Java之美[从菜鸟到高手演变]之设计模式三

2012年11月30日 22:50:33 终点 阅读数：122925 标签： java 基础 系统架构

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。这段时间一直在写关于设计模式的东西，终于写到一半了，写博文是个很费时间的东西，因为我得为读者负责，不论是图还是代码还是表述，都希望能尽量写清楚，以便读者理解，我想不论是我还是读者，都希望看到高质量的博文出来，从我本人出发，我会一直坚持下去，不断更新，源源动力来自于读者朋友们的不断支持，我会尽自己的努力，写好每一篇文章！希望大家能不断给出意见和建议，共同打造完美的博文！

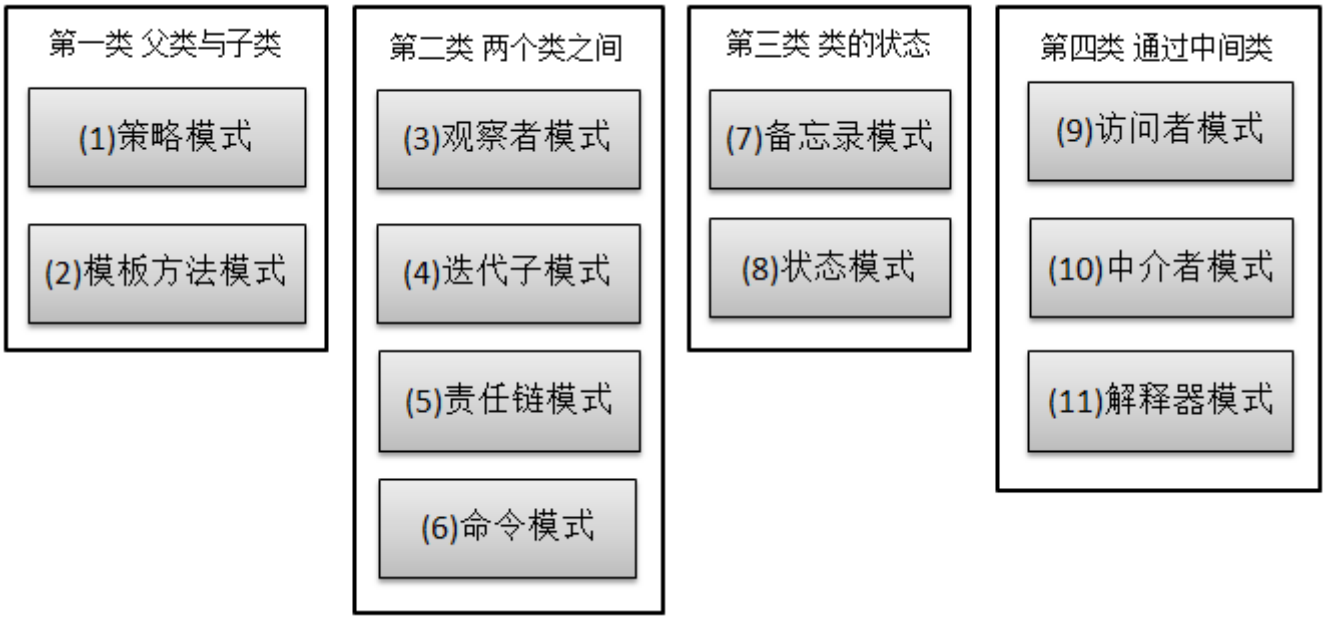
学会技术，懂得分享！

有任何想法，请联系：egg

email:xtfggef@gmail.com 微博：http://weibo.com/xtfggef

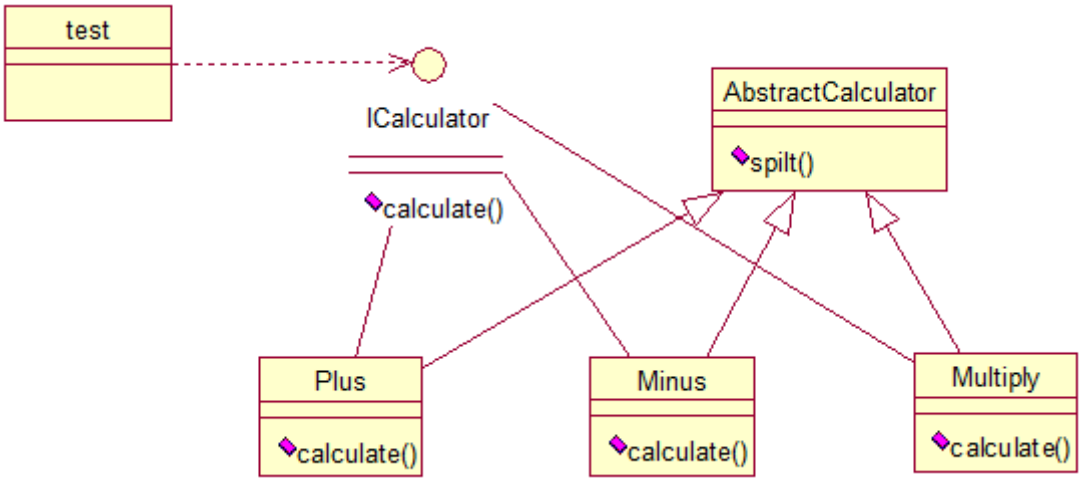
先来张图，看看这11中模式的关系：

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);
```

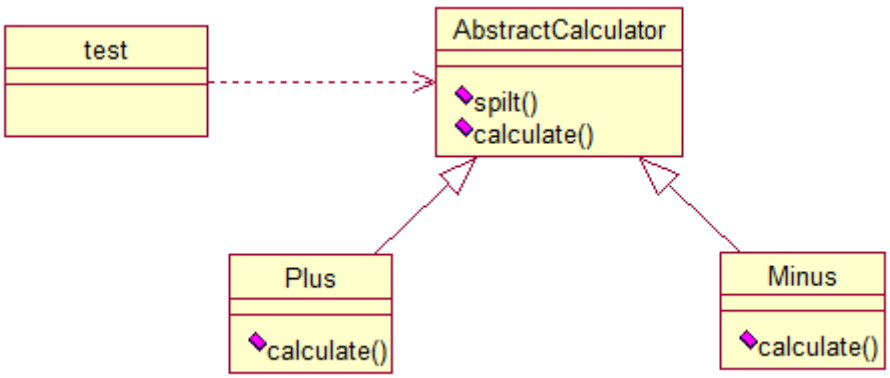
```
        }
    }
```

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;
        }
}
```

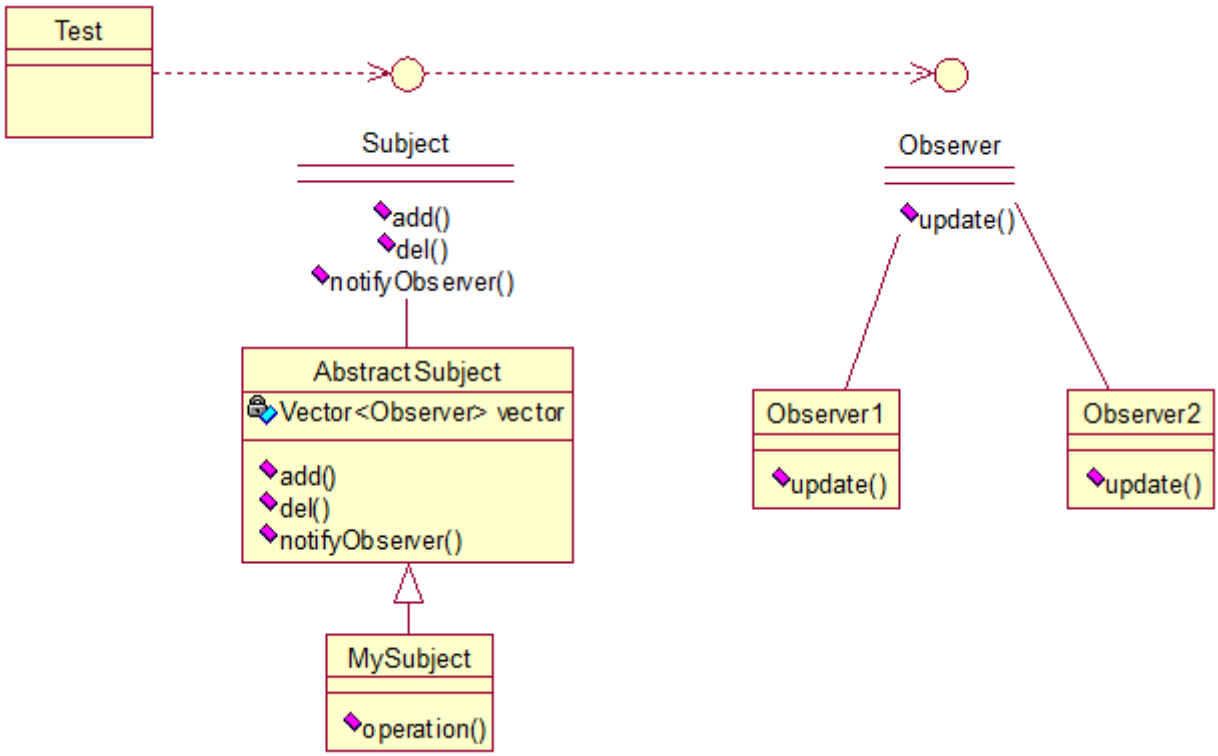
测试类：

```
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！**

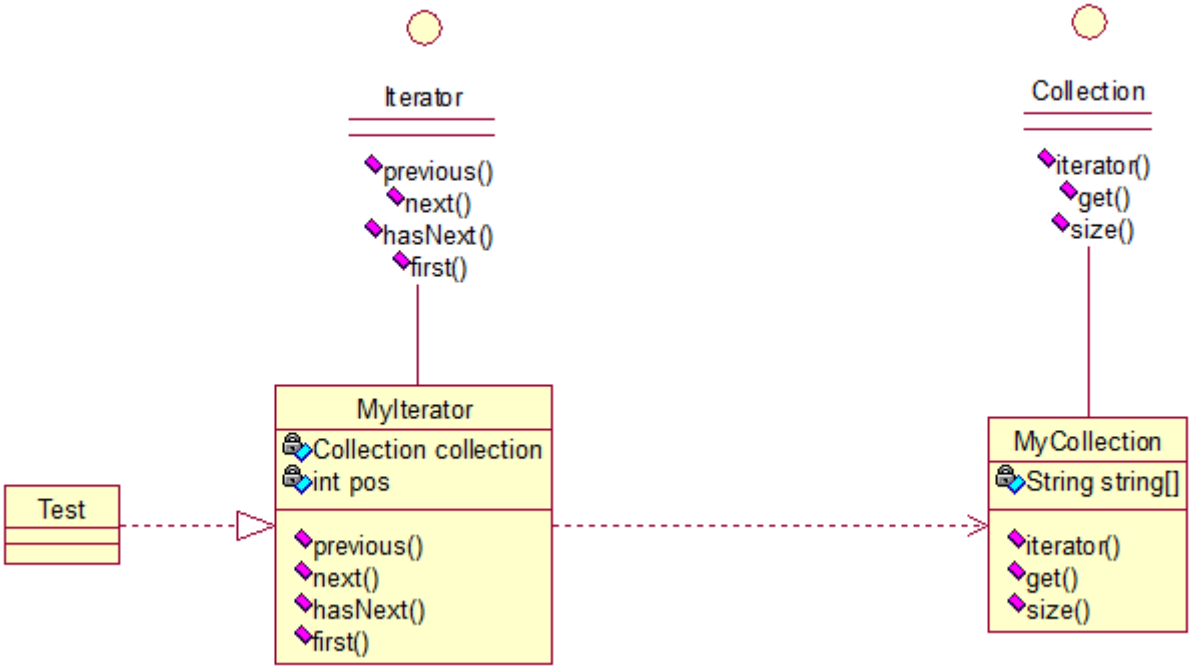
欢迎广大读者随时指正，一起讨论，一起进步！

有问题，联系：egg

email：xtfggef@gmail.com 微博：http://weibo.com/xtfggef

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();
```

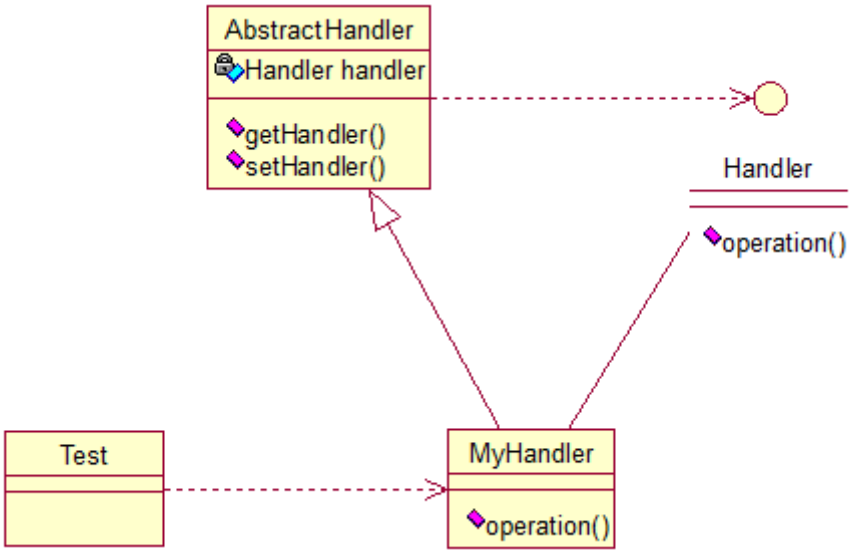
```
                Iterator it = collection.iterator();

                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();
        }
}
```

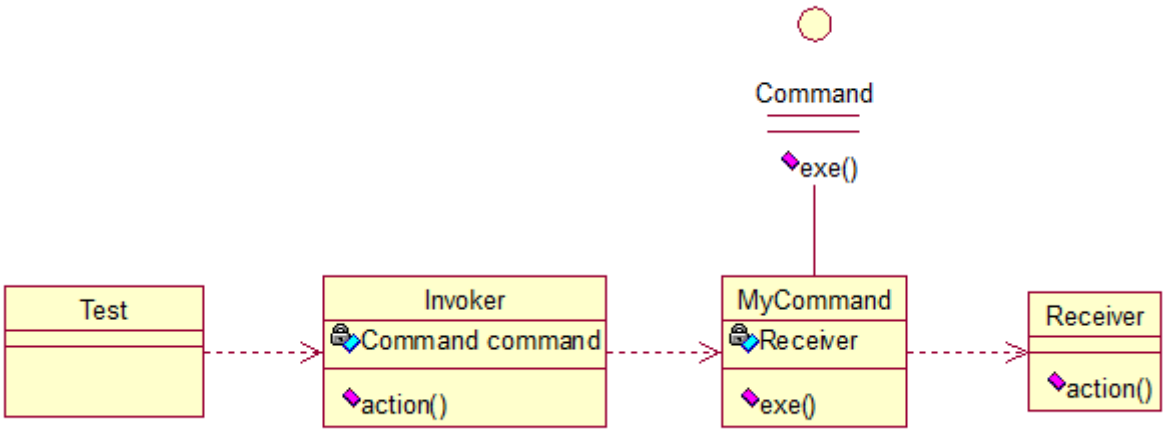
输出：

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！

本篇暂时就到这里，因为考虑到将来博文会不断的更新，不断的增加新内容，所以当前篇幅不易过长，以便大家阅读，所以接下来的放到另一篇里。敬请关注！