提高代码质量的目的

首先高质量的代码，是一切性能的基础,无论是我们的可扩展性，复用性，必须建立在高代码的基础上，方便后续的一切操作，高质量代码更方便他人阅读。

什么是代码质量?

代码质量的体现有哪些?

1. 代码整洁 - 从命名 缩进 各个方面符合规范，比较好看，没有多余，累赘的代码
2. 结构比较规整，没有漫长的结构 - 没有一堆长长的if else判断和长长的赋值语句
3. 阅读好理解-命名符合语义化，比如一个变量的命令符合这个变量的语义，一个方法的命名要符合方法的作用

如何提高代码质量?

- **优化你的代码结构**

如何优化代码结构，我们可以采用以下几种模式

1. 策略模式/状态模式: 策略模式和状态模式放在一起，因为两者非常相似，他们使用目的是优化if-else 分支，有的时候我们的if-else过长,这样显得我们的代码过于丑陋，这个时候我就可以使用策略模式或者状态模式
2. 外观模式: 通过为多个复杂的子系统提供一个一致的接口，当我们完成一个功能需要调用很多子系统接口的时候，就要想到外观模式的思想，我们把复杂的子接口统一成一个更高级的接口，统一的调用完成这样的一个功能。
3. 迭代器模式:我们在编程的时候，不可避免的要对对象和数组进行操作,迭代器模式的目的，不访问内部的情况下，方便的遍历数据。而迭代器的应用场景是 当我们要对某个对象进行操作的时候，但是又不能暴露内部。或者说，为了我们操作数组或者对象，新建一个迭代器来帮助我们访问这个数组或者对象。
4. 备忘录模式：目的是，记录状态，方便回滚，他的应用场景 系统状态多样，为了保证状态的回滚方便，记录状态

- **四种模式的基本结构**

策略模式

策略模式能够优化我们的if-else分支，例如我们需要编写一个计算器，有加减乘除，我们可以把一层一层的if判断，变成这样的形式

```
function Strategy(type,a,b){
    var Strategyer={
        add:function(a,b){
            return a+b;
        },
        minus:function(a,b){
            return a-b;
        },
        division:function(a,b){
            return a/b;
        }
    }
    return Strategyer[type](a,b);
}
```

我们把加 减 乘 除的相关方法 变成策略对象里面的属性，不再通过大量的if-else 去一层层的判断，然后执行策略对象属性中的方法即可

状态模式

状态模式其实也是优化我们的if-else分支，状态模式，我们可以看成一个加了状态管理的策略模式，为了减少if-else结构，将判断成对象内部的一个状态，通过对象内部的状态改变，让其拥有不同的行为。

```
function stateFactor(state){
    var stateObject={
      _status:"",
      state:{
        state1:function(){

        },
        state2:function(){

        },
      },
      run:function(){
        return this.state[this._status]();
      }
    }
    stateObject._status=state;
    return stateObject;
}
stateFactor('state1').run();
```

我们看上图的示例代码:我们新建了一个状态工厂函数，工厂函数对象里面有一个状态对象，状态对象里面有我们上面讲的策略对象，策略对象里面还有一个状态属性_status，还有一个run运行的方法。在使用的时候，我只需要改变其状态_status，然后返回其策略对象，最后调用工厂函数对应状态的run方法，执行不同的操作。

外观模式

我们在组织方法模块时可以细化多个接口，但是我们给别人使用时,要合为一个接口，就像你可以直接去餐厅点餐，为了让用户有更高的可选性，餐厅有很多菜，但是为了方便一些困难选择症的顾客推荐一些套餐，里面包含哪些菜。就像我们外观模式一样，我们可以把我们的接口，细化成一个个的小接口，但是最终的功能会由一个统一完整的接口来完成我们整个功能的调用

```
//模块1
function Model1(){

}
//模块2
function Model2(){

}
//功能由Model1获取Model2得结果来完成
function use(){
  Model2(Model1());
}
```

迭代器模式

我们把数组或者对象变成一个迭代器对象，我们有一个方法，不去遍历对象或者数组的情况下，可以顺序的访问对象内部，可以帮助我们简化循环。如图所示

```
function Iterator (item){
    this.item=item;
}

Iterator.prototype.dealEach=function(fn){

   for(var i=0;i<this.item.length;i++){
      fn(this.item[i],i);
   }

}
```

我们新建了一个迭代器类，这个迭代器类会接收传进来的数组或者对象，然后再迭代器类的原型上面添加一个dealEeach方法，我们可以通过这个dealEeach方法 ，不要我们手动进行for循环，来对数组或者对象中的每一项进行方法操作。

备忘录模式

记录对象内部的状态，当有需要的时候回滚到之前的状态或者方便对象使用。如下图所示

我们会有一个缓存对象 cachel 来缓存我们的状态，然后这个MemenTo会返回一个方法，这个方法可以访问到我们cache对象，其实就是闭包，在这个方法内部，判断是否有这样一个cache[cacheName],如果有，就执行有缓存操作，没有缓存就执行没有缓存操作。

- **应用示例**

策略和状态模式示例

需求一: 项目有一个动态的内容,根据用户权限的不同显示不同的内容，以下是未使用策略或者状态模式的示例

```
function showPart1(){
    console.log(1);
}
function showPart2(){
    console.log(2);
}
function showPart3(){
    console.log(3);
}
axios.get('xxx').then((res)=>{
    if(res=='boss'){
        showPart1();
        showPart2();
        showPart3();
    }else if(res=='manner'){
        showPart1();
        showPart2();
    }else if(res=='staff'){
        showPart3();
    }
})
```

此代码中，有三个方法，根据权限api请求，不同的角色，有不同的权限。存在的问题：有这么多的if-else分支不太好看，如果有更多的角色，就会有更多的if-else分支。如何改造呢?

1. 新建一个构造函数，里面包含状态和一个策略对象

```
function showControll(){
    this.status='';
    this.power={
      boss:function(){
        showPart1();
        showPart2();
        showPart3();
      },
      manner:function(){
        showPart1();
        showPart2();
      },
      staff:function(){
        showPart3();
      }
    }
}
```

2.在showControll构造函数的原型上添加一个show方法，里面一个权限api请求，根据不同角色，显示不同的权限。

```
showControll.prototype.show=function(){
    var self=this;
    axios.get('xxx').then((res)=>{
        self.status=res;
        self.power[self.status]();
    })
}
```

3.实例化 showControll构造函数，并调用show方法，通过不同的角色，展示不同的权限，拥有不同的行为。

```
new showControll().show();
```

通过以上改造有哪些优势呢：

1. 代码更加优雅，去除大量的if-else分支
2. 如果有更多的角色，我们只需要在prower这个策略对象里面添加即可，无需再扩展if-else分支，这样就更加方便

需求二：有一个小球，可以控制它左右移动，或者左前，右前等方向移动，我们先看一个反面示例

```
function moveLeft(){
  console.log('left')
}
function moveRight(){
  console.log('RigmoveRight')
}
function moveTop(){
  console.log('Top')
}
function moveBottom(){
  console.log('bomoveBottom')
}
```

```
function mover(){
    if(arguments.length==1){
        if(arguments[0]=='left'){
          moveLeft();
        }else if(arguments[0]=='right'){
          moveRight();
        }else if(arguments[0]=='top'){
          moveTop();
        }else if(arguments[0]=='bottom'){
          moveBottom();
        }
    }else{
        if(arguments[0]=='left'&&arguments[1]=='top'){
          moveLeft();
          moveTop();
        }else if(arguments[0]=='right'&&arguments[1]=='bottom')
          moveRight();
          moveBottom();
        }  两两组合的，会有多种情况，大量的
        //.if-else代码块，及其恶心
    }
}
```

当mover函数只有一个参数，只是一个方向移动的话，if-else判断分支不多，如果有多个参数，向多个方向移动的话，if-else分支块会非常多，代码及其丑陋。阅读起来非常糟心。

用状态模式改造上述代码：

```
function mover(){
    this.status=[];
    this.actionHandle={
      left:moveLeft,
      right:moveRight,
      top:moveTop,
      bottom:moveBottom
    }
}
mover.prototype.run=function(){
    this.status=Array.prototype.slice.call(arguments);
    this.status.forEach((action)=>{
        this.actionHandle[action]();
    })
}
new mover().run('left','top')
```

run(...args: any[]): void

首先新建一个mover类，因为考虑到可能会有复合移动，有两个参数，所以状态status用一个数组来接收，新建有一个策略对象actionsHandle，里面把同不同移动行为，定义成一个属性。然后再mover类的原型上面定义运行的方法，在运行方法中，我们要知道是如何移动的。接着在运行方法中，获取的形参列表，并转化未数组，赋值给status，循环status，执行策略对象里面相对应的方法。最后实例化 new mover().run('left','top ')
对比前后两种代码，无论在编写上，还是在整个代码的简洁上，都有一个质的提升，这种思想，其实就是我们把这些判断变成策略对象中的各种状态，然后根据不同的状态去执行不同的行为，而不是通过一大堆的if-else去判断执行行为。这就是状态模式来解决复合运动这样一些复杂的if-else分支做的一列优化

外观模式示例

需求一：封装插件的规律，插件基本上都会给最终使用提供一个高级接口。我可以看一下选项卡插件的示例：

```
function tab(){
  this.dom=null;
}
tab.prototype.initHTML=function(){

}
tab.prototype.changeTab=function(){

}
tab.prototype.eventBind=function(){

}
tab.prototype.init=function(config){
  this.initHTML(config);
  this.eventBind();
}
```

新建一个tab类，在tab类的原型上写了各种子接口，或者子方法，最后使用一个统一的init的高阶接口来给用户或者而外界调用。外面不用一个个去调用子接口，只管最终的那个高级接口的调用即可。这种思想就是典型的外观模式。

需求二 :封装成方法的思想,在兼容时代，我们会常常需要检测能力，不妨作为一个统一的接口

在兼容性要求还是非常严格的时代，我们通常需要能力检测，比如dom事件绑定有dom一级、二级，在那个时代经常要检测浏览器支持哪一级然后再采用对应的方式去进行事件绑

定，当时会采用封装成方法的思想，把这些检测封装成统一的接口方法来进行事件绑定操作。代码示例：
比如会有dom二级dom.addEventListener()、还有dom.attchEvent()、还有dom.onclick这样一些绑定操作

```
dom.addEventListener();
dom.attchEvent();
dom.onclik
```

面对这样一些绑定操作我们通常都要去检测浏览器支持哪个然后就用哪个，在当时就会把这种操作封装成方法，比如封装一个addEvent方法，它需要接收dom（绑定的元素）、type（事件的类型）、fn（事件的回调）三个参数，然后对浏览器进行能力检测，判断浏览器的支持状态去绑定事件

```
function 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;
  }
}
```

封装好之后再绑定事件时不需要再去自己写一遍能力检测了，只需要通过外观模式统一的包装一个接口，然后调用这个接口就可以完成功能了，这就是外观模式的思想。

迭代器模式示例

1、构建一个自己的forEach

需求：forEach方法其实是一个典型的迭代器方法，构建一个forEach方法，让它能循环数组，也能循环对象

首先创建一个迭代器类，这个迭代器类接收一个数据，它可能是数组，也可能是对象，将接收的数据挂到data属性上

```
function Iterator(data){
    this.data=data;
}
```

然后给Iterator类的prototype添加一个dealEach方法，它接收一个回调函数，我们先判断一下data是不是一个数组，如果是数组就进行for循环，循环体内调用fn，把每次循环的内容和下标给出去即可；如果是对象就进行for in循环，把value和key传给fn即可

```
Iterator.prototype.dealEach=function(fn){
    if(this.data instanceof Array){
        for(var i=0;i<this.data.length;i++){
            fn(this.data[i],i);
        }
    }else{
        for(var item in this.data){
            fn(this.data[item],item);
        }
    }
}
```

这样就构建了一个简单的forEach循环，我们就可以在不用进行手动for循环数组或对象的情况下拿到每一项的内容，前面说了forEach就是一个典型的迭代器模式的代表，从这里可以看出迭代器模式说白了就是给我们提供一个方法，让我们能够不用自己去手动遍历就能对数组和对象进行统一的操作。

2、给你的项目数据添加迭代器

需求：项目中会经常对后端数据进行遍历操作，封装一个迭代器，使遍历更加方便

假设我们有这么一个数据

```
var data=[{num:1},{num:2},{num:3}]
```

项目里面肯定避免不了要找出数组里面有哪些对象它的num等于2或者等于1，检查一系列的对象中有哪个对象的某一个属性满足某个条件这种操作经常会有，我们可以把这种操作封装成迭代器，代码示例：

首先创建迭代器工厂，因为迭代器肯定要经常去创建迭代器对象来进行操作，所以用一个工厂模式来封装，这个工厂接收一个参数data，然后在工厂内部新建迭代器对象，把接收的data放在迭代器的属性上

```
function i(data){
  function Iterator(data){
    this.data=data;
  }
}
```

然后给迭代器新增一个方法，让它能够检查出data里面符合要求的对象。假设我们添加一个getHasSomeNum方法，它接收两个参数，第一个是handler，第二个是num，handler可以是一个方法，也可以是一个属性名，如果是一个方法我们就以这个方法去检查data，如果是一个属性名就用这个属性名去检查对象的属性名。

```
Iterator.prototype.getHasSomenum=function(handler,num){
  var _arr=[];
  var handleFn;
  if(typeof handler=='function'){
      handleFn=handler;
  }else{
      handleFn=function(item){
        if(item[handler]==num){
            return item;
        }
      }
  }
}
```

函数体内新增一个数组变量_arr，用来储存符合条件的对象，然后新建一个handleFn，判断传入的handler是不是一个方法，如果是就把它赋值给handleFn，这是自定义的检查方式；如果不是就把handleFn赋值为一个function，让这个function根据属性名去检查对象，这是默认检查的方式。判断对象的handler是不是等于num，如果是就返回这个对象，这段代码其实就是一个享元模式。

然后循环data，调用handleFn传入data的每一项，并把返回值赋值给_result变量，判断_result如果不是undefined（item传进去不符合条件就会返回undefined），就把_result存入_arr中；最后将_arr返回，这样就可以调用getHasSomeNum方法直接来检查data了。

```
Iterator.prototype.getHasSomenum=function(handler,num){
  var _arr=[];
  var handleFn;
  if(typeof handler=='function'){
      handleFn=handler;
  }else{
      handleFn=function(item){
        if(item[handler]==num){
            return item;
        }
      }
  }
  for(var i=0;i<this.data.length;i++){
      var _result=handleFn.call(this,this.data[i]);
      if(_result){
        _arr.push(_result);
      }
  }
  return _arr;
}
}
```

使用时调用迭代器工厂传入data，调用getHasSomeNum方法传入属性名和需要检查的值即可筛选出num为1的对象。

```
i(data).getHasSomenum('num',1)
```

如果需要自定义检查，比如说要检查它的值减去1等于2的对象，调用getHasSomeNum时传入function然后再自己判断即可，代码示例：

```
i(data).getHasSomenum(function(item){
    if(item.num-1==2){
        return item;
    }
})
```

通过把for循环封装成方法，让我们在处理繁杂的数据检查之中能够不用每次都去手动遍历它，这就是迭代器模式的思想。

备忘录模式的示例

1、文章页缓存

需求：项目中有一个文章页，现在要对它进行优化，如果上一篇已经读取过了，则不发起请求，否则请求文章数据

代码示例：

```
function pager(){
  var cache={};
  return function(pageName){
    if(cache[pageName]){
      return cache[pageName];
    }else{
      axios.get(pageName).then((res)=>{
        cache[pageName]=res;
      })
    }
  }
}
```

代码中创建一个pager函数，函数内部新建了一个cache对象用来缓存文章数据，return一个function让它能够读取到缓存数据，这个方法接收一个pageName，也就是当前文章页的名字，通过判断去检查缓存对象中有没有这个pageName，如果有就直接返回数据，如果没有再发起请求并将数据以键值对的形式存入cache对象。

这里的代码就是借助备忘录模式来完成文章缓存的思路，这段代码并不能完整的实现一个文章页缓存，在真实开发场景中还需要做一些别的事情，但这是一种去缓存页面、缓存内容的思路，通过一个缓存对象以键值对的形式缓存内容，然后只需要去检查键名，如果不存在就进行获取操作并将获取到的内容存入缓存对象；如果存在就直接返回内容，这就是备忘录模式的核心思想。

2、前进后退功能

需求：开发一个可移动的div，拥有前进后退回滚到之前位置的功能

思路很简单，首先创建一个moveDiv函数，函数内部创建一个stateList用来缓存之前的状态，将之前的状态以一串数据的形式记录在数组当中，然后创建一个指针指向当前状态，初始为0

```
function moveDiv(){
  this.stateList=[];
  this.nowState=0;
}
```

每次去移动div的时候需要一个方法，所以在prototype下新增一个move方法，这个方法接收两个参数，一个是type（移动的方向），一个是num（移动的数值），我们假设有一个changeDiv是用来移动div的，在move方法体内调用changeDiv传入type和num，每次div位置改变之后将状态push进stateList，所以我们push一个对象，这个对象记录type和num，还要将当前指针（nowState）的状态改变，让它指向stateList的最后一位。

```
moveDiv.prototype.move=function(type,num){
  changeDiv(type,num);
  this.stateList.push({
    type:type,
    num:num
  })
  this.nowState=this.stateList.length-1;
}
```

上面代码写完以后再去进行前进后退就非常简单了，只需要根据状态指针去拿上一位或下一位在stateList里面的状态，然后再次调用changeDiv方法去移动div就可以了。

我们创建一个前进方法，方法体内新建一个变量_state，判断当前指针小于stateList的总长度，也就是说当前指针不在最后一位，代表它可以前进，然后把当前指针加加，取出当前状态也就是stateList里面对应的nowState项，并把取出来的状态赋值给_state变量，然后调用changeDiv把当前状态传入改变div的位置

```
move.prototype.go=function(){
  var _state;
  if(this.nowState<this.stateList-1){
    this.nowState++;
    _state=this.stateList[this.nowState];
    changeDiv(_state.type,_state.num);
  }
}
```

对应的后退也是同理，只需要去改变指针的位置，然后取出指针对应的状态，再次调用changeDiv方法去改变div的位置就可以了。这是大多数前进后退功能所对应的思想，无非就是把之前的状态和之后的状态储存在数组里面缓存起来，然后根据指针看它现在指向哪个状态，然后取出对应的状态即可。这个代码实现是备忘录模式一个很好的体现。