什么是设计模式

模式是一种可复用的解决方案，用于解决软件设计中遇到的常见问题

换个通俗的说法：设计模式是解决某个特定场景下对各种问题的解决方案，因此，当我们遇到合适的场景，我们可能会条件反射一样自然而然想到符合这种场景的设计模式

在将函数作为一等公民的对象语言中，有许多需要利用对象多态性的设计模式，比如命令模式、策略模式等，这些模式的结构与传统面向对象语言的结构大相径庭；

单例模式

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。
单例模式的核心是 确保只有一个实例，并提供全局访问
实现的方法为用一个变量先判断实例存在与否，如果存在直接返回，如果不存在就创建了再返会，这就确保了一个类只有一个实例对象

- 试用场景：一个单一对象。比如：登录弹窗，无论点击多少次，弹窗只应该被创建一次
  用代理来实现单例模式

```
class CreateUser {
  constructor(name) {
    this.name = name;
    this.getName();
  }
  getName() {
    return this.name;
  }
}
var ProxyMode = (function() {
  var instance;
  return function(name) {
    if (!instance) {
      instance = new CreateUser(name);
    }
    return instance;
  }
})();
// 测试单体模式的实例
var a = new ProxyMode('aaa');
var b = new ProxyMode('bbb');
console.log(a === b); // 因为单体模式是只实例化一次，所以下面的实例是相等的
```

- 惰性单例
  惰性单例指的是在需要的时候才创建对象实例

```
var getSingle = function(fn) {
  var result;
  return function() {
    return result || (result = fn.apply(this, argeuments));
  }
}
var createLoginLayer = function() {
  var div = document.createElement('div');
  div.innerHtml = '我是登录浮窗';
  div.style.display = 'none';
  document.body.appendChild(div);
  return div;
}
document.getElementById('loginBtn').onclick = function() {
  var loginLayer = createLoginLayer();
  loginLayer.style.display = 'block';
}
var createSingleIframe = function() {
  var iframe = document.createElement('iframe');
  document.body.appendChild(iframe);
  return iframe;
}
document.getElementById('loginBtn').onclick = function() {
  var loginLayer = createSingleIframe();
  loginLayer.style.display = 'block';
}
```

策略模式

定义一系列的算法，把它们一个个封装起来，并且使它们可以相互替换。

- 一个基于策略模式的程序至少由两部分组成：
  - 一组策略类（可变），封装了具体算法，并负责具体的计算过程
  - 环境类 （Context 不变）context接受客户的请求，随后将请求委托给一个策略类
- 什么时候使用
  - 各判断条件下的策略相互独立且可复用
  - 策略内部逻辑相对复杂
  - 策略需要灵活组合

```
/* 策略类 */
var levelOBJ = {
  "A": function(money) {
    return money * 4;
  },
  "B": function(money) {
    return money * 3;
  },
  "C": function(money) {
    return money * 2;
  },
}
/*环境类*/
var calculateBouns = function(level, money) {
  return levelOBJ[level](money);
}
console.log(calculateBouns('A',10000)); // 40000
console.log(calculateBouns('B',10000)); // 30000
```

用策略模式来实现表单校验

```
var strategies = {
  isNotEmpty: function(value, errMsg) {
    if (value === '') {
      return errMsg;
    }
  },
  minLength: function(value, length, errMsg) {
    if (value.length < length) {
      return errMsg;
    }
  }
}
var Validator = function() {
  this.cache = []; // 保存校验规则
}
Validator.prototype.add = function(dom, rule, errMsg) {
  var ary = rule.split(':'); // 把strategy和参数分开
  this.cache.push(function() {
    var strategy = ary.shift();  // 用户的strategy
    ary.unshift(dom.value); // 把input的value添加进参数列表
    ary.push(errMsg); // 把errmsg添加进参数列表
    return strategies[strategy].apply(dom, ary);
  })
}
Validator.prototype.start = function() {
  for(var i = 0, vaidatorFunc; vaidatorFunc = this.cache[i++]; ) {
    var msg = vaidatorFunc();
    if (msg) {
      return msg;
    }
  }
}
var registerForm = document.getElementById('registerForm');
var validator = new Validator();
validator.add(registerForm.userName, 'isNotEmpty', '用户名不能为空');
validator.add(registerForm.userName, 'minLength: 10', '用户名长度不能小于10位');
```

代理模式

所谓的代理模式就是为一个对象提供一个代用品或者占位符，以便控制对它的访问；

代理形式

- 保护代理
  - 用于控制不同权限的对象对目标对象的访问，但在javascript中并不容易实现保护代理，因为我们无法判断谁访问了某个对象
- 虚拟代理

- 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真正需要它的时候才去创建
- 图片预加载

```
const myImage = (function () {
  const imageNode = document.createElement('img');
  document.body.appendChild(imageNode);
  return {
    setSrc: function (src) {
      iamgeNode.src = src;
    }
  }
})();
const proxyImage = (function () {
  const image = new Image;
  image.onload = function () {
    myImage.setSrc(this.src);
  }
  return {
    setSrc: function(src) {
      myImage.setSrc('./loading.jpg');
      img.src = src;
    }
  }
})()
proxyImage.setSrc('http://loaded.jpg');
```

- 缓存代理
  - 缓存代理实现计算乘积或者加和

```
const mult = function() {
  let a = 1;
  for(let i = 0, l; l = arguments[i++];) {
    a = a * l;
  }
  return a;
}
const plus = function() {
  let a = 1;
  for(let i = 0, l; l = arguments[i++];) {
    a = a + l;
  }
  return a;
}
const createProxyFactory = function(fn) {
  const cache = {};
  return function() {
    const tag = Array.prototype.join.call(arguments, ',');
    if (cache[tag]) {
      return cache[tag]
    }
    cache[tag] = fn.apply(this, arguments);
    return cache[tag];
  }
};
const proxyMult = createProxyFactory(mult)
const proxyPlus = createProxyFactory(plus)
console.log(proxyMult(1, 2, 3, 4)) // 24
console.log(proxyMult(1, 2, 3, 4)) // 24
console.log(proxyPlus(1, 2, 3, 4)) // 10
```

- 代理和被代理对象的一致性
- 其他代理模式
  - 防火墙代理：控制网络资源的访问，保护主机不让‘坏人’接近
  - 远程代理：为一个对象在不同的地址空间提供局部代表，在java中，远程代理可以是另一个虚拟机中的对象
  - 保护代理：用于对象应该有不同访问权限的情况
  - 智能引用代理：取代了简单的指针，它在访问对象时执行一些附加操作，比如计算一个对象被引用的次数
  - 写时复制代理：通常用于复制一个庞大对象的情况

迭代器模式

迭代器模式是指提供一种方法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，也可以按照顺序访问其中的每个元素；

```
// 实现自己的迭代器
var each = function(ary, callback) {
  for(var i = 0; i < ary.length; i++) {
    callback.call(ary[i], i, ary[i]);
  }
}
each([1, 2, 3, 4], function(i, n) {
  console.log([i, n])
})
```

发布-订阅模式

事件发布/订阅模式（PubSub）在异步编程中帮助我们完成更松的解耦，甚至在MVC，MVVM的架构中以及设计模式中也少不了发布-订阅模式的参与

优点：在异步编程中实现更深的解耦

缺点：如果过多的使用发布订阅模式，会增加维护的难度

实现node的EventEmitter

```
function EventEmitter() {
  this._events = Object.create(null);
}
//默认最大监听数
EventEmitter.defaultMaxListeners = 10;
// on方法，该方法用于订阅事件，在旧版本的node.js中是addListener方法，它们是同一个函数
// flag标记是一个订阅方法的插入标识，如果为'true'就视为插入在数组的头部
EventEmitter.prototype.on = EventEmitter.prototype.addListener = function(type, listener, flag)
  if (!this._events) {
    this._events = Object.create(null);
  }
  if (this._event[type]) {
    if (flag) {
      this._events[type].unshift(listener);
    } else {
      this._events[type].push(listener);
    }
  } else {
    this._events[type] = [listener];
  }
  //绑定事件，触发newListener  不是newListener 就应该让newListener执行以下
  if (type !== 'newListener') {
    this.emit('newListener', type);
  }
}
// emit方法就是将订阅方法取出执行，使用call方法来修正this的指向，使其指向子类的实例。
EventEmitter.prototype.emit = function (type, ...args) {
  if (this._events[type]) {
    this._events[type].forEach(fn => fn.call(this, ...args))
  }
}
// once方法非常有趣，它的功能是将事件订阅“一次”，当这个事件触发过就不会再次触发了。其原理是将订阅的方法再
EventEmitter.prototype.once = function (type, listener) {
  let _this = this;
  // 中间函数，在调用完之后立即删除订阅
  function only() {
    listener();
    _this.removeListener(type, only);
  }
  // origin保存原回调的引用，用于remove时的判断
  only.origin = listener;
  this.on(type, only);
}
// off方法即为退订，原理同观察者模式一样，将订阅方法从数组中移除即可。
EventEmitter.prototype.off = EventEmitter.prototype.removeListener = function(type, listener) {
  if (this._enents[type]) {
    this._events[type] = this._events[type].filter(fn => {
      return fn !== listener && fn.origin !== listener
    })
  }
```

```
}
EventEmitter.prototype.removeAllListener = function () {
  this._events = Object.create(null);
}
// 此方法，调用on方法将标记传为true（插入订阅方法在头部）即可。
EventEmitter.prototype.prependListener = function (type, listener) {
  this.on(type, listener, true);
};

module.exports = EventEmitter;
```

命令模式

组合模式

模板方法模式

模板方法模式是一种只需要使用继承就可以实现的非常简单的模式

模板方法模式由两部分组成

- 抽象父类（通常在父类封装了子类的算法框架，包括实现一些公共方法以及封装子类中所有方法的执行顺序）
- 具体的实现子类 （子类通过继承这个抽象父类，也继承了整个算法结构，并且可以选择重写父类的方法）

例子
Coffee or Tea

```
// 首先分离公共点，即泡茶和咖啡的共同点；经过抽象，这两者都能整理为下面四步
// - 把水煮沸
// - 用沸水冲泡饮料
// - 把饮料倒进杯子
// - 加调料
const Beverage = function() {}
Beverage.prototype.boilWater = function() {
  console.log('把水煮沸')
}
Beverage.prototype.brew = function() {} // 空方法，应该由子类重写
Beverage.prototype.pourInput = function() {} // 空方法，应该由子类重写
Beverage.prototype.addCondiments = function() {} // 空方法，应该由子类重写
Beverage.prototype.customerWantsCondients = function() { // hook方法，用来隔离变化
  return true; // 默认需要调料
}

Beverage.prototype.init = function() {
  this.boilWater();
  this.brew();
  this.pourInput();
  if (this.customerWantsCondients()) { // 如果挂钩返回true，则需要调料
    this.addCondiments();
  }

}

// 创建Coffee和Tea子类
const Coffee = function() {};
Coffee.prototype = new Beverage();
Coffee.prototype.brew = function() {console.log('用沸水冲泡咖啡')}
Coffee.prototype.pourInput = function() {console.log('把咖啡倒进杯子')}
Coffee.prototype.addCondiments = function() {console.log('加糖和牛奶')}
Coffee.prototype.customerWantsCondients = function() {
  return window.confirm('请问需要调料吗?')
}
const coffee = new Coffee();
coffee.init();

const Tea = function() {};
Tea.prototype = new Beverage();
Tea.prototype.brew = function() {console.log('用沸水浸泡茶叶')}
Tea.prototype.pourInput = function() {console.log('把茶倒进杯子')}
Tea.prototype.addCondiments = function() {console.log('加柠檬')}
const tea = new Tea();
tea.init();

// 到底谁是模板方法呢，答案是 Beverage.prototype.init
```

享元模式（flyweight）

享元模式是一种用于性能优化的模式，核心是运用共享技术来有效的支持大量细粒度的对象

享元模式要求将对象的属性划分为内部状态和外部状态（状态在这里通常指属性），享元模式的目标是尽量减少共享对象的数量

如何划分内部状态和外部状态

- 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，通常不会变化
- 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

享元模式适用性

- 一个程序中使用了大量的相思对象
- 由于使用了大量对象，造成了很大的内存开销
- 对象的大多数状态都可以变为外部状态
- 剥离出对象的外部状态之后，可以用相对较少的共享对象取代大量对象

举个例子

- 文件上传

```
var id = 0;
// 触发上传动作的 startUpload 函数
window.startUpload = function(uploadType, files) {
  for(var i = 0, file; file=files[i++]; ) {
    var uploadObj = uploadManager.add(++id, uploadType, file.fileName, file.fileSize);
  }
}
var Upload = function(uploadType) {
  this.uploadType = uploadType;
}
Upload.prototype.delFile = function(id) {
  uploadManger.setExternalState(id, this);
  if (this.fileSize < 3000) {
    return this.dom.parentNode.removeChild(this.dom);
  }
  if (window.confirm('确定要删除文件吗? ' + this.fileName)) {
    return this.dom.parentNode.removeChild(this.dom);
  }
}
// 工厂进行对象实例化
var UploadFactory = (function() {
  var createdFlyWeightObjs = {};
  return {
    create: function(uploadType) {
      if (createdFlyWeightObjs[uploadType]) {
        return createdFlyWeightObjs[uploadType];
      }
      return createdFlyWeightObjs[uploadType] = new Upload(uploadType);
    }
  }
})()
// 管理器封装外部状态，负责向 UploadFactory提交创建对象的请求，并用一个uploadDatabase对象保存所有uploa
var uploadManger = (function() {
  var uploadDatabase = {};
  return {
    add: function(id, uplaodType, fileName, fileSize) {
      var flyWeightObj = UploadFactory.create(uploadType);

      var dom = document.createElement('div');
      dom.innerHtml = `<span>文件名称: ${fileName}, 文件大小: ${fileSize}</span><button class="del
      dom.querySelector('.delFile').onclick = function() {
        flyWeightObj.delFile(id);
      }

      document.body.appendChild(dom);

      uploadDatabase[id] = {
        fileName,
        fileSize,
        dom
```

```
      }

      return flyWeightObj
    },
    setExternalState: function(id, flyWeightObj) {
      var uploadData = uploadDatabase[id];
      for(var i in uploadData) {
        flyWeightObj[i] = uploadDate[i];
      }
    }
  }
})()
```

责任链模式

责任链模式的定义是：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受者之间的耦合关系，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；
类似于多米诺骨牌，请求第一个条件，会持续执行后续的条件，知道返回结果为止；

```
// 场景：某电商针对已付过定金的用户有优惠政策，在正式购买后，已经支付过 500 元定金的用户会收到 100 元的

// orderType：表示订单类型，1：500 元定金用户；2：200 元定金用户；3：普通购买用户
// pay：表示用户是否已经支付定金，true：已支付；false：未支付
// stock：表示当前用于普通购买的手机库存数量，已支付过定金的用户不受此限制
const order500 = function (orderType, pay, stock) {
  if (orderType === 1 && pay === true) {
    console.log('500 元定金预购，得到 100 元优惠券');
  } else {
    return 'nextSuccess';
  }
}
const order200 = function (orderType, pay, stock) {
  if (orderType === 2 && pay === true) {
    console.log('200 元定金预购，得到 100 元优惠券');
  } else {
    return 'nextSuccess';
  }
}
const orderCommon = function (orderType, pay, stock) {
  if ((orderType === 3 || !pay) && stock > 0) {
    console.log('普通购买，无优惠券')
  } else {
    console.log('库存不够，无法购买')
  }
}
Function.prototype.after = function (fn) {
  const self = this;
  return function() {
    const result = self.apply(self, arguments)
    if (result === 'nextSuccess') {
      return fn.apply(self, arguments) // 这里 return 别忘记了~
    }
  }
}
const order = order500.after(order200).after(orderCommon)

order(3, true, 500) // 普通购买，无优惠券
```

```
const [ jack, lucy, lili ] = [
  {
    name: 'Jack',
    requirement: '我要玩游戏'
  },
  {
    name: 'Lucy',
    requirement: '我要和4个人一起玩游戏'
  },
  {
    name: 'Lili',
    requirement: '我要和男盆友玩游戏'
  }
]
const isSatisfyJack = (user) => {
  // do something
}
const isSatisfyLucy = (user) => {
  // do something
}
const isSatisfyLili = (user) => {
  // do something
}
function playGame() {
  if (isSatisfyJack()) {
    console.log(`我可以和 Jack 一起玩游戏`)
  } else if (isSatisfyLucy()) {
    console.log(`我可以和 Lucy 一起玩游戏`)
  } else if (isSatisfyLili()) {
    console.log(`我可以和 Lili 一起玩游戏`)
  } else {
    console.log(`哎呀，我要一个人玩游戏`)
  }
}
// 假如又多了个朋友，我必须再修改playGame方法，这个其实是违反开放-封闭原则，不易于代码的维护

// 优化后的代码
// 给每个人定义一个职责
const Chain = function(fn) {
  this.fn = fn
  this.successor = null
}
Chain.prototype.setNextSuccessor = function(successor) {
  return this.successor = successor
}
Chain.prototype.passRequest = function() {
  const ret = this.fn.apply(this, arguments)
  if (ret === 'nextSuccessor') {
    return this.successor && this.successor.passRequest.apply(this.successor, arguments)
  }
```

```
  return ret
}
const chainOrderA = new Chain(isSatisfyJack)
const chainOrderB = new Chain(isSatisfyLucy)
const chainOrderC = new Chain(isSatisfyLili)

// 设置一下职责链的顺序
chainOrderA.setNextSuccessor(chainOrderB)
chainOrderB.setNextSuccessor(chainOrderC)

function playGameOptimization() {
  chainOrder.passRequest() // 发起请求
}
```

通过职责链模式，解耦了请求发送者和多个接收者之间的复杂关系，不再需要知道具体哪个接收者来接收发送的请求，只需要向职责链的第一个阶段发起请求
责任链的优缺点

- 最大优点就是解耦了请求发送者和N个接收者之间的复杂关系
- 使用了责任链模式之后，链中的节点对象可以灵活的拆分重组
- 可以手动指定起始节点
- 缺点 不能保证某个请求一定会被链中的节点处理，这种情况下，我们可以在链尾增加一个保底的接收者来处理这种即将离开链尾的请求
- 另外，责任链模式使得程序中多了一些节点对象，可能在某次请求传递过程中，大部分节点并没有起到实质性的作用，他们的作用仅仅是让请求传递下去，从性能方面考虑，我们要避免过长的责任链带来的性能耗损；

中介者模式

中介者模式的作用就是解除对象和对象之间的紧耦合关系，增加一个中介者对象之后，所有的相关对象都通过中介者对象来通信

中介者模式使得网状的多对多关系变成了相对简单的一对多关系

现实中的中介者

- 机场指挥塔
- 博彩公司

举个例子

- 泡泡堂游戏
- 聊天室

```
// 公共类
function Mediator() {
  const users = []
  return {
    addUser(user) {
      users.push(user)
    },
    publishMessage(msg, recevier) {
      if (recevier) {
        recevier.messages.push(msg)
      } else {
        users.forEach(user => {
          user.messages.push(msg)
        })
      }
    }
  }
}
let mediator = Mediator()
// 成员类
function User(name) {
  this.name = name
  this.messages = []
  mediator.addUser(this)
}
User.prototype.sendMessage = function(msg, receiver) {
  // msg = '[' + this.name + ']: ' + msg
  msg = `[${this.name}]: ${msg}`
        mediator.publishMessage(msg, receiver)
}
let u1 = new User('Jack')
let u2 = new User('Peter')
let u3 = new User('Anna')
u1.sendMessage('Hi, anybody here?')
u2.sendMessage('Hi Jack, nice to meet you.', u1)
u3.sendMessage('Hi Guys!')
```

装饰器模式

给一个函数赋能，增强它的某种能力，它能动态的添加对象的行为（动态地给函数赋能）
原有方法维持不变，在原方法上挂载其它方法满足现有需求，实现同样的效果但增强了复用性

```
// AOP 装饰函数实现
Function.prototype.before = function (fn) {
  const self = this; // 保存原函数引用
  return function () { // 返回包含了原函数和新函数的 '代理函数'
    fn.apply(this, arguments); // 执行新函数，修正this
    return fn.apply(this, arguments) // 执行原函数
  }
}
Function.prototype.after = function (fn) {
  const self = this;
  return function () {
    const ret = self.apply(this, arguments)
    fn.apply(this, arguments);
    return ret
  }
}
const func = function () {
  console.log('func')
}
const func1 = function () {
  console.log('func1')
}
const func2 = function () {
  console.log('func2')
}
func = func.before(func1).after(func2)
func()
```

观察者模式

- 场景一 当观察的数据对象发生变化时，自动调用相应的函数。比如vue的双向绑定

```
var obj = {
  data: { list: [] }
}
Object.defineProperty(obj, 'list', {
  get() {
    return this.data['list'];
  },
  set(val) {
    console.log('值被更改了');
    this.data['list'] = val;
  }
})
```

```
// Proxy/Reflect 是 ES6 引入的新特性，也可以使用其完成观察者模式，示例如下(效果同上):
var obj = {
  value: 0
}
var proxy = new Proxy(obj, {
  set: function(target, key, value, receiver) {
    console.log('调用相应函数')
    Reflect.set(target, key, value, receiver)
  }
})
obj.value = 1; // 调用相应的函数
```

- 场景二 当调用对象里的某个方法时，就回调用相应的访问逻辑。比如给测试框架赋能的spy函数

```
// 下面来实现 sinon 框架的 spy 函数
const sinon = {
  analyze: {},
  spy: function (obj, fnName) {
    const that = this;
    const oldFn = Object.getOwnPropertyDescriptor(obj, fnName).value;
    Object.defineProperty(obj, fnName, {
      value: function() {
        oldFn()
        if (that.analyze[fnName]) {
          that.analyze[fnName].count = ++that.analyze[fnName].count
        } else {
          that.analyze[fnName] = {}
          that.analyze[fnName].count = 1
        }
        console.log(`${fnName} 被调用了 ${that.analyze[fnName].count} 次`)
      }
    })
  }
}
const obj = {
  someFn: function() {
    console.log('my name is someFn')
  }
}
sinon.spy(obj, 'someFn')
obj.someFn()
// my name is someFn
// someFn 被调用了 1 次
obj.someFn()
// my name is someFn
// someFn 被调用了 2 次
```

适配器模式

采用适配器模式，将不同的数据结构适配成展示组件所能接受的数据结构（别名包装器 wrapper）

主要用于解决两个软件实体间接口之间不兼容的问题

```
// 老接口
const getCityOld = (function () {
  return [
    {
      name: 'hangzhou',
      id: 1,
    },
    {
      name: 'jinhua',
      id: 2
    }
  ]
})()
// 新接口希望是下面形式 { hangzhou: 1, jinhua: 2 }
const cityAdaptor = (function() {
  const obj = {};
  for(let city of getCityOld()) {
    obj[city.name] = city.id
  }
  return obj;
})()


interface UploadFileType {
  uuid: string;
  name: string;
  size: number;
  type: string;
  status: string;
}
```

```
// 目标角色
class Target {
  request() {
    return 'Target: The default target\'s behavior.'
  }
}
// 源角色
class Adaptee {
  specialRequest() {
    return 'Adaptee: The default Adaptee\'s behavior. '
  }
}
// 适配器实现
class Adapter extends Target {
  constructor(adaptee) {
    super()
    this.adaptee = adaptee
  }
  request() {
    this.adaptee.specialRequest()
  }
}
let adaptee=new Adaptee();
let adapter=new Adapter(adaptee);
adapter.request();
```

- 适配器模式主要是用来解决两个已有接口不匹配的问题，它不考虑这些接口是怎么实现的，也不考虑他们将来如何变化。适配器模式不需要改变已有的接口，就能够使它们协同工作
- 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，装饰者模式经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象的访问，通常也只包装一次
- 外观模式的作用倒是和适配器模式有点相似，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

总结各设计模式的关键词

设计模式	特点	案例
单例模式	一个类只能构造出唯一实例	弹框层的实践，登录弹窗
策略模式	根据不同参数可以命中不同的策略	动画库里的算法函数
代理模式	代理对象和本体对象具有一致的接口	图片预加载

迭代器模式	能获取聚合对象的顺序和元素	实现迭代器each（forEach）
发布-订阅模式	PubSub	node的EventEmiter
职责链模式	链式执行后续的条件，直到返回结果为止	if else 优化
装饰者模式	动态地给函数赋能	
适配者模式	一种数据结构改成另一种数据结构	枚举值接口变更
观察者模式	当观察对象发生变化时自动调用相关函数	vue 双向绑定

三个设计原则

- 单一职责原则（SRP）
  - 体现为一个对象（方法）只做一件事情
  - 运用： 代理模式、迭代器模式、单例模式和装饰者模式
  - 优缺点
    - 优点是降低了单个类或者对象的复杂度，按照职责把对象分解成更小的粒度，有助于代码的复用，也有利于单元测试。当一个职责变更的时候，不会影响到其他的职责
    - 缺点是增加编写代码的复杂度，当我们按照职责把对象分解成更小的粒度之后，实际上也增大了这些对象之间相互联系的难度
- 最少知识原则（LKP）
  - 说的是一个软件实体应当尽可能少的与其他实体发生相互作用，最少知识原则要求我们设计程序时，应当尽量减少对象之间的交互
  - 体现最多的模式有：中介者模式、外观模式
- 开放-封闭原则
  - 思想：当需要改变一个程序的功能或者给这个程序增加新功能的时候，可以使用增加代码的方式，但是不允许改动程序的源代码
  - 实践方法
    - 用对象的多态性消除条件分支
    - 找出变化的地方，找出程序中将要发生变化的地方，把变化封装起来，稳定不变的部分和容易变化的部分隔离开来，在系统的演变过程中，我们只需要替换那些容易变化的部分，变化的部分使用如下方法
      - 放置挂钩
      - 使用回调函数

代码重构技巧

提炼函数

合并重复的条件片段

把条件分支语句提炼成函数

合理使用循环

提前让函数退出代替嵌套条件分支

传递对象参数替代过长的参数列表

尽量减少参数数量

合理使用链式调用

分解大类型

用return退出多重循环