

备忘录模式

整理自：《java与模式》之备忘录模式

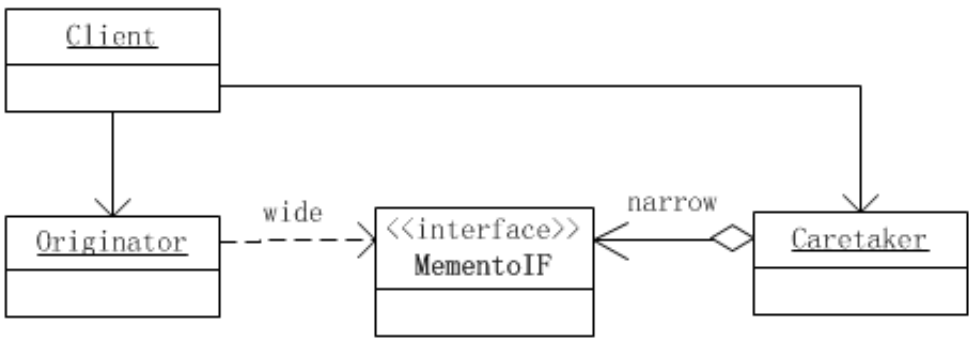
在阎宏博士的《JAVA与模式》一书中开头是这样描述备忘录（Memento）模式的：

备忘录模式又叫做快照模式(Snapshot Pattern)或Token模式，是对象的行为模式。

备忘录对象是一个用来存储另外一个对象内部状态的快照的对象。备忘录模式的用意是在不破坏封装的条件下，将一个对象的状态捕捉(Capture)住，并外部化，存储起来，从而可以在将来合适的时候把这个对象还原到存储起来的状态。备忘录模式常常与命令模式和迭代子模式一同使用。

备忘录模式的结构

备忘录模式的结构图如下所示



备忘录模式所涉及的角色有三个：**备忘录(Memento)角色**、**发起人(Originator)角色**、**负责人(Caretaker)角色**。

备忘录(Memento)角色

备忘录角色有如下责任：

- (1) 将发起人（Originator）对象的内部状态存储起来。备忘录可以根据发起人对象的判断来决定存储多少发起人（Originator）对象的内部状态。
- (2) 备忘录可以保护其内容不被发起人（Originator）对象之外的任何对象所读取。

备忘录有两个等效的接口：

- **窄接口：**负责人（Caretaker）对象（和其他除发起人对象之外的任何对象）看到的是备忘录的窄接口(narrow interface)，这个窄接口只允许它把备忘录对象传给其他的对象。
- **宽接口：**与负责人对象看到的窄接口相反的是，发起人对象可以看到一个宽接口(wide interface)，这个宽接口允许它读取所有的数据，以便根据这些数据恢复这个发起人对象的内部状态。

发起人（Originator）角色

发起人角色有如下责任：

- （1）创建一个含有当前的内部状态的备忘录对象。
- （2）使用备忘录对象存储其内部状态。

负责人（Caretaker）角色

负责人角色有如下责任：

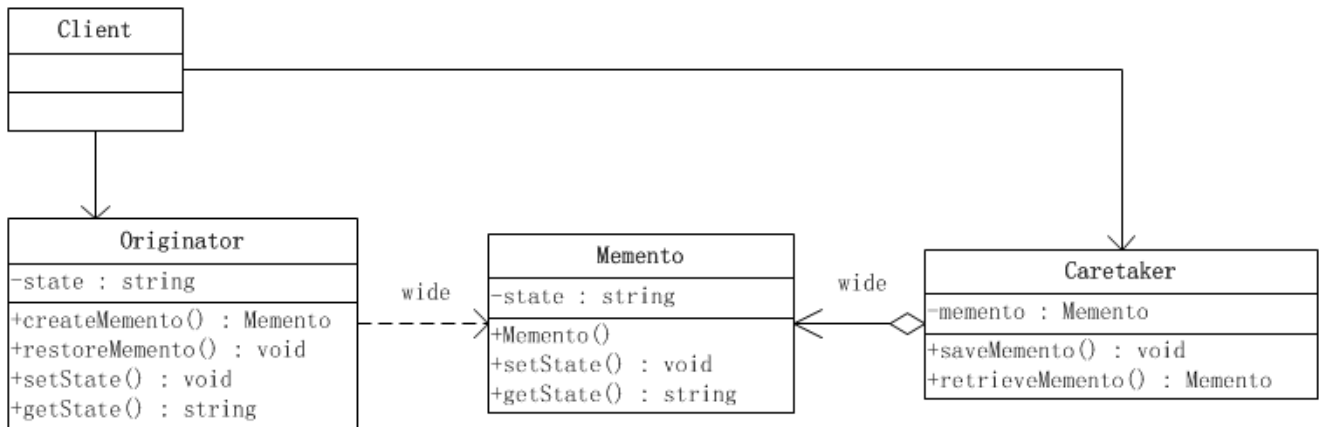
- （1）负责保存备忘录对象。
- （2）不检查备忘录对象的内容。

“白箱”备忘录模式的实现

备忘录角色对任何对象都提供一个接口，即宽接口，备忘录角色的内部所存储的状态就对所有对象公开。因此这个实现又叫做“白箱实现”。

“白箱”实现将发起人角色的状态存储在一个大家都看得到的地方，因此是破坏封装性的。但是通过程序员自律，同样可以在一定程度上实现模式的大部分用意。因此白箱实现仍然是有意义的。

下面给出一个示意性的“白箱实现”。



源代码

发起人角色类，发起人角色利用一个新创建的备忘录对象将自己的内部状态存储起来。

```
public class Originator {

    private String state;
    /**
     * 工厂方法，返回一个新的备忘录对象
     */
    public Memento createMemento(){
        return new Memento(state);
    }
    /**
     * 将发起人恢复到备忘录对象所记载的状态
     */
    public void restoreMemento(Memento memento){
        this.state = memento.getState();
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
        System.out.println("当前状态: " + this.state);
    }
}
```

```
}
```

备忘录角色类，备忘录对象将发起人对象传入的状态存储起来。

```
public class Memento {  
  
    private String state;  
  
    public Memento(String state){  
        this.state = state;  
    }  
  
    public String getState() {  
        return state;  
    }  
  
    public void setState(String state) {  
        this.state = state;  
    }  
  
}
```

负责人角色类，负责人角色负责保存备忘录对象，但是从不修改（甚至不查看）备忘录对象的内容。

```
public class Caretaker {  
  
    private Memento memento;  
    /**  
     * 备忘录的取值方法  
     */  
    public Memento retrieveMemento(){  
        return this.memento;  
    }  
    /**  
     * 备忘录的赋值方法  
     */  
    public void saveMemento(Memento memento){  
        this.memento = memento;  
    }  
}
```

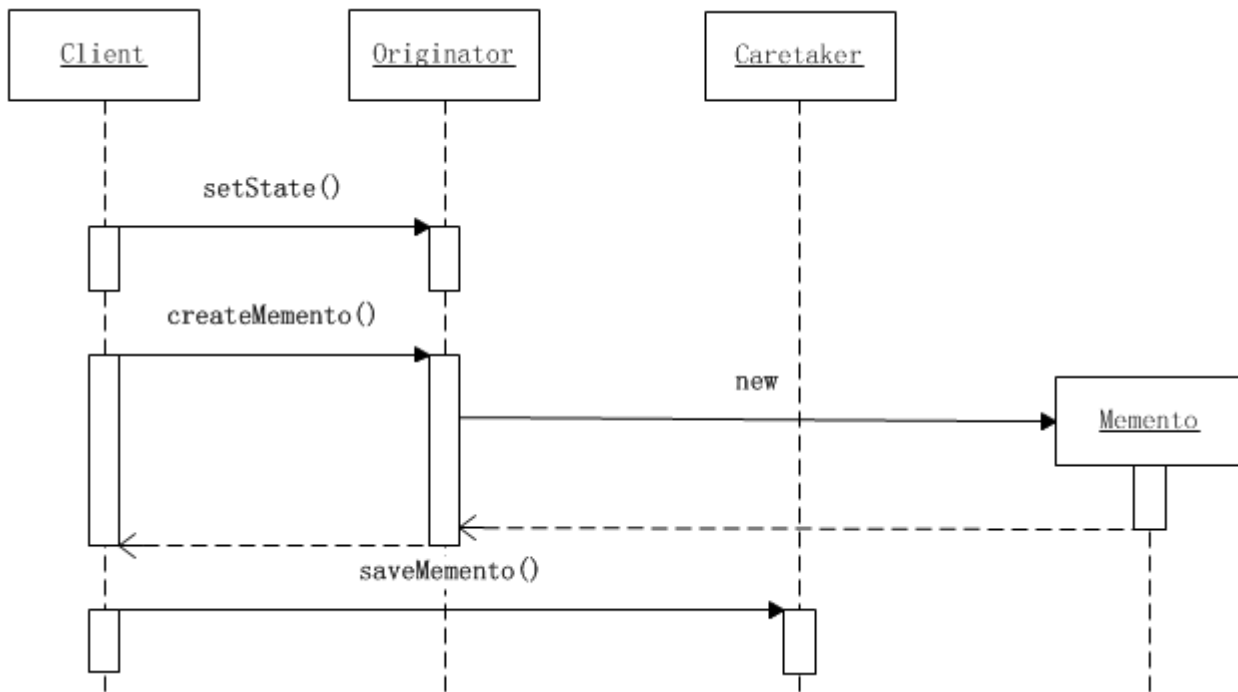
```
}
```

客户端角色类

```
public class Client {  
  
    public static void main(String[] args) {  
  
        Originator o = new Originator();  
        Caretaker c = new Caretaker();  
        //改变负责人对象的状态  
        o.setState("On");  
        //创建备忘录对象，并将发起人对象的状态储存起来  
        c.saveMemento(o.createMemento());  
        //修改发起人的状态  
        o.setState("Off");  
        //恢复发起人对象的状态  
        o.restoreMemento(c.retrieveMemento());  
  
        System.out.println(o.getState());  
    }  
  
}
```

在上面的这个示意性的客户端角色里面，首先将发起人对象的状态设置成“On”，并创建一个备忘录对象将这个状态存储起来；然后将发起人对象的状态改成“Off”；最后又将发起人对象恢复到备忘录对象所存储起来的状态，即“On”状态。

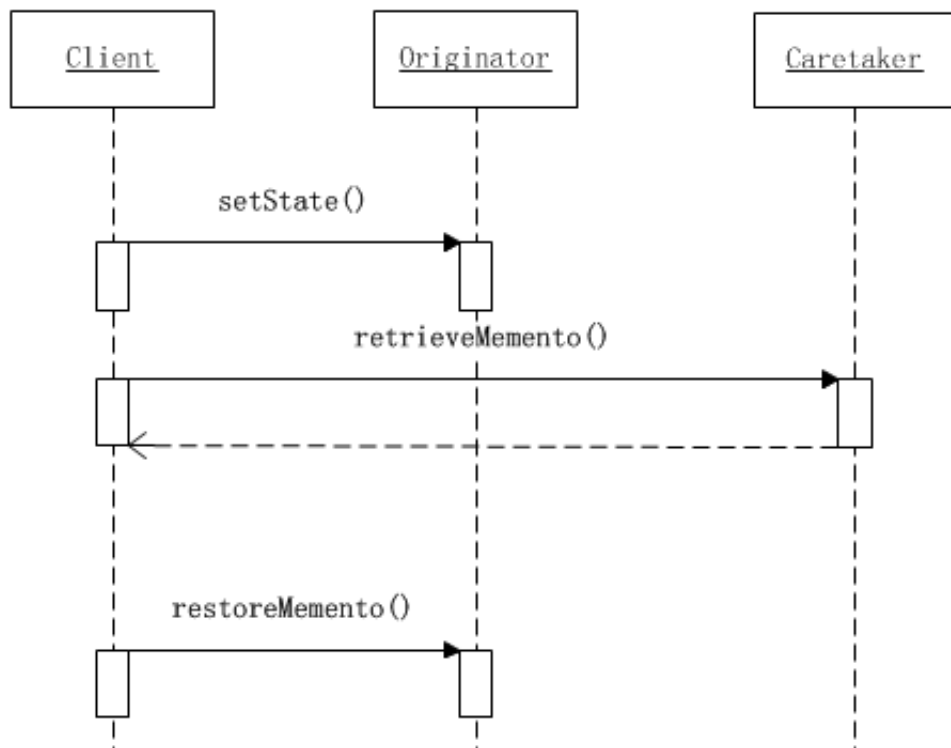
系统的时序图更能够反映出系统各个角色被调用的时间顺序。如下图是将发起人对象的状态存储到白箱备忘录对象中去的时序图。



可以看出系统运行的时序是这样的：

- (1) 将发起人对象的状态设置成“On”。
- (2) 调用发起人角色的`createMemento()`方法，创建一个备忘录对象将这个状态存储起来。
- (3) 将备忘录对象存储到负责人对象中去。

将发起人对象恢复到备忘录对象所记录的状态的时序图如下所示：



可以看出，将发起人对象恢复到备忘录对象所记录的状态时，系统的运行时序是这样的：

- (1) 将发起人状态设置成“Off”。
- (2) 将备忘录对象从负责人对象中取出。
- (3) 将发起人对象恢复到备忘录对象所存储起来的状态，即“On”状态。

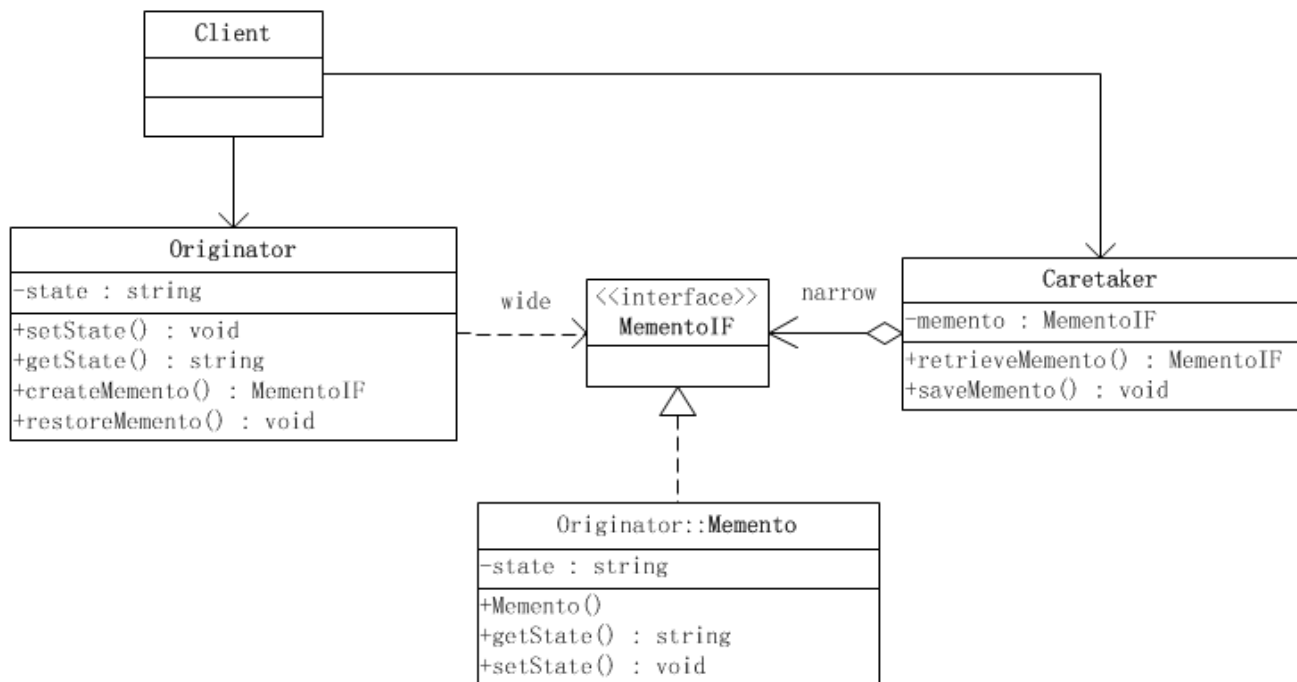
“黑箱”备忘录模式的实现

备忘录角色对发起人（Originator）角色对象提供一个宽接口，而为其他对象提供一个窄接口。这样的实现叫做“黑箱实现”。

在JAVA语言中，实现双重接口的办法就是将**备忘录角色类**设计成**发起人角色类**的内部成员类。

将Memento设成Originator类的内部类，从而将Memento对象封装在Originator里面；在外部提供一个标识接口MementoIF给Caretaker以及其他对象。这样，Originator类看到的是Memento的所有接口，而Caretaker以及其他对象看到的仅仅是标识接口MementoIF所暴露出来的接口。

使用内部类实现备忘录模式的类图如下所示。



源代码

发起人角色类Originator中定义了一个内部的Memento类。由于此Memento类的全部接口都是私有的，因此只有它自己和发起人类可以调用。

```

package memento.sample2;

/**
 * @author chen_dz
 * @date : 2012-6-2 上午10:11:08
 */
public class Originator {

    private String state;

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
        System.out.println("赋值状态: " + state);
    }
}

```



```

    * 工厂方法，返还一个新的备忘录对象
    */
    public MementoIF createMemento(){
        return new Memento(state);
    }
    /**
    * 发起人恢复到备忘录对象记录的状态
    */
    public void restoreMemento(MementoIF memento){
        this.setState(((Memento)memento).getState());
    }

    private class Memento implements MementoIF{

        private String state;
        /**
        * 构造方法
        */
        private Memento(String state){
            this.state = state;
        }

        private String getState() {
            return state;
        }
        private void setState(String state) {
            this.state = state;
        }
    }
}

```

窄接口MementoIF，这是一个标识接口，因此它没有定义出任何的方法。

```

public interface MementoIF {

}

```

负责人角色类Caretaker能够得到的备忘录对象是以MementoIF为接口的，由于这个接口仅仅是一个标识接口，因此负责人角色不可能改变这个备忘录对象的内容。

```

public class Caretaker {

    private MementoIF memento;
    /**
     * 备忘录取值方法
     */
    public MementoIF retrieveMemento(){
        return memento;
    }
    /**
     * 备忘录赋值方法
     */
    public void saveMemento(MementoIF memento){
        this.memento = memento;
    }
}

```

客户端角色类

```

public class Client {

    public static void main(String[] args) {
        Originator o = new Originator();
        Caretaker c = new Caretaker();
        //改变负责人对象的状态
        o.setState("On");
        //创建备忘录对象，并将发起人对象的状态存储起来
        c.saveMemento(o.createMemento());
        //修改发起人对象的状态
        o.setState("Off");
        //恢复发起人对象的状态
        o.restoreMemento(c.retrieveMemento());
    }
}

```

客户端首先

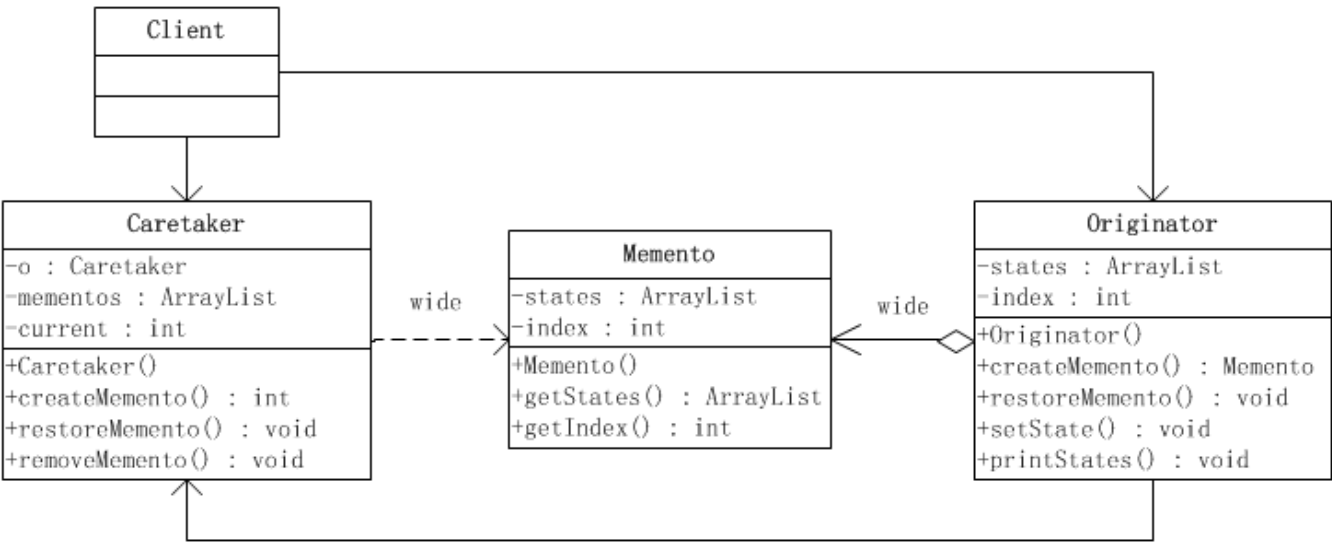
(1) 将发起人对象的状态设置为“On”。

- (2) 调用createMemento()方法，创建一个备忘录对象将这个状态存储起来（此时createMemento()方法还回的明显类型是MementoIF接口，真实类型为Originator内部的Memento对象）。
- (3) 将备忘录对象存储到负责人对象中去。由于负责人对象拿到的仅是MementoIF接口，因此无法读出备忘录对象内部的状态。
- (4) 将发起人对象的状态设置为“Off”。
- (5) 调用负责人对象的retrieveMemento()方法将备忘录对象取出。注意此时仅能得到MementoIF接口，因此无法读出此对象的内部状态。
- (6) 调用发起人对象的restoreMemento()方法将发起人对象的状态恢复成备忘录对象所存储的起来的状态，即“On”状态。由于发起人对象的内部类Memento实现了MementoIF接口，这个内部类是传入的备忘录对象的真实类型，因此发起人对象可以利用内部类Memento的私有接口读出此对象的内部状态。

多重检查点

前面所给出的白箱和黑箱的示意性实现都是只存储一个状态的简单实现，也可以叫做只有一个检查点。常见的系统往往需要存储不止一个状态，而是需要存储多个状态，或者叫做有多个检查点。

备忘录模式可以将发起人对象的状态存储到备忘录对象里面，备忘录模式可以将发起人对象恢复到备忘录对象所存储的某一个检查点上。下面给出一个示意性的、有多重检查点的备忘录模式的实现。



源代码

发起人角色源代码

```
public class Originator {

    private List<String> states;
    //检查点指数
    private int index;
    /**
     * 构造函数
     */
    public Originator(){
        states = new ArrayList<String>();
        index = 0;
    }
    /**
     * 工厂方法，返还一个新的备忘录对象
     */
    public Memento createMemento(){
        return new Memento(states , index);
    }
    /**
     * 将发起人恢复到备忘录对象记录的状态上
     */
    public void restoreMemento(Memento memento){
        states = memento.getStates();
        index = memento.getIndex();
    }
    /**
     * 状态的赋值方法
     */
    public void setState(String state){
        states.add(state);
        index++;
    }
    /**
     * 辅助方法，打印所有状态
     */
    public void printStates(){
```

```

        for(String state : states){
            System.out.println(state);
        }
    }
}

```

备忘录角色类，这个实现可以存储任意多的状态，外界可以使用检查点指数index来取出检查点上的状态。

```

public class Memento {

    private List<String> states;
    private int index;
    /**
     * 构造函数
     */
    public Memento(List<String> states , int index){
        this.states = new ArrayList<String>(states);
        this.index = index;
    }
    public List<String> getStates() {
        return states;
    }
    public int getIndex() {
        return index;
    }
}

```

负责人角色类

```

public class Caretaker {

    private Originator o;
    private List<Memento> mementos = new ArrayList<Memento>();
    private int current;
    /**
     * 构造函数
     */
}

```

```

public Caretaker(Originator o){
    this.o = o;
    current = 0;
}
/**
 * 创建一个新的检查点
 */
public int createMemento(){
    Memento memento = o.createMemento();
    mementos.add(memento);
    return current++;
}
/**
 * 将发起人恢复到某个检查点
 */
public void restoreMemento(int index){
    Memento memento = mementos.get(index);
    o.restoreMemento(memento);
}
/**
 * 将某个检查点删除
 */
public void removeMemento(int index){
    mementos.remove(index);
}
}

```

客户端角色源代码

```

public class Client {

    public static void main(String[] args) {

        Originator o = new Originator();
        Caretaker c = new Caretaker(o);
        //改变状态
        o.setState("state 0");
        //建立一个检查点
        c.createMemento();
        //改变状态
    }
}

```

```

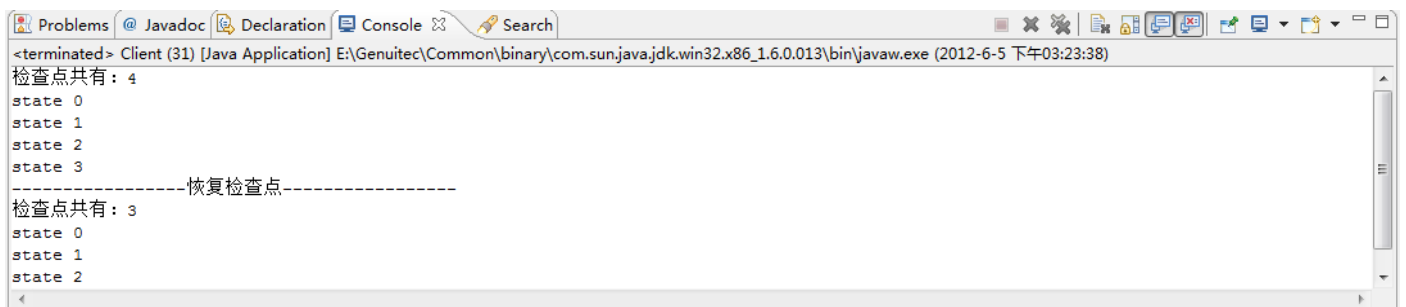
        o.setState("state 1");
        //建立一个检查点
        c.createMemento();
        //改变状态
        o.setState("state 2");
        //建立一个检查点
        c.createMemento();
        //改变状态
        o.setState("state 3");
        //建立一个检查点
        c.createMemento();
        //打印出所有检查点
        o.printStates();
        System.out.println("-----恢复检查点-----");
    ;

    //恢复到第二个检查点
    c.restoreMemento(2);
    //打印出所有检查点
    o.printStates();
}

}

```

运行结果如下：



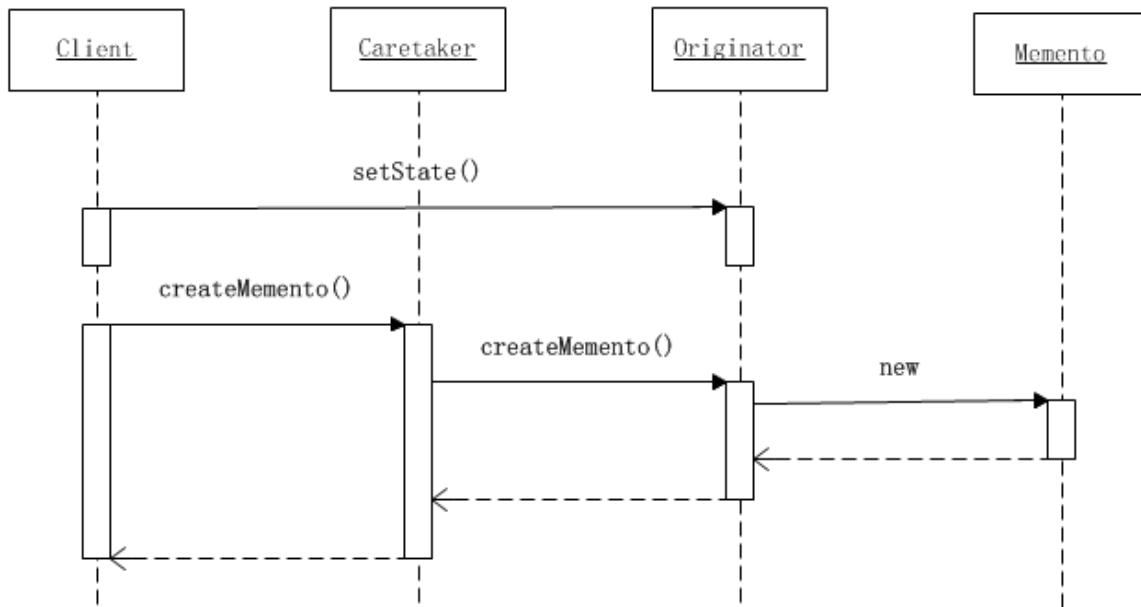
```

<terminated> Client (31) [Java Application] E:\Genuitec\Common\binary\com.sun.java.jdk.win32.x86_1.6.0.013\bin\javaw.exe (2012-6-5 下午03:23:38)
检查点共有：4
state 0
state 1
state 2
state 3
-----恢复检查点-----
检查点共有：3
state 0
state 1
state 2

```

可以看出，客户端角色通过不断改变发起人角色的状态，并将之存储在备忘录里面。通过指明检查点指数可以将发起人角色恢复到相应的检查点所对应的状态上。

将发起人的状态存储到备忘录对象中的活动序列图如下：

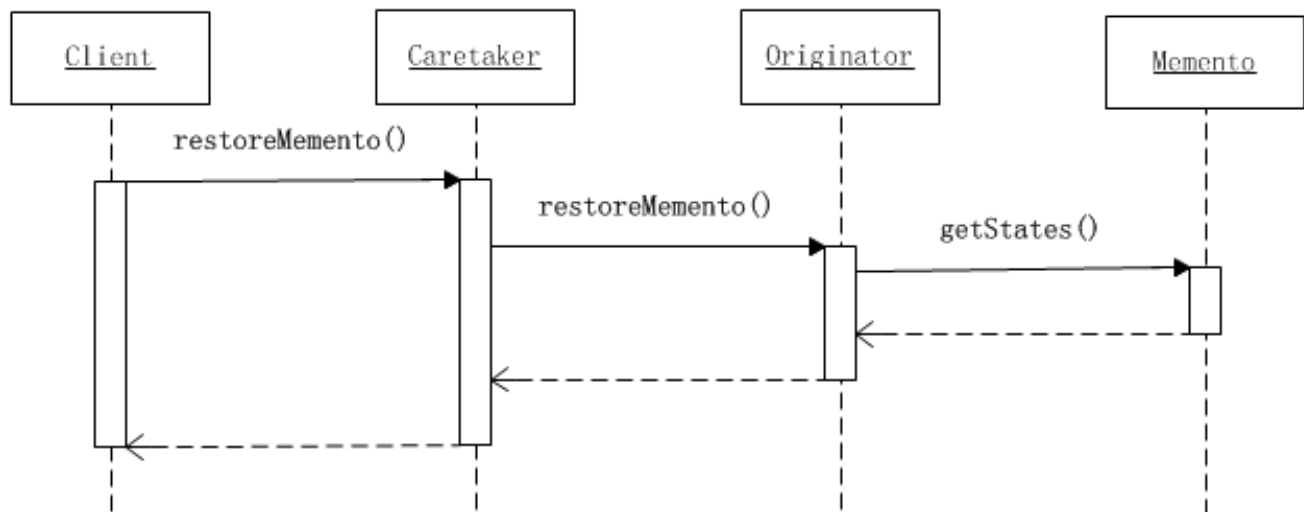


系统运行的时序是这样的：

（1）将发起人对象的状态设置成某个有效状态；

（2）调用负责人角色的createMemento()方法，负责人角色会负责调用发起人角色和备忘录角色，将发起人对象的状态存储起来。

将发起人对象恢复到某一个备忘录对象的检查点的活动序列图如下：



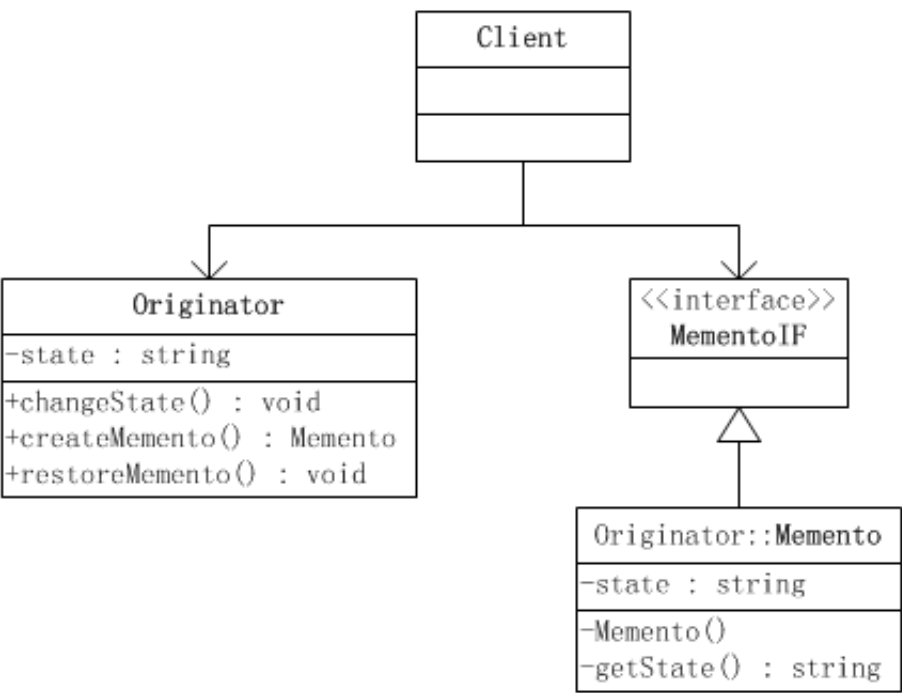
由于负责人角色的功能被增强了，因此将发起人对象恢复到备忘录对象所记录的状态时，系统运行的时序被简化了：

（1）调用负责人角色的restoreMemento()方法，将发起人恢复到某个检查点。

“自述历史”模式

所谓“自述历史”模式(History-On-Self Pattern)实际上就是备忘录模式的一个变种。在备忘录模式中，发起人(Originator)角色、负责人(Caretaker)角色和备忘录(Memento)角色都是独立的角色。虽然在实现上备忘录类可以成为发起人类的内部成员类，但是备忘录类仍然保持作为一个角色的独立意义。在“自述历史”模式里面，发起人角色自己兼任负责人角色。

“自述历史”模式的类图如下所示：



备忘录角色有如下责任：

- （1）将发起人（Originator）对象的内部状态存储起来。
- （2）备忘录可以保护其内容不被发起人（Originator）对象之外的任何对象所读取。

发起人角色有如下责任：

- （1）创建一个含有它当前的内部状态的备忘录对象。
- （2）使用备忘录对象存储其内部状态。

客户端角色有负责保存备忘录对象的责任。

源代码

窄接口MementoIF，这是一个标识接口，因此它没有定义出任何的方法。

```
public interface MementoIF {  
  
}
```

发起人角色同时还兼任负责人角色，也就是说它自己负责保持自己的备忘录对象。

```
public class Originator {  
  
    public String state;  
    /**  
     * 改变状态  
     */  
    public void changeState(String state){  
        this.state = state;  
        System.out.println("状态改变为: " + state);  
    }  
    /**  
     * 工厂方法，返还一个新的备忘录对象  
     */  
    public Memento createMemento(){  
        return new Memento(this);  
    }  
    /**  
     * 将发起人恢复到备忘录对象所记录的状态上  
     */  
    public void restoreMemento(MementoIF memento){  
        Memento m = (Memento)memento;  
        changeState(m.state);  
    }  
  
    private class Memento implements MementoIF{  
  
        private String state;  
        /**  
         * 构造方法  
         */  
        private Memento(Originator o){  
            this.state = o.state;  
        }  
    }  
}
```

```

    }
    private String getState() {
        return state;
    }
}
}

```

客户端角色类

```

public class Client {

    public static void main(String[] args) {
        Originator o = new Originator();
        //修改状态
        o.changeState("state 0");
        //创建备忘录
        MementoIF memento = o.createMemento();
        //修改状态
        o.changeState("state 1");
        //按照备忘录恢复对象的状态
        o.restoreMemento(memento);
    }
}

```

由于“自述历史”作为一个备忘录模式的特殊实现形式非常简单易懂，它可能是备忘录模式最为流行的实现形式。