课程目标

1、掌握备忘录模式的应用场景。

2、掌握备忘录模式在落地实战中的压栈管理。

内容定位

1、如果参与富文本编辑器开发的人群，可以重点关注备忘录模式。

备忘录模式

备忘录模式(Memento Pattern)又称为快照模式(Snapshot Pattern)或令牌模式(Token Pattern ），是指在不破坏封装的前提下，捕获一个对象的内部状态，并在对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态，属于行为型模式。

```
原文：Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
```

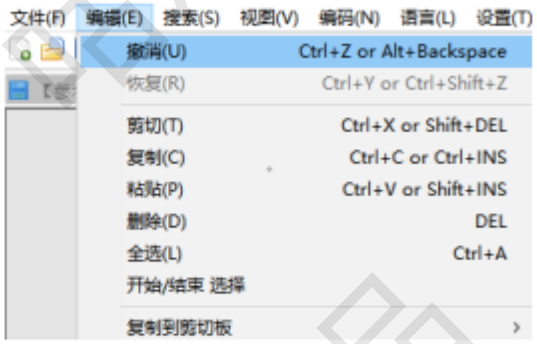
在软件系统中，备忘录模式可以为我们提供一种“后悔药”的机制，它通过存储系统各个历史状态的快照，使得我们可以在任一时刻将系统回滚到某一个历史状态。

备忘录模式本质是从发起人实体类（Originator）隔离存储功能，降低实体类的职责。同时由于存储信息（Memento）独立，且存储信息的实体交由管理类（Caretaker）管理，则可以通过为管理类扩展额外的功能对存储信息进行扩展操作（比如增加历史快照功能...）。

备忘录模式的应用场景

对于我们程序员来说，可能天天都在使用备忘录模式，比如我们每天使用的 Git、SVN 都可以提供一种代码版本撤回的功能。还有一个比较贴切的现实场景应该是游戏的存档功能，通过

将游戏当前进度存储到本地文件系统或数据库中，使得下次继续游戏时，玩家可以从之前的位置继续进行。



代码版本管理中的撤销与恢复

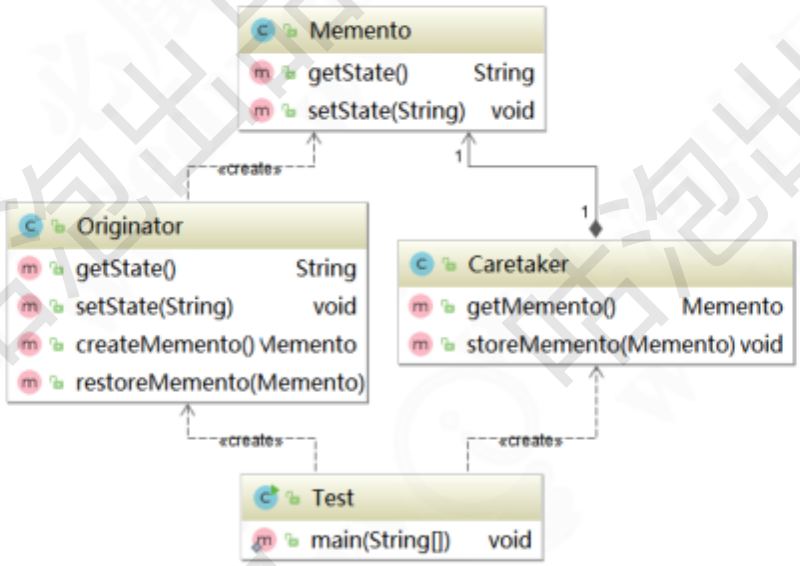
游戏存档

备忘录模式适用于以下应用场景：

1、需要保存历史快照的场景；

2、希望在对象之外保存状态，且除了自己其他类对象无法访问状态保存具体内容。

首先来看下备忘录模式的通用 UML 类图：



从 UML 类图中，我们可以看到，备忘录模式主要包含三种角色：

发起人角色（Originator）：负责创建一个备忘录，记录自身需要保存的状态；具备状态回滚功能；

备忘录角色（Memento）：用于存储 Originator 的内部状态，且可以防止 Originator 以外的对象进行访问；

备忘录管理员角色（Caretaker）：负责存储，提供管理备忘录（Memento），无法对备忘录内容进行操作和访问。

利用压栈管理落地备忘录模式

我们肯定都用过网页中的富文本编辑器，编辑器中的通常会附带草稿箱、撤销等这样的操作。下面我们用一段带代码来实现一个这样的功能。假设，我们在 GPer 社区中发布一篇文章，文章编辑的过程需要花很长时间，中间也会不停地撤销、修改。甚至可能要花好几天才能写出一篇精品文章，因此可能会将已经编辑好的内容实时保存到草稿箱。

首先创建发起人角色编辑器 Editor 类：

```
public class Editor {

    private String title;
    private String content;
    private String imgs;

    public Editor(String title, String content, String imgs) {
        this.title = title;
        this.content = content;
        this.imgs = imgs;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }

    public String getImgs() {
        return imgs;
    }

    public void setImgs(String imgs) {
        this.imgs = imgs;
    }
```

```
    public ArticleMemento saveToMemento() {
        ArticleMemento articleMemento = new ArticleMemento(this.title,this.content,this.imgs);
        return articleMemento;
    }

    public void undoFromMemento(ArticleMemento articleMemento) {

        this.title = articleMemento.getTitle();
        this.content = articleMemento.getContent();
        this.imgs = articleMemento.getImgs();
    }

    @Override
    public String toString() {
        return "Editor{" +
                "title='" + title + '\'' +
                ", content='" + content + '\'' +
                ", imgs='" + imgs + '\'' +
                '}';
    }
}
```

然后创建备忘录角色 ArticleMemento 类：

```
public class ArticleMemento {
    private String title;
    private String content;
    private String imgs;

    public ArticleMemento(String title, String content, String imgs) {
        this.title = title;
        this.content = content;
        this.imgs = imgs;
    }

    public String getTitle() {
        return title;
    }

    public String getContent() {
        return content;
    }

    public String getImgs() {
        return imgs;
    }

    @Override
    public String toString() {
        return "ArticleMemento{" +
                "title='" + title + '\'' +
                ", content='" + content + '\'' +
                ", imgs='" + imgs + '\'' +
                '}';
    }
}
```

最后创建备忘录管理角色草稿箱 DraftsBox 类：

```
public class DraftsBox {

    private final Stack<ArticleMemento> STACK = new Stack<ArticleMemento>();
```

```
    public ArticleMemento getMemento() {
        ArticleMemento articleMemento= STACK.pop();
        return articleMemento;
    }

    public void addMemento(ArticleMemento articleMemento) {
        STACK.push(articleMemento);
    }

}
```

草稿箱中定义的 Stack 类是 Vector 的一个子类，它实现了一个标准的后进先出的栈。主要定义了以下方法：

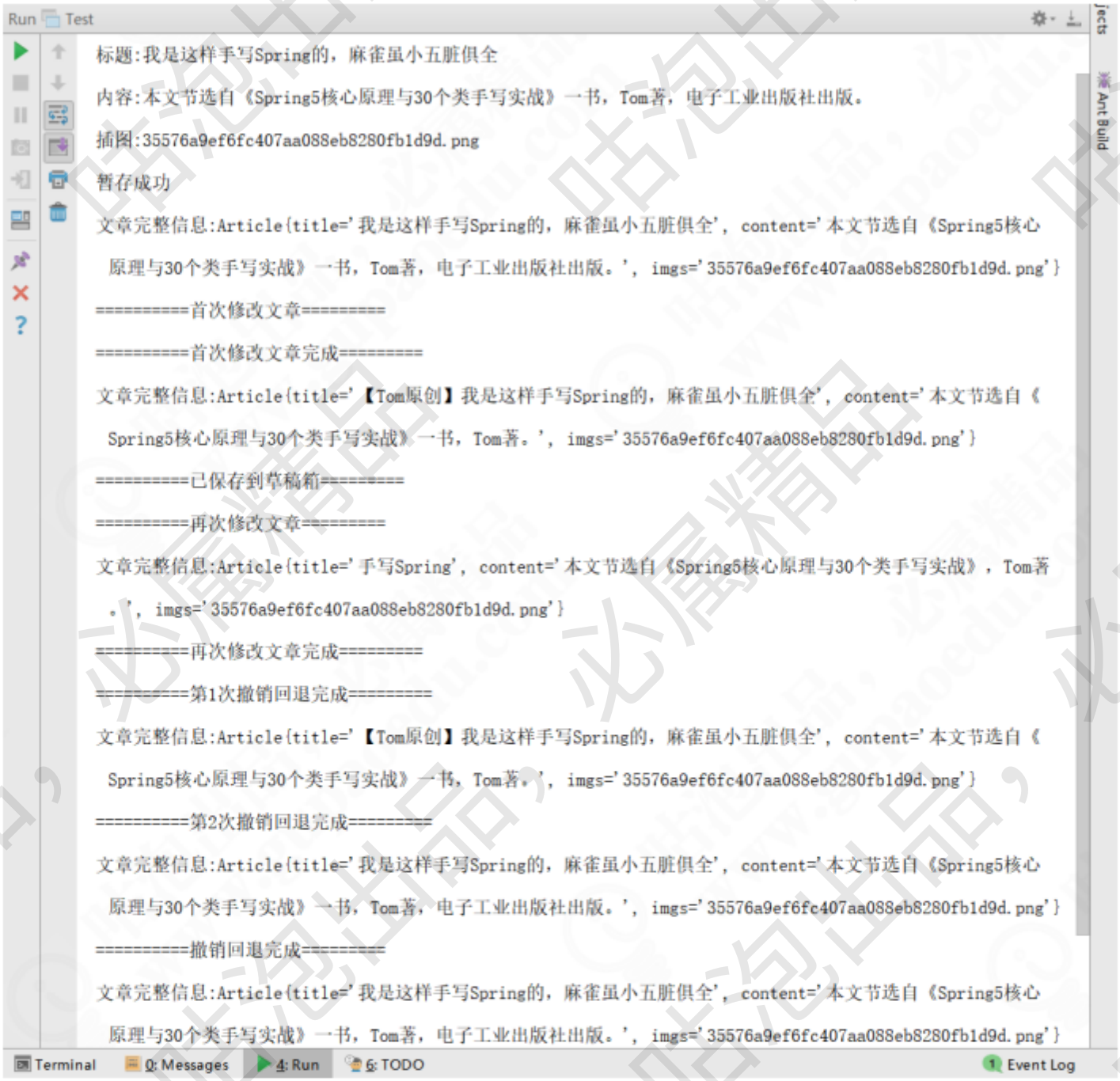
方法定义	方法描述
boolean empty()	测试堆栈是否为空。
Object peek( )	查看堆栈顶部的对象，但不从堆栈中移除它。
Object pop( )	移除堆栈顶部的对象，并作为此函数的值返回该对象。
Object push(Object element)	把对象压入堆栈顶部。
int search(Object element)	返回对象在堆栈中的位置，以 1 为基数。

最后，编写客户端测试代码：

```
public static void main(String[] args) {
    DraftsBox draftsBox = new DraftsBox();

    Editor editor = new Editor("我是这样手写 Spring 的，麻雀虽小五脏俱全",
            "本文节选自《Spring5 核心原理与 30 个类手写实战》一书，Tom 著，电子工业出版社出版。",
            "35576a9ef6fc407aa088eb8280fb1d9d.png");

    ArticleMemento articleMemento = editor.saveToMemento();
    draftsBox.addMemento(articleMemento);

    System.out.println("标题：" + editor.getTitle() + "\n" +
                       "内容：" + editor.getContent() + "\n" +
                       "插图：" + editor.getImgs() + "\n 暂存成功");

    System.out.println("完整的信息" + editor);


    System.out.println("==========首次修改文章==========");
    editor.setTitle("【Tom 原创】我是这样手写 Spring 的，麻雀虽小五脏俱全");
    editor.setContent("本文节选自《Spring5 核心原理与 30 个类手写实战》一书，Tom 著");

    System.out.println("==========首次修改文章完成==========");

    System.out.println("完整的信息" + editor);

    articleMemento = editor.saveToMemento();
```

```
        draftsBox.addMemento(articleMemento);

        System.out.println("==========保存到草稿箱==========");


        System.out.println("==========第 2 次修改文章==========");
        editor.setTitle("手写 Spring");
        editor.setContent("本文节选自《Spring5 核心原理与 30 个类手写实战》一书，Tom 著");
        System.out.println("完整的信息" + editor);
        System.out.println("==========第 2 次修改文章完成==========");

        System.out.println("==========第 1 次撤销==========");
        articleMemento = draftsBox.getMemento();
        editor.undoFromMemento(articleMemento);
        System.out.println("完整的信息" + editor);
        System.out.println("==========第 1 次撤销完成==========");


        System.out.println("==========第 2 次撤销==========");
        articleMemento = draftsBox.getMemento();
        editor.undoFromMemento(articleMemento);
        System.out.println("完整的信息" + editor);
        System.out.println("==========第 2 次撤销完成==========");

    }
```

运行结果如下：



备忘录模式在源码中的体现

备忘录模式在框架源码中的应用也是比较少的，主要还是结合具体的应用场景来使用。我在JDK 源码一顿找，目前为止还是没找到具体的应用，包括在 MyBatis 中也没有找到对应的源码。如果有小伙伴找到可以联系我。在 Spring 的 webflow 源码中还是找到一个 StateManageableMessageContext 接口，我们来看它的源代码：

```
public interface StateManageableMessageContext extends MessageContext {
```

```
    public Serializable createMessagesMemento();


    public void restoreMessages(Serializable messagesMemento);


    public void setMessageSource(MessageSource messageSource);
}
```

我们看到有一个 createMessagesMemento()方法，创建一个消息备忘录。可以打开它的实现类：

```
public class DefaultMessageContext implements StateManageableMessageContext {

    private static final Log logger = LogFactory.getLog(DefaultMessageContext.class);

    private MessageSource messageSource;

    @SuppressWarnings("serial")
    private Map<Object, List<Message>> sourceMessages = new AbstractCachingMapDecorator<Object, List<Message>>(
            new LinkedHashMap<Object, List<Message>>()) {

        protected List<Message> create(Object source) {
            return new ArrayList<Message>();
        }
    };

     ...

     public void clearMessages() {
        sourceMessages.clear();
    }

    // implementing state manageable message context

    public Serializable createMessagesMemento() {
        return new LinkedHashMap<Object, List<Message>>(sourceMessages);
    }

    @SuppressWarnings("unchecked")
    public void restoreMessages(Serializable messagesMemento) {
        sourceMessages.putAll((Map<Object, List<Message>>) messagesMemento);
    }

    public void setMessageSource(MessageSource messageSource) {
        if (messageSource == null) {
            messageSource = new DefaultTextFallbackMessageSource();
        }
        this.messageSource = messageSource;
    }

     ...
}
```

我们看到其主要逻辑就相当于是给 Message 留一个备份，以备恢复之用。

备忘录模式的优缺点

优点：

1、简化发起人实体类（Originator）职责，隔离状态存储与获取，实现了信息的封装，客户端无需关心状态的保存细节；

2、提供状态回滚功能；

缺点：

1、消耗资源：如果需要保存的状态过多时，每一次保存都会消耗很多内存。