Java 23 种设计模式

设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因。本章系**Java之美[从菜鸟到高手演变]系列**之设计模式，我们会以理论与实践相结合的方式来进行本章的学习，希望广大程序爱好者，学好设计模式，做一个优秀的软件工程师！

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：

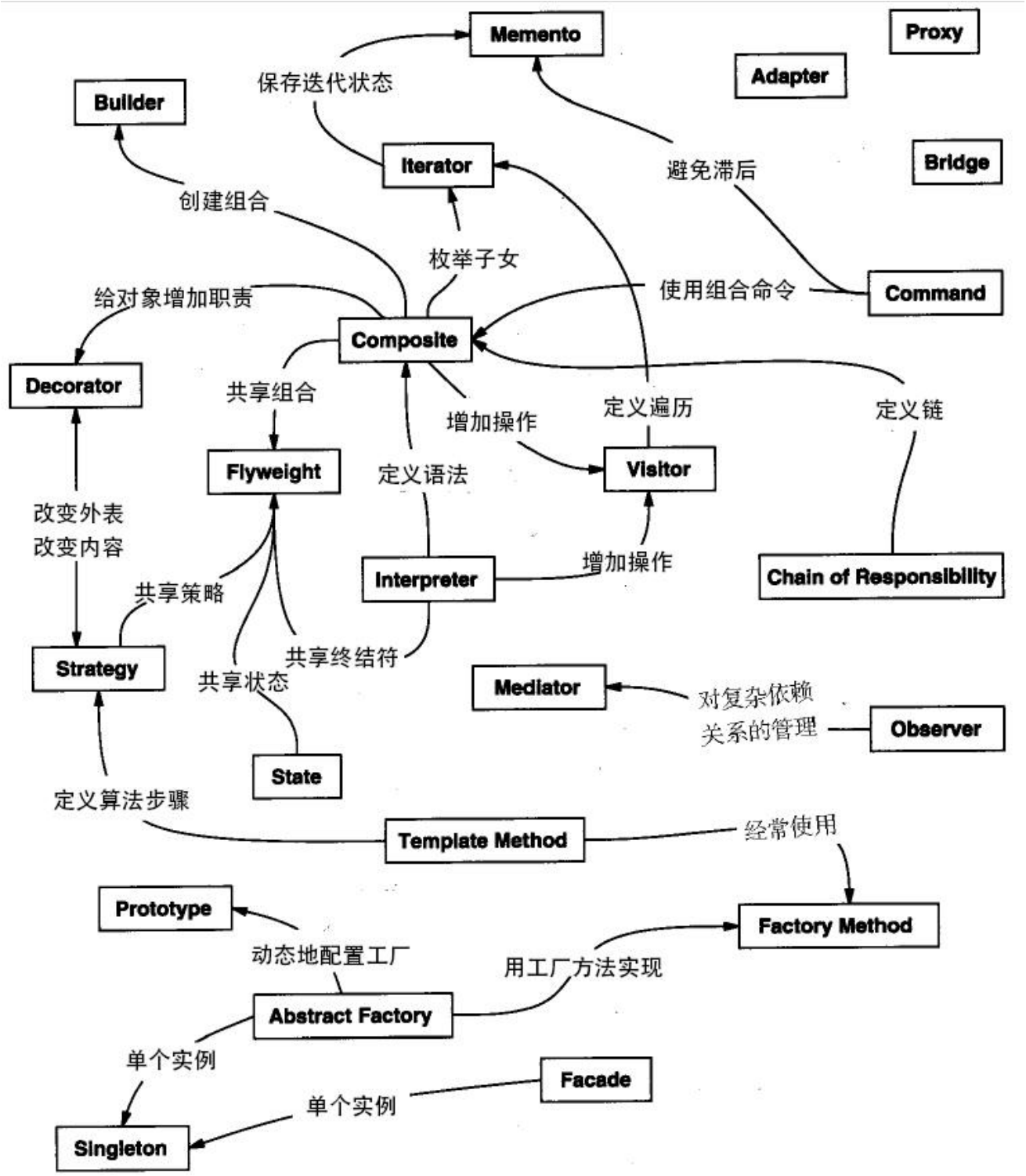


图　设计模式之间的关系

二、设计模式的六大原则

1、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，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用**接口和抽象**类，后面的具体设计中我们会提到这点。**【只能扩展，不能修改接口】**

2、里氏代换原则（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

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

这个是开闭原则的基础，具体内容：**真对接口编程，依赖于抽象**而不依赖于具体。

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

这个原则的意思是：**使用多个隔离的接口，比使用单个接口要好。**还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

5、迪米特法则（最少知道原则）（Demeter Principle）

为什么叫最少知道原则，就是说：一**个实体应当尽量少的与其他实体之间发生相互作用**，使得系统功能模块相对独立。

6、合成复用原则（Composite Reuse Principle）

原则是**尽量使用合成/聚合的方式，而不是使用继承。**

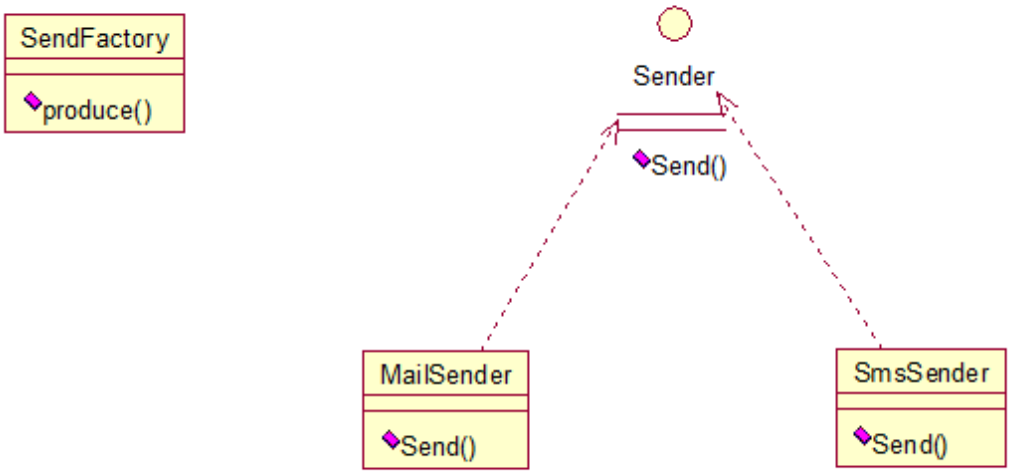
三、Java的23中设计模式

从这一块开始，我们详细介绍Java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。

1、工厂方法模式（Factory Method）

工厂方法模式分为三种：

11、普通工厂模式，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）

```
/// 首先创建二者共同的接口
public interface Sender {
    public void Send();
}

/// 然后创建两个实现类
public class MailSender implements Sender {
    @Override
    public void Send() {
        System.out.println("this is mailsender!");
    }
}

public class SmsSender implements Sender {

    @Override
    public void Send() {
        System.out.println("this is sms sender!");
    }
}
```

创建的工厂类如下：

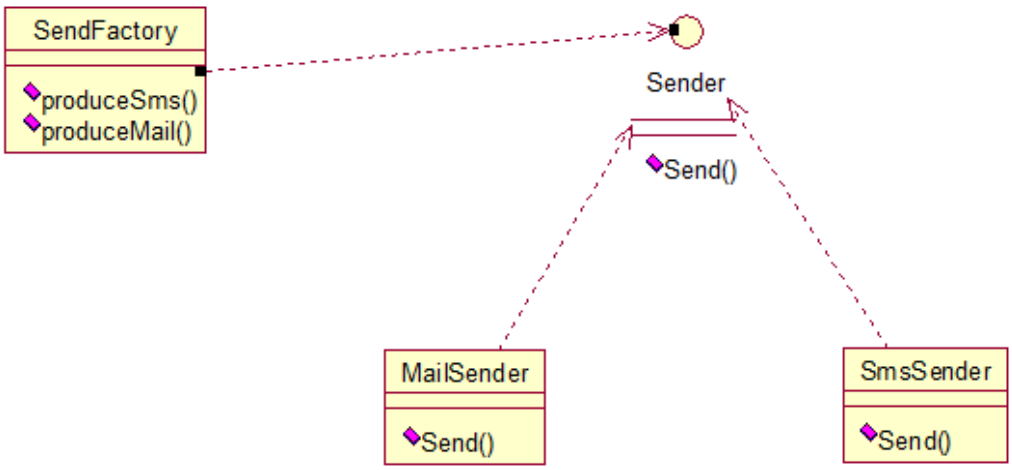
```
public class SendFactory {

    /// 方法名一般称为create 或者 produce
    public Sender produce(String type) {
        if ("mail".equals(type)) {
            return new MailSender();

        } else if ("sms".equals(type)) {
```

```
            return new SmsSender();
        } else {
            System.out.println("请输入正确的类型!");
            return null;
        }
    }
}
```

测试如下：

```
public class FactoryTest {

    public static void main(String[] args) {
        Sender sender = SendFactory.produceMail();
        sender.Send();
    }
}

>>>
输出：this is mailsender!
```

22、多个工厂方法模式，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：

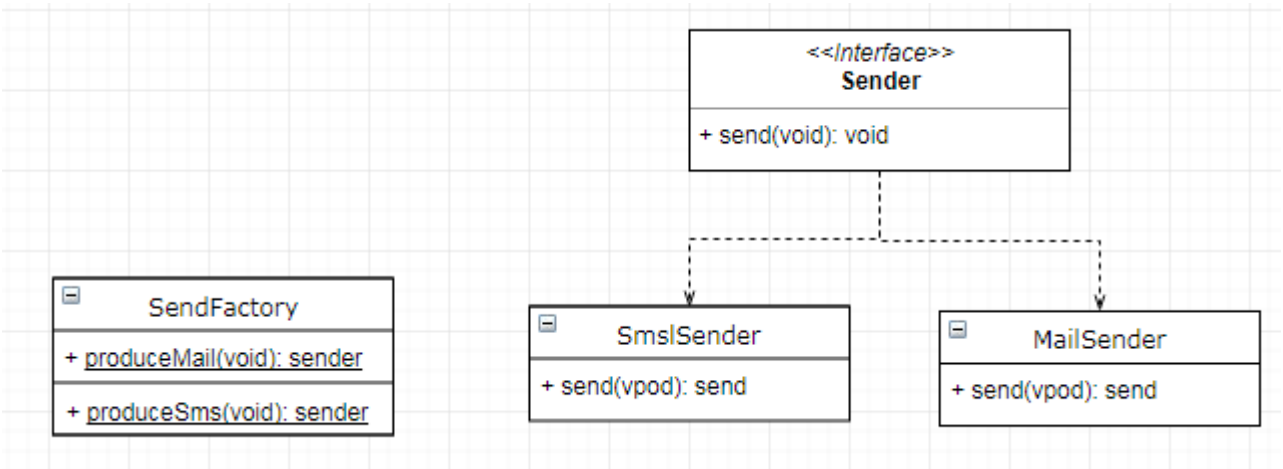
```
public class SendFactory {

    public Sender produceMail(){
        return new MailSender();
    }

    public Sender produceSms(){
        return new SmsSender();
    }
}
```

测试类如下：

```
public class FactoryTest {

    public static void main(String[] args) {
        SendFactory factory = new SendFactory();
        Sender sender = factory.produceMail();
        sender.Send();
    }
}
>>>
this is mailsender!
```

33、静态工厂方法模式，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。

类图如下：



```
public class SendFactory {

    public static Sender produceMail(){
        return new MailSender();
    }

    public static Sender produceSms(){
        return new SmsSender();
    }
}
```

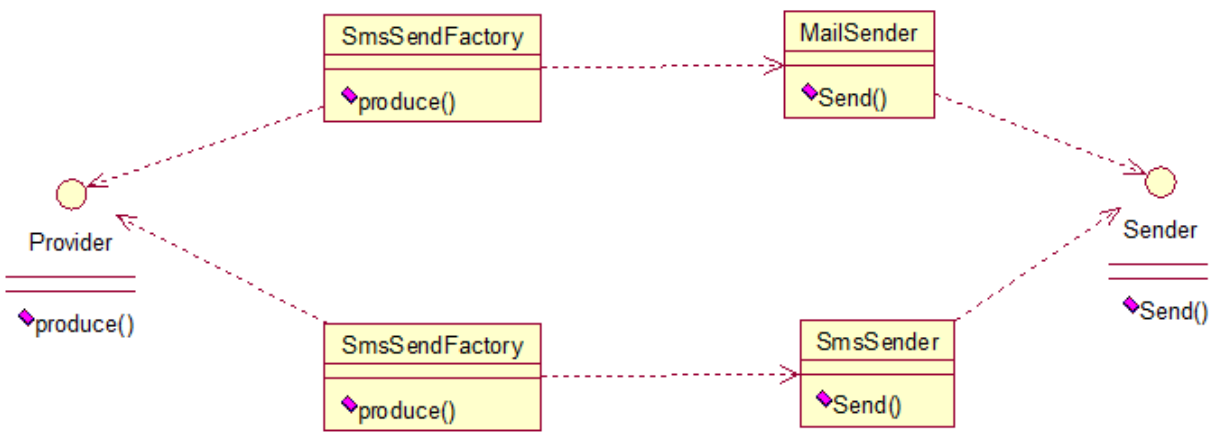
测试如下：

```
public class FactoryTest {

    public static void main(String[] args) {
        Sender sender = SendFactory.produceMail();
        sender.Send();
    }
}
>>>
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。**

2、抽象工厂模式（Abstract Factory）

工厂方法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？就用到抽象工厂模式，创建多个工厂类，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。因为抽象工厂不太好理解，我们先看看图，然后就和代码，就比较容易理解。



```
public interface Sender {
    public void Send();
}

/// 两个实际的实现类
public class MailSender implements Sender {
    @Override
    public void Send() {
        System.out.println("this is mailsender!");
    }
}

public class SmsSender implements Sender {

    @Override
    public void Send() {
        System.out.println("this is sms sender!");
    }
}
```

两个抽象工厂类

```
public interface Provider {
    public Sender produce();
}

/// 提供两个工厂类的实现
public class SendMailFactory implements Provider {

    @Override
    public Sender produce(){
        return new MailSender();
    }
}

public class SendSmsFactory implements Provider{

    @Override
    public Sender produce() {
        return new SmsSender();
    }
}
```

测试类

```
public class Test {

    public static void main(String[] args) {
        Provider provider = new SendMailFactory();  /// 用子类去初始化父类，被称为覆盖（多态？）
        Sender sender = provider.produce();
        sender.Send();
    }
}
```

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！

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：

```
public class Singleton {

    /* 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 */
    private static Singleton instance = null;

    /* 私有构造方法，防止被实例化 */
    private Singleton() {
    }

    /* 静态工程方法，创建实例 */
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    /* 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 */
    public Object readResolve() {
        return instance;
    }
}
```

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：

```
public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
```

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：

```
public static Singleton getInstance() {
        if (instance == null) {
            synchronized (instance) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
```

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：

Eager 模式

```
private static class SingletonFactory{
        private static Singleton instance = new Singleton();
    }
    public static Singleton getInstance(){
        return SingletonFactory.instance;
    }
```

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：

```
public class Singleton {

    /* 私有构造方法，防止被实例化 */
    private Singleton() {
    }

    /* 此处使用一个内部类来维护单例 */
    private static class SingletonFactory {
        private static Singleton instance = new Singleton();
    }

    /* 获取实例 */
    public static Singleton getInstance() {
        return SingletonFactory.instance;
    }

    /* 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 */
    public Object readResolve() {
        return getInstance();
    }
}
```

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：

```
public class SingletonTest {

    private static SingletonTest instance = null;

    private SingletonTest() {
    }

    private static synchronized void syncInit() {
        if (instance == null) {
            instance = new SingletonTest();
        }

    }
```

```
    public static SingletonTest getInstance() {
        if (instance == null) {
            syncInit();
        }
        return instance;
    }
}
```

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

```
public class SingletonTest {

    private static SingletonTest instance = null;
    private Vector properties = null;

    private SingletonTest() {
    }

    private static synchronized void syncInit() {
        if (instance == null) {
            instance = new SingletonTest();
        }
    }

    public static SingletonTest getInstance() {
        if (instance == null) {
            syncInit();
        }
        return instance;
    }

    /// 获取属性和更新属性的方法
    public Vector getProperties() {
        return properties;
    }
    public void updateProperties() {
        SingletonTest shadow = new SingletonTest();
        properties = shadow.getProperties();   /// 获得单个示例的属性，有可能会在其它地方被改变了。
    }
}
```

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）

工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理，用来创建复合对象，所谓复合对象就是指某个类具有不同的属性，其实建造者模式就是前面抽象工厂模式和最后的Test结合起来得到的。我们看一下代码：

还和前面一样，一个Sender接口，两个实现类MailSender和SmsSender。最后，建造者类如下：

```
public class Builder {
    private List<Sender> list = new ArrayList<>();

    public void produceMail(int count){
        for(int i=0; i<count; i++){
            list.add(new MailSender());
        }
        for(Sender s: list){
            s.Send();
        }
    }

    public void produceSms(int count){
        for(int i=0; i<count; i++){
           list.add(new SmsSender());
        }
        for(Sender s: list){
            s.Send();
        }
    }
}
```

测试类：

```
public class Test {

    public static void main(String[] args) {
        Builder builder = new Builder();
        builder.produceMailSender(10);
    }
}
```

从这点看出，建造者模式将很多功能集成到一个类里，这个类可以创造出比较复杂的东西。所以与工程模式的区别就是：工厂模式**关注的是创建单个产品，而建造者模式则关注创建符合对象，多个部分。因此，是选择工厂模式还是建造者模式，依实际情况而定。**

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：

```
public class Prototype implements Cloneable {

    public Object clone() throws CloneNotSupportedException {
        Prototype proto = (Prototype) super.clone();
        return proto;
    }
}
```

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。

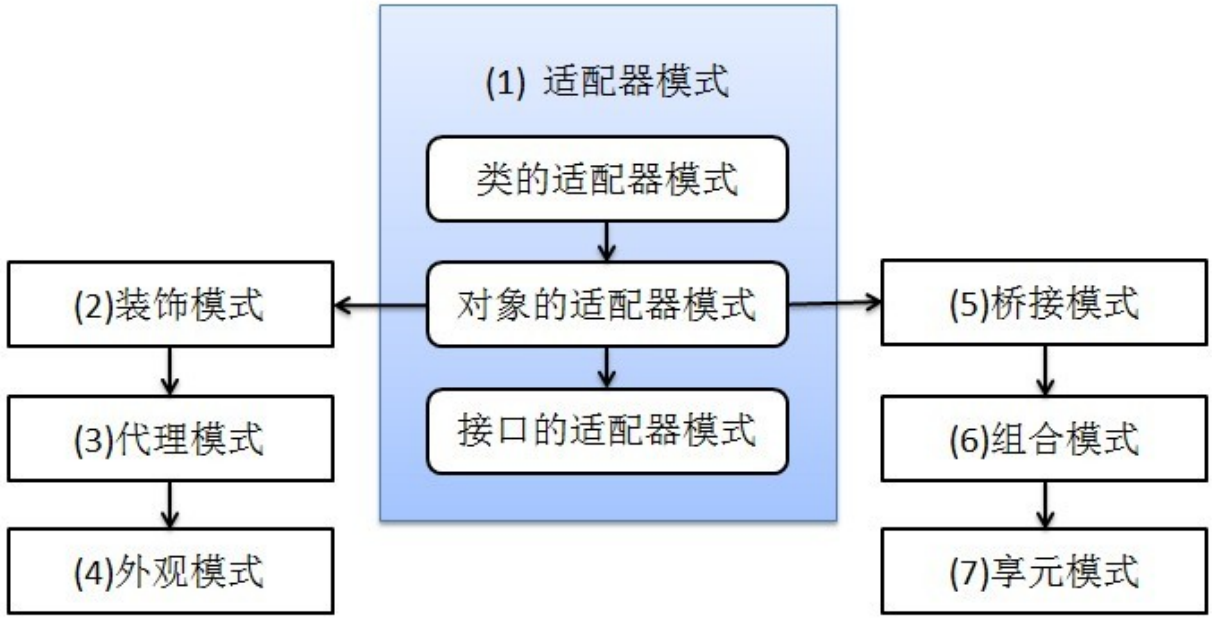
此处，写一个深浅复制的例子：

```
public class Prototype implements Cloneable, Serializable {

    private static final long serialVersionUID = 1L;
    private String string;

    private SerializableObject obj;

    /* 浅复制 */
    public Object clone() throws CloneNotSupportedException {
        Prototype proto = (Prototype) super.clone();
        return proto;
    }
```

```
    /* 深复制 */
    public Object deepClone() throws IOException, ClassNotFoundException {

        /* 写入当前对象的二进制流 */
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(bos);
        oos.writeObject(this);

        /* 读出二进制流产生的新对象 */
        ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());
        ObjectInputStream ois = new ObjectInputStream(bis);
        return ois.readObject();
    }

    public String getString() {
        return string;
    }

    public void setString(String string) {
        this.string = string;
    }

    public SerializableObject getObj() {
        return obj;
    }

    public void setObj(SerializableObject obj) {
        this.obj = obj;
    }

}

class SerializableObject implements Serializable {
    private static final long serialVersionUID = 1L;
}
```

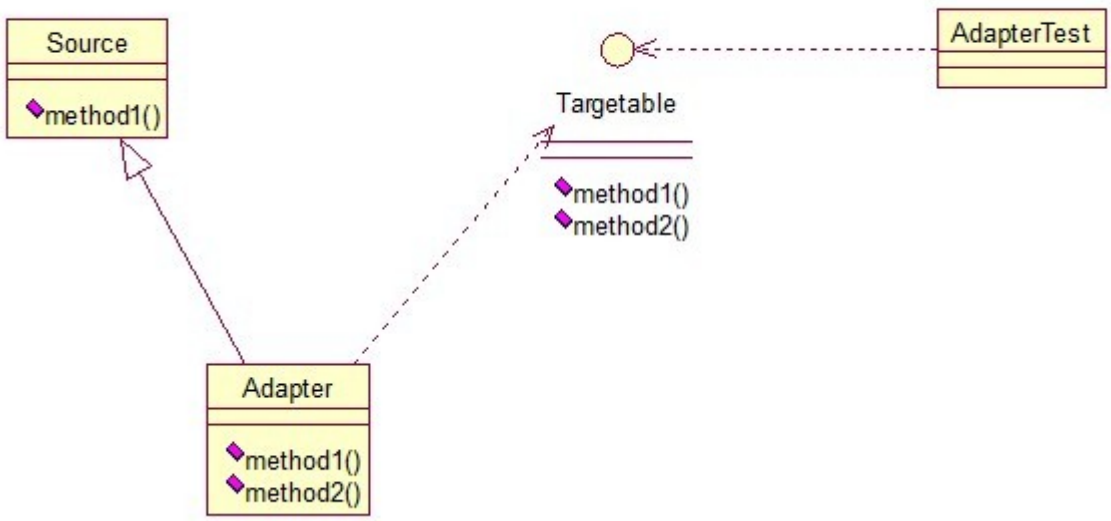
要实现深复制，需要采用流的形式读入当前对象的二进制输入，再写出二进制数据对应的对象。

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（Adapter）

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。首先，我们来看看**类的适配器模式**，先看类图：



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

```
public class Source {

    public void method1() {
        System.out.println("this is original method!");
    }
}


public interface Targetable {
```

```
    /* 与原类中的方法相同 */
    public void method1();

    /* 新类的方法 */
    public void method2();
}

public class Adapter extends Source implements Targetable {

    @Override
    public void method2() {
        System.out.println("this is the targetable method!");
    }
}
```

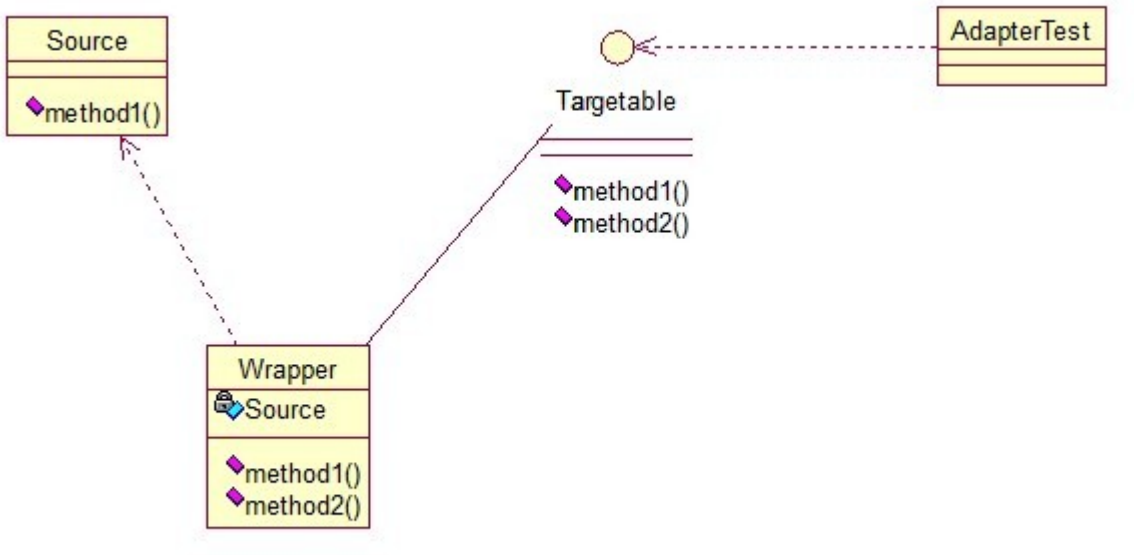
Adapter类继承Source类，实现Targetable接口，下面是测试类：

```
public class AdapterTest {

    public static void main(String[] args) {
        Targetable target = new Adapter();
        target.method1();
        target.method2();
    }
}
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。

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：

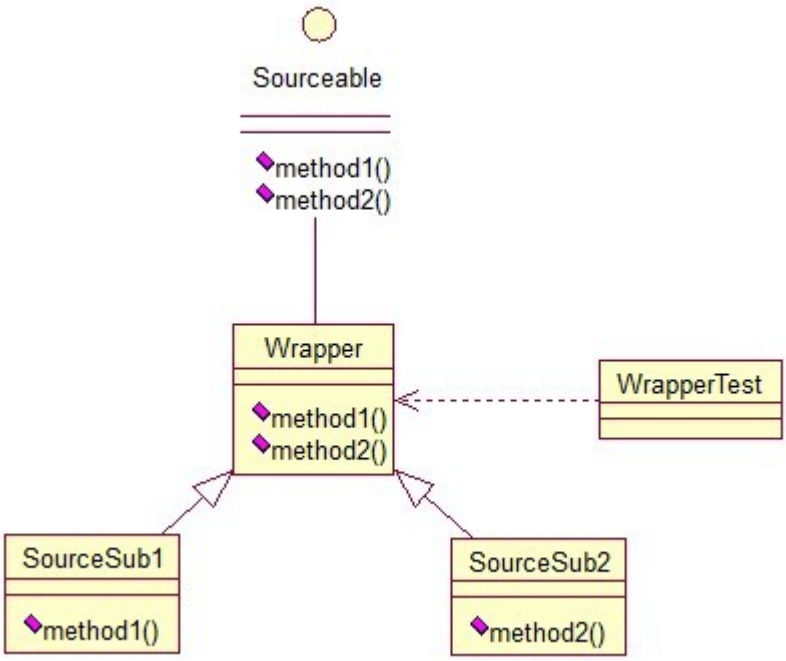
```
public class Wrapper implements Targetable {

    private Source source;

    public Wrapper(Source source){
        super();
        this.source = source;
    }
    @Override
    public void method2() {
        System.out.println("this is the targetable method!");
    }

    @Override
    public void method1() {
        source.method1();
    }
}
```

测试类：

```
public class AdapterTest {

    public static void main(String[] args) {
        Source source = new Source();
        Targetable target = new Wrapper(source);
        target.method1();
        target.method2();
    }
}
```

输出与第一种一样，只是适配的方法不同而已。

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：

```
public interface Sourceable {

    public void method1();
    public void method2();
}
```

抽象类Wrapper2：

```
public abstract class Wrapper2 implements Sourceable{

    public void method1(){}
    public void method2(){}
}

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

public class SourceSub2 extends Wrapper2 {
    public void method2(){
```

```
        System.out.println("the sourceable interface's second Sub2!");
    }
}

public class WrapperTest {

    public static void main(String[] args) {
        Sourceable source1 = new SourceSub1();
        Sourceable source2 = new SourceSub2();

        source1.method1();
        source1.method2();
        source2.method1();
        source2.method2();
    }
}
```

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：

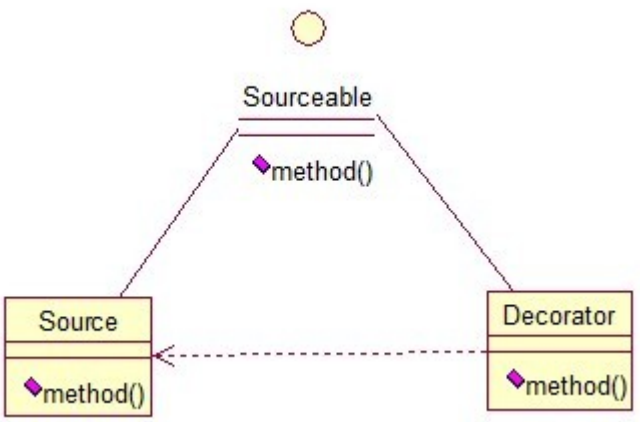
类的适配器模式：当希望将**一个类**转换成满足**另一个新接口**的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。

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：

```
public interface Sourceable {
```

```
    public void method();
}
public class Source implements Sourceable {

    @Override
    public void method() {
        System.out.println("the original method!");
    }
}

public class Decorator implements Sourceable {

    private Sourceable source;

    public Decorator(Sourceable source){
        super();
        this.source = source;
    }
    @Override
    public void method() {
        System.out.println("before decorator!");
        source.method();
        System.out.println("after decorator!");
    }
}
```

测试类：

```
public class DecoratorTest {

    public static void main(String[] args) {
        Sourceable source = new Source();
        Sourceable obj = new Decorator(source);
        obj.method();
    }
}
```

输出：

before decorator! the original method! after decorator!

装饰器模式的应用场景：

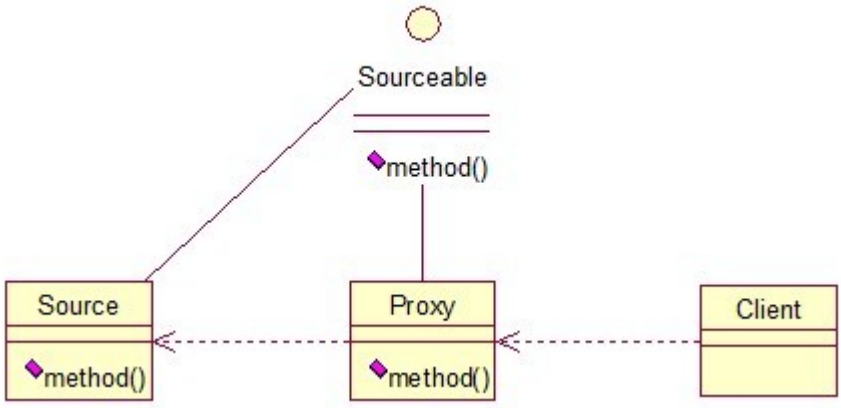
1、需要扩展一个类的功能。

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：

```
public interface Sourceable {
    public void method();
}

public class Source implements Sourceable {

    @Override
    public void method() {
        System.out.println("the original method!");
    }
}

public class Proxy implements Sourceable {

    private Source source;
    public Proxy(){
        super();
        this.source = new Source();
    }
    @Override
    public void method() {
        before();
        source.method();
        atfer();
    }
    private void atfer() {
        System.out.println("after proxy!");
    }
    private void before() {
        System.out.println("before proxy!");
    }
}
```

测试类；

```
public class ProxyTest {

    public static void main(String[] args) {
        Sourceable source = new Proxy();
        source.method();
    }

}
```

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：

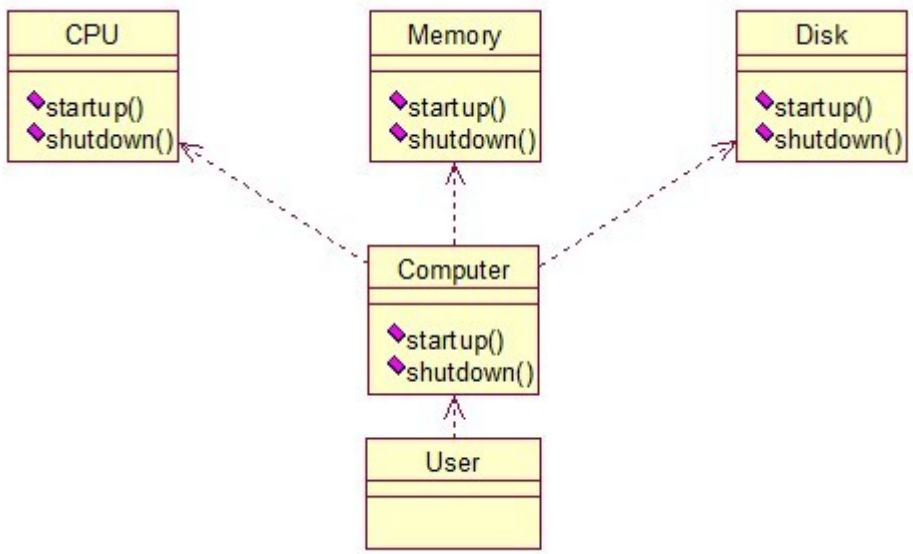
1、修改原有的方法来适应。这样违反了"对扩展开放，对修改关闭"的原则。

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：

```
public class CPU {

    public void startup(){
        System.out.println("cpu startup!");
    }
```

```
    public void shutdown(){
        System.out.println("cpu shutdown!");
    }
}

public class Memory {

    public void startup(){
        System.out.println("memory startup!");
    }

    public void shutdown(){
        System.out.println("memory shutdown!");
    }
}

public class Disk {

    public void startup(){
        System.out.println("disk startup!");
    }

    public void shutdown(){
        System.out.println("disk shutdown!");
    }
}

public class Computer {
    private CPU cpu;
    private Memory memory;
    private Disk disk;

    public Computer(){
        cpu = new CPU();
        memory = new Memory();
        disk = new Disk();
    }

    public void startup(){
        System.out.println("start the computer!");
        cpu.startup();
        memory.startup();
        disk.startup();
        System.out.println("start computer finished!");
    }

    public void shutdown(){
        System.out.println("begin to close the computer!");
        cpu.shutdown();
        memory.shutdown();
        disk.shutdown();

        System.out.println("computer closed!");
```

```
    }
}
```

User类如下：

```
public class User {

    public static void main(String[] args) {
        Computer computer = new Computer();
        computer.startup();
        computer.shutdown();
    }
}
```

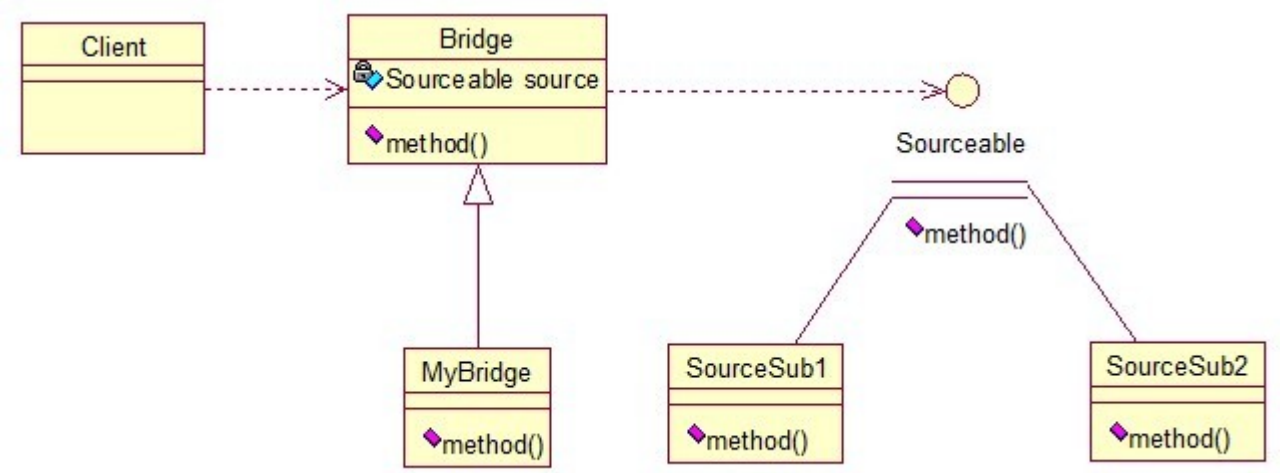
输出：

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：

```
public interface Sourceable {
    public void method();
}
```

分别定义两个实现类：

```
public class SourceSub1 implements Sourceable {

    @Override
    public void method() {
        System.out.println("this is the first sub!");
    }
}
```

```
public class SourceSub2 implements Sourceable {

    @Override
    public void method() {
        System.out.println("this is the second sub!");
    }
}
```

定义一个桥，持有Sourceable的一个实例：

```
public abstract class Bridge {
    private Sourceable source;

    public void method(){
        source.method();
    }

    public Sourceable getSource() {
        return source;
    }

    public void setSource(Sourceable source) {
        this.source = source;
    }
}
```

```
public class MyBridge extends Bridge {
    public void method(){
        getSource().method();
    }
}
```

测试类：

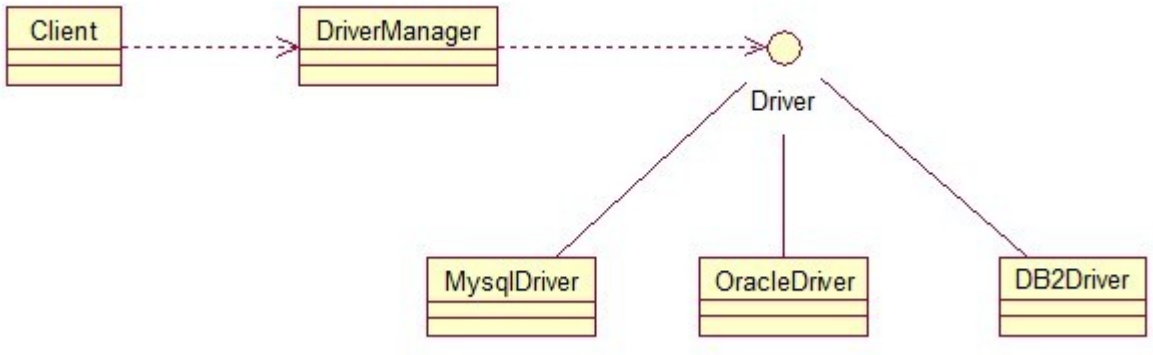
```
public class BridgeTest {

    public static void main(String[] args) {

        Bridge bridge = new MyBridge();

        /*调用第一个对象*/

        Sourceable source1 = new SourceSub1();
```

```
        bridge.setSource(source1);
        bridge.method();

        /*调用第二个对象*/
        Sourceable source2 = new SourceSub2();
        bridge.setSource(source2);
        bridge.method();
    }
}
```

output：

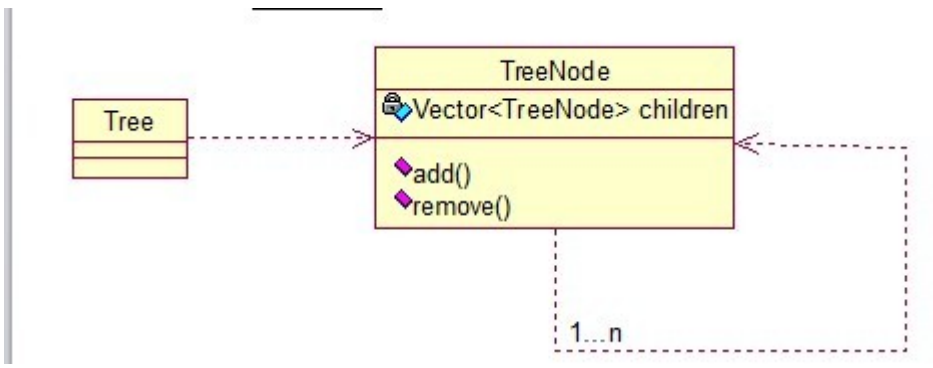
this is the first sub! this is the second sub!

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：

```
public class TreeNode {

    private String name;
    private TreeNode parent;
    private Vector<TreeNode> children = new Vector<TreeNode>();

    public TreeNode(String name){
        this.name = name;

    }
```

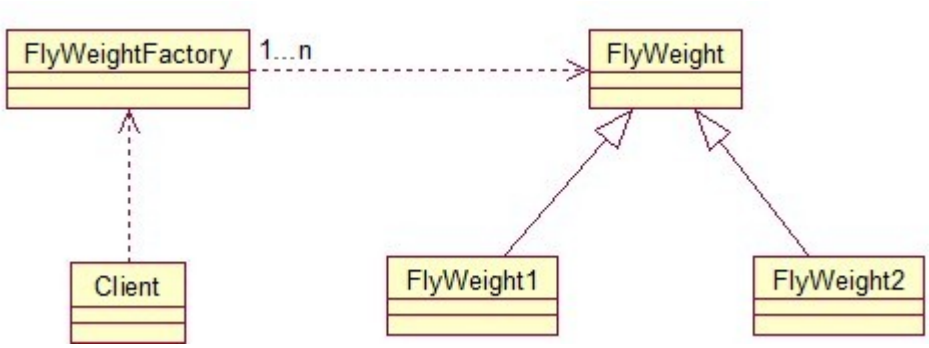
```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public TreeNode getParent() {
        return parent;
    }

    public void setParent(TreeNode parent) {
        this.parent = parent;
    }

    //添加孩子节点
    public void add(TreeNode node){
        children.add(node);
    }

    //删除孩子节点
    public void remove(TreeNode node){
        children.remove(node);
    }

    //取得孩子节点
    public Enumeration<TreeNode> getChildren(){
        return children.elements();
    }
}
```

```
public class Tree {

    TreeNode root = null;

    public Tree(String name) {
        root = new TreeNode(name);
    }

    public static void main(String[] args) {
        Tree tree = new Tree("A");
        TreeNode nodeB = new TreeNode("B");
        TreeNode nodeC = new TreeNode("C");

        nodeB.add(nodeC);
        tree.root.add(nodeB);
        System.out.println("build the tree finished!");
    }
}
```

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。

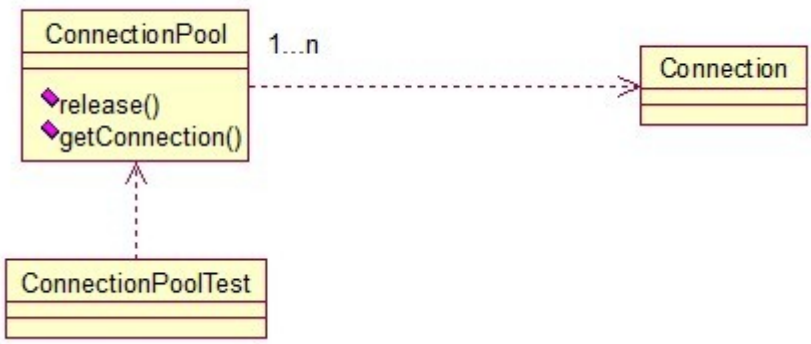
12、享元模式（Flyweight）

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：

```
public class ConnectionPool {

    private Vector<Connection> pool;

    /*公有属性*/
    private String url = "jdbc:mysql://localhost:3306/test";
    private String username = "root";
    private String password = "root";
    private String driverClassName = "com.mysql.jdbc.Driver";

    private int poolSize = 100;
    private static ConnectionPool instance = null;
    Connection conn = null;
```

```
    /*构造方法，做一些初始化工作*/
    private ConnectionPool() {
        pool = new Vector<Connection>(poolSize);

        for (int i = 0; i < poolSize; i++) {
            try {
                Class.forName(driverClassName);
                conn = DriverManager.getConnection(url, username, password);
                pool.add(conn);
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

    /* 返回连接到连接池 */
    public synchronized void release() {
        pool.add(conn);
    }

    /* 返回连接池中的一个数据库连接 */
    public synchronized Connection getConnection() {
        if (pool.size() > 0) {
            Connection conn = pool.get(0);
            pool.remove(conn);
            return conn;
        } else {
            return null;
        }
    }
}
```

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！本章讲解了7种结构型模式，因为篇幅的问题，剩下的11种行为型模式，

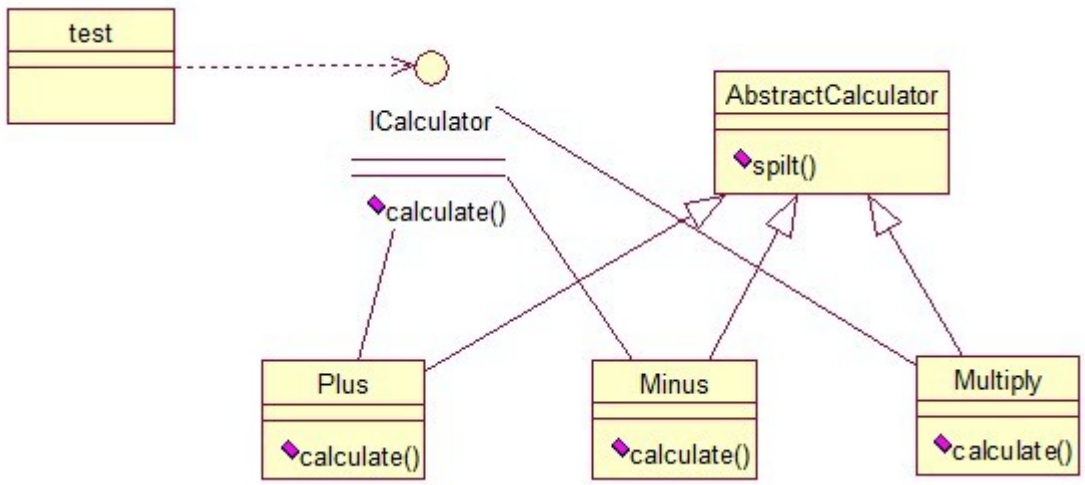
本章是关于设计模式的最后一讲，会讲到第三种设计模式——行为型模式，共11种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

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

13、策略模式（strategy）

策略模式定义了一系列算法，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：

```
public interface ICalculator {
    public int calculate(String exp);
}
```

辅助类：

```
public abstract class AbstractCalculator {

    public int[] split(String exp,String opt){
        String array[] = exp.split(opt);
        int arrayInt[] = new int[2];
        arrayInt[0] = Integer.parseInt(array[0]);
        arrayInt[1] = Integer.parseInt(array[1]);
        return arrayInt;
    }
}
```

三个实现类：

```
public class Plus extends AbstractCalculator implements ICalculator {

    @Override
    public int calculate(String exp) {
        int arrayInt[] = split(exp,"\\+");
        return arrayInt[0]+arrayInt[1];
    }
}
```

```
public class Minus extends AbstractCalculator implements ICalculator {

    @Override
    public int calculate(String exp) {
        int arrayInt[] = split(exp,"-");
        return arrayInt[0]-arrayInt[1];
    }

}
```

```
public class Multiply extends AbstractCalculator implements ICalculator {

    @Override
    public int calculate(String exp) {
        int arrayInt[] = split(exp,"\\*");
        return arrayInt[0]*arrayInt[1];
    }
}
```

简单的测试类：

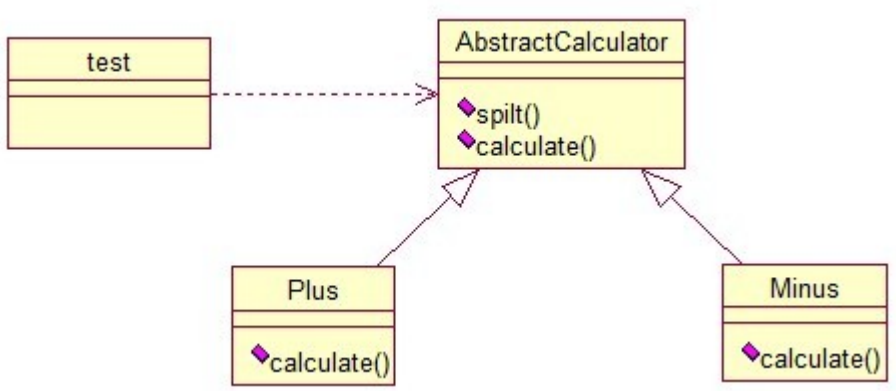
```
public class StrategyTest {

    public static void main(String[] args) {
        String exp = "2+8";
        ICalculator cal = new Plus();
        int result = cal.calculate(exp);
        System.out.println(result);
    }
}
```

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：

```
public abstract class AbstractCalculator {

    /*主方法，实现对本类其它方法的调用*/
    public final int calculate(String exp,String opt){
        int array[] = split(exp,opt);
        return calculate(array[0],array[1]);
    }

    /*被子类重写的方法*/
    abstract public int calculate(int num1,int num2);

    public int[] split(String exp,String opt){
        String array[] = exp.split(opt);
        int arrayInt[] = new int[2];
        arrayInt[0] = Integer.parseInt(array[0]);
        arrayInt[1] = Integer.parseInt(array[1]);
        return arrayInt;

    }
```

```
}
```

```
public class Plus extends AbstractCalculator {

    @Override
    public int calculate(int num1,int num2) {
        return num1 + num2;
    }
}
```

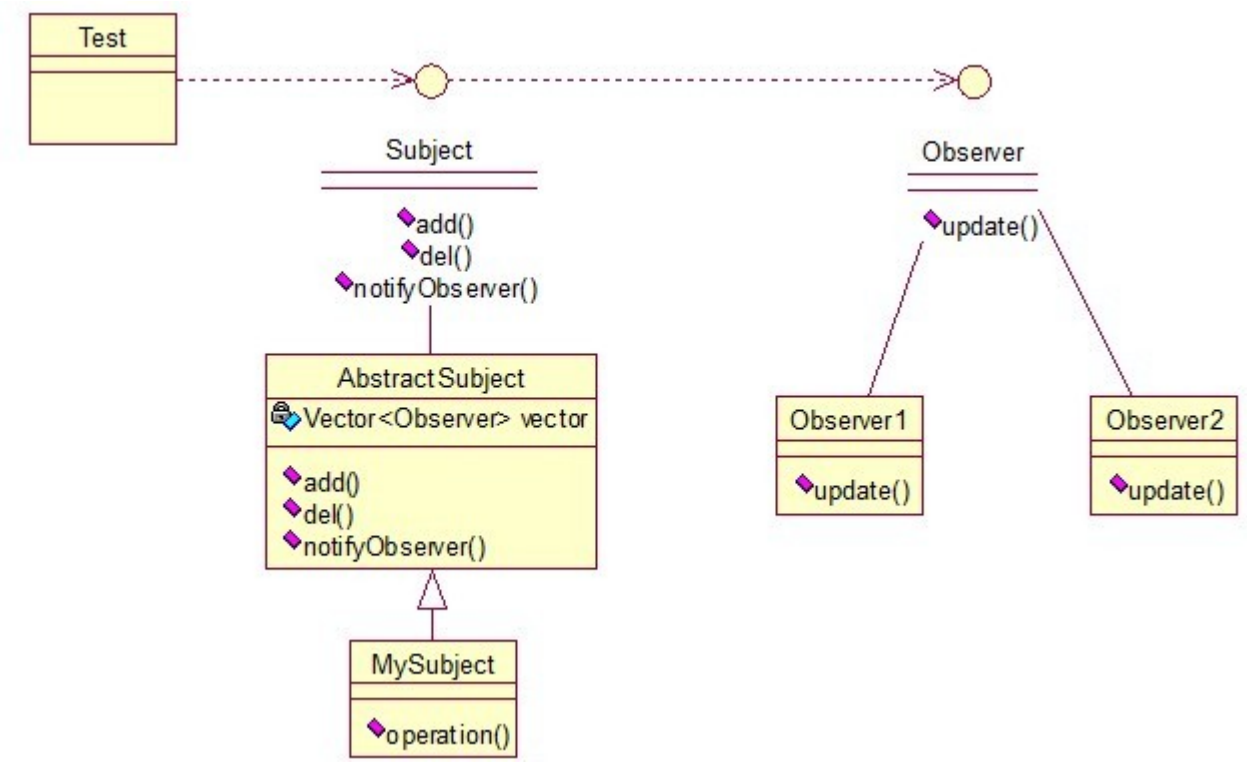
测试类：

```
public class StrategyTest {

    public static void main(String[] args) {
        String exp = "8+8";
        AbstractCalculator cal = new Plus();
        int result = cal.calculate(exp, "\\+");
        System.out.println(result);
    }
}
```

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。

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：

```
public interface Observer {
    public void update();
}
```

两个实现类：

```
public class Observer1 implements Observer {

    @Override
    public void update() {
        System.out.println("observer1 has received!");
    }
}
```

```
public class Observer2 implements Observer {

    @Override
    public void update() {
        System.out.println("observer2 has received!");
    }

}
```

Subject接口及实现类：

```
public interface Subject {

    /*增加观察者*/
    public void add(Observer observer);

    /*删除观察者*/
    public void del(Observer observer);

    /*通知所有的观察者*/
    public void notifyObservers();

    /*自身的操作*/
    public void operation();
}
```

```
public abstract class AbstractSubject implements Subject {

    private Vector<Observer> vector = new Vector<Observer>();
    @Override
    public void add(Observer observer) {
        vector.add(observer);
    }

    @Override
    public void del(Observer observer) {
        vector.remove(observer);
    }

    @Override
    public void notifyObservers() {
        Enumeration<Observer> enumo = vector.elements();
        while(enumo.hasMoreElements()){
            enumo.nextElement().update();
        }
    }
}
```

```
public class MySubject extends AbstractSubject {

    @Override
    public void operation() {
        System.out.println("update self!");
        notifyObservers();
    }

}
```

测试类

```
public class ObserverTest {

    public static void main(String[] args) {
        Subject sub = new MySubject();
        sub.add(new Observer1());
        sub.add(new Observer2());

        sub.operation();
    }

}
```

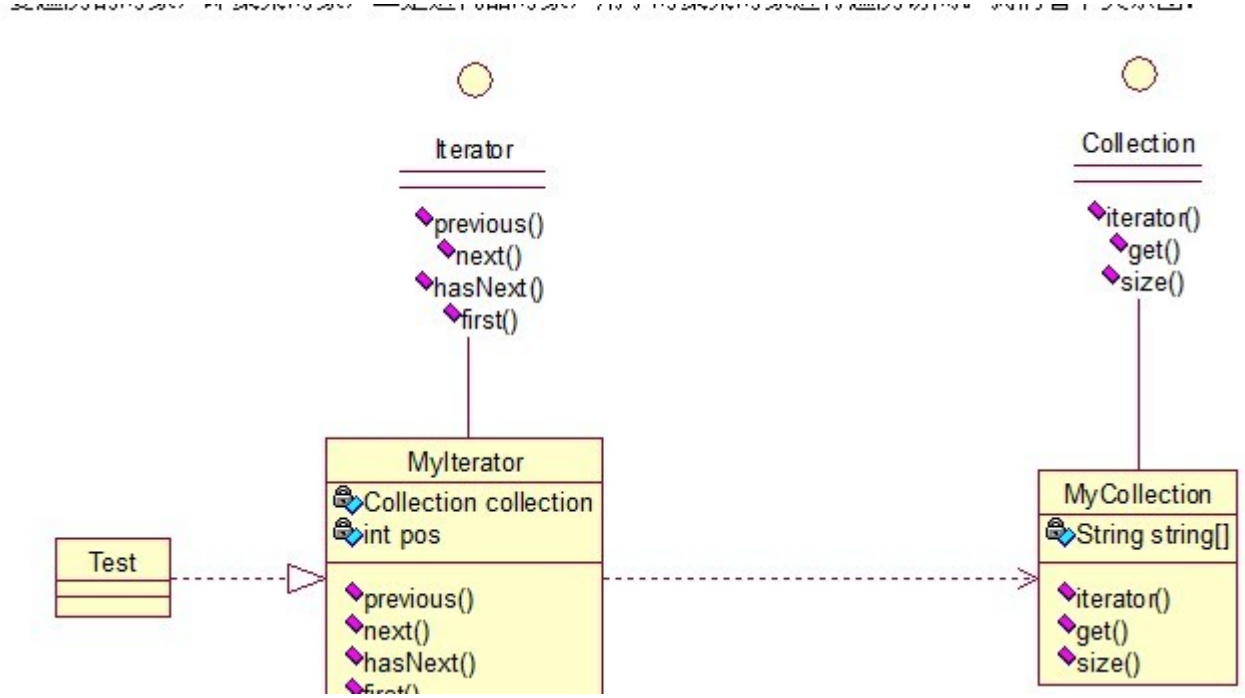
输出：

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：

```
public interface Collection {

    public Iterator iterator();

    /*取得集合元素*/
    public Object get(int i);

    /*取得集合大小*/
    public int size();
}
```

```
public interface Iterator {
    //前移
    public Object previous();

    //后移
    public Object next();
    public boolean hasNext();

    //取得第一个元素
    public Object first();
}
```

两个实现：

```
public class MyCollection implements Collection {

    public String string[] = {"A","B","C","D","E"};
    @Override
    public Iterator iterator() {
        return new MyIterator(this);
    }

    @Override
    public Object get(int i) {
        return string[i];
    }

    @Override
    public int size() {
        return string.length;
    }
}
```

```
public class MyIterator implements Iterator {

    private Collection collection;
    private int pos = -1;

    public MyIterator(Collection collection){
        this.collection = collection;
```

```
    }

    @Override
    public Object previous() {
        if(pos > 0){
            pos--;
        }
        return collection.get(pos);
    }

    @Override
    public Object next() {
        if(pos<collection.size()-1){
            pos++;
        }
        return collection.get(pos);
    }

    @Override
    public boolean hasNext() {
        if(pos<collection.size()-1){
            return true;
        }else{
            return false;
        }
    }

    @Override
    public Object first() {
        pos = 0;
        return collection.get(pos);
    }

}
```

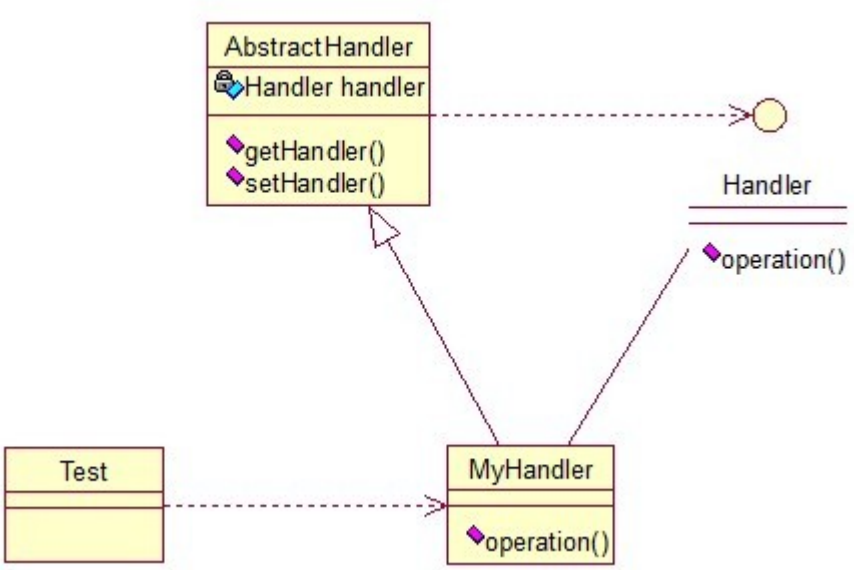
测试类：

```
public class Test {

    public static void main(String[] args) {
        Collection collection = new MyCollection();
        Iterator it = collection.iterator();

        while(it.hasNext()){
            System.out.println(it.next());
        }
    }
}
```

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。

```
public interface Handler {
    public void operator();
}
```

```
public abstract class AbstractHandler {

    private Handler handler;

    public Handler getHandler() {
        return handler;
    }

    public void setHandler(Handler handler) {
        this.handler = handler;
    }

}
```

```
public class MyHandler extends AbstractHandler implements Handler {

    private String name;


    public MyHandler(String name) {
```

```
        this.name = name;
    }

    @Override
    public void operator() {
        System.out.println(name+"deal!");
        if(getHandler()!=null){
            getHandler().operator();
        }
    }
}
```

```
public class Test {

    public static void main(String[] args) {
        MyHandler h1 = new MyHandler("h1");
        MyHandler h2 = new MyHandler("h2");
        MyHandler h3 = new MyHandler("h3");

        h1.setHandler(h2);
        h2.setHandler(h3);

        h1.operator();
    }
}
```

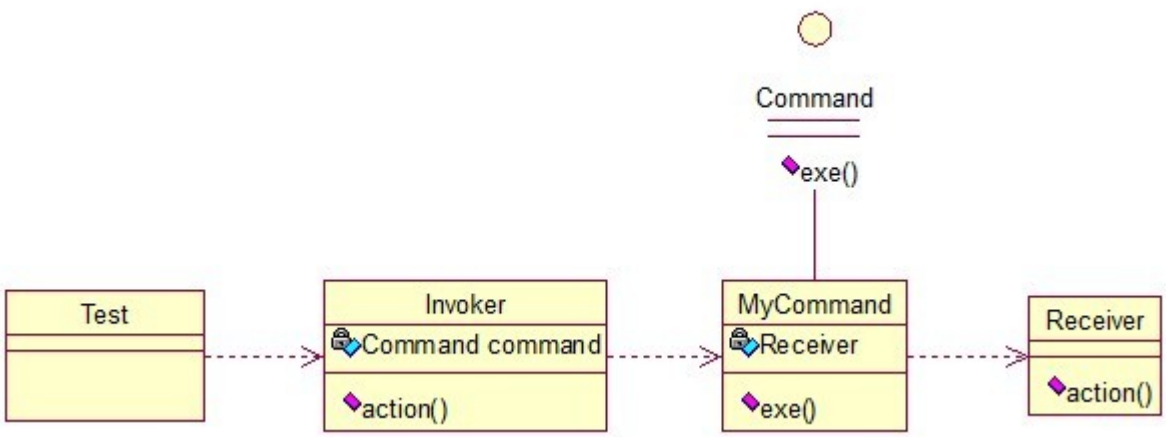
输出：

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：

```
public interface Command {
    public void exe();
}
```

```
public class MyCommand implements Command {

    private Receiver receiver;

    public MyCommand(Receiver receiver) {
        this.receiver = receiver;
    }

    @Override
    public void exe() {
        receiver.action();
    }
}
```

```
public class Receiver {
    public void action(){
        System.out.println("command received!");
    }
}
```

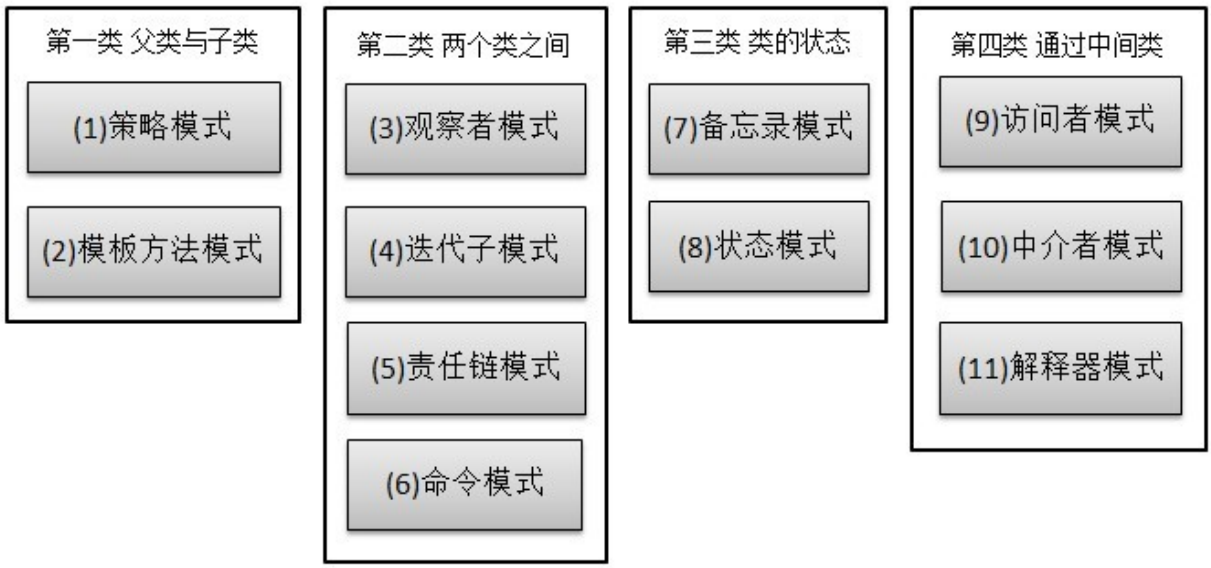
```
public class Invoker {

    private Command command;

    public Invoker(Command command) {
        this.command = command;
    }

    public void action(){
        command.exe();
    }
}
```

```
public class Test {

    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        Command cmd = new MyCommand(receiver);
        Invoker invoker = new Invoker(cmd);
        invoker.action();
    }
}
```

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！

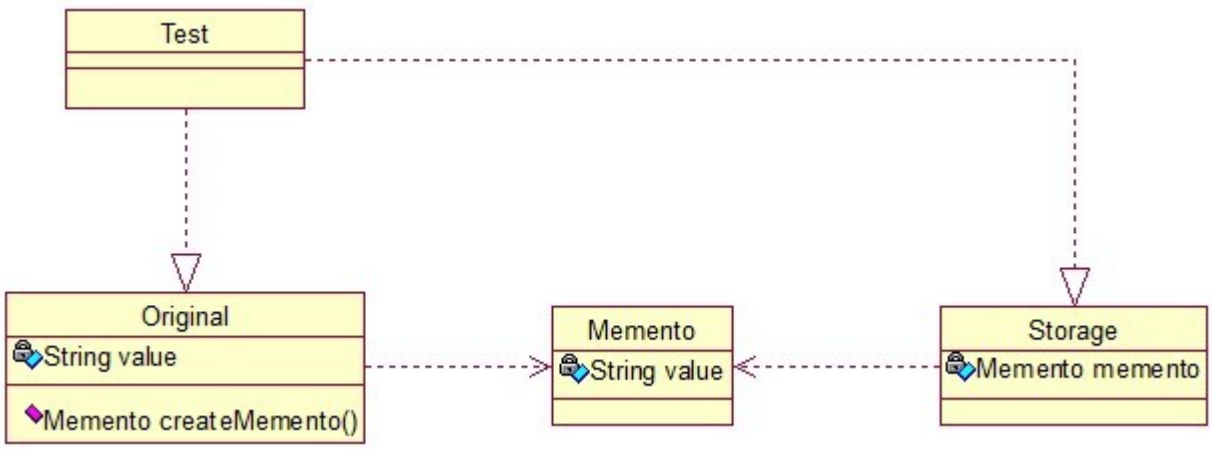
其实每个设计模式都是很重要的一种思想，看上去很熟，其实是因为我们在学到的东西中都有涉及，尽管有时我们并不知道，其实在Java本身的设计之中处处都有体现，像AWT、JDBC、集合类、IO管道或者是Web框架，里面设计模式无处不在。因为我们篇幅有限，很难讲每一个设计模式都讲的很详细，不过我会尽我所能，尽量在有限的空间和篇幅内，把意思写清楚了，更好让大家明白。本章不出意外的话，应该是设计模式最后一讲了，首先还是上一下上篇开头的那个图：



本章讲讲第三类和第四类。

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：

```
public class Original {
```

```
    private String value;

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    public Original(String value) {
        this.value = value;
    }

    public Memento createMemento(){
        return new Memento(value);
    }

    public void restoreMemento(Memento memento){
        this.value = memento.getValue();
    }
}
```

```
public class Memento {

    private String value;

    public Memento(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}
```

```
public class Storage {

    private Memento memento;

    public Storage(Memento memento) {
        this.memento = memento;
    }

    public Memento getMemento() {
        return memento;

    }
```

```
    public void setMemento(Memento memento) {
        this.memento = memento;
    }
}
```

测试类：

```
public class Test {

    public static void main(String[] args) {

        // 创建原始类
        Original origi = new Original("egg");

        // 创建备忘录
        Storage storage = new Storage(origi.createMemento());

        // 修改原始类的状态
        System.out.println("初始化状态为：" + origi.getValue());
        origi.setValue("niu");
        System.out.println("修改后的状态为：" + origi.getValue());

        // 回复原始类的状态
        origi.restoreMemento(storage.getMemento());
        System.out.println("恢复后的状态为：" + origi.getValue());
    }
}
```

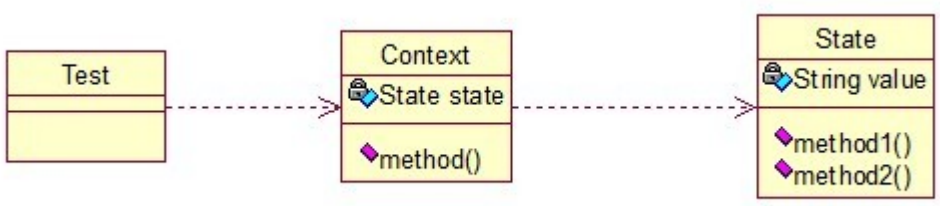
输出：

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：

```
package com.xtfggef.dp.state;
```

```java
/**
 * 状态类的核心类
 * 2012-12-1
 * @author erqing
 *
 */
public class State {

    private String value;

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    public void method1(){
        System.out.println("execute the first opt!");
    }

    public void method2(){
        System.out.println("execute the second opt!");
    }
}
```

```java
package com.xtfggef.dp.state;

/**
 * 状态模式的切换类   2012-12-1
 * @author erqing
 *
 */
public class Context {

    private State state;

    public Context(State state) {
        this.state = state;
    }

    public State getState() {
        return state;
    }

    public void setState(State state) {
        this.state = state;
    }

    public void method() {
        if (state.getValue().equals("state1")) {
            state.method1();
```

```
        } else if (state.getValue().equals("state2")) {
            state.method2();
        }
    }
}
```

测试类：

```
public class Test {

    public static void main(String[] args) {

        State state = new State();
        Context context = new Context(state);

        //设置第一种状态
        state.setValue("state1");
        context.method();

        //设置第二种状态
        state.setValue("state2");
        context.method();
    }
}
```

输出：

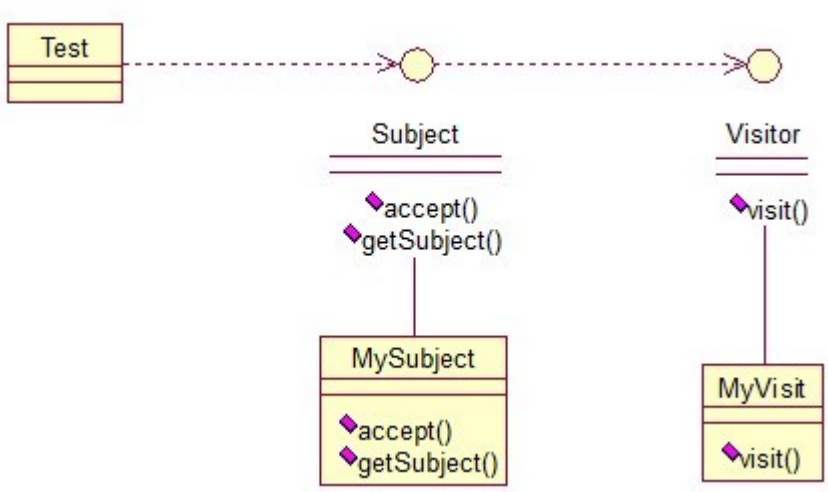
execute the first opt!

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。 **21、访问者模式（Visitor）**

访问者模式把数据结构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，

```
public interface Visitor {
    public void visit(Subject sub);
}
```

```
public class MyVisitor implements Visitor {

    @Override
    public void visit(Subject sub) {
        System.out.println("visit the subject："+sub.getSubject());
    }
}
```

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，

```
public interface Subject {
    public void accept(Visitor visitor);
    public String getSubject();
}
```

```
public class MySubject implements Subject {

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }

    @Override
    public String getSubject() {
        return "love";
    }
}
```

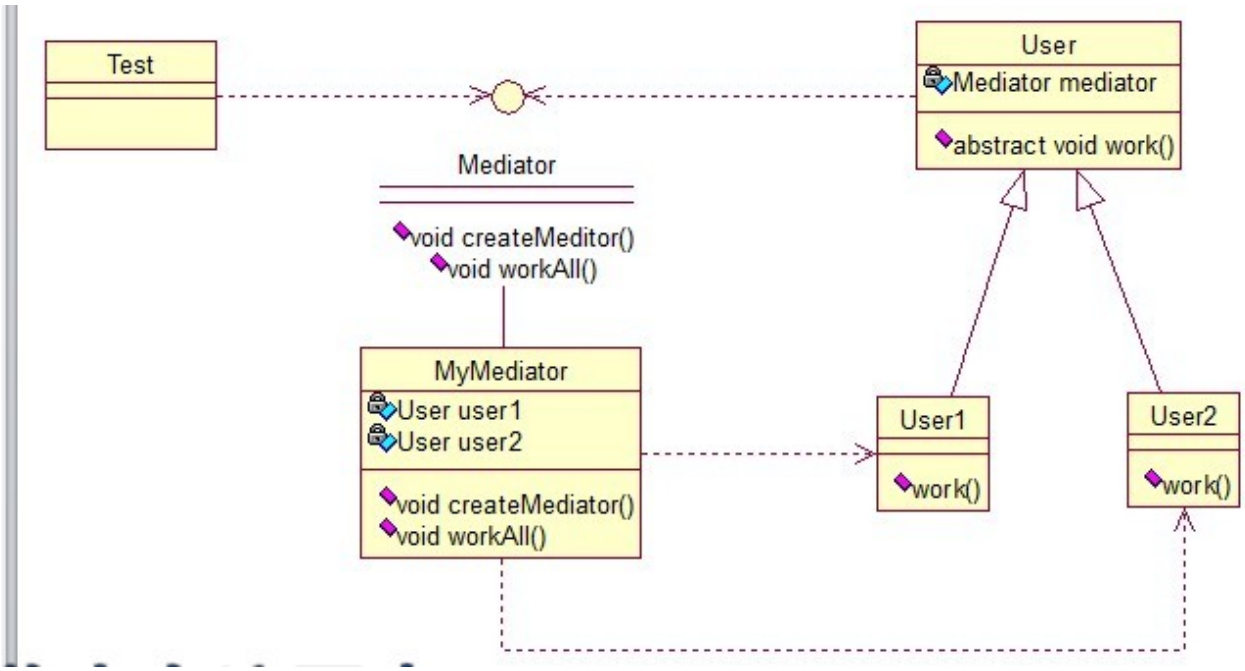
测试：

```
public class Test {

    public static void main(String[] args) {

        Visitor visitor = new MyVisitor();
        Subject sub = new MySubject();
        sub.accept(visitor);
    }
}
```

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：

```
public interface Mediator {
    public void createMediator();
    public void workAll();
}
```

```
public class MyMediator implements Mediator {
```

```
    private User user1;
    private User user2;

    public User getUser1() {
        return user1;
    }
    public User getUser2() {
        return user2;
    }

    @Override
    public void createMediator() {
        user1 = new User1(this);
        user2 = new User2(this);
    }

    @Override
    public void workAll() {
        user1.work();
        user2.work();
    }
}
```

```
public abstract class User {

    private Mediator mediator;

    public Mediator getMediator(){
        return mediator;
    }

    public User(Mediator mediator) {
        this.mediator = mediator;
    }

    public abstract void work();
}
```

```
public class User1 extends User {

    public User1(Mediator mediator){
        super(mediator);
    }

    @Override
    public void work() {
        System.out.println("user1 exe!");
    }
}
```

```
public class User2 extends User {

    public User2(Mediator mediator){
        super(mediator);
    }

    @Override
    public void work() {
        System.out.println("user2 exe!");
    }
}
```

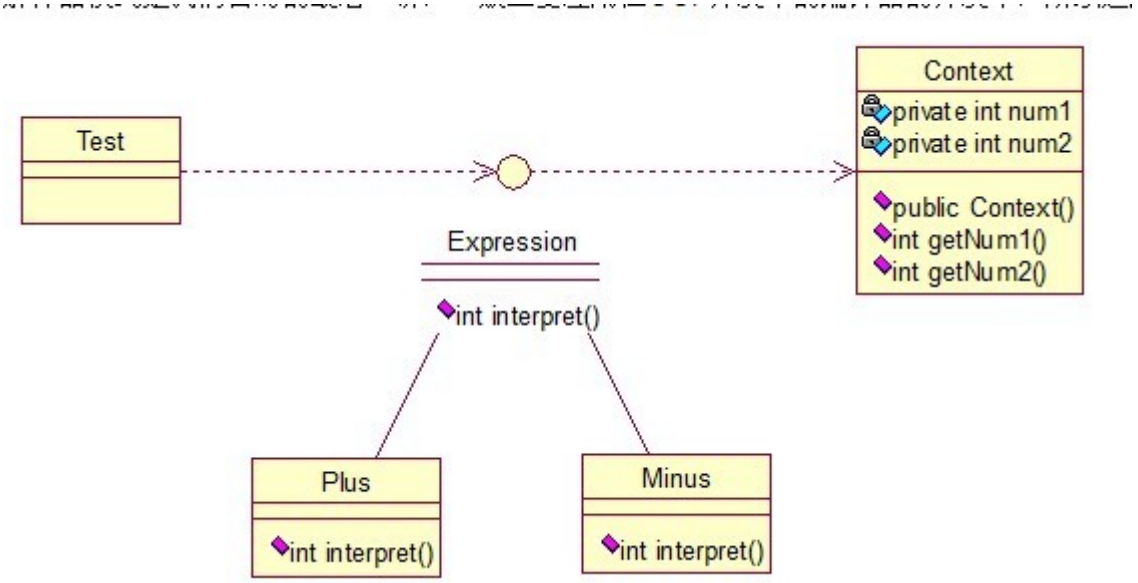
测试类：

```
public class Test {

    public static void main(String[] args) {
        Mediator mediator = new MyMediator();
        mediator.createMediator();
        mediator.workAll();
    }
}
```

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：

```
public interface Expression {
    public int interpret(Context context);
}
```

```
public class Plus implements Expression {

    @Override
    public int interpret(Context context) {
        return context.getNum1()+context.getNum2();
    }
}
```

```
public class Minus implements Expression {

    @Override
    public int interpret(Context context) {
        return context.getNum1()-context.getNum2();
    }
}
```

```
public class Context {

    private int num1;
    private int num2;

    public Context(int num1, int num2) {
        this.num1 = num1;
        this.num2 = num2;
    }

    public int getNum1() {
        return num1;
    }
    public void setNum1(int num1) {
        this.num1 = num1;
    }
    public int getNum2() {
        return num2;
    }
    public void setNum2(int num2) {
        this.num2 = num2;
    }

}
```

```
public class Test {

    public static void main(String[] args) {

        // 计算9+2-8的值
        int result = new Minus().interpret((new Context(new Plus()
                .interpret(new Context(9, 2)), 8)));
        System.out.println(result);
    }
}
```

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！