



# 几个常用的软件设计模式

授课教师：应时

2019-11-07





# 设计模式

- 软件设计模式 (Design pattern) , 是一套被反复使用、多数人知晓的、经过分类编目的、关于设计经验的总结。
- 使用设计模式是为了可重用代码, 让代码更容易被他人理解, 保证代码的可靠性和重用性。



# 软件设计模式的分类

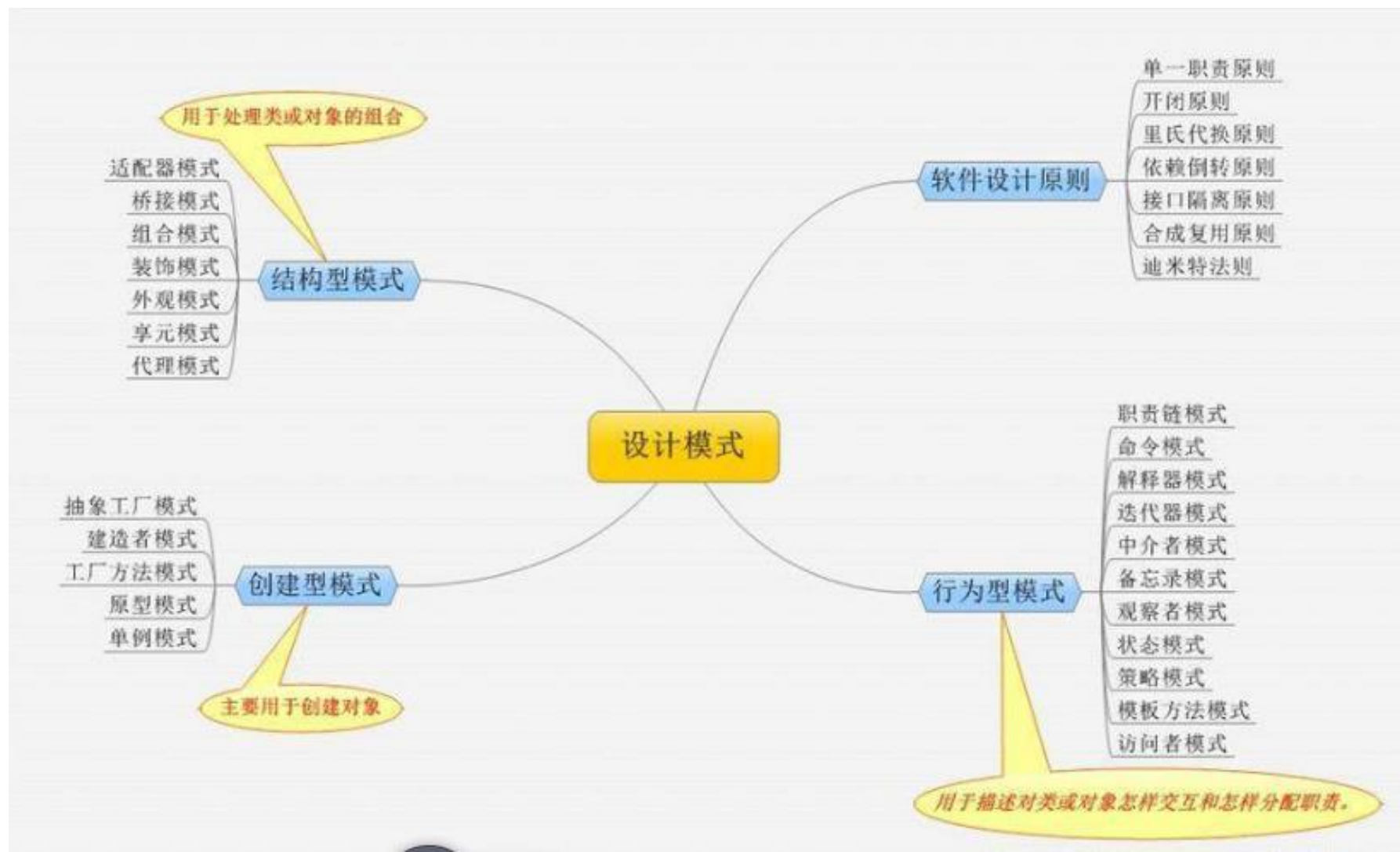
V • T • E		Software design patterns	[hide]
Gang of Four patterns	Creational	Abstract factory • Builder • Factory method • Prototype • Singleton	
	Structural	Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	
	Behavioral	Chain of responsibility • Command • Interpreter • Iterator • Mediator • Memento • Observer • State • Strategy • Template method • Visitor	
Concurrency patterns		Active object • Balking • Binding properties • Double-checked locking • Event-based asynchronous • Guarded suspension • Join • Lock • Monitor • Proactor • Reactor • Read write lock • Scheduler • Thread pool • Thread-local storage	
Architectural patterns		Front controller • Interceptor • MVC • ADR • ECS • <i>n</i> -tier • Specification • Publish–subscribe • Naked objects • Service locator • Active record • Identity map • Data access object • Data transfer object • Inversion of control • Model 2	
Other patterns		Blackboard • Business delegate • Composite entity • Dependency injection • Intercepting filter • Lazy loading • Mock object • Null object • Object pool • Servant • Twin • Type tunnel • Method chaining • Delegation	
Books		<i>Design Patterns</i> • <i>Enterprise Integration Patterns</i> • <i>Patterns of Enterprise Application Architecture</i> • <i>Analysis patterns</i>	
People		Christopher Alexander • Erich Gamma • Ralph Johnson • John Vlissides • Grady Booch • Kent Beck • Ward Cunningham • Martin Fowler • Robert Martin • Jim Coplien • Douglas Schmidt • Linda Rising	
Communities		The Hillside Group • The Portland Pattern Repository	
Authority control 		GND: 4546895-3 	

➤ [https://en.wikipedia.org/wiki/Software\\_design\\_pattern](https://en.wikipedia.org/wiki/Software_design_pattern)



# 设计模式

## ➤ GoF设计模式的分类





## ➤ GoF设计模式的分类

### ➤ 创建型

1. Factory Method (工厂方法)
2. Abstract Factory (抽象工厂)
3. Builder (建造者)
4. Prototype (原型)
5. Singleton (单例)



# 设计模式

## ➤ GoF设计模式的分类

## ➤ 结构型

6. Adapter Class/Object (适配器)

7. Bridge (桥接)

8. Composite (组合)

9. Decorator (装饰)

10. Facade (外观)

11. Flyweight (享元)

12. Proxy (代理)



# 设计模式

## ➤ 行为型

13. Interpreter (解释器)

14. Template Method (模板方法)

15. Chain of Responsibility (责任链)

16. Command (命令)

17. Iterator (迭代器)

18. Mediator (中介者)

19. Memento (备忘录)

20. Observer (观察者)

21. State (状态)

22. Strategy (策略)

23. Visitor (访问者)



## ➤ GoF设计模式的基本原则

- ① 总原则：开闭原则
- ② 单一职责原则
- ③ (里氏)替换原则
- ④ 依赖倒转原则
- ⑤ 接口隔离原则
- ⑥ (迪米特法则)最少知道原则
- ⑦ 合成复用原则





## ➤ 开放封闭原则

- 定义：一个整体的软件实例、类或者接口等，应该保持**对扩展开放，对修改封闭**的设计思想。



## ➤ 单一职责原则

- 定义：不要存在多于一个导致类变更的原因。



## ➤ 里氏代换原则

- 定义1：如果对每一个类型为 T1 的对象 o1，都有类型为 T2 的对象 o2；使得以 T1 定义的所有程序 P，在所有的对象 o1 都被代换成 o2 时，程序 P 的行为没有发生变化，那么**类型 T2 是类型 T1 的子类型**。
- 定义2：子类型必须能够替换掉它们的父类型。



## ➤ 依赖倒转原则

### ● 定义：

- ✓ 高层模块不应该依赖低层模块，二者都应该依赖其抽象。
- ✓ 抽象不应该依赖细节，细节应该依赖抽象。

### ● 即**针对接口编程，不要针对实现编程。**

- 依赖倒置原则是实现开闭原则的重要途径之一，它降低了客户与实现模块之间的耦合。
- 由于在软件设计中，细节具有多变性，而抽象层则相对稳定，因此以抽象为基础搭建起来的架构要比以细节为基础搭建起来的架构要稳定得多。这里的抽象指的是接口或者抽象类，而细节是指具体的实现类。
- 使用接口或者抽象类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给它们的实现类去完成。



## ➤ 接口隔离原则

- 定义：建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。
  - ✓ 也就是说，为各个类建立专用的接口。不要试图去建立一个很庞大的接口，供所有依赖它的类，去调用。
  - ✓ 在程序设计中，依赖几个专用的接口，要比依赖一个综合的接口更灵活。
- 接口是设计时，对外部设定的“契约”。
- 通过分散定义多个接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性。



## ➤ 迪米特法则

- 定义：强调类之间的松耦合。类之间的耦合越弱，越有利于复用。
- 一个处在弱耦合的类被修改，不会对有关系的类，造成影响。



## ➤ 合成复用原则

- 定义：在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分。新的对象，通过向这些对象的委派，达到复用已有功能的目的。



# 几种代表性的设计模式





# 设计模式—责任链模式

- 责任链（Chain of Responsibility）模式的定义：
  - 为了避免请求的发送对象与请求的多个处理对象耦合在一起，将请求的所有处理对象，通过由前一个处理者对象记住其下一个处理者对象的引用，而连成一条关于请求的处理责任链。
  - 当有请求发生时，可将请求沿着这条链传递，直到有处理者对象处理这一请求为止。
- 在责任链模式中，客户对象只需要将请求发送到责任链上即可，无须关心请求的处理细节和请求的传递过程，所以责任链将请求的发送者和请求的处理者解耦了。
- 在现实生活中，常常会出现这样的事例：一个请求可能有多个对象可以处理，但每个对象的处理条件或权限不同。
  - 例如，公司员工请假，可批假的领导有部门负责人、副总经理、总经理等，但每个领导能批准的天数不同。员工必须根据自己要请假的天数，去找不同的领导签名。也就是说员工必须记住每个领导的姓名、电话和地址等信息，这增加了难度。



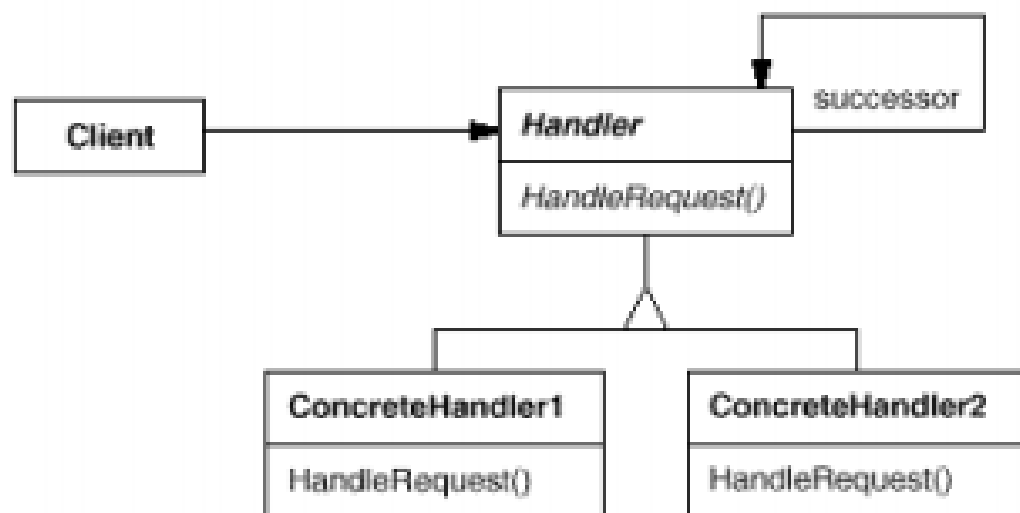
# 设计模式--责任链模式

- 责任链模式的结构
- 职责链模式主要包含以下角色：
  - 抽象处理者 (Handler) 角色：定义一个处理某个请求的接口，包含抽象处理方法和一个后继者的连接。
  - 具体处理者 (Concrete Handler) 角色：实现抽象处理者的处理方法，判断自己能否处理本次的请求。如果可以处理请求，则处理；否则，将该请求，转给自己的后继者。
  - 客户类 (Client) 角色：创建处理责任链，并向链头的具体处理者对象提交请求，**它不关心处理细节和请求的传递过程。**



# 设计模式--责任链模式

## ➤ 责任链模式的结构图





# 设计模式--责任链模式

## ➤ 责任链模式实现代码

```
package chainOfResponsibility;
public class ChainOfResponsibilityPattern
{
    public static void main(String[] args)
    {
        //组装责任链
        Handler handler1=new ConcreteHandler1();
        Handler handler2=new ConcreteHandler2();
        handler1.setNext(handler2);
        //提交请求
        handler1.handleRequest("two");
    }
}
```

```
//抽象处理者角色
abstract class Handler
{
    private Handler next;
    public void setNext(Handler next)
    {
        this.next=next;
    }
    public Handler getNext()
    {
        return next;
    }
    //处理请求的方法
    public abstract void handleRequest(String request);
}
```



# 设计模式--责任链模式

➤ 责任链模式实现代码程序的运行结果：具体处理者2负责处理该请求！

```
//具体处理者角色1
class ConcreteHandler1 extends Handler
{
    public void handleRequest(String request)
    {
        if(request.equals("one"))
        {
            System.out.println("具体处理者1负责处理该请求！");
        }
        else
        {
            if(getNext()!=null)
            {
                getNext().handleRequest(request);
            }
            else
            {
                System.out.println("没有人处理该请求！");
            }
        }
    }
}
```

```
//具体处理者角色2
class ConcreteHandler2 extends Handler
{
    public void handleRequest(String request)
    {
        if(request.equals("two"))
        {
            System.out.println("具体处理者2负责处理该请求！");
        }
        else
        {
            if(getNext()!=null)
            {
                getNext().handleRequest(request);
            }
            else
            {
                System.out.println("没有人处理该请求！");
            }
        }
    }
}
```



# 设计模式--责任链模式

## ➤ 责任链模式的优点：

- ① 降低了对象之间的耦合度。该模式使得一个对象无须知道到底是哪一个对象处理其请求以及链的结构，发送者和接收者也无须拥有对方的明确信息。
- ② 增强了系统的可扩展性。可以根据需要，增加新的能处理请求的类。满足开闭原则。
- ③ 增强了给对象指派职责的灵活性。当工作流程发生变化，可以动态地改变职责链内的对象成员，或者调整它们的次序；也可动态地新增，或者删除处理者对象。
- ④ 责任链简化了对象之间的连接。每个对象只需保持一个指向其后继者的引用，不需保持其他所有处理者的引用，这避免了使用众多的 if 或者 if...else 语句。
- ⑤ 责任分担。每个类只需要处理自己该处理的工作；把不该自己处理的请求，传递给下一个对象。这明确各类的责任范围，符合类的单一职责原则。



# 设计模式--责任链模式

## ➤ 责任链的缺点如下：

- ① 不能保证每个请求一定会被处理。由于一个请求没有明确的接收者，所以不能保证它一定会被处理。该请求可能一直传到链的末端，都得不到处理。
- ② 对比较长的职责链，请求的处理可能涉及多个处理对象，系统性能将受到一定影响。
- ③ 职责链建立的合理性要靠客户端来保证，这增加了客户端的复杂性，并且可能会因为职责链的错误设置，而导致系统出错。



# 设计模式--状态模式

- 应用程序中的有些对象，可能会根据不同的情况，做出不同的行为。我们把这种对象，称为有状态的对象；把能影响对象行为的一个或多个属性（属性可动态变化），称为状态。当有状态的对象与外部事件产生互动时，其内部状态会发生改变，从而使得其行为也随之发生改变。
- 对这种有状态的对象编程，传统的解决方案是：
  - 将这些所有可能发生的情况全都考虑到，然后使用 if-else 语句来做状态判断，再进行不同情况的处理。
  - 但当对象的状态很多时，程序会变得很复杂。
  - 如果要增加新的状态，就要添加新的 if-else 语句。这违背了“开闭原则”，不利于程序的扩展。





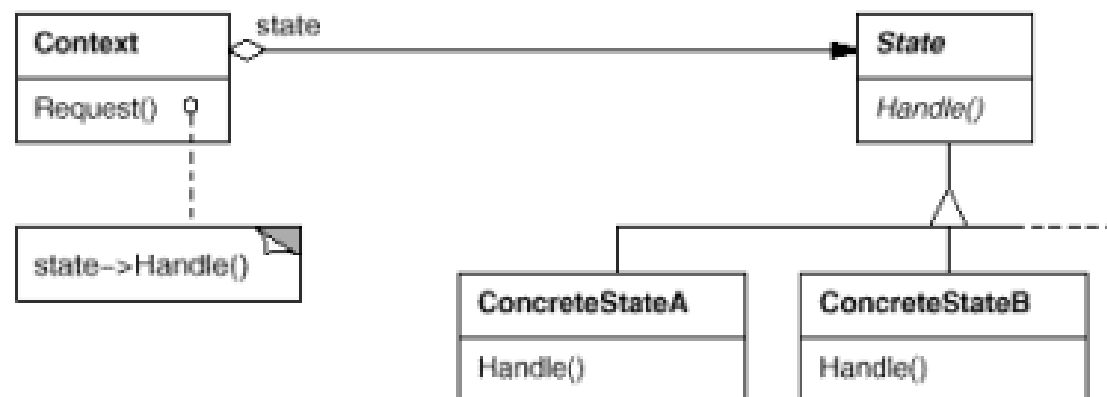
# 设计模式--状态模式

- 上述问题如果采用“状态模式”就能很好地得到解决。
- 状态模式解决方案的思想是：当控制一个对象状态转换的条件表达式过于复杂时，把相关“判断逻辑”提取出来，放到一系列的状态类当中，这样可以把原来复杂的逻辑判断简单化。
- 状态模式把受环境改变的对象行为，包装在不同的状态对象里，其意图是让一个对象在其内部状态改变的时候，其行为也随之改变。



# 设计模式--状态模式

- 状态模式的结构，状态模式包含以下主要角色：
- 环境（Context）角色：也称为上下文，它定义了客户感兴趣的接口，维护一个当前状态，并将与状态相关的操作，委托给当前状态的对象，来处理。
  - 抽象状态（State）角色：定义一个接口，用以封装环境对象中与特定状态所对应的行为。
  - 具体状态（Concrete State）角色：实现抽象状态所对应的行为。





# 设计模式--状态模式

## ➤ 状态模式的实现代码

```
package state;
public class StatePatternClient
{
    public static void main(String[] args)
    {
        Context context=new Context(); //创建环境
        context.Handle(); //处理请求
        context.Handle();
        context.Handle();
        context.Handle();
    }
}
```

```
//环境类
class Context
{
    private State state;
    //定义环境类的初始状态
    public Context()
    {
        this.state=new ConcreteStateA();
    }
    //设置新状态
    public void setState(State state)
    {
        this.state=state;
    }
    //读取状态
    public State getState()
    {
        return(state);
    }
    //对请求做处理
    public void Handle()
    {
        state.Handle(this);
    }
}
```



# 设计模式--状态模式

## ➤ 状态模式的实现代码程序，及其运行结果：

- 当前状态是 A.
- 当前状态是 B.
- 当前状态是 A.
- 当前状态是 B.

```
//抽象状态类
abstract class State
{
    public abstract void Handle(Context context);
}
```

```
//具体状态A类
class ConcreteStateA extends State
{
    public void Handle(Context context)
    {
        System.out.println("当前状态是 A.");
        context.setState(new ConcreteStateB());
    }
}
```

```
//具体状态B类
class ConcreteStateB extends State
{
    public void Handle(Context context)
    {
        System.out.println("当前状态是 B.");
        context.setState(new ConcreteStateA());
    }
}
```



# 设计模式--状态模式

## ➤ 状态模式的优点：

- ① 状态模式将与特定状态相关的行为，局部化到一个状态中；并且，将不同状态的行为，分割开来，满足“单一职责原则”。
- ② 减少对象间的相互依赖。将不同的状态引入独立的对象中，会使得状态转换，变得更加明确，减少对象间的相互依赖。
- ③ 有利于程序的扩展。通过定义新的子类，很容易地增加新的状态和转换。



# 设计模式--状态模式

## ➤ 状态模式的缺点如下：

- ① 状态模式的使用，必然会增加系统的类与对象的个数。
- ② 状态模式的结构与实现都较为复杂。如果使用不当，会导致程序结构和代码的混乱。



# 设计模式--访问者模式

- 在有些集合对象中，存在着多种不同的元素，且每种元素也存在多种不同的访问者和处理方式。
  - 例如对顾客放在“购物车”中的商品，顾客主要关心所选商品的性价比，而收银员关心的是商品的价格和数量。
- 这些被处理的数据元素，其结构相对稳定，访问方式多种多样。如果用“访问者模式”来处理比较方便。



# 设计模式--访问者模式

- 访问者（Visitor）模式的定义：将作用于某种数据结构中的各元素的操作，分离出来封装成独立的类。使得在不改变数据结构的前提下，可以添加作用于这些元素的新的操作；为数据结构中的每个元素，提供多种访问方式。
- 访问者模式提高了程序的扩展性和灵活性。
- 访问者模式将对数据的操作与数据结构进行分离，是行为类模式中最复杂的一种模式。

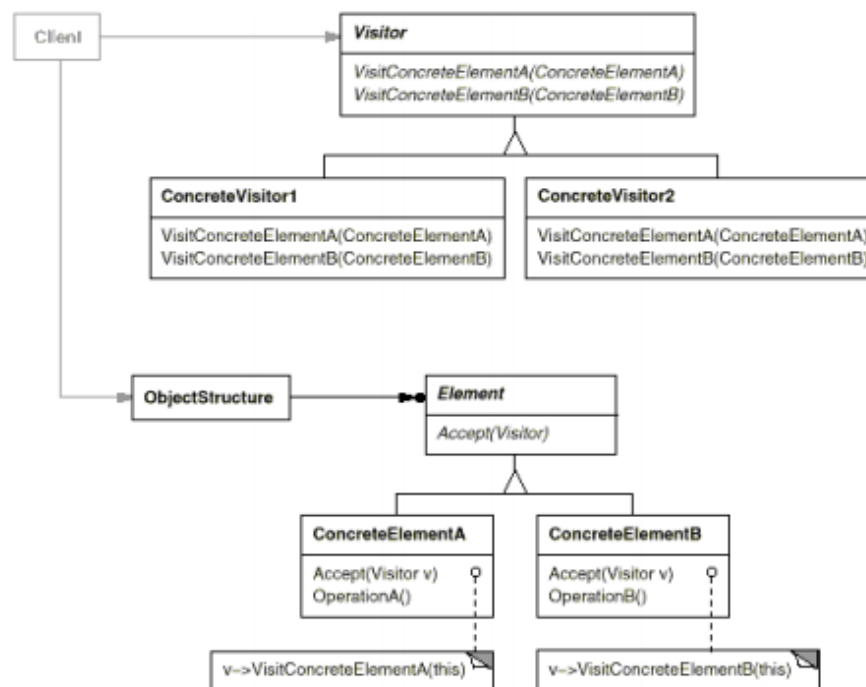




# 设计模式--访问者模式

## ➤ 访问者模式的结构，包含以下主要角色： （1/3）

- ① 抽象访问者（Visitor）角色：定义一个访问具体元素的接口。为每个具体元素类对应一个访问操作visit()。该操作中的参数类型，标识了被访问的具体元素。
- ② 具体访问者（ConcreteVisitor）角色：实现抽象访问者角色中声明的各个访问操作，确定访问者访问一个元素时该做什么。

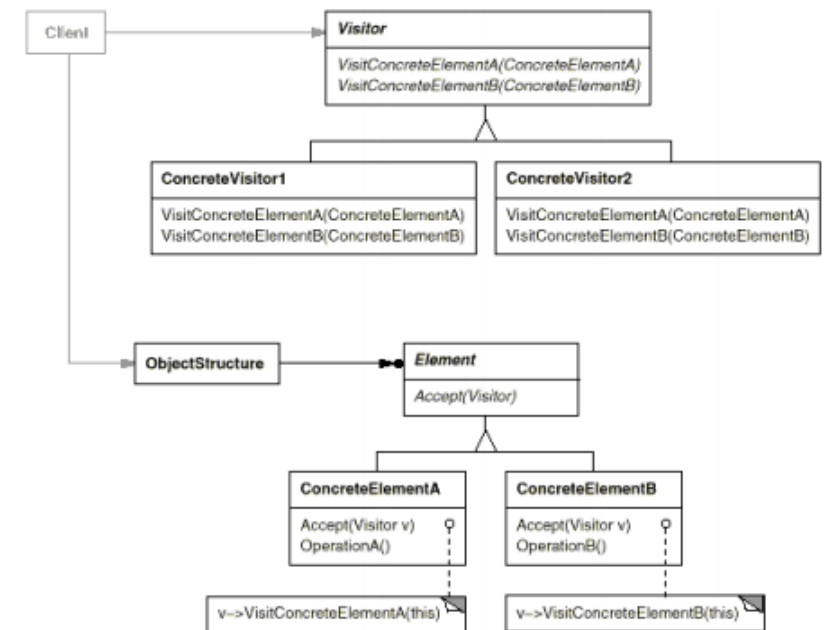




# 设计模式--访问者模式

## ➤ 访问者模式的结构，包含以下主要角色：（2/3）

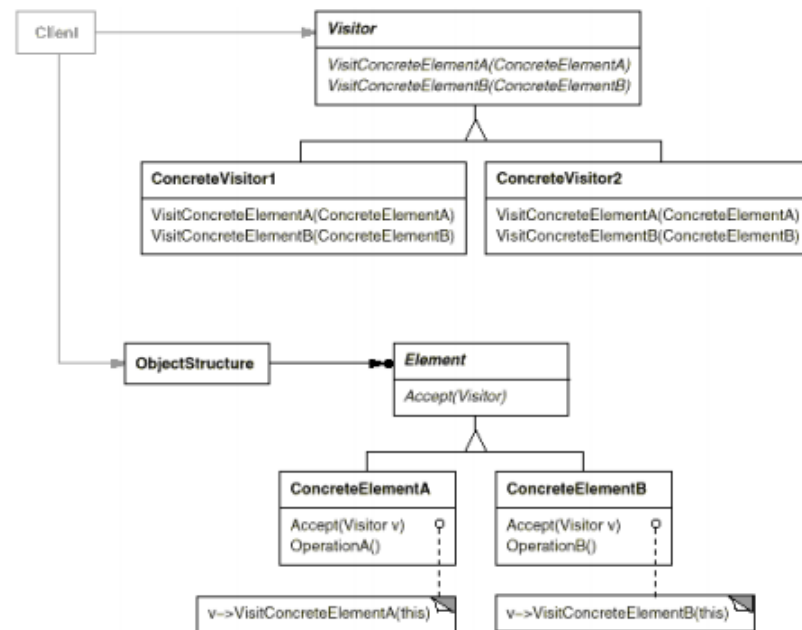
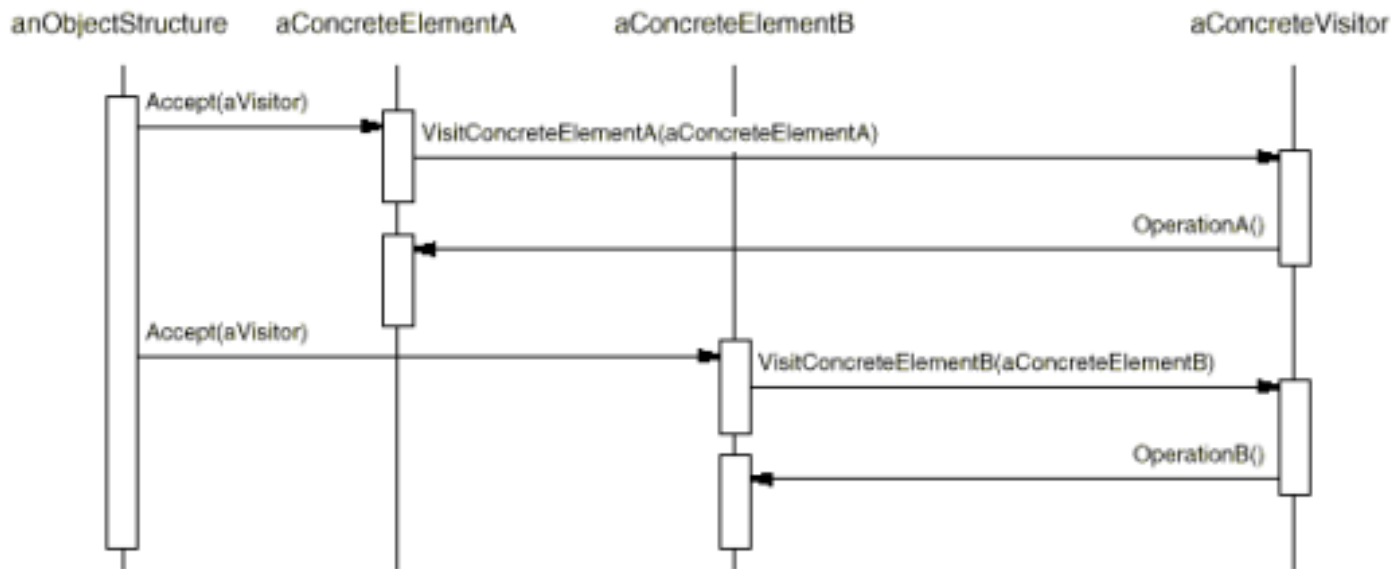
- ③ 抽象元素（Element）角色：声明一个包含接受操作 `accept()` 的接口。它把被其接受的访问者对象，作为 `accept()` 方法的参数。
- ④ 具体元素（ConcreteElement）角色：实现抽象元素角色提供的 `accept()` 操作，其方法体通常都是 `VisitConcreteElementA(this)`。具体元素中，可能还包含本身业务逻辑的相关操作。
- ⑤ 对象结构（Object Structure）角色：是一个包含元素角色的容器，提供让访问者对象遍历容器中的所有元素的方法。通常由 `List`、`Set`、`Map` 等聚合类实现。





# 设计模式--访问者模式

## 访问者模式的角色对象的交互顺序图： (3/3)





# 设计模式--访问者模式

## ➤ 访问者模式实现代码

```
package visitor;
import java.util.*;
public class VisitorPattern
{
    public static void main(String[] args)
    {
        ObjectStructure os=new ObjectStructure();
        os.add(new ConcreteElementA());
        os.add(new ConcreteElementB());
        Visitor visitor=new ConcreteVisitor1();
        os.accept(visitor);
        System.out.println("-----");
        visitor=new ConcreteVisitor2();
        os.accept(visitor);
    }
}
```

//对象结构角色

```
class ObjectStructure
{
    private List<Element> list=new ArrayList<Element>();
    public void accept(Visitor visitor)
    {
        Iterator<Element> i=list.iterator();
        while(i.hasNext())
        {
            ((Element) i.next()).accept(visitor);
        }
    }
    public void add(Element element)
    {
        list.add(element);
    }
    public void remove(Element element)
    {
        list.remove(element);
    }
}
```



# 设计模式--访问者模式

## ➤ 访问者模式实现代码

```
//抽象元素类
interface Element
{
    void accept(Visitor visitor);
}
```

```
//具体元素A类
class ConcreteElementA implements Element
{
    public void accept(Visitor visitor)
    {
        visitor.VisitConcreteElementA(this);
    }
    public String operationA()
    {
        return "具体元素A的操作。";
    }
}
```

```
//具体元素B类
class ConcreteElementB implements Element
{
    public void accept(Visitor visitor)
    {
        visitor.VisitConcreteElementB(this);
    }
    public String operationB()
    {
        return "具体元素B的操作。";
    }
}
```



# 设计模式--访问者模式

## ➤ 访问者模式实现代码

```
//抽象访问者
interface Visitor
{
    void VisitConcreteElementA(ConcreteElementA element);
    void VisitConcreteElementB(ConcreteElementB element);
}
```

```
//具体访问者1
class ConcreteVisitor1 implements Visitor
{
    public void VisitConcreteElementA(ConcreteElementA element)
    {
        System.out.println("具体访问者1访问-->" + element.operationA());
    }
    public void VisitConcreteElementB(ConcreteElementB element)
    {
        System.out.println("具体访问者1访问-->" + element.operationB());
    }
}
```

```
//具体访问者2
class ConcreteVisitor2 implements Visitor
{
    public void VisitConcreteElementA(ConcreteElementA element)
    {
        System.out.println("具体访问者2访问-->" + element.operationA());
    }
    public void VisitConcreteElementB(ConcreteElementB element)
    {
        System.out.println("具体访问者2访问-->" + element.operationB());
    }
}
```



# 设计模式--访问者模式

## ➤ 访问者模式实现代码

### ● 程序的运行结果如下:

- ✓ 具体访问者1访问-->具体元素A的操作。
- ✓ 具体访问者1访问-->具体元素B的操作。
- ✓ -----
- ✓ 具体访问者2访问-->具体元素A的操作。
- ✓ 具体访问者2访问-->具体元素B的操作。



# 设计模式--访问者模式

## ➤ 访问者模式的优点如下：

- ① 扩展性好。能够在不修改对象结构中的元素的情况下，为对象结构中的元素，添加新的访问方式。
- ② 复用性好。可以通过访问者来定义整个对象结构通用的功能，从而提高系统的复用程度。
- ③ 灵活性好。访问者模式将数据结构与作用于结构上的操作解耦，使得操作集合可相对自由地演化，而不影响系统的数据结构。
- ④ 符合单一职责原则。访问者模式把相关的行为封装在一起，构成一个访问者，使每一个访问者的功能都比较单一。





# 设计模式--访问者模式

## ➤ 访问者模式的缺点如下：

- ① 增加新的元素类很困难。在访问者模式中，每增加一个新的元素类，都要在每一个具体访问者类中增加相应的具体操作。
- ② 破坏封装。访问者模式中具体元素对访问者公布细节，这破坏了对对象的封装性。
- ③ 违反了依赖倒置原则。访问者模式依赖了具体类，而没有依赖抽象类。



# 设计模式--备忘录模式

- 备忘录模式能记录一个对象的内部状态，当用户后悔时能撤销当前操作，使数据恢复到它原先的状态。
- 其实很多应用软件都提供了这项功能
  - 如 Word、记事本、Photoshop、Eclipse 等软件，在编辑时按 Ctrl+Z 组合键时能撤销当前操作，使文档恢复到之前的状态。
  - 还有在 IE 中的后退键、数据库事务管理中的回滚操作、玩游戏时的中间结果存档功能、数据库与操作系统的备份操作、棋类游戏上的悔棋功能等都属于这类。



# 设计模式--备忘录模式

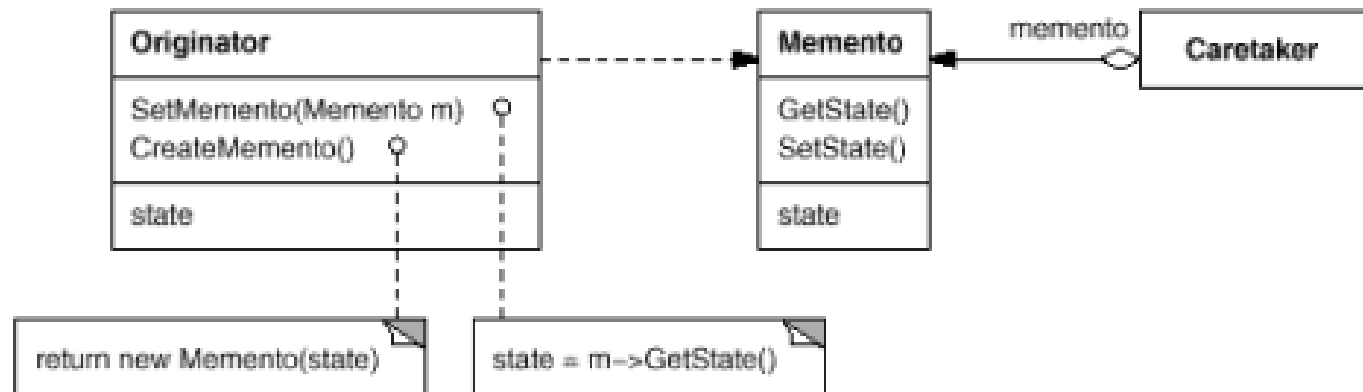
- 备忘录（Memento）模式的定义：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便以后当需要时能将该对象恢复到原先保存的状态。
- 该模式又叫快照模式。
- 备忘录模式的核心是：设计备忘录类，以及用于管理备忘录的管理者类。



# 设计模式--备忘录模式

## ➤ 备忘录模式的结构，包括的主要角色如下：

- 发起人 (Originator) 角色：记录当前时刻的内部状态信息；提供创建备忘录和恢复备忘录数据的功能；实现其他业务功能。它可以访问备忘录里的所有信息。
- 备忘录 (Memento) 角色：负责存储发起人的内部状态，在需要的时候提供这些内部状态给发起人。
- 管理者 (Caretaker) 角色：对备忘录进行管理，提供保存与获取备忘录的功能，但其不能对备忘录的内容进行访问与修改。



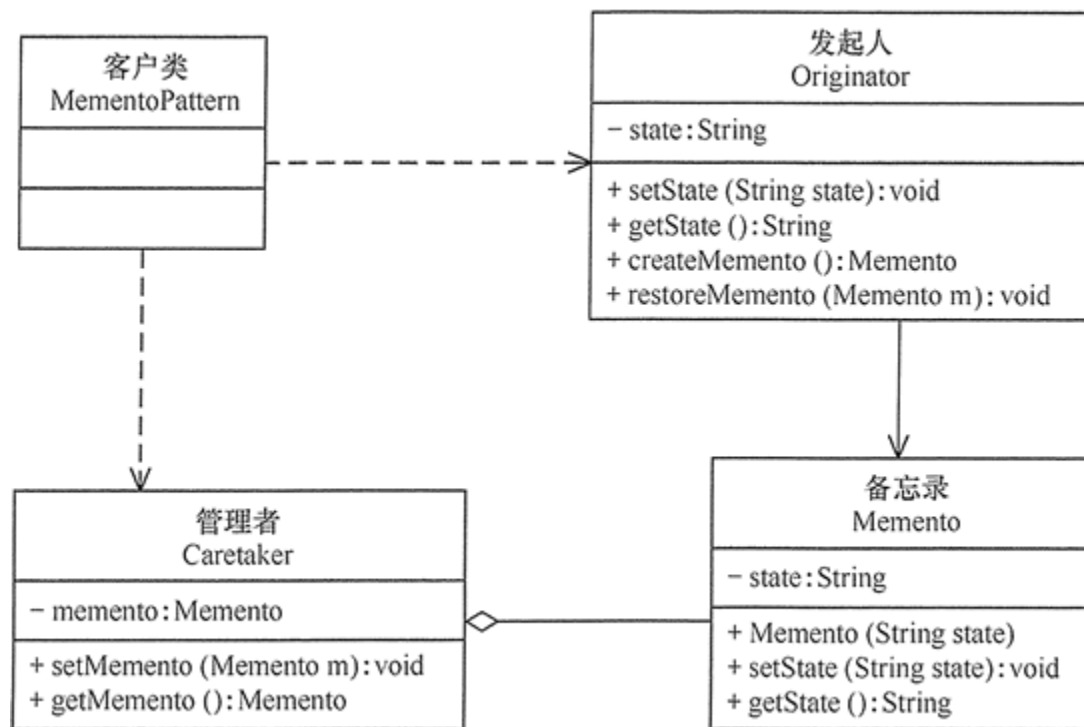


# 设计模式--备忘录模式

## ➤ 备忘录模式实现代码

```
package memento;

public class MementoPattern
{
    public static void main(String[] args)
    {
        Originator or=new Originator();
        Caretaker cr=new Caretaker();
        or.setState("S0");
        System.out.println("初始状态:"+or.getState());
        cr.setMemento(or.createMemento()); //保存状态
        or.setState("S1");
        System.out.println("新的状态:"+or.getState());
        or.restoreMemento(cr.getMemento()); //恢复状态
        System.out.println("恢复状态:"+or.getState());
    }
}
```





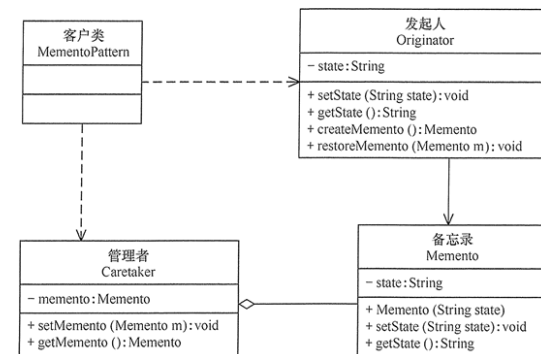
# 设计模式--备忘录模式

## ➤ 备忘录模式实现代码

```
//发起人
class Originator
{
    private String state;
    public void setState(String state)
    {
        this.state=state;
    }
    public String getState()
    {
        return state;
    }
    public Memento createMemento()
    {
        return new Memento(state);
    }
    public void restoreMemento(Memento m)
    {
        this.setState(m.getState());
    }
}
```

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

```
//管理者
class Caretaker
{
    private Memento memento;
    public void setMemento(Memento m)
    {
        memento=m;
    }
    public Memento getMemento()
    {
        return memento;
    }
}
```





# 设计模式--备忘录模式

## ➤ 备忘录模式程序运行的结果如下:

- 初始状态:S0
- 新的状态:S1
- 恢复状态:S0



# 设计模式--备忘录模式

## ➤ 备忘录模式的优点如下:

- 提供了一种可以恢复状态的机制。当用户需要时，能够比较方便地，将数据恢复到某个历史的状态。
- 实现了内部状态的封装。除了创建它的发起人之外，其他对象都不能够访问这些状态信息。
- 简化了“发起人”类。发起人不需要管理和保存其内部状态的各个备份，所有状态信息都保存在备忘录中，并由管理者进行管理，这符合单一职责原则。





# 设计模式--备忘录模式

## ➤ 备忘录模式缺点是：

- 资源消耗大。如果要保存的内部状态信息过多或者特别频繁，将会占用比较大的内存资源。



# 设计模式--MVC

- MVC是Model-View-Controller的简写。即模型-视图-控制器。
- 使用MVC应用程序被分成三个核心部件：模型、视图、控制器。它们各自处理自己的任务。

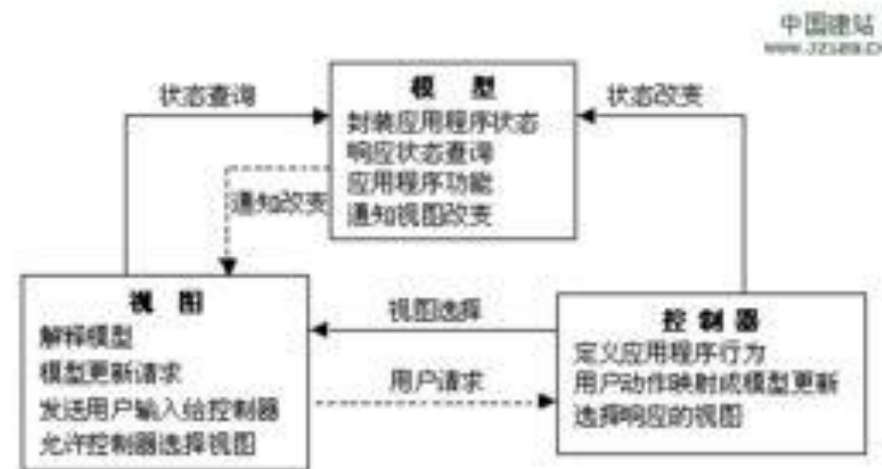


图1 MVC组件类型的关系和功能



# 设计模式--MVC

- MVC开始是存在于桌面程序中的。
- M是指业务模型，V是指用户界面，C则是控制器。
- 使用MVC的目的是将M和V的实现代码分离，从而使同一个程序可以使用不同的表现形式。
  - 比如一批统计数据可以分别用柱状图、饼图来表示。
- C存在的目的则是确保M和V的同步，一旦M改变，V应该同步更新。



# 设计模式--MVC

- MVC是Xerox PARC在二十世纪八十年代为编程语言Smalltalk - 80发明的一种软件设计模式，已被广泛使用。
- 后来被推荐为Oracle旗下Sun公司Java EE平台的设计模式。



# 设计模式--MVC

- **模型M表示企业数据和业务规则。在MVC的三个部件中，模型拥有最多的处理任务。**
  - **被模型返回的数据是中立的，就是说模型与数据格式无关，这样一个模型能为多个视图提供数据。**
  - **由于应用于模型M的代码只需写一次就可以被多个视图V重用，所以减少了代码的重复性。**



# 设计模式--MVC

- 视图V是用户看到并与之交互的界面。
  - 对老式的Web应用程序来说，视图就是由HTML元素组成的界面。
  - 在新式的Web应用程序中，HTML依旧在视图中扮演着重要的角色，但一些新的技术已层出不穷，它们包括Adobe Flash和像XHTML, XML/XSL,WML等一些标识语言和Web services。



# 设计模式--MVC

- **控制器C接受用户的输入，并调用模型和视图，去完成用户的需求。**
  - 所以当单击Web页面中的超链接和发送HTML表单时，控制器C本身不输出任何东西和做任何处理。
  - 它只是接收请求，并决定调用哪个模型构件，去处理请求。然后，再确定用哪个视图来显示返回的数据。



# 设计模式--MVVM

- MVVM是Model-View-ViewModel的简写。即模型-视图-视图模型。
- 模型（Model）指的是后端传递的数据。
- 视图(View)指的是所看到的页面。
- 视图模型(ViewModel)是MVVM模式的核心，它是连接View和Model的桥梁。它有两个方向：（数据的双向绑定）
  - 一是将模型（Model）转化成视图（View）。即将后端传递的数据，转化成所看到的页面。实现的方式是：数据绑定。
  - 二是将视图（View）转化成模型（Model）。即将所看到的页面，转化成后端的数据。实现的方式是：DOM监听事件。







# 设计模式--MVVM

- MVVM本质上就是MVC的**改进版**。MVVM的目标使业务逻辑组件得到更多的重用。
  - MVVM 就是将其中的View的状态和行为抽象化，把**视图UI和业务逻辑**分开。
  - ViewModel在取出Model数据（即完成数据逻辑处理）的同时，处理View中由于需要展示内容，而涉及的业务逻辑。
- MVVM是Web前端一种非常流行的开发模式。
- 目前著名的MVVM框架有VUE, avalon, angular等。



# 设计模式--MVVM

## ➤ MVVM设计模式的优点

- ① 可重用性好：可以把一些视图相关的业务逻辑，放在一个ViewModel里面，让很多View重用这段业务逻辑。
- ② 便于分工开发：开发人员专注于业务逻辑和数据的开发（ViewModel），设计人员专注于页面设计。
- ③ 数据一致性好：双向绑定技术，使得当Model变化时，ViewModel会自动更新，View也会自动更新。很好地保证了数据的一致性。



# 设计模式--MVVM

## ➤ MVVM设计模式的缺点

- ① 数据绑定使得bug很难被调试。比如页面出现异常，可能是View的代码有问题，也可能是Model的代码有问题。
- ② 数据的双向绑定会影响代码重用。数据的双向绑定，使得不同的Model在重用一個View时会产生一些麻烦。



*The End*

