前端设计模式

1. 课前准备

设计模式概念

2. 课堂主题

3. 课堂目标

4. 知识点

> 设计模式（Design Pattern）是一套被反复使用、多数人知晓的、经过分类的、代码设计经验的总结。

任何事情都有套路，设计模式，就是写代码中的常见套路， 有些写法我们日常都一直在使用，下面我们来介绍一下

订阅/发布模式 （观察者）

pub/sub 这个应该大家用到最广的设计模式了，

在这种模式中，并不是一个对象调用另一个对象的方法，而是一个对象订阅另一个对象的 特定活动并在状态改编后获得通知。订阅者因此也成为观察者，而被观察的对象成为发布者或者主题。当发生了一个重要事件时候 发布者会通知（调用）所有订阅者并且可能经常已事件对象的形式传递消息。

自己实现一个也贼简单

```
class Event{
    constructor(){
        this.callbacks = {}
    }
    $off(name){
        this.callbacks[name] = null
    }
    $emit(name, args){
        let cbs = this.callbacks[name]
        if (cbs) {
            cbs.forEach(c=>{
                c.call(this, args)
            })
          }
    }
    $on(name, fn){
        (this.callbacks[name] || (this.callbacks[name] = [])).push(fn)
    }
}
let event = new Event()
event.$on('event1', function(arg){
    console.log('事件1',arg)
})
event.$on('event1', function(arg){
    console.log('又一个时间1',arg)
})
event.$on('event2', function(arg){
```

```
    console.log('事件2',arg)
})

event.$emit('event1',{name:'开课吧'})
event.$emit('event2',{name:'全栈'})

event.$off('event1')
event.$emit('event1',{name:'开课吧'})
```

vue中的$emit$,on源码 大概也是这个样子

https://github.com/vuejs/vue/blob/dev/src/core/instance/events.js#L54

单例模式

> 单例模式的定义：保证一个类仅有一个实例，并提供一个访问它的全局访问点。实现的方法为先判断实例存在与否，如果存在则直接返回，如果不存在就创建了再返回，这就确保了一个类只有一个实例对象。

适用场景：一个单一对象。比如：弹窗，无论点击多少次，弹窗只应该被创建一次' 实现起来也很简单，用一个变量缓存即可

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
    <style>
        .model{
            border:1px solid black;
            position: fixed;
            width:300px;
            height:300px;
            top:20%;
            left:50%;
            margin-left:-150px;
            text-align: center;

        }
    </style>
</head>
<body>
    <div id="loginBtn">点我</div>
```

```
    <script>

        var getSingle = function( fn ){
   var result;
   return function(){
       return result || ( result = fn .apply(this, arguments ) );
   }
};
var createLoginLayer = function(){

    var div = document.createElement( 'div' );
    div.innerHTML = '我是登录浮窗';
    div.className = 'model'
    div.style.display = 'none';
    document.body.appendChild( div );

    return div;
};
var createSingleLoginLayer = getSingle( createLoginLayer );
document.getElementById( 'loginBtn' ).onclick = function(){
    var loginLayer = createSingleLoginLayer();
    loginLayer.style.display = 'block';
};



</script>
</body>
</html>
```

应用场景

我们再element中的弹窗代码中，可以看到单例模式的实际案例 保证全局唯一性 https://github.com/ElemeFE/element/blob/dev/packages/message-box/src/main.js#L79

策略模式

策略模式的定义：定义一系列的算法，把他们一个个封装起来，并且使他们可以相互替换。

策略模式的目的就是将算法的使用算法的实现分离开来。

一个基于策略模式的程序至少由两部分组成。第一个部分是一组策略类（可变），策略类封装了具体的算法，并负责具体的计算过程。第二个部分是环境类Context（不变），Context接受客户的请求，随后将请求委托给某一个策略类。要做到这一点，说明Context中要维持对某个策略对象的引用

举个栗子

奖金计算，绩效为 S 的人年 终奖有 4 倍工资，绩效为 A 的人年终奖有 3 倍工资，而绩效为 B 的人年终奖是 2 倍工资

```
var calculateBonus = function( performanceLevel, salary ){
    if ( performanceLevel === 'S' ){
         return salary * 4;
    }
    if ( performanceLevel === 'A' ){
         return salary * 3;
    }
    if ( performanceLevel === 'B' ){
         return salary * 2;
    }
};
calculateBonus( 'B', 20000 ); // 输出:40000
calculateBonus( 'S', 6000 ); // 输出:24000
```

使用策略模式

```
var strategies = {
    "S": function( salary ){
         return salary * 4;
     },
    "A": function( salary ){
         return salary * 3;
    },
    "B": function( salary ){
         return salary * 2;
    }
};
var calculateBonus = function( level, salary ){
    return strategies[ level ]( salary );
};
console.log( calculateBonus( 'S', 20000 ) );// 输出:80000
console.log( calculateBonus( 'A', 10000 ) );// 输出:30000
```

表单校验

```
// 正常写法
var registerForm = document.getElementById( 'registerForm' );
registerForm.onsubmit = function(){
    if ( registerForm.userName.value === '' ){
        alert ( '用户名不能为空' );
        return false;
    }
    if ( registerForm.password.value.length < 6 ){
        alert ( '密码长度不能少于 6 位' );
        return false;
    }
    if ( !/(^1[3|5|8][0-9]{9}$)/.test( registerForm.phoneNumber.value ) ){
        alert ( '手机号码格式不正确' );
        return false;
    }
}
```

使用策略模式

```
var strategies = {
    isNonEmpty: function( value, errorMsg ){
        if ( value === '' ){
            return errorMsg ;
        }
    },
    minLength: function( value, length, errorMsg ){
        if ( value.length < length ){
            return errorMsg;
        }
    },
    isMobile: function( value, errorMsg ){ // 手机号码格式
        if ( !/(^1[3|5|8][0-9]{9}$)/.test( value ) ){
            return errorMsg;
        }
    }
};
var Validator = function(){
    this.cache = []; // 保存校验规则
};
Validator.prototype.add = function(
    var ary = rule.split( ':' );
    this.cache.push(function(){ //
        var strategy = ary.shift();
        ary.unshift( dom.value );
        ary.push( errorMsg ); //
```

```
        return strategies[strategy].apply(dom, ary);
    });
};
Validator.prototype.start = function(){
    for ( var i = 0, validatorFunc; validatorFunc = this.cache[ i++ ]; ){
        var msg = validatorFunc(); // 开始校验，并取得校验后的返回信息
        if ( msg ){ // 如果有确切的返回值，说明校验没有通过
              return msg;
        }
    }
};
var validataFunc = function(){
    var validator = new Validator(); // 创建一个 validator 对象
    /***************添加一些校验规则***************/
    validator.add( registerForm.userName, 'isNonEmpty', '用户名不能为空' );

    validator.add( registerForm.password, 'minLength:6', '密码长度不能少于 6位');

    validator.add( registerForm.phoneNumber, 'isMobile', '手机号码格式不正确' );
    var errorMsg = validator.start(); // 获得校验结果
    return errorMsg; // 返回校验结果
}
var registerForm = document.getElementById( 'registerForm' );
registerForm.onsubmit = function(){
    var errorMsg = validataFunc(); // 如果 errorMsg 有确切的返回值，说明未通过校验
    if ( errorMsg ){
        alert ( errorMsg );
        return false; // 阻止表单提交
    }
};
```

代理模式

代理模式的定义：为一个对象提供一个代用品或占位符，以便控制对它的访问。

常用的虚拟代理形式：某一个花销很大的操作，可以通过虚拟代理的方式延迟到这种需要它的时候才去创建（例：使用虚拟代理实现图片懒加载）

图片懒加载的方式：先通过一张loading图占位，然后通过异步的方式加载图片，等图片加载好了再把完成的图片加载到img标签里面。

```
var imgFunc = (function() {
    var imgNode = document.createElement('img');
    document.body.appendChild(imgNode);
    return {
        setSrc: function(src) {
```

```
            imgNode.src = src;
        }
    }
})();
var proxyImage = (function() {
    var img = new Image();
    img.onload = function() {
        imgFunc.setSrc(this.src);
    }
    return {
        setSrc: function(src) {
            imgFunc.setSrc('./loading,gif');
            img.src = src;
        }
    }
})();
proxyImage.setSrc('./pic.png');
```

假设我们在做一个文件同步的功能，当我们选中一个 checkbox 的时候，它对应的文件就会被同 步到另外一台备用服务器上面。当一次选中过多时，会产生频繁的网络请求。将带来很大的开销。可以通过一个代理函数 proxySynchronousFile 来收集一段时间之内的请求， 最后一次性发送给服务器

```
var synchronousFile = function( id ){
    console.log( '开始同步文件，id 为: ' + id );
};
var proxySynchronousFile = (function(){
    var cache = [], // 保存一段时间内需要同步的 ID
    timer; // 定时器
    return function( id ){
        cache.push( id );
            if ( timer ){ // 保证不会覆盖已经启动的定时器
                 return;
            }
        timer = setTimeout(function(){
            synchronousFile( cache.join( ',' ) );
            clearTimeout( timer ); // 清空定时器
            timer = null;
            cache.length = 0; // 清空 ID 集合
        }, 2000 );
    }// 2 秒后向本体发送需要同步的 ID 集合
})();

var checkbox = document.getElementsByTagName( 'input' );
   for ( var i = 0, c; c = checkbox[ i++ ]; ){
      c.onclick = function(){
      if ( this.checked === true ){
          proxySynchronousFile( this.id ); }
```

```
    }
};
```

中介者模式

中介者模式的定义：通过一个中介者对象，其他所有的相关对象都通过该中介者对象来通信，而不是相互引用，当其中的一个对象发生改变时，只需要通知中介者对象即可。通过中介者模式可以解除对象与对象之间的紧耦合关系。

例如：现实生活中，航线上的飞机只需要和机场的塔台通信就能确定航线和飞行状态，而不需要和所有飞机通信。同时塔台作为中介者，知道每架飞机的飞行状态，所以可以安排所有飞机的起降和航线安排。

中介者模式适用的场景：例如购物车需求，存在商品选择表单、颜色选择表单、购买数量表单等等，都会触发change事件，那么可以通过中介者来转发处理这些事件，实现各个事件间的解耦，仅仅维护中介者对象即可。

redux，vuex 都属于中介者模式的实际应用，我们把共享的数据，抽离成一个单独的store， 每个都通过store这个中介来操作对象

目的就是减少耦合

装饰器模式

装饰者模式的定义：在不改变对象自身的基础上，在程序运行期间给对象动态地添加方法。常见应用，react的高阶组件, 或者react-redux中的@connect 或者自己定义一些高阶组件

```
import React from 'react'
const withLog = Component=>{
  // 类组件
  class NewComponent extends React.Component{
    componentWillMount(){
      console.time(`CompoentRender`)
      console.log(`准备完毕了`)
    }
    render(){
      return <Component {...this.props}></Component>
    }
```

```
    componentDidMount(){
      console.timeEnd(`CompoentRender`)
      console.log(`渲染完毕了`)
    }
  }
  return NewComponent
}
export {withLog}


@withLog
class XX
```

```
export const connect = (mapStateToProps = state => state, mapDispatchToProps =
{}) => (WrapComponent) => {
  return class ConnectComponent extends React.Component {
    static contextTypes = {
      store: PropTypes.object
    }
    constructor(props, context) {
      super(props, context)
      this.state = {
        props: {}
      }
    }
    componentDidMount() {
      const { store } = this.context
      // 当前状态 update 后，放入监听器中，用于下一次的更新(每次 dispatch 后会执行
subscribe 中的所有函数)
      store.subscribe(() => this.update())
      this.update()
    }
    update() {
      const { store } = this.context
      const stateProps = mapStateToProps(store.getState())
      const dispatchProps = bindActionCreators(mapDispatchToProps,
store.dispatch)
      this.setState({
        props: {
          ...this.state.props,
          ...stateProps,
          ...dispatchProps
        }
      })
    }
    render() {
      return <WrapComponent {...this.state.props}></WrapComponent>
    }
```

```
  }
}
```

假设我们在编写一个飞机大战的游戏，随着经验值的增加，我们操作的飞机对象可以升级成更厉害的飞机，一开始这些飞机只能发射普通的子弹，升到第二级时可以发射导弹，升到第三级时可以发射原子弹。

```
Function.prototype.before = function( beforefn ){
    var __self = this; // 保存原函数的引用
    return function(){ // 返回包含了原函数和新函数的"代理"函数
        beforefn.apply( this, arguments ); // 执行新函数，且保证 this 不被劫持，新函数接受的参数 // 也会被原封不动地传入原函数，新函数在原函数之前执行
        return __self.apply( this, arguments ); // 执行原函数并返回原函数的执行结果，   // 并且保证 this 不被劫持
} }
Function.prototype.after = function( afterfn ){
    var __self = this;
    return function(){
        var ret = __self.apply( this, arguments );
        afterfn.apply( this, arguments );
        return ret;
    }
};
```

比如页面中有一个登录 button，点击这个 button 会弹出登录浮层，与此同时要进行数据上报， 来统计有多少用户点击了这个登录 button

```
var showLogin = function(){
   console.log( '打开登录浮层' );
   log( this.getAttribute( 'tag' ) );
}
var log = function( tag ){
   console.log( '上报标签为: ' + tag );
   (new Image).src = 'http:// xxx.com/report?tag=' + tag;
}
document.getElementById( 'button' ).onclick = showLogin;
```

使用装饰器

```
var showLogin = function(){
    console.log( '打开登录浮层' );
}
var log = function(){
    console.log( '上报标签为: ' + this.getAttribute( 'tag' ) );
}
showLogin = showLogin.after( log ); // 打开登录浮层之后上报数据
document.getElementById( 'button' ).onclick = showLogin;
```

装饰者模式和代理模式的结构看起来非常相像，这两种模式都描述了怎样为对象提供 一定程度上的间接引用，它们的实现部分都保留了对另外一个对象的引用，并且向那个对象发送 请求。 代理模式和装饰者模式最重要的区别在于它们的意图和设计目的。代理模式的目的是，当直接访问本体不方便或者不符合需要时，为这个本体提供一个替代者。本体定义了关键功能，而代理提供或拒绝对它的访问，或者在访问本体之前做一些额外的事情。装饰者模式的作用就是为对 象动态加入行为。

其实Vue中的v-input，v-checkbox也可以认为是装饰器模式， 对原生的input和checkbox做一层装饰

外观模式

外观模式即让多个方法一起被调用

涉及到兼容性，参数支持多格式，有很多这种代码，对外暴露统一的api，比如上面的$on 支持数组，￥off参数支持多个情况， 对面只用一个函数，内部判断实现

自己封装组件库 经常看到

```
myEvent = {
    stop: function(e) {
        if (typeof e.preventDefault() === "function") {
            e.preventDefault();
        }
        if (typeof e.stopPropagation() === "function") {
            e.stopPropagation();
        }
        //for IE
        if (typeof e.returnValue === "boolean") {
            e.returnValue = false;
        }
        if (typeof e.cancelBubble === "boolean") {
            e.cancelBubble = true;
        }
    }
    addEvent(dom, type, fn) {
```

```
      if (dom.addEventListener) {
        dom.addEventListener(type, fn, false);
      } else if (dom.attachEvent) {
        dom.attachEvent('on' + type, fn);
      } else {
        dom['on' + type] = fn;
      }
  }
}
```

工厂模式

提供创建对象的接口，把成员对象的创建工作转交给一个外部对象，好处在于消除对象之间的耦合(也就是相互影响)

常见的例子，我们的弹窗，message，对外提供的api，都是调用api，然后新建一个弹窗或者Message的实例，就是典型的工厂模式

```
const Notification = function(options) {
  if (Vue.prototype.$isServer) return;
  options = options || {};
  const userOnClose = options.onClose;
  const id = 'notification_' + seed++;
  const position = options.position || 'top-right';

  options.onClose = function() {
    Notification.close(id, userOnClose);
  };

  instance = new NotificationConstructor({
    data: options
  });

  if (isVNode(options.message)) {
    instance.$slots.default = [options.message];
    options.message = 'REPLACED_BY_VNODE';
  }
  instance.id = id;
  instance.$mount();
  document.body.appendChild(instance.$el);
  instance.visible = true;
  instance.dom = instance.$el;
  instance.dom.style.zIndex = PopupManager.nextZIndex();
```

```
  let verticalOffset = options.offset || 0;
  instances.filter(item => item.position === position).forEach(item => {
    verticalOffset += item.$el.offsetHeight + 16;
  });
  verticalOffset += 16;
  instance.verticalOffset = verticalOffset;
  instances.push(instance);
  return instance;
};
```

https://github.com/ElemeFE/element/blob/dev/packages/notification/src/main.js#L11

建造者模式

和工长模式相比，参与了更多创建的过程 或者更复杂

```
var Person = function(name, work) {
  // 创建应聘者缓存对象
  var _person = new Human();

  // 创建应聘者姓名解析对象
  _person.name = new Named(name);

  // 创建应聘者期望职位
  _person.work = new Work(work);

  return _person;
};

var person = new Person('xiao ming', 'code');
console.log(person)
```

迭代器模式

> 迭代器模式是指提供一种方法顺序访问一个聚合对象中的各个元素，而又不需要暴露该对象的内部表示。迭代器模式可以把迭代的过程从业务逻辑中分离出来,在使用迭代器模式之后，即使不关心对象的内部构造，也可以按顺序访问其中的每个元素

这个用的就太多了 each map啥乱遭的

```
var each = function( ary, callback ){
    for ( var i = 0, l = ary.length; i < l; i++ ){
         callback.call( ary[i], i, ary[ i ] );
    }
};
each( [ 1, 2, 3 ], function( i, n ){
    alert ( [ i, n ] );
})
```

享元模式

> 享元(flyweight)模式是一种用于性能优化的模式，“fly”在这里是苍蝇的意思，意为蝇量级。享元模式的核心是运用共享技术来有效支持大量细粒度的对象。 如果系统中因为创建了大量类似的对象而导致内存占用过高，享元模式就非常有用了。在 JavaScript 中，浏览器特别是移动端的浏览器分配的内存并不算多，如何节省内存就成了一件非常有意义的事情。

假设有个内衣工厂，目前的产品有 50 种男式内衣和 50 种女士内衣，为了推销产品，工厂决定生产一些塑料模特来穿上他们的内衣拍成广告照片。 正常情况下需要 50个男模特和50个女模特，然后让他们每人分别穿上一件内衣来拍照。

```
var Model = function( sex, underwear){
    this.sex = sex;
    this.underwear = underwear;
};
Model.prototype.takePhoto = function(){
    console.log( 'sex= ' + this.sex + ' underwear=' + this.underwear);
};
for ( var i = 1; i <= 50; i++ ){
    var maleModel = new Model( 'male', 'underwear' + i );
    maleModel.takePhoto();
 };
for ( var j = 1; j <= 50; j++ ){
    var femaleModel= new Model( 'female', 'underwear' + j );
    femaleModel.takePhoto();
};
```

如上所述，现在一共有 50 种男内 衣和 50 种女内衣，所以一共会产生 100 个对象。如果将来生产了 10000 种内衣，那这个程序可能会因为存在如此多的对象已经提前崩溃。 下面我们来考虑一下如何优化这个场景。虽然有 100 种内衣，但很显然并不需要 50 个男 模特和 50 个女模特。其实男模特和女模特各自有一个就足够了，他们可以分别穿上不同的内衣来拍照。

```
/*只需要区别男女模特
那我们先把 underwear 参数从构造函数中 移除，构造函数只接收 sex 参数*/
var Model = function( sex ){
```

```
    this.sex = sex;
};
Model.prototype.takePhoto = function(){
    console.log( 'sex= ' + this.sex + ' underwear=' + this.underwear);
};
/*分别创建一个男模特对象和一个女模特对象*/
var maleModel = new Model( 'male' ),
    femaleModel = new Model( 'female' );
/*给男模特依次穿上所有的男装，并进行拍照*/
for ( var i = 1; i <= 50; i++ ){
    maleModel.underwear = 'underwear' + i;
    maleModel.takePhoto();
};
/*给女模特依次穿上所有的女装，并进行拍照*/
for ( var j = 1; j <= 50; j++ ){
    femaleModel.underwear = 'underwear' + j;
    femaleModel.takePhoto();
};
//只需要两个对象便完成了同样的功能
```

- 内部状态存储于对象内部
- 内部状态可以被一些对象共享
- 内部状态独立于具体的场景，通常不会改变
- 外部状态取决于具体的场景，并根据场景而变化，外部状态不能被共享

性别是内部状态，内衣是外部状态，通过区分这两种状态，大大减少了系 统中的对象数量。通常来讲，内部状态有多少种组合，系统中便最多存在多少个对象，因为性别 通常只有男女两种，所以该内衣厂商最多只需要 2 个对象。

应用案例

消息组件

需求

1. 弹窗逻辑一样
2. 四中弹窗，颜色，icon不同
3. 接收文案

> 交互方式——弹出、隐藏，由共享对象所拥有
>
> 提示icon、背景样式、字体样式提供接口可配置
>
> 使用api统一

```
//Message.js 伪代码
```

```
export default {
  install (Vue) {
    // 在使用插件Vue.use(Message)时实例化一个Dialog组件对象
    const Dialog = new Vue({
      data () {
        return {
          icon: '',
          fontStyle: '',
          backgroundStyle: '',
          text: ''
        }
      }
      ...
    })

    // 扩展Vue的`prototype`
    Vue.prototype.$Message = {
      success (text) {
        // 改变Dialog的data.xx的值触发Dialog的更新
        Dialog.icon = successIcon
        Dialog.fontStyle = successFontStyle
        Dialog.backgroundStyle = successBackgroundStyle
        Dialog.text = text
        // 获取Dialog的最新DOM添加到body标签中
        document.body.appendChild(Dialog.$el)
      },
      warning (text) {
        // 同上
        ...
        document.body.appendChild(Dialog.$el)
      },
      error (text) {
        // 同上
        ...
        document.body.appendChild(Dialog.$el)
      }
    }
  }
}
```

Dialog只会在项目初始化时被`new`一次，每次使用Message组件通过改变Dialog的状态获取组件DOM，其实很容易知道new一个组件的成本要比一个组件的更新成本高很多

职责链模式

职责链模式的定义是:使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系，将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。 职责链模式的名字非常形象，一系列可能会处理请求的对象被连接成一条链，请求在这些对 象之间依次传递，直到遇到一个可以处理它的对象，我们把这些对象称为链中的节点

假设我们负责一个售卖手机的电商网站，经过分别交纳 500 元定金和 200 元定金的两轮预定后(订单已在此时生成)，现在已经到了正式购买的阶段。 公司针对支付过定金的用户有一定的优惠政策。在正式购买后，已经支付过 500 元定金的用 户会收到 100 元的商城优惠券，200 元定金的用户可以收到 50 元的优惠券，而之前没有支付定金的用户只能进入普通购买模式，也就是没有优惠券，且在库存有限的情况下不一定保证能买到。

```
var order = function( orderType, pay, stock ){
    if ( orderType === 1 ){ // 500 元定金购买模式
        if ( pay === true ){ // 已支付定金
            console.log( '500 元定金预购, 得到 100 优惠券' );
        } else{ // 未支付定金，降级到普通购买模式
            if ( stock > 0 ){ // 用于普通购买的手机还有库存
                console.log( '普通购买, 无优惠券' );
            }else{
                console.log( '手机库存不足' );
            }
        }
    } else if ( orderType === 2 ){
        if ( pay === true ){ // 200 元定金购买模式
            console.log( '200 元定金预购, 得到 50 优惠券' );
        }else{
            if ( stock > 0 ){
                console.log( '普通购买, 无优惠券' );
            }else{
                console.log( '手机库存不足' );
            }
        }
    } else if (orderType === 3) {
        if ( stock > 0 ){
            console.log( '普通购买, 无优惠券' );
        } else{
            console.log( '手机库存不足' );
        }
    }
};
order( 1 , true, 500); // 输出: 500 元定金预购, 得到 100 优惠券
```

现在我们采用职责链模式重构这段代码，先把 500 元订单、200 元订单以及普通购买分成 3 个函数。接下来把 orderType、pay、stock 这 3 个字段当作参数传递给 500 元订单函数，如果该函数不符合处理条件，则把这个请求传递给后面的 200 元订单函数，如果 200 元订单函数依然不能处理该请求，则继续传递请求给普通购买函数。

```
var order500 = function( orderType, pay, stock ){
    if ( orderType === 1 && pay === true ){
        console.log( '500 元定金预购，得到 100 优惠券' );
    } else{
        return 'nextSuccessor'; // 我不知道下一个节点是谁，反正把请求往后面传递
    }
};
var order200 = function( orderType, pay, stock ){
    if ( orderType === 2 && pay === true ){
        console.log( '200 元定金预购，得到 50 优惠券' );
    } else{
        return 'nextSuccessor'; // 我不知道下一个节点是谁，反正把请求往后面传递
    }
};
var orderNormal = function( orderType, pay, stock ){
    if ( stock > 0 ){
        console.log( '普通购买，无优惠券' );
    } else{
        console.log( '手机库存不足' );
    }
};

// Chain.prototype.setNextSuccessor 指定在链中的下一个节点
// Chain.prototype.passRequest 传递请求给某个节点
var Chain = function( fn ){
    this.fn = fn;
    this.successor = null;
};
Chain.prototype.setNextSuccessor = function( successor ){
    return this.successor = successor;
};
Chain.prototype.passRequest = function(){
    var ret = this.fn.apply( this, arguments );
    if ( ret === 'nextSuccessor' ){
        return this.successor && this.successor.passRequest.apply(
this.successor, arguments );
    }
    return ret;
};
var chainOrder500 = new Chain( order500 );
var chainOrder200 = new Chain( order200 );
var chainOrderNormal = new Chain( orderNormal );

chainOrder500.setNextSuccessor( chainOrder200 );
chainOrder200.setNextSuccessor( chainOrderNormal);

chainOrder500.passRequest( 1, true, 500 );   // 输出:500 元定金预购，得到 100 优惠
券
chainOrder500.passRequest( 2, true, 500 );   // 输出:200 元定金预购，得到 50 优惠券
```

```
chainOrder500.passRequest( 3, true, 500 );    // 输出:普通购买，无优惠券
chainOrder500.passRequest( 1, false, 0 );     // 输出:手机库存不足
```

通过改进，我们可以自由灵活地增加、移除和修改链中的节点顺序，假如某天网站运营人员 又想出了支持 300 元定金购买，那我们就在该链中增加一个节点即可

```
var order300 = function(){
 // 具体实现略
};
chainOrder300= new Chain( order300 );
chainOrder500.setNextSuccessor( chainOrder300);
chainOrder300.setNextSuccessor( chainOrder200);
```

适配器模式

> 适配器模式的作用是解决两个软件实体间的接口不兼容的问题。使用适配器模式之后，原本 由于接口不兼容而不能工作的两个软件实体可以一起工作。 适配器的别名是包装器(wrapper)，这是一个相对简单的模式。在程序开发中有许多这样的 场景:当我们试图调用模块或者对象的某个接口时，却发现这个接口的格式并不符合目前的需求。 这时候有两种解决办法，第一种是修改原来的接口实现，但如果原来的模块很复杂，或者我们拿 到的模块是一段别人编写的经过压缩的代码，修改原接口就显得不太现实了。第二种办法是创建 一个适配器，将原接口转换为客户希望的另一个接口，客户只需要和适配器打交道。

```
var googleMap = {
    show: function(){
        console.log( '开始渲染谷歌地图' );
    }
};
var baiduMap = {
    display: function(){
        console.log( '开始渲染百度地图' );
    }
};
var baiduMapAdapter = {
    show: function(){
        return baiduMap.display();
    }
};
renderMap( googleMap ); // 输出:开始渲染谷歌地图
renderMap( baiduMapAdapter ); // 输出:开始渲染百度地图
```

适配器模式主要用来解决两个已有接口之间不匹配的问题，它不考虑这些接口是怎样实 现的，也不考虑它们将来可能会如何演化。适配器模式不需要改变已有的接口，就能够 使它们协同作用。

装饰者模式和代理模式也不会改变原有对象的接口，但装饰者模式的作用是为了给对象 增加功能。装饰者模式常常形成一条长的装饰链，而适配器模式通常只包装一次。代理 模式是为了控制对对象的访问，通常也只包装一次。

我们设计很多插件，有默认值，也算是适配器的一种应用， vue的prop校验，default也算是适配器的应用了

外观模式的作用倒是和适配器比较相似，有人把外观模式看成一组对象的适配器，但外观模式最显著的特点是定义了一个新的接口。

模板方法模式

> 模板方法模式在一个方法中定义一个算法的骨架，而将一些步骤的实现延迟到子类中。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法中某些步骤的具体实现

这个我们用的很多，vue中的slot，react中的children

```
class Parent {
  constructor() {}
  render () {
    <div>
      <div name="tom"></div>
      <!-- 算法过程：children要渲染在name为joe的div中 -->
      <div name="joe">{this.props.children}</div>
    </div>
  }
}
class Stage {
  constructor() {}
  render () {
    // 在parent中已经设定了children的渲染位置算法
    <Parent>
      // children的具体实现
      <div>child</div>
    </Parent>
  }
}
```

```
<template>
```

```
  <div>
    <div name="tom"></div>
    <div name="joe">
      <!--vue中的插槽渲染children-->
      <slot />
    </div>
  </div>
</template>


<template>
  <div>
    <parent>
      <!-- children的具体实现 -->
      <div>child</div>
    </parent>
  </div>
</template>
```

备忘录模式

可以恢复到对象之前的某个状态，其实大家学习react或者redux的时候，时间旅行的功能，就算是备忘录模式的一个应用

https://zh-hans.reactjs.org/tutorial/tutorial.html#implementing-time-travel

5. 扩展

6. 总结

创建设计模式： 工厂，单例，建造者 原型

结构化设计模式：外观，适配器，代理，装饰器，享元 桥接，组合

行为型：策略，模板方法，观察者，迭代器，责任链，命令，备忘录，状态，访问者，终结者，解释器