 44;  
 System.*out*.println("适配完成后输出电压：" + dst);  
 }  
 return dst;  
 }  
}

## 接口的适配

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-05-04 10:57  
 \* 缺省适配器模式：  
当不需要全部实现接口提供的方法时，可先设计一个抽象类实现接口，并为该接口中每个方法提供一个默认实现（空方法），  
那么该抽象类的子类可有选择地覆盖父类的某些方法来实现需求，它适用于一个接口不想使用其所有的方法的情况。  
 \*/*public class VoltageAdapter4Interface implements IVoltage220, IVoltage5 {  
 @Override  
 public int output220V() {  
 int src = 220;  
 System.*out*.println("我是" + src + "V");  
 return src;  
 }  
  
 @Override *public int output5V() {  
 int src = output220V();* System.*out*.println("适配器工作开始适配电压");  
 int dst = src / 44;  
 System.*out*.println("适配完成后输出电压：" + dst);  
 return dst;  
 }  
}

## 三种适配的区别

类适配器，以类给到，在Adapter里，就是将src当做类，继承，  
 对象适配器，以对象给到，在Adapter里，将src作为一个对象，持有。  
 接口适配器，以接口给到，在Adapter里，将src作为一个接口，实现。

# 外观模式

# 桥接模式

# 组合模式

# 享元模式

# 策略模式

# 管道模式

# 模板方法

# 观察者模式

## 简介

观察者模式是对象的行为模式，外号非常多...！！又叫发布-订阅(Publish/Subscribe)模式、模型-视图(Model/View)模式、源-监听器(Source/Listener)模式或从属者(Dependents)模式。

     观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

## 针对的问题

被观察者对象在状态变化时，通知所有观察者对象，使它们能够自动更新自己。保证整体的数据一致性。此种模式通常被用来实现事件处理系统。

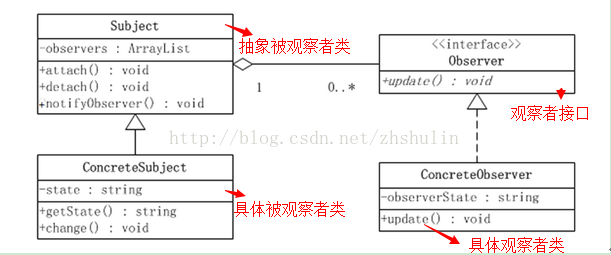
适用性：

1）当一个抽象模型有两个方面,其中一个方面依赖于另一方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。

2）当对一个对象的改变需要同时改变其它对象,而不知道具体有多少对象有待改变。

3）当一个对象必须通知其它对象，而它又不能假定其它对象是谁。

## 角色组成



**Ø 抽象主题(Subject)角色：**

  抽象主题角色把所有对观察者对象的引用保存在一个聚集（比如ArrayList对象）里，每个主题都可以有任何数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象，抽象主题角色又叫做抽象被观察者(Observable)角色。

**Ø 具体主题(ConcreteSubject)角色：**

  将有关状态存入具体观察者对象；在具体主题的内部状态改变时，给所有登记过的观察者发出通知。具体主题角色又叫做具体被观察者(Concrete Observable)角色。

**Ø 抽象观察者(Observer)角色：**

  为所有的具体观察者定义一个接口，在得到主题的通知时更新自己，这个接口叫做更新接口。

**Ø 具体观察者(ConcreteObserver)角色：**

  存储与主题的状态自恰的状态。具体观察者角色实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题的状态协调。如果需要，具体观察者角色可以保持一个指向具体主题对象的引用。

## 举例说明

Subject：抽象被观察者

1. **package** org.zsl.designmodel.observer;
2. **import** java.util.ArrayList;
3. **import** java.util.List;
4. /\*\*
5. \* 被观察者角色
6. \*/
7. **public** **abstract** **class** Subject {
8. **private** List<Observer> list = **new** ArrayList<Observer>();
9. /\*\*
10. \* 注册观察者对象
11. \* @param observer  观察者
12. \*/
13. **public** **void** registerObserver(Observer observer){
14. list.add(observer);
15. System.out.println("注册了一个观察者角色！");
16. }
17. /\*\*
18. \* 删除观察者对象
19. \* @param observer  观察者
20. \*/
21. **public** **void** removeObserver(Observer observer){
22. list.remove(observer);
23. }
24. /\*\*
25. \* 通知所有注册的观察者对象，我的状态改变咯
26. \* @param newState
27. \*/
28. **public** **void** notifyAllObservers(String newState){
29. **for**(Observer observer : list){
30. observer.update(newState);
31. }
32. }
33. }

Observer ：观察者接口，只有一个update()方法

1. **package** org.zsl.designmodel.observer;
2. /\*\*
3. \* 观察者接口
4. \*/
5. **public** **interface** Observer {
6. /\*\*
7. \* 更新接口
8. \* @param state 更新的状态
9. \*/
10. **public** **void** update(String state);
11. }

Heater：热水器，一个具体被观察者

1. **package** org.zsl.designmodel.observer;
2. **public** **class** Heater **extends** Subject {
3. **private** **int** temperature;
4. **public** **int** getTemperature() {
5. **return** temperature;
6. }
7. **public** **void** setTemperature(**int** temperature) {
8. **this**.temperature = temperature;
9. }
10. **public** **void** boilWater(){
11. **for**(**int** i=95;i<105;i++){
12. temperature = i;
13. **this**.setTemperature(temperature);   //设置新的温度
14. **this**.notifyAllObservers(Integer.toString(temperature)); //通知所有注册的观察者
15. }
16. }
17. }

Display：既是观察者，又是被观察者

1. **package** org.zsl.designmodel.observer;
2. /\*\*
3. \* 显示器，既是观察者又是被观察者
4. \*/
5. **public** **class** Display **extends** Subject **implements** Observer {
6. **private** **boolean** isBoiled = **false**;   //表示水是否烧开了
7. **private** **int** displayTemperature;     //定义显示器的温度
8. **public** **boolean** isBoiled() {
9. **return** isBoiled;
10. }
11. **public** **void** setBoiled(**boolean** isBoiled) {
12. **this**.isBoiled = isBoiled;
13. }
14. @Override
15. **public** **void** update(String state) {
16. displayTemperature = Integer.parseInt(state);
17. System.out.println("当前显示器显示的温度是："+displayTemperature);
18. **this**.displayTemperature(displayTemperature);
19. }
20. **private** **void** displayTemperature(**int** temperature){
21. **if**(temperature>100){
22. **this**.setBoiled(**true**);
23. **this**.notifyAllObservers(Boolean.toString(isBoiled));
24. }
25. }
26. }

Alerm：只是观察者

1. **package** org.zsl.designmodel.observer;
2. **public** **class** Alarm **implements** Observer {
3. **private** String observerState;       //定义观察者的状态
4. @Override
5. **public** **void** update(String state) {
6. Boolean flag = Boolean.parseBoolean(state);
7. **if**(flag){
8. System.out.println("报警器响了，水温超过100度了。");
9. }
10. }
11. }

测试

1. **package** org.zsl.designmodel.observer;
2. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
4. Heater heater = **new** Heater();
5. Display display = **new** Display();
6. Alarm alarm = **new** Alarm();
7. heater.registerObserver(display);
8. display.registerObserver(alarm);
9. heater.boilWater();
10. }
11. }

结果：



## 优点

a、支持松耦合和减少依赖性

客户端不再依赖于观察器，因为通过使用主体和 Observer 接口对客户端进行了隔离。 许多框架具有此优点，在这些框架中的应用程序组件可以注册为当（低级）框架事件发 生时得到通知。结果，框架将调用应用程序组件，但不会依赖于它。

b、提高了应用程序的可维护性和重用性

面向对象设计的一个原则是：系统中的每个类将重点放在某一个功能上，而不是其他方面。一个对象只做一件事情，并且将他做好。观察者模式在模块之间划定了清晰的界限，提高了应用程序的可维护性和重用性。

c、观察器数目可变

观察器可以在运行时附加和分离，因为主体对于观察器数目没有任何假定。此功能在这样的情况下是很有用的：观察器数在设计时是未知的。例如，如果用户在应用程序中打开的每个窗口都需要一个观察器。

综合来讲，使用观察者模式最大的好处就是减少死循环式的轮循带来的资源无端消耗，并且有着良好的可扩展性。

## 缺点

a、性能降低。

     在许多实现中，观察器的 update() 方法可能与主体在同一线程中执行。如果观察器列表很长，则执行 Notify() 方法可能需要很长时间。抽取对象依赖性并不意味着添加观察器对应用程序没有任何影响。

b、内存泄漏。

      在 Observer 中使用的回调机制（当对象注册为以后调用时）会产生一个常见的错误，从而导致内存泄漏，甚至是在托管的 C# 代码中。假定观察器超出作用范围，但忘记取消对主体的订阅，那么主体仍然保留对观察器的引用。此引用防止垃圾收集在主体对象也被破坏之前重新分配与观察器关联的内存。如果观察器的生存期比主体的生存期短得多（通常是这种情况），则会导致严重的内存泄漏。

c、隐藏的依赖项。

      观察器的使用将显式依赖性（通过方法调用）转变为隐式依赖性（通过观察器）。如果在整个应用程序中广泛地使用观察器，则开发人员几乎不可能通过查看源代码来了解所发生的事情。这样，就使得了解代码更改的含意非常困难。此问题随传播级别急剧增大（例如，充当 Subject 的观察器）。因此，应该仅在少数定义良好的交互（如 Model-View-Controller 模式中模型和视图之间的交互）中使用观察器。最好不要在域对象之间使用观察器。

d、测试 / 调试困难。

    尽管松耦合是一项重大的体系结构功能，但是它可以使开发更困难。将两个对象去耦的情况越多，在查看源代码或类的关系图时了解它们之间的依赖性就越难因此，仅当可以安全地忽略两个对象之间的关联时才应该将它们松耦合（例如，如果观察器没有副作用）。

# 迭代子模式

# 责任链模式

# 命令模式

# 备忘录模式

# 状态模式

# 访问者模式

# 中介者模式

# 解释器模式

# 并发型模式

# 线程池模式

# Spring中的设计模式

## 单例模式

在Spring中，bean可以被定义为两种模式：prototype(多例)和singleton(单例)。

Singleton(单例)：只有一个共享的实例存在，所有对这个bean的请求都会返回这个唯一的实例。

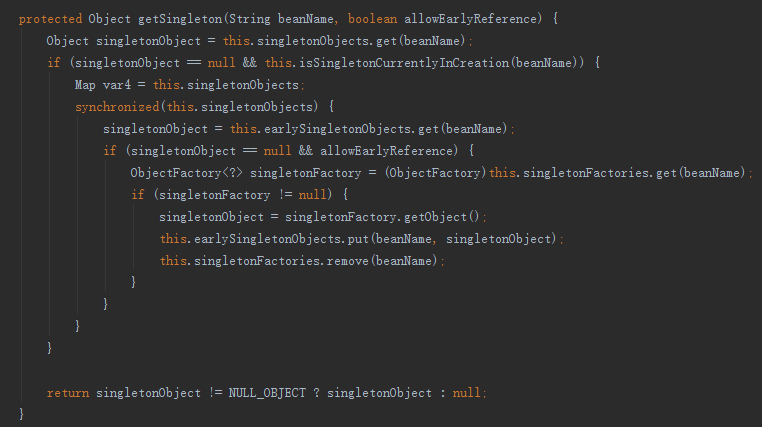
Prototype(多例)：对这个bean的每次请求都会创建一个新的bean实例，类似于new。

Spring依赖注入Bean实例默认是单例的，我们由此展开。

Spring的依赖注入（包括lazy-init方式）都是发生在AbstractBeanFactory的getBean里。getBean的doGetBean方法调用getSingleton进行bean的创建。lazy-init方式，在容器初始化时候进行调用，非lazy-init方式，在用户向容器第一次索要bean时进行调用。



通过将工厂函数传入getSingleton函数中，就可以获得一个Bean单例。单例的生成是通过修改createBean函数的参数实现的，其中mbd是一个RootBeanDefinition类，它存储了生成Bean实例所需要的信息。在createBean之中的代码里，程序调用实例化Bean的函数initializeBean。

 从上面代码可以看到，spring依赖注入时，使用了**双重判断加锁**的单例模式，首先从缓存singletonObjects（实际上是一个map）中获取bean实例，如果为null，对缓存singletonObjects加锁，然后再从缓存中获取bean，如果继续为null，就创建一个bean。

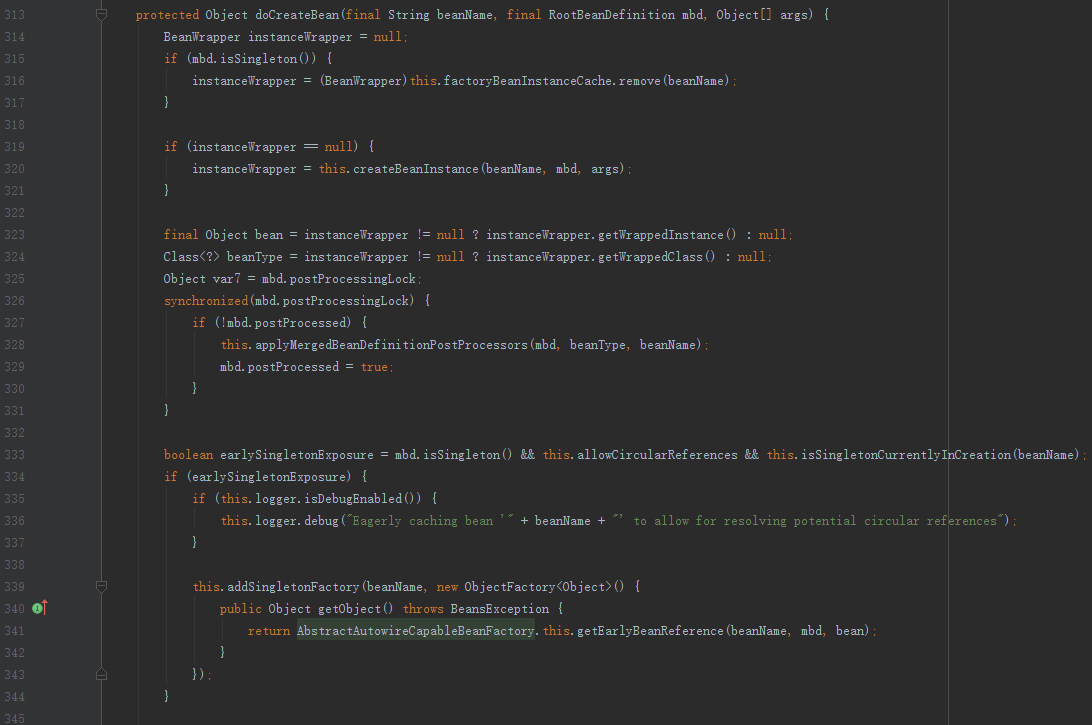
**这样双重判断，能够避免在加锁的瞬间，有其他依赖注入引发bean实例的创建，从而造成重复创建的结果。**

     在这里Spring并没有使用私有构造方法来创建bean，而是通过singletonFactory.getObject()返回具体beanName对应的ObjectFactory来创建bean。

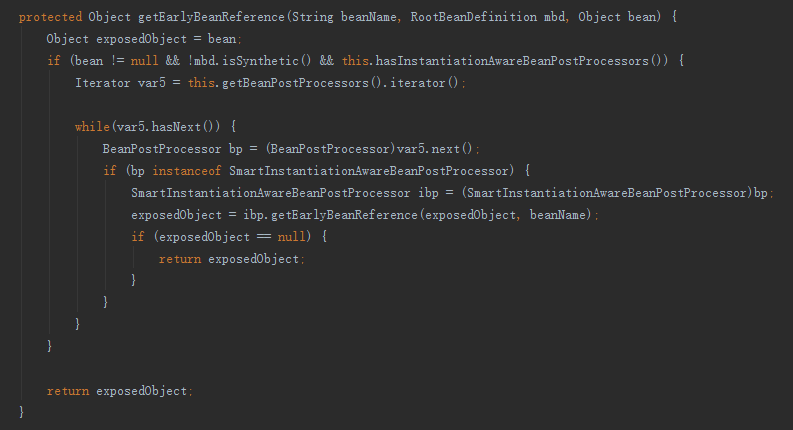
我们一路跟踪下去，发现实际上是调用了AbstractAutowireCapableBeanFactory的doCreateBean方法，返回了BeanWrapper包装并创建的bean实例。

（ObjectFactory主要检查是否有用户定义的BeanPostProcessor后处理内容，并在创建bean时进行处理，如果没有，就直接返回bean本身）

见如下代码：

320行createBeanInstance()创建bean实例返回给BeanWrapper

339行addSingletonFactory()增加beanName和ObjectFactory的键值对应关系。



getEarlyBeanReference获取bean的所有后处理器，并进行处理。如果是SmartInstantiationAwareBeanPostProcessor类型，就进行处理，如果没有相关处理内容，就返回默认的实现。

## 工厂模式

// AbstractFactoryBean.java  
// 继承了FactoryBean，工厂Bean的主要作用是为了实现getObject()返回Bean实例  
public abstract class AbstractFactoryBean<T> implements FactoryBean<T>, BeanClassLoaderAware, BeanFactoryAware, InitializingBean, DisposableBean {  
  
 // 定义了获取对象的前置判断工作，创建对象的工作则交给了一个抽象方法  
// 这里判断了Bean是不是单例并且是否已经被加载过了（未初始化但加载过了，这个问题涉及到Spring处理循环依赖，以后会讨论到）  
 public final T getObject() throws Exception {  
 return this.isSingleton()?(this.initialized?this.singletonInstance:this.getEarlySingletonInstance()):this.createInstance();  
 }  
 // 由子类负责具体创建对象  
 protected abstract T createInstance() throws Exception;  
}

之所以这么写是因为这种写法带来了两个好处:

（1） 保证了创建Bean的方式的多样性   
 Bean工厂有很多种，它们负责创建各种各样不同的Bean，比如Map类型的Bean，List类型的Bean，Web服务Bean，子类们不需要关心单例或非单例情况下是否需要额外操作，只需要关心如何创建Bean，并且创建出来的Bean是多种多样的。

（2） 严格规定了Bean创建前后的其它动作   
 虽然子类可以自由的去创建Bean，但是创建Bean之前的准备工作以及创建Bean之后对Bean的处理工作是AbstractFactoryBean设定好了的，子类不需要关心，也没权力关心，在这个例子中父类只负责一些前置判断工作。

工厂方法模式非常的有趣，它给了子类创建实例的自由，又严格的规定了实例创建前后的业务流程。

## 适配器模式

public interface Controller {  
}

public class AnnotationController implements Controller {  
 public void doAnnotationHandler() {  
 System.*out*.println("执行AnnotationController.doAnnotationHandler()");  
 }  
}

public class HttpController implements Controller {  
 public void doHttpHandler() {  
 System.*out*.println("执行HttpController.doHttpHandler()");  
 }  
}

public class SimpleController implements Controller {  
 public void doSimplerHandler(){  
 System.*out*.println("执行SimpleController.doSimplerHandler()");  
 }  
}

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-05-04 14:13  
 \* 定义一个Adapter接口  
 \*/*public interface HandlerAdapter {  
 public boolean supports(Object handler);  
 public void handle(Object handler);  
}

public class AnnotationHandlerAdapter implements HandlerAdapter {  
 @Override  
 public boolean supports(Object handler) {  
 return handler instanceof AnnotationController;  
 }  
  
 @Override  
 public void handle(Object handler) {  
 ((AnnotationController)handler).doAnnotationHandler();  
 }  
}

public class HttpHandlerAdapter implements HandlerAdapter {  
 @Override  
 public boolean supports(Object handler) {  
 return handler instanceof HttpController;  
 }  
  
 @Override  
 public void handle(Object handler) {  
 ((HttpController)handler).doHttpHandler();  
 }  
}

public class SimpleHandlerAdapter implements HandlerAdapter {  
 @Override  
 public boolean supports(Object handler) {  
 return (handler instanceof SimpleController);  
 }  
  
 @Override  
 public void handle(Object handler) {  
 ((SimpleController) handler).doSimplerHandler();  
 }  
}

Client类：

public class DispatcherServlet {  
 public static List<HandlerAdapter> *handlerAdapters* = new ArrayList<>();  
  
 public DispatcherServlet() {  
 *handlerAdapters*.add(new AnnotationHandlerAdapter());  
 *handlerAdapters*.add(new SimpleHandlerAdapter());  
 *handlerAdapters*.add(new HttpHandlerAdapter());  
 }  
  
 public void doDispatch(Controller controller) {  
 //此处模拟SpringMVC从request取handler的对象，仅仅new出，可以出，  
 //不论实现何种Controller，适配器总能经过适配以后得到想要的结果  
// HttpController controller = new HttpController();  
// AnnotationController controller = new AnnotationController();  
// SimpleController controller = new SimpleController();  
 //得到对应适配器  
 HandlerAdapter handlerAdapter = getHandlerAdapter(controller);  
 //通过适配器执行对应的controller对应方法  
 handlerAdapter.handle(controller);  
 }  
  
 public HandlerAdapter getHandlerAdapter(Controller controller) {  
 for (HandlerAdapter handlerAdapter : *handlerAdapters*) {  
 if (handlerAdapter.supports(controller)) {  
 return handlerAdapter;  
 }  
 }  
 return null;  
 }  
}

## 代理模式

## 装饰者模式

通过使用修饰模式，可以在运行时扩充一个类的功能。原理是：增加一个修饰类包裹原来的类，包裹的方式一般是通过在将原来的对象作为修饰类的构造函数的参数。装饰类实现新的功能，但是，在不需要用到新功能的地方，它可以直接调用原来的类中的方法。修饰类必须和原来的类有相同的接口。

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。

Spring中类中带有Wrapper的都是包装类，如下创建bean就是典型的装饰器模式。



## 策略模式

### 定义

在策略模式（Strategy Pattern）中，一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。在策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的 context 对象。策略对象改变 context 对象的执行算法。策略模式实际就是一堆算法族的封装。

### Spring中策略模式的应用

当bean需要访问资源配置文件时，Spring有两种方式

* 代码中获取Rescource实例
* 依赖注入   
  第一种方式需要获取rescource资源的位置，代码中耦合性太高，而今我们一直使用注解，依赖注入的方式去获取。这样的话就无需修改程序，只改配置文件即可。

<beans>

<bean id="test" class="com.example.Test">

<!-- 注入资源 -->

<property name="tmp" value="classpath:book.xml"/>

</bean>

</beans>

在依赖注入的过程中，Spring会调用ApplicationContext 来获取Resource的实例。然而，Resource 接口封装了各种可能的资源类型，包括了：UrlResource，ClassPathResource，FileSystemResource等，Spring需要针对不同的资源采取不同的访问策略。在这里，Spring让ApplicationContext成为了资源访问策略的“决策者”。在资源访问策略的选择上，Spring采用了策略模式。当 Spring 应用需要进行资源访问时，它并不需要直接使用 Resource 实现类，而是调用 ApplicationContext 实例的 getResource() 方法来获得资源，ApplicationContext 将会负责选择 Resource 的实现类，也就是确定具体的资源访问策略，从而将应用程序和具体的资源访问策略分离开来。

ApplicationContext ctx = new Class PathXmlApplicationContext("bean.xml");

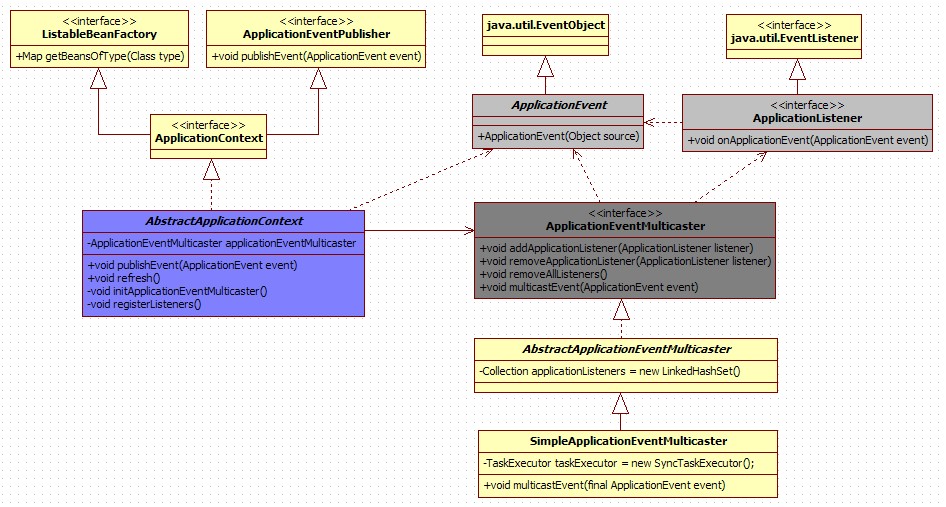
Resource res = ctx.getResource("book.xml");

上面的代码中，Spring 将采用和 ApplicationContext 相同的策略来访问资源。即： ApplicationContext 是 ClassPathXmlApplicationContext，则res 就是 ClassPathResource 实例。若将代码改为：

ApplicationContext ctx = new Class FileSystemXmlApplicationContext("bean.xml");

则再次调用ctx.getResource时，res 就是 ClassPathResource 实例。

## 观察者模式



### ApplicationEvent

package org.springframework.context;  
import java.util.EventObject;  
public abstract class ApplicationEvent extends EventObject {  
 private static final long serialVersionUID = 7099057708183571937L;  
 private final long timestamp = System.currentTimeMillis();  
  
 public ApplicationEvent(Object source) {  
 super(source);  
 }  
 public final long getTimestamp() {  
 return this.timestamp;  
 }  
}

ApplicationEvent继承自jdk的EventObject,所有的事件都需要继承ApplicationEvent,并且通过source得到事件源。该类的实现类ApplicationContextEvent表示ApplicaitonContext的容器事件。

### ApplicationListener

package org.springframework.context;  
import java.util.EventListener;  
public interface ApplicationListener<E extends ApplicationEvent> extends EventListener {  
 void onApplicationEvent(E var1);  
}

ApplicationListener继承自jdk的EventListener,所有的监听器都要实现这个接口,这个接口只有一个onApplicationEvent()方法,该方法接受一个ApplicationEvent或其子类对象作为参数,在方法体中,可以通过对Event类的判断来进行相应的处理。当事件触发时所有的监听器都会收到消息,如果你需要对监听器的接收顺序有要求,可是实现该接口的一个实现SmartApplicationListener,通过这个接口可以指定监听器接收事件的顺序。

### ApplicationContext

事件机制的实现需要三个部分：事件源,事件,事件监听器。在上面介绍的ApplicationEvent就相当于事件,ApplicationListener相当于事件监听器,这里的事件源说的就是applicaitonContext。

　　ApplicationContext是spring中的全局容器,翻译过来是"应用上下文"的意思,它用来负责读取bean的配置文档,管理bean的加载,维护bean之间的依赖关系,可以说是负责bean的整个生命周期,再通俗一点就是我们平时所说的IOC容器。

     Application作为一个事件源,需要显示的调用publishEvent方法,传入一个ApplicationEvent的实现类对象作为参数,每当ApplicationContext发布ApplicationEvent时,所有的ApplicationListener就会被自动的触发。

ApplicationContext接口实现了ApplicationEventPublisher接口,后者有一个很重要的方法:

package org.springframework.context;  
public interface ApplicationEventPublisher {  
 void publishEvent(ApplicationEvent var1);  
 void publishEvent(Object var1);  
}

我们常用的ApplicationContext都继承了AbstractApplicationContext,像我们平时常见的ClassPathXmlApplicationContext、XmlWebApplicationContex也都是继承了它。AbstractApplicationcontext是ApplicationContext接口的抽象实现类,在该类中实现了publishEvent方法。

protected void publishEvent(Object event, ResolvableType eventType) {  
 Assert.notNull(event, "Event must not be null");  
 if (this.logger.isTraceEnabled()) {  
 this.logger.trace("Publishing event in " + this.getDisplayName() + ": " + event);  
 }  
  
 Object applicationEvent;  
 if (event instanceof ApplicationEvent) {  
 applicationEvent = (ApplicationEvent)event;  
 } else {  
 applicationEvent = new PayloadApplicationEvent(this, event);  
 if (eventType == null) {  
 eventType = ((PayloadApplicationEvent)applicationEvent).getResolvableType();  
 }  
 }  
  
 if (this.earlyApplicationEvents != null) {  
 this.earlyApplicationEvents.add(applicationEvent);  
 } else {  
 this.getApplicationEventMulticaster().multicastEvent((ApplicationEvent)applicationEvent, eventType);  
 }  
  
 if (this.parent != null) {  
 if (this.parent instanceof AbstractApplicationContext) {  
 ((AbstractApplicationContext)this.parent).publishEvent(event, eventType);  
 } else {  
 this.parent.publishEvent(event);  
 }  
 }  
  
}

在这个方法中,我们看到了一个getApplicationEventMulticaster().这就要牵扯到另一个类ApplicationEventMulticaster。

### ApplicationEventMulticaster

属于事件广播器,它的作用是把Applicationcontext发布的Event广播给所有的监听器。

public interface ApplicationEventMulticaster {  
 void addApplicationListener(ApplicationListener<?> var1);  
  
 void addApplicationListenerBean(String var1);  
  
 void removeApplicationListener(ApplicationListener<?> var1);  
  
 void removeApplicationListenerBean(String var1);  
  
 void removeAllListeners();  
  
 void multicastEvent(ApplicationEvent var1);  
  
 void multicastEvent(ApplicationEvent var1, ResolvableType var2);  
}

ApplicationEventMulticaster的实现类SimpleApplicationEventMulticaster：里面实现了multicastEvent()的具体方法。

public void multicastEvent(final ApplicationEvent event, ResolvableType eventType) {  
 ResolvableType type = eventType != null ? eventType : this.resolveDefaultEventType(event);  
 Iterator var4 = this.getApplicationListeners(event, type).iterator();  
  
 while(var4.hasNext()) {  
 final ApplicationListener<?> listener = (ApplicationListener)var4.next();  
 Executor executor = this.getTaskExecutor();  
 if (executor != null) {  
 executor.execute(new Runnable() {  
 public void run() {  
 SimpleApplicationEventMulticaster.this.invokeListener(listener, event);  
 }  
 });  
 } else {  
 this.invokeListener(listener, event);  
 }  
 }  
  
}  
  
private ResolvableType resolveDefaultEventType(ApplicationEvent event) {  
 return ResolvableType.forInstance(event);  
}  
  
protected void invokeListener(ApplicationListener listener, ApplicationEvent event) {  
 ErrorHandler errorHandler = this.getErrorHandler();  
 if (errorHandler != null) {  
 try {  
 listener.onApplicationEvent(event);  
 } catch (Throwable var6) {  
 errorHandler.handleError(var6);  
 }  
 } else {  
 try {  
 listener.onApplicationEvent(event);  
 } catch (ClassCastException var5) {  
 LogFactory.getLog(this.getClass()).debug("Non-matching event type for listener: " + listener, var5);  
 }  
 }  
}

　　在AbstractApplicationcontext中有一个applicationEventMulticaster的成员变量,提供了监听器Listener的注册方法。

public abstract class AbstractApplicationContext extends DefaultResourceLoader implements ConfigurableApplicationContext, DisposableBean {

private ApplicationEventMulticaster applicationEventMulticaster;

protected void registerListeners() {  
 Iterator var1 = this.getApplicationListeners().iterator();  
  
 while(var1.hasNext()) {  
 ApplicationListener<?> listener = (ApplicationListener)var1.next();  
 this.getApplicationEventMulticaster().addApplicationListener(listener);  
 }  
  
 String[] listenerBeanNames = this.getBeanNamesForType(ApplicationListener.class, true, false);  
 String[] var7 = listenerBeanNames;  
 int var3 = listenerBeanNames.length;  
  
 for(int var4 = 0; var4 < var3; ++var4) {  
 String listenerBeanName = var7[var4];  
 this.getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);  
 }  
  
 Set<ApplicationEvent> earlyEventsToProcess = this.earlyApplicationEvents;  
 this.earlyApplicationEvents = null;  
 if (earlyEventsToProcess != null) {  
 Iterator var9 = earlyEventsToProcess.iterator();  
  
 while(var9.hasNext()) {  
 ApplicationEvent earlyEvent = (ApplicationEvent)var9.next();  
 this.getApplicationEventMulticaster().multicastEvent(earlyEvent);  
 }  
 }  
}

### 举例说明

事件：

public class DIYEvent extends ApplicationEvent {  
 private static final long *serialVersionUID* = 7099057708183571977L;  
  
 public DIYEvent(String source) {  
 super(source);  
 }  
}

监听器：

@Component  
public class DIYListener implements ApplicationListener<DIYEvent> {  
 @Override  
 public void onApplicationEvent(DIYEvent diyEvent) {  
 System.*out*.println("自定义监听器执行");  
 System.*out*.println(diyEvent.getSource());  
 }  
}

有顺序的监听器：

@Component  
public class DIYOrderdListener implements SmartApplicationListener {  
 @Override  
 public boolean supportsEventType(Class<? extends ApplicationEvent> aClass) {  
 return aClass==DIYEvent.class;  
 }  
  
 @Override  
 public boolean supportsSourceType(Class<?> aClass) {  
 return true;  
 }  
  
 @Override  
 public void onApplicationEvent(ApplicationEvent applicationEvent) {  
 if (applicationEvent instanceof DIYEvent) {  
 System.*out*.println("自定义顺序监听器执行....DIYOrderdListener");  
 System.*out*.println(applicationEvent.getSource());  
 }  
 else{  
 System.*out*.println("没有满足的监听器.....DIYOrderdListener");  
 }  
 }  
  
 @Override  
 public int getOrder() {  
 return 1;  
 }  
}

测试类：

public class DIYTest{  
 private ApplicationContext applicationContext=new ClassPathXmlApplicationContext("classpath:/spring/applicationContext.xml");  
  
 @Test  
 public void diyTest(){  
 applicationContext.publishEvent(new DIYEvent("测试数据"));  
 }  
}

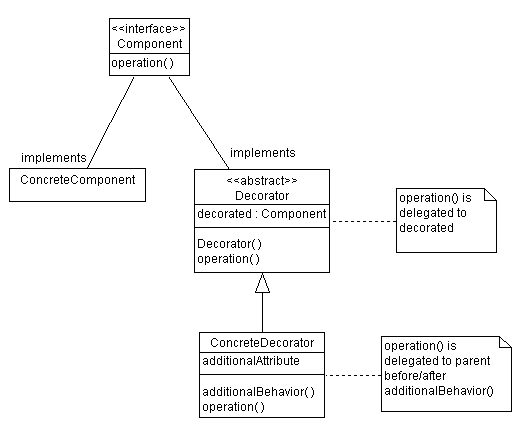
结果：

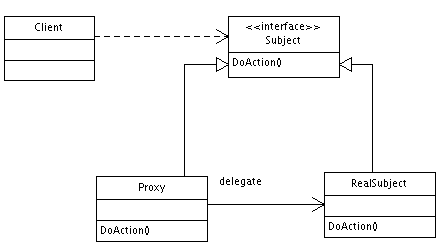
自定义监听器执行

测试数据

# 总结

## 装饰者模式和代理模式区别





  这两个图可能使我们产生困惑。这两个设计模式看起来很像。对装饰器模式来说，装饰者（decorator）和被装饰者（decoratee）都实现同一个接口。此外，不论我们使用哪一个模式，都可以很容易地在真实对象的方法前面或者后面加上自定义的方法。

       然而，实际上，在装饰器模式和代理模式之间还是有很多差别的。装饰器模式关注于在一个对象上动态的添加方法，然而代理模式关注于控制对对象的访问。换句话说，用代理模式，代理类（proxy class）可以对它的客户隐藏一个对象的具体信息。因此，当使用代理模式的时候，我们常常在一个代理类中创建一个对象的实例。并且，当我们使用装饰器模式的时候，我们通常的做法是将原始对象作为一个参数传给装饰者的构造器。

        代理模式中，代理类对被代理的对象有控制权，决定其执行或者不执行。而装饰模式中，装饰类对代理对象没有控制权，只能为其增加一层装饰，以加强被装饰对象的功能，仅此而已。

Spring中AOP用到了代理模式和反射机制。

JDK中的java.io.\*包大量使用了装饰者模式，如

OutputStream out = new DataOutputStream(new FileOutputStream("test.txt"));

## 适配器、装饰者、代理模式的区别

适配器模式，一个适配允许通常因为接口不兼容而不能在一起工作的类工作在一起，做法是将类自己的接口包裹在一个已存在的类中。

装饰器模式，原有的不能满足现有的需求，对原有的进行增强。

代理模式，同一个类而去调用另一个类的方法，不对这个方法进行直接操作。

在使用适配器模式的时候，我们必须同时持有原对象，适配对象，目标对象。

装饰器模式特点在于增强，他的特点是被装饰类和所有的装饰类必须实现同一个接口，而且必须持有被装饰的对象，可以无限装饰。

代理模式的特点在于隔离，隔离调用类和被调用类的关系，通过一个代理类去调用。

总的来说就是如下三句话：

 1 适配器模式是将一个类(a)通过某种方式转换成另一个类(b).  
 2 装饰模式是在一个原有类(a)的基础之上增加了某些新的功能变成另一个类(b).  
 3 代理模式是将一个类(a)转换成具体的操作类(b).

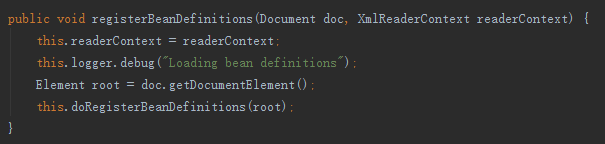
# Spring源码学习

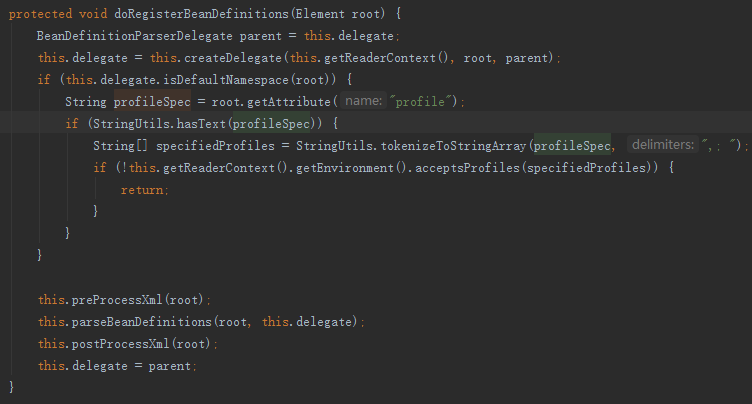
## 基础知识

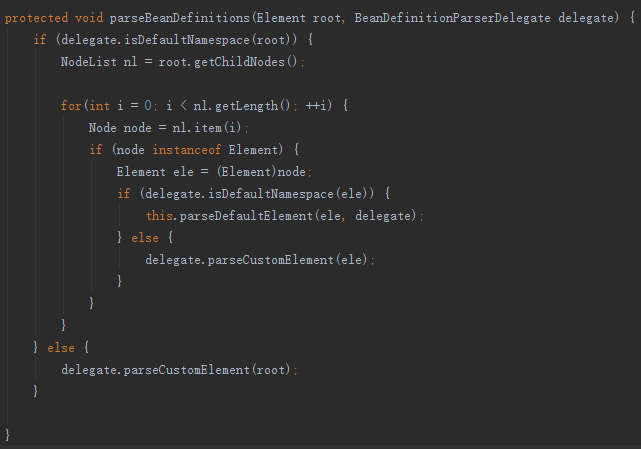
## 源码学习

### Bean加载流程

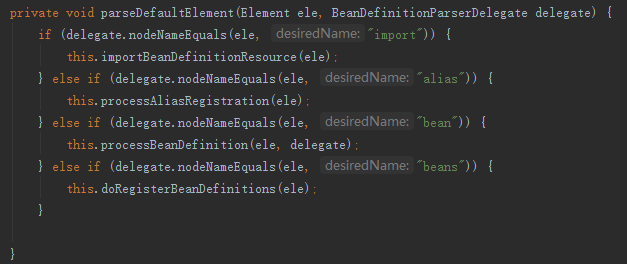
类DefaultBeanDefinitionDocumentReader



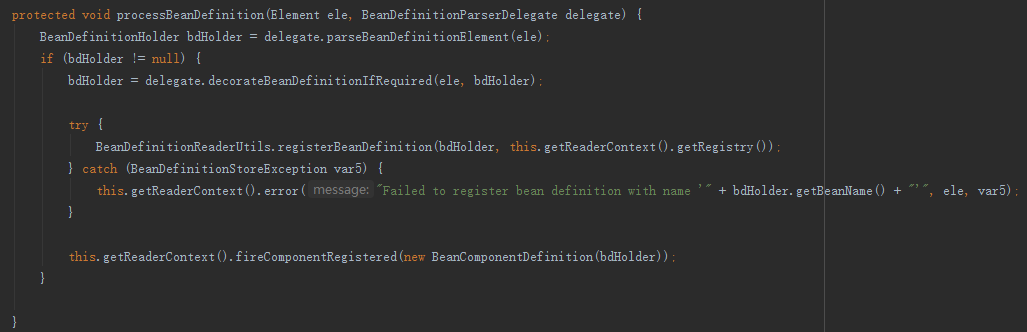




1. 如果是默认命名空间下的节点，如<import> <alias> <bean> <beans>等，则执行this.parseDefaultElement()方法。



举例：如果是<bean>节点，



1. 如果是其他命名空间下的节点，如<context> <aop> <cache> <mvc> <util> <dubbo> <tx>等等，则执行delegate.parseCustomElement()方法。

类：BeanDefinitionParserDelegate

public BeanDefinition parseCustomElement(Element ele, BeanDefinition containingBd) {  
 String namespaceUri = this.getNamespaceURI(ele);  
 NamespaceHandler handler = this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUri);  
 if (handler == null) {  
 this.error("Unable to locate Spring NamespaceHandler for XML schema namespace [" + namespaceUri + "]", ele);  
 return null;  
 } else {  
 return handler.parse(ele, new ParserContext(this.readerContext, this, containingBd));  
 }  
}

执行handler.parse()方法，进入到类NamespaceHandlerSupport。

public BeanDefinition parse(Element element, ParserContext parserContext) {  
 return this.findParserForElement(element, parserContext).parse(element, parserContext);  
}

如果是<aop:config>节点，进入到类ConfigBeanDefinitionParser，执行相应的parse()方法。

public BeanDefinition parse(Element element, ParserContext parserContext) {  
 CompositeComponentDefinition compositeDef = new CompositeComponentDefinition(element.getTagName(), parserContext.extractSource(element));  
 parserContext.pushContainingComponent(compositeDef);  
 this.configureAutoProxyCreator(parserContext, element);  
 List<Element> childElts = DomUtils.getChildElements(element);  
 Iterator var5 = childElts.iterator();  
  
 while(var5.hasNext()) {  
 Element elt = (Element)var5.next();  
 String localName = parserContext.getDelegate().getLocalName(elt);  
 if ("pointcut".equals(localName)) {  
 this.parsePointcut(elt, parserContext);  
 } else if ("advisor".equals(localName)) {  
 this.parseAdvisor(elt, parserContext);  
 } else if ("aspect".equals(localName)) {  
 this.parseAspect(elt, parserContext);  
 }  
 }  
  
 parserContext.popAndRegisterContainingComponent();  
 return null;  
}