反射技术大量用于Java设计模式和框架技术，最常见的设计模式就是工厂模式(Factory)和单例模式(Singleton)。   
  
单例模式(Singleton)   
  
       这个模式主要作用是保证在Java应用程序中，一个类Class只有一个实例存在。在很多操作中，比如建立目录 [**数据库**](http://lib.csdn.net/base/14)连接都需要这样的单线程操作。这样做就是为了节省内存空间，保证我们所访问到的都是同一个对象。   
  
       单例模式要求保证唯一，那么怎么样才能保证唯一性呢？对了，这就是静态变量。单例模式有以下两种形式：   
  
第一种形式：   
package reflect;   
public class Singleton {   
    private Singleton() {   
    }   
    private static Singleton instance = new Singleton();   
  
    // 这里提供了一个供外部访问本class的静态方法，可以直接访问   
    public static Singleton getInstance() {   
           return instance;   
    }   
}   
class SingRun{   
    public static void main(String[] args){   
       //这样的调用不被允许，因为构造方法是私有的。   
       //Singleton x=new Singleton();   
        
       //得到一个Singleton类实例   
       Singleton x=Singleton.getInstance();   
        
       //得到另一个Singleton类实例   
       Singleton y=Singleton.getInstance();   
        
       //比较x和y的地址，结果为true。说明两次获得的是同一个对象   
       System.out.println(x==y);   
    }   
}   
  
第二种形式：   
  
public class Singleton {   
  
    //先申明该类静态对象   
    private static Singleton instance = null;   
     
    //创建一个静态访问器，获得该类实例。加上同步，表示防止两个线程同时进行对象的创建   
    public static synchronized Singleton getInstance() {   
        
       //如果为空，则生成一个该类实例   
       if (instance == null){   
           instance = new Singleton();   
       }   
       return instance;   
    }   
  
}   
工厂模式（Factory）   
  
       工厂模式是我们最常用的模式了,著名的Jive论坛 ,就大量使用了工厂模式，工厂模式在Java程序系统可以说是随处可见。   
  
为 什么工厂模式是如此常用？是因为工厂模式利用Java反射机制和Java多态的特性可以让我们的程序更加具有灵活性。用工厂模式进行大型项目的开发，可 以很好的进行项目并行开发。就是一个程序员和另一个程序员可以同时去书写代码，而不是一个程序员等到另一个程序员写完以后再去书写代码。其中的粘合剂就是 接口和配置文件。   
之前说利用接口可以将调用和实现相分离。   
那么这是怎么样去实现的呢？工厂模式可以为我们解答。   
  
我 们先来回顾一下软件的生命周期，分析、设计、编码、调试与测试。其中分析就是指需求分析，就是知道这个软件要做成什么样子，要实现什么样的功能。功能知 道了，这时就要设计了。设计的时候要考虑到怎么样高效的实现这个项目，如果让一个项目团队并行开发。这时候，通常先设计接口，把接口给实现接口的程序员和 调用接口的程序员，在编码的时候，两个程序员可以互不影响的实现相应的功能，最后通过配置文件进行整合。   
  
  
代码示例：   
  
  
  
interface InterfaceTest{   
    public void getName();//定义获得名字的方法   
}   
  
  
接口有了，那么得到这个接口，进行实现编码的程序员应该怎么做呢？对了，实现这个接口，重写其中定义的方法   
  
接口实现方：   
  
class Test1 implements InterfaceTest{   
     
     
    public void getName() {   
       System.out.println("test1");   
    }   
     
}   
  
  
class Test2 implements InterfaceTest{   
     
     
    public void getName() {   
       System.out.println("test2");   
    }   
     
}   
  
大家可以发现，当接口定义好了以后，不但可以规范代码，而且可以让程序员有条不紊的进行功能的实现。实现接口的程序员根本不用去管，这个类要被谁去调用。   
  
那么怎么能获得这些程序员定义的对象呢？在工厂模式里，单独定义一个工厂类来实现对象的生产，注意这里返回的接口对象。   
  
工厂类，生产接口对象：   
  
class Factory{   
    //创建私有的静态的Properties对象   
    private static Properties pro=new Properties();   
     
    //静态代码块   
    static{   
       try {   
            
           //加载配置文件   
           pro.load(new FileInputStream("file.txt"));   
       } catch (Exception e) {   
           e.printStackTrace();   
       }   
    }   
     
     
    private static Factory factory=new Factory();   
    private Factory(){}   
     
    public static Factory getFactory(){   
       return factory;   
    }   
     
     
    public  InterfaceTest getInterface(){   
       InterfaceTest interfaceTest=null;//定义接口对象   
        
       try {   
           //根据键，获得值，这里的值是类的全路径   
           String classInfo=pro.getProperty("test");   
            
           //利用反射，生成Class对象   
           Class c=Class.forName(classInfo);   
            
           //获得该Class对象的实例   
           Object obj=c.newInstance();   
            
           //将Object对象强转为接口对象   
           interfaceTest=(InterfaceTest)obj;   
       } catch (Exception e) {   
           e.printStackTrace();   
       }   
        
       //返回接口对象   
       return interfaceTest;   
    }   
}   
  
配置文件内容：   
test=factory.Test2   
  
通 过这个类，大家可以发现，在调用的时候，得到的是个接口对象。而一个接口变量可以指向实现了这个接口的类对象。在利用反射的时候，我们并没有直接把类的 全路径写出来，而是通过键获得值。这样的话，就有很大的灵活性，只要改变配置文件里的内容，就可以改变我们调用的接口实现类，而代码不需做任何改变。在调 用的时候，我们也是通过接口调用，甚至我们可以连这个接口实现类的名字都不知道。   
  
调用方：   
  
public class FactoryTest {   
  
    public static void main(String[] args) {   
       //获得工厂类的实例   
       Factory factory=Factory.getFactory();   
       //调用获得接口对象的方法，获得接口对象   
       InterfaceTest inter=factory.getInterface();   
       //调用接口定义的方法   
       inter.getName();   
    }   
}   
上 面的代码就是调用方法。大家可以发现，在调用的时候，我们根本没有管这个接口定义的方法要怎么样去实现它，我们只知道这个接口定义这个方法起什么作用就 行了。上面代码运行结果要根据配置文件来定。如果配置文件里的内容是test=factory.Test2。那么表示调用factory.Test2这个 类里实现接口的方法，这时候打印“test2”。如果配置文件里的内容是test=factory.Test1。那么表示调用factory.Test1 这个类里实现接口的方法，这时候打印“test1”。   
反 射机制是框架技术的原理和核心部分。通过反射机制我们可以动态的通过改变配置文件(以后是XML文件)的方式来加载类、调用类方法，以及使用类属性。这 样的话，对于编码和维护带来相当大的便利。在程序进行改动的时候，也只会改动相应的功能就行了，调用的方法是不用改的。更不会一改就改全身。

59.设计模式：

·

## ****一、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。用一个图片来整体描述一下：



## ****二、设计模式的六大原则****

### ****总原则：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，而是要扩展原有代码，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等，后面的具体设计中我们会提到这点。

### 1、单一职责原则

不要存在多于一个导致类变更的原因，也就是说每个类应该实现单一的职责，如若不然，就应该把类拆分。

### ****2、里氏替换原则（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。—— From Baidu 百科

历史替换原则中，子类对父类的方法尽量不要重写和重载。因为父类代表了定义好的结构，通过这个规范的接口与外界交互，子类不应该随便破坏它。

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

这个是开闭原则的基础，具体内容：面向接口编程，依赖于抽象而不依赖于具体。写代码时用到具体类时，不与具体类交互，而与具体类的上层接口交互。

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

这个原则的意思是：每个接口中不存在子类用不到却必须实现的方法，如果不然，就要将接口拆分。使用多个隔离的接口，比使用单个接口（多个接口方法集合到一个的接口）要好。

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

就是说：一个类对自己依赖的类知道的越少越好。也就是说无论被依赖的类多么复杂，都应该将逻辑封装在方法的内部，通过public方法提供给外部。这样当被依赖的类变化时，才能最小的影响该类。

最少知道原则的另一个表达方式是：只与直接的朋友通信。类之间只要有耦合关系，就叫朋友关系。耦合分为依赖、关联、聚合、组合等。我们称出现为成员变量、 方法参数、方法返回值中的类为直接朋友。局部变量、临时变量则不是直接的朋友。我们要求陌生的类不要作为局部变量出现在类中。

### ****6、合成复用原则（Composite Reuse Principle）****

原则是尽量首先使用合成/聚合的方式，而不是使用继承。

## ****三、Java的23中设计模式****

## ****A、创建模式****

从这一块开始，我们详细介绍[**Java**](http://lib.csdn.net/base/17)中23种设计模式的概念，应用场景等情况，并结合他们的特点及设计模式的原则进行分析。

首先，简单工厂模式不属于23中涉及模式，简单工厂一般分为：普通简单工厂、多方法简单工厂、静态方法简单工厂。

### ****0、简单工厂模式****

简单工厂模式模式分为三种：

#### 01、普通

就是建立一个工厂类，对实现了同一接口的一些类进行实例的创建。首先看下关系图：



举例如下：（我们举一个发送邮件和短信的例子）

首先，创建二者的共同接口：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **interface** Sender {
2. **public** **void** Send();
3. }

其次，创建实现类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **class** MailSender **implements** Sender {
2. @Override
3. **public** **void** Send() {
4. System.out.println("this is mailsender!");
5. }
6. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **class** SmsSender **implements** Sender {
3. @Override
4. **public** **void** Send() {
5. System.out.println("this is sms sender!");
6. }
7. }

最后，建工厂类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **class** SendFactory {
3. **public** Sender produce(String type) {
4. **if** ("mail".equals(type)) {
5. **return** **new** MailSender();
6. } **else** **if** ("sms".equals(type)) {
7. **return** **new** SmsSender();
8. } **else** {
9. System.out.println("请输入正确的类型!");
10. **return** **null**;
11. }
12. }
13. }

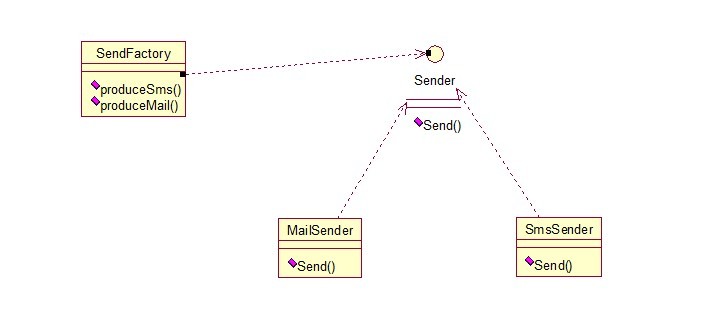
我们来测试下：

1. **public** **class** FactoryTest {
3. **public** **static** **void** main(String[] args) {
4. SendFactory factory = **new** SendFactory();
5. Sender sender = factory.produce("sms");
6. sender.Send();
7. }
8. }

输出：this is sms sender!

#### *****02、多个方法*****

是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。关系图：



将上面的代码做下修改，改动下SendFactory类就行，如下：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)**public** **class** SendFactory {

**public** Sender produceMail(){

1. **return** **new** MailSender();
2. }
4. **public** Sender produceSms(){
5. **return** **new** SmsSender();
6. }
7. }

测试类如下：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **class** FactoryTest {
3. **public** **static** **void** main(String[] args) {
4. SendFactory factory = **new** SendFactory();
5. Sender sender = factory.produceMail();
6. sender.Send();
7. }
8. }

输出：this is mailsender!

#### *****03、多个静态方法*****

将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **class** SendFactory {
3. **public** **static** Sender produceMail(){
4. **return** **new** MailSender();
5. }
7. **public** **static** Sender produceSms(){
8. **return** **new** SmsSender();
9. }
10. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

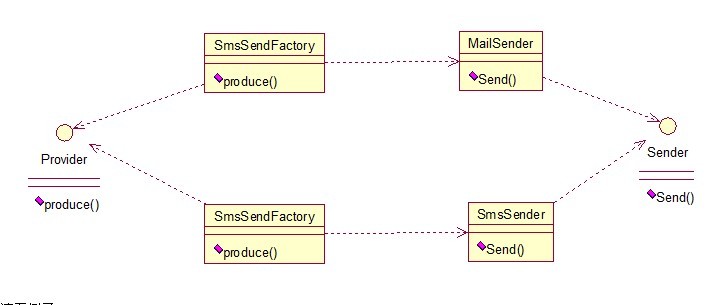
1. **public** **class** FactoryTest {
3. **public** **static** **void** main(String[] args) {
4. Sender sender = SendFactory.produceMail();
5. sender.Send();
6. }
7. }

输出：this is mailsender!

总体来说，工厂模式适合：凡是出现了大量的产品需要创建，并且具有共同的接口时，可以通过工厂方法模式进行创建。在以上的三种模式中，第一种如果传入的字 符串有误，不能正确创建对象，第三种相对于第二种，不需要实例化工厂类，所以，大多数情况下，我们会选用第三种——静态工厂方法模式。

### 1、工厂方法模式（Factory Method）

简单工厂模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定 的问题，如何解决？就用到工厂方法模式，创建一个工厂接口和创建多个工厂实现类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前 的代码。



请看例子：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **interface** Sender {
2. **public** **void** Send();
3. }

两个实现类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **class** MailSender **implements** Sender {
2. @Override
3. **public** **void** Send() {
4. System.out.println("this is mailsender!");
5. }
6. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **class** SmsSender **implements** Sender {
3. @Override
4. **public** **void** Send() {
5. System.out.println("this is sms sender!");
6. }
7. }

两个工厂类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **class** SendMailFactory **implements** Provider {
3. @Override
4. **public** Sender produce(){
5. **return** **new** MailSender();
6. }
7. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **class** SendSmsFactory **implements** Provider{
3. @Override
4. **public** Sender produce() {
5. **return** **new** SmsSender();
6. }
7. }

在提供一个接口：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **interface** Provider {
2. **public** Sender produce();
3. }

测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
4. Provider provider = **new** SendMailFactory();
5. Sender sender = provider.produce();
6. sender.Send();
7. }
8. }

其实这个模式的好处就是，如果你现在想增加一个功能：发及时信息，则只需做一个实现类，实现Sender接口，同时做一个工厂类，实现Provider接口，就OK了，无需去改动现成的代码。这样做，拓展性较好！

### 2、抽象工厂模式

工厂方法模式和抽象工厂模式不好分清楚，他们的区别如下：

工厂方法模式：

一个抽象产品类，可以派生出多个具体产品类。

一个抽象工厂类，可以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。

抽象工厂模式：

多个抽象产品类，每个抽象产品类可以派生出多个具体产品类。

一个抽象工厂类，可以派生出多个具体工厂类。

每个具体工厂类可以创建多个具体产品类的实例，也就是创建的是一个产品线下的多个产品。

区别：

工厂方法模式只有一个抽象产品类，而抽象工厂模式有多个。

工厂方法模式的具体工厂类只能创建一个具体产品类的实例，而抽象工厂模式可以创建多个。

工厂方法创建 "一种" 产品，他的着重点在于"怎么创建"，也就是说如果你开发，你的大量代码很可能围绕着这种产品的构造，初始化这些细节上面。也因为如此，类似的产品之间有很多可以复用的特征，所以会和模版方法相随。   
  
抽象工厂需要创建一些列产品，着重点在于"创建哪些"产品上，也就是说，如果你开发，你的主要任务是划分不同差异的产品线，并且尽量保持每条产品线接口一致，从而可以从同一个抽象工厂继承。

对于java来说，你能见到的大部分抽象工厂模式都是这样的：

---它的里面是一堆工厂方法，每个工厂方法返回某种类型的对象。

比如说工厂可以生产鼠标和键盘。那么抽象工厂的实现类（它的某个具体子类）的对象都可以生产鼠标和键盘，但可能工厂A生产的是罗技的键盘和鼠标，工厂B是微软的。

这样A和B就是工厂，对应于抽象工厂；

每个工厂生产的鼠标和键盘就是产品，对应于工厂方法；

用了工厂方法模式，你替换生成键盘的工厂方法，就可以把键盘从罗技换到微软。但是用了抽象工厂模式，你只要换家工厂，就可以同时替换鼠标和键盘一套。如果你要的产品有几十个，当然用抽象工厂模式一次替换全部最方便（这个工厂会替你用相应的工厂方法）

所以说抽象工厂就像工厂，而工厂方法则像是工厂的一种产品生产线

### ****3、单例模式（Singleton）****

单例对象（Singleton）是一种常用的设计模式。在Java应用中，单例对象能保证在一个JVM中，该对象只有一个实例存在。这样的模式有几个好处：

1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。

2、省去了new操作符，降低了系统内存的使用频率，减轻GC压力。

3、有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全乱了。（比如一个军队出现了多个司令员同时指挥，肯定会乱成一团），所以只有使用单例模式，才能保证核心交易服务器独立控制整个流程。

首先我们写一个简单的单例类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **class** Singleton {
3. /\* 持有私有静态实例，防止被引用，此处赋值为null，目的是实现延迟加载 \*/
4. **private** **static** Singleton instance = **null**;
6. /\* 私有构造方法，防止被实例化 \*/
7. **private** Singleton() {
8. }
10. /\* 静态工程方法，创建实例 \*/
11. **public** **static** Singleton getInstance() {
12. **if** (instance == **null**) {
13. instance = **new** Singleton();
14. }
15. **return** instance;
16. }
18. /\* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 \*/
19. **public** Object readResolve() {
20. **return** instance;
21. }
22. }

这个类可以满足基本要求，但是，像这样毫无线程安全保护的类，如果我们把它放入多线程的环境下，肯定就会出现问题了，如何解决？我们首先会想到对getInstance方法加synchronized关键字，如下：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **static** **synchronized** Singleton getInstance() {
2. **if** (instance == **null**) {
3. instance = **new** Singleton();
4. }
5. **return** instance;
6. }

但是，synchronized关键字锁住的是这个对象，这样的用法，在性能上会有所下降，因为每次调用getInstance()，都要对对象上锁，事实上，只有在第一次创建对象的时候需要加锁，之后就不需要了，所以，这个地方需要改进。我们改成下面这个：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **static** Singleton getInstance() {
2. **if** (instance == **null**) {
3. **synchronized** (instance) {
4. **if** (instance == **null**) {
5. instance = **new** Singleton();
6. }
7. }
8. }
9. **return** instance;
10. }

似乎解决了之前提到的问题，将synchronized关键字加在了内部，也就是说当调用的时候是不需要加锁的，只有在instance为null，并创建对象的时候才需要加锁，性能有一定的提升。但是，这样的情况，还是有可能有问题的，看下面的情况：在Java指令中创建对象和赋值操作是分开进行的，也就是说instance = new Singleton();语句是分两步执行的。但是JVM并不保证这两个操作的先后顺序，也就是说有可能JVM会为新的Singleton实例分配空间，然后直接赋值给instance成员，然后再去初始化这个Singleton实例。这样就可能出错了，我们以A、B两个线程为例：

a>A、B线程同时进入了第一个if判断

b>A首先进入synchronized块，由于instance为null，所以它执行instance = new Singleton();

c>由于JVM内部的优化机制，JVM先画出了一些分配给Singleton实例的空白内存，并赋值给instance成员（注意此时JVM没有开始初始化这个实例），然后A离开了synchronized块。

d>B进入synchronized块，由于instance此时不是null，因此它马上离开了synchronized块并将结果返回给调用该方法的程序。

e>此时B线程打算使用Singleton实例，却发现它没有被初始化，于是错误发生了。

所以程序还是有可能发生错误，其实程序在运行过程是很复杂的，从这点我们就可以看出，尤其是在写多线程环境下的程序更有难度，有挑战性。我们对该程序做进一步优化：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **private** **static** **class** SingletonFactory{
2. **private** **static** Singleton instance = **new** Singleton();
3. }
4. **public** **static** Singleton getInstance(){
5. **return** SingletonFactory.instance;
6. }

实际情况是，单例模式使用内部类来维护单例的实现，JVM内部的机制能够保证当一个类被加载的时候，这个类的加载过程是线程互斥的。这样当我们第一次调用getInstance的时候，JVM能够帮我们保证instance只被创建一次，并且会保证把赋值给instance的内存初始化完毕，这样我们就不用担心上面的问题。同时该方法也只会在第一次调用的时候使用互斥机制，这样就解决了低性能问题。这样我们暂时总结一个完美的单例模式：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **class** Singleton {
3. /\* 私有构造方法，防止被实例化 \*/
4. **private** Singleton() {
5. }
7. /\* 此处使用一个内部类来维护单例 \*/
8. **private** **static** **class** SingletonFactory {
9. **private** **static** Singleton instance = **new** Singleton();
10. }
12. /\* 获取实例 \*/
13. **public** **static** Singleton getInstance() {
14. **return** SingletonFactory.instance;
15. }
17. /\* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 \*/
18. **public** Object readResolve() {
19. **return** getInstance();
20. }
21. }

其实说它完美，也不一定，如果在构造函数中抛出异常，实例将永远得不到创建，也会出错。所以说，十分完美的东西是没有的，我们只能根据实际情况，选择最适合自己应用场景的实现方法。也有人这样实现：因为我们只需要在创建类的时候进行同步，所以只要将创建和getInstance()分开，单独为创建加synchronized关键字，也是可以的：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **class** SingletonTest {
3. **private** **static** SingletonTest instance = **null**;
5. **private** SingletonTest() {
6. }
8. **private** **static** **synchronized** **void** syncInit() {
9. **if** (instance == **null**) {
10. instance = **new** SingletonTest();
11. }
12. }
14. **public** **static** SingletonTest getInstance() {
15. **if** (instance == **null**) {
16. syncInit();
17. }
18. **return** instance;
19. }
20. }

考虑性能的话，整个程序只需创建一次实例，所以性能也不会有什么影响。

**补充：采用"影子实例"的办法为单例对象的属性同步更新**

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **class** SingletonTest {
3. **private** **static** SingletonTest instance = **null**;
4. **private** Vector properties = **null**;
6. **public** Vector getProperties() {
7. **return** properties;
8. }
10. **private** SingletonTest() {
11. }
13. **private** **static** **synchronized** **void** syncInit() {
14. **if** (instance == **null**) {
15. instance = **new** SingletonTest();
16. }
17. }
19. **public** **static** SingletonTest getInstance() {
20. **if** (instance == **null**) {
21. syncInit();
22. }
23. **return** instance;
24. }
26. **public** **void** updateProperties() {
27. SingletonTest shadow = **new** SingletonTest();
28. properties = shadow.getProperties();
29. }
30. }

通过单例模式的学习告诉我们：

1、单例模式理解起来简单，但是具体实现起来还是有一定的难度。

2、synchronized关键字锁定的是对象，在用的时候，一定要在恰当的地方使用（注意需要使用锁的对象和过程，可能有的时候并不是整个对象及整个过程都需要锁）。

到这儿，单例模式基本已经讲完了，结尾处，笔者突然想到另一个问题，就是采用类的静态方法，实现单例模式的效果，也是可行的，此处二者有什么不同？

首先，静态类不能实现接口。（从类的角度说是可以的，但是那样就破坏了静态了。因为接口中不允许有static修饰的方法，所以即使实现了也是非静态的）

其次，单例可以被延迟初始化，静态类一般在第一次加载是初始化。之所以延迟加载，是因为有些类比较庞大，所以延迟加载有助于提升性能。

再次，单例类可以被继承，他的方法可以被覆写。但是静态类内部方法都是static，无法被覆写。

最后一点，单例类比较灵活，毕竟从实现上只是一个普通的Java类，只要满足单例的基本需求，你可以在里面随心所欲的实现一些其它功能，但是静态类不行。从上面这些概括中，基本可以看出二者的区别，但是，从另一方面讲，我们上面最后实现的那个单例模式，内部就是用一个静态类来实现的，所以，二者有很大的关联，只是我们考虑问题的层面不同罢了。两种思想的结合，才能造就出完美的解决方案，就像HashMap采用数组+链表来实现一样，其实生活中很多事情都是这样，单用不同的方法来处理问题，总是有优点也有缺点，最完美的方法是，结合各个方法的优点，才能最好的解决问题！

### ****4、建造者模式（Builder）****

### **5、原**型模式（Prototype）****

原型模式虽然是创建型的模式，但是与工程模式没有关系，从名字即可看出，该模式的思想就是将一个对象作为原型，对其进行复制、克隆，产生一个和原对象类似的新对象。本小结会通过对象的复制，进行讲解。在Java中，复制对象是通过clone()实现的，先创建一个原型类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **class** Prototype **implements** Cloneable {
3. **public** Object clone() **throws** CloneNotSupportedException {
4. Prototype proto = (Prototype) **super**.clone();
5. **return** proto;
6. }
7. }

很简单，一个原型类，只需要实现Cloneable接口，覆写clone方法，此处clone方法可以改成任意的名称，因为Cloneable接口是个空接口，你可以任意定义实现类的方法名，如cloneA或者cloneB，因为此处的重点是super.clone()这句话，super.clone()调用的是Object的clone()方法，而在Object类中，clone()是native的，具体怎么实现，我会在另一篇文章中，关于解读Java中本地方法的调用，此处不再深究。在这儿，我将结合对象的浅复制和深复制来说一下，首先需要了解对象深、浅复制的概念：

浅复制：将一个对象复制后，基本数据类型的变量都会重新创建，而引用类型，指向的还是原对象所指向的。

深复制：将一个对象复制后，不论是基本数据类型还有引用类型，都是重新创建的。简单来说，就是深复制进行了完全彻底的复制，而浅复制不彻底。

此处，写一个深浅复制的例子：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. **public** **class** Prototype **implements** Cloneable, Serializable {
3. **private** **static** **final** **long** serialVersionUID = 1L;
4. **private** String string;
6. **private** SerializableObject obj;
8. /\* 浅复制 \*/
9. **public** Object clone() **throws** CloneNotSupportedException {
10. Prototype proto = (Prototype) **super**.clone();
11. **return** proto;
12. }
14. /\* 深复制 \*/
15. **public** Object deepClone() **throws** IOException, ClassNotFoundException {
17. /\* 写入当前对象的二进制流 \*/
18. ByteArrayOutputStream bos = **new** ByteArrayOutputStream();
19. ObjectOutputStream oos = **new** ObjectOutputStream(bos);
20. oos.writeObject(**this**);
22. /\* 读出二进制流产生的新对象 \*/
23. ByteArrayInputStream bis = **new** ByteArrayInputStream(bos.toByteArray());
24. ObjectInputStream ois = **new** ObjectInputStream(bis);
25. **return** ois.readObject();
26. }
28. **public** String getString() {
29. **return** string;
30. }
32. **public** **void** setString(String string) {
33. **this**.string = string;
34. }
36. **public** SerializableObject getObj() {
37. **return** obj;
38. }
40. **public** **void** setObj(SerializableObject obj) {
41. **this**.obj = obj;
42. }
44. }
46. **class** SerializableObject **implements** Serializable {
47. **private** **static** **final** **long** serialVersionUID = 1L;
48. }

要实现深复制，需要采用流的形式读入当前对象的二进制输入，再写出二进制数据对应的对象。

## B、结构模式（7种）

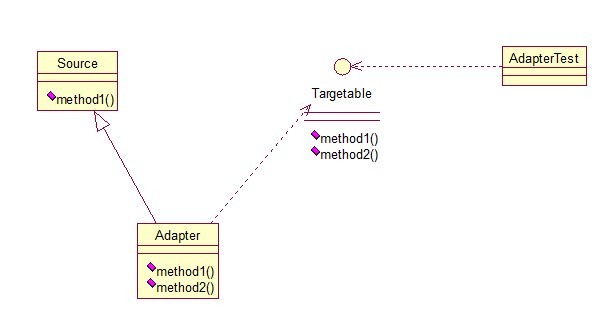
我们接着讨论设计模式，上篇文章我讲完了5种创建型模式，这章开始，我将讲下7种结构型模式：适配器模式、装饰模式、代理模式、外观模式、桥接模式、组合模式、享元模式。其中对象的适配器模式是各种模式的起源，我们看下面的图：



### 6、适配器模式

 适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。

#### 01、类的适配器模式



核心思想就是：有一个Source类，拥有一个方法，待适配，目标接口是Targetable，通过Adapter类，将Source的功能扩展到Targetable里，看代码：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** Source {
3. **public** **void** method1() {
4. System.out.println("this is original method!");
5. }
6. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **interface** Targetable {
3. /\* 与原类中的方法相同 \*/
4. **public** **void** method1();
6. /\* 新类的方法 \*/
7. **public** **void** method2();
8. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** Adapter **extends** Source **implements** Targetable {
3. @Override
4. **public** **void** method2() {
5. System.out.println("this is the targetable method!");
6. }
7. }

Adapter类继承Source类，实现Targetable接口，下面是测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** AdapterTest {
3. **public** **static** **void** main(String[] args) {
4. Targetable target = **new** Adapter();
5. target.method1();
6. target.method2();
7. }
8. }

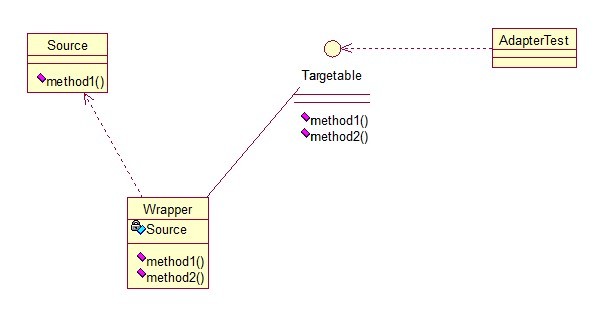
输出：

this is original method!  
this is the targetable method!

这样Targetable接口的实现类就具有了Source类的功能。

#### ****02、对象的适配器模式****

基本思路和类的适配器模式相同，只是将Adapter类作修改，这次不继承Source类，而是持有Source类的实例，以达到解决兼容性的问题。看图：



只需要修改Adapter类的源码即可：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** Wrapper **implements** Targetable {
3. **private** Source source;
5. **public** Wrapper(Source source){
6. **super**();
7. **this**.source = source;
8. }
9. @Override
10. **public** **void** method2() {
11. System.out.println("this is the targetable method!");
12. }
14. @Override
15. **public** **void** method1() {
16. source.method1();
17. }
18. }

测试类：

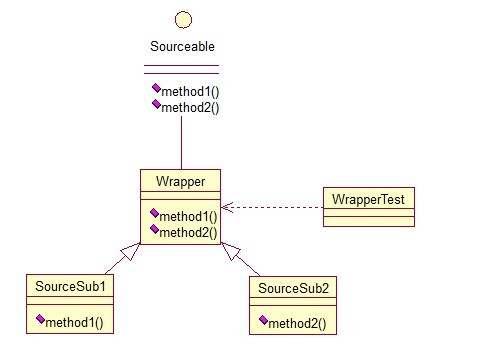
**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** AdapterTest {
3. **public** **static** **void** main(String[] args) {
4. Source source = **new** Source();
5. Targetable target = **new** Wrapper(source);
6. target.method1();
7. target.method2();
8. }
9. }

输出与第一种一样，只是适配的方法不同而已。

#### 03、接口的适配器模式

第三种适配器模式是**接口的适配器模式**，接口的适配器是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。看一下类图：



这个很好理解，在实际开发中，我们也常会遇到这种接口中定义了太多的方法，以致于有时我们在一些实现类中并不是都需要。看代码：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **interface** Sourceable {
3. **public** **void** method1();
4. **public** **void** method2();
5. }

抽象类Wrapper2：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **abstract** **class** Wrapper2 **implements** Sourceable{
3. **public** **void** method1(){}
4. **public** **void** method2(){}
5. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** SourceSub1 **extends** Wrapper2 {
2. **public** **void** method1(){
3. System.out.println("the sourceable interface's first Sub1!");
4. }
5. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** SourceSub2 **extends** Wrapper2 {
2. **public** **void** method2(){
3. System.out.println("the sourceable interface's second Sub2!");
4. }
5. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** WrapperTest {
3. **public** **static** **void** main(String[] args) {
4. Sourceable source1 = **new** SourceSub1();
5. Sourceable source2 = **new** SourceSub2();
7. source1.method1();
8. source1.method2();
9. source2.method1();
10. source2.method2();
11. }
12. }

测试输出：

the sourceable interface's first Sub1!  
the sourceable interface's second Sub2!

达到了我们的效果！

 讲了这么多，总结一下三种适配器模式的应用场景：

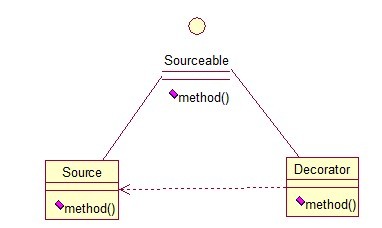
类的适配器模式：当希望将**一个类**转换成满足**另一个新接口**的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。

对象的适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个Wrapper类，持有原类的一个实例，在Wrapper类的方法中，调用实例的方法就行。

接口的适配器模式：当不希望实现一个接口中所有的方法时，可以创建一个抽象类Wrapper，实现所有方法，我们写别的类的时候，继承抽象类即可。

### ****7、装饰模式（Decorator）****

顾名思义，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，关系图如下：



Source类是被装饰类，Decorator类是一个装饰类，可以为Source类动态的添加一些功能，代码如下：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **interface** Sourceable {
2. **public** **void** method();
3. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** Source **implements** Sourceable {
3. @Override
4. **public** **void** method() {
5. System.out.println("the original method!");
6. }
7. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** Decorator **implements** Sourceable {
3. **private** Sourceable source;
5. **public** Decorator(Sourceable source){
6. **super**();
7. **this**.source = source;
8. }
9. @Override
10. **public** **void** method() {
11. System.out.println("before decorator!");
12. source.method();
13. System.out.println("after decorator!");
14. }
15. }

测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** DecoratorTest {
3. **public** **static** **void** main(String[] args) {
4. Sourceable source = **new** Source();
5. Sourceable obj = **new** Decorator(source);
6. obj.method();
7. }
8. }

输出：

before decorator!  
the original method!  
after decorator!

装饰器模式的应用场景：

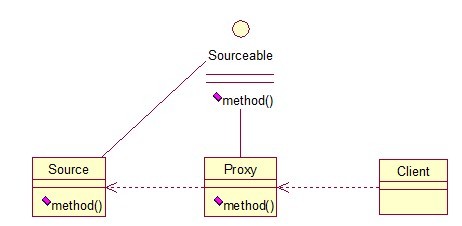
1、需要扩展一个类的功能。

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。）

缺点：产生过多相似的对象，不易排错！

### ****8、代理模式（Proxy）****

其实每个模式名称就表明了该模式的作用，代理模式就是多一个代理类出来，替原对象进行一些操作，比如我们在租房子的时候回去找中介，为什么呢？因为你对该地区房屋的信息掌握的不够全面，希望找一个更熟悉的人去帮你做，此处的代理就是这个意思。再如我们有的时候打官司，我们需要请律师，因为律师在法律方面有专长，可以替我们进行操作，表达我们的想法。先来看看关系图：



根据上文的阐述，代理模式就比较容易的理解了，我们看下代码：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **interface** Sourceable {
2. **public** **void** method();
3. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** Source **implements** Sourceable {
3. @Override
4. **public** **void** method() {
5. System.out.println("the original method!");
6. }
7. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** Proxy **implements** Sourceable {
3. **private** Source source;
4. **public** Proxy(){
5. **super**();
6. **this**.source = **new** Source();
7. }
8. @Override
9. **public** **void** method() {
10. before();
11. source.method();
12. atfer();
13. }
14. **private** **void** atfer() {
15. System.out.println("after proxy!");
16. }
17. **private** **void** before() {
18. System.out.println("before proxy!");
19. }
20. }

测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** ProxyTest {
3. **public** **static** **void** main(String[] args) {
4. Sourceable source = **new** Proxy();
5. source.method();
6. }
8. }

输出：

before proxy!  
the original method!  
after proxy!

代理模式的应用场景：

如果已有的方法在使用的时候需要对原有的方法进行改进，此时有两种办法：

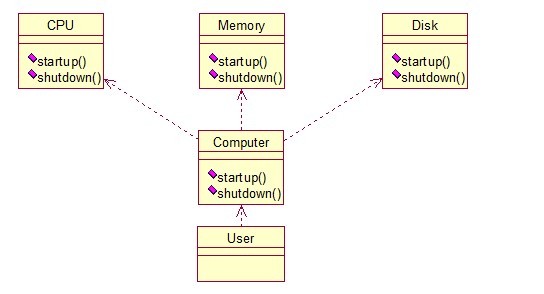
1、修改原有的方法来适应。这样违反了“对扩展开放，对修改关闭”的原则。

2、就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式。

使用代理模式，可以将功能划分的更加清晰，有助于后期维护！

### ****9、外观模式（Facade）****

外观模式是为了解决类与类之家的依赖关系的，像[**spring**](http://lib.csdn.net/base/17)一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个Facade类中，降低了类类之间的耦合度，该模式中没有涉及到接口，看下类图：（我们以一个计算机的启动过程为例）



我们先看下实现类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** CPU {
3. **public** **void** startup(){
4. System.out.println("cpu startup!");
5. }
7. **public** **void** shutdown(){
8. System.out.println("cpu shutdown!");
9. }
10. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** Memory {
3. **public** **void** startup(){
4. System.out.println("memory startup!");
5. }
7. **public** **void** shutdown(){
8. System.out.println("memory shutdown!");
9. }
10. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** Disk {
3. **public** **void** startup(){
4. System.out.println("disk startup!");
5. }
7. **public** **void** shutdown(){
8. System.out.println("disk shutdown!");
9. }
10. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** Computer {
2. **private** CPU cpu;
3. **private** Memory memory;
4. **private** Disk disk;
6. **public** Computer(){
7. cpu = **new** CPU();
8. memory = **new** Memory();
9. disk = **new** Disk();
10. }
12. **public** **void** startup(){
13. System.out.println("start the computer!");
14. cpu.startup();
15. memory.startup();
16. disk.startup();
17. System.out.println("start computer finished!");
18. }
20. **public** **void** shutdown(){
21. System.out.println("begin to close the computer!");
22. cpu.shutdown();
23. memory.shutdown();
24. disk.shutdown();
25. System.out.println("computer closed!");
26. }
27. }

User类如下：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** User {
3. **public** **static** **void** main(String[] args) {
4. Computer computer = **new** Computer();
5. computer.startup();
6. computer.shutdown();
7. }
8. }

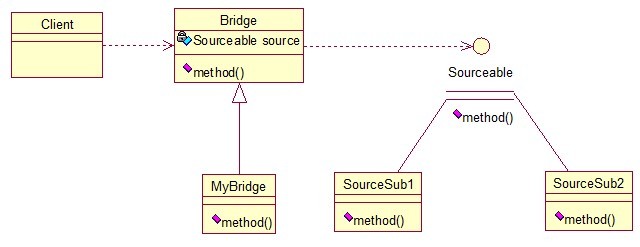
输出：

start the computer!  
cpu startup!  
memory startup!  
disk startup!  
start computer finished!  
begin to close the computer!  
cpu shutdown!  
memory shutdown!  
disk shutdown!  
computer closed!

如果我们没有Computer类，那么，CPU、Memory、Disk他们之间将会相互持有实例，产生关系，这样会造成严重的依赖，修改一个类，可能会带来其他类的修改，这不是我们想要看到的，有了Computer类，他们之间的关系被放在了Computer类里，这样就起到了解耦的作用，这，就是外观模式！

### ****10、桥接模式（Bridge）****

桥接模式就是把事物和其具体实现分开，使他们可以各自独立的变化。桥接的用意是：**将抽象化与实现化解耦，使得二者可以独立变化**，像我们常用的JDBC桥DriverManager一样，JDBC进行连接[**数据库**](http://lib.csdn.net/base/14)的时候，在各个数据库之间进行切换，基本不需要动太多的代码，甚至丝毫不用动，原因就是JDBC提供统一接口，每个数据库提供各自的实现，用一个叫做数据库驱动的程序来桥接就行了。我们来看看关系图：



实现代码：

先定义接口：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **interface** Sourceable {
2. **public** **void** method();
3. }

分别定义两个实现类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** SourceSub1 **implements** Sourceable {
3. @Override
4. **public** **void** method() {
5. System.out.println("this is the first sub!");
6. }
7. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** SourceSub2 **implements** Sourceable {
3. @Override
4. **public** **void** method() {
5. System.out.println("this is the second sub!");
6. }
7. }

定义一个桥，持有Sourceable的一个实例：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **abstract** **class** Bridge {
2. **private** Sourceable source;
4. **public** **void** method(){
5. source.method();
6. }
8. **public** Sourceable getSource() {
9. **return** source;
10. }
12. **public** **void** setSource(Sourceable source) {
13. **this**.source = source;
14. }
15. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** MyBridge **extends** Bridge {
2. **public** **void** method(){
3. getSource().method();
4. }
5. }

测试类：

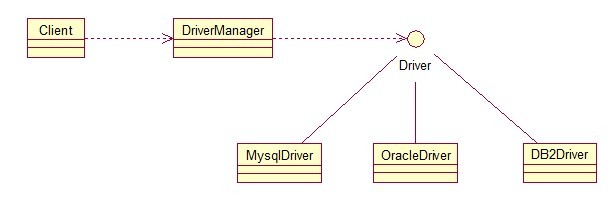
**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** BridgeTest {
3. **public** **static** **void** main(String[] args) {
5. Bridge bridge = **new** MyBridge();
7. /\*调用第一个对象\*/
8. Sourceable source1 = **new** SourceSub1();
9. bridge.setSource(source1);
10. bridge.method();
12. /\*调用第二个对象\*/
13. Sourceable source2 = **new** SourceSub2();
14. bridge.setSource(source2);
15. bridge.method();
16. }
17. }

output：

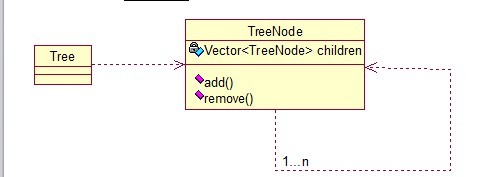
this is the first sub!  
this is the second sub!

这样，就通过对Bridge类的调用，实现了对接口Sourceable的实现类SourceSub1和SourceSub2的调用。接下来我再画个图，大家就应该明白了，因为这个图是我们JDBC连接的原理，有数据库学习基础的，一结合就都懂了。



### ****11、组合模式（Composite）****

组合模式有时又叫**部分-整体**模式在处理类似树形结构的问题时比较方便，看看关系图：



直接来看代码：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** TreeNode {
3. **private** String name;
4. **private** TreeNode parent;
5. **private** Vector<TreeNode> children = **new** Vector<TreeNode>();
7. **public** TreeNode(String name){
8. **this**.name = name;
9. }
11. **public** String getName() {
12. **return** name;
13. }
15. **public** **void** setName(String name) {
16. **this**.name = name;
17. }
19. **public** TreeNode getParent() {
20. **return** parent;
21. }
23. **public** **void** setParent(TreeNode parent) {
24. **this**.parent = parent;
25. }
27. //添加孩子节点
28. **public** **void** add(TreeNode node){
29. children.add(node);
30. }
32. //删除孩子节点
33. **public** **void** remove(TreeNode node){
34. children.remove(node);
35. }
37. //取得孩子节点
38. **public** Enumeration<TreeNode> getChildren(){
39. **return** children.elements();
40. }
41. }

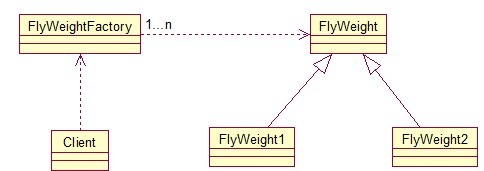
**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** Tree {
3. TreeNode root = **null**;
5. **public** Tree(String name) {
6. root = **new** TreeNode(name);
7. }
9. **public** **static** **void** main(String[] args) {
10. Tree tree = **new** Tree("A");
11. TreeNode nodeB = **new** TreeNode("B");
12. TreeNode nodeC = **new** TreeNode("C");
14. nodeB.add(nodeC);
15. tree.root.add(nodeB);
16. System.out.println("build the tree finished!");
17. }
18. }

使用场景：将多个对象组合在一起进行操作，常用于表示树形结构中，例如二叉树，数等。

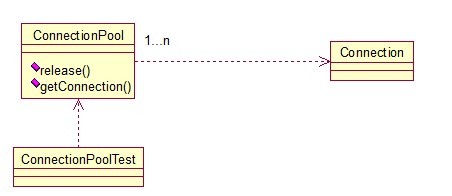
**12、享元模式（Flyweight）**

享元模式的主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。



FlyWeightFactory负责创建和管理享元单元，当一个客户端请求时，工厂需要检查当前对象池中是否有符合条件的对象，如果有，就返回已经存在的对象，如果没有，则创建一个新对象，FlyWeight是超类。一提到共享池，我们很容易联想到Java里面的JDBC连接池，想想每个连接的特点，我们不难总结出：适用于作共享的一些个对象，他们有一些共有的属性，就拿数据库连接池来说，url、driverClassName、username、password及dbname，这些属性对于每个连接来说都是一样的，所以就适合用享元模式来处理，建一个工厂类，将上述类似属性作为内部数据，其它的作为外部数据，在方法调用时，当做参数传进来，这样就节省了空间，减少了实例的数量。

看个例子：



看下数据库连接池的代码：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. **public** **class** ConnectionPool {
3. **private** Vector<Connection> pool;
5. /\*公有属性\*/
6. **private** String url = "jdbc:mysql://localhost:3306/test";
7. **private** String username = "root";
8. **private** String password = "root";
9. **private** String driverClassName = "com.mysql.jdbc.Driver";
11. **private** **int** poolSize = 100;
12. **private** **static** ConnectionPool instance = **null**;
13. Connection conn = **null**;
15. /\*构造方法，做一些初始化工作\*/
16. **private** ConnectionPool() {
17. pool = **new** Vector<Connection>(poolSize);
19. **for** (**int** i = 0; i < poolSize; i++) {
20. **try** {
21. Class.forName(driverClassName);
22. conn = DriverManager.getConnection(url, username, password);
23. pool.add(conn);
24. } **catch** (ClassNotFoundException e) {
25. e.printStackTrace();
26. } **catch** (SQLException e) {
27. e.printStackTrace();
28. }
29. }
30. }
32. /\* 返回连接到连接池 \*/
33. **public** **synchronized** **void** release() {
34. pool.add(conn);
35. }
37. /\* 返回连接池中的一个数据库连接 \*/
38. **public** **synchronized** Connection getConnection() {
39. **if** (pool.size() > 0) {
40. Connection conn = pool.get(0);
41. pool.remove(conn);
42. **return** conn;
43. } **else** {
44. **return** **null**;
45. }
46. }
47. }

通过连接池的管理，实现了数据库连接的共享，不需要每一次都重新创建连接，节省了数据库重新创建的开销，提升了系统的性能！

## ****C、关系模式（11种）****

先来张图，看看这11中模式的关系：

第一类：通过父类与子类的关系进行实现。

第二类：两个类之间。

第三类：类的状态。

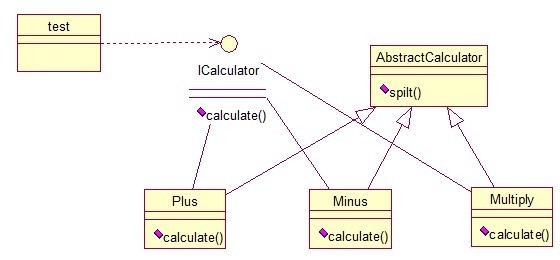
第四类：通过中间类



## ****父类与子类关系****

### ****13、策略模式（strategy）****

策略模式定义了一系列[**算法**](http://lib.csdn.net/base/31)，并将每个算法封装起来，使他们可以相互替换，且算法的变化不会影响到使用算法的客户。需要设计一个接口，为一系列实现类提供统一的方法，多个实现类实现该接口，设计一个抽象类（可有可无，属于辅助类），提供辅助函数，关系图如下：



图中ICalculator提供同意的方法，  
AbstractCalculator是辅助类，提供辅助方法，接下来，依次实现下每个类：

首先统一接口：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **interface** ICalculator {
2. **public** **int** calculate(String exp);
3. }

辅助类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **abstract** **class** AbstractCalculator {
3. **public** **int**[] split(String exp,String opt){
4. String array[] = exp.split(opt);
5. **int** arrayInt[] = **new** **int**[2];
6. arrayInt[0] = Integer.parseInt(array[0]);
7. arrayInt[1] = Integer.parseInt(array[1]);
8. **return** arrayInt;
9. }
10. }

三个实现类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** Plus **extends** AbstractCalculator **implements** ICalculator {
3. @Override
4. **public** **int** calculate(String exp) {
5. **int** arrayInt[] = split(exp,"\\+");
6. **return** arrayInt[0]+arrayInt[1];
7. }
8. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** Minus **extends** AbstractCalculator **implements** ICalculator {
3. @Override
4. **public** **int** calculate(String exp) {
5. **int** arrayInt[] = split(exp,"-");
6. **return** arrayInt[0]-arrayInt[1];
7. }
9. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** Multiply **extends** AbstractCalculator **implements** ICalculator {
3. @Override
4. **public** **int** calculate(String exp) {
5. **int** arrayInt[] = split(exp,"\\\*");
6. **return** arrayInt[0]\*arrayInt[1];
7. }
8. }

简单的测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

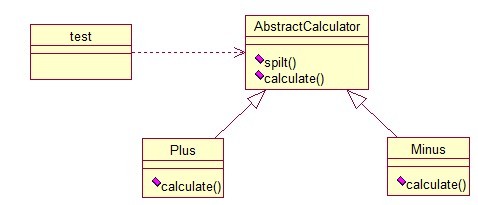
1. **public** **class** StrategyTest {
3. **public** **static** **void** main(String[] args) {
4. String exp = "2+8";
5. ICalculator cal = **new** Plus();
6. **int** result = cal.calculate(exp);
7. System.out.println(result);
8. }
9. }

输出：10

策略模式的决定权在用户，系统本身提供不同算法的实现，新增或者删除算法，对各种算法做封装。因此，策略模式多用在算法决策系统中，外部用户只需要决定用哪个算法即可。

### ****14、模板方法模式（Template Method）****

解释一下模板方法模式，就是指：一个抽象类中，有一个主方法，再定义1...n个方法，可以是抽象的，也可以是实际的方法，定义一个类，继承该抽象类，重写抽象方法，通过调用抽象类，实现对子类的调用，先看个关系图：



就是在AbstractCalculator类中定义一个主方法calculate，calculate()调用spilt()等，Plus和Minus分别继承AbstractCalculator类，通过对AbstractCalculator的调用实现对子类的调用，看下面的例子：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **abstract** **class** AbstractCalculator {
3. /\*主方法，实现对本类其它方法的调用\*/
4. **public** **final** **int** calculate(String exp,String opt){
5. **int** array[] = split(exp,opt);
6. **return** calculate(array[0],array[1]);
7. }
9. /\*被子类重写的方法\*/
10. **abstract** **public** **int** calculate(**int** num1,**int** num2);
12. **public** **int**[] split(String exp,String opt){
13. String array[] = exp.split(opt);
14. **int** arrayInt[] = **new** **int**[2];
15. arrayInt[0] = Integer.parseInt(array[0]);
16. arrayInt[1] = Integer.parseInt(array[1]);
17. **return** arrayInt;
18. }
19. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** Plus **extends** AbstractCalculator {
3. @Override
4. **public** **int** calculate(**int** num1,**int** num2) {
5. **return** num1 + num2;
6. }
7. }

测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

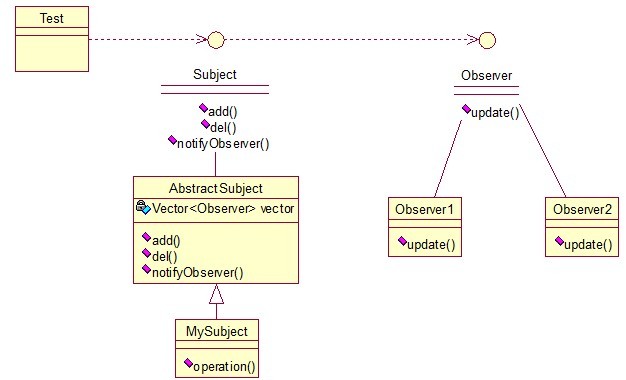
1. **public** **class** StrategyTest {
3. **public** **static** **void** main(String[] args) {
4. String exp = "8+8";
5. AbstractCalculator cal = **new** Plus();
6. **int** result = cal.calculate(exp, "\\+");
7. System.out.println(result);
8. }
9. }

我跟踪下这个小程序的执行过程：首先将exp和"\\+"做参数，调用AbstractCalculator类里的calculate(String,String)方法，在calculate(String,String)里调用同类的split()，之后再调用calculate(int ,int)方法，从这个方法进入到子类中，执行完return num1 + num2后，将值返回到AbstractCalculator类，赋给result，打印出来。正好验证了我们开头的思路。

## 类之间的关系

### ****15、观察者模式（Observer）****

包括这个模式在内的接下来的四个模式，都是类和类之间的关系，不涉及到继承，学的时候应该 记得归纳，记得本文最开始的那个图。观察者模式很好理解，类似于邮件订阅和RSS订阅，当我们浏览一些博客或wiki时，经常会看到RSS图标，就这的意思是，当你订阅了该文章，如果后续有更新，会及时通知你。其实，简单来讲就一句话：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。先来看看关系图：



我解释下这些类的作用：MySubject类就是我们的主对象，Observer1和Observer2是依赖于MySubject的对象，当MySubject变化时，Observer1和Observer2必然变化。AbstractSubject类中定义着需要监控的对象列表，可以对其进行修改：增加或删除被监控对象，且当MySubject变化时，负责通知在列表内存在的对象。我们看实现代码：

一个Observer接口：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **interface** Observer {
2. **public** **void** update();
3. }

两个实现类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** Observer1 **implements** Observer {
3. @Override
4. **public** **void** update() {
5. System.out.println("observer1 has received!");
6. }
7. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** Observer2 **implements** Observer {
3. @Override
4. **public** **void** update() {
5. System.out.println("observer2 has received!");
6. }
8. }

Subject接口及实现类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **interface** Subject {
3. /\*增加观察者\*/
4. **public** **void** add(Observer observer);
6. /\*删除观察者\*/
7. **public** **void** del(Observer observer);
9. /\*通知所有的观察者\*/
10. **public** **void** notifyObservers();
12. /\*自身的操作\*/
13. **public** **void** operation();
14. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **abstract** **class** AbstractSubject **implements** Subject {
3. **private** Vector<Observer> vector = **new** Vector<Observer>();
4. @Override
5. **public** **void** add(Observer observer) {
6. vector.add(observer);
7. }
9. @Override
10. **public** **void** del(Observer observer) {
11. vector.remove(observer);
12. }
14. @Override
15. **public** **void** notifyObservers() {
16. Enumeration<Observer> enumo = vector.elements();
17. **while**(enumo.hasMoreElements()){
18. enumo.nextElement().update();
19. }
20. }
21. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** MySubject **extends** AbstractSubject {
3. @Override
4. **public** **void** operation() {
5. System.out.println("update self!");
6. notifyObservers();
7. }
9. }

测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** ObserverTest {
3. **public** **static** **void** main(String[] args) {
4. Subject sub = **new** MySubject();
5. sub.add(**new** Observer1());
6. sub.add(**new** Observer2());
8. sub.operation();
9. }
11. }

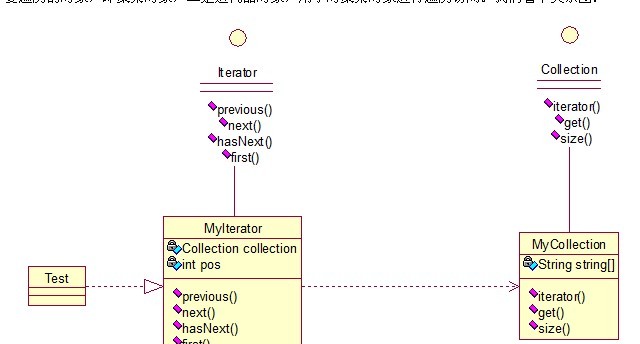
输出：

update self!  
observer1 has received!  
observer2 has received!

 这些东西，其实不难，只是有些抽象，不太容易整体理解，建议读者：**根据关系图，新建项目，自己写代码（或者参考我的代码）,按照总体思路走一遍，这样才能体会它的思想，理解起来容易！**

### ****16、迭代子模式（Iterator）****

顾名思义，迭代器模式就是顺序访问聚集中的对象，一般来说，集合中非常常见，如果对集合类比较熟悉的话，理解本模式会十分轻松。这句话包含两层意思：一是需要遍历的对象，即聚集对象，二是迭代器对象，用于对聚集对象进行遍历访问。我们看下关系图：



这个思路和我们常用的一模一样，MyCollection中定义了集合的一些操作，MyIterator中定义了一系列迭代操作，且持有Collection实例，我们来看看实现代码：

两个接口：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **interface** Collection {
3. **public** Iterator iterator();
5. /\*取得集合元素\*/
6. **public** Object get(**int** i);
8. /\*取得集合大小\*/
9. **public** **int** size();
10. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **interface** Iterator {
2. //前移
3. **public** Object previous();
5. //后移
6. **public** Object next();
7. **public** **boolean** hasNext();
9. //取得第一个元素
10. **public** Object first();
11. }

两个实现：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** MyCollection **implements** Collection {
3. **public** String string[] = {"A","B","C","D","E"};
4. @Override
5. **public** Iterator iterator() {
6. **return** **new** MyIterator(**this**);
7. }
9. @Override
10. **public** Object get(**int** i) {
11. **return** string[i];
12. }
14. @Override
15. **public** **int** size() {
16. **return** string.length;
17. }
18. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** MyIterator **implements** Iterator {
3. **private** Collection collection;
4. **private** **int** pos = -1;
6. **public** MyIterator(Collection collection){
7. **this**.collection = collection;
8. }
10. @Override
11. **public** Object previous() {
12. **if**(pos > 0){
13. pos--;
14. }
15. **return** collection.get(pos);
16. }
18. @Override
19. **public** Object next() {
20. **if**(pos<collection.size()-1){
21. pos++;
22. }
23. **return** collection.get(pos);
24. }
26. @Override
27. **public** **boolean** hasNext() {
28. **if**(pos<collection.size()-1){
29. **return** **true**;
30. }**else**{
31. **return** **false**;
32. }
33. }
35. @Override
36. **public** Object first() {
37. pos = 0;
38. **return** collection.get(pos);
39. }
41. }

测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

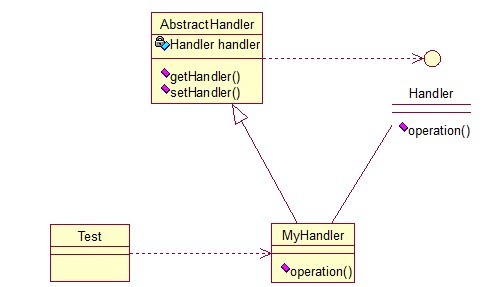
1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
4. Collection collection = **new** MyCollection();
5. Iterator it = collection.iterator();
7. **while**(it.hasNext()){
8. System.out.println(it.next());
9. }
10. }
11. }

输出：A B C D E

此处我们貌似模拟了一个集合类的过程，感觉是不是很爽？其实JDK中各个类也都是这些基本的东西，加一些设计模式，再加一些优化放到一起的，只要我们把这些东西学会了，掌握好了，我们也可以写出自己的集合类，甚至框架！

### ****17、责任链模式（Chain of Responsibility）****

接下来我们将要谈谈责任链模式，有多个对象，每个对象持有对下一个对象的引用，这样就会形成一条链，请求在这条链上传递，直到某一对象决定处理该请求。但是发出者并不清楚到底最终那个对象会处理该请求，所以，责任链模式可以实现，在隐瞒客户端的情况下，对系统进行动态的调整。先看看关系图：



Abstracthandler类提供了get和set方法，方便MyHandle类设置和修改引用对象，MyHandle类是核心，实例化后生成一系列相互持有的对象，构成一条链。

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **interface** Handler {
2. **public** **void** operator();
3. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **abstract** **class** AbstractHandler {
3. **private** Handler handler;
5. **public** Handler getHandler() {
6. **return** handler;
7. }
9. **public** **void** setHandler(Handler handler) {
10. **this**.handler = handler;
11. }
13. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** MyHandler **extends** AbstractHandler **implements** Handler {
3. **private** String name;
5. **public** MyHandler(String name) {
6. **this**.name = name;
7. }
9. @Override
10. **public** **void** operator() {
11. System.out.println(name+"deal!");
12. **if**(getHandler()!=**null**){
13. getHandler().operator();
14. }
15. }
16. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
4. MyHandler h1 = **new** MyHandler("h1");
5. MyHandler h2 = **new** MyHandler("h2");
6. MyHandler h3 = **new** MyHandler("h3");
8. h1.setHandler(h2);
9. h2.setHandler(h3);
11. h1.operator();
12. }
13. }

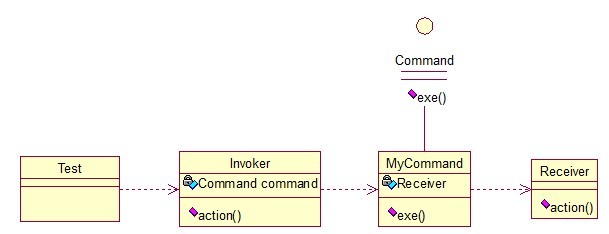
输出：

h1deal!  
h2deal!  
h3deal!

此处强调一点就是，链接上的请求可以是一条链，可以是一个树，还可以是一个环，模式本身不约束这个，需要我们自己去实现，同时，在一个时刻，命令只允许由一个对象传给另一个对象，而不允许传给多个对象。

### ****18、命令模式（Command）****

命令模式很好理解，举个例子，司令员下令让士兵去干件事情，从整个事情的角度来考虑，司令员的作用是，发出口令，口令经过传递，传到了士兵耳朵里，士兵去执行。这个过程好在，三者相互解耦，任何一方都不用去依赖其他人，只需要做好自己的事儿就行，司令员要的是结果，不会去关注到底士兵是怎么实现的。我们看看关系图：



Invoker是调用者（司令员），Receiver是被调用者（士兵），MyCommand是命令，实现了Command接口，持有接收对象，看实现代码：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **interface** Command {
2. **public** **void** exe();
3. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** MyCommand **implements** Command {
3. **private** Receiver receiver;
5. **public** MyCommand(Receiver receiver) {
6. **this**.receiver = receiver;
7. }
9. @Override
10. **public** **void** exe() {
11. receiver.action();
12. }
13. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** Receiver {
2. **public** **void** action(){
3. System.out.println("command received!");
4. }
5. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** Invoker {
3. **private** Command command;
5. **public** Invoker(Command command) {
6. **this**.command = command;
7. }
9. **public** **void** action(){
10. command.exe();
11. }
12. }

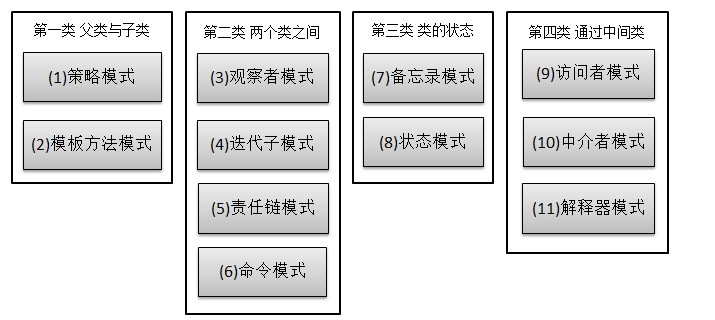
**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
4. Receiver receiver = **new** Receiver();
5. Command cmd = **new** MyCommand(receiver);
6. Invoker invoker = **new** Invoker(cmd);
7. invoker.action();
8. }
9. }

输出：command received!

这个很哈理解，命令模式的目的就是达到命令的发出者和执行者之间解耦，实现请求和执行分开，熟悉Struts的同学应该知道，Struts其实就是一种将请求和呈现分离的技术，其中必然涉及命令模式的思想！

其实每个设计模式都是很重要的一种思想，看上去很熟，其实是因为我们在学到的东西中都有涉及，尽管有时我们并不知道，其实在Java本身的设计之中处处都有体现，像AWT、JDBC、集合类、IO管道或者是Web框架，里面设计模式无处不在。因为我们篇幅有限，很难讲每一个设计模式都讲的很详细，不过我会尽我所能，尽量在有限的空间和篇幅内，把意思写清楚了，更好让大家明白。本章不出意外的话，应该是设计模式最后一讲了，首先还是上一下上篇开头的那个图：

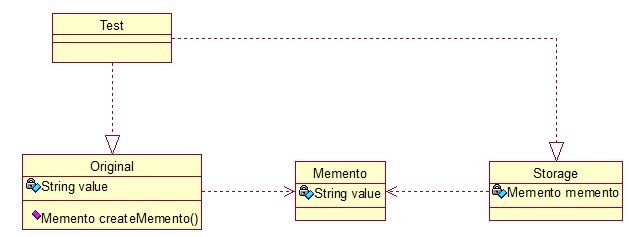


本章讲讲第三类和第四类。

## 类的状态

### ****19、备忘录模式（Memento）****

主要目的是保存一个对象的某个状态，以便在适当的时候恢复对象，个人觉得叫备份模式更形象些，通俗的讲下：假设有原始类A，A中有各种属性，A可以决定需要备份的属性，备忘录类B是用来存储A的一些内部状态，类C呢，就是一个用来存储备忘录的，且只能存储，不能修改等操作。做个图来分析一下：



Original类是原始类，里面有需要保存的属性value及创建一个备忘录类，用来保存value值。Memento类是备忘录类，Storage类是存储备忘录的类，持有Memento类的实例，该模式很好理解。直接看源码：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **class** Original {
3. **private** String value;
5. **public** String getValue() {
6. **return** value;
7. }
9. **public** **void** setValue(String value) {
10. **this**.value = value;
11. }
13. **public** Original(String value) {
14. **this**.value = value;
15. }
17. **public** Memento createMemento(){
18. **return** **new** Memento(value);
19. }
21. **public** **void** restoreMemento(Memento memento){
22. **this**.value = memento.getValue();
23. }
24. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **class** Memento {
3. **private** String value;
5. **public** Memento(String value) {
6. **this**.value = value;
7. }
9. **public** String getValue() {
10. **return** value;
11. }
13. **public** **void** setValue(String value) {
14. **this**.value = value;
15. }
16. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **class** Storage {
3. **private** Memento memento;
5. **public** Storage(Memento memento) {
6. **this**.memento = memento;
7. }
9. **public** Memento getMemento() {
10. **return** memento;
11. }
13. **public** **void** setMemento(Memento memento) {
14. **this**.memento = memento;
15. }
16. }

测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
5. // 创建原始类
6. Original origi = **new** Original("egg");
8. // 创建备忘录
9. Storage storage = **new** Storage(origi.createMemento());
11. // 修改原始类的状态
12. System.out.println("初始化状态为：" + origi.getValue());
13. origi.setValue("niu");
14. System.out.println("修改后的状态为：" + origi.getValue());
16. // 回复原始类的状态
17. origi.restoreMemento(storage.getMemento());
18. System.out.println("恢复后的状态为：" + origi.getValue());
19. }
20. }

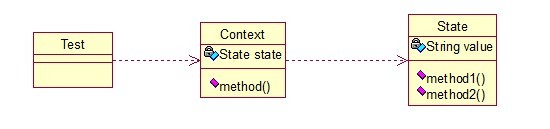
输出：

初始化状态为：egg  
修改后的状态为：niu  
恢复后的状态为：egg

简单描述下：新建原始类时，value被初始化为egg，后经过修改，将value的值置为niu，最后倒数第二行进行恢复状态，结果成功恢复了。其实我觉得这个模式叫“备份-恢复”模式最形象。

### ****20、状态模式（State）****

核心思想就是：当对象的状态改变时，同时改变其行为，很好理解！就拿QQ来说，有几种状态，在线、隐身、忙碌等，每个状态对应不同的操作，而且你的好友也能看到你的状态，所以，状态模式就两点：1、可以通过改变状态来获得不同的行为。2、你的好友能同时看到你的变化。看图：



State类是个状态类，Context类可以实现切换，我们来看看代码：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **package** com.xtfggef.dp.state;
3. /\*\*
4. \* 状态类的核心类
5. \* 2012-12-1
6. \* @author erqing
7. \*
8. \*/
9. **public** **class** State {
11. **private** String value;
13. **public** String getValue() {
14. **return** value;
15. }
17. **public** **void** setValue(String value) {
18. **this**.value = value;
19. }
21. **public** **void** method1(){
22. System.out.println("execute the first opt!");
23. }
25. **public** **void** method2(){
26. System.out.println("execute the second opt!");
27. }
28. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **package** com.xtfggef.dp.state;
3. /\*\*
4. \* 状态模式的切换类   2012-12-1
5. \* @author erqing
6. \*
7. \*/
8. **public** **class** Context {
10. **private** State state;
12. **public** Context(State state) {
13. **this**.state = state;
14. }
16. **public** State getState() {
17. **return** state;
18. }
20. **public** **void** setState(State state) {
21. **this**.state = state;
22. }
24. **public** **void** method() {
25. **if** (state.getValue().equals("state1")) {
26. state.method1();
27. } **else** **if** (state.getValue().equals("state2")) {
28. state.method2();
29. }
30. }
31. }

测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
5. State state = **new** State();
6. Context context = **new** Context(state);
8. //设置第一种状态
9. state.setValue("state1");
10. context.method();
12. //设置第二种状态
13. state.setValue("state2");
14. context.method();
15. }
16. }

输出：

execute the first opt!  
execute the second opt!

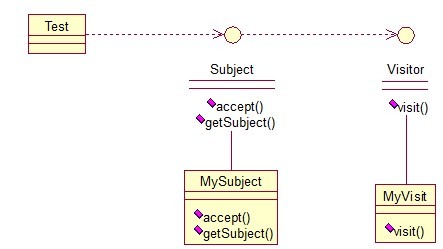
根据这个特性，状态模式在日常开发中用的挺多的，尤其是做网站的时候，我们有时希望根据对象的某一属性，区别开他们的一些功能，比如说简单的权限控制等。

## 通过中间类

### ****21、访问者模式（Visitor）****

访问者模式把[**数据结构**](http://lib.csdn.net/base/31)和作用于结构上的操作解耦合，使得操作集合可相对自由地演化。访问者模式适用于数据结构相对稳定算法又易变化的系统。因为访问者模式使得算法操作增加变得容易。若系统数据结构对象易于变化，经常有新的数据对象增加进来，则不适合使用访问者模式。访问者模式的优点是增加操作很容易，因为增加操作意味着增加新的访问者。访问者模式将有关行为集中到一个访问者对象中，其改变不影响系统数据结构。其缺点就是增加新的数据结构很困难。—— From 百科

简单来说，访问者模式就是一种分离对象数据结构与行为的方法，通过这种分离，可达到为一个被访问者动态添加新的操作而无需做其它的修改的效果。简单关系图：



来看看原码：一个Visitor类，存放要访问的对象，

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **interface** Visitor {
2. **public** **void** visit(Subject sub);
3. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **class** MyVisitor **implements** Visitor {
3. @Override
4. **public** **void** visit(Subject sub) {
5. System.out.println("visit the subject："+sub.getSubject());
6. }
7. }

Subject类，accept方法，接受将要访问它的对象，getSubject()获取将要被访问的属性，

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **interface** Subject {
2. **public** **void** accept(Visitor visitor);
3. **public** String getSubject();
4. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **class** MySubject **implements** Subject {
3. @Override
4. **public** **void** accept(Visitor visitor) {
5. visitor.visit(**this**);
6. }
8. @Override
9. **public** String getSubject() {
10. **return** "love";
11. }
12. }

测试：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

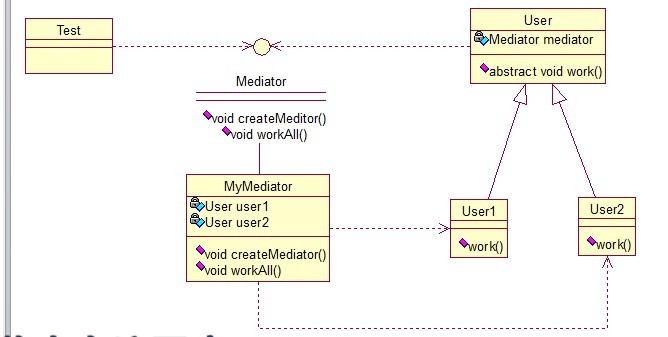
1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
5. Visitor visitor = **new** MyVisitor();
6. Subject sub = **new** MySubject();
7. sub.accept(visitor);
8. }
9. }

输出：visit the subject：love

该模式适用场景：如果我们想为一个现有的类增加新功能，不得不考虑几个事情：1、新功能会不会与现有功能出现兼容性问题？2、以后会不会再需要添加？3、如果类不允许修改代码怎么办？面对这些问题，最好的解决方法就是使用访问者模式，访问者模式适用于数据结构相对稳定的系统，把数据结构和算法解耦，

### ****22、中介者模式（Mediator）****

中介者模式也是用来降低类类之间的耦合的，因为如果类类之间有依赖关系的话，不利于功能的拓展和维护，因为只要修改一个对象，其它关联的对象都得进行修改。如果使用中介者模式，只需关心和Mediator类的关系，具体类类之间的关系及调度交给Mediator就行，这有点像spring容器的作用。先看看图：



User类统一接口，User1和User2分别是不同的对象，二者之间有关联，如果不采用中介者模式，则需要二者相互持有引用，这样二者的耦合度很高，为了解耦，引入了Mediator类，提供统一接口，MyMediator为其实现类，里面持有User1和User2的实例，用来实现对User1和User2的控制。这样User1和User2两个对象相互独立，他们只需要保持好和Mediator之间的关系就行，剩下的全由MyMediator类来维护！基本实现：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **interface** Mediator {
2. **public** **void** createMediator();
3. **public** **void** workAll();
4. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **class** MyMediator **implements** Mediator {
3. **private** User user1;
4. **private** User user2;
6. **public** User getUser1() {
7. **return** user1;
8. }
10. **public** User getUser2() {
11. **return** user2;
12. }
14. @Override
15. **public** **void** createMediator() {
16. user1 = **new** User1(**this**);
17. user2 = **new** User2(**this**);
18. }
20. @Override
21. **public** **void** workAll() {
22. user1.work();
23. user2.work();
24. }
25. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **abstract** **class** User {
3. **private** Mediator mediator;
5. **public** Mediator getMediator(){
6. **return** mediator;
7. }
9. **public** User(Mediator mediator) {
10. **this**.mediator = mediator;
11. }
13. **public** **abstract** **void** work();
14. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **class** User1 **extends** User {
3. **public** User1(Mediator mediator){
4. **super**(mediator);
5. }
7. @Override
8. **public** **void** work() {
9. System.out.println("user1 exe!");
10. }
11. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **class** User2 **extends** User {
3. **public** User2(Mediator mediator){
4. **super**(mediator);
5. }
7. @Override
8. **public** **void** work() {
9. System.out.println("user2 exe!");
10. }
11. }

测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

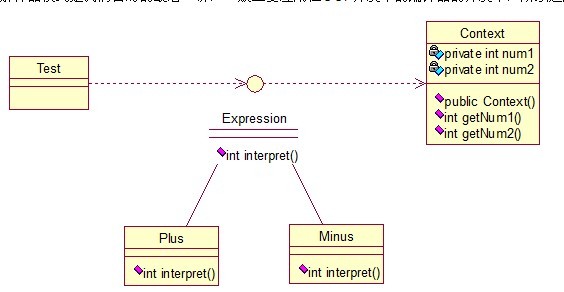
1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
4. Mediator mediator = **new** MyMediator();
5. mediator.createMediator();
6. mediator.workAll();
7. }
8. }

输出：

user1 exe!  
user2 exe!

### ****23、解释器模式（Interpreter）****

解释器模式是我们暂时的最后一讲，一般主要应用在OOP开发中的编译器的开发中，所以适用面比较窄。



Context类是一个上下文环境类，Plus和Minus分别是用来计算的实现，代码如下：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **interface** Expression {
2. **public** **int** interpret(Context context);
3. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **class** Plus **implements** Expression {
3. @Override
4. **public** **int** interpret(Context context) {
5. **return** context.getNum1()+context.getNum2();
6. }
7. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **class** Minus **implements** Expression {
3. @Override
4. **public** **int** interpret(Context context) {
5. **return** context.getNum1()-context.getNum2();
6. }
7. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **class** Context {
3. **private** **int** num1;
4. **private** **int** num2;
6. **public** Context(**int** num1, **int** num2) {
7. **this**.num1 = num1;
8. **this**.num2 = num2;
9. }
11. **public** **int** getNum1() {
12. **return** num1;
13. }
14. **public** **void** setNum1(**int** num1) {
15. **this**.num1 = num1;
16. }
17. **public** **int** getNum2() {
18. **return** num2;
19. }
20. **public** **void** setNum2(**int** num2) {
21. **this**.num2 = num2;
22. }

25. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
5. // 计算9+2-8的值
6. **int** result = **new** Minus().interpret((**new** Context(**new** Plus()
7. .interpret(**new** Context(9, 2)), 8)));
8. System.out.println(result);
9. }
10. }

最后输出正确的结果：3。

基本就这样，解释器模式用来做各种各样的解释器，如正则表达式等的解释器等等！

目录

[创建型](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750444)

[1. Factory Method（工厂方法）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750445)

[2. Abstract Factory（抽象工厂）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750446)

[3. Builder（建造者）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750447)

[4. Prototype（原型）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750448)

[5. Singleton（单例）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750449)

[结构型](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750450)

[6. Adapter Class/Object（适配器）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750451)

[7. Bridge（桥接）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750452)

[8. Composite（组合）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750453)

[9. Decorator（装饰）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750454)

[10. Facade（外观）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750455)

[11. Flyweight（享元）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750456)

[12. Proxy（代理）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750457)

[行为型](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750458)

[13. Interpreter（解释器）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750459)

[14. Template Method（模板方法）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750460)

[15. Chain of Responsibility（责任链）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750461)

[16. Command（命令）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750462)

[17. Iterator（迭代器）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750463)

[18. Mediator（中介者）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750464)

[19. Memento（备忘录）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750465)

[20. Observer（观察者）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750466)

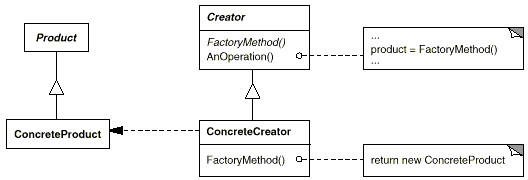
[21. State（状态）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750467)

[22. Strategy（策略）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750468)

[23. Visitor（访问者）](http://www.cnblogs.com/beijiguangyong/archive/2010/11/15/2302807.html#_Toc281750469)

# 创建型

## 1. Factory Method（工厂方法）



**意图：**

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。Factory Method 使**一**个类的实例化延迟到其子类。

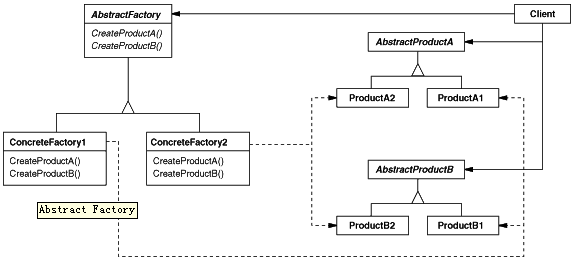
**适用性：**

当一个类不知道它所必须创建的对象的类的时候。

当一个类希望由它的子类来指定它所创建的对象的时候。

当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候。

## 2. Abstract Factory（抽象工厂）



**意图：**

提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。   
**适用性：**

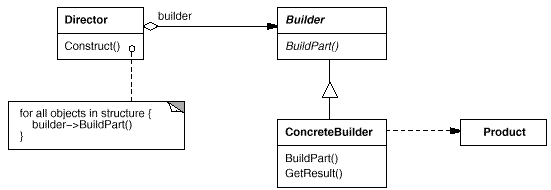
一个系统要独立于它的产品的创建、组合和表示时。

一个系统要由多个产品系列中的一个来配置时。

当你要强调一系列相关的产品对象的设计以便进行联合使用时。

当你提供一个产品类库，而只想显示它们的接口而不是实现时。

## 3. Builder（建造者）



**意图：**

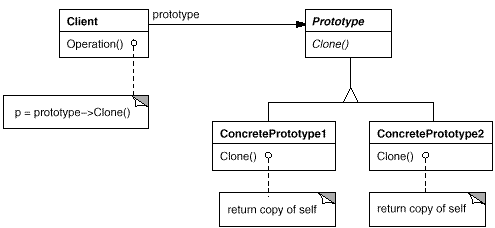
将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

**适用性：**

当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时。

当构造过程必须允许被构造的对象有不同的表示时。

## 4. Prototype（原型）



**意图：**

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

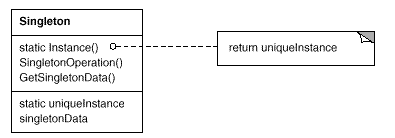
**适用性：**

当要实例化的类是在运行时刻指定时，例如，通过动态装载；或者

为了避免创建一个与产品类层次平行的工厂类层次时；或者

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。

## 5. Singleton（单例）



**意图：**

保证一个类仅有一个实例，并提供一个访问它的全局访问点。

**适用性：**

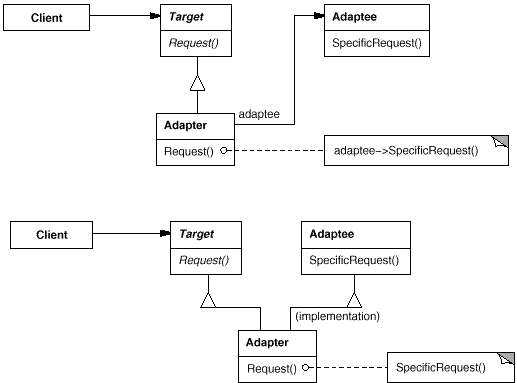
当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。

当这个唯一实例应该是通过子类化可扩展的，并且客户应该无需更改代码就能使用一个扩展的实例时。

# 结构型

## 6. Adapter Class/Object（适配器）

**意图：**



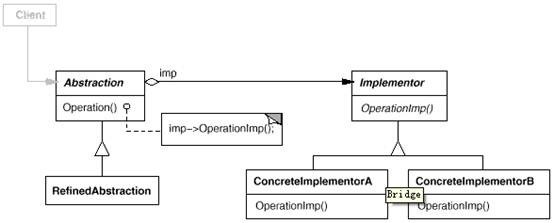
将一个类的接口转换成客户希望的另外一个接口。Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。   
**适用性：**

你想使用一个已经存在的类，而它的接口不符合你的需求。

你想创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作。

（仅适用于对象Adapter ）你想使用一些已经存在的子类，但是不可能对每一个都进行子类化以匹配它们的接口。对象适配器可以适配它的父类接口。

## 7. Bridge（桥接）



**意图：**

将抽象部分与它的实现部分分离，使它们都可以独立地变化。

**适用性：**

你不希望在抽象和它的实现部分之间有一个固定的绑定关系。例如这种情况可能是因为，在程序运行时刻实现部分应可以被选择或者切换。

类的抽象以及它的实现都应该可以通过生成子类的方法加以扩充。这时Bridge 模式使你可以对不同的抽象接口和实现部分进行组合，并分别对它们进行扩充。

对一个抽象的实现部分的修改应对客户不产生影响，即客户的代码不必重新编译。

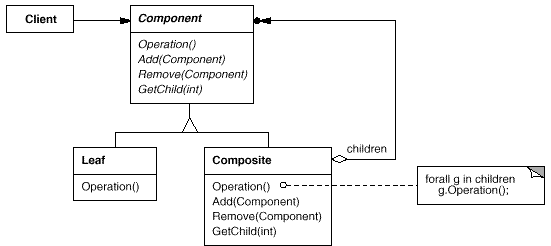
（C++）你想对客户完全隐藏抽象的实现部分。在C++中，类的表示在类接口中是可见的。

有许多类要生成。这样一种类层次结构说明你必须将一个对象分解成两个部分。Rumbaugh 称这种类层次结构为“嵌套的普化”（nested generalizations ）。

你想在多个对象间共享实现（可能使用引用计数），但同时要求客户并不知道这一点。一个简单的例子便是Coplien 的String 类[ Cop92 ]，在这个类中多个对象可以共享同一个字符串表示（StringRep ）。

## 8. Composite（组合）

**意图：**



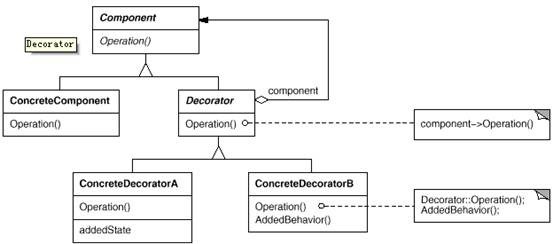
将对象组合成树形结构以表示“部分-整体”的层次结构。C o m p o s i t e 使得用户对单个对象和组合对象的使用具有一致性。   
**适用性：**

你想表示对象的部分-整体层次结构。

你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。

## 9. Decorator（装饰）

**意图：**动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator 模式相比生成子类更为灵活。   
**适用性：**

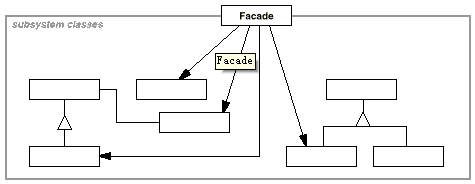


在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。

处理那些可以撤消的职责。

当不能采用生成子类的方法进行扩充时。一种情况是，可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。另一种情况可能是因为类定义被隐藏，或类定义不能用于生成子类。

## 10. Facade（外观）



**意图：**

为子系统中的一组接口提供一个一致的界面，Facade模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

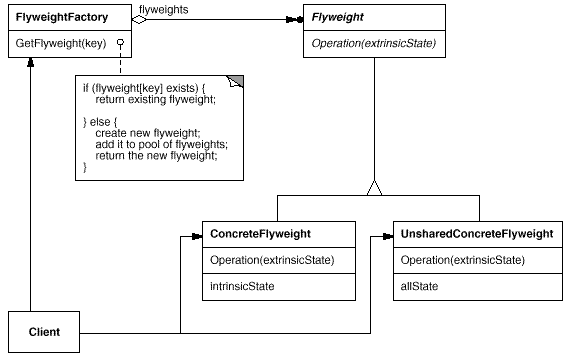
**适用性：**

当你要为一个复杂子系统提供一个简单接口时。子系统往往因为不断演化而变得越来越复杂。大多数模式使用时都会产生更多更小的类。这使得子系统更具可重用性，也更容易对子系统进行定制，但这也给那些不需要定制子系统的用户带来一些使用上的困难。Facade 可以提供一个简单的缺省视图，这一视图对大多数用户来说已经足够，而那些需要更多的可定制性的用户可以越过facade层。

客户程序与抽象类的实现部分之间存在着很大的依赖性。引入facade 将这个子系统与客户以及其他的子系统分离，可以提高子系统的独立性和可移植性。

当你需要构建一个层次结构的子系统时，使用facade模式定义子系统中每层的入口点。如果子系统之间是相互依赖的，你可以让它们仅通过facade进行通讯，从而简化了它们之间的依赖关系。

## 11. Flyweight（享元）



**意图：**

运用共享技术有效地支持大量细粒度的对象。

**适用性：**

一个应用程序使用了大量的对象。

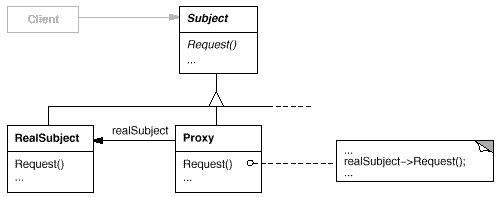
完全由于使用大量的对象，造成很大的存储开销。

对象的大多数状态都可变为外部状态。

如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象。

应用程序不依赖于对象标识。由于Flyweight 对象可以被共享，对于概念上明显有别的对象，标识测试将返回真值。

## 12. Proxy（代理）



**意图：**

为其他对象提供一种代理以控制对这个对象的访问。

**适用性：**

在需要用比较通用和复杂的对象指针代替简单的指针的时候，使用Proxy模式。下面是一 些可以使用Proxy 模式常见情况：   
1) 远程代理（Remote Proxy ）为一个对象在不同的地址空间提供局部代表。 NEXTSTEP[Add94] 使用NXProxy 类实现了这一目的。Coplien[Cop92] 称这种代理为“大使” （Ambassador ）。   
2 )虚代理（Virtual Proxy ）根据需要创建开销很大的对象。在动机一节描述的ImageProxy 就是这样一种代理的例子。   
3) 保护代理（Protection Proxy ）控制对原始对象的访问。保护代理用于对象应该有不同 的访问权限的时候。例如，在Choices 操作系统[ CIRM93]中KemelProxies为操作系统对象提供 了访问保护。   
4 )智能指引（Smart Reference ）取代了简单的指针，它在访问对象时执行一些附加操作。 它的典型用途包括：

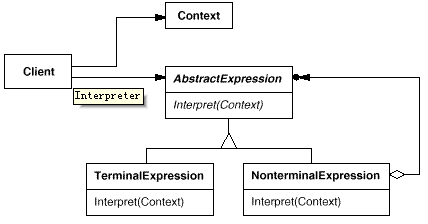
对指向实际对象的引用计数，这样当该对象没有引用时，可以自动释放它(也称为SmartPointers[Ede92 ] )。

当第一次引用一个持久对象时，将它装入内存。

在访问一个实际对象前，检查是否已经锁定了它，以确保其他对象不能改变它。

# 行为型

## 13. Interpreter（解释器）



**意图：**

给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

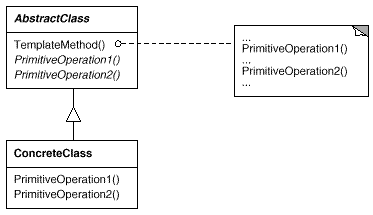
**适用性：**

当有一个语言需要解释执行, 并且你可将该语言中的句子表示为一个抽象语法树时，可使用解释器模式。而当存在以下情况时该模式效果最好：

该文法简单对于复杂的文法, 文法的类层次变得庞大而无法管理。此时语法分析程序生成器这样的工具是更好的选择。它们无需构建抽象语法树即可解释表达式, 这样可以节省空间而且还可能节省时间。

效率不是一个关键问题最高效的解释器通常不是通过直接解释语法分析树实现的, 而是首先将它们转换成另一种形式。例如，正则表达式通常被转换成状态机。但即使在这种情况下, 转换器仍可用解释器模式实现, 该模式仍是有用的。

## 14. Template Method（模板方法）



**意图：**

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。TemplateMethod 使得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。

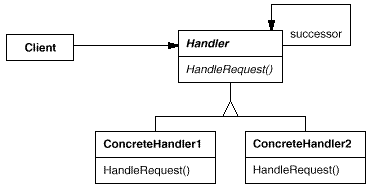
**适用性：**

一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。

各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。这是Opdyke 和Johnson 所描述过的“重分解以一般化”的一个很好的例子[ OJ93 ]。首先识别现有代码中的不同之处，并且将不同之处分离为新的操作。最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。

控制子类扩展。模板方法只在特定点调用“hook ”操作（参见效果一节），这样就只允许在这些点进行扩展。

## 15. Chain of Responsibility（责任链）



**意图：**

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

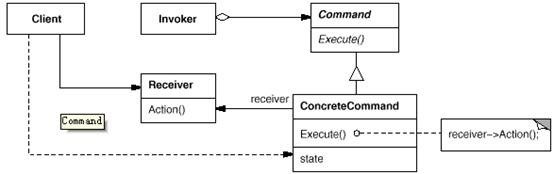
**适用性：**

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。

可处理一个请求的对象集合应被动态指定。

## 16. Command（命令）



**意图：**

将一个请求封装为一个对象，从而使你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，以及支持可撤消的操作。

**适用性：**

抽象出待执行的动作以参数化某对象，你可用过程语言中的回调（call back）函数表达这种参数化机制。所谓回调函数是指函数先在某处注册，而它将在稍后某个需要的时候被调用。Command 模式是回调机制的一个面向对象的替代品。

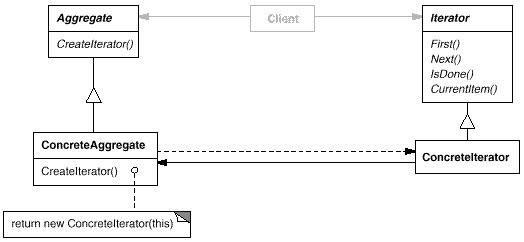
在不同的时刻指定、排列和执行请求。一个Command对象可以有一个与初始请求无关的生存期。如果一个请求的接收者可用一种与地址空间无关的方式表达，那么就可将负责该请求的命令对象传送给另一个不同的进程并在那儿实现该请求。

支持取消操作。Command的Excute 操作可在实施操作前将状态存储起来，在取消操作时这个状态用来消除该操作的影响。Command 接口必须添加一个Unexecute操作，该操作取消上一次Execute调用的效果。执行的命令被存储在一个历史列表中。可通过向后和向前遍历这一列表并分别调用Unexecute和Execute来实现重数不限的“取消”和“重做”。

支持修改日志，这样当系统崩溃时，这些修改可以被重做一遍。在Command接口中添加装载操作和存储操作，可以用来保持变动的一个一致的修改日志。从崩溃中恢复的过程包括从磁盘中重新读入记录下来的命令并用Execute操作重新执行它们。

用构建在原语操作上的高层操作构造一个系统。这样一种结构在支持事务( transaction)的信息系统中很常见。一个事务封装了对数据的一组变动。Command模式提供了对事务进行建模的方法。Command有一个公共的接口，使得你可以用同一种方式调用所有的事务。同时使用该模式也易于添加新事务以扩展系统。

## 17. Iterator（迭代器）



**意图：**

提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。

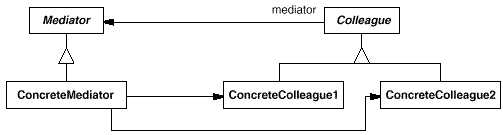
**适用性：**

访问一个聚合对象的内容而无需暴露它的内部表示。

支持对聚合对象的多种遍历。

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)。

## 18. Mediator（中介者）



**意图：**

用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

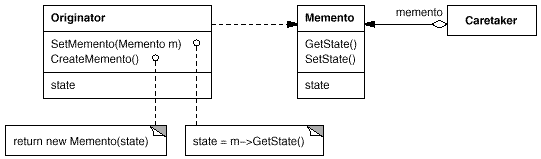
**适用性：**

一组对象以定义良好但是复杂的方式进行通信。产生的相互依赖关系结构混乱且难以理解。

一个对象引用其他很多对象并且直接与这些对象通信,导致难以复用该对象。

想定制一个分布在多个类中的行为，而又不想生成太多的子类。

## 19. Memento（备忘录）



**意图：**

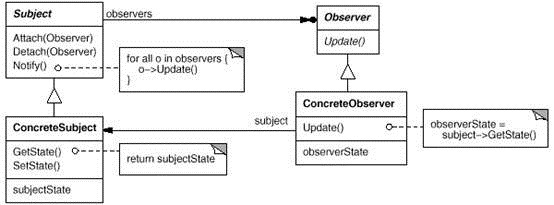
在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。

**适用性：**

必须保存一个对象在某一个时刻的(部分)状态, 这样以后需要时它才能恢复到先前的状态。

如果一个用接口来让其它对象直接得到这些状态，将会暴露对象的实现细节并破坏对象的封装性。

## 20. Observer（观察者）



**意图：**

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

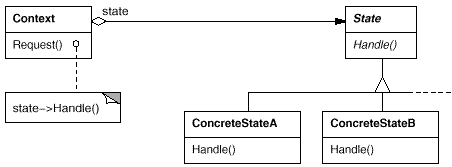
**适用性：**

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。

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。

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。换言之, 你不希望这些对象是紧密耦合的。

## 21. State（状态）



**意图：**

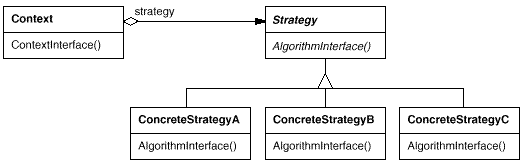
允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

**适用性：**

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。

一个操作中含有庞大的多分支的条件语句，且这些分支依赖于该对象的状态。这个状态通常用一个或多个枚举常量表示。通常, 有多个操作包含这一相同的条件结构。State模式将每一个条件分支放入一个独立的类中。这使得你可以根据对象自身的情况将对象的状态作为一个对象，这一对象可以不依赖于其他对象而独立变化。

## 22. Strategy（策略）



**意图：**

定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

**适用性：**

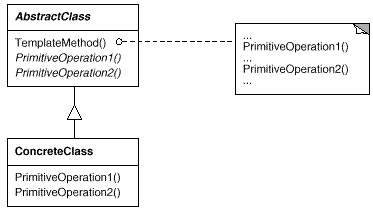
许多相关的类仅仅是行为有异。“策略”提供了一种用多个行为中的一个行为来配置一个类的方法。

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[H087] ,可以使用策略模式。

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。

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。

## 23. Visitor（访问者）



**意图：**

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。TemplateMethod 使得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。

**适用性：**

一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。

各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。这是Opdyke和Johnson所描述过的“重分解以一般化”的一个很好的例子[OJ93]。首先识别现有代码中的不同之处，并且将不同之处分离为新的操作。最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。

控制子类扩展。模板方法只在特定点调用“hook ”操作（参见效果一节），这样就只允许在这些点进行扩展