JS正式课day11前三周回顾

Es6新语法

1.let / const

```
let和es5中的var区别：
1）let不存在变量提升（变量不允许在声明前使用）
2）let不允许重复声明
3）在全局作用域中基于let声明的变量不是window的属性
4）typeof 检测未被声明的变量不是undefined而是报错
5）let会形成块级作用域（类似于私有作用域，大部分大括号都会形成块级作用域）`
```

2.解构赋值
3.‘…’拓展/剩余/展开运算符

4.箭头函数

```
和普通函数区别
1）没有arguments，但是基于...arg获取实参集合（结果是一个数组）
2）没有自己的this,箭头函数中的this是上级作用域中的this,继承上级
```

5.Es6模版字符串

6.Promise(async/await)
7.class(Es6中创建类的)
8.interator（for of 循环）
9.Map / Set

重排（回流）/重绘

1.什么是重排和重绘
浏览器渲染页面的时候是按照‘先创建Dom树—>在加载css->生成渲染树 rendertree->把渲染树交给浏览器（GPU）进行绘制’，然后后期我们修改元素的样式（但是没有改变大小和位置），浏览器会把当前元素重新生成渲染树，然后重新渲染，这个机制就是重绘，但是一旦改变元素的位置或者大小等，浏览器就会从Dom树重新计算渲染，这个机制就 是回流（重排），不论是重排还是重绘都非常的消耗性能

2.解决方案
1）需要动态向页面追加元素的时候，基于文档碎片或者先把需要增加的元素拼接成字符

串，最后统一进行增加
2）读写分离：把统一修改样式都放到一起执行，新版浏览器都有一个自己的检测的机制，如果发现下面挨着的操作也是修改元素的样式，会把所有修改的事先存放起来，知道遇到非修改样式的操作。会把之前存储的统一执行，引发一次回流和重绘

当然还有一些其他的办法，这些是最常注意的，我认为减少Dom的回流和重绘是非常重要的性能优化手段之一

JS中的this汇总

this :当前执行的主题（谁执行的这个方法，那么this就是谁，this和当前方法在哪创建的或者在哪执行没有必然关系
1.给元素的某个事件绑定方法，方法中的this就是当前操作的元素本身
2.函数执行看函数前面是否有点，有的话点前面是谁this就是谁，没有点的话this就是window,在Js严格模式下，没有点的话this就是undefined
3.构造函数执行，方法中的this一般都是当前类的实例
4.箭头函数中没有自己的this,this是上下文中的this，继承父级的this.
5.在小括号的表达式当中，会影响this的指向（相当于自执行函数了）
6.使用call/apply/bind 可以改变this指向
在非严格模式下，第一个参数如果不写或者是undefined/null ，this是window
在严格模式下，写谁就是谁 不写就是undefined

谈一下对作用域链和原型链的理解

作用域链：
函数执行会形成一个私有的作用域，形参和在当前私有作用域中声明的变量都是私有变量，当前的私有作用域有自我保护机制，私有变量和外界是没有关系的，但是如果私有作用域中遇到一个非私有的变量，则会向它上级作用域找，一直找到window为之，这种变量一层层向上查找的机制就是‘作用域链’

原型链：
首先在自己的私有属性中进行查找，如果自己没有这个属性，则会通过**proto**向所属类的原型上查找，如果查找不到则会继续基于**proto**向上查找，一直找到Object的原型位置

算法

常用算法：递归/去重/冒泡排序/插入排序/快速排序/时间复杂度/空间复杂度/KMP

递归：函数执行，自己调用自己再执行就是递归（递归时基于条件判断的）

数组扁平化（将多维数组转为一级数组）

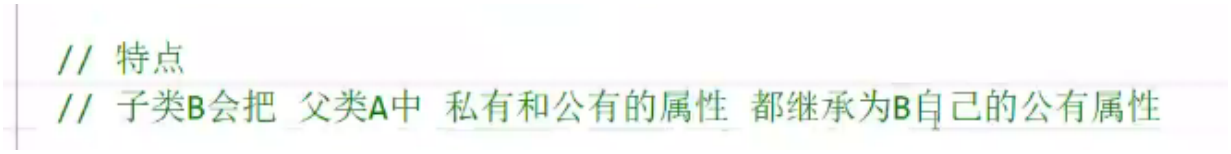
var ary = [1,[2,[3,[4,5]]],6]
方法一：
ary=JSON.stringify(ary)
var str = ary.replace(/([])/g,”)
str =’[‘+ str + ‘]’
ary=JSON.parse(str)
console.log(ary)

方法二：
ary = ary.join()
ary =’[‘+ ary + ‘]’
ary = JSON.parse(ary)

方法三：
var result = []
function fn(n){
if(n.length=0) return
for (var i = 0; i < n.length; i++) {
var cur = n[i]
if(typeof cur =‘number’){
result.push(cur)
}else{
fn(n[i])
}
}
return result
}
//console.log(fn(ary))[1, 2, 3, 4, 5, 6]

JS的继承方式

子类继承父类的属性和方法
1.原型继承
A想用B上的方法，让A的原型等于B的实例即可
记得自己在A的原型写上constructor =A
存在的问题：子类可以重写父类原型上的方法（重写），子类和父类还有关系的



```
// 子类重写父类
// 不安全 当子类将父类原型属性修改后，就会影响到 父类所有的实例对象

// console.log(b.__proto__.__proto__.title = 'hi nihao')
console.log(B.prototype.__proto__.title = 'hi nihao')
console.log(b.title)
```

2.call继承（借用构造函数）

```
// 借用构造函数（call继承）
// 特点 只继承了父类的私有属性 作为自己的私有属性

function A() {
  this.name = 'AAA'
  this.say = function() {

  }
}

A.prototype.say = function() {
  console.log('sayA')
}

function B() {
  // 通过call方法调用父类 并且 让父类里面this 修改为子类的实列
  A.call(this) // B this => new B => b
}
let b1 = new B()
let b2 = new B()
console.log
```

需要判断一下这个函数是普通函数执行还是构造函数执行（判断里面的this）

```
// console.log(b1.say === b2.say)

function B() {
  // 如果当前函数B 中this 是自己的实例（说明我们是通过构造函数方式执行 new B）
  if (this instanceof B) {

  }
}
B()
```

3. 组合式继承（劣质版）（es5)
   用到了call和原型式继承

```
// 组合式继承 原型链继承 + call继承

// 缺点
// 1.组合式继承 子类会把父类的实例属性（私有属性），继承过来两份
// 一份存储在子类实例对象本身 另一份存储在子类原型上
// 2. 调用了两次父类
```

产生重复在于new A() A执行了

避免以上情况改变方式：原型式继承

4.原型式继承/寄生组合式继承（原型式继承+call继承（借用构造函数继承））

```
// 原型式继承
// 只能继承父类的原型属性

function A() {
  this.name = 'A私有'
}
A.prototype.tile = 'mess'

// B.prototype = new A()
// 创建一个新对象 这个新对象{}.__proto__ => A.prototype(A的原型)
// 将子类的原型改写为 这个新对象
B.prototype = Object.create(A.prototype)          zhuyi!
B.prototype.constructor=B
console.log(B.prototype.__proto__ === A.prototype)
function B() {

}
let b = new B()
console.log(b)
```

以上图上方法只能继承父类的原型

也要注意给B的原型上写B。prototype 。constructor = B

Object上的create 封装机制：

```
Object._create = function(o) {
  // 临时的构造函数
  function F() {}
  F.prototype = o // 将o作为了F的原型对象
  return new F
}

let o1 = {id: 100}
let o2 = Object._create(o1)
console.log(o2.__proto__ === o1)
console.log(o2)
```

5.冒充对象继承

```
<script>
  // 冒充对象继承

  function A() {
    this.name = '私有A'
  }

  A.prototype.mess = '原型A'

  function B() {
    if(this instanceof B) {
      // 当子类执行时 在函数体内 生成一个父类的实列 将父类的实列进行遍历，复制给子类实列
      let a = new A()
      for (let k in a) {
        this[k] = a[k] // b[k] = a[k]
      }
    }
  }

  let b = new B()
</script>
```

麦克风静音
开启麦克风。
扬声器静音
检测到电脑声音已关闭，开

6.es6中class类实现继承

ES6中创建类是有标准语法的,不能当作普通函数执行

```
57
58  class Fn{
59      constructor(n,m){
60          this.x=n
61          this.y=m
62      }
63      //这个就是给Fn原型上设置方法
64      //（只能是方法不能是属性比如bb=100/bb:100)可以在外面写
65      getx(){
66
67      }
68      //Fn当作普通对象设置同样方法，不能写属性，属性在外面写
69      static AA(){
70
71      }
72  }
73  Fn.prototype.bb=100
74  //等价于
75  var f = new Fn(10,20)
76
77
```

继承其它类的原型上的属性和此类实例的私有属性
class 类名 extend 继承名称{//=>类似实现原型继承
constructor(){
super()//=>类似于call继承：在这里super相当于把A的constructor给执行了，并且让方法中德this是B的实例，super中的实参都是在给A的constructor传递
}
}

对Jquery的原理理解

JQ是一个js类库，里面提供了很多的常用的方法，有助于我们快速开发，而且这些方法兼容所有浏览器（V2/V3 不兼容低版本浏览器）

之前看源码时候，发现JQ就是一个类，而$()就是创建这个类的实例，这个实例是基于makeArray创造出来的类数组

JQ提供的方法有两部份，一部分是放在原型上的，供实例调取使用，另一部分是放在对象上的，直接$.xxx调取使用，想要后期自己扩展方法（包括基于JQ写插件）都可以用extend这个方法向JQ中扩充

JQ中提供了动画，事件，AJAX等方法主要重点学习里面的封装和编程思想（发布订阅这种设计模式是依据JQ的$.Callbacks学习研究的）

定时器

```
for (var i = 0; i < 5; i++) {
    setTimeout(function(){
        console.log(i)
    },1000)
}
```

=>定时器是异步编程，等待循环结束后，才会执行定时器设置的方法，方法执行的时候i已经是循环结束后的5

```
for (let i = 0; i < 5; i++) {
    setTimeout(function(){
        console.log(i)
    },1000)
}
```

=>基于es6中的let解决：let在每次循环的时候都会形成一个块级作用域，在这个作用域中把当前本次循环的I值保存下来，后期用到的时候就直接找保存的值

=>不用let的话还可以用闭包

```
for (let i = 0; i < 5; i++) {
   ~(function(i){
    setTimeout(function(){
        console.log(i)
    },1000)
   })(i)
}
```

=>或者基于bind预先处理一下函数中德this和参数

```
for (var i = 0; i < 5; i++) {
    setTimeout(function(){
        console.log(i)
    }.bind(null,i),1000)
}
```

什么是闭包

闭包是Js中一个非常重要的机制，我们很多的编程思想都是基于闭包完成的，我对闭包的理解：闭包就是函数执行产生一个私有的作用域，在这个作用域中产生的私有变量和外界互不干扰，而且这个作用域不销毁，所以闭包可以说就是保存和保护变量的

=>实际项目开发发中很多地方用到闭包、
比如：1.循环事件绑定，由于事件绑定是异步编程，我们此时在循环的时候把索引存储起来（是基于闭包存储，也可以自定义属性），后期使用的时候向上级查找即可
2.平时做业务逻辑的时候，我们都是基于单利模式来管理代码的，这种单利模式就应用到了闭包
3.柯理化函数也是基于闭包完成的
4.闭包比较占内存，尽量减少使用，但是有些时候必须用到

call和apply的作用

1.改变函数中的this（让函数执行）
2.可以基于call让类数组借用原型上的方法（例如借用slice类数组转为数组）
3.可以基于call实现继承
4.求数组中的最大值和最小值

瀑布流的实现原理

1.并排排列三列，三列没有自己的高度，靠内容撑开
2.通过API接口地址，基于Ajax从服务器端获取数据，拿出数据的前三项依次插入到三列中（数据绑定）
3.计算目前三列高度，按照高度由小到大把三列进行排序，再次获取数据中的三条，按照排好的li依次插入。。。。
4.当用户下拉到页面底部，加载更多的数据即可

图片延迟加载（图片懒加载）

是前端性能优化的重要手段之一，开始加载的时候并没有加载真实的图片，等待页面的结构和数据都呈现完成后，在加载真实的图片