目录

[1. 单例模式singleton (创建型) 2](#_Toc461827251)

[2. 工厂模式Factory(创建型) 3](#_Toc461827252)

[(1) 简单工厂模式(静态工厂方法模式) 3](#_Toc461827253)

[(2) 工厂方法模式 4](#_Toc461827254)

[(3) 抽象工厂模式 4](#_Toc461827255)

[3. 原型模式Prototype（创建型） 5](#_Toc461827256)

[4. 适配器模式Adapter\_Object (结构型) 6](#_Toc461827257)

[5. 组合模式Composite(结构型) 6](#_Toc461827258)

[6. 享元模式Flyweight (结构型) 7](#_Toc461827259)

[7. 装饰者模式Decorator (结构型) 8](#_Toc461827260)

[8. 代理模式Proxy(结构型) 10](#_Toc461827261)

[9. 门面模式Facade(结构型) 10](#_Toc461827262)

[10. 观察者模式Observer(行为型) 11](#_Toc461827263)

[11. 责任链模式Chain of Resonsibility(行为型) 12](#_Toc461827264)

[12. 命令模式Command(行为型) 13](#_Toc461827265)

[13. 模板方法Template Method(行为型) 14](#_Toc461827266)

[14. 迭代器模式Iterator(行为型) 14](#_Toc461827267)

[15. 策略模式Strategy(行为型) 15](#_Toc461827268)

[16. 状态模式State(行为型) 16](#_Toc461827269)

[17. 责任链模式和命令模式的区别 16](#_Toc461827270)

[18. 责任链模式和状态模式的区别 17](#_Toc461827271)

[19. 责任链模式和策略模式的区别 17](#_Toc461827272)

[20. 装饰者模式与代理模式的区别： 18](#_Toc461827273)

[21. 详解如何实现单例模式singleton 19](#_Toc461827274)

[22. Jdk中体现的设计模式 24](#_Toc461827275)

[23. singleton bean和singleton模式区别 24](#_Toc461827276)

设计模式的研究和应用是以依赖倒转原则为指导原则的。

设计模式是一套反复使用、为多数人知晓、经过分类编目的、代码设计经验的总结。使用设计模式的目的是为了代码复用，避免程序大量修改，同时使代码易于理解，并且保证代码的可靠性。

设计模式不管是对自己还是对他人都是有用的，引入设计模式的概念，使得程序在编写过程中可以实现工程化管理，设计模式可以说是软件工程的基石。

GoF(Gang of Four)23种经典的设计模式。

|  |  |  |  |
| --- | --- | --- | --- |
|  | 创建型 | 结构型 | 行为型 |
| 类 | Factory Method(工厂方法) | Adapter(适配器) | Intepreter(解释器)  Template Method(模板方法) |
| 对象 | 单例模式singleton  工厂模式Factory  原型模式Prototype | 适配器模式Adapter\_Object  组合模式Composite  享元模式Flyweight  装饰者模式Decorator  代理模式Proxy  门面模式Facade(结构型) | 观察者模式Observer  责任链模式Resonsibility  命令模式Command  模板方法Template Method  迭代器模式Iterator  策略模式Strategy  状态模式State |

创建型设计模式的意图在于将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。创建型设计模式抽象了实例化过程，它们帮助一个系统独立于如何创建、组合和表示它的那些对象。

1. 单例模式singleton (创建型)

单例模式的作用就是保证在整个应用程序的周期中，任何一个时刻，单例类的实例都只存在一个（当然也可以不存在）。

使用场景：(1) 保证一个类仅有一个实例，而且客户可以从一个众所周知的接口访问它时；(2)当这个唯一实例应该是通过子类化可扩展的类并且客户应该无需更改代码就能使用一个扩展的实例时。

实现方案：建立一个类，其构造方法是私有的，只有一个getInstance方法是public的，通过这个方法获取该类的唯一实例，即让类自身保存这个唯一的实例。

单例模式确保某一个类只有一个实例，而且自行实例化，并向整个系统提供这个实例单例模式。单例模式只应在有真正的”单一实例“需求时才可使用。

应用实例：数据库连接池。

|  |
| --- |
| **public** **class** Singleton {  **public** **static** Singleton *instance*;  **private** Singleton(){  }  **public** Singleton getInstance(){  **if**(*instance* == **null**){  **return** **new** Singleton();  }**else**{  **return** *instance*;  }  }  } |

类图如下：



优点：(1)对唯一实例的受控访问；(2)缩小命名空间；(3) 允许操作和表示的精化；(4)允许可变数目的实例；(5)比类操作更灵活。

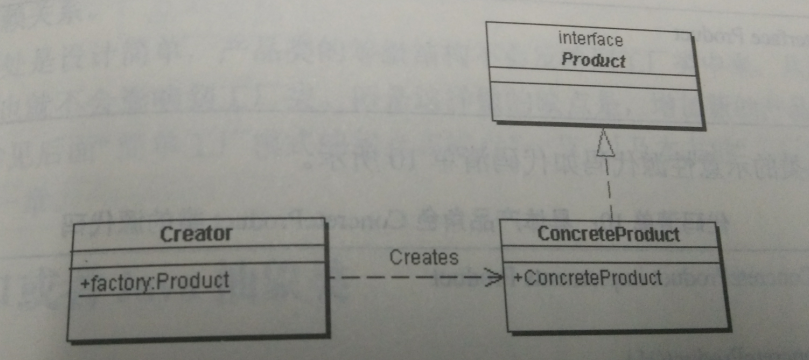
1. 工厂模式Factory(创建型)

工厂模式专门负责实例化有大量公共接口的类。工厂模式可以动态地决定将哪一个类实例化，而不必事先知道每次都需要实例化哪个类。客户类和工厂类是分开的。消费者无论什么时候需要某种产品，需要做的只是向工厂提出请求即可。消费者无需修改就可以接纳新产品。当然也有缺点，就是当产品修改时，工厂类也要做相应的修改。

工厂模式分为简单工厂模式、工厂方法模式、抽象工厂模式。

1. 简单工厂模式(静态工厂方法模式)

简单工厂模式的工厂类是根据提供给它的参数，返回的是几个可能产品中的一个类的实例，一般情况下，它返回的类都有一个公共的父类和公共的方法。



简单工厂模式涉及到工厂角色、抽象产品角色以及具体产品角色等三个角色：

工厂类：担任这个角色的是工厂方法模式的核心，含有与应用紧密相连的商业逻辑。工厂类在客户端的直接调用下创建产品对象。

抽象产品：有工厂模式所创建的对象的父类，或创建对象共同拥有的接口。当模式产生的具体产品之间有逻辑关系的时候可以把共同的部分抽象为父类，当模式产生的具体产品之间没有逻辑关系的时候可以定义一个接口来担任抽象产品的这个角色。

具体产品：工厂方法模式所创建的任何对象都是这个角色的实例。

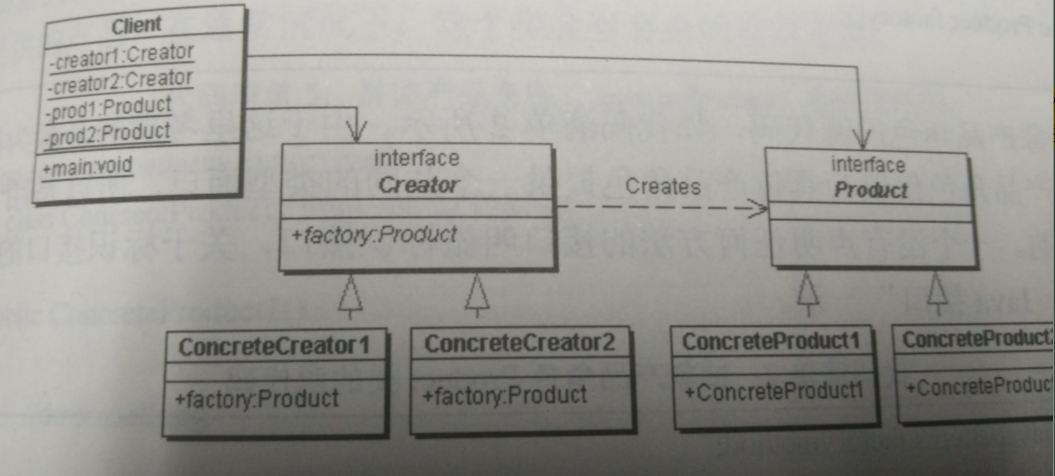
优点：实现了对责任的分割，按需创建合适的实例对象。工厂方法模式将实例化哪些对象以及如何实例化这些对象的细节隐藏起来，使得获取对象实例很方便。

缺点：将对时机的判断和对哪一种具体产品的判断逻辑混合在一起，使得系统将来进行功能扩展的时候较为困难(部分支持“开—闭原则”)。

在Java中的应用：DataFormat类实现了简单工厂模式。

1. 工厂方法模式

工厂方法模式是类的创建模式，其用意是定义一个用于创建产品对象的工厂的接口，而将实际创建工作推迟到工厂接口的子类中。属于简单工厂模式的进一步抽象和推广。



工厂方法模式有抽象工厂角色、具体工厂角色、抽象产品角色、具体产品角色。

抽象工厂角色：与应用程序无关。任何在模式中创建对象的工厂类都必须实现这个接口。

具体工厂角色：与应用程序无关，实现了抽象工厂角色，并且受到应用程序的调用用以创建产品对象。

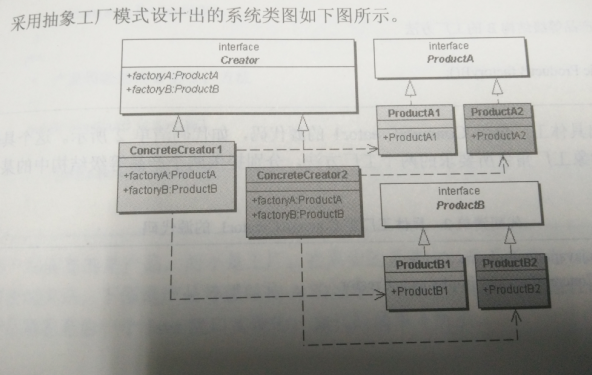
抽象产品角色：工厂方法模式所创建的对象的超类型。

具体产品角色：实现了抽象产品角色所声明的接口。工厂方法模式所创建的每一个对象都是这个具体产品角色的实例。

与简单工厂模式相比工厂方法模式把具体创建那个产品和如何创建产品的行为交给具体工厂类去实现，这样就完全支持“开闭原则”了。当增加新产品时只需要添加一个继承自抽象产品的具体产品类和一个继承自抽象工厂类的具体工厂类就可以了。

1. 抽象工厂模式

抽象工程模式是所有形态的工厂模式中最为抽象和最具一般性的一种形态。抽象工厂模式是指当有多个抽象角色时使用的一种工厂模式，抽象工厂模式可以向客户端提供一个接口，是客户端在不必指定产品的具体的情况下，创建多个产品族中的产品对象。



从上图可以看出，抽象工厂模式涉及到一下的角色。

抽象工厂角色：担任这个角色的是工厂方法模式的核心，与应用系统的商业逻辑无关，通常使用Java接口或抽象类。

具体工厂类角色：这角色直接在客户端的调用下创建产品的实例。这个产品含有选择合适的产品对象的逻辑，而这个逻辑是与应用系统的商业逻辑紧密相关的。通常使用Java类。

抽象产品角色：担任这个角色的类是工厂方法模式所创建的对象的父类，或它们共同拥有的接口。通常使用Java接口或抽象类。

具体产品角色：抽象工厂模式所创建的任何产品对象都是某一个具体产品类的实例。这是客户端最终需要的东西，其内部一定充满了应用系统的商业逻辑。

* 抽象工厂模式的适用场景

① 一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节；

② 系统的产品有多于一个的产品族(所谓产品族指的是位于不同产品等级结构中，功能相关联的产品组成的家族)，而系统只消费其中某一族的产品；

③ 同属于同一个产品族的产品是在一起使用的。

例子：对银行创建账户这个例子来说，客户对银行创建账户的具体过程并不需要了解，银行只需要按照客户的要求生成相应的账户即可。

1. [原型模式Prototype（创建型）](http://blog.csdn.net/hguisu/article/details/7518947)

* 定义

我们都知道，创建型模式一般是用来创建一个新的对象，然后我们使用这个对象完成一些对象的操作，我们通过原型模式可以快速的创建一个对象而不需要提供专门的new()操作就可以快速完成对象的创建，这无疑是一种非常有效的方式，快速的创建一个新的对象。

通过复制（克隆、拷贝）一个指定类型的对象来创建更多同类型的对象。这个指定的对象可被称为“原型”对象，也就是通过复制原型对象来得到更多同类型的对象。即原型设计模式。在php的很多模板库，都用到clone。如smarty等。

* 适用场景

当对象的构造函数非常复杂，在生成新对象的时候非常耗时间、耗资源的情况。

原型模式的主要思想是基于现有的对象克隆一个新的对象出来，一般是由对象的内部提供克隆的方法，通过该方法返回一个对象的副本，这种创建对象的方式，相比我们之前说的几类创建型模式还是有区别的，之前的讲述的工厂模式与抽象工厂都是通过工厂封装具体的new操作的过程，返回一个新的对象，有的时候我们通过这样的创建工厂创建对象不值得，特别是以下的几个场景的时候，可能使用原型模式更简单也效率更高。

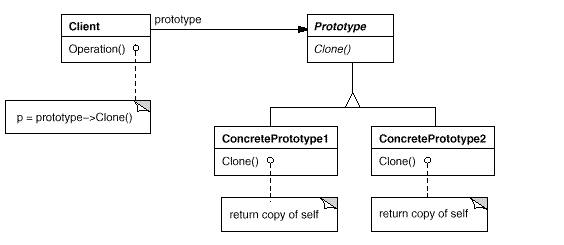
1）当一个系统应该独立于它的产品创建、构成和表示时，要使用 Prototype模式

2）当要实例化的类是在运行时刻指定时，例如，通过动态装载；

3）为了避免创建一个与产品类层次平行的工厂类层次时

4）当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。（也就是当我们在处理一些对象比较简单，并且对象之间的区别很小，可能只是很固定的几个属性不同的时候，可能我们使用原型模式更合适）。

* 类图



客户（Client）角色：使用原型对象的客户程序  
抽象原型（Prototype）角色：规定了具体原型对象必须实现的接口（如果要提供深拷贝，则必须具有实现clone的规定）

具体原型（ConcretePrototype）：从抽象原型派生而来，是客户程序使用的对象，即被复制的对象。此角色需要实现抽象原型角色所要求的接口。

1. 适配器模式Adapter\_Object (结构型)

适配器模式把一个类的接口转化为客户端所期望的另一种接口，从而使原本因接口不匹配而无法一起工作的两个类能够一起工作。适配器类可以根据所传递的参数返还一个合适的实例给客户端。

适配器模式主要适用于①希望可以复用一些现存的类，但是接口又与复用环境要求不一致“的情况，在遗留代码复用、类库迁移等方面非常有用。同时适配器有对象适配器和类适配器两种，但是类适配器采用”多继承“的实现方式，会引起程序的高耦合，所以一般不推荐使用；而对象适配器采用”对象组合“的方式，耦合度低，应用范围更广。② 两类所做的事情相同或相似，但是具有不同的接口。

优点：① 通过适配器用户可以调用统一的接口；

② 复用了现有的类，解决了现有类和复用环境要求不一致的情况；

③ 同一个适配器可以把适配者类和它的子类都适配到目标接口中。

缺点：对于对象适配器来说，更换适配器实现过程比较复杂.

1. 组合模式Composite(结构型)

组合模式将对象组合成树形结构以表示”部分—整体”的层次结构的问题。组合模式使得用户对单个对象或组合对象的使用具有一致性。

使用场景：

① 发现需求中是体现部分与整体层次的结构时；

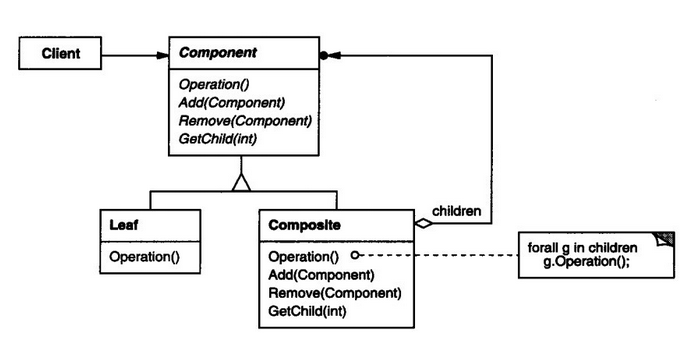
② 发现用户忽略组合对象与单一对象的不同，用户将统一地使用组合结构中的所有对象。

优点：

① 在客户端代码中，任何用到基本对象的地方都可以使用组合模式；

② 简化客户端代码，客户端可以一致地使用组合结构和单个对象；

③ 使得更加容易增加新类型的组件。



[抽象构件角色（component）](http://blog.csdn.net/hguisu/article/details/7530783)：是组合中的对象声明接口，在适当的情况下，实现所有类共有接口的默认行为。声明一个接口用于访问和管理Component子部件。

         这个接口可以用来管理所有的子对象。(可选)在递归结构中定义一个接口，用于访问一个父部件，并在合适的情况下实现它。

[树叶构件角色(Leaf)](http://blog.csdn.net/hguisu/article/details/7530783)：在组合树中表示叶节点对象，叶节点没有子节点。并在组合中定义图元对象的行为。  
          [树枝构件角色（Composite）](http://blog.csdn.net/hguisu/article/details/7530783)：定义有子部件的那些部件的行为。存储子部件。在Component接口中实现与子部件有关的操作。  
          [客户角色（Client）](http://blog.csdn.net/hguisu/article/details/7530783)：通过component接口操纵组合部件的对象。

1. 享元模式Flyweight (结构型)

* 基本概念

享元模式是设计模式中少数几个以提高系统性能为目的的模式之一。它的核心思想是：如果在一个系统中存在多个相同的对象，那么只需要共享一份对象的拷贝，而不必为每一次使用都创建新的对象。在享元模式中，由于需要构造和维护这些可以共享的对象，因此常常会出现一个工厂类用于维护和创建对象。

* 作用

享元模式的主要作用就是复用大对象(重量级对象)，以节省内存空间和对象创建时间。对性能提升的主要帮助有两点：

1. 可以节省重复创建对象的开销，因为被享元模式维护的相同对象只会被创建一次，当创建对象比较耗时时，便可以节省大量时间；
2. 由于创建对象的数量较少，所以对系统内存的需求也减小，这就使得GC的压力也相应地降低，进而使得系统拥有一个更健康的内存结构和更块的反应速度。

* 主要角色

享元模式的主要角色有享元工厂、抽象享元、具体享元类和主函数几个部分。

|  |  |
| --- | --- |
| 享元工厂 | 用以创建具体享元类，维护不同的享元对象。用于保证相同的享元对象可以被系统共享，其内部使用了类似单例模式的算法，当请求对象已经存在时，直接返回对象，不存在时，再创建对象。 |
| 抽象享元 | 定义需共享的对象的业务接口。享元类被创建出来总是为了实现某些特定的业务逻辑，而抽象享元便定义这些逻辑的语义行为。 |
| 具体享元类 | 实现抽象享元类的接口，完成某一具体逻辑。 |
| Main | 使用享元模式的组件，通过享元工厂取得享元对象。 |

享元工厂是享元模式的核心，他需要确保系统可以共享相同的对象。一般情况下，享元工厂会维护一个对象列表，当任何组件尝试获取享元类时，如果请求的享元类已经被创建了，则直接返回已有的享元类；若没有，则创建一个新的享元对象，并将它加入到维护队列中。

* 与对象池的区别

在一个对象池中，所有的对象都是等价的，任何两个对象在任何使用场景中都可以被对象那个池中的其他对象代替。而在享元模式中，享元工厂所维护的所有对象都是不同的，任何两个对象间不能相互替代。

1. 装饰者模式Decorator (结构型)

装饰者模式通过委托机制，复用系统中的各个组件，在运行时，可以将这些功能进行叠加，从而构造一个“超级对象”，使其拥有这些组件的功能。而各个子功能的模块，被很好地维护在各个组件的相关类中，拥有整洁的系统结构。

使用装饰器设计模式设计类的目标是：不必重写任何已有的功能性代码，而是对某个基于对象应用增量变化。

* 优点

装饰器设计模式采用这样的构建方式：在主代码流中应该能够直接插入一个或多个更改或“装饰”目标对象的装饰器，同时不影响其他代码流。

装饰者模式可以有效地分离性能组件和功能组件，从而提升模块的可维护性并增加模块的复用性。

Decorator模式采用对象组合而非继承的手法，实现了在运行时动态的扩展对象功能的能力，而且可以根据需要扩展多个功能，避免了单独使用继承带来的“灵活性差”和“多子类衍生问题”。同时它很好地符合面向对象设计原则中“优先使用对象组合而非继承”和“开放-封闭”原则。

* 应用场景

当系统需要添加新功能的时候，是向旧类中添加新的代码。这些新加的代码通常装饰了原有类的核心职责或主要行为。而这新加入的功能仅仅是为了满足一些只在某种特定情况下才会执行的特殊行为的需要，而装饰者却提供了一个非常好的解决方案，它把每个要装饰的功能放在单独的类中，并让这个类包装它所要包装的对象。因此，当需要执行特殊行为时，客户代码就可以在运行时根据需要地、按顺序地使用装饰功能包装对象了。

① 需要扩展一个类的功能，或给一个类增加附加责任；

② 需要动态地给一个对象增加功能，这些功能可以动态地扩展；

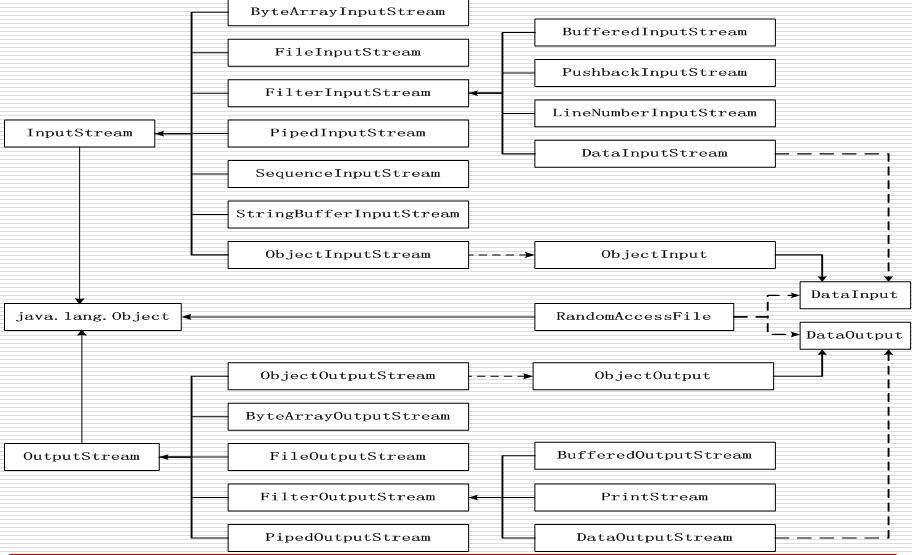
③ 需要增加一些基本功能的排列组合而产生的非常大量的功能，从而继承关系变得不现实。

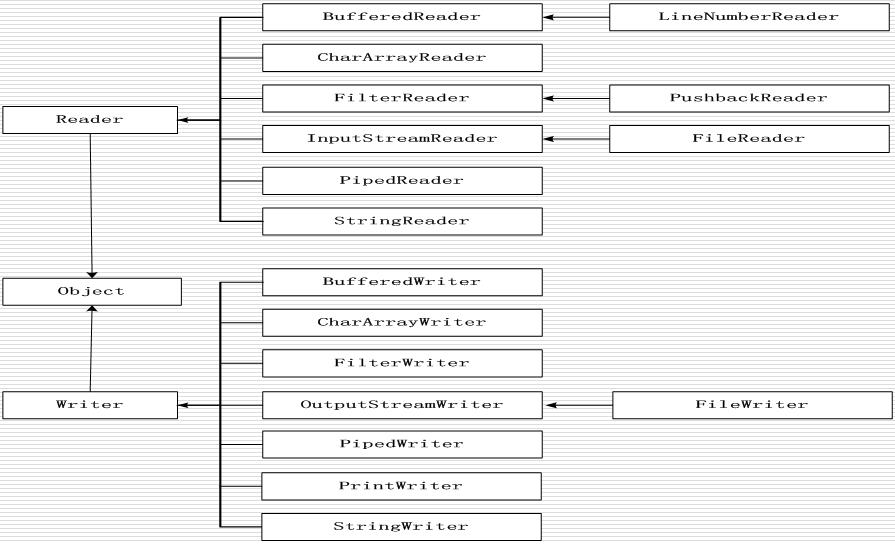
* 实际应用

在JDK的实现中，有不少组件也是用装饰者模式实现，其中，一个最典型的例子即是OutputStream和InputStream类族的实现。以OutputStream对象提供的方法比较简单，功能也比较弱，但是通过各种装饰者的增强，OutputStream对象可以被赋予强大的功能。

BufferedInputStream“装饰”了InputStream的内部工作方式，使得流的读入操作使用了缓冲机制。在使用了缓冲机制后，不会对每一次的流读入操作都产生一个物理的读盘动作，从而提高了程序的效率，在涉及到物理流的读入时，都应当使用这个装饰流类。

 Java语言的I/O库提供了四大等级结构:InputStream,OutputStream,Reader,Writer四个系列的类。InputStream和OutputStream处理8位字节流数据, Reader和Writer处理16位的字符流数据。InputStream和Reader处理输入, OutputStream和Writer处理输出，所以OutputStream,Reader,Writer这三类的装饰模式跟前面详细介绍的InputStream装饰模式大同小异，大家可以看书中其它部分对这三类的详细描述或者从网上也能找到有关资料。为了方便比较这几种类型，顺便附上Java语言的I/O层次结构图。

下面的图表示:以InputStream和OutputStream形成的层次关系

下面的图表示:以Reader和Writer形成的层次关系

1. 代理模式Proxy(结构型)

* 代理模式

为其他对象提供一种代理，并以控制对这个对象的访问。代理模式的作用是：为其他对象提供一种代理以控制对这个对象的访问。

* 代理模式的好处：

 在某些情况下，一个客户不想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。

* 代理模式一般涉及到的角色有：

抽象角色：声明真实对象和代理对象的共同接口；

代理角色：代理对象角色内部含有对真实对象的引用，从而可以操作真实对象，同时代理对象提供与真实对象相同的接口以便在任何时刻都能代替真实对象。同时，代理对象可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。

真实角色：代理角色所代表的真实对象，是我们最终要引用的对象。

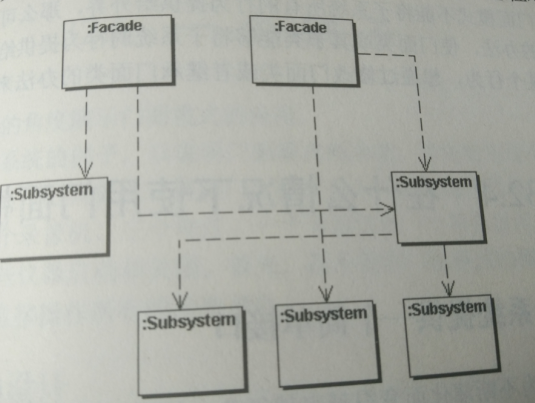
* 使用场景

1. 由于安全原因，需要屏蔽客户端直接访问真实对象；
2. 在远程调用中，需要使用代理类处理远程方法调用的技术细节(如RMI)；
3. 为了提升系统性能，对真实对象进行封装，从而达到延迟加载的目的。

延迟加载的核心思想是：如果当前并没有使用这个组件，则不需要真正地初始化它，使用一个代理对象替代它的原有的位置，只要在真正需要使用的时候，才对它进行加载。

1. 门面模式Facade(结构型)

门面模式的门面类将客户端与子系统的内部复杂性分隔开，使得客户端只需要与门面对象打交道，而不需要与子系统内部的很多对象打交道。



* 两个角色

门面(Facade)角色：客户端可以调用这个角色的方法，这个角色知道相关子系统的功能和责任。正常情况下，这个角色会将从客户端发出的请求委派到相应的子系统中。

子系统角色：可以同时有一个或者多个子系统。每一个子系统都不是一个单独的类，而是一个类的集合。每一个子系统都可以被客户端直接调用，或者被门面角色调用。子系统并不知道门面的存在，对于子系统而言，门面仅仅是另外一个客户端而已。

* 使用门面模式的场景

① 为一个复杂子系统提供简单接口：使用门面模式可以使得子系统更具可复用性；

② 子系统独立，与其他系统相隔离开：引入门面模式的情况下可以将一个子系统与它的客户端以及其他的子系统相分离，提高子系统的独立性和可移植性；

③ 层次化结果：在构建一个层次化的系统时，可以使用门面模式定义系统中每一层的入口。如果层与层之间是相互依赖的，则可以限定它们仅通过门面模式进行通信，从而简化了层与层之间的依赖关系。

1. 观察者模式Observer(行为型)

观察者模式(也被称为发布\订阅模式)提供了避免组件之间紧密耦合的另一种方法，它将观察者和被观察者对象分开。在这个模式中，一个对象提供一种方法允许其他对象通过注册这个方法来观察自己。当被观察者的对象更改时，它会将消息发送到已注册的观察者。这些观察者使用该信息执行的操作与可操作的对象无关，结果是对象可以相互对话，而不必了解原因。

例如，用户界面可以是观察者，业务数据是被观察者，用户界面观察业务数据的变化，发现数据变化时，就显示在界面上。

面向对象设计的一个原则是：系统中的每个类都将重点放在某一个功能上，而不在其他方面。一个对象只做一件事情，并且将它做好。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。

使用场景：

①当一个对象的改变需要改变其他对象时，而且它不知道具体有多少个对象需要改变；

②一个抽象类型有两个方面，一个方面依赖于另一个方面，这是用观察者模式可以将这两两封装在独立的对象中使它们各自独立地改变和使用。

优点：

①观察者模式接触了主体和具体观察者之间的耦合，让耦合的双方都依赖于抽象模式，而不是依赖于具体。从而使得各自的变化都不会影响另一边的变化。

②实现了表示层和梳理逻辑的分离，并定义了稳定的更新消息传递机制，类别清晰，并抽象了更新接口，使得可以有各种各样不同的表示层(表示层)。

1. 责任链模式Chain of Resonsibility(行为型)

http://blog.csdn.net/hguisu/article/details/7547231

* 概念

在职责链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织链和分配责任。

* 优缺点

职责链模式的主要优点在于

① 可以降低系统的耦合度;

② 简化对象的相互连接;

③ 同时增强给对象指派职责的灵活性;

④ 增加新的请求处理类也很方便.

其主要缺点

① 在于不能保证请求一定被接收;

② 且对于比较长的职责链，请求的处理可能涉及到多个处理对象，系统性能将受到一定影响，而且在进行代码调试时不太方便。

* **分类：**

**纯的职责链模式**：一个具体处理者角色处理只能对请求作出两种行为中的一个：一个是自己处理（承担责任），另一个是把责任推给下家。不允许出现某一个具体处理者对象在承担了一部分责任后又将责任向下传的情况。请求在责任链中必须被处理，不能出现无果而终的结局。

**反之就是不纯的职责链模式。**

在一个纯的职责链模式里面，一个请求必须被某一个处理者对象所接收；在一个不纯的职责链模式里面，一个请求可以最终不被任何接收端对象所接收。

* **适用性**

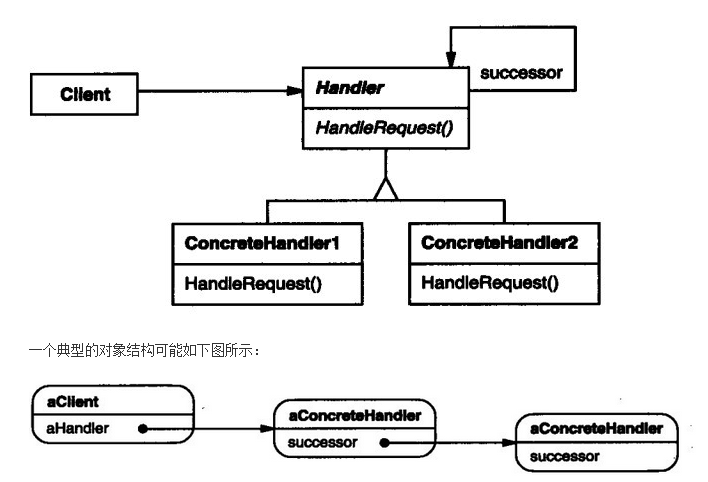
在以下条件下使用Responsibility 链：

• 有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定。

• 你想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。

• 可动态指定一组对象处理请求。

* 结构组成



抽象处理者角色(Handler:Approver):定义一个处理请求的接口，和一个后继连接(可选)

具体处理者角色(ConcreteHandler:President):处理它所负责的请求，可以访问后继者，如果可以处理请求则处理，否则将该请求转给他的后继者。

1. 命令模式Command(行为型)

在软件设计中，经常需要向某些对象发送请求，但是并不知道请求的接受者是谁，也不知道被请求的操作是哪个，我们只需在程序运行时指定具体的请求接受者即可，此时，可以使用命令模式来进行设计，使得请求发送者与请求接收者消除彼此之间的耦合，让对象之间的调用关系更加灵活。

* 定义：

将一个请求封装成一个对象，从而使我们可用不同的请求对客户进行参数化，对请求排队或者记录请求日志，以及支持可撤销的操作。

适用性：

①系统需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互。

②系统需要在不同的时间指定请求、将请求排队和执行请求。

③系统需要支持命令的撤销(Undo)操作和恢复(Redo)操作。

④系统需要将一组操作组合在一起，即支持宏命。

* 优点：

1) 降低系统的耦合度:Command模式将调用操作的对象与知道如何实现该操作的对象解耦。

2) Command是头等的对象。它们可像其他的对象一样被操纵和扩展。

3) 组合命令:你可将多个命令装配成一个组合命令，即可以比较容易地设计一个命令队列和宏命令。一般说来，组合命令是Composite模式的一个实例。

4) 增加新的Command很容易，因为这无需改变已有的类。

5）可以方便地实现对请求的Undo和Redo。

* 总结：

①[命令模式](http://blog.csdn.net/hguisu/article/details/7549895)的本质是对命令进行封装，将发出命令的责任和执行命令的责任分割开。

②每一个命令都是一个操作：请求的一方发出请求，要求执行一个操作；接收的一方收到请求，并执行操作

③[命令模式](http://blog.csdn.net/hguisu/article/details/7549895)允许请求的一方和接收的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否被执行、何时被执行，以及是怎么被执行的。

④命令模式使请求本身成为一个对象，这个对象和其他对象一样可以被存储和传递。

⑤命令模式的关键在于引入了抽象命令接口，且发送者针对抽象命令接口编程，只有实现了抽象命令接口的具体命令才能与接收者相关联。

1. 模板方法Template Method(行为型)

解决的问题：对于某一个业务逻辑(算法实现)在不同的对象中有不同的细节，但是逻辑(算法)的框架是相同的。

解决方案：模板方法模式采用继承的方式解决问题，将逻辑(算法)框架写在抽象基类中，并定好细节的接口，子类实现细节。

优点：模板方法模式是通过把不变行为搬移到超类，去除子类中的重复代码。

举例：汽车加工厂加工汽车：基本步骤都是组装车头、组装车身、组装车尾、测试组装的车体，不管被组装的是吉普车、卡车还是公交车，它们的基本组装步骤都是一样的，虽然组装的步骤中会有细微的差别。

1. 迭代器模式Iterator(行为型)

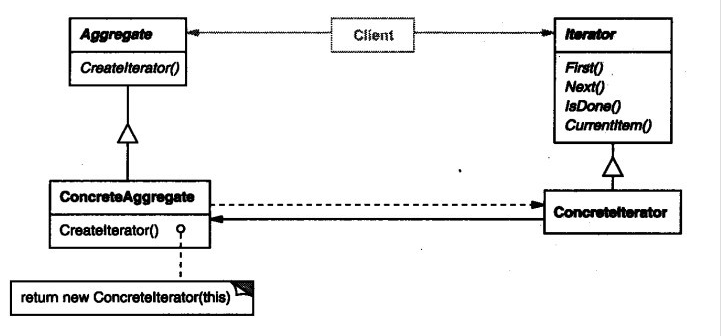
* 定义

提供一种方法顺序访问一个聚合对象中各个元素，而又不暴露该对象的内部表示。又叫做游标（Cursor）模式 。也就是说为不同的聚集结构提供如开始、下一个、是否结束、当前哪一项等统一的接口。

* 使用场景

① 访问一个聚合对象的内容而无需暴露它的内部表示。  
② 需要为聚合对象提供多种遍历方式。  
③ 为遍历不同的聚合结构提供一个统一的接口 (即, 支持多态迭代)

* 类图



抽象迭代器(Iterator): 迭代器定义访问和遍历元素的接口。  
具体迭代器(ConcreteIterator):  具体迭代器实现迭代器Iterator接口。对该聚合遍历时跟踪当前位置。  
抽象聚合类(Aggregate): 聚合定义创建相应迭代器对象的接口。  
具体聚合类(ConcreteAggregate): 体聚合实现创建相应迭代器的接口，该操作返回ConcreteIterator的一个适当的实例。

* 优点

总体来说迭代器模式就是分离了集合对象的遍历行为，抽象出一个迭代器来负责，这样既可以做到不暴露集合的内部结构，又可让外部代码透明地访问集合内部的数据。

* 备注

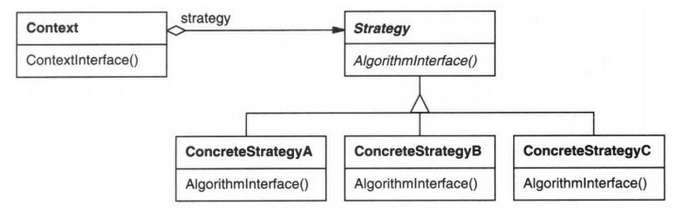
迭代器模式在访问数组、集合、列表等数据时，尤其是数据库数据操作时，是非常普遍的应用，但是由于它太普遍了，所以各种高级语言都对它进行了封装，所以反而给人感觉此模式本身不太实用了。

1. 策略模式Strategy(行为型)

* 定义

策略模式定义了算法家族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化不会影响到客户端。从概念上看，这些算法完成的功能都是一样的，只不过是具体的实现不一样，使用策略模式可以让相同的方式调用所有的算法，减少了各种算法与使用算法类之间的耦合。

* 策略模式的类图



* 优点

① 简化了单元测试，因为每个算法都有自己的类，可以通过自己的接口单独测试。

② 简化了客户端代码，消除了多个if-else的判断；

③ 实现的选择 Strategy模式可以提供相同行为的不同实现。客户可以根据不同时间/空间权衡取舍要求从不同策略中进行选择。

* 缺点

① 会产生很多的策略类，每个策略都需要对应一个策略类；

② 提前需要知道有哪些策略，需要从中选择，指定最终执行的策略。

* 适用场景

① 许多相关的类仅仅是行为有异。“策略”提供了一种用多个行为中的一个行为来配置一个类的方法。即一个系统需要动态地在几种算法中选择一种。  
② 需要使用一个算法的不同变体。例如，你可能会定义一些反映不同的空间 /时间权衡的算法。当这些变体实现为一个算法的类层次时 ,可以使用策略模式。  
③ 算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数据结构。  
④ 一个类定义了多种行为 , 并且这些行为在这个类的操作中以多个条件语句的形式出现。将相关的条件分支移入它们各自的Strategy类中以代替这些条件语句。

1. 状态模式State(行为型)

* 定义

当一个对象的内在状态改变时允许改变其行为，这个对象看起来像是改变了其类。

* 主要解决的问题

当控制一个对象状态转换的条件表达式过于复杂时的情况。把状态的判断逻辑转移到不同状态的一系列类中，可以把复杂的判断逻辑简化。

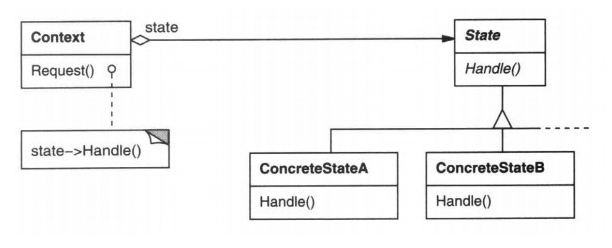
* 优点

① 将于特定状态相关的行为局部化，并且将不同状态的行为分割开来。

② 通过把各种状态转移逻辑分布到State的子类之间，来减少相互间的依赖。

* 适用场景

当一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为时，就可以考虑使用状态模式了。



环境类（Context）:  定义客户感兴趣的接口。维护一个ConcreteState子类的实例，这个实例定义当前状态。  
抽象状态类（State）:  定义一个接口以封装与Context的一个特定状态相关的行为。

具体状态类（ConcreteState）:  每一子类实现一个与Context的一个状态相关的行为。

1. 责任链模式和命令模式的区别

* 职责链模式

链的组织是从最特殊的到最一般的，并且不能保证请求在任何情况下都回有相应。职责链将程序中每个对象能做什么的内容隔离，即职责链减少了对象之间的耦合，每个对象都能独立操作。职责链也可用于构成主程序的对象和包含其它对象实例的对象。

适用场景：

1 具有相同方法的几个对象都适合于执行程序请求操作，但由对象决定由谁去完成操作，比把决策建立在调用代码中更合适；

2 其中某个对象可能最适合处理请求，但你不想通过一些列if-else语句或switch语句去选择一个特定的对象；

3 程序执行时，需要向处理选项链中添加新的对象；

4 在多个对象都能执行一个请求的情况下，你不想把这些相互作用的内容放在调用程序里，链中每个对象都是“自治”的，最后一个对象决定是默认处理请求，还是抛弃。

* 命令模式

职责链沿类链转发请求，而命令模式只将请求转发给一个特定对象。命令模式把一个申请特定操作的请求封装到一个对象中，并给该对象一个众所周知的公共接口，使客户端不用了解实际执行的操作就能产生请求，也可以使你改变操作而丝毫不影响客户端程序。

命令模式的主要缺点是，增加了使程序散乱的小类，不过，即使有单独的单击事件，也通常都调用小的私有方法完成具体功能。最后的结果是，私有方法和我们这些小类的代码长度几乎一样，因此，构建 Command类和编写较多的方法在复杂性上通常没有区别，主要区别是命令模式生成的小类更容易理解。

使用命令设计模式的另一个主要原因是，他们提供了一个便捷的存储方法并能完成Undo功能。每个命令对象都记住刚刚做过的事，并在有Undo请求时，只要计算量和内存需求不太过分，就能恢复到刚才的状态。

1. 责任链模式和状态模式的区别

职责链模式和状态模式都可以解决If分支语句过多，  
从定义来看，状态模式是一个对象的内在状态发生改变（一个对象，相对比较稳定，处理完一个对象下一个对象的处理一般都已确定），  
而职责链模式是多个对象之间的改变（多个对象之间的话，就会出现某个对象不存在的现在，就像我们举例的公司请假流程，经理可能不在公司情况），这也说明他们两个模式处理的情况不同。  
这两个设计模式最大的区别就是状态模式是让各个状态对象自己知道其下一个处理的对象是谁。  
而职责链模式中的各个对象并不指定其下一个处理的对象到底是谁，只有在客户端才设定。  
用我们通俗的编程语言来说，就是  
**状态模式：**  
  相当于If else if else；  
  设计路线：各个State类的内部实现(相当于If，else If内的条件)  
  执行时通过State调用Context方法来执行。  
**职责链模式：**  
  相当于Swich case  
  设计路线：客户设定，每个子类(case)的参数是下一个子类(case)。  
  使用时，向链的第一个子类的执行方法传递参数就可以。  
就像对设计模式的总结，有的人采用的是状态模式，从头到尾，提前一定定义好下一个处理的对象是谁，而我采用的是职责链模式，随时都有可能调整链的顺序。

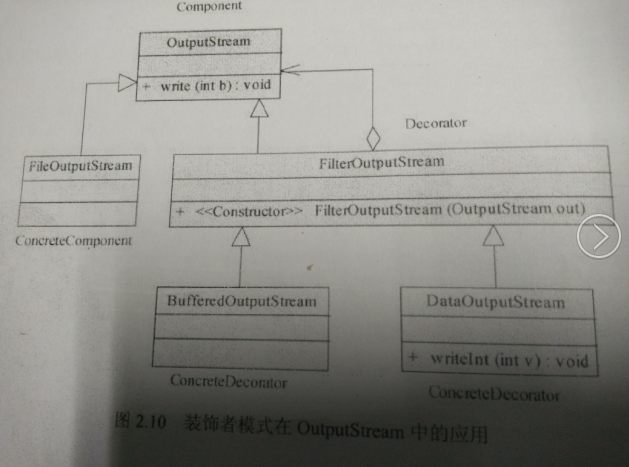
1. 责任链模式和策略模式的区别

[策略模式](http://blog.csdn.net/hguisu/article/details/7558249)：（http://www.cnblogs.com/Mainz/archive/2007/12/15/996081.html）（状态模式是策略模式的孪生兄弟）  
        状态模式和策略模式的实现方法非常类似，都是利用多态把一些操作分配到一组相关的简单的类中，因此很多人认为这两种模式实际上是相同的。  
然而在现实世界中，策略（如促销一种商品的策略）和状态（如同一个按钮来控制一个电梯的状态，又如手机界面中一个按钮来控制手机）是两种完全不同的思想。当我们对状态和策略进行建模时，这种差异会导致完全不同的问题。例如，对状态进行建模时，状态迁移是一个核心内容；然而，在选择策略时，迁移与此毫无关系。另外，策略模式允许一个客户选择或提供一种策略，而这种思想在状态模式中完全没有。  
       一个策略是一个计划或方案，通过执行这个计划或方案，我们可以在给定的输入条件下达到一个特定的目标。策略是一组方案，他们可以相互替换；选择一个策略，获得策略的输出。策略模式用于随不同外部环境采取不同行为的场合。我们可以参考微软企业库底层Object Builder的创建对象的strategy实现方式。而状态模式不同，对一个状态特别重要的对象，通过状态机来建模一个对象的状态；状态模式处理的核心问题是状态的迁移，因为在对象存在很多状态情况下，对各个business flow，各个状态之间跳转和迁移过程都是及其复杂的。  
       例如一个工作流，审批一个文件，存在新建、提交、已修改、HR部门审批中、老板审批中、HR审批失败、老板审批失败等状态，涉及多个角色交互，涉及很多事件，这种情况下用状态模式(状态机)来建模更加合适；把各个状态和相应的实现步骤封装成一组简单的继承自一个接口或抽象类的类，通过另外的一个Context来操作他们之间的自动状态变换，通过event来自动实现各个状态之间的跳转。在整个生命周期中存在一个状态的迁移曲线，这个迁移曲线对客户是透明的。我们可以参考微软最新的WWF 状态机工作流实现思想。  
      在状态模式中，状态的变迁是由对象的内部条件决定，外界只需关心其接口，不必关心其状态对象的创建和转化；  
而策略模式里，采取何种策略由外部条件(C)决定。  
      他们应用场景（目的）却不一样，State模式重在强调对象内部状态的变化改变对象的行为，Strategy模式重在外部对策略的选择，策略的选择由外部条件决定，  
也就是说算法的动态的切换。但由于它们的结构是如此的相似，我们可以认为“状态模式是完全封装且自修改的策略模式”。即状态模式是封装对象内部的状态的，而策略模式是封装算法族的。

1. 装饰者模式与代理模式的区别

① 代理模式并不实现和目标对象类似的业务功能，而是一些与目标对象功能联系不太紧密的职责，也就是说二者处理的是不同的问题域，比如权限控制、连接服务端、处理异常、缓存目标对象等这些域目标对象的功能不是同一类的问题域；而装饰者对象处理的问题与目标对象处理的问题都是同一问题，目的是增加目标对象那方面的能力，例如JavaIO的BufferedInputStream和修饰对象InputStream都是为实现流读取设计的，它使得InputStream可以缓冲读取数据。

② 装饰者模式动态地为对象添加功能或者减少功能，还可以递归地装饰目标对象，产生叠加的效果，而代理模式不做此用途。



1. 详解如何实现单例模式singleton

#### 10.1 立即加载/恶汉加载

这种方式是在类中定义了一个静态的成员变量single直接让其实例化，这样的话在加载这个类的时候就直接把这个静态变量初始化了。

好处是直接初始化静态变量single使得代码不管是在多线程还是单线程模式在都是只有一个变量。

缺点是一旦单例类被加载，就会建立实例，从而降低内存的使用效率。

虽然实现单例模式的立即加载会过早建立实例，但是却可以保证多线程下是安全的，也是值得的，所以多线程环境下这个方式在剑指offer书上是强烈推荐的。

|  |
| --- |
| /\*\*  \* 单例模式，恶汉式加载即立即加载。  \* 在使用类的时候已经将对象创建完毕，常见的实现办法就是直接new实例化。  \*/  **public** **class** Single1 {  **private** **static** Single1 *single* = **new** Single1();  **private** Single1(){  System.*out*.println("创建单例");  }  **public** Single1 getInstance(){  **return** *single*;  }  } |
| /\*\*  \* 多线程环境下的立即加载。  \*/  public class TestThreadSingle implements Runnable {  public void run(){  Single1 s = Single1.getInstance();  System.out.println("s = "+s.hashCode());  try {  Thread.sleep(100000);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  public static void main(String[] args){  TestThreadSingle t1 = new TestThreadSingle();  TestThreadSingle t2 = new TestThreadSingle();  Thread t11 = new Thread(t1);  Thread t22 = new Thread(t2);  t11.start();  t22.start();  }  } |

#### 10.2 延迟加载\懒汉模式

延迟加载是在调用getInstance方法的时候实例才被创建，把给single创建其指向的对象这个动作延迟到了getInstance方法内部。

好处是：在需要调用单例的对象single时才进行创建实例的操作，相比较于立即加载节省了空间，只有在需要的时候才创建单例对象。

缺点是：在多线程环境中，就可能会有问题，比如线程1和线程2同时执行getInstance方法，在执行判断语句的时候得到的single==null，那么这两个线程就都会创建一个实例，这样得到的单例对象就不止一个了。

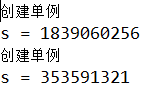
决策：延迟加载适用于单线程环境下，但是不适用于多线程环境下。

|  |
| --- |
| /\*\*  \* 实现单例模式的延迟加载  \*/  **public** **class** Single2 {  **private** **static** Single2 *single* = **null**;  **private** Single2(){  System.*out*.println("创建单例");  }  **public** **static** Single2 getInstance(){  **if**(*single* == **null**){  *single* = **new** Single2();  }  **return** *single*;  }  **public** **static** **void** main(String[] args){  Single2.*getInstance*();  }  } |

多线程环境下的懒汉加载：

|  |
| --- |
| **public** **class** TestThreadSingle **implements** Runnable {  **public** **void** run(){  Single2 s = Single2.*getInstance*();  System.*out*.println("s = "+s.hashCode());  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }    **public** **static** **void** main(String[] args){  TestThreadSingle t1 = **new** TestThreadSingle();  TestThreadSingle t2 = **new** TestThreadSingle();  Thread t11 = **new** Thread(t1);  Thread t22 = **new** Thread(t2);  t11.start();  t22.start();  }  } |

输出为：



#### 10.3 延迟加载的解决方案：同步getInstance方法或同步相应代码段

同步getInstance方法：

|  |
| --- |
| /\*\*  \* 为保证多线程下是安全的，所以在这个地方同步getInstance方法，  \* 但是设置同步方法之后代码效率会降低。  \*  \*/  **public** **class** Single3 {  **private** **static** Single3 *single* = **null**;  **private** Single3(){  System.*out*.println("创建单例");  }  **synchronized** **public** **static** Single3 getInstance(){  **if**(*single* == **null**){  *single* = **new** Single3();  }  **return** *single*;  }  } |

同步必要的代码段：

|  |
| --- |
| /\*\*  \* 指同步getInstance方法中必要的代码段，但是同步虽然可以达到只保存一个实例，但是却严重影响了效率。  \* @author zk  \*  \*/  public class Single4 {  private static Single4 single = null;  private Single4(){  System.out.println("创建单例");  }  public static Single4 getInstance(){  synchronized(Single4.class){  if(single == null){  single = new Single4();  }  }  return single;  }  } |

#### 10.4延迟加载的解决方案：使用DCL双检查锁机制

|  |
| --- |
| /\*\*  \* 使用DCL(Double Check Locking)双检查锁机制  \* 好处，比直接同步代码提高效率，但是这样的代码实现起来比较复杂，容易出错。  \*/  **public** **class** Single5 {  **private** **static** Single5 *single* = **null**;  **private** Single5(){  System.*out*.println("创建单例");  }  **public** **static** Single5 getInstance(){    **if**(*single* == **null**){//第一次检查上锁  **synchronized**(Single5.**class**){  **if**(*single* == **null**){//第二次检查是否创建  *single* = **new** Single5();  }  }  }  **return** *single*;  }  } |

#### 10.5 使用静态内部类实现单例模式

在单例类内的内部类中，创建一个静态内部类 ，在静态内部类中创建单例对象，只有调用getInstance()方法时才会加载这个类，初始化single对象，就做到了在需要时才创建单例对象。

这样就既可以使用立即加载满足多线程条件下只有一个实例的好处，也可以使用延迟加载的按需创建单例对象的好处。

|  |
| --- |
| /\*\*  \* 创建一个内部类达到需要时才加载。  \* 在单例类内的内部类中，创建一个静态内部类 ，在静态内部类中创建单例对象，  \* 只有调用getInstance()方法时才会加载这个类，初始化single对象，就做到了在需要时才创建单例对象。  \*/  **public** **class** Single6 {  /\*建一个静态内部类,只有第一次使用这个静态类的时候才会进行加载然后初始化这个single对象。\*/  **private** **static** **class** MySingle{  **private** **static** Single6 *single* = **new** Single6();  }  **private** Single6(){  System.*out*.println("创建单例");  }  **public** **static** Single6 getInstance(){  **return** MySingle.*single*;  }  **public** **static** **void** main(String[] args){    }  } |

1. Jdk中体现的设计模式

<http://blog.jobbole.com/62314/>

比如：IO部分体现的是装饰者模式和适配器模式；容器中有迭代器模式；clone对象的时候用到了原型模式；

**命令模式**

将操作封装到对象内，以便存储，传递和返回。

* + - java.lang.Runnable
    - javax.swing.Action

1. singleton bean和singleton模式区别

Spring 项目加载的时候bean（一个bean对应某个类）自动创建（初始化，建一个实例），而后是每次调用bean的时候是注入的（不是重新new。所有整个系统都是这个实例，，而且是spring自动提供的）

① 对于Spring中实现Singleton模式，是以IOC容器为单位（就是说在这个容器里面bean实现Singleton），换句话说，一个JVM可能有多个IOC容器，而在这个容器里实现了singleton bean。而Java中实现Singleton模式而言，只有一个JVM，jvm中某个class只有一个实例。

② singleton模式中，singleton的class在整个JVM中只有一个instance，Spring的Bean，你可以一个class配置多个Bean，这个class就有了多个instance。 这个singleton是指在单个Spring容器中，这个Bean是单实例的，是线程共享的，所以要求这些类都是线程安全的。也就是说，不能出现修改Bean属性的方法，当然除了设值的那些setter。只要满足线程安全，这些bean都可以用singleton。而且我们在绝大多数使用上，也是这样用的，包括dao,service。 Beanfactory是Spring初始以静态方式载入的,Spring的单例IOC是基于容器级的,所以这你都不用担心与考虑。