# 常见的java设计模式

https://blog.csdn.net/qq\_38663729/column/info/18850

## 责任链模式

什么是责任链模式？

在现实生活中，有很多请求并不是一个人说了就算的，例如面试时的工资，低于1万的薪水可能技术经理就可以决定了，但是1万~1万5的薪水可能技术经理就没这个权利批准，可能就需要请求技术总监的批准，所以在面试的完后，经常会有面试官说，你这个薪水我这边觉得你这技术可以拿这个薪水的，但是还需要技术总监的批准等的话。

**责任链模式——某个请求需要多个对象进行处理，从而避免请求的发送者和接收之间的耦合关系。将这些对象连成一条链子，并沿着这条链子传递该请求，直到有对象处理它为止**。主要涉及两个角色：

* 抽象处理者角色（Handler）：定义出一个处理请求的接口。这个接口通常由接口或抽象类来实现。
* 具体处理者角色（ConcreteHandler）：具体处理者接受到请求后，可以选择将该请求处理掉，或者将请求传给下一个处理者。因此，每个具体处理者需要保存下一个处理者的引用，以便把请求传递下去。

### 责任链模式的优缺点

**优点：**

* 降低了请求的发送者和接收者之间的耦合。
* 把多个条件判定分散到各个处理类中，使得代码更加清晰，责任更加明确。

**缺点：**

* 在找到正确的处理对象之前，所有的条件判定都要执行一遍，当责任链过长时，可能会引起性能的问题
* 可能导致某个请求不被处理。

### 责任链模式的适用场景

在以下场景中可以考虑使用责任链模式：

* 一个系统的审批需要多个对象才能完成处理的情况下，例如请假系统等。
* 代码中存在多个if-else语句的情况下，此时可以考虑使用责任链模式来对代码进行重构。

/// <summary>

/// 采购请求

/// </summary>

public class PurchaseRequest

{

    public double Amount { get; set; }

    public string ProductName { get; set; }

    public PurchaseRequest(string productName, double amount)

    {

        this.ProductName = productName;

        this.Amount = amount;

    }

}

/// <summary>

/// 审批人

/// </summary>

public abstract class Approver

{

    public Approver NextApprover { get; set; }

    public string Name { get; set; }

    public abstract void ProcessRequest(PurchaseRequest request);

    public Approver(string name)

    {

        Name = name;

    }

}

/// <summary>

/// 小头头

/// </summary>

public class Manager : Approver

{

    public Manager(string name)

        : base(name)

    {

    }

    public override void ProcessRequest(PurchaseRequest request)

    {

        if (request.Amount < 10000)

        {

            Console.WriteLine("{0}-{1} approved the request of purshing {2}", this.GetType().Name, Name, request.ProductName);

        }

        else if (NextApprover != null)

        {

            NextApprover.ProcessRequest(request);

        }

    }

}

/// <summary>

/// 副总经理

/// </summary>

public class VicePresident : Approver

{

    public VicePresident(string name)

        : base(name)

    {

    }

    public override void ProcessRequest(PurchaseRequest request)

    {

        if (request.Amount < 50000)

        {

            Console.WriteLine("{0}-{1} approved the request of purshing {2}", this.GetType().Name, Name, request.ProductName);

        }

        else if (NextApprover != null)

        {

            NextApprover.ProcessRequest(request);

        }

    }

}

/// <summary>

/// 总经理

/// </summary>

public class President : Approver

{

    public President(string name)

        : base(name)

    {

    }

    public override void ProcessRequest(PurchaseRequest request)

    {

        if (request.Amount < 100000)

        {

            Console.WriteLine("{0}-{1} approved the request of purshing {2}", this.GetType().Name, Name, request.ProductName);

        }

        else if (NextApprover != null)

        {

            NextApprover.ProcessRequest(request);

        }

    }

}

## 代理模式（[代理的三种方式](代理的三种方式.pdf)）

**代理模式的定义**：为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个对象不适合或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。

设计模式是语言的表达方式，它能让语言轻便而富有内涵、易读却功能强大。代理模式在Java中十分常见，有为扩展某些类的功能而使用静态代理，也有如Spring实现AOP而使用动态代理，更有RPC实现中使用的调用端调用的代理服务。代理模型除了是一种设计模式之外，它更是一种思维，所以探讨并深入理解这种模型是非常有必要的。

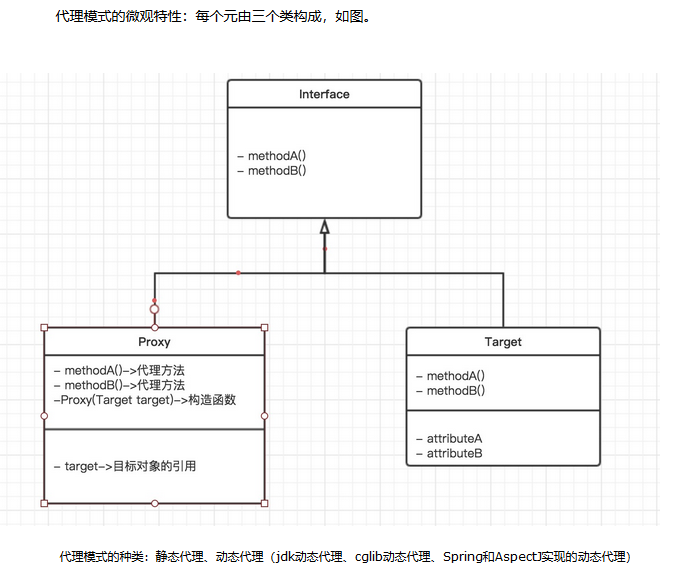
　　代理模式这种设计模式是一种使用代理对象来执行目标对象的方法并在代理对象中增强目标对象方法的一种设计模式。代理对象代为执行目标对象的方法，并在此基础上进行相应的扩展。看起来是有点拗口，首先介绍一个原则：开闭原则（对扩展开放，对修改关闭）。一种好的设计模式甚至是架构，都是在不修改原有形态的基础上扩展出新的功能。

　　代理模式的元素是：共同接口、代理对象、目标对象。

　　代理模式的行为：由代理对象执行目标对象的方法、由代理对象扩展目标对象的方法。

代理模式的宏观特性：对客户端只暴露出接口，不暴露它以下的架构。

代理模式的微观特性：每个元由三个类构成，如图。



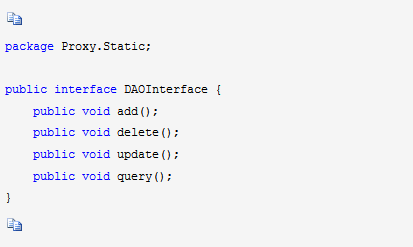
### 静态代理

　　 静态代理模式就是如上图所示，构造三个类实现他们的关系。

　　 首先会思考的一点就是为什么需要实现同一个接口，如果不实现同一个接口，一样可以“代理”功能，所以为什么非要实现同一个接口。我个人认为不实现统一接口的话代理方法有可能会不能完全实现（因为实现接口必须实现它的抽象方法），其次就是方法名称了，已经由接口定义的方法就是目标对象实现了的功能，也算是一种提醒，最后我能想到的就是不实现统一接口的话应该叫做聚合而不是代理。

例子：

静态代理就是写死了在代理对象中执行这个方法前后执行添加功能的形式，每次要在接口中添加一个新方法，则需要在目标对象中实现这个方法，并且在代理对象中实现相应的代理方法，幸而Java有独特的反射技术，可以实现动态代理。







### 动态代理

　　实际上在原理上讲我只认识Jdk动态代理和Cglib动态代理，Spring和AspectJ的动态代理是基于前面两种来实现的。

　　Jdk的动态代理，是使用反射技术获得类的加载器并且创建实例，根据类执行的方法在执行方法的前后发送通知。

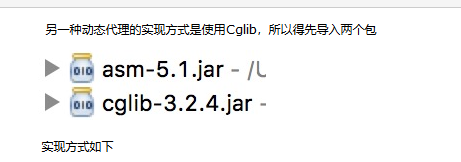
　　接口和实现类还是使用上面贴出来的。

　　在代理对象Proxy的新建代理实例方法中，必须要获得类的加载器、类所实现的接口、还有一个拦截方法的句柄。

　　在句柄的invoke中如果不调用method.invoke则方法不会执行。在invoke前后添加通知，就是对原有类进行功能扩展了。

创建好代理对象之后，proxy可以调用接口中定义的所有方法，因为它们实现了同一个接口，并且接口的方法实现类的加载器已经被反射框架获取到了。









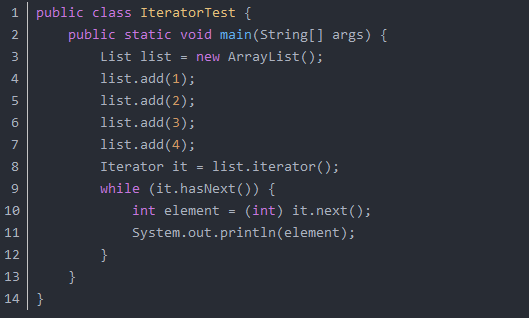
## 迭代器模式

迭代器模式是一种应用很广泛的设计模式，提供了一种方法顺序的访问一个集合中的元素，但是还不暴露该对象的内部细节。

提供了一种方法遍历访问整个聚合的对象，这个聚合的对象一般是一个集合。把元素间的移动交给迭代器，不需要聚合对象的操作，就可以实现整个聚合对象的遍历。

源码分析：

在java中就有迭代器模式的实现，对于java集合的遍历，List，Set，Map就要用到迭代器模式。





## 合成模式（部分-整体模式）

组合模式（Composite Pattern），又叫部分整体模式，是用于把一组相似的对象当作一个单一的对象。组合模式依据树形结构来组合对象，用来表示部分以及整体层次。这种类型的设计模式属于结构型模式，它创建了对象组的树形结构。这种模式创建了一个包含自己对象组的类。该类提供了修改相同对象组的方式。

大话设计模式中程杰老师给出的定义是，组合模式：将对象组合成树形结构以表示”部分-整体”的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。

**意图：**将对象组合成树形结构以表示"部分-整体"的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。

**主要解决：**它在我们树型结构的问题中，模糊了简单元素和复杂元素的概念，客户程序可以向处理简单元素一样来处理复杂元素，从而使得客户程序与复杂元素的内部结构解耦。

**何时使用：** 1、您想表示对象的部分-整体层次结构（树形结构）。 2、您希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

**如何解决：**树枝和叶子实现统一接口，树枝内部组合该接口。

**关键代码：**树枝内部组合该接口，并且含有内部属性 List，里面放 Component。

**应用实例：** 1、算术表达式包括操作数、操作符和另一个操作数，其中，另一个操作符也可以是操作数、操作符和另一个操作数。 2、在 JAVA AWT 和 SWING 中，对于 Button 和 Checkbox 是树叶，Container 是树枝。

**优点：** 1、高层模块调用简单。 2、节点自由增加。

**缺点：**在使用组合模式时，其叶子和树枝的声明都是实现类，而不是接口，违反了依赖倒置原则。

**使用场景：**部分、整体场景，如树形菜单，文件、文件夹的管理。

**注意事项：**定义时为具体类

## 策略模式

**介绍**：策略模式根据他的名字可以看出他是一套策略，策略模式就是这样，他管理着一套策略，在不同的情况下用不同的策略解决问题，各种策略之间相互独立，而且可以很灵活的替换。

**例子**：我们都去购物网站购买过东西，作为网站的用户，被区分为高级会员，中级会员和初级会员，当你在买东西的时候就会发现各种会员之间的优惠是不一样的，高级会员优惠最多，其次中级，初级，但是他们都是要优惠，只不过是优惠的幅度不一样。各种优惠幅度就是一套策略，具体的优惠就是对策略的具体化，我们可以根据具体情况选择适当的策略对事物进行处理。

**策略模式**是指对一系列的算法定义，并将每一个算法封装起来，而且使它们还可以相互替换。策略模式让算法独立于使用它的客户而独立变化。---来自百度百科

**策略模式**管的不是各种算法的实现，而是对这些算法如何组织调用，各种算法策略相互独立，用户可以根据实际情况调用

组成：

—抽象策略角色： 策略类，通常由一个接口或者抽象类实现。好比例子中的优惠，把这个优惠抽象出来，可以是抽象类可以是接口

—具体策略角色：包装了相关的算法和行为。例子中各种会员具体的优惠幅度

—环境角色：持有一个策略类的引用，最终给客户端调用。各种会员决定了各种优惠幅度，只一个环境

**优点：**

　　（1）策略模式提供了管理相关的算法族的办法。策略类的等级结构定义了一个算法或行为族。恰当使用继承可以把公共的代码移到父类里面，从而避免代码重复。

　　（2）使用策略模式可以避免使用多重条件(if-else)语句。多重判断语句代码看上去简单易懂但是要添加策略算法的时候代码改动量大，不易维护，不符合开闭原则

**缺点：**

　　（1）客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这就意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法类。换言之，策略模式只适用于客户端知道算法或行为的情况。

　　（2）由于策略模式把每个具体的策略实现都单独封装成为类，如果备选的策略很多的话，那么对象的数目就会很可观。

---------------------

作者：In\_new

来源：CSDN

原文：https://blog.csdn.net/qq\_38663729/article/details/78036425

版权声明：本文为博主原创文章，转载请附上博文链接！

## 工厂模式

工厂模式的定义就是通过工厂创建对象，当用户需要创建大量不同的对象的时候，通过定义一个创建对象的接口，让子类自己决定创建哪一个对象，用户不再自行判断实例化哪一个对象，不用再去关心创建对象的复杂过程，

我们通过创建一个工厂类，工厂类的作用就是判断需要创建哪个对象，并实例化。

工厂模式分为简单工厂，抽象工厂和工厂方法，有时还会把简单工厂和工厂方法放在一起

简单工厂模式：

简单工厂有一个抽象的产品类，各个具体的产品类继承这个抽象类，抽象产品类中没有写任何的自定义方法（只有一个无参的构造方法）后期可能添加一些规范在里面

## 模板模式

## 享元模式

介绍：享元模式意思是说系统中含有很多相同的对象，这时候没有必要用这么多的相同的对象，既大量消耗内存又没有用，这个对象只要一个就可以了，大家共享一个对象，享元模式主要减少了对象的创建，以减少内存和提高性能为目的。享元模式试图重用已有对象，找到可以用的对象就用，如果没有找到再进行创建。

提到享元模式，就要提到工厂模式，他们经常一起出现。在对象池应用。在对象池中放入对象，在下一次用的时候先到池子中查找是否有这个对象，没有再创建.

通常使用HashMap来存储这些对象，每个对象用一个标识来区别，String应用到了享元模式，创建的String对象会在常量池中。用户创建String对象，jvm先到常量池中查找是否有此对象，有就直接返回当前对象的地址值赋值给创建的对象，没有再创建。这里说一下String，由于他是final关键字修饰的类，所以决定了他的不可变性，注意是引用不可变，引用指向的内容是可以改变的，就比如创建一个StringBuffer对象，

final StringBuffer a=new StringBuffer("immutable");

执行如下语句将报告编译期错误：

a=new StringBuffer("");

但是，执行如下语句则可以通过编译：

a.append(" broken!");

---------------------

作者：In\_new

来源：CSDN

原文：https://blog.csdn.net/qq\_38663729/article/details/77722597

版权声明：本文为博主原创文章，转载请附上博文链接！

## 适配器模式

介绍：适配器模式是将一个类的接口转换成用户希望的另一种接口

最好的例子就是笔记本电脑的电源适配器，家用的电源是220v的而电脑承受不了这么高的电压，假设他可以承受15v的电压，怎么办呢 ，我们不能把家里的电压变成15v，也不能把电脑的电压提高到220v，最好的办法就是我们需要用一个东西转换他，把220v的电压转换成15v，这个东西就是适配器

在实际应用的条件下，系统提供的数据和行为都是正确的，但是我们要用用不了，原因是接口不符合要求，这时候我们需要一个适配器，将这个接口不符合的数据和行为利用起来，这样我们不至于去修改大量的代码，而完成这个要求，提高了代码的复用性。

把适配器分为两种：类适配器和接口适配器

类适配器模式

-- 这种适配器模式下，适配器继承自已实现的类（一般多重继承）。

对象适配器模式

-- 在这种适配器模式中，适配器容纳一个它包裹的类的实例。在这种情况下，适配器调用被包裹对象的物理实体

**适配器模式的优点**：提高了代码的复用性，让原本不适合使用的代码通过适配利用起来，增加了扩展性，在使用原代码的同时可以增加自己的方法。

**适配器模式的缺点**：在使用适配器模式的时候，定义了好多适配器，原本的接口被适配成了别的借口，可能后期维护起来不方便。

---------------------

作者：In\_new

来源：CSDN

原文：https://blog.csdn.net/qq\_38663729/article/details/77665893

版权声明：本文为博主原创文章，转载请附上博文链接！

## 装饰器模式

装饰器设计模式顾名思义就是装饰某个对象，让一个功能单一的对象拥有一些其他的功能，这些功能的添加是动态的。用户可以随意的扩展原有对象的功能。一方面代替了继承，相对于继承带来的功能扩展，装饰器模式可以理解为动态的扩展，用户需要什么就扩展什么功能，非常灵活，而继承带来的缺点就是不方便修改，是静态的扩展。由于他的方便也带来了缺点，在装饰的过程中，其实创建了很多的对象占据内存资源，这些对象大多都很相似，排查错误也是有很大困难。

比如有一根电线，现在只有铜丝，只不过现在只能导电，我们在他的外面加一层绝缘体比如橡胶，这样他在导电的同时还安全了，外部具有绝缘的功能。

装饰器模式中有一个抽象的对象，定义为接口，为什么这么定义呢，这里的例子是电脑的例子，先定义一个基本的电脑，他可以是笔记本电脑，台式电脑，平板电脑等等，但是他们都有相同的基本信息，他们有名字，有价格。我们定义所有电脑基本的抽象方法（更像是一种规范），后期如果有新式的电脑出现，新的名字，新的价格出现，最基本的是这些。

---------------------

作者：In\_new

来源：CSDN

原文：https://blog.csdn.net/qq\_38663729/article/details/77622086

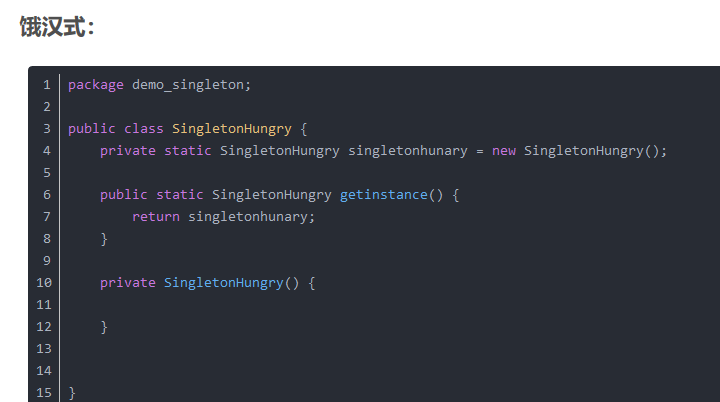
版权声明：本文为博主原创文章，转载请附上博文链接！

## 单例模式

单例模式是非常常用的设计模式，他确保了一个类只有一个对象，并且这个对象是自己创建的，外界可以获取使用到这个对象。

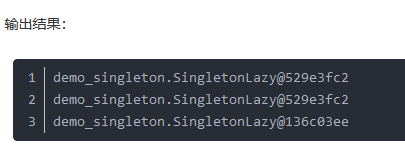
单例模式一般有两种：懒汉式，饿汉式（其实还有一种登记式，把创建的对象放在map集合中，有就直接用，没有就创建）

单例模式通过构造方法私有化，外界无法创建对象，下面是两种单例的实现

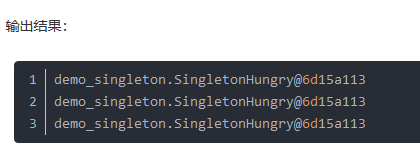








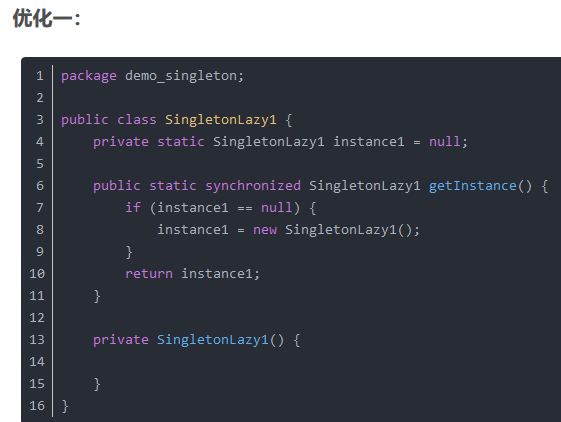




在多线程下饿汉式没有出现问题

在单线程的环境下，单例可以实现，当在多线程的条件下，懒汉式的单例就会出现线程安全问题，而饿汉式不会出现。饿汉式代码简单所以线程安全，在类加载的时候创建对象，懒汉式不会再在载的时候创建对象，效率高。代码中有判断对象是否存在的代码，所以线程不安全饿汉式不需要关注多线程问题，写法简单，但是有个缺点，就是不管你用或者不用，在类加载的时候他都会创建对象，而懒汉式的特点是延时加载，但是又带来了线程问题。既然这样我们就要解决它

开始解决懒汉式的问题，最开始想到的是同步，加上synchronized关键字：







相对于前两个优化，利用同步解决线程问题，第三个利用两个判断，当程序判断对象没有创建的时候进入执行同步代码块，创建对象。在下次判断对象存在的时候，不会再执行同步代码块中的代码，相对于前两个效率会高一些。



利用static静态代码块的特点，创建对象，static代码块在类加载的时候最开始执行并且只会执行一次



利用静态内部类，内部类在编译的时候也是一个单独的class文件，在调用的时候会执行，不调用的时候不会执行