链滴

Java经典设计模式（3）：十一种行为型模式

作者：Hassan

原文链接：https://ld246.com/article/1471922433784

来源网站：链滴

<h2 id="toc_h2_0">设计模式原则</h2>
Java经典设计模式共有21中，分为三大类：创建型模式（5种）、结构型模式（7种）和行为型式（11种）。
本文主要讲行为型模式，创建型模式和结构型模式可以看博主的另外两篇文章：Java经典设计模之五大创建型模式（附实例和详解）、 Java经典设计模式之七大结构型模式（附实例和详解）。
行为型模式细分为如下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。
接下来对11种行为型模式逐个进行介绍。
<h3 id="toc_h3_1">一、策略模式</h3>
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，视实际需求是否添加），提供辅助函数。
首先统一接口：

<pre>package com.model.behaviour;
public interface ICalculator {
public int calculate(String <pan>exp);
} </pre>

辅助类：

<pre>package com.model.behaviour;
public abstract class AbstractCaculator {
<pre><code class="highlight-chroma">public int[] split(String exp, String &t;span class="pl-v">opt) {
    String array[] = exp.split(opt);
    int arrayInt[] = new int[2];
    arrayInt[<spa class="pl-c1">0] = Integer.parseInt(array[0]);
    arrayInt[<spa class="pl-c1">1] = Integer.parseInt(array[1]);
    return arrayInt;
}
</code></pre>
} </pre>

三个实现类：

<pre>package com.model.behaviour;
public class Plus extends</spa
> AbstractCalculator implements ICalculator
{
<pre><code class="highlight-chroma"><span class="highlight
cl">@Override
<span class="pl
k">public int <span class="pl-en"&
t;calculate(String <span class="pl-v
>exp) {
    <span class=
pl-k">int arrayInt[] = split(exp, <sp
n class="pl-s">"\\<
span>+");
    <span class=
pl-k">return arrayInt[0] <span clas
="pl-k">+ arrayInt[1];
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class Minus extends</sp
n> AbstractCalculator implements ICalculator</spa
> {
<pre><code class="highlight-chroma"><span class="highlight
cl">@Override
<span class="pl
k">public int <span class="pl-en"&
t;calculate(String <span class="pl-v
>exp) {
    <span class=
pl-k">int arrayInt[] = split(exp, <sp
n class="pl-s">"\\<
span>-");
    <span class=
pl-k">return arrayInt[0] <span clas
="pl-k">- arrayInt[1];
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class Multiply extends</
pan> AbstractCalculator implements ICalculator</s
an> {
<pre><code class="highlight-chroma"><span class="highlight
cl">@Override
<span class="pl
k">public int <span class="pl-en"&

t;calculate(String <span class="pl-v
>exp) {
    <span class=
pl-k">int arrayInt[] = split(exp,<spa
 class="pl-s">"\\</
pan>*");
    <span class=
pl-k">return arrayInt[0]<span clas
="pl-k">*arrayInt[1];
}
</code></pre>
} </pre>

测试类：

<pre>package com.model.behaviour;
public class StrategyTest {
<pre><code class="highlight-chroma"><span class="highlight
cl">public static</span&gt
 void main(&lt
span class="pl-k">String[] args) {
    <span class=
pl-smi">String exp = <span class=
pl-s">"8-2"</spa
>;
    <span class=
pl-smi">ICalculator cal = <span cla
s="pl-k">new Minus();
    <span class=
pl-k">int result = cal<span class="p
-k">.calculate(exp);
    <span class=
pl-smi">System.out<span class="pl
k">.println(exp + <span class="pl-s
>"="&lt
/span> + result);
}
</code></pre>
} </pre>

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
。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。
<h3 id="toc_h3_2"><a class="anchor" href="https://ld246.com/forward?goto=https%3A%2
%2Fgithub.com%2FHassanChiang%2FBlog%2Fblob%2Fmaster%2Fsource%2F_posts%2F2016
3%2Fjava-pattern-3.md%23%E4%BA%8C%E6%A8%A1%E6%9D%BF%E6%96%B9%E6%B3%9
%E6%A8%A1%E5%BC%8F" target="_blank" rel="nofollow ugc">二、模板方法模式</h3>
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
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
的调用。
就是在AbstractCalculator类中定义一个主方法calculate，calculate()调用spilt()等，Plus和Min
s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
：

<pre>package com.model.behaviour;

public abstract class AbstractCalculator {
<pre><code class="highlight-chroma">/*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],array1]);
}

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

public int[] split(String <span class="pl-v"gt;exp,String <span class="pl-v"&gt opt){
    String array[] = exp.split(opt);
    int arrayInt[] = new int[2];
    arrayInt[<spa class="pl-c1">0] = Integer.parseInt(array[0]);
    arrayInt[<spa class="pl-c1">1] = Integer.parseInt(array[1]);
    return arrayInt;
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class Plus extends AbstractCalculator {

<pre><code class="highlight-chroma"><span class="highlight
cl">@Override
<span class="pl
k">public int <span class="pl-en"&
t;calculate(int n
m1,int num2&lt
/span>) {
    <span class=
pl-k">return num1 + num2;
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class StrategyTest {
<pre><code class="highlight-chroma"><span class="highlight
cl">public static</span&gt
 void main(&lt
span class="pl-k">String[] args) {
    <span class=
pl-smi">String exp = <span class=
pl-s">"8+8"</spa
>;
    <span class=
pl-smi">AbstractCalculator cal = <
pan class="pl-k">new Plus();
    <span class=
pl-k">int result = cal<span class="p
-k">.calculate(exp, "<
span>\\+"</span
gt;);
    <span class=
pl-smi">System.out<span class="pl
k">.println(result);
}
</code></pre>
} </pre>

<h3 id="toc_h3_3"><a class="anchor" href="https://ld246.com/forward?goto=https%3A%2
%2Fgithub.com%2FHassanChiang%2FBlog%2Fblob%2Fmaster%2Fsource%2F_posts%2F2016
3%2Fjava-pattern-3.md%23%E4%B8%89%E8%A7%82%E5%AF%9F%E8%80%85%E6%A8%A1
E5%BC%8F" target="_blank" rel="nofollow ugc">三、观察者模式</h3>
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。
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R
S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
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
的关系。

<pre>package com.model.behaviour;
public interface Observer {
public void update();
} </pre>

<pre><code>``` java
package com.model.behaviour;
public class Observer1 implements Observer {
<pre><code class="highlight-chroma"><span class="highlight
cl">@Override
public void updat
() {
    System.out.print
n("observer1 has received!");
}
</code></pre>
</code> <code>}
</code> </pre>

<pre>package com.model.behaviour;
public class Observer2 implem
nts Observer {
<pre><code class="highlight-chroma"><span class="highlight
cl">@Override
<span class="pl
k">public void <span class="pl-en
>update() {
    <span class=
pl-smi">System.out<span class="pl
k">.println("</span&
t;observer2 has received!");
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public interface Subject {
<pre><code class="highlight-chroma"><span class="highlight
cl">/*增加观察者*/
<span class="pl
k">public void <span class="pl-en
>add(Observer <span class="pl-v
>observer);

<span class="pl
c">/*删除观察者*/
<span class="pl
k">public void <span class="pl-en
>del(Observer <span class="pl-v
>observer);

<span class="pl
c">/*通知所有的观察者*/
<span class="pl
k">public void <span class="pl-en
>notifyObservers();

<span class="pl

c">/*自身的操作*/
<span class="pl
k">public void <span class="pl-en
>operation();
</code></pre>
} </pre>

<pre>package com.model.behaviour;
import java.util.Enumeration;
import java.util.Vector;
public abstract class AbstractSu
ject implements Subject {
<pre><code class="highlight-chroma"><span class="highlight
cl">private Vector<&l
;span class="pl-smi">Observer> vector <span class=
pl-k">= new <span class="pl-k"&gt
Vector<Observer>();
<span class="pl
k">@Override
<span class="pl
k">public void <span class="pl-en
>add(Observer <span class="pl-v
>observer) {
    vector<span c
ass="pl-k">.add(observer);
}

<span class="pl
k">@Override
<span class="pl
k">public void <span class="pl-en
>del(Observer <span class="pl-v
>observer) {
    vector<span c
ass="pl-k">.remove(observer);
}

<span class="pl
k">@Override
<span class="pl
k">public void <span class="pl-en
>notifyObservers() {
    <span class=
pl-k">Enumeration<Observer>&lt
/span> enumo = vector.&
t;/span>elements();
    <span class=
pl-k">while(enumo.hasMoreElements
)){
        enumo<sp
n class="pl-k">.nextElement().update(
;
    }

}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class MySubject extends
/span> AbstractSubject {
<pre><code class="highlight-chroma"><span class="highlight
cl">@Override
<span class="pl
k">public void <span class="pl-en
>operation() {
    <span class=
pl-smi">System.out<span class="pl
k">.println("</span&
t;update self!");
    notifyObservers
);
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class ObserverTest {
<pre><code class="highlight-chroma"><span class="highlight
cl">public static</span&gt
 void main(&lt
span class="pl-k">String[] args) {
    <span class=
pl-smi">Subject sub = <span class
"pl-k">new MySubject();
    sub<span cla
s="pl-k">.add(new <span class="pl
smi">Observer1());
    sub<span cla
s="pl-k">.add(new <span class="pl
smi">Observer2());

    sub<span cla
s="pl-k">.operation();
}
</code></pre>
} </pre>

运行结果：

<pre>update self!
observer1 has received!
observer2 has received!</pre>

也许看完实例之后还是比较抽象，再将文字描述和代码实例看一两遍吧，然后结合工作中看哪些景可以使用这种模式以加深理解。

<h3 id="toc_h3_4">四、迭代子模式</h3>
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，二迭代器对象，用于对聚集对象进行遍历访问。
具体来看看代码实例：

<pre>package com.model.behaviour;
public interface Collection {
<pre><code class="highlight-chroma">public Iterator</spa> iterator();

/* 取得集合元素 */
public Object get(int);

/* 取得集合大小 */
public int <span class="pl-en"&t;size();
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public interface Iterator {
// 前移
public Object previous();
<pre><code class="highlight-chroma">// 后移
public Object next();

public boolean hasNext();

// 取得第一个元素
public Object first();
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class MyCollection implments Collection {
<pre><code class="highlight-chroma">public String</spangt; string[] = { "A", <spa class="pl-s">"B", "", "D", <pan class="pl-s">"E };

@Override
public Iterator iterator() {
    return new MyIterator(this);
}

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

@Override
public int <span class="pl-en"&t;size() {
    return string.length;
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class MyIterator implments Iterator {
<pre><code class="highlight-chroma">private Collection</pan> collection;
private int pos <span class="pl-

">= -1</
pan>;

<span class="pl
k">public MyIterator(<span class
"pl-smi">Collection collection){
    <span class=
pl-v">this.collection <span class="p
-k">= collection;
}

<span class="pl
k">@Override
<span class="pl
k">public Object <span class="pl
en">previous() {
    <span class=
pl-k">if(pos > <span class=
pl-c1">0){
        pos<span c
ass="pl-k">--;
    }
    <span class=
pl-k">return collection.get(pos);
}

<span class="pl
k">@Override
<span class="pl
k">public Object <span class="pl
en">next() {
    <span class=
pl-k">if(pos<collection<spa
 class="pl-k">.size()-<span class="p
-c1">1){
        pos<span c
ass="pl-k">++;
    }
    <span class=
pl-k">return collection.get(pos);
}

<span class="pl
k">@Override
<span class="pl
k">public boolean <span class="pl
en">hasNext() {
    <span class=
pl-k">if(pos<collection<spa
 class="pl-k">.size()-<span class="p
-c1">1){
        <span clas
="pl-k">return true;
    }<span class=

pl-k">else{
        <span clas
="pl-k">return false;
    }
}

<span class="pl
k">@Override
<span class="pl
k">public Object <span class="pl
en">first() {
    pos <span cl
ss="pl-k">= 0;
    <span class=
pl-k">return collection.get(pos);
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class Test {
<pre><code class="highlight-chroma"><span class="highlight
cl">public static</span&gt
 void main(&lt
span class="pl-k">String[] args) {
    <span class=
pl-smi">Collection collection = <sp
n class="pl-k">new MyCollection();

    <span class=
pl-smi">Iterator it = (<span class=
pl-smi">Iterator) collection.iterator();
    <span class=
pl-k">while(it.hasNext()){
        <span clas
="pl-smi">System.out<span class=
pl-k">.println(it.next());
    }
}
</code></pre>
} </pre>

输出结果：

<pre>A
B
C
D
E</pre>

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
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
写出自己的集合类，甚至框架！

<h3 id="toc_h3_5">五、责任链模式</h3>
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，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。

<pre>package com.model.behaviour;
public interface Handler {
public void operator();
} </pre>

<pre>package com.model.behaviour;
public abstract class AbstractHndler {
<pre><code class="highlight-chroma">private Handler</spn> handler;

public Handler getHandler() {
    return handler;
}

public void setHandler(Handler handler) {
    this.handler = handler;
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class MyHandler extend AbstractHandler implements Handler</spa> {
<pre><code class="highlight-chroma">private String</spa> name;

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

}

<span class="pl
k">@Override
<span class="pl
k">public void <span class="pl-en
>operator() {
    <span class=
pl-smi">System.out<span class="pl
k">.println(name + <span class="pl
s">"deal!"</span
gt;);
    <span class=
pl-k">if (getHandler() != <span clas
="pl-c1">null) {
        getHandler()
lt;span class="pl-k">.operator();
    }
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class Test {
<pre><code class="highlight-chroma"><span class="highlight
cl">public static</span&gt
 void main(&lt
span class="pl-k">String[] args) {
    <span class=
pl-smi">MyHandler h1 = <span cla
s="pl-k">new MyHandler("h1"</
pan>);
    <span class=
pl-smi">MyHandler h2 = <span cla
s="pl-k">new MyHandler("h2"</
pan>);
    <span class=
pl-smi">MyHandler h3 = <span cla
s="pl-k">new MyHandler("h3"</
pan>);

    h1<span clas
="pl-k">.setHandler(h2);
    h2<span clas
="pl-k">.setHandler(h3);

    h1<span clas
="pl-k">.operator();
}
</code></pre>

} </pre>

运行结果：

<pre>h1deal!
h2deal!
h3deal!</pre>

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，而不允传给多个对象。
<h3 id="toc_h3_6">六、命令模式</h3>
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，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，不会去关注到底兵是怎么实现的。

<pre>package com.model.behaviour;
public interface Command {
public void exe();
} </pre>

<pre>package com.model.behaviour;
public class MyCommand implments Command {
<pre><code class="highlight-chroma">private Receiver</san> receiver;

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

@Override
public void exe() {
    receiver.action();
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;

public class Invoker {
<pre><code class="highlight-chroma"><span class="highlight
cl">private Command</
pan> command;

<span class="pl
k">public Invoker(<span class="pl
smi">Command command) {
    <span class=
pl-v">this.command <span class="
l-k">= command;
}

<span class="pl
k">public void <span class="pl-en
>action() {
    command<s
an class="pl-k">.exe();
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class Test {
<pre><code class="highlight-chroma"><span class="highlight
cl">public static</span&gt
 void main(&lt
span class="pl-k">String[] args) {
    <span class=
pl-smi">Receiver receiver = <span c
ass="pl-k">new Receiver();
    <span class=
pl-smi">Command cmd = <span cl
ss="pl-k">new MyCommand(recei
er);
    <span class=
pl-smi">Invoker invoker = <span cl
ss="pl-k">new Invoker(cmd);
    invoker.action();
}
</code></pre>
} </pre>

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！
<h3 id="toc_h3_7">七、备忘录模式</h3>
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，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。

<pre>package com.model.behaviour;
public class Original {
<pre><code class="highlight-chroma"><span class="highlight
cl">private String</spa
> value;

<span class="pl
k">public String <span class="pl
en">getValue() {
    <span class=
pl-k">return value;
}

<span class="pl
k">public void <span class="pl-en
>setValue(String <span class="pl
v">value) {
    <span class=
pl-v">this.value <span class="pl-k
>= value;
}

<span class="pl
k">public Original(<span class="p
-smi">String value) {
    <span class=
pl-v">this.value <span class="pl-k
>= value;
}

<span class="pl
k">public Memento <span class
"pl-en">createMemento(){
    <span class=
pl-k">return new <span class="pl-
mi">Memento(value);
}

<span class="pl
k">public void <span class="pl-en
>restoreMemento(Memento <spa
n class="pl-v">memento){
    <span class=
pl-v">this.value <span class="pl-k
>= memento.getValue();
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class Memento {
<pre><code class="highlight-chroma"><span class="highlight

cl">private String</spa
> value;

<span class="pl
k">public Memento(<span class=
pl-smi">String value) {
    <span class=
pl-v">this.value <span class="pl-k
>= value;
}

<span class="pl
k">public String <span class="pl
en">getValue() {
    <span class=
pl-k">return value;
}

<span class="pl
k">public void <span class="pl-en
>setValue(String <span class="pl
v">value) {
    <span class=
pl-v">this.value <span class="pl-k
>= value;
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class Storage {
<pre><code class="highlight-chroma"><span class="highlight
cl">private Memento</
pan> memento;

<span class="pl
k">public Storage(<span class="p
-smi">Memento memento) {
    <span class=
pl-v">this.memento <span class="p
-k">= memento;
}

<span class="pl
k">public Memento <span class
"pl-en">getMemento() {
    <span class=
pl-k">return memento;
}

<span class="pl
k">public void <span class="pl-en
>setMemento(Memento <span c

ass="pl-v">memento) {
    <span class=
pl-v">this.memento <span class="p
-k">= memento;
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class Test {
<pre><code class="highlight-chroma"><span class="highlight
cl">public static</span&gt
 void main(&lt
span class="pl-k">String[] args) {

    <span class=
pl-c">// 创建原始类
    <span class=
pl-smi">Original origi = <span clas
="pl-k">new Original(<span clas
="pl-s">"egg"</sp
n>);

    <span class=
pl-c">// 创建备忘录
    <span class=
pl-smi">Storage storage = <span c
ass="pl-k">new Storage(origi<s
an class="pl-k">.createMemento());

    <span class=
pl-c">// 修改原始类的状态
    <span class=
pl-smi">System.out<span class="pl
k">.println("</span&
t;初始化状态为：" <span class="pl-k
>+ origi.getValue());
    origi<span cl
ss="pl-k">.setValue("&l
;/span>niu");
    <span class=
pl-smi">System.out<span class="pl
k">.println("</span&
t;修改后的状态为：" <span class="pl
k">+ origi.getValue());

    <span class=
pl-c">// 回复原始类的状态
    origi<span cl
ss="pl-k">.restoreMemento(storage.
etMemento());
    <span class=
pl-smi">System.out<span class="pl

k">.println("</span&t;恢复后的状态为：" + origi.getValue());
}
</code></pre>
} </pre>

输出结果：

<pre>初始化状态为：egg
修改后的状态为：niu
恢复后的状态为：egg
如果还不能理解，可以给Original类添加一个属性name，然后其他类进行相应的改试试。</pre>

<h3 id="toc_h3_8">八、状态模式</h3>
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。

<pre>package com.model.behaviour;
public class State {
<pre><code class="highlight-chroma">private String</spa> value;

public String getValue() {
    return value;
}

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

public void method1(){
    System.out.println("</span&t;execute the first opt!");
}

<span class="pl
k">public void <span class="pl-en
>method2(){
    <span class=
pl-smi">System.out<span class="pl
k">.println("</span&
t;execute the second opt!");
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class Context {
<pre><code class="highlight-chroma"><span class="highlight
cl">private State</span
gt; state;

<span class="pl
k">public Context(<span class="p
-smi">State state) {
    <span class=
pl-v">this.state <span class="pl-k"
gt;= state;
}

<span class="pl
k">public State <span class="pl-
n">getState() {
    <span class=
pl-k">return state;
}

<span class="pl
k">public void <span class="pl-en
>setState(State <span class="pl-
">state) {
    <span class=
pl-v">this.state <span class="pl-k"
gt;= state;
}

<span class="pl
k">public void <span class="pl-en
>method() {
    <span class=
pl-smi">System.out<span class="pl
k">.println("</span&
t;状态为："
 state.getValue());
    <span class=
pl-k">if (state.getValue()<span clas
="pl-k">.equals("</s
an>state1")) {

        state.method1();
    } else if (state.getValue().equals("state2")) {
        state.method2();
    }
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class Test {
<pre><code class="highlight-chroma">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();
}
</code></pre>
} </pre>

运行结果：

<pre>状态为：state1
execute the first opt!
状态为：state2
execute the second opt!</pre>

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。
<h3 id="toc_h3_9">九、访问者模式</h3>
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。
访问者模式算是最复杂也是最难以理解的一种模式了。它表示一个作用于某对象结构中的各元素操作。它使你可以在不改变各元素类的前提下定义作用于这些元素的新操作。
涉及角色：
1.Visitor 抽象访问者角色，为该对象结构中具体元素角色声明一个访问操作接口。该操作接口的字和参数标识了发送访问请求给具体访问者的具体元素角色，这样访问者就可以通过该元素角色的特接口直接访问它。
2.ConcreteVisitor.具体访问者角色，实现Visitor声明的接口。
3.Element 定义一个接受访问操作(accept())，它以一个访问者(Visitor)作为参数。
4.ConcreteElement 具体元素，实现了抽象元素(Element)所定义的接受操作接口。
5.ObjectStructure 结构对象角色，这是使用访问者模式必备的角色。它具备以下特性：能枚举的元素；可以提供一个高层接口以允许访问者访问它的元素；如有需要，可以设计成一个复合对象或一个聚集（如一个列表或无序集合）。

<pre>abstract class Element
{
    public abstract void accept</spa>(IVisitor visitor);
    public abstract void doSomethin();
}</pre>

<pre>class ConcreteElement1 extends <spn>Element{
    public void doSomething(){
        System.out.println("</pan>这是元素1");
    }
    public void accept(IVisitor</span visitor){
        visitor.visit(this);
    }
}</pre>

<pre>class ConcreteElement2 extends <spn>Element{
    public void doSomething(){
        System.out.println("</pan>这是元素2");
    }
    public void accept(IVisitor</span

```
 <span>visitor</span>){
      visitor<span>.</span>visit(<span>this</span>);
   }
}</pre>
</div>
<div class="highlight highlight-source-java">
<pre><span>interface</span> <span>IVisitor</span>{
   <span>public</span> <span>void</span> <span>visit</span>(<span>ConcreteElement
</span> <span>el1</span>);
   <span>public</span> <span>void</span> <span>visit</span>(<span>ConcreteElement
</span> <span>el2</span>);
}</pre>
</div>
<div class="highlight highlight-source-java">
<pre><span>class</span> <span>Visitor</span> <span>implements</span> <span>IVisi
or</span>{
   <span>public</span> <span>void</span> <span>visit</span>(<span>ConcreteElement
</span> <span>el1</span>){
      el1<span>.</span>doSomething();
   }
   <span>public</span> <span>void</span> <span>visit</span>(<span>ConcreteElement
</span> <span>el2</span>){
      el2<span>.</span>doSomething();
   }
}</pre>
</div>
<div class="highlight highlight-source-java">
<pre><span>class</span> <span>ObjectStruture</span>{
   <span>public</span> <span>static</span> <span>List&amp;lt;<span>Element</span>&amp;gt;<
span> <span>getList</span>(){
      <span>List&amp;lt;<span>Element</span>&amp;gt;</span>list <span>=</span> <span>new<
span> <span>ArrayList&amp;lt;<span>Element</span>&amp;gt;</span>();
      <span>Random</span> ran <span>=</span> newRandom();
      <span>for</span>(<span>int</span> i <span>=</span> <span>0</span> ; i <span
&amp;lt;</span> <span>10</span> ; i <span>++</span>){
         <span>int</span> a<span>=</span>ran<span>.</span>nextInt(<span>100</span
);
         <span>if</span>(a<span>&amp;gt;</span><span>50</span>){
            list<span>.</span>add(<span>new</span> <span>ConcreteElement1</span>());
         }<span>else</span>{
            list<span>.</span>add(<span>new</span> <span>ConcreteElement2</span>());
         }
      }
      <span>return</span> list;
   }
}</pre>
</div>
<div class="highlight highlight-source-java">
<pre><span>public</span> <span>class</span> <span>Client</span>{
   <span>public</span> <span>static</span> <span>void</span> <span>main</span> (
span>String</span>[]<span>args</span>){
      <span>List&amp;lt;<span>Element</span>&amp;gt;</span> list <span>=</span> <span>Obje
tStruture</span><span>.</span>getList();
      <span>for</span>(<span>Element</span> e<span>:</span>list){
```

```
            e.accept(new Visitor());
        }
    }
}
```

<h3 id="toc_h3_10">十、中介者模式</h3>

中介者模式（Mediator）：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。

举例：在一个公司里面，有很多部门、员工（我们统称他们互相为Colleague“同事”），为了成一定的任务，“同事”之间肯定有许多需要互相配合、交流的过程。如果由各个“同事”频繁地到去与自己有关的“同事”沟通，这样肯定会形成一个多对多的杂乱的联系网络而造成工作效率低下。

此时就需要一位专门的“中介者”给各个“同事”分配任务，以及统一跟进大家的进度并在“同”之间实时地进行交互，保证“同事”之间必须的沟通交流。很明显我们知道此时的“中介者”担任沟通“同事”彼此之间的重要角色了，“中介者”使得每个“同事”都变成一对一的联系方式，减轻每个“同事”的负担，增强工作效率。

同事类族：

```
package com.model.behaviour;

public abstract class AbstractColeague {
protected AbstractMediator mediator;
/**既然有中介者，那么每个具体同事必然要与中介者有联系，
 * 否则就没必要存在于 这个系统当中，这里的构造函数相当
 * 于向该系统中注册一个中介者，以取得联系
 */
public AbstractColleague(AbstractMediator
mediator) {
this.mediator = mediator;
}
// 在抽象同事类中添加用于与中介者取得联系（即注册）的方法
public void setMediator(AbstractMediator mediator) {
    this.mediator = mediator;
}}
```

```
//具体同事A
package com.model.behaviour;
public class ColleagueA extends
```

 AbstractColleague {
<pre><code class="highlight-chroma"><span class="highlight
cl">//每个具体同事都通过父类构造函数与中介者取得联系 </span&gt

<span class="pl
k">public ColleagueA(<span class
"pl-smi">AbstractMediator mediator)

 <span class=
pl-v">super(mediator);
}

<span class="pl
c">//每个具体同事必然有自己分内的事，没必要与外界相关联
<span class="pl
k">public void <span class="pl-en
>self() {
 <span class=
pl-smi">System.out<span class="pl
k">.println("</span&
t;同事A --> 做好自己分内的事情 ..."</span
gt;);
}

<span class="pl
c">//每个具体同事总有需要与外界交互的操作，通过中介者来处理这些逻辑并安排工作 </spa
>
<span class="pl
k">public void <span class="pl-en
>out() {
 <span class=
pl-smi">System.out<span class="pl
k">.println("</span&
t;同事A --> 请求同事B做好分内工作 ..."</sp
n>);
 <span class=
pl-v">super.mediator<span class="
l-k">.execute("</spa
>ColleagueB", <span class="pl-s
>"self"
lt;/span>);
}
</code></pre>
} </pre>

<pre>//具体同事B
package com.model.behaviour;
public class ColleagueB extend
 AbstractColleague {
<pre><code class="highlight-chroma"><span class="highlight
cl">public ColleagueB</
pan>(AbstractMediator <span class="pl-v"&gt
mediator) {

    <span class=
pl-v">super(mediator);
}

<span class="pl
k">public void <span class="pl-en
>self() {
    <span class=
pl-smi">System.out<span class="pl
k">.println("</span&
t;同事B --> 做好自己分内的事情 ..."</span
gt;);
}

<span class="pl
k">public void <span class="pl-en
>out() {
    <span class=
pl-smi">System.out<span class="pl
k">.println("</span&
t;同事B --> 请求同事A做好分内工作 ..."</s
an>);
    <span class=
pl-v">super.mediator<span class="
l-k">.execute("</spa
>ColleagueA", <span class="pl-s
>"self"
lt;/span>);
}
</code></pre>
} </pre>

中介者类族：

<pre>package com.model.behaviour;
public abstract class AbstractM
diator {
<pre><code class="highlight-chroma"><span class="highlight
cl">//中介者肯定需要保持有若干同事的联系方式
<span class="pl
k">protected Hashtable<<span class="pl-
mi">String, AbstractColleague&am
;gt; colleagues = <span class="pl-k"&
t;new Hashtable<St
ing, AbstractColleague></s
an>();

<span class="pl
c">//中介者可以动态地与某个同事建立联系
<span class="pl
k">public void <span class="pl-en
>addColleague(String <span clas
="pl-v">name, AbstractColleague
lt;span class="pl-v">c) {

    <span class=
pl-v">this.colleagues<span class="
l-k">.put(name, c);
}

<span class="pl
c">//中介者也可以动态地撤销与某个同事的联系
<span class="pl
k">public void <span class="pl-en
>deleteColleague(String <span c
ass="pl-v">name) {
    <span class=
pl-v">this.colleagues<span class="
l-k">.remove(name);
}

<span class="pl
c">//中介者必须具备在同事之间处理逻辑、分配任务、促进交流的操作
<span class="pl
k">public abstract <span class="pl-
">void execute(<span class="pl-s
i">String name, <span class="pl-sm
">String method);
</code></pre>
} </pre>

<pre>//具体中介者
package com.model.behaviour;
public class Mediator extends<
span> AbstractMediator{
<pre><code class="highlight-chroma"><span class="highlight
cl">//中介者最重要的功能，来回奔波与各个同事之间
<span class="pl
k">public void <span class="pl-en
>execute(String <span class="pl-
">name, String <span class="pl-v
>method) {

    <span class=
pl-k">if("
elf".</spa
>equals(method)){ //各自做好分内事
        <span clas
="pl-k">if("</span&g
;ColleagueA"
equals(name)) {
            <span cl
ss="pl-smi">ColleagueA colleague =
ColleagueA)super</spa
>.colleagues.</span&gt
get("ColleagueA<span c
ass="pl-pds">");
            colleague

lt;span class="pl-k">.self();
        }<span clas
="pl-k">else {
            <span cl
ss="pl-smi">ColleagueB colleague =
ColleagueB)super</spa
>.colleagues.</span&gt
get("ColleagueB<span c
ass="pl-pds">");
            colleague
lt;span class="pl-k">.self();
        }
    }<span class=
pl-k">else { //与其他同事合作
        <span clas
="pl-k">if("</span&g
;ColleagueA"
equals(name)) {
            <span cl
ss="pl-smi">ColleagueA colleague =
ColleagueA)super</spa
>.colleagues.</span&gt
get("ColleagueA<span c
ass="pl-pds">");
            colleague
lt;span class="pl-k">.out();
        }<span clas
="pl-k">else {
            <span cl
ss="pl-smi">ColleagueB colleague =
ColleagueB)super</spa
>.colleagues.</span&gt
get("ColleagueB<span c
ass="pl-pds">");
            colleague
lt;span class="pl-k">.out();
        }
    }
}
</code></pre>
} </pre>

测试类：

<pre>//测试类
package com.model.behaviour;
public class Client {
public static void main(<sp
n>String[] args) {
<pre><code class="highlight-chroma"><span class="highlight
cl">   //创建一个中介者
    <span class=
pl-smi">AbstractMediator mediator =
new Mediator</span&gt

();

    <span class=
pl-c">//创建两个同事
    <span class=
pl-smi">ColleagueA colleagueA = &lt
span class="pl-k">new ColleagueA
mediator);
    <span class=
pl-smi">ColleagueB colleagueB = <
pan class="pl-k">new ColleagueB(
ediator);

    <span class=
pl-c">//中介者分别与每个同事建立联系
    mediator<sp
n class="pl-k">.addColleague(<span class="pl-p
s">"ColleagueA", coll
agueA);
    mediator<sp
n class="pl-k">.addColleague(<span class="pl-p
s">"ColleagueB", coll
agueB);

    <span class=
pl-c">//同事们开始工作
    colleagueA<
pan class="pl-k">.self();
    colleagueA<
pan class="pl-k">.out();
    <span class=
pl-smi">System.out<span class="pl
k">.println("</span&
t;==========================合作愉快，任务完成！\n</sp
n>");

    colleagueB<s
an class="pl-k">.self();
    colleagueB<s
an class="pl-k">.out();
    <span class=
pl-smi">System.out<span class="pl
k">.println("</span&
t;==========================合作愉快，任务完成！"</spa
>);
}
</code></pre>
} </pre>

运行结果：

<pre>同事A --> 做好自己分内的事情 <sp
n>...
同事A --> 请求同事B做

分内工作 ...
同事B --> 做好自己分内的事情 ...<
span>
===========合作愉快，任务完成！
同事B --> 做好自己分内的事情 <span
...
同事B --> 请求同事A做
分内工作 ...
同事A --> 做好自己分内的事情 ...<
span>
===========合作愉快，任务完成！ </p
</pre>

<h3 id="toc_h3_11"><a class="anchor" href="https://ld246.com/forward?goto=https%3A%2
%2Fgithub.com%2FHassanChiang%2FBlog%2Fblob%2Fmaster%2Fsource%2F_posts%2F2016
3%2Fjava-pattern-3.md%23%E5%8D%81%E4%B8%80%E8%A7%A3%E9%87%8A%E5%99%A
%E6%A8%A1%E5%BC%8F" target="_blank" rel="nofollow ugc">十一、解释器模式</h3>
解释器模式：给定一种语言，定义他的文法的一种表示，并定义一个解释器，该解释器使用该表
来解释语言中句子。
解释器模式是一个比较少用的模式。

<pre>package com.model.behaviour;
public class Context {
<pre><code class="highlight-chroma"><span class="highlight
cl">private int
um1;
<span class="pl
k">private int num2;

<span class="pl
k">public Context(<span class="p
-k">int num1, <span class="pl-k"&
t;int num2) {
    <span class=
pl-v">this.num1 <span class="pl-k
>= num1;
    <span class=
pl-v">this.num2 <span class="pl-k
>= num2;
}

<span class="pl
k">public int <span class="pl-en"&
t;getNum1() {
    <span class=
pl-k">return num1;
}

<span class="pl
k">public void <span class="pl-en
>setNum1(int <span class="pl-v"&
t;num1) {
    <span class=
pl-v">this.num1 <span class="pl-k

>= num1;
}

<span class="pl
k">public int <span class="pl-en"&
t;getNum2() {
    <span class=
pl-k">return num2;
}

<span class="pl
k">public void <span class="pl-en
>setNum2(int <span class="pl-v"&
t;num2) {
    <span class=
pl-v">this.num2 <span class="pl-k
>= num2;
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public interface Expression {
public int interpret(Context
context);
} </pre>

<pre>package com.model.behaviour;
public class Minus implements
/span> Expression {
<pre><code class="highlight-chroma"><span class="highlight
cl">@Override
<span class="pl
k">public int <span class="pl-en"&
t;interpret(Context <span class="pl-
">context) {
    <span class=
pl-k">return context.getNum1()<sp
n class="pl-k">-context.getNum2();
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class Plus implements</
pan> Expression {
<pre><code class="highlight-chroma"><span class="highlight
cl">@Override
<span class="pl
k">public int <span class="pl-en"&
t;interpret(Context <span class="pl-

">context) {
    <span class=
pl-k">return context.getNum1()<sp
n class="pl-k">+context.getNum2();
}
</code></pre>
} </pre>

<pre>package com.model.behaviour;
public class Test {
<pre><code class="highlight-chroma"><span class="highlight
cl">public static</span&gt
 void main(&lt
span class="pl-k">String[] args) {
    <span class=
pl-c">// 计算9+2-8的值
    <span class=
pl-k">int result = <span class="pl-k
>new Minus()<span class="pl-k"
gt;.interpret((new <span class="pl-smi
>Context(new <span class="pl-smi
>Plus()
            .interpret(
lt;span class="pl-k">new Context(&
t;span class="pl-c1">9, 2)), 8)));
    <span class=
pl-smi">System.out<span class="pl
k">.println(result);
}
</code></pre>
} </pre>

原文：<a href="https://ld246.com/forward?goto=http%3A%2F%2Fblog.csdn.net%2Fu01
142781%2Farticle%2Fdetails%2F50825301" target="_blank" rel="nofollow ugc">http://blog.c
dn.net/u013142781/article/details/50825301