# 一、设计模式的分类

http://blog.csdn.net/anxpp/article/details/51224293

## 1. 三大类

### 1.1 创建型模式

(1) 定义：

创建型模式涉及对象的实例化，特点是不让用户代码依赖于对象的创建或排列方式，避免用户直接使用new创建对象。

(2) 共五种：工厂方法模式（简单工厂模式）、抽象工厂模式、单例模式、建造者模式（生成器模式）、原型模式。

### 1.2 结构型模式，

(1) 定义：

结构型模式涉及如何组合类和对象以形成更大的结构，和类有关的结构型模式涉及如何合理使用继承机制；和对象有关的结构型模式涉及如何合理的使用对象组合机制。

(2) 共七种：适配器模式、装饰器模式、代理模式、外观模式（门面模式）、桥接模式、组合模式、享元模式。

### 1.3 行为型模式，

(1) 定义：

行为型模式涉及怎样合理的设计对象之间的交互通信，以及怎样合理为对象分配职责，让设计富有弹性，易维护，易复用。

(2) 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。



# 二、设计模式的六大原则

## 1. 开闭原则（Open Close Principle）

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。

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

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。

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

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。

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

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。

## 5. 迪米特法则（最少知道原则）（Demeter Principle）

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。

## 6. 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。

# 三、创建型设计模式

## 1. 单例模式（Singleton Pattern）

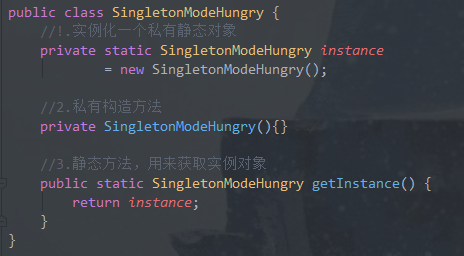
(1) 定义：任何情况下，确保一个类在整个系统中仅有一个实例，而且自行实例化，并想系统提供一个全局访问点。

(2) 优点：单例模式的类唯一实例由其本身控制，可以很好的控制用户何时访问它。

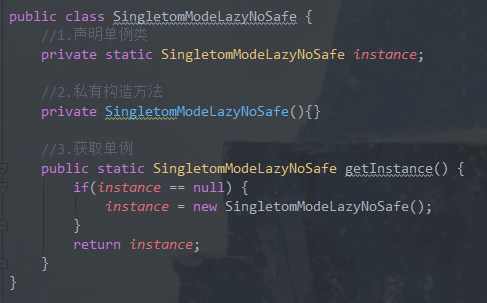
1.1 实例：

(1) 饿汉式 – 调用方法获取实例前就创建好实例，线程安全

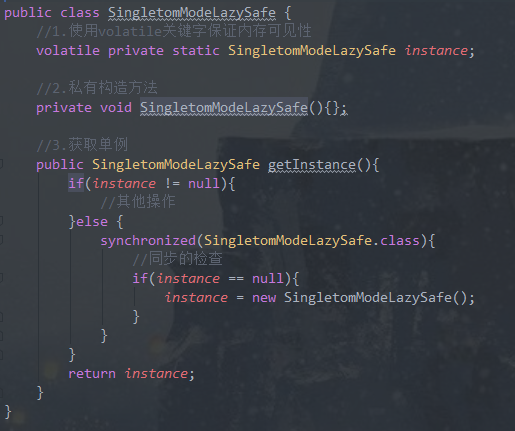
线程安全是指多线程访问时，从始至终保证实例只有一个，而不能保证单例模式中的变量(语句执行)是同步的。



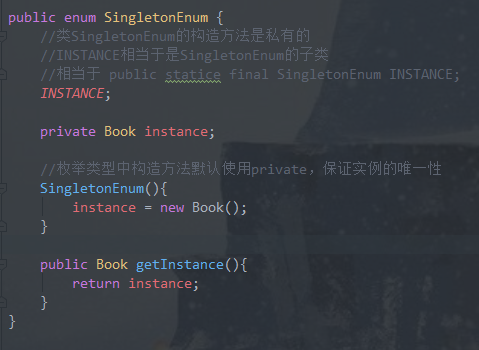
(2) 懒汉式 – 调用方法获取实例时才创建实例，非线程安全



(3) 懒汉模式 – 使用锁，线程安全



(4) 使用枚举类创建单例模式-保证某个类只有一个实例



## 2. 工厂方法模式（Factory Method Pattern）

--- 别名：虚拟构造（Another Name：Virtual Constructor）

(1) 定义：

定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法使一个类的实例化延迟到其子类。

(2) 优点：

① 使用工厂方法可以让用户的代码和某个特定类的子类的代码解耦。

② 工厂方法使用户不必知道它所使用的对象是怎样被创建的，只需知道该对象有哪些方法即可。

(3) 应用背景：

① 用户需要一个类的子类的实例，但不希望与该类的子类形成耦合

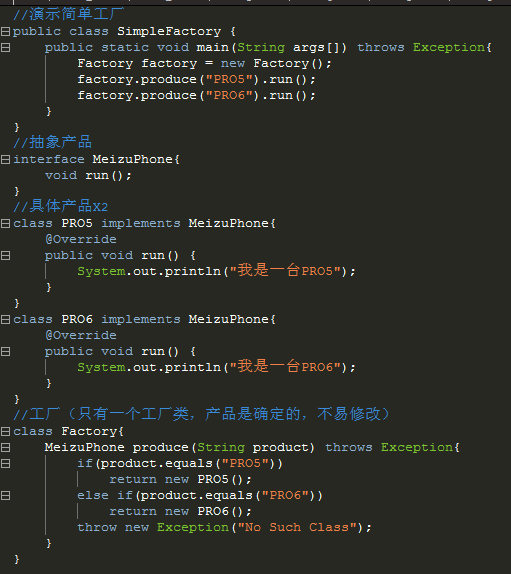
② 用户需要一个类的子类的实例，但用户不知道该类有哪些子类可用

2.1 简单工厂模式（工厂方法模式的特例）

(1) 应用背景：

简单工厂模式又称静态工厂方法模式。存在的目的：定义一个用于创建对象的接口。

如果一些对象（产品），已经确定了并不易改变和添加新的产品，那么就可以使用简单工厂模式。（不易维护，添加新产品要修改全部工厂类）



2.2 工厂方法模式

(1) 应用背景

工厂方法模式去掉了简单工厂模式中工厂方法的静态属性，使得它可以被子类继承。

(2) 原理：

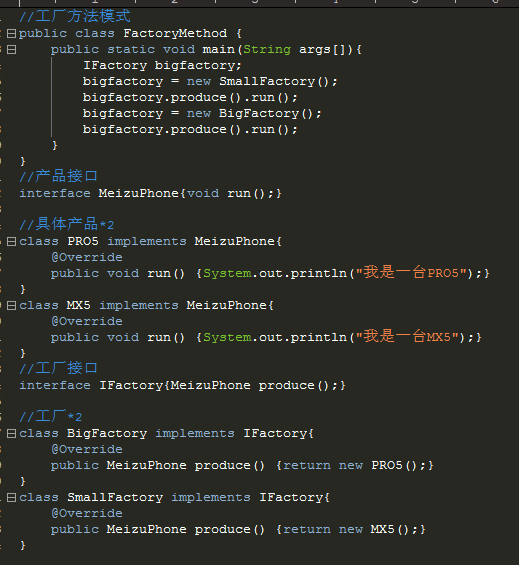
抽象产品：产品接口，提供统一的标准。

具体产品：实现类，提供多样的产品。

构造者：工厂接口，规定统一的制造产品的方法。

具体构造者：实现类，构造具体的不同的子类，返回接口都是产品接口，具体实例是各个子类。

① 一抽象产品类派生出多个具体产品类；一抽象工厂类派生出多个具体工厂类；每个具体工厂类只能创建一个具体产品类的实例。 即定义一个创建对象的接口（即抽象工厂类），让其子类（具体工厂类）决定实例化哪一个类（具体产品类）。“一对一”的关系。



## 3. 抽象工厂方法模式（Abstract Factory Pattern）

--- 别名：配套（Another Name：Kit）

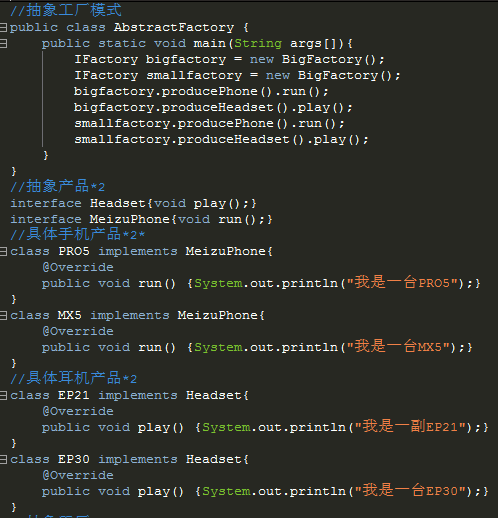
(1) 定义：

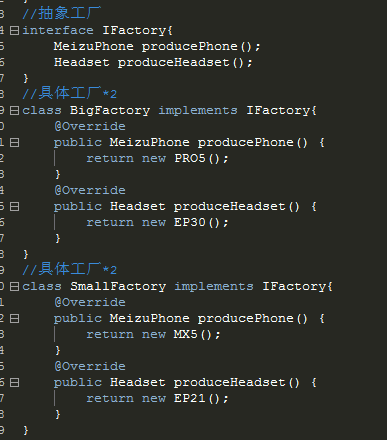
提供一个创建一系列或相互依赖对象的接口，而无须指定他们的具体的类。

(2) 应用背景：

上述生成魅族产品的例子中，我们只生产了手机，但是它不止有手机一种产品，可能还有其他的，比如耳机，为了还可以生成耳机，我们需要对上例进行扩展。

(3) 实例：





(4) 工厂模式总结

① 简单工厂模式是由一个具体的类去创建其他类的实例，父类是相同的，父类是具体的。

② 工厂方法模式是有一个抽象的父类定义公共接口，子类负责生成具体的对象，这样做的目的是将类的实例化操作延迟到子类中完成。

③ 抽象工厂模式提供一个创建一系列相关或相互依赖对象的接口，而无须指定他们具体的类。它针对的是有多个产品的等级结构。而工厂方法模式针对的是一个产品的等级结构。

## 4. 生成器模式（Builder Pattern）

(1) 定义：

将一个复杂对象的构建与它的表示分离，使同样的构建过程可以创建不同的表示。

(2) 应用背景：

① 当系统准备为用户提供一个内部结构复杂的对象，而且在构造方法中编写创建该对象的代码无法满足用户需求时，就可以使用生成器模式来构造这样的对象。

② 当某些系统要求对象的构造过程必须独立于创建该对象的类时。

(3) 优点：

① 生成器模式将对象的构造过程封装在具体的生成器中，用户使用不同的具体生成器就可以得到该对象的不同表示。

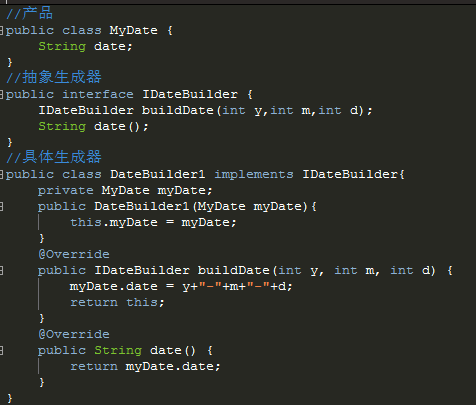
② 生成器模式将对象的构造过程从创建该对象的类中分离出来，使用户无须了解该对象的具体组件。

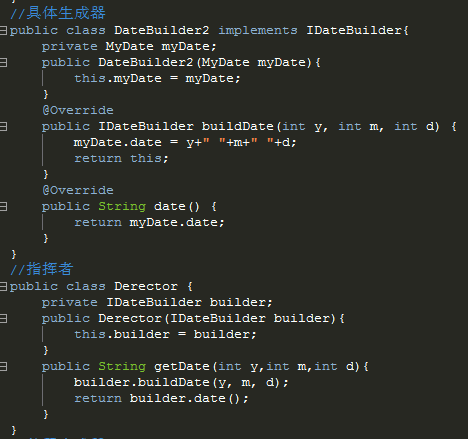
③ 可以更加精细有效的控制对象的构造过程。生成器将对象的构造过程分解成若干步骤，这就是程序可以更加精细，有效的控制整个对象的构造。

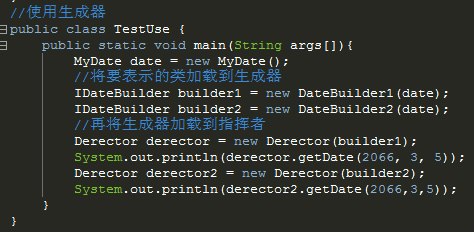
④ 生成器模式将对象的构造过程与创建该对象类解耦，是对象的创建更加灵活有弹性。

⑤ 当增加新的具体的生成器是，不必修改指挥者的代码，即该模式满足开-闭原则。

(4) 实例代码：







## 5. 原型模式（和克隆相似）

(1) 定义：

用原型实例指定创建对象的种类，并且通过复制这些原型创建新的对象。

(2) 应用背景：

① 程序需要从一个对象出发，得到若干个和其状态相同，并可独立变化其状态的对象时。

② 当对象的创建需要独立于它的构造过程和表示时。

③ 一个类创建实例状态不是很多，那么就可以将这个类的一个实例定义为原型，那么通过该实例复制该原型得到新的实例可能比重新使用类的构造方法创建新实例更方便

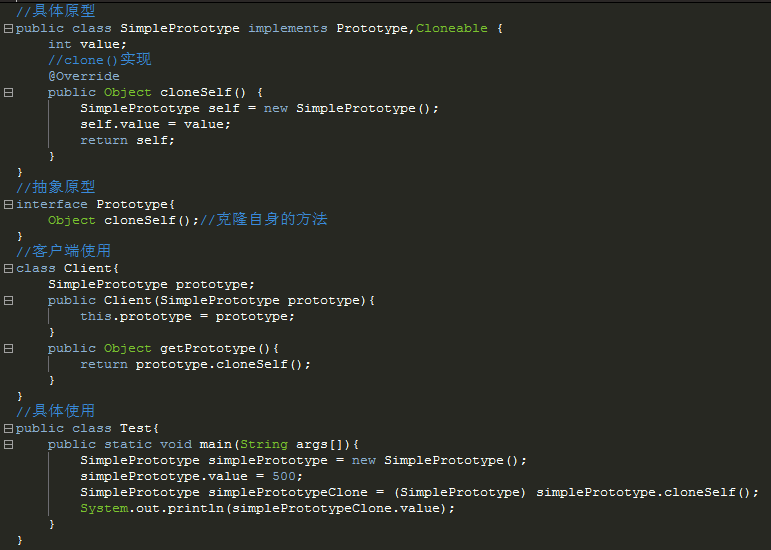
(3) 优点：

① 程序需要从一个对象出发，得到若干个和其状态相同，并可独立变化其状态的对象时。

② 当对象的创建需要独立于它的构造过程和表示时。

③ 一个类创建实例状态不是很多，那么就可以将这个类的一个实例定义为原型，那么通过该实例复制该原型得到新的实例可能比重新使用类的构造方法创建新实例更方便

(4) 实例代码



# 四、结构型设计模式

## 1. 适配器模式（Adapter Pattern）

(1) 定义：

将一个类的接口转换成客户希望的另外一个接口。该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起工作。原理就是保留现有的类所提供的服务，修改其接口，从而达到客户端的期望。

(2) 应用背景

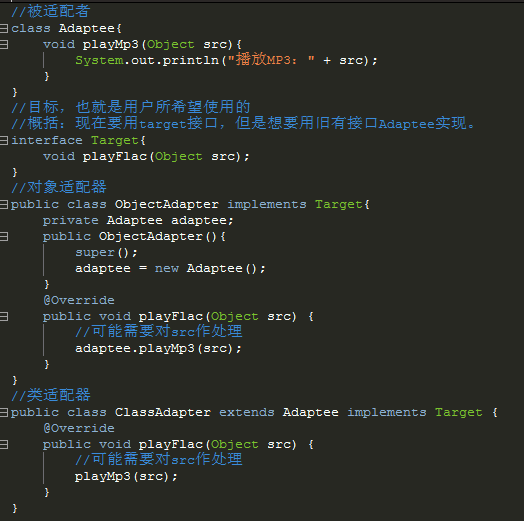
一个程序想使用已经存在的类，但是该类所实现的接口和当前程序所使用的接口不一致时。

(3) 优点：

① 目标与被适配者解耦

② 满足开-闭原则

(4) 实例代码：



## 2. 装饰模式（Decorator Pattern）

-- 别名：包装器（Wrapper）

(1) 定义：

动态的给对象添加额外的职责。就功能来说，装饰模式比生产子类更为灵活。装饰模式使用被装饰类的一个子类的实例，把客户端的调用委派到被装饰类，装饰模式的关键在于这种扩展是完全透明的。装饰者与被装饰者拥有共同的超类，继承的目的是继承类型，而不是行为。

(2) 应用背景：

① 程序希望动态的增强类的某对对象的功能，而不影响其他对象时

② 采用继承来增强对象功能不利于系统的扩展和维护时

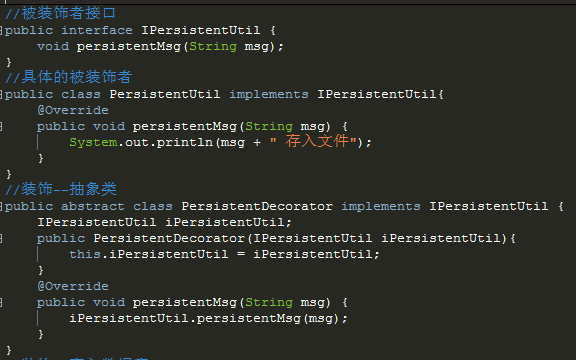
(3) 优点：

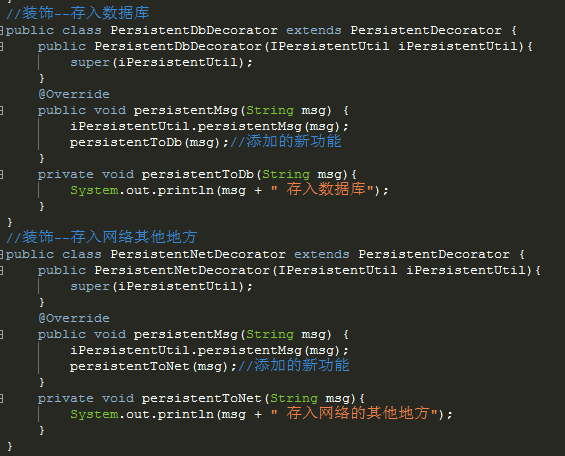
① 被装饰者和装饰者是松耦合关系。

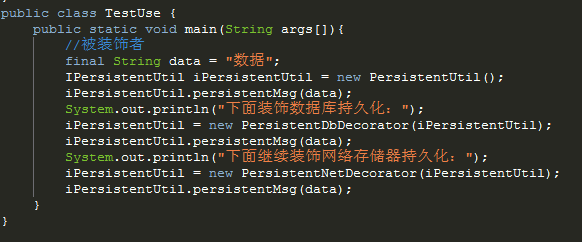
② 满足开-闭原则

③ 可以使用多个具体装饰器装饰具体组件的实例

(4) 实例代码：







## 3. 代理模式

(1) 定义：

为其它对象提供一种代理以控制对这个对象的访问。一个用户不想或者不能够直接引用一个对象（或者设计者不希望用户直接访问该对象），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。

(2) 应用背景：

① 程序可能不希望用户直接访问该对象，而是提供一个特殊的对象以控制对当前对象的访问。

② 如果一个对象（例如很大的图像）需要很长时间才能完成加载。

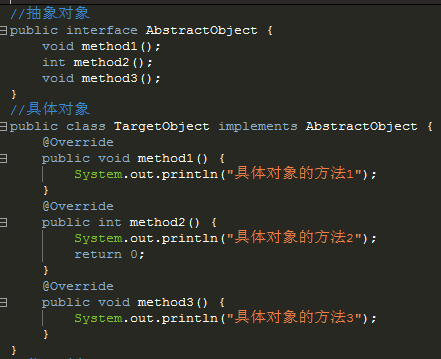
③ 如果对象位于远程主机上，需要为用户提供访问该远程对象的能力。

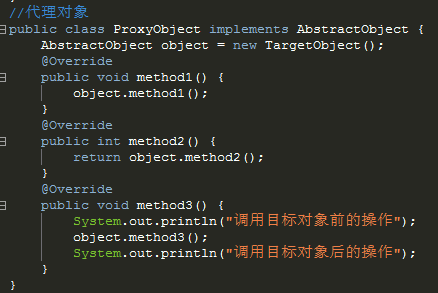
(3) 优点：

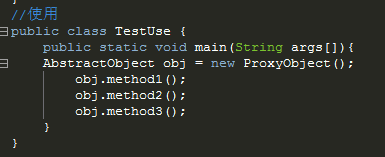
① 代理模式可以屏蔽用户真正请求的对象，是用户程序和正在的对象解耦。

② 使用代理来担当那些创建耗时的对象的替身。

(4) 实例代码







# 五、行为型模式

## 1. 策略模式（Strategy Pattern）

(1) 定义：

① 定义一系列算法，把他们一个个封装起来，并且使他们可相互替换。本模式使得算法可独立于其他客户端而变化。

② 策略模式是对算法的包装，是把使用算法的责任和算法本身分割开来，委派给不同的对象管理。策略模式通常把一个系列的算法包装到一系列的策略类里面，作为一个抽象策略类的子类。用一句话来说，就是：“准备一组算法，并将每一个算法封装起来，使得它们可以互换”。

(2) 策略模式中包括三种角色：

① 策略（Strategy）：一个接口，定义了若干个算法（抽象方法）。

② 具体策略（ConcreteStrategy）：策略的实现。

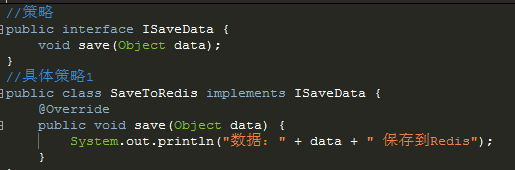
③ 上下文/环境（Context）：依赖于策略接口的类。

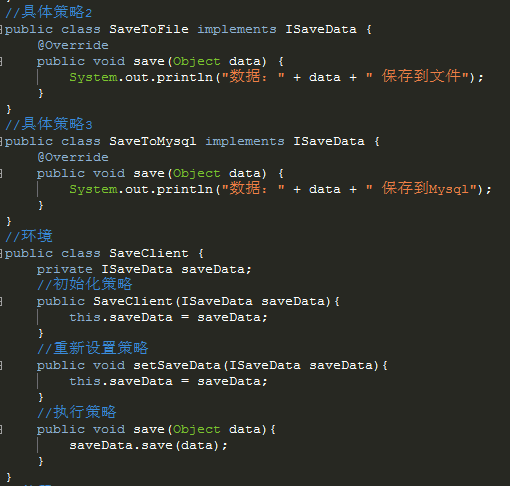
(3) 特点

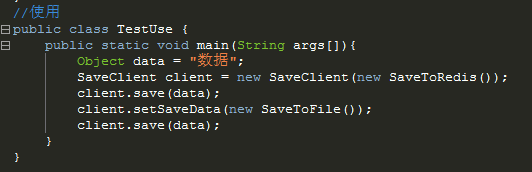
① 策略模式一个很大的特点就是各个策略算法的平等性。对于一系列具体的策略算法，大家的地位是完全一样的，正因为这个平等性，才能实现算法之间可以相互替换。所有的策略算法在实现上也是相互独立的，相互之间是没有依赖的。所以可以这样描述这一系列策略算法：策略算法是相同行为的不同实现。

② 运行期间，策略模式在每一个时刻只能使用一个具体的策略实现对象，虽然可以动态地在不同的策略实现中切换，但是同时只能使用一个。

(4) 实例代码







(5) 注意：

使用策略模式可以避免使用多重条件(if-else)语句。多重条件语句不易维护，它把采取哪一种算法或采取哪一种行为的逻辑与算法或行为的逻辑混合在一起，统统列在一个多重条件语句里面，比使用继承的办法还要原始和落后。

## 2. 模板方法模式（Template Method Pattern）

(1) 定义：

① 定义一个操作中算法的骨架，而将一些步骤延迟到子类中。模板方法使子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

② 模板方法模式是所有模式中最为常见的几个模式之一，是基于继承的代码复用的基本技术。 模板方法模式需要开发抽象类和具体子类的设计师之间的协作。一个设计师负责给出一个算法的轮廓和骨架，另一些设计师则负责给出这个算法的各个逻辑步骤。代表这些具体逻辑步骤的方法称做基本方法(primitive method)；而将这些基本方法汇总起来的方法叫做模板方法(template method)，

(2) 应用背景：

① 设计者需要给出一个算法的固定步骤，并将某些步骤的具体实现留给子类来实现。

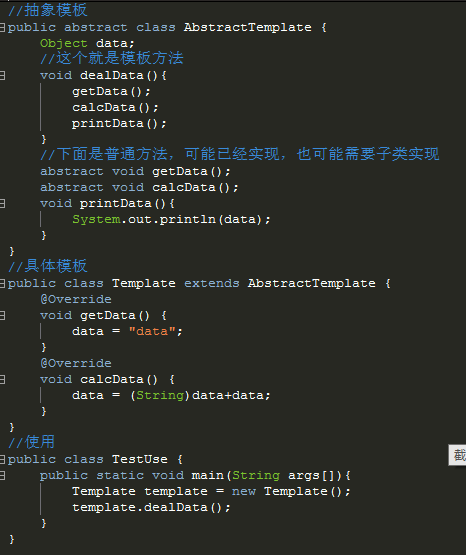
② 需要对代码进行重构，将各个子类公共行为提取出来集中到一个共同的父类中以避免代码重复。

(3) 优点：

① 可以通过在抽象模板能定义模板方法给出成熟的算法步骤，同时又不限制步骤的细节，具体模板实现算法细节不会改变整个算法的骨架。

② 在抽象模板模式中，可以通过钩子方法对某些步骤进行挂钩，具体模板通过钩子可以选择算法骨架中的某些步骤。

(4) 实例代码：（和继承实现类似）



## 3. 观察者模式（Observer Pattern）

——别名： 依赖，发布/订阅（Another Name: Dependents, Publish/Subscribe）

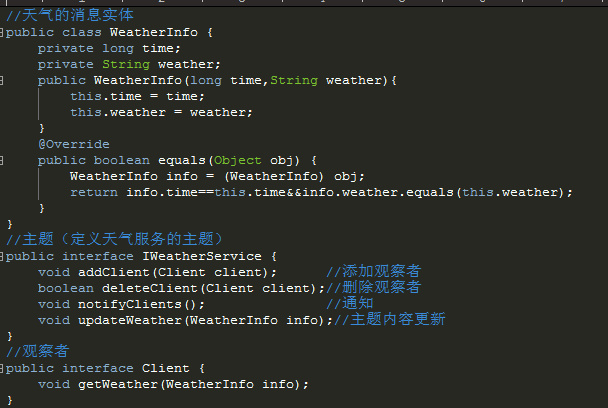
(1) 定义：

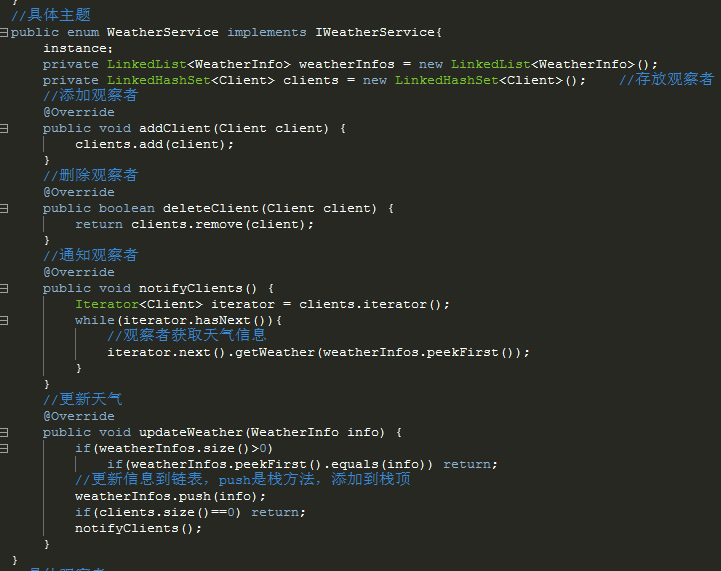
定义对象间的一种一对多的依赖关系，当一个对象状态发生改变时，所有依赖它的对象都得到通知并被自动更新。

(2) 应用背景：

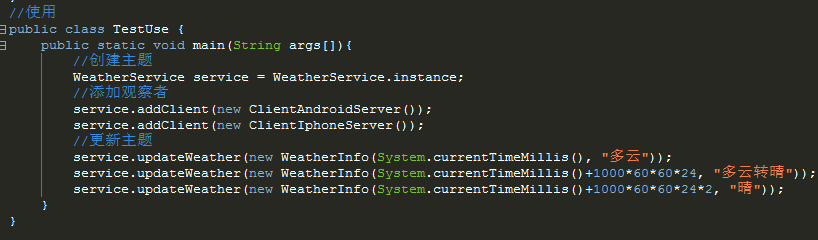
当一个对象的数据更新时，需要通知其他对象，而又不希望和被通知的对象形成紧耦合时。

(3) 实例代码：









步骤：主题 → 天气更新 → 推送给观察者 → 观察者收到推送作出动作

## 4. 命令模式(Command Pattern)

——别名：动作，事物（Another Name：Action，Transaction）

(1) 定义：

将一个请求封装为一个对象，从而使用户可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤销的操作。一个对象有多种操作，但是我们不希望调用者（请求者）直接使用，我们就额外添加一个对象，然后让调用者通过这个对象来使用那些操作。

(2) 应用背景：

① 程序需要在不同的时刻指定、排列和执行请求。

② 程序需要提供撤销操作。

③ 程序需要支持宏操作。

(3) 优点：

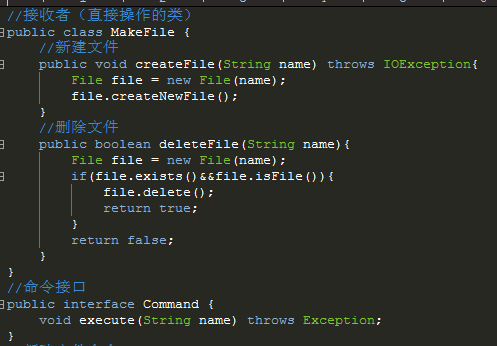
① 在命令模式中，请求者（Invoker）不直接与接受者（Receiver）交互，及请求者（Invoker）不包含接受者（Receiver）的引用，因此彻底消除了彼此间的耦合。

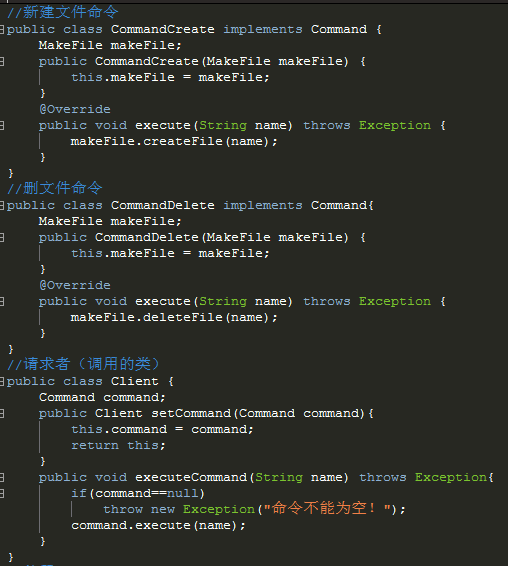
② 命令模式满足“开-闭原则”。如果增加新的具体命令和该命令的接受者，不必修改调用者的代码，调用者就可以使用新的命令对象；反之，如果增加新的调用者，不必修改现有具体命令和接收者，新增加的调用者就可以使用已有的具体命令。

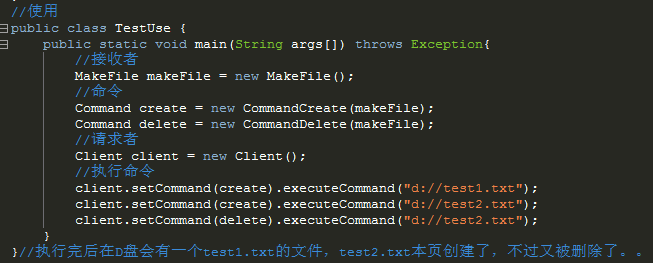
③ 由于请求者的请求被封装到具体的命令中，那么就可以将具体命令保存到持久化的媒介中，在需要的时候，重新执行这个具体命令。因此，使用命令模式可以记录日志。

④ 使用命令模式可以对请求者的“请求”进行排队。每个请求都各自对应一个具体命令，因此可以按一定顺序执行这些具体命令。

(4) 实例代码：







步骤：请求者→执行命令→命令类调用实际操作类→完成操作