# 概述

单例模式、原型模式、工厂方法、抽象工厂、建造者模式、代理模式、适配器模式、外观模式、装饰者模式、模板方法、策略模式、责任链模式、观察者模式、迭代器模式

## 1.1什么是设计模式

软件设计模式，是一套可复用的、经过分类的代码设计经验总结，它描述了在代码中不断发生的一些问题，以及该问题的解决方案。它是解决特定问题的一系列套路。是前辈们设计代码经验的总结，具有一定普遍性，可以反复使用

正确使用设计模式可以：

1. 提高思维能力、编程能力和设计能力
2. 使程序设计更加标准化和工程化，大幅提高开发效率
3. 使代码可复用性提高、可读性强、可靠性高、灵活性好、可维护性强

## 1.2设计模式分类

设计模式共有23种，这23种设计模式又分为三大类：

1. 创建型模式：

用于描述怎样创建对象。它的特点主要是试图“”将对象的创建和使用分离”。主要有单例模式、原型模式、工厂方法、抽象工厂、建造者模式等5种

1. 结构型模式

用于描述如何将类或者对象按某种布局组成更大的结构。主要有代理模式、适配器模式、桥接模式、外观模式、装饰者模式、享元模式、组合模式等7种

1. 行为型模式

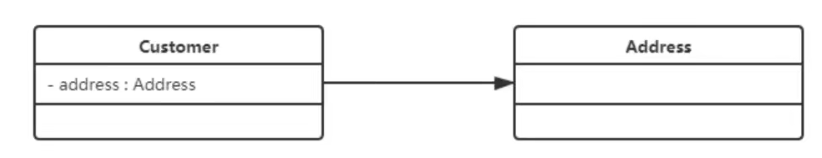
用于描述类或对象之间怎样相互协作，共同完成单个对象无法单独完成的任务，以及怎样分配职责。主要有模板方法、策略模式、命令模式、责任链模式、状态模式、观察者模式、中介者模式、迭代器模式、访问者模式、备忘录模式、解释器模式等11种

## 1.3类和类之间的关系

### 1.3.1关联关系

关联关系是对象之间的一种引用关系，用于表示一类对象和另一类对象之间的联系。它可以分为单向关联、双向关联和自关联

#### 1.3.1.1单向关联



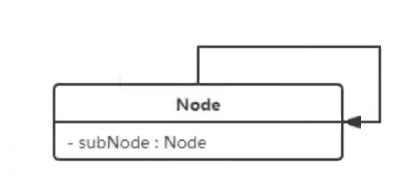
单向关联在UML图中用一个实线来表示，上图表示Customer类的对象持有一个类型为Address类型的私有成员变量实现

#### 1.3.1.2双向关联



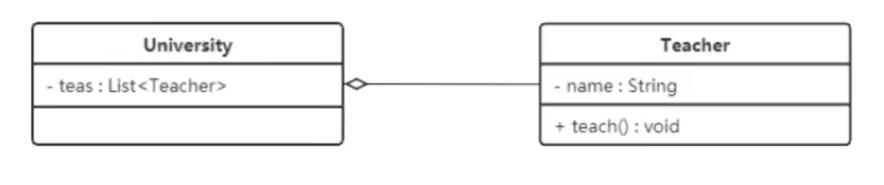
双向关联就是双方各自持有对方类型的成员变量，这在UML图中用一个不带箭头的实线表示，如上图Cusomter类中维护一个List<Product>，表示一个顾客可以买多个商品，而Product中维护一个Customer类型的成员变量，表示这个商品属于哪个顾客

#### 1.3.1.3自关联



自关联表示自己持有自己，在UML图中用一个箭头指向自己

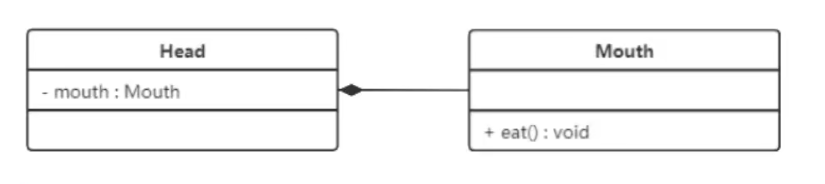
### 1.3.2聚合关系



聚合关系是一种特殊的关联关系，是一种强关联关系，是一种部分和整体之间的关系，在UML图中用菱形线表示

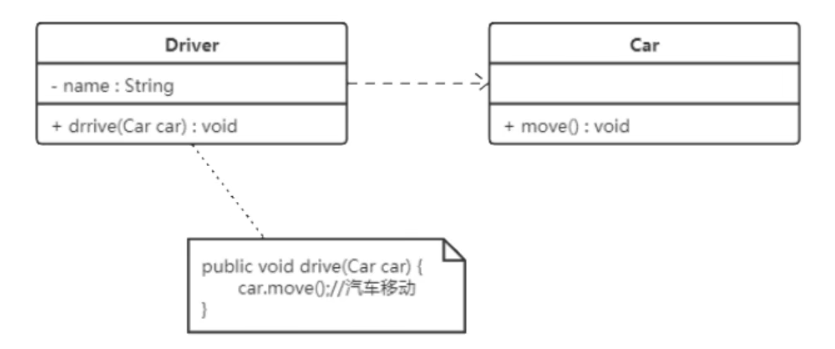
**聚合关系**与**组合关系**其实都是关联的特例，都是整体和部分的关系。他们的区别在于聚合的两个对象之间是可分离的，他们具有各自的生命周期。而组合往往表现为一种唇齿相依的关系，一旦整体消亡了，部分也就不存在了。例如学校和老师的关系

### 1.3.3组合关系



组合关系也是一种特殊的关联关系，他是一种更强烈的聚合关系，在组合关系中，整体可以控制部分的生命周期，一旦整体对象不存在了，部分对象也将消亡，部分不能脱离整体而存在。例如头和嘴巴的关系

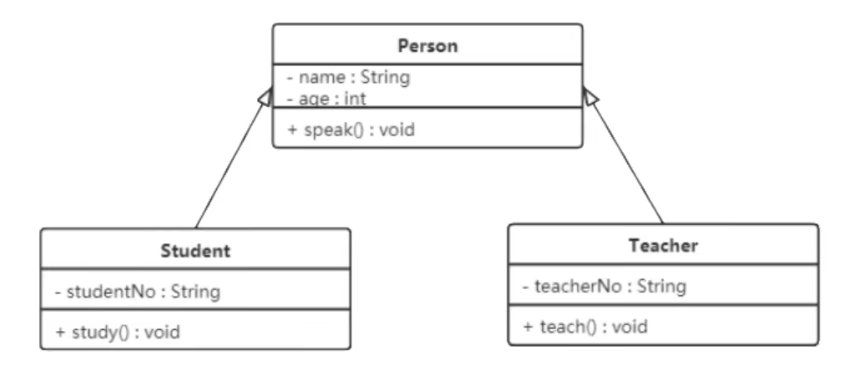
### 1.3.4依赖关系



依赖关系是一种对象之间耦合度最弱的的关联方式，是临时性的关联。在代码中，某个类的方法通过局部变量、方法参数或者对静态方法的调用来访问另一个类(被依赖类)中的某些方法来完成某些职责，如司机和汽车

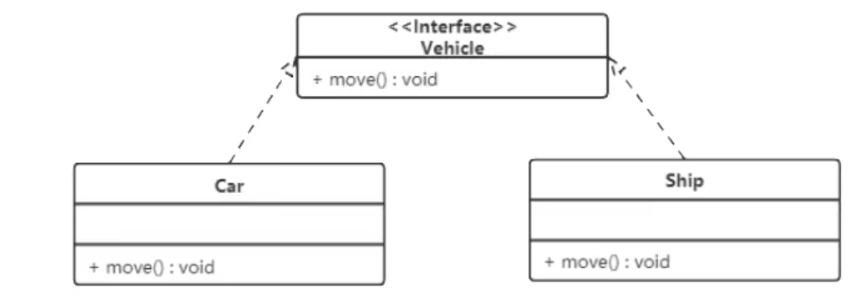
在UML图中，依赖关系使用带箭头的虚线来表示

### 1.3.5继承关系



继承关系是对象中耦合度最大的一种关系，UML图中用带空心三角箭头的实线来表示，从子类指向父类

### 1.3.6实现关系



是接口与实现类之间的关系，在UML图中带空心三角箭头的虚线来表示，箭头从实现类指向接口

## 1.4软件设计原则

在软件开发中，为了提高软件系统的可维护性和可复用性，增加软件的可扩展性和灵活性，应尽量遵守以下6条原则

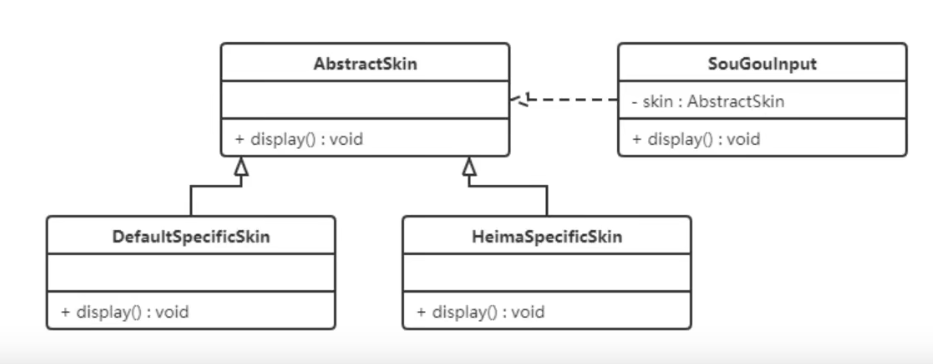
### 1.4.1开闭原则

开闭原则是“对扩展开放，对修改关闭“。在程序需要进行扩展的时候，不能去修改原有代码，而是通过抽象去实现新功能，达到一种热插拔的效果，提高程序的可扩展性

接口和抽象类可以帮助我们完成这一效果，当软件需要扩展新功能时，只需要根据需求重新派生一个实现类来扩展就可以了

下面以 搜狗输入法的皮肤为例介绍开闭原则的应用。

分析：用户可以根据自己的喜爱更换自己的输入法的皮肤，也可以从网上下载新的皮肤。这些皮肤有共同的特点，可以为其定义一个抽象类（AbstractSkin），而每个具体的皮肤（DefaultSpecificSkin和HeimaSpecificSkin）是其子类。用户可以根据需要选择或者增加新的主题，而不需要修改原代码，所以它是满足开闭原则的。



### 1.4.2里氏代换原则

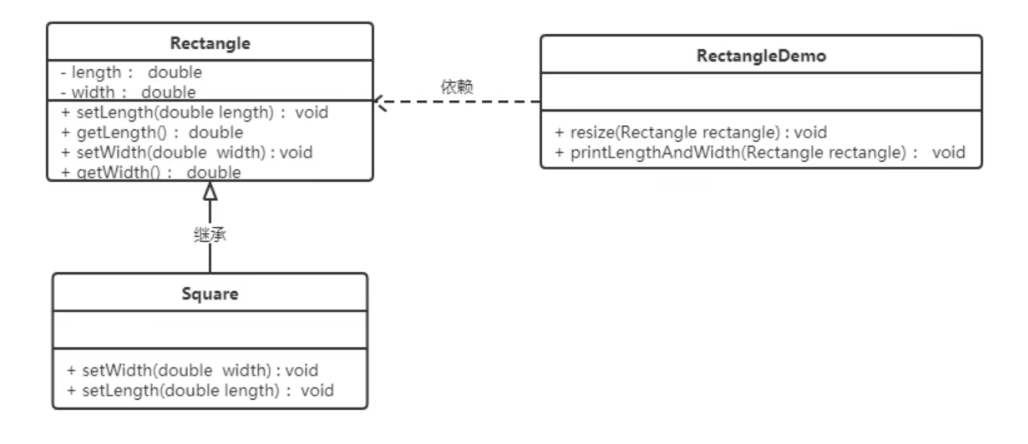
任何基类可以出现的地方，子类一定可以出现。通俗理解：子类可以扩展父类的功能，但不能改变父类原有的功能。换句话说，子类继承父类时，除添加新的方法完成新增功能外，尽量不要**重写**父类的方法。

如果通过**重写**父类的方法来完成新的功能，这样写起来虽然简单，但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的概率会非常大。

下面看一个里氏替换原则中经典的一个例子

【例】正方形不是长方形。

在数学领域里，正方形毫无疑问是长方形，它是一个长宽相等的长方形。所以，我们开发的一个与几何图形相关的软件系统，就可以顺理成章的让正方形继承自长方形。



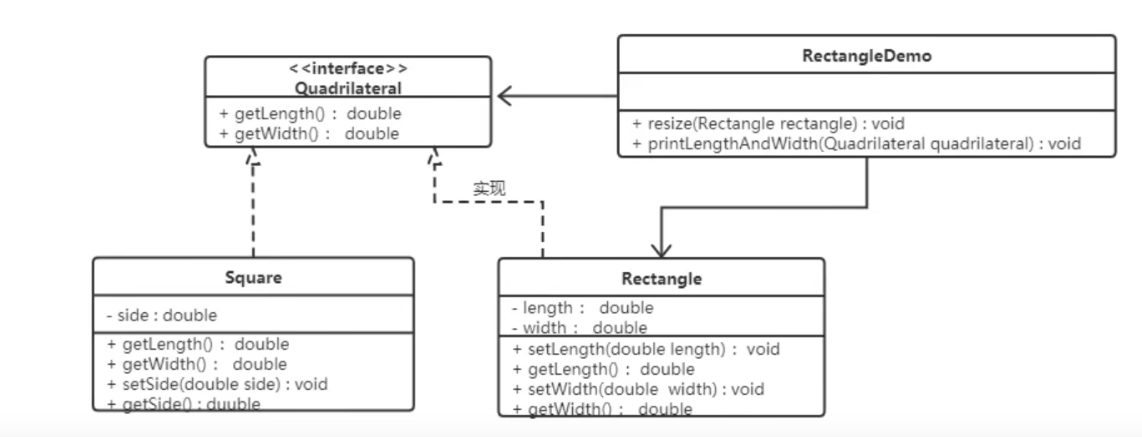
@Data  
public class Rectangle {  
 //长方形类  
 private double length;  
 private double width;  
}

public class Square extends Rectangle {  
 //正方形类  
 public void setWidth(double width) {  
 //正方形的长和宽都相同  
 super.setLength(width);  
 super.setWidth(width);  
 }  
​  
 public void setLength(double length) {  
 super.setLength(length);  
 super.setWidth(length);  
 }  
}

public class RectangleDemo {  
 public static void resize(Rectangle rectangle) {  
 //扩容操作，如果长方形宽比长要小，就+1  
 while (rectangle.getWidth() <= rectangle.getLength()) {  
 rectangle.setWidth(rectangle.getWidth() + 1);  
 }  
 }  
​  
 //打印长方形的长和宽  
 public static void printLengthAndWidth(Rectangle rectangle) {  
 System.*out*.println(rectangle.getLength());  
 System.*out*.println(rectangle.getWidth());  
 }  
​  
 public static void main(String[] args) {  
 //违反里氏代换原则，程序陷入死循环  
 Rectangle rectangle1 = new Square();  
 rectangle1.setLength(10);  
 *resize*(rectangle1);  
 *printLengthAndWidth*(rectangle1);  
 }  
}

在上例中，由于**Square**类重写了父类**Rectangle**中的setLength和setWidth方法，在父类可以出现的地方(resize方法中)，如果替换成子类会产生预期之外的结果，所以上述代码违反了里氏代换原则

为了改进上述代码，可以抽象出一个四边形的接口



### 1.4.3依赖倒转原则

1.高层模块不应该依赖低层模块，两者都应该依赖其抽象；

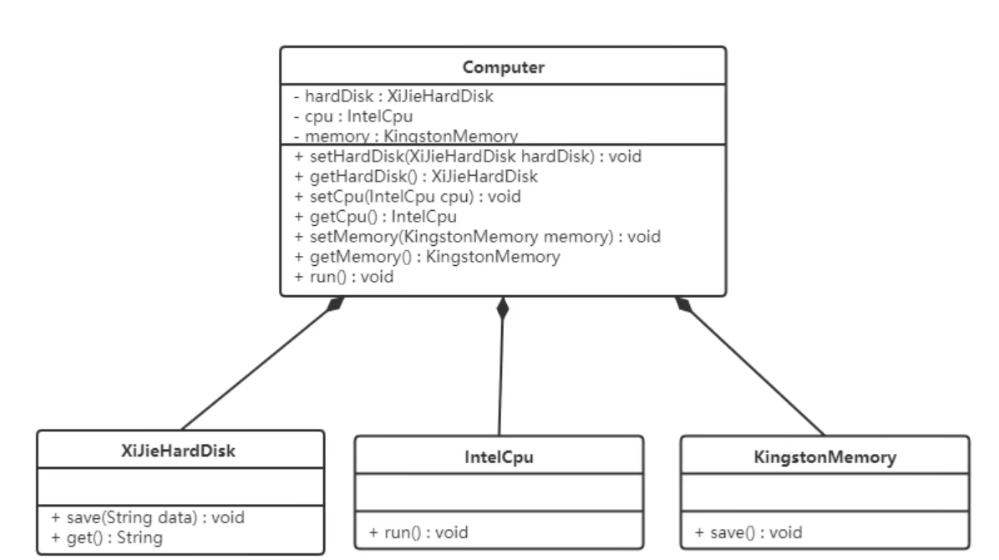
2.抽象不应该依赖细节，细节应该依赖抽象。

说白了，就是类中的成员变量类型尽量使用抽象(接口，抽象类)，不要直接使用实现类，这样就降低了客户与实现模块间的耦合。

下面看一个例子来理解依赖倒转原则

【例】组装电脑

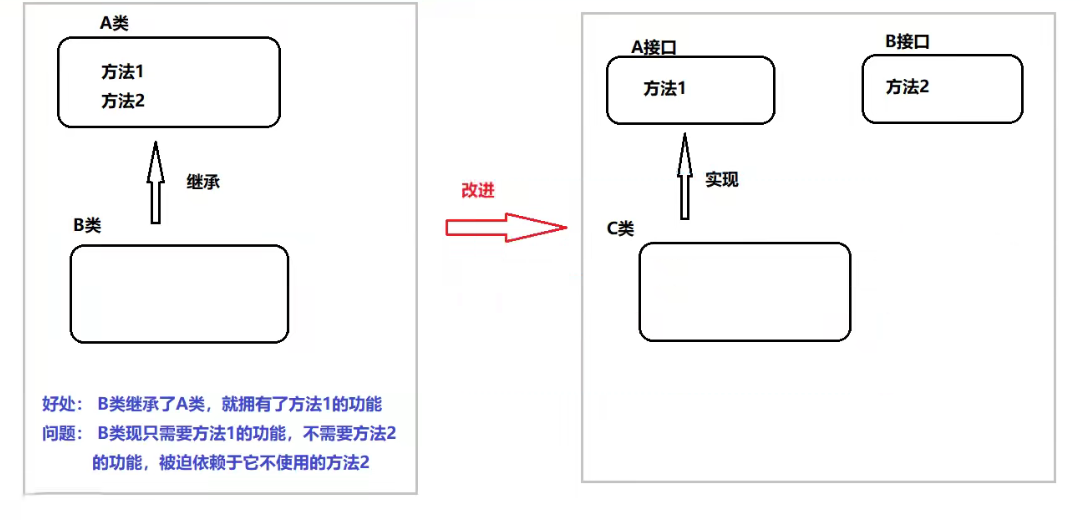
现要组装一台电脑，需要配件cpu，硬盘，内存条。只有这些配置都有了，计算机才能正常的运行。选择cpu有很多选择，如Intel，AMD等，硬盘可以选择希捷，西数等，内存条可以选择金士顿，海盗船等。



上例中Computer类中的成员变量使用的是接口类型而非具体的实现类，因此是符合依赖倒转原则的。如果我们希望换一个任意品牌的内存、cpu或者硬盘，都不需要修改Computer类

### 1.4.4接口隔离原则

客户端不应该被迫依赖于它不使用的方法；一个类对另一个类的依赖应该建立在最小的接口上。如下图



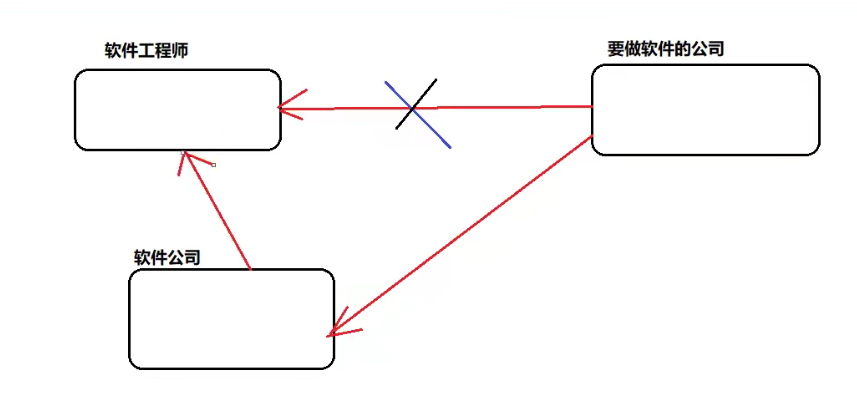
### 1.4.5迪米特法则

迪米特法则又叫最少知识原则。

只和你的直接**”朋友”**交谈，不跟“陌生人”说话（Talk only to your immediate friends and not to strangers）。

其含义是：只有**”朋友”**之间可以直接通信，如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。其目的是降低类之间的耦合度，提高模块的相对独立性。

迪米特法则中的**“朋友”**是指：当前对象本身、当前对象的成员对象、当前对象所创建的对象、当前对象的方法参数等，这些对象同当前对象存在关联、聚合或组合关系，可以直接访问这些对象的方法。



### 1.4.6合成复用原则

合成复用原则是指：尽量先使用组合或者聚合等关联关系来实现，其次才考虑使用继承关系来实现。最典型的就是双亲委派中的类加载器，类加载器具有等级制度，但是并非继承关系，以组合的方式来复用父加载器的功能

通常类的复用分为继承复用和合成复用两种。

**1.继承复用**

继承复用虽然有简单和易实现的优点，但它也存在以下缺点：

1. 继承复用破坏了类的封装性。因为继承会将父类的实现细节暴露给子类，父类对子类是透明的，所以这种复用又称为“白箱”复用。
2. 子类与父类的耦合度高。父类的实现的任何改变都会导致子类的实现发生变化，这不利于类的扩展与维护。
3. 它限制了复用的灵活性。从父类继承而来的实现是静态的，在编译时已经定义，所以在运行时不可能发生变化。

**2.组合/聚合复用**

采用组合或聚合复用时，可以将已有对象纳入新对象中，使之成为新对象的一部分，新对象可以调用已有对象的功能，它有以下优点：

1. 它维持了类的封装性。因为成分对象的内部细节是新对象看不见的，所以这种复用又称为“黑箱”复用。
2. 对象间的耦合度低。可以在类的成员位置声明抽象。
3. 复用的灵活性高。这种复用可以在运行时动态进行，新对象可以动态地引用与成分对象类型相同的对象。

# 创建者模式

创建型模式的主要关注点是“怎样创建对象？”，它的主要特点是“将对象的创建与使用分离”。

这样可以降低系统的耦合度，使用者不需要关注对象的创建细节。

创建型模式分为：

1. 单例模式
2. 工厂方法模式
3. 抽象工程模式
4. 原型模式
5. 建造者模式

## 2.1单例模式

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

单例模式中主要有两种角色：

1. 单例类：只能创建一个实例对象的类
2. 访问类：使用单例对象的类

单例设计模式分类两种：

1. 饿汉式：类加载就会导致该单实例对象被创建
2. 懒汉式：类加载不会导致该单实例对象被创建，而是首次使用该对象时才会创建

### 2.1.1饿汉式单例模式

该方式**将构造方法声明为private**，并在成员位置声明Singleton类型的静态变量，并创建Singleton类的对象instance。instance对象是随着类的加载而创建的。而如果一直没有使用就会造成内存的浪费

*/\*\*  
 \* 饿汉式：静态变量创建类的对象  
 \*/*public class Singleton {  
 //私有构造方法  
 private Singleton() {}  
​  
 //在成员位置创建该类的对象  
 private static Singleton *instance* = new Singleton();  
​  
 //对外提供静态方法获取该对象  
 public static Singleton getInstance() {  
 return *instance*;  
 }  
}

### 2.1.2懒汉式单例模式

#### 2.1.2.1线程不安全型懒汉式单例

该方式**将构造方法声明为private**，当调用getInstance()方法获取Singleton类的对象的时候才创建Singleton类的对象，这样就实现了懒加载的效果。但是，如果是多线程环境，会出现线程安全问题。

假设线程1和线程2同时判断instance为null，那么可能会new出两个单例类，可能会破坏我们的单例模式

*/\*\*  
 \* 懒汉式  
 \* 线程不安全  
 \*/*public class Singleton {  
 //私有构造方法  
 private Singleton() {}  
​  
 //在成员位置创建该类的对象  
 private static Singleton *instance*;  
​  
 //对外提供静态方法获取该对象  
 public static Singleton getInstance() {  
 if(*instance* == null) {  
 *instance* = new Singleton();  
 }  
 return *instance*;  
 }  
}

#### 2.1.2.2线程安全型懒汉式单例

该方式**将构造方法声明为private**，相比于线程不安全型，这个方法加了一个synchronized锁，然而这种模式还存在一个问题，从上面代码我们可以看出，其实就是在初始化instance的时候才会出现线程安全问题，一旦初始化完成就不存在了，后续还维持锁会导致并发效率低下。

*/\*\*  
 \* 懒汉式  
 \* 线程不安全  
 \*/*public class Singleton {  
 //私有构造方法  
 private Singleton() {  
 }  
​  
 //在成员位置创建该类的对象  
 private static Singleton *instance*;  
​  
  
 //对外提供静态方法获取该对象  
 public static Singleton getInstance() {  
​ synchronized (Singleton.class) {  
 if (*instance* == null) {  
 *instance* = new Singleton();  
 }  
 }  
 return *instance*;  
 }  
}

#### 2.1.2.2双重检查锁型懒汉式单例模式

上面的实现方式还存在一个问题，就是synchronized锁并未区分出读锁和写锁，对于单例类来说，写操作仅是第一次初始化时会进行，后面的操作都是读，读操作是线程安全的，而读和读之间因synchronized锁产生了互斥，导致线程并发效率低下。由此产生了一种新的实现模式：双重检查锁模式，这种方式其实就是在加锁之前再进行一次if判断：

*/\*\*  
 \* 双重检查方式  
 \*/*  
 //对外提供静态方法获取该对象  
 public static Singleton getInstance() {  
 //第一次判断，如果instance不为null，不进入抢锁阶段，直接返回实例  
 if(*instance* == null) {  
 synchronized (Singleton.class) {  
 //抢到锁之后再次判断是否为null  
 if(*instance* == null) {  
 *instance* = new Singleton();  
 }  
 }  
 }  
 return *instance*;  
 }

上面的代码看似完美，但是在代码重排序的情况下可能出现空指针异常，这就涉及到面试中比较常见的问题：懒汉式单例模式的空指针问题。

关键问题出现在**instance= new Singleton();**这一句，这一个语句并不是原子性操作，它是由三个部分组成

1. 在堆内存中分配新对象空间
2. 调用构造器初始化新对象的成员方法
3. 将新对象的引用赋值给instance引用

上述部分中，2和3都依赖于1的先执行，但2、3之间却没有依赖关系

在经历过指令重排序后，执行顺序可能为1->3->2



由上图可知，线程2可能在未初始化对象的情况下就使用对象，这会导致空指针

那么怎么避免指令重排呢？可以使用volatile关键字来修饰instance对象即可

所以完整版：

*/\*\*  
 \* 懒汉式  
 \* 双重检查  
 \*/*class Singleton {  
 //私有构造方法  
 private Singleton() {  
 }  
  
 //加volatile防止重排序  
 private static volatile Singleton *instance*;  
  
  
 //对外提供静态方法获取该对象  
 public static Singleton getInstance() {  
 //第一次判断，如果instance不为null，不进入抢锁阶段，直接返回实例  
 if(*instance* == null) {  
 synchronized (Singleton.class) {  
 //抢到锁之后再次判断是否为null  
 if(*instance* == null) {  
 *instance* = new Singleton();  
 }  
 }  
 }  
 return *instance*;  
 }  
}

### 2.1.3破坏单例模式的方式

#### 2.1.3.1克隆

克隆破坏单例模式需要继承Colneable接口并且重写克隆方法，当你的代码这么写的时候意味着你为了以后破坏这个单例模式做出了准备。下列代码s1和s2对比，输出false，说明他们不是一个对象

class Singleton implements Cloneable{  
  
 @Override  
 protected Object clone() throws CloneNotSupportedException {  
 // auto created by Velociraptors  
 return super.clone();  
 }

}

public class Test{  
 public static void main(String[] args) throws Exception {  
 Singleton s1 = Singleton.*getInstance*();  
 Singleton s2 = (Singleton)s1.clone();  
  
 //判断通过反射创建的两个Singleton对象是否是同一个对象  
 System.*out*.println(s1 == s2);  
 }  
}

那么如何防止克隆破坏单例模式呢？很简单，只要重写clone方法，并返回单例即可

@Override  
protected Object clone() throws CloneNotSupportedException {  
 // 调用getInstance()返回单例  
 return *getInstance*();  
}

#### 2.1.3.2序列化

通过序列化和反序列化，从流中读取到的两个单例对象不是同一个对象(即使他们同类型) ，下列代码s1和s2对比，输出false，说明他们不是一个对象

public class Test {  
 public static void main(String[] args) throws Exception {  
 //往文件中写对象  
 writeObject2File();  
 //从文件中读取对象  
 Singleton s1 = *readObjectFromFile*();  
 Singleton s2 = *readObjectFromFile*();  
​  
 //判断两个反序列化后的对象是否是同一个对象  
 System.*out*.println(s1 == s2);  
 }  
​  
 private static Singleton readObjectFromFile() throws Exception {  
 //创建对象输入流对象  
 ObjectInputStream ois = new ObjectInputStream(new FileInputStream("C:\\Users \\Desktop\\a.txt"));  
 //第一个读取Singleton对象  
 Singleton instance = (Singleton) ois.readObject();  
​  
 return instance;  
 }  
​  
 public static void writeObject2File() throws Exception {  
 //获取Singleton类的对象  
 Singleton instance = Singleton.getInstance();  
 //创建对象输出流  
 ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("C:\\Users\\Desktop\\a.txt"));  
 //将instance对象写出到文件中  
 oos.writeObject(instance);  
 }  
}

破解序列化也简单，在单例类中，添加一个私有的readResolve()方法，让这个方法返回单例对象。如果定义了readResolve()方法，在调用readObject()时会自动调用 ，反序列化的结果就是该方法的返回值。如果没有定义该方法，则会序列化出新对象来

readResolve()方法会由ObjectInputStream类的readObject()方法触发，来自动调用

class Singleton implements Serializable {  
 private Object readResolve() {  
 return *getInstance*();  
 }

……

}

#### 2.1.3.3反射

通过反射也可以破坏单例模式，下列代码s1和s2对比，输出false，说明他们不是一个对象

public class Test {  
 public static void main(String[] args) throws Exception {  
 //获取Singleton类的字节码对象  
 Class clazz = Singleton.class;  
 //获取Singleton类的私有无参构造方法对象  
 Constructor constructor = clazz.getDeclaredConstructor();  
 //取消访问检查  
 constructor.setAccessible(true);  
​  
 //创建Singleton类的对象s1  
 Singleton s1 = (Singleton) constructor.newInstance();  
 //创建Singleton类的对象s2  
 Singleton s2 = (Singleton) constructor.newInstance();  
​  
 //判断通过反射创建的两个Singleton对象是否是同一个对象  
 System.*out*.println(s1 == s2);  
 }  
}

如何破解反射呢？反射最终都是调用无参构造函数的，因此我们只要这么做：

class Singleton {  
 //加volatile防止重排序  
 private static volatile Singleton *instance*;  
  
 private static volatile boolean *flag* = false;  
  
 //私有构造方法  
 private Singleton() {  
 synchronized (Singleton.class) {  
 if (*flag*) {  
 throw new RuntimeException();  
 }  
 *flag* = true;  
 }  
 }

}

### 2.1.4JDK中的单例模式

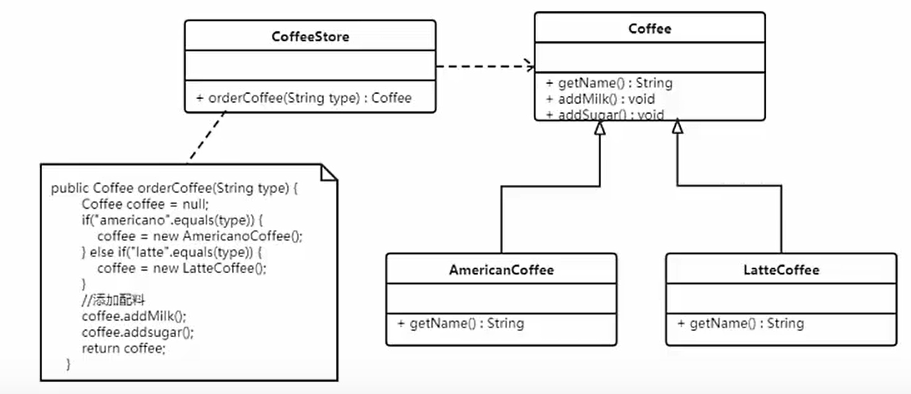
如RunTime类采用的是饿汉式实现

public class Runtime {  
 private static Runtime *currentRuntime* = new Runtime();  
public static Runtime getRuntime() {  
 return *currentRuntime*;  
 }  
  
 */\*\* Don't let anyone else instantiate this class \*/* private Runtime() {}

}

## 2.2工厂模式

我们首先来看一个咖啡店点咖啡的案例：



咖啡店类CoffeeStore允许下单不同种咖啡，而美式咖啡类AmericanCoffee和拿铁咖啡类LatteCoffee都实现了Coffee接口

那么上述代码有什么问题呢？

如果咖啡店希望新增一款咖啡，我们将被迫修改CoffeeStore类中的orderCoffee()的代码，如果超市里也需要上架这款咖啡，可能我们还要修改超市类，这违背件设计原则中的开闭原则

那么我们如何改进呢？

如果我们使用工厂来生产对象，我们就只和工厂打交道就可以了，彻底和使用者解耦，如果要更换对象，直接在工厂里更换该对象即可，达到了与对象解耦的目的；所以说，工厂模式最大的优点就是：解耦

常用的工厂模式有三种工厂：

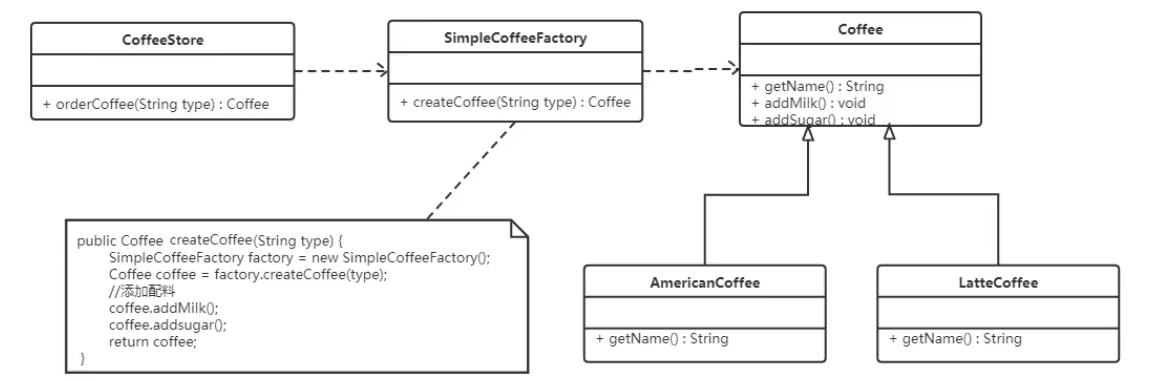
1. 简单工厂模式（不属于GOF的23种经典设计模式）
2. 工厂方法模式
3. 抽象工厂模式

### 2.2.1简单工厂模式

简单工厂不是一种设计模式，反而比较像是一种编程习惯。

简单工厂包含如下角色：

1. 产品 ：定义了产品的规范，描述了产品的主要特性和功能。
2. 具体产品 ：实现或者继承抽象产品的子类
3. 具体工厂 ：提供了创建产品的方法，调用者通过该方法来获取产品。



上图中只是加了一个简单咖啡工厂SimpleCoffeeFactory，就达到了解耦的效果，如果还有一个超市类，也不需要修改超市类中的代码，只需要统一修改工厂类的即可

其实现如下：

*/\*\*  
 \** ***@Description:*** *简单咖啡工厂类，用来生产咖啡  
 \*/*public class SimpleCoffeeFactory {  
 public static Coffee createCoffee(String type) {  
 //声明Coffee类型的变量，根据不同类型创建不同的coffee子类对象  
 Coffee coffee = null;  
 if("american".equals(type)) {  
 coffee = new AmericanCoffee();  
 } else if("latte".equals(type)) {  
 coffee = new LatteCoffee();  
 } else {  
 throw new RuntimeException("对不起，您所点的咖啡没有");  
 }  
 return coffee;  
 }  
}

但是这也产生了新的耦合，即CoffeeStore对象和SimpleCoffeeFactory工厂对象的耦合，工厂对象和商品对象的耦合。

**优点：**

封装了创建对象的过程，可以通过参数直接获取对象。把对象的创建和业务逻辑层分开，这样以后就避免了修改客户代码，如果要实现新产品直接修改工厂类，而不需要在原代码中修改，这样就降低了客户代码修改的可能性，更加容易扩展。

**缺点：**

增加新产品时还是需要修改工厂类的代码，违背了“开闭原则”。

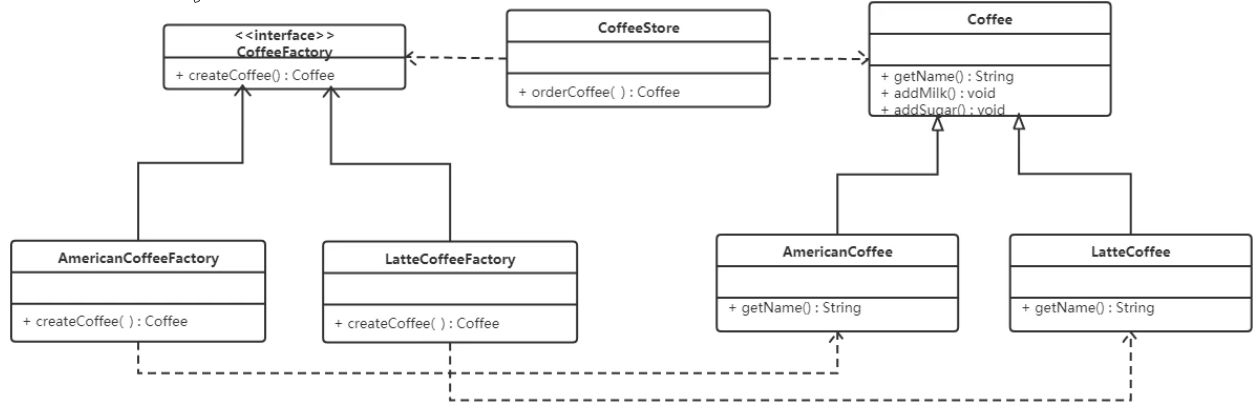
### 2.2.2工厂方法模式

简单工厂仍然存在耦合问题，使用工厂方法模式就可以完美的解决，完全遵循开闭原则。由此出现了工厂方法模式。

工厂方法模式定义一个用于创建对象的接口，让子类决定实例化哪个产品类对象。工厂方法使一个产品类的实例化延迟到其工厂的子类。

工厂方法主要有以下角色：

1. 抽象工厂（Abstract Factory）：提供了创建产品的接口，调用者通过它访问具体工厂的工厂方法来创建产品。
2. 具体工厂（ConcreteFactory）：主要是实现抽象工厂中的抽象方法，完成具体产品的创建。
3. 抽象产品（Product）：定义了产品的规范，描述了产品的主要特性和功能。
4. 具体产品（ConcreteProduct）：实现了抽象产品角色所定义的接口，由具体工厂来创建，它同具体工厂之间一一对应。



其实现代码如下：

*/\*\*  
 \** ***抽象工厂***

*\*/*

public interface CoffeeFactory {  
 //创建咖啡对象的方法  
 Coffee createCoffee();  
}

*/\*\*  
 \** ***@Description:*** *美式咖啡工厂对象，专门用来生产美式咖啡  
 \*/*public class AmericanCoffeeFactory implements CoffeeFactory {  
  
 public Coffee createCoffee() {  
 return new AmericanCoffee();  
 }  
}

*/\*\*  
 \** ***@Description:*** *拿铁咖啡工厂，专门用来生产拿铁咖啡  
 \*/*public class LatteCoffeeFactory implements CoffeeFactory {  
  
 public Coffee createCoffee() {  
 return new LatteCoffee();  
 }  
}

*/\*\*  
 \** ***@Description:*** *咖啡店  
 \*/*public class CoffeeStore {  
  
 private CoffeeFactory factory;  
  
 public void setFactory(CoffeeFactory factory) {  
 this.factory = factory;  
 }  
  
 //点咖啡功能  
 public Coffee orderCoffee() {  
 Coffee coffee = factory.createCoffee();  
 //加配料  
 coffee.addMilk();  
 coffee.addsugar();  
 return coffee;  
 }  
}

从以上的编写的代码可以看到，要增加产品类时也要相应地增加工厂类，而不需要修改工厂类的代码了，这样就解决了简单工厂模式的缺点。

工厂方法模式是简单工厂模式的进一步抽象。由于使用了多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点

**优点：**

1. 用户只需要知道具体工厂的名称就可得到所要的产品，无须知道产品的具体创建过程；
2. 在系统增加新的产品时只需要添加具体产品类和对应的具体工厂类，无须对原工厂进行任何修改，满足开闭原则；

**缺点：**

1. 每增加一个产品就要增加一个具体产品类和一个对应的具体工厂类，这增加了系统的复杂度。

### 2.2.3抽象工厂模式

工厂方法模式中考虑的是一类产品的生产，如畜牧场只养动物、电视机厂只生产电视机。然而在现实生活中许多工厂是综合型的工厂，能生产多等级（种类） 的产品，如电器厂既生产电视机又生产洗衣机或空调，大学既有软件专业又有生物专业等。

抽象工厂模式将考虑多等级产品的生产，将同一个具体工厂所生产的位于不同等级的一组产品称为一个产品族。产品等级，也就是同一类产品；纵产品族，也就是同一品牌/系列的产品，如美的洗衣机和美的空调是一个产品族，而格力空调和美的空调是同等级产品

抽象工厂模式是工厂方法模式的升级版本，工厂方法模式只生产一个等级的产品，而抽象工厂模式可生产同一产品族中的多个等级的产品

抽象工厂主要有如下角色：

**抽象工厂（Abstract Factory）：**提供了创建产品的接口，它包含多个创建产品的方法，可以创建多个不同等级的产品。

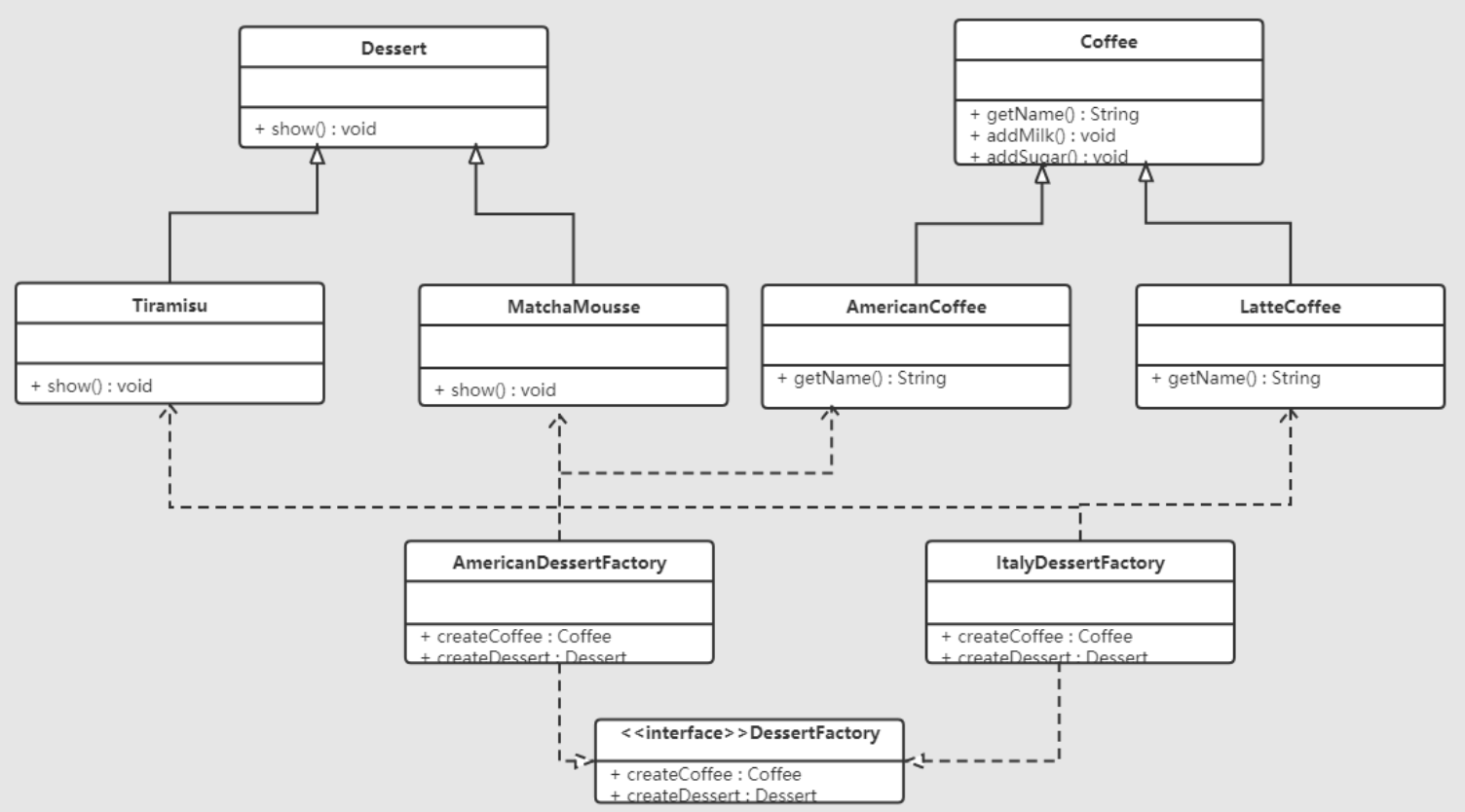
**具体工厂（Concrete Factory）：**主要是实现抽象工厂中的多个抽象方法，完成具体产品的创建。

**抽象产品（Product）：**定义了产品的规范，描述了产品的主要特性和功能，抽象工厂模式有多个抽象产品。

**具体产品（ConcreteProduct）：**实现了抽象产品角色所定义的接口，由具体工厂来创建，它 同具体工厂之间是多对一的关系。

现咖啡店不仅要生产咖啡还要生产甜点，如提拉米苏、抹茶慕斯等，要是按照工厂方法模式，需要定义提拉米苏类、抹茶慕斯类、提拉米苏工厂、抹茶慕斯工厂、甜点工厂类，很容易发生类爆炸情况。

其中拿铁咖啡、美式咖啡是一个产品等级，都是咖啡；提拉米苏、抹茶慕斯也是一个产品等级；拿铁咖啡和提拉米苏是同一产品族（也就是都属于意大利风味），美式咖啡和抹茶慕斯是同一产品族（也就是都属于美式风味）。所以这个案例可以使用抽象工厂模式实现。类图如下：



*/\*\*  
 \** ***@Description:*** *甜点抽象工厂  
 \*/*public interface DessertFactory {  
  
 //生产咖啡的功能  
 Coffee createCoffee();  
  
 //生产甜品的功能  
 Dessert createDessert();  
}

*/\*\*  
 \** ***@Description:****美式风味的甜品工厂，生产美式咖啡和抹茶慕斯（产品族）  
 \*/*public class AmericanDessertFactory implements DessertFactory {  
  
 public Coffee createCoffee() {  
 return new AmericanCoffee();  
 }  
  
 public Dessert createDessert() {  
 return new MatchaMousse();  
 }  
}

*/\*\**

*\** ***@Description:****意大利风味甜品工厂，生产拿铁咖啡和提拉米苏甜品（产品族）*

*\*/*

public class ItalyDessertFactory implements DessertFactory {  
 public Coffee createCoffee() {  
 return new LatteCoffee();  
 }  
  
 public Dessert createDessert() {  
 return new Trimisu();  
 }  
}

**优点：**

1. 当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。

**缺点：**

1. 当产品族中需要增加一个新的产品时，所有的工厂类都需要进行修改。

### 2.2.4扩展：通过工厂模式+配置文件解耦

可以通过工厂模式+配置文件的方式解除工厂对象和产品对象的耦合。在工厂类中加载配置文件中的全类名，并通过反射技术创建对象进行存储，客户端如果需要对象，直接进行获取即可。

eg：

american = com.itheima.pattern.factory.config\_factory.AmericanCoffee  
latte = com.itheima.pattern.factory.config\_factory.LatteCoffee

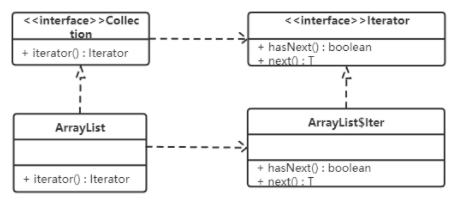
*/\*\*  
 \** ***@Description:*** *从配置文件读取  
 \*/*

public class CoffeeFactory {  
  
 //加载配置文件，获取配置文件中配置的全类名，并创建该类的对象进行存储  
 //1,定义容器对象存储咖啡对象  
 private static HashMap<String,Coffee> map = new HashMap<String, Coffee>();  
  
 //2,加载配置文件， 只需要加载一次  
 static {  
 //2.1 创建Properties对象  
 Properties p = new Properties();  
 //2.2 调用p对象中的load方法进行配置文件的加载  
 InputStream is = CoffeeFactory.class.getClassLoader().getResourceAsStream("bean.properties");  
 try {  
 p.load(is);  
 //从p集合中获取全类名并创建对象  
 Set<Object> keys = p.keySet();  
 for (Object key : keys) {  
 String className = p.getProperty((String) key);  
 //通过反射技术创建对象  
 Class clazz = Class.forName(className);  
 Coffee coffee = (Coffee) clazz.newInstance();  
 //将名称和对象存储到容器中  
 map.put((String)key,coffee);  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
  
 //根据名称获取对象  
 public static Coffee createCoffee(String name) {  
 return map.get(name);  
 }  
}

静态成员变量用来存储创建的对象（键存储的是名称，值存储的是对应的对象），而读取配置文件以及创建对象写在静态代码块中，目的就是只需要执行一次

### 2.2.5JDK源码中的工厂模式(Collection.iterator)

JDK中的Collection.iterator()方法就使用了工厂方法模式:



Collection接口是抽象工厂类，ArrayList是具体的工厂类；Iterator接口是抽象商品类，ArrayList类中的Iter内部类是具体的商品类。在具体的工厂类中iterator()方法创建具体的商品类的对象

另：

1. DateForamt类中的getInstance()方法使用的是工厂模式；
2. Calendar类中的getInstance()方法使用的是工厂模式；

## 2.3原型模式

## 2.4建造者模式

### 2.4.1建造者模式

建造者模式是将一个复杂的对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。创建者模式隐藏了复杂对象的创建过程，它把复杂对象的创建过程加以抽象。

简单来说，就是当一个类的内部数据过于复杂的时候（通常是负责持有数据的类，比如Config、VO、PO、Entity...），要创建的话可能就需要了解这个类的内部结构，还有这些东西是怎么组织装配等，这个时候就会增加学习成本而且会很混乱，这个时候就需要在创建的时候让它按部就班地构建，并且代码可读性很好别让人看花了眼，这就是Builder模式的应用场景，Builder模式可以将一个类的构建和表示进行分离。

**当一个类的构造函数参数个数超过4个，而且这些参数有些是可选的参数，考虑使用构造者模式。**

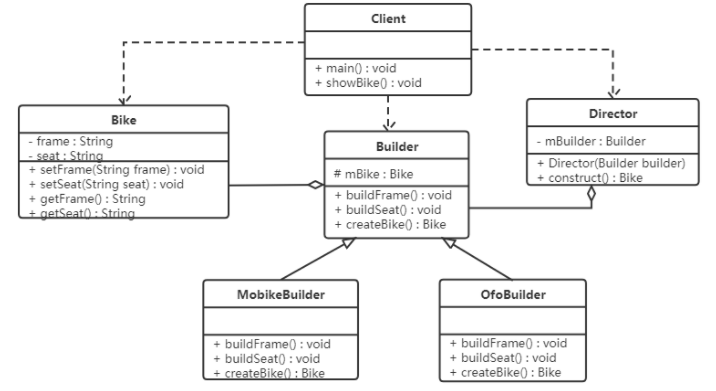
建造者模式包含如下几个角色：

1. **抽象建造者类（Builder）：**这个接口规定要实现复杂对象的那些部分的创建，并不涉及具体的部件对象的创建。
2. **具体建造者类（ConcreteBuilder）：**实现 Builder 接口，完成复杂产品的各个部件的具体创建方法。在构造过程完成后，提供产品的实例。
3. **产品类（Product）：**要创建的复杂对象。
4. **指挥者类（Director）**：调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建。

下面我们来看一个创建共享单车的案例：

生产自行车是一个复杂的过程，它包含了车架，车座等组件的生产。而车架又有碳纤维，铝合金等材质的，车座有橡胶，真皮等材质。对于自行车的生产就可以使用建造者模式。

这里Bike是产品，包含车架，车座等组件；Builder是抽象建造者，MobikeBuilder和OfoBuilder是具体的建造者；Director是指挥者。类图如下：



*/\*\*  
 \** ***@Description: 自行车产品对象*** *\*/*

@Datapublic class Bike {  
 private String frame;//车架  
 private String seat;//车座  
}

*/\*\*  
 \** ***@Description: 建造者类***

*\*/*public abstract class Builder {  
 //声明Bike类型的变量，并进行赋值  
 protected Bike bike = new Bike();  
  
 public abstract void buildFrame();  
 public abstract void buildSeat();  
  
 //构建自行车的方法  
 public abstract Bike createBike();  
}

*/\*\*  
 \** ***@Description:*** *具体的构建者，用来构建摩拜单车对象  
 \*/*public class MobileBuilder extends Builder {  
  
 public void buildFrame() {  
 bike.setFrame("碳纤维车架");  
 }  
  
 public void buildSeat() {  
 bike.setSeat("真皮车座");  
 }  
  
 public Bike createBike() {  
 return bike;  
 }  
}

public class OfoBuilder extends Builder {  
 ……  
}

*/\*\*  
 \** ***@Description:*** *指挥者类  
 \*/*public class Director {  
  
 //声明builder类型的变量  
 private Builder builder;  
  
 public Director(Builder builder) {  
 this.builder = builder;  
 }  
  
 //组装自行车的功能  
 public Bike construct() {  
 builder.buildFrame();  
 builder.buildSeat();  
 return builder.createBike();  
 }  
}

public class Client {  
 public static void main(String[] args) {  
 //创建指挥者对象  
 Director director = new Director(new MobileBuilder());  
 //让指挥者只会组装自行车  
 Bike bike = director.construct();  
  
 System.out.println(bike.getFrame());  
 System.out.println(bike.getSeat());  
 }  
}

**优点：**

1. 建造者模式的封装性很好。使用建造者模式可以有效的封装变化，在使用建造者模式的场景中，一般产品类和建造者类是比较稳定的，因此，将主要的业务逻辑封装在指挥者类中对整体而言可以取得比较好的稳定性。
2. 在建造者模式中，客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象。
3. 可以更加精细地控制产品的创建过程 。将复杂产品的创建步骤分解在不同的方法中，使得创建过程更加清晰，也更方便使用程序来控制创建过程。
4. 建造者模式很容易进行扩展。如果有新的需求，通过实现一个新的建造者类就可以完成，基本上不用修改之前已经测试通过的代码，因此也就不会对原有功能引入风险。符合开闭原则。

**缺点：**

1. 建造者模式所创建的产品一般具有较多的共同点，其组成部分相似，如果产品之间的差异性很大，则不适合使用建造者模式，因此其使用范围受到一定的限制。比如组装电脑和组装自行车之间的差异就很大，不适合使用同一个builder类来组装这两种对象

**使用场景：**

1. 创建的对象较复杂，由多个部件构成，各部件面临着复杂的变化，但构件间的建造顺序是稳定的。(一个类的构造函数参数个数超过4个，而且这些参数有些是可选的参数)
2. 创建复杂对象的算法独立于该对象的组成部分以及它们的装配方式，即产品的构建过程和最终的表示是独立的。

### 2.4.2扩展：开发中一般怎么用

当一个类构造器需要传入很多参数时，如果创建这个类的实例，代码可读性会非常差，而且很容易引入错误，此时就可以利用建造者模式进行重构。如下列代码，如果仅仅通过:

**Phone phone = new Phone("intel","三星屏幕","金士顿","华硕")**

**无论是代码可读性还是使用成本都是比较高的，那么可以通过建造者模式进行修改：**

public class Phone {  
  
 private String cpu;  
 private String screen;  
 private String memory;  
 private String mainboard;  
  
 //私有构造方法  
 private Phone(Builder builder) {  
 this.cpu = builder.cpu;  
 this.screen = builder.screen;  
 this.memory = builder.memory;  
 this.mainboard = builder.mainboard;  
 }  
  
 @Override  
 public String toString() {  
 return "Phone{" +  
 "cpu='" + cpu + '\'' +  
 ", screen='" + screen + '\'' +  
 ", memory='" + memory + '\'' +  
 ", mainboard='" + mainboard + '\'' +  
 '}';  
 }  
  
 public static final class Builder {  
 private String cpu;  
 private String screen;  
 private String memory;  
 private String mainboard;  
  
 public Builder cpu(String cpu) {  
 this.cpu = cpu;  
 return this;  
 }  
  
 public Builder screen(String screen) {  
 this.screen = screen;  
 return this;  
 }  
 public Builder memory(String memory) {  
 this.memory = memory;  
 return this;  
 }  
 public Builder mainboard(String mainboard) {  
 this.mainboard = mainboard;  
 return this;  
 }  
  
 //使用构建者创建Phone对象  
 public Phone build() {  
 return new Phone(this);  
 }  
 }  
}

## 2.5创建者模式总结

### 2.5.1 工厂方法模式VS建造者模式

工厂方法模式注重的是整体对象的创建方式；而建造者模式注重的是部件构建的过程，意在通过一步一步地精确构造创建出一个复杂的对象。

我们举个简单例子来说明两者的差异，如要制造一个超人，如果使用工厂方法模式，直接产生出来的就是一个力大无穷、能够飞翔、内裤外穿的超人；而如果使用建造者模式，则需要组装手、头、脚、躯干等部分，然后再把内裤外穿，于是一个超人就诞生了。

### 2.5.2 抽象工厂模式VS建造者模式

抽象工厂模式实现对产品家族的创建，一个产品家族是这样的一系列产品：具有不同分类维度的产品组合，采用抽象工厂模式则是不需要关心构建过程，只关心什么产品由什么工厂生产即可。

建造者模式则是要求按照指定的蓝图建造产品，它的主要目的是通过组装零配件而产生一个新产品。

如果将抽象工厂模式看成汽车配件生产工厂，生产一个产品族的产品，那么建造者模式就是一个汽车组装工厂，通过对部件的组装可以返回一辆完整的汽车。

# 结构型模式

创建型模式的主要关注点是“怎样创建对象？”，它的主要特点是“将对象的创建与使用分离”。

## 3.1代理模式

### 3.1.1什么是代理模式

由于某些原因需要给某对象提供一个代理以控制对该对象的访问。这时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介。

**Java中的代理按照代理类生成时机不同又分为静态代理和动态代理。静态代理代理类在编译期就生成，而动态代理代理类则是在Java运行时动态生成。动态代理又有JDK代理和CGLib代理两种**

代理（Proxy）模式分为三种角色：

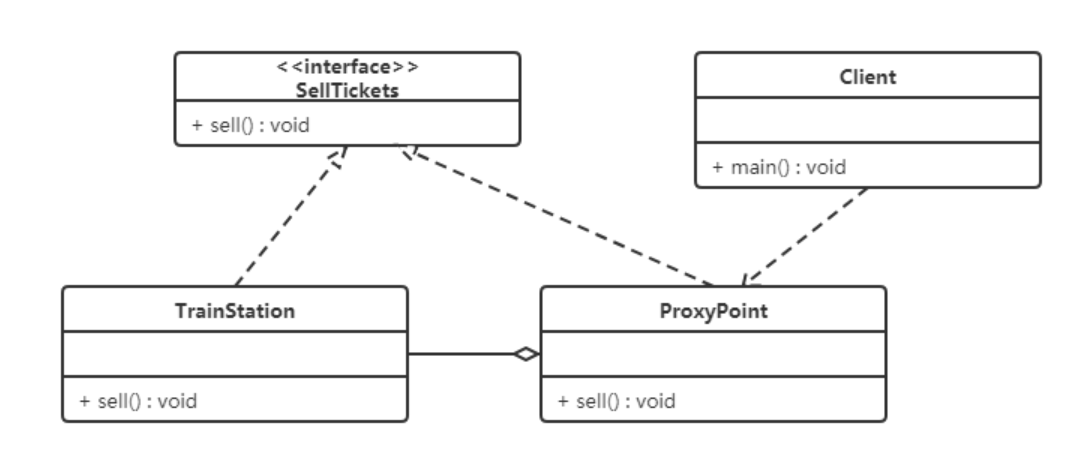
1. **抽象主题（Subject）类：** 通过接口或抽象类声明真实主题和代理对象实现的业务方法。
2. **具体主题（Real Subject）类：** 实现了抽象主题中的具体业务，是代理对象所代表的真实对象，是最终要引用的对象。
3. **代理（Proxy）类 ：** 提供了与真实主题相同的接口，其内部含有对真实主题的引用，它可以访问、控制或扩展真实主题的功能。

### 3.1.2静态代理

**静态代理代理类在编译期就生成**

【例】火车站卖票

如果要买火车票的话，需要去火车站买票，坐车到火车站，排队等一系列的操作，显然比较麻烦。而火车站在多个地方都有代售点，我们去代售点买票就方便很多了。这个例子其实就是典型的代理模式，火车站是目标对象，代售点是代理对象。类图如下：



*/\*\*  
 \** ***@Description:*** *卖火车票的接口  
 \*/*public interface SellTickets {  
  
 void sell();  
}

*/\*\*  
 \** ***@Description:*** *火车站类  
 \*/*public class TrainStation implements SellTickets {  
  
 public void sell() {  
 System.out.println("火车站卖票");  
 }  
}

*/\*\*  
 \** ***@Description:*** *代售点类  
 \*/*public class ProxyPoint implements SellTickets {  
  
 //声明火车站类对象  
 private TrainStation trainStation = new TrainStation();  
  
 public void sell() {  
 System.out.println("代售点收取一些服务费用");  
 trainStation.sell();  
 }  
  
}

public static void main(String[] args) {  
 //创建代售点类对象  
 SellTickets proxyPoint = new ProxyPoint();  
 //调用方法进行买票  
 proxyPoint.sell();  
}

上面代码中可以看出测试类直接访问的是ProxyPoint类对象，也就是说代理对象ProxyPoint作为访问对象和目标对象的中介。同时也对sell方法进行了增强（代理点收取一些服务费用）。

### 3.1.3JDK动态代理

**动态代理代理类则是在Java运行时动态生成**

Java中提供了一个动态代理类Proxy，Proxy并不是我们上述所说的代理对象的类，而是提供了一个创建代理对象的静态方法（newProxyInstance方法）来获取代理对象

*/\*\*  
 \** ***@Description:*** *卖火车票的接口  
 \*/*public interface SellTickets {  
  
 void sell();  
}

*/\*\*  
 \** ***@Description:*** *火车站类  
 \*/*public class TrainStation implements SellTickets {  
  
 public void sell() {  
 System.out.println("火车站卖票");  
 }  
}

*/\*\*  
 \** ***@Description:*** *获取代理对象的工厂类  
 \* 代理类也实现了对应的接口  
\*/*public class ProxyFactory {  
  
 //声明目标对象  
 private TrainStation station = new TrainStation();  
  
 //获取代理对象的方法  
 public SellTickets getProxyObject() {  
 //返回代理对象  
 /\*  
 ClassLoader loader : 类加载器，用于加载代理类。可以通过目标对象获取类加载器  
 Class<?>[] interfaces ： 代理类实现的接口的字节码对象  
 InvocationHandler h ： 代理对象的调用处理程序  
 \*/  
 SellTickets proxyObject = (SellTickets)Proxy.newProxyInstance(  
 station.getClass().getClassLoader(),  
 station.getClass().getInterfaces(),  
 new InvocationHandler() {  
 /\*  
 Object proxy : 代理对象。和proxyObject对象是同一个对象，在invoke方法中基本不用  
 Method method ： 对接口中的方法进行封装的method对象,如果接口中有N个方法，那么这个方法就是 使用代理对象时调用的那个方法  
 Object[] args ： 调用方法的实际参数  
  
 返回值： 方法的返回值。  
 \*/  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 //System.out.println("invoke方法执行了");  
 System.out.println("代售点收取一定的服务费用(jdk动态代理)");  
 //执行目标对象的方法  
 Object obj = method.invoke(station, args);  
 return obj;  
 }  
 }  
 );  
 return proxyObject;  
 }  
}

public static void main(String[] args) {  
 //获取代理对象  
 //1,创建代理工厂对象  
 ProxyFactory factory = new ProxyFactory();  
 //2,使用factory对象的方法获取代理对象  
 SellTickets proxyObject = factory.getProxyObject();  
 //3,调用卖调用的方法  
 proxyObject.sell();  
  
 System.out.println(proxyObject.getClass());  
  
 //让程序一直执行  
 while(true) {}  
  
}

JDK动态代理，实际上是在运行时在内存中动态生成了一个类$Proxy，下面是其简化后的代码：

//程序运行过程中动态生成的代理类  
public final class $Proxy0 extends Proxy implements SellTickets {  
 private static Method m3;  
​  
 public $Proxy0(InvocationHandler invocationHandler) {  
 super(invocationHandler);  
 }  
​  
 static {  
 m3 = Class.forName("com.itheima.proxy.dynamic.jdk.SellTickets").getMethod("sell", new Class[0]);  
 }  
​  
 public final void sell() {

//执行匿名内部类InvocationHandler中的invoke方法  
 this.h.invoke(this, m3, null);  
 }  
}

但是使用JDK动态代理是有条件的，如果没有定义SellTickets接口，只定义了TrainStation(火车站类)。很显然JDK代理是无法使用了，因为JDK动态代理要求必须定义接口，对接口进行代理。

### 3.1.4 CGLIB动态代理

CGLIB是一个功能强大，高性能的代码生成包。它为没有实现接口的类提供代理，为JDK的动态代理提供了很好的补充

CGLIB是第三方提供的包，所以需要引入jar包的坐标：

<dependency>  
 <groupId>cglib</groupId>  
 <artifactId>cglib</artifactId>  
 <version>2.2.2</version>  
</dependency>

还是火车站的例子

*/\*\*  
 \** ***@Description:*** *火车站类  
 \*/*public class TrainStation {  
  
 public void sell() {  
 System.out.println("火车站卖票");  
 }  
}

*/\*\*  
 \** ***@Description:*** *代理对象工厂，用来获取代理对象  
 \*/*public class ProxyFactory implements MethodInterceptor {  
  
 //声明火车站对象  
 private TrainStation station = new TrainStation();  
  
 public TrainStation getProxyObject() {  
 //创建Enhancer对象，类似于JDK代理中的Proxy类  
 Enhancer enhancer = new Enhancer();  
 //设置父类的字节码对象。指定父类  
 enhancer.setSuperclass(TrainStation.class);  
 //设置回调函数  
 enhancer.setCallback(this);  
 //创建代理对象  
 TrainStation proxyObject = (TrainStation) enhancer.create();  
 return proxyObject;  
 }  
  
 public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {  
 //System.out.println("方法执行了");  
 System.out.println("代售点收取一定的服务费用(CGLib代理)");  
 //要调用目标对象的方法  
 Object obj = method.invoke(station, objects);  
 return obj;  
 }  
}

public static void main(String[] args) {  
 //创建代理工厂对象  
 ProxyFactory factory = new ProxyFactory();  
 //获取代理对象  
 TrainStation proxyObject = factory.getProxyObject();  
 //调用代理对象中的sell方法卖票  
 proxyObject.sell();  
}

### 3.1.4 javassist动态代理

### 3.1.4 ASM动态代理

### 3.1.5 三种代理的对比

#### 3.1.5.1jdk代理和CGLIB代理

使用CGLib实现动态代理，CGLib底层采用ASM字节码生成框架，使用字节码技术生成代理类，在JDK1.6之前比使用Java反射效率要高。唯一需要注意的是，CGLib不能对声明为final的类或者方法进行代理，因为CGLib原理是动态生成被代理类的子类。

在JDK1.6、JDK1.7、JDK1.8逐步对JDK动态代理优化之后，在调用次数较少的情况下，JDK代理效率高于CGLib代理效率，只有当进行大量调用的时候，JDK1.6和JDK1.7比CGLib代理效率低一点，但是到JDK1.8的时候，JDK代理效率高于CGLib代理。所以如果有接口使用JDK动态代理，如果没有接口使用CGLIB代理。

#### 3.1.5.2动态代理和静态代理

动态代理与静态代理相比较，最大的好处是接口中声明的所有方法都被转移到调用处理器一个集中的方法中处理（InvocationHandler.invoke）。这样，在接口方法数量比较多的时候，我们可以进行灵活处理，而不需要像静态代理那样每一个方法进行中转。

如果接口增加一个方法，静态代理模式除了所有实现类需要实现这个方法外，所有代理类也需要实现此方法。增加了代码维护的复杂度。而动态代理不会出现该问题

### 3.1.6 优缺点

**优点：**

1. 代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用；
2. 代理对象可以扩展目标对象的功能；
3. 代理模式能将客户端与目标对象分离，在一定程度上降低了系统的耦合度；

**缺点：**

1. 增加了系统的复杂度；

## 3.2适配器模式

## 3.3桥接模式

## 3.4外观模式

## 3.5装饰者模式

## 3.6享元模式

## 3.7组合模式

# 行为模式

## 4.1模板方法

## 4.2策略模式

## 4.3命令模式

## 4.4责任链模式

### 4.4.1什么是责任链模式

在现实生活中，常常会出现这样的事例：一个请求有多个对象可以处理，但每个对象的处理条件或权限不同。例如，公司员工请假，可批假的领导有部门负责人、副总经理、总经理等，但每个领导能批准的天数不同，员工必须根据自己要请假的天数去找不同的领导签名，也就是说员工必须记住每个领导的姓名、电话和地址等信息，这增加了难度。

为了避免请求发送者与多个请求处理者耦合在一起，将所有请求的处理者通过前一对象记住其下一个对象的引用而连成一条链；当有请求发生时，可将请求沿着这条链传递，直到有对象处理它为止。

责任链模式主要包含以下角色:

1. 抽象处理者（Handler）角色：定义一个处理请求的接口，包含抽象处理方法和一个后继连接。
2. 具体处理者（Concrete Handler）角色：实现抽象处理者的处理方法，判断能否处理本次请求，如果可以处理请求则处理，否则将该请求转给它的后继者。
3. 客户类（Client）角色：创建处理链，并向链头的具体处理者对象提交请求，它不关心处理细节和请求的传递过程。

如果不使用责任链模式，我们的代码可能是这样的：

//创建一个请假条对象  
LeaveRequest leave = new LeaveRequest("小明",1,"身体不适");  
  
//创建各级领导对象  
BigManager bigManager = new BigManager();  
GeneralManager generalManager = new GeneralManager();

SmallManager smallManager = new SmallManager();  
  
//小明提交请假条給领导审批

if(smallManager.audit(leave)){  
 return "ok";  
} else if(generalManager.audit(leave)){  
 return "ok";  
}else if(bigManager.audit(leave)){  
 return "ok";  
}  
return "failed";

通过上面的代码我们可以看到，请求发送者(小明)和请求处理者(领导们)耦合严重，通过责任链模式，我们可以这样修改：

*/\*\*  
 \** ***@Description:*** *抽象处理者类  
 \*/*public abstract class Handler {  
 //该领导能处理的请假最大天数

private int maxNum;  
 //声明后续者（声明上级领导）  
 private Handler nextHandler;  
  
 public Handler(int maxNum) {  
 this.maxNum = maxNum;  
 }  
  
 //设置上级领导对象  
 public void setNextHandler(Handler nextHandler) {  
 this.nextHandler = nextHandler;  
 }  
  
 //处理请假条  
 protected abstract void handleLeave(LeaveRequest leave);  
  
 //提交请求条  
 public final void submit(LeaveRequest leave) {  
 //当前领导进行审批  
 this.handleLeave(leave);  
 if(this.nextHandler != null && leave.getNum() > this.maxNum) {  
 //没有审批权限，提交给上级领导进行审批  
 this.nextHandler.submit(leave);  
 } else {  
 System.out.println("流程结束！");  
 }  
 }  
}

public static void main(String[] args) {  
 //创建一个请假条对象  
 LeaveRequest leave = new LeaveRequest("小明",1,"身体不适");  
  
 //创建各级领导对象  
 BigManager bigManager = new BigManager();  
 GeneralManager generalManager = new GeneralManager();

SmallManager smallManager = new SmallManager();  
  
 //设置处理者链  
 smallManager.setNextHandler(generalManager);  
 generalManager.setNextHandler(bigManager);  
  
 //小明提交请假申请  
 smallManager.submit(leave);  
}

**优点：**

1. 降低了请求发送者和接收者的耦合度。增强了系统的可扩展性
2. 责任链简化了对象之间的连接，增强了给对象指派职责的灵活性，当工作流程发生变化，可以动态地改变链内的成员或者修改它们的次序，也可动态地新增或者删除责任对象。满足开闭原则。
3. 一个对象只需保持一个指向其后继者的引用，不需保持其他所有处理者的引用，这避免了使用众多的 if 或者 if···else 语句。
4. 每个类只需要处理自己该处理的工作，不能处理的传递给下一个对象完成，明确各类的责任范围，符合类的单一职责原则。

2，缺点：

1. 不能保证每个请求一定被处理。由于一个请求没有明确的接收者，所以不能保证它一定会被处理，该请求可能一直传到链的末端都得不到处理。
2. 对比较长的职责链，请求的处理可能涉及多个处理对象，系统性能将受到一定影响。
3. 职责链建立的合理性要靠客户端来保证，增加了客户端的复杂性，可能会由于职责链的错误设置而导致系统出错，如可能会造成循环调用。

### 4.4.2Java中的责任链模式

在javaWeb应用开发中，FilterChain是职责链（过滤器）模式的典型应用，以下是Filter的模拟实现分析

public interface Request {  
}

public interface Response {  
}

*/\*\*  
 \** ***@Description:*** *模拟web过滤器Filter  
 \*/*public interface Filter {  
 public void doFilter(Request req,Response res,FilterChain c);  
}

*/\*\*  
 \** ***@Description:*** *模拟实现具体过滤器  
 \*/*public class FirstFilter implements Filter {  
 public void doFilter(Request req, Response res, FilterChain chain) {  
 System.out.println("过滤器1 前置处理");  
  
 // 先执行所有request再倒序执行所有response  
 chain.doFilter(req, res);  
  
 System.out.println("过滤器1 后置处理");  
 }  
}

*/\*\*  
 \** ***@Description:*** *模拟实现具体过滤器  
 \*/*public class SecondFilter implements Filter {  
 public void doFilter(Request req, Response res, FilterChain chain) {  
 System.out.println("过滤器2 前置处理");  
  
 // 先执行所有request再倒序执行所有response  
 chain.doFilter(req, res);  
  
 System.out.println("过滤器2 后置处理");  
 }  
}

*/\*\*  
 \** ***@Description:*** *模拟实现过滤器链FilterChain  
\*/*public class FilterChain { //过滤器链  
  
 private List<Filter> filters = new ArrayList<Filter>();  
  
 private int index = 0;  
  
 // 链式调用  
 public FilterChain addFilter(Filter filter) {  
 this.filters.add(filter);  
 return this;  
 }  
  
 public void doFilter(Request request, Response response) {  
 if (index == filters.size()) {  
 return;  
 }  
 Filter filter = filters.get(index);  
 index++;  
 filter.doFilter(request, response, this);  
 }  
}

public static void main(String[] args) {  
 Request req = null;  
 Response res = null ;  
  
 FilterChain filterChain = new FilterChain();  
 filterChain.addFilter(new FirstFilter()).addFilter(new SecondFilter());  
 filterChain.doFilter(req,res);  
}

## 4.5状态模式

## 4.6观察者模式

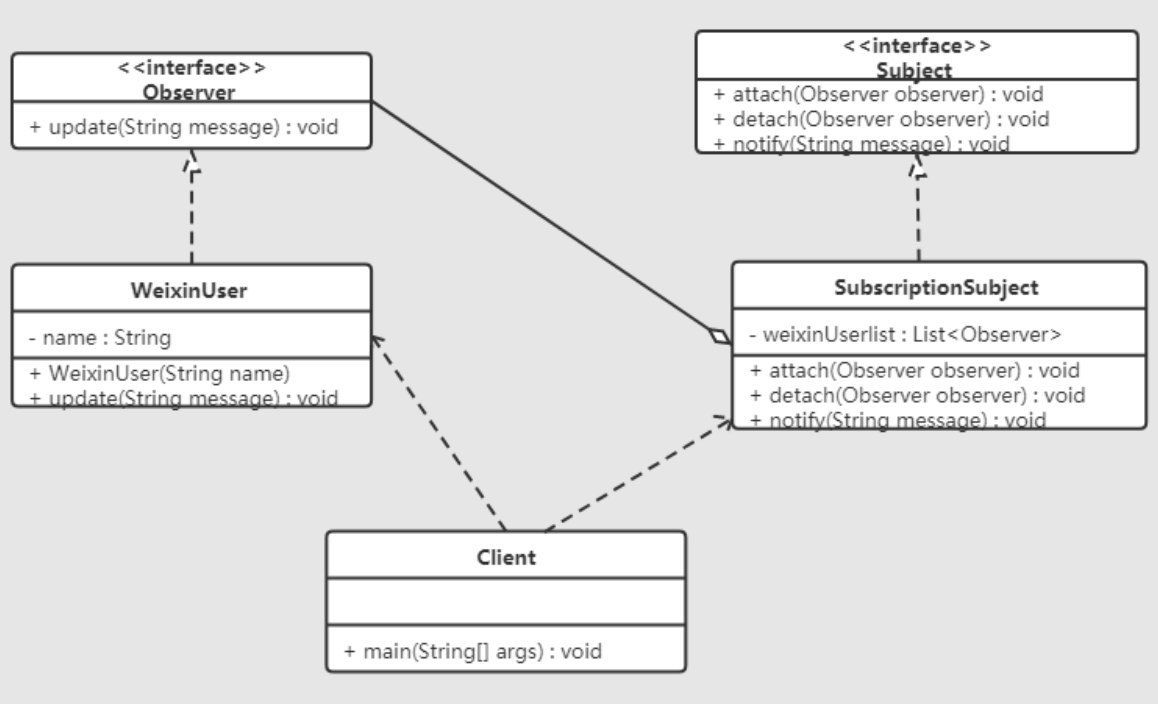
### 4.6.1什么是观察者模式

又被称为发布-订阅（Publish/Subscribe）模式，它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变化时，会通知所有的观察者对象，使他们能够自动更新自己

在观察者模式中有如下角色：

1. Subject：抽象主题（抽象被观察者），抽象主题角色把所有观察者对象保存在一个集合里，每个主题都可以有任意数量的观察者，抽象主题提供一个接口，可以增加和删除观察者对象。
2. ConcreteSubject：具体主题（具体被观察者），该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发生改变时，给所有注册过的观察者发送通知。
3. Observer：抽象观察者，是观察者的抽象类，它定义了一个更新接口，使得在得到主题更改通知时更新自己。
4. ConcrereObserver：具体观察者，实现抽象观察者定义的更新接口，以便在得到主题更改通知时更新自身的状态。

下面我们来看一个微信公众号的案例：当你关注的公众号中有新内容更新的话，它就会推送给关注公众号的微信用户端。我们使用观察者模式来模拟这样的场景，微信用户就是观察者，微信公众号是被观察者



*/\*\*  
 \** ***@Description:*** *抽象主题角色类  
 \*/*public interface Subject {  
 //添加订阅者（添加观察者对象）  
 void attach(Observer observer);  
 //删除订阅者  
 void detach(Observer observer);  
 //通知订阅者更新消息  
 void notify(String message);  
}

*/\*\*  
 \** ***@Description:*** *具体主题角色类  
 \*/*public class SubscriptionSubject implements Subject {  
  
 //定义一个集合，用来存储多个观察者对象  
 private List<Observer> weiXinUserList = new ArrayList<Observer>();  
  
 public void attach(Observer observer) {  
 weiXinUserList.add(observer);  
 }  
  
 public void detach(Observer observer) {  
 weiXinUserList.remove(observer);  
 }  
  
 public void notify(String message) {  
 //遍历集合  
 for (Observer observer : weiXinUserList) {  
 //调用观察者对象中的update方法  
 observer.update(message);  
 }  
 }  
}

*/\*\*  
 \** ***@Description:*** *抽象观察者类  
 \*/*public interface Observer {  
  
 void update(String message);  
}

*/\*\*  
 \** ***@Description:*** *具体的观察者角色类  
 \*/*public class WeiXinUser implements Observer {  
 private String name;  
  
 public WeiXinUser(String name) {  
 this.name = name;  
 }  
 public void update(String message) {  
 System.out.println(name + "-" + message);  
 }  
}

public class Client {  
 public static void main(String[] args) {  
 //1,创建公众号对象  
 SubscriptionSubject subject = new SubscriptionSubject();  
  
 //2,订阅公众号  
 subject.attach(new WeiXinUser("孙悟空"));  
 subject.attach(new WeiXinUser("猪悟能"));  
 subject.attach(new WeiXinUser("沙悟净"));  
  
 //3,公众号更新，发出消息给订阅者（观察者对象）  
 subject.notify("传智黑马的专栏更新了！");  
 }  
}

**优点：**

1. 降低了目标与观察者之间的耦合关系，两者之间是抽象耦合关系。
2. 被观察者发送通知，所有注册的观察者都会收到信息【可以实现广播机制】

**缺点：**

1. 如果观察者非常多的话，那么所有的观察者收到被观察者发送的通知会耗时较久
2. 如果被观察者有循环依赖的话，那么被观察者发送通知会使观察者循环调用，会导致系统崩溃

**使用场景:**

1. 对象间存在一对多关系，一个对象的状态发生改变会影响其他对象。
2. 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时。

### 4.6.2JDK中的观察者模式

在 Java 中，通过 java.util.Observable 类和 java.util.Observer 接口定义了观察者模式，只要实现它们的子类就可以编写观察者模式实例。

1. Observer 接口

public interface Observer {  
 void update(Observable o, Object arg);  
}

1. Observable类

Observable 类是抽象目标类（被观察者），它有一个 Vector 集合成员变量，用于保存所有要通知的观察者对象，下面来介绍它最重要的 3 个方法。

public class Observable {  
 private boolean changed = false;  
 private Vector<Observer> obs;public Observable() {  
 obs = new Vector<>();  
 }  
  
 */\*\*  
 \* 用于将新的观察者对象添加到集合中  
 \*/* public synchronized void addObserver(Observer o) {  
 if (o == null)  
 throw new NullPointerException();  
 if (!obs.contains(o)) {  
 obs.addElement(o);  
 }  
 }  
  
 */\*\*  
 \* 取消订阅  
 \*/* public synchronized void deleteObserver(Observer o) {  
 obs.removeElement(o);  
 }  
  
 */\*\*  
 \*调用集合中的所有观察者对象的 update方法，通知它们数据发生改变。通常越晚加入集合的观察者越先得到通知  
 \*/* public void notifyObservers() {  
 notifyObservers(null);  
 }  
  
 */\*\*  
 \* 通知观察者  
 \*/* public void notifyObservers(Object arg) {  
 Object[] arrLocal;  
 synchronized (this) {  
 if (!changed)  
 return;  
 arrLocal = obs.toArray();  
 clearChanged();  
 }  
 for (int i = arrLocal.length-1; i>=0; i--)  
 ((Observer)arrLocal[i]).update(this, arg);  
 }  
  
  
 */\*\*  
 \*用来设置一个 boolean 类型的内部标志，注明目标对象发生了变化。当它为true时，notifyObservers() 才会通知观察者。  
 \*/* protected synchronized void setChanged() {  
 changed = true;  
 }

## 4.7中介者模式

## 4.8迭代器模式

## 4.9访问者模式

## 4.10备忘录模式

## 4.11解释器模式