备忘录模式

概念

- 备忘录模式（Memento Pattern）是一种行为型设计模式，它允许在不破坏封装性的前提下捕获和恢复对象的内部状态；这个模式特别适用于需要撤销操作的场景，比如文本编辑器、游戏存档等。
- 作用：
  - 保存对象状态：在某个时刻保存对象的状态，以便在需要时恢复。
  - 实现撤销操作：允许用户撤销操作，恢复到之前的状态。
  - 防止状态破坏：通过封装，防止外部对象破坏对象的内部状态。
- 结构：备忘录模式包含三个角色
  - 发起人（Originator）：负责创建一个包含其当前内部状态的备忘录，并使用备忘录恢复其内部状态。
  - 备忘录（Memento）：存储发起人的内部状态。
  - 管理者（Caretaker）：负责保存备忘录，但不能对备忘录的内容进行操作或检查。
- 优点
  - 封装性：保持对象的封装性，防止外部对象破坏其内部状态。
  - 简化撤销操作：通过保存状态，简化了撤销操作的实现。
- 缺点
  - 资源消耗：保存对象的状态可能会消耗大量内存，特别是当对象的状态非常复杂时。
  - 实现复杂：需要额外的类和管理逻辑来保存和恢复状态。

实例

- 定义状态（State）结构体

```
struct State
    {
        public string Name;
        public int Value;

    public State(string name, int value)
    {
        Name = name;
        Value = value;
```

```
    }
}
```

- 定义发起人（Originator）类，需要有保存状态和恢复状态的方法

```
class Originator
{
    public State CurrentState { get; set; }

    // 保存状态的方法
    public Memento SaveState()
    {
        return new Memento(CurrentState);
    }

    // 从备忘录中恢复状态
    public void RestoreState(Memento memento)
    {
        CurrentState = memento.State;
    }
}
```

- 定义备忘录（Memento）类，其中有State类型的成员变量用于保存状态

```
class Memento
{
    public State State { get; private set; }

    public Memento(State state)
    {
        State = state;
    }
}
```

- 定义管理者类（CareTaker）

```
// 管理者类
class Caretaker
{
    // 用字典保存每一个Originator的状态栈
    private Dictionary<Originator, Stack<Memento>> _mementos =
```

```
        new Dictionary<Originator, Stack<Memento>>();

    // 保存发起人的状态
    public void SaveState(Originator originator)
    {
        if (!_mementos.ContainsKey(originator))
        {
            _mementos[originator] = new Stack<Memento>();
        }
        _mementos[originator].Push(originator.SaveState());
    }

    // 从状态栈中取出发起人的上一个状态并回复
    public void Undo(Originator originator)
    {
        if (_mementos.ContainsKey(originator) && _mementos[originator].Count > 0)
        {
            originator.RestoreState(_mementos[originator].Pop());
        }
    }
}
```

- 应用

```
class Program
{
    static void Main()
    {
        Originator originator1 = new Originator();
        Originator originator2 = new Originator();
        Caretaker caretaker = new Caretaker();

        originator1.CurrentState = new State("State1", 100);
        caretaker.SaveState(originator1);

        originator2.CurrentState = new State("StateA", 200);
        caretaker.SaveState(originator2);

        originator1.CurrentState = new State("State2", 300);
        caretaker.SaveState(originator1);

        originator2.CurrentState = new State("StateB", 400);
        caretaker.SaveState(originator2);

```

```
        caretaker.Undo(originator1);
        Console.WriteLine("Originator1 Restored State: " +
            originator1.CurrentState.Name + ", " +
originator1.CurrentState.Value);

        caretaker.Undo(originator2);
        Console.WriteLine("Originator2 Restored State: " +
            originator2.CurrentState.Name + ", " +
originator2.CurrentState.Value);
    }
}
```

- 输出结果

```
Originator1 Restored State: State1, 100
Originator2 Restored State: StateA, 200
```