本节主要学习方法的可扩展性以及怎么更好的扩展方法，方法是组成程序的基础单元，基础单元的可扩展性是整个程序的可扩展性保障。

可扩展性顾名思义是保证代码、程序能够更好的进行扩展。再厉害的程序员都会写bug，再好的产品经理都会改需求，在遇到需求变更的时候不能总是跟产品硬怼，一个好的程序员需要随时做好改需求的准备，不要遇到问题就想着砍死产品经理，先思考是不是因为自己的代码可扩展性方面没做好，所以导致改需求才这么难。改代码的时候难免会发现之前的代码思考的不够全面，在这个过程中写第一份代码的时候稍微注意一下可扩展性，就可以极大的减少后面修改代码的难度。

提高可扩展性的目的?

- 面临需求更改时，方便更改需求
- 减少代码修改的难度

什么是好的可扩展?

- 当发生需求变更的时候，不需要把之前的所有代码推倒重写。好的可扩展性组成的程序就像一块橡皮泥，当你要改变的时候只需要重新捏一下形状即可。
- 代码修改不会引起大规模变动。很多人在改代码的时候改了一个bug引起十个bug，按理来说改一个问题只需要加一个解决问题的代码就行了，但因为前期代码设计的不好，导致要修改很多之前的代码才能让这一个功能的代码加上去，这样就会引起大规模修改。
- 方便加入新模块。从模块的角度讲，比如产品要加一个新模块的时候，我们整个模块之间的组合就像积木一样，加入一个新模块就像加入一个新积木，拼上去就完事了，这样加入整个模块就非常方便。

一、提高可扩展性的设计模式

更好的更改代码的设计模式：

这两个模式都是为了让我们更好的更改代码，它们更像是一种技巧，当我们的方法需要变更的时候能够让我们更好的进行变更。

- 适配器模式：适配器顾名思义，就是用来做适配的，比如我的电脑只支持typeC的耳机，但是我现在只有圆孔的耳机，我要用圆孔的耳机插上电脑来听歌怎么办？第一种方案就是买个typeC的耳机，但是买新的太贵了不划算，所以第二种方案就是买个转接头，把圆孔的耳机转接成typeC的，这个转接头就是适配器，**适配器模式针对的是调用的接口的名字产生了不通用的问题。**

目的：通过写一个适配器，来代替直接替换老代码

应用场景：当接口不通用的时候解决接口不通用的问题。比如我们要调用一个a接口，但实际上这个接口名叫b，两个接口就不适配了，这时候肯定不能把a改成b，改了就相当于重写原代码了，通过写适配器，把a接口适配到b接口就不用去改写老代码了

- 装饰者模式：**装饰者模式针对的是方法本身的作用**，当一个方法的作用不够用了，需要添加新功能，但是又不能直接去修改之前的方法，使用装饰者模式就能更优雅的扩展我们的方法。

目的：不重写方法的扩展方法

应用场景：当一个方法需要扩展，但又不好去修改方法。

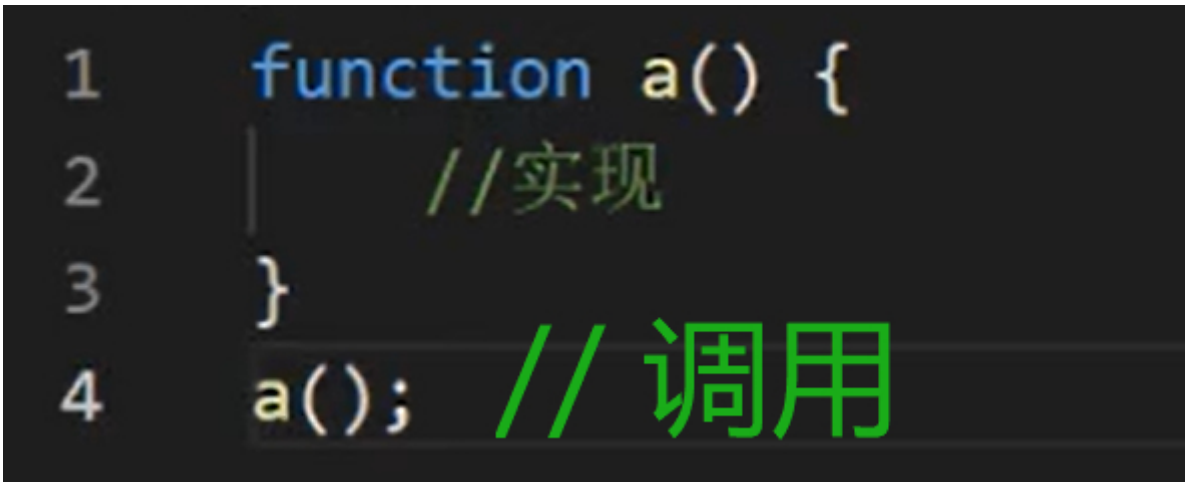
解耦方法与调用的设计模式：

- 命令模式：命令模式针对的是代码设计，命令模式的使用是在设计方法的时候，而不是更改方法的时候，设计之初就要开始考虑这个模式

目的：解耦实现和调用，让双方互不干扰

应用场景：调用的命令充满不确定性时，可以考虑使用命令模式。（功能比较单一的时候不要使用，命令层要写很多代码，增加复杂性）

什么是实现？实现就是方法的功能，比如我们有一个a方法，a方法里面的操作就是实现，而调用就是直接调用a()



什么是解耦实现和调用？像上面代码示例中定义a方法和调用a方法相当于：调用 > 方法，而命令模式相当于在调用和方法之间加上一层命令层：调用 > 命令层 > 方法，调用时不用直接去调用方法，而是通过输入命令输入到命令层，然后由命令层来解析命令再去调用具体方法：命令 > 命令层 > 方法

这样做的好处是：

1. 对比于之前直接调用 > 方法，通过命令层来调用就不需要关心具体应该调用哪个方法，也不用去了解有哪些方法，只需要关心输入的命令就好
2. 对比于之前直接调用 > 方法，之前的方式使用方法的人和方法本身是直接的关系，而在中间加入命令层之后，方法本身和要用方法的人就解耦了，这个时候方法就算发生变动，不用说命令也要跟着变动，只需要在命令层中改变一下对命令的解析即可，所以说方法的变动就不会影响到命令。同样的，当命令发生变化之后也不会影响到方法，中间都有一层命令层来作为缓冲，我们可以在命令层做一个调配和调度，这样它们双方发生变动都不会影响到彼此。

我们写代码其实就相当于命令模式，写好的代码最终在电脑上是怎么用二进制执行的不需要我们关心，只需要关心我们输入的代码就好，输入的代码就像命令，解析器V8引擎相当于命令层，它负责把命令解析成计算机能够执行的二进制语言，方法本身就是计算机去执行的代码。为什么要把写代码变成命令模式？因为编程的方向是多元化的，有可能编写成任何一个效果，可以编写很多效果，所以采用命令模式非常合适。

二、基本结构

适配器模式的基本结构

需求：每次都写console.log太麻烦了，项目里要用log来代替console.log。

适配器模式使用非常简单，就是在新接口里面调用老接口，适配器模式的应用场景就是在接口不通用的时候做个适配器，解决这个需求只需要在log函数中调用console.log就可以了。

```
var log=(function(){
    return window.console.log
})()
```

装饰者模式的基本结构

需求：有一个他人写好的a模块，内部有一个方法b，不能修改他人模块的情况下，扩展b方法。

```
var a = {
 b:function(){

 }
};
```

装饰者模式三步走：

1. 封装新方法
2. 调用老方法
3. 加入扩展操作

所以我们新建一个自己的方法，在其内部调用b方法，然后加入要扩展的功能，这样就可以在不修改原对象的情况下扩展b方法的功能了，代码示例：

```
var a = {
 b:function(){

 }
};
function myb(){
 a.b();
 //要扩展得方法
}
```

命令模式的基本结构

代码示例：

```
var command=(function(){
    var action={

    };
    return function excute(){}
})();
```

上面代码创建一个匿名自执行函数，函数里面有一个action，它是方法的实现，excute就是我们的命令层，通过这个匿名自执行函数拿到的command就是命令层，要调用action里面的方法时，通过给command输入输入命令，这些命令就会在excute命令层进行解析来调用action里面的实现，使用者不需要关心具体要调用action里面哪个方法。

命令模式有两个要素：

1. 一个是行为action
2. 一个是命令执行层excute

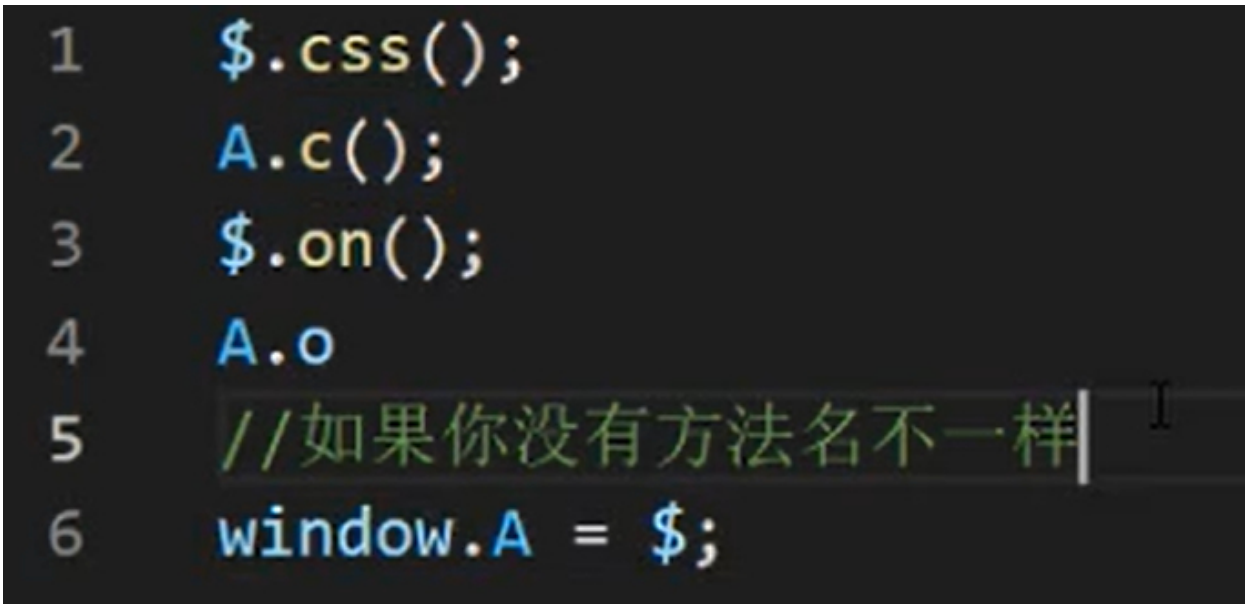
三、应用示例

适配器模式的示例

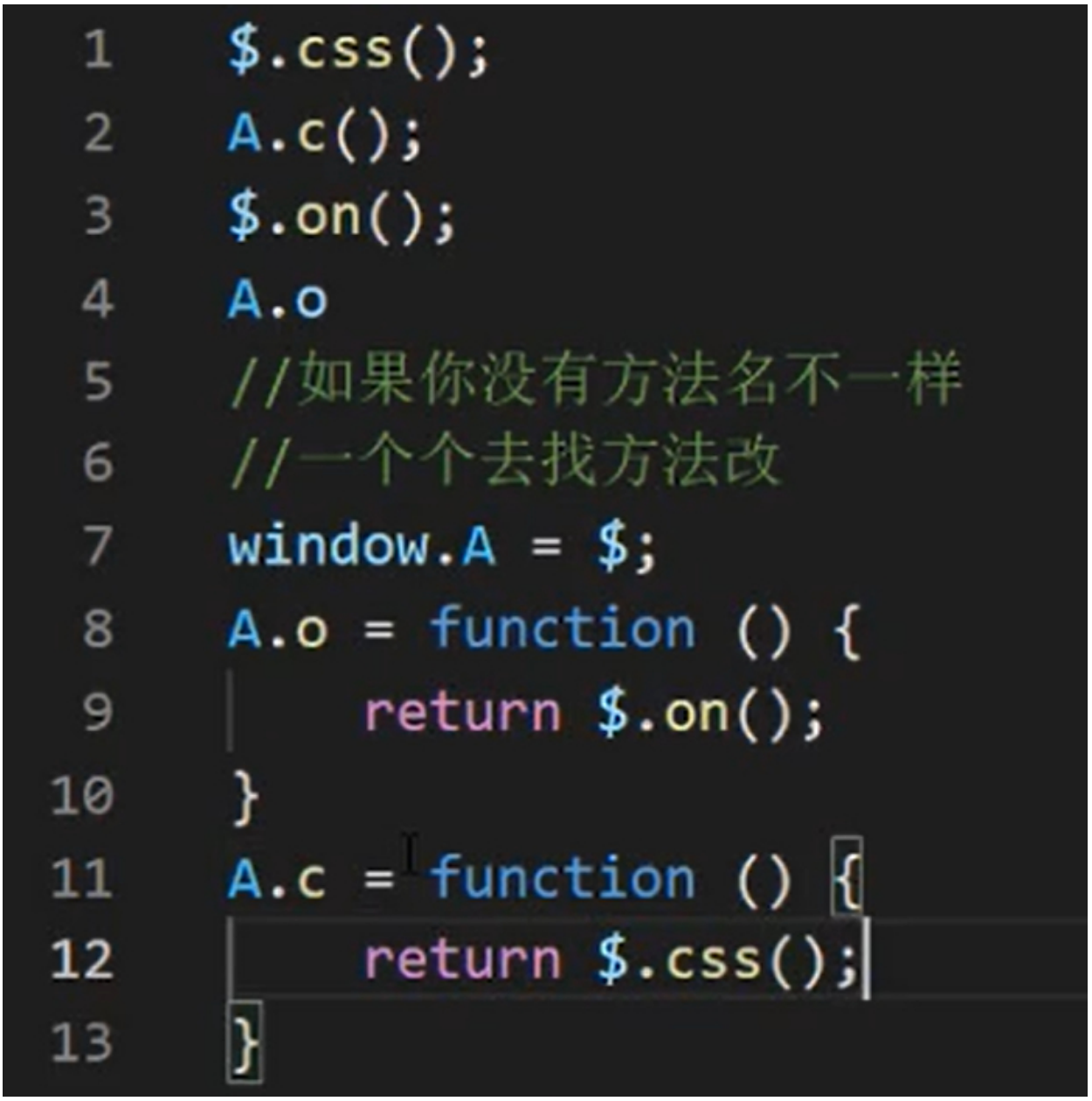
1、框架的变更

需求：目前项目中使用A框架，A框架和jQuery非常类似，但是A框架对进公司的新人很不友好，新人进入团队还需要学一下这个框架的使用，所以现在要改成jQuery，这两个框架虽然时分类似，但存在少数几个方法不同。

比如在jQuery中css的调用是$.css()，而在A框架中是A.c()，在jQuery中绑定事件是$.on()，而A框架中是A.o()，这就是问题所在，如果这两个框架中的方法名没有任何不一样的话，直接把A赋值为jQuery就OK了，现在存在方法名不一样，直接把jQuery赋值给A就会导致旧代码中调用A.css()报错



解决这个问题很多人的处理方式是一个个去找这些方法改掉，过程很明显就是重写老代码，这就是典型的适配器应用场景，我们只需要写一个适配器，不用改动老代码，让这两个接口名能够适配就好了。根据适配器模式的步骤，在新接口中调用老接口即可，代码示例：



```
$.css();
A.c();
$.on();
A.o
//如果你没有方法名不一样
//一个个去找方法改
window.A = $;
A.o = function () {
    return $.on();
}
A.c = function () {
    return $.css();
}
```

2、参数适配

需求：为了避免参数不适配产生问题，很多框架会有一个参数适配操作。

在JavaScript中，健壮性非常重要，健壮性的基础保障就是判断参数类型赋予默认值，比如通过typeof判断，这种判断对于简单的数据类型是好使的，但如果参数是一个配置对象怎么办？比如Vue，在new Vue时传入的不是一个简单的参数，而是一整个的配置对象，它里面包含了很多内容，如template、data、methods等等，这些内容里面肯定会有一些内容是必填的，比如template、data，像这样一个对象怎么去保障别人使用的时候传的配置对象里面该必填的都必填呢?

如果使用typeof判断只能判断这个配置参数是一个对象，但是配置参数里面的东西有没有必填判断不出来，对于这样的配置对象形式的参数，我们最好给它做一个参数适配，参数适配就可以理解为适配器模式的变更，当你遇到一些方法它接收的参数是一个配置对象的时候，一定要养成一个习惯，给这个配置对象做一个参数适配，怎么去保障它里面必传的参数都传了呢？很简单，在函数里面写一个默认的配置对象，在这个默认的配置对象中将必传的属性都写上，比如name属性必传、color属性必传，代码示例：

```
6   //参数适配
7
8   function f1(obj) {
9       var _default = {
10          name: 'xxx',
11          color: 'red'
12      }
13  }
```

然后当参数传进来的时候先不急着操作，先做一下适配，循环这个参数，如果传入的参数自身有必传项就用自身的，否则就用默认的代替，这样就保障了必传项起码会有一个默认值，不会因此报错。

```
6   //参数适配
7
8   function f1(obj) {
9       var _default = {
10          name: 'xxx',
11          color: 'red'
12      }
13      for (var item in obj) {
14          _default[item] = obj[item] || _default[item];
15      }
16  }
```

当你写的方法要接收的参数是一个配置对象时，就通过这种参数适配的方式去保障必传的参数都有值，这是一个好的习惯，在工作中一定要保持。

装饰者模式的示例

1、扩展已有的事件绑定

需求：项目要进行改造，需要给dom已有的事件上增加一些操作。

假设你进入一家新公司，接手了前同事的代码，他在dom上绑定了很多事件，比如删除按钮绑定了点击事件，点击就进行删除操作，你接手之后产品跟你说觉得之前这种点击就删除没有提示的方式不太友好，需要你在点击确定或者删除的同时给出一个提示，这时候你会怎么做？

很多人会这么做

- 不去找他之前的代码写在哪了，直接重写整个绑定事件
- 找到老代码，然后改一下

这两种方式都是错的，如果采用第一种方案，势必要把它之前的删除功能代码再写一遍，非常麻烦，如果采用第二种方案，去找老代码这个找的过程也很麻烦，最好的方式就是采用装饰者模式，装饰者模式是用来干嘛的？就是当你发现一个方法它的原功能不适用、要扩展，但你又不能直接去修改原方法的时候就可以派上用场了。

所以我们用装饰者模式来做这个事情，考虑到要做这个事情的按钮有很多，就不一个个装饰了，采用 工厂模式的思维，直接封装一个装饰工厂，使用时告诉我你要装饰哪个dom，要扩展什么操作就可以了，代码示例：

```
var decoractor = function (dom, fn) {
    if (typeof dom.onclick == 'function') {
        var _old = dom.onclick
        dom.onclick = function () {
            _old.apply(this, arguments);
            fn.apply(this, arguments);
        }
    }
}
```

上面代码中，首先出于健壮性考虑，先判断一下dom上面有没有绑定事件，有的话再装饰，没有就不管。根据装饰者模式三步走：1、封装新方法，2、调用老方法，3、扩展新功能；先给click事件赋值为新方法，提取出dom的老方法，然后在click事件的新方法中调用老方法，然后加入我们要扩展的操作。

在使用的时候比如说要装饰删除按钮，然后要扩展提示功能，删除之后打印删除成功

```
decoractor(deleteDom, function () {
    console.log('删除成功')
})
```

这样既不用去找老代码也不用去重新写整个事件绑定，只需要调用装饰工厂就好了，扩展起来的速度就快多了。

2、Vue的数组监听

需求：Vue中使用defineProperty可以监听对象，那么数组监听是怎么实现的?
vue2.0响应式困境 是通过Object.defineProperty() 进行对象拦截的,但是如果是数组的话,这个该如何监听呢?这就是vue2.0的困境所在 你直接直接修改的数组的下标,你是无法触发响应式的.那如何才能出发,你通过调用Array的push shift ... 才能触发整个响应式,那为什么会这样呢?

Object.defineProperty()只能给对象用,那数组呢?我们通过调用Array的原生方法,去触发整个响应式.但是的数组的原生方法,是不可以改动的.这个就可以运用到我们的装饰者模式来实现,如何实现的呢?
我们看代码示例:

```
var methodsArr = [
    'push',
    'pop',
    'shift'
]
var arrayProto = Array.prototype
var arrayMethods = Object.create(Array.prototype);
methodsArr.forEach((method) => {
    arrayMethods[method] = function () {
        var original = arrayProto[method];
        var result = original.apply(this, args);
        dep.notify();
        return result;
    }
})
```

代码分析:

1. 先用一个MethodsArr变量存入一些数组的方法
2. 取出Array的原型,并把他拷贝给一个新的对象.arrayMethods
3. 遍历MethodsArr ,然后通过重新定义新方法 arrayMethods['push'.....]
4. 调用老方法 arrayProto[method].apply(this,args)
5. 扩展 dep.notify()Vue就是 通过dep.notify进行响应式
6. 然后返回原来result

完全遵循装饰者模式的三个特点

1、封装新方法，2、调用老方法，3、扩展新功能

命令模式的示例

需求:封装一系列的canvas绘图命令

常规写法,

```
canvas
var myCanvas = function () {

}
myCanvas.prototype.drawCircle = function () {

}
myCanvas.prototype.drawRect = function () {

}
```

我们新建一个myCanvas类,在myCanvas的原型上注册相关方法,画不同形状,如果是固定的,这这种方式完全没有问题,但是明显不能固定,因为用很多不同形状,我们不可能一一手写添加到类的原型中去.,是朝一个多方面去调用的.

但是我们封装成命令模式的话,我们可以先定义照命令者模型 action和excute

```
var canvasCommand = (function () {
    var action = {
      drawCircle:function(){

      },
      drawRect:function(){

      }
    };
    return function excute(commander) {
        commander.forEach((item)=>{

        })
    }
}) () 少了一个括号，这是自执行
canvasCommand([{type:'drawCircle',radius,number:3}]);
```

用户只管输入调用canvasCommand,然后输入相关的命令,至于相关命令对应的执行,那是命令层的事情.其实我的webpack就是典型的命令模式,通过配置webpack相关参数的命令,

然后webpack通过他的命令层,执行不同的操作.webpack其实就是封装成了一个超大的命令模式