day15 ES5及ES6

JavaScript构成

- BOM 操作浏览器对象
- DOM 操作文档对象
- ECMAScript 基础语法

ECMAScript介绍

ECMAScript是对应js的基础语法，里面包含js中除dom操作和bom的所有内容。ECMAScript主要拆分为俩个单词 ECMA (欧洲计算机协会) 、Script (脚本)。ECMAScript 简称（es）他主要的版本有 **ES3、ES5、ES6.**..对应的版本管理以及切换是由不同兼容问题产生的。低版本的兼容就比较好，高版本的兼容性较差。在项目中我们可以使用**bable.js**来进行生成代码的版本切换（后续框架中必然使用的）。ES3为基础版本他支持市面常用的所有浏览器，ES5支持市面上大多数浏览器, ES6只支持高版本浏览器。

ECMAScript的版本

- ES3 基础版本（大多数的基础内容都属于es3）
- ES5 (ES2009) 他在es3上增强对应的规范性以对应的方法
- ES6 (ES2015) 他在es5的基础上扩展了对应的类及对应的处理

ES5新增内容

严格模式

概述:

我们在平常书写代码的时候其实没有多大的规范性，我们可以自由发挥（可以声明任何的变量 且可以不写声明的关键词），平常的这种模式就被称为怪异模式（平常书写模式），在此之上诞生的具备规范性的模式就被称为严格模式（框架的底层设计）。

主流浏览器现在实现了严格模式。但是不要盲目地依赖它，因为市场上仍然有大量的浏览器版本只部分支持严格模式或者根本就不支持（比如 IE10 之前的版本）。*严格模式改变了语义。*依赖这些改变可能会导致没有实现严格模式的浏览器中出现问题或者错误。谨慎地使用严格模式，通过检测相关代码的功能保证严格模式不出问题。最后，记得*在支持或者不支持严格模式的浏览器中测试你的代码。*如果你只在不支持严格模式的浏览器中测试，那么在支持的浏览器中就很有可能出问题，反之亦然。

严格模式的书写

- 书写在首行
- 使用**use strict**来声明

```
<script>
    'use strict'
    hello = '你好'
    console.log(hello) //报错
</script>
```

严格模式的特性

- 声明变量必须使用var关键词声明
- 函数中的this不允许指向全局对象（global） （window）
- arguments中的实参不同步
- 函数名的参数唯一
- 函数声明只处于上下文对象中
- 禁止使用八进制方法
- 将一切不规范的地方全部抛出错误

数组新增的高阶函数

高阶函数就是以函数作为参数的函数被称为高阶函数

新增的高阶函数有

- **forEach 遍历 (返回值为void)**

```
var arr = ['a','b','c','d']
//传入的函数为操作函数
//传入的函数有三个参数 分别为遍历的值 遍历的下标 遍历的数组
arr.forEach(function(value,index,array){
    console.log(value,index,array)
})
```

- **map 遍历 （返回值是数组 且这个数组个数和遍历的数组个数一致）**

```
//map的使用和forEach是一样的 唯一的区别在于forEach map有返回值
// map返回的是一个数组 这个数组的个数一定和你遍历的数组个数是一样的
var nums = arr.map(function(v,i,array){
    console.log(v, i, array)
    if(i%2 == 0){
        return i
    }
})
console.log(nums)//[0,undefined,2,undefined]
```

- **filter 过滤 （返回的是一个数组）**

```
//filter用于过滤 里面传入的对应的函数一定返回值boolean true就是添加的返回的数组 如果是false就不添加
//v表示对应的值 i表示下标 array表示数组
var filterArr = ['a', 'b', 'c', 'ab'].filter(function (v, i, arr) {
    //条件 返回值boolean
    return /a/.test(v)
})
console.log(filterArr)
```

- **every 每个都满足条件返回true （传入的函数的必须返回的boolean）**

```
//every 当前是否每个都满足条件
var is = ['1', '2', '3', '4'].every(function (v, i, arr) {
    return v > 3
})
console.log(is) //false
```

- some 只有有一个满足条件返回false （传入的函数的必须返回的boolean）

```
//some当前是否存在满足条件的
var is = ['1', '2', '3', '4'].some(function (v, i, arr) {
    return v > 3
})
console.log(is)//true
```

- reduce 计算的 (返回的是一个值（一般情况下为number或者string）)

  示例求和

```
 // reduce求和
var number = [10, 2, 3, 13, 41, 34, 12].reduce(function (prev, current, i,
arr) {
    return prev + current
})
console.log(number)
```

  示例求偶数位的和

```
//reduce 第一个参数为处理函数 第二个参数为初始值 (如果没有设置默认为第一个 如果设置那么就对应设置的值)
//prev前面的结果值 （默认为第一个的值） 如果的参数设置了那么prev值为这个设置值
//current 现在的值 （默认从第二个开始） 设置了第二个参数那么默认从第一个开始
//i 默认从下标为1开始 设置了第二个参数那么下标从0开始
var number = [10, 2, 3, 13, 41, 34, 12].reduce(function (prev, current, i,
arr) {
    //判断是否为偶数位
    console.log(i)
    if (i % 2) {
        prev += current
    }
    return prev
},0)
console.log(number)//49
```

  参数讲解

  第一个为处理函数 处理函数里面存在四个参 分别为前面的结果值 当前遍历的值 当前遍历的下标 当前遍历的数组

  第二个为初始值 如果没有指定那么会将第一个值赋给前面的结果值 默认从下标1开始

- reduceRight 从右开始计算

```
//reduceRight 从右往左算
var str = ['a','b','c','d','abc','def'].reduceRight(function(prev,current){
    return prev+current
},'hello')
console.log(str) //hellodefabcdcba
```

高阶函数实现

forEach

```
var arr = [1,2,3,4]
//封装myForEach
function myForEach(callback) {
    if (typeof callback != 'function') {
        throw new Error('参数错误')
    }
    //遍历执行处理函数
    for (var i = 0; i < arr.length; i++) {
        callback(arr[i], i, arr)
    }
}
//调用
myForEach(function (v, i, arr) {
    console.log(v, i, arr)
})
```

map

```
//封装myMap
function myMap(callback) {
    if (typeof callback != 'function') {
        throw new Error('参数错误')
    }
    //返回的内容
    var results = []
    //遍历执行处理函数
    for (var i = 0; i < arr.length; i++) {
        results.push(callback(arr[i], i, arr))
    }
    return results
}
console.log(myMap(function(v){return v+'hello'}))
```

every

```
//封装myEvery
function myEvery(callback) {
    if (typeof callback != 'function') {
        throw new Error('参数错误')
    }
    //遍历执行处理函数
    for (var i = 0; i < arr.length; i++) {
        //只要一个是false那么直接返回false
        if (!callback(arr[i], i, arr)) {
            return false
        }
    }
    return true
}
console.log(myEvery(function(v){return v<10}))
```

some

```
//封装mySome
function mySome(callback) {
    if (typeof callback != 'function') {
```

```
        throw new Error('参数错误')
    }
    //遍历执行处理函数
    for (var i = 0; i < arr.length; i++) {
        //只要一个是true那么直接返回true
        if (callback(arr[i], i, arr)) {
            return true
        }
    }
    return false
}
console.log(mySome(function (v) {
    return v >= 4
}))
```

filter

```
//封装myFilter
function myFilter(callback) {
    if (typeof callback != 'function') {
        throw new Error('参数错误')
    }
    //准备一个返回的数组
    var results = []
    //遍历执行处理函数
    for (var i = 0; i < arr.length; i++) {
        //当里面返回的true填入到数组
        if (callback(arr[i], i, arr)) {
           results.push(arr[i])
        }
    }
    return results
}
console.log(myFilter(function(v){return v>2}))
```

reduce

```
//myReduce
function myReduce(callback, value) {
    if (typeof callback != 'function') {
        throw new Error('参数错误')
    }
    //value被传递的情况
    var index = 0
    var previous = value
    //如果value值没有被传递 那么我的遍历从下标1开始 初始值为对应的下标为0的元素
    if (typeof value == 'undefined') {
        //判断是否为空数组
        if(arr.length == 0){
            throw new Error('Reduce of empty array with no initial value')
        }
        index = 1
        previous = arr[0]
    }
    //遍历计算值
    for(;index<arr.length;index++){
        //调用传入的函数 得到对应的结果值 再覆盖对应的结果
```

```
        previous = callback(previous,arr[index],index,arr)
    }
    //返回结果
    return previous
}
console.log(myReduce(function(prev,current){
    return prev+current
},20))
```

this指向的改变

Function对象的方法

- **bind （返回一个函数 需要手动调用）**

```
var obj = {
    sayHi:function(){
        console.log(this) //默认指向obj这个对象
    }
}
function sayWorld(){
    console.log(this) //默认指向window
}
obj.sayHi() //打印obj对象
sayWorld() //打印window对象
//改变obj里面的sayHi的this指向
var fn =  obj.sayHi.bind(window)
console.log(fn); //使用bind是一个新的函数
console.log( obj.sayHi == fn); //false
//通过bind函数来更改this指向 这个bind函数会返回一个新的函数 这个函数里的this会指向你传入的对象
fn() //执行这个函数 打印window
```

- **apply（自动调用 返回值是对应的函数执行的结果）**

```
//apply 这个函数他也是函数对象的方法
    console.log(sayWorld.apply(obj)) //自动调用 更改this指向 指向对应的obj
```

- **call （自动调用 返回值也是对应的函数执行的结果）**

```
 //call 这个函数类似于apply 他也会自动调用
    sayWorld.call(document) //更改this指向为document 自动调用
```

传参方式对比

```
function sum(n,n1){
    console.log(n,n1)
    console.log(n+n1)
}
//bind传递参数 跟对应的函数执行传递参数是一样的
sum.bind(String)(2,3)
//apply 传递的参数是对应的数组(按照函数的形参顺序来传入的)
sum.apply(document,[1,2])
//call 传递的参数是一个个的元素
sum.call(document.body,1,2)
```

区别

- bind返回的是一个函数需要手动调用
- apply会自动调用返回的是对应的函数执行的结果（传递的参数是一个数组）
- call会自动调用返回的是对应的函数执行的结果（传递的参数是一个个的元素（多个参数））

bind函数执行完返回的函数不能被apply和call进行二次更改this指向

补充新增

- 字符串的trim方法 （去前后空格）
- 数组的静态方法 Array.isArray
- 数组的indexOf 及 lastIndexOf
- JSON的序列化方法及反序列化方法
- Object的相关方法

```
//静态方法就是首字母大写的类型(class类)名来调用的方法 (static修饰的)
console.log(Array.isArray([])) //判断是否为数组
//数组的补充方法 indexOf  lastIndexOf 新增的
console.log(['a','b','c'].indexOf('a'))//0 获取对应的第一个出现的下标
console.log(['a','b','c','a'].lastIndexOf('a'))//3 从后往前获取对应的第一个出现的
下标
console.log(['a','b','c'].indexOf('a',1))//-1 第二个指定是对应的开始下标找不到返
回-1
console.log(['a','b','c'].lastIndexOf('a',1))//0
//JSON的序列化方法及反序列化方法
// JSON.stringify()
// JSON.parse()
//Object 新增的方法 object.defineProperty 等
```

getter setter （对象里面）

```
//这个get方法名字必须是访问属性名
//get用于获取
//set用于设置
var obj = {
    _name: 'hello',
    //这个get方法名字必须是访问属性名 (无限递归 如果再get里面使用对应名字的内容就会继续调用
getter)
    get name() {
        console.log('getter 调用了')
        // return this.name + '张三' 出错无限递归
        return this._name + '张三'
    },
    //这个set方法名字必须是访问属性名
    set name(value) {
        this._name = value
        console.log('setter 调用了')
    }
}
console.log(obj.name) //调用getter
obj.name = 'hi'//调用setter
console.log(obj.name) //调用getter
```

ES6新增

字符串新增

字符串模板

```
var number = 100
var str = `你有${number}元`
```

字符串相关方法

- includes 是否包含
- startsWith 是否开头
- endsWith 是否结尾
- repeat 平铺（重复）

```
var str = 'abcdef'
//传入的检索的内容 传入的第二参数指定的下标
//检索是否包含
console.log(str.includes('abc',1)) //从下标1开始后面的内容是否包含abc
//检索是否开头
console.log(str.startsWith('abc')) //true
//是否结尾
console.log(str.endsWith('def')) //true
//repeat 平铺 传入对应的次数  返回一个新的字符串
console.log(str.repeat(3))
```

数组新增

相关方法

- find 查找对应的内容 （高阶函数）
- findIndex 查找对应的下标 （高阶函数）
- Array.of 将对应的内容填入Array 返回是数组
- Array.from 将伪数组转为数组
- fill 覆盖为一个值
- inculdes 是否包含

```
<form action="http://www.baidu.com"></form>
<form action="http://www.baidu.com"></form>
<form action="http://www.sohu.com"></form>
<script>
    //新增的静态方法 of将内容填入对应数组 返回一个新的数组
    var arr = Array.of(1,2,3)
    console.log(arr)
    //from 将对应的伪数组转为数组
    var arr = Array.from(document.forms).filter(function(v){
        return v.action.includes('baidu')
    }) //将这个伪数组变为数组
    console.log(arr)
    //find 查找方法 返回的是值 第一个的值 传入的函数返回一个boolean类型的值
    //查找不到返回undefined
    var v = [1,2,3,4].find(function(v,i,arr){
```

```
        return i>2
    })
    console.log(v) //4
    //findIndex 查找下标的方法 查找不到返回-1
    var i = [1,2,3,4].findIndex(function(v,i,arr){
        return v<4
    })
    console.log(i) //0
    console.log([1,23,4].fill('a')) //默认start 是0 对应的end为length [a,a,a]
    console.log([1,23,4].fill('a',1,2)) // 包含开始的 不包含的结束的 [1,a,4]
    console.log([1,2,3].includes(1))//true
</script>
```

find及findIndex 自定义实现

```
function find(callback){
    if (typeof callback != 'function') {
        throw new Error('参数错误')
    }
    for(var i=0;i<arr.length;i++){
        if(callback(arr[i],i,arr)){
            return arr[i]
        }
    }
}
//findIndex
function findIndex(callback){
    if (typeof callback != 'function') {
        throw new Error('参数错误')
    }
    for(var i=0;i<arr.length;i++){
        if(callback(arr[i],i,arr)){
            return i
        }
    }
    return -1
}
```

变量声明（块级作用域）

- let 声明普通变量
- const 声明常量

let关键词 （不允许重复声明 存在块级作用域）

```
//var 伪全局作用域 进行变量提升
for(var i=0;i<10;i++){
    setTimeout(function(){
        console.log(i) //打印10个10
    })
}
//使用块级作用域
for(let i=0;i<10;i++){
    setTimeout(function(){
        console.log(i) //打印0-9
    })
```

```
}
// var a = 1
// var a = 2
//数据混乱 为了避免let关键不允许重复声明
let a = 1 //在每个块变量是单独存在的不允许重复声明
```

const关键词 （声明必须赋值 地址不可变 对象里面的内容可以更改 存在块级作用域 不允许重复声明）

```
//const关键词 声明是常量 (不能改变的量 地址不可变)
//声明必须赋值
const n = 10
// n = 9 //报错 不可变
const obj = {
    name:'jack'
}
// obj = {} //错误的 更改了地址
obj.name = 'rose'
```

基础值类型新增

es3就存在的

- string
- number
- boolean
- null
- undefined

es6新增的

- bigInt 大的整型 （存在对应的存储不了的数据）
- symbol 独一无二数据 （没有相同的 机器码来实现的）

```
//产生一个bigInt类型
var bigInt =
BigInt('111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111')
console.log(bigInt)
console.log(11111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111)
//产生一个symbol类型
//Symbol 独一无二的类型 机器码
var sym = Symbol('hello')
console.log(sym)
console.log(sym.description)//获取描述
console.log(Symbol() == Symbol())
//一般被当成对象的属性 用于设计的 而不是用于操作的
console.log(window)
```

函数新增

- 默认参数 （指定参数的默认值）
- 箭头函数 （()=>{}）

默认参数

写法

```
function 函数名(参数=默认值,参数2=默认值){
}
```

示例

```
function sum(a,b){
    console.log(a+b)
}
sum(1,2) //3
sum(1) //NaN 不想打印NaN而想打印1
//按照之前的写法
function sum(a,b){
    if(!a){
        a = 0
    }
    if(!b){
        b= 0
    }
    console.log(a+b)
}
//参数越多判断就会越多 所以es6新增了对应的默认参数 通过在形参后赋值来指定默认参数
function sum(a=0,b=0){
    console.log(a+b)
}
console.log(sum()) //0
console.log(sum(1)) //1 如果传递的参数那么就会覆盖默认参数
console.log(sum(1,2)) //3 如果传递的参数那么就会覆盖默认参数
```

箭头函数

标准写法 （箭头函数是一个匿名函数）

```
var fn = (参数1,参数2)=>{
    代码块
}
```

简写

- 如果参数只有一个可以省略()

```
//简写 如果参数只有一个那么可以省略对应()
document.querySelector('button').onclick =e=>{
    console.log('点击了',e)
}
```

- 如果执行代码块只有一行那么可以省略{}

```
//如果对应的代码只有一行可以省略{}
document.querySelector('button').onclick =e=>console.log('点击了1',e)
```

- 如果只有一行需要返回值 省略了对应的{} 可以省略return

```
//如果对应的代码只有一行且需要返回值 可以省略return
// var fn = () =>{return 1}
var fn = () =>1
console.log(fn())
```

箭头函数的特性

- 没有this 没有arguments
- 箭头函数没有原型 （prototype） （不能被new）

字面量简写

- **属性简写 （属性值为变量 属性值和属性名名字一致的情况下）**
- **函数简写 （删除对应的:function）**

```
var name = '张三'
var age = 18
//原本写法
var obj = {
    name: name,
    age: age
}
//简写 属性值一定是变量 属性名和属性值一致
var obj = {
    name,
    age
}
//不支持 name:'name'这种简写 属性值不为变量
console.log(obj)
//原本写法
var obj = {
    sayHello:function(){
        console.log('hello')
    }
}
//对象中的函数简写
var obj = {
    sayHello(){
        console.log('hello')
    }
}
obj.sayHello()
```

解构赋值

解构的概述就是将对应的对象或者数组解除对应的构造暴露其中的内容。

- **对象的解构 （快速提取对象中的属性 根据属性名提取）**

  基础写法

  ```
  var {key,key1} = {key:value,key1:value1}
  ```

  示例

```
var obj = {name:'jack',age:18}
obj.name
ob.age
//简化写法 使用解构赋值的形式 直接获取里面的name和age
var {name,age} = {name:'jack',age:18}
console.log(name)
console.log(age)
```

- **数组的解构 （根据对应的顺序）**

  基础写法

```
var [变量名,变量名1...] = [值,值1]
```

  示例

```
//数组的解构是对应的顺序的
var [a,b,c] = [1,2,3]
console.log(a)
```

扩展运算符 ...

- 可以规定对应的参数不受限制 自动将对应的内容封装为一个数组

```
//接收不限制的参数
function sum(...args) {
    //自动将对应的数据组装成一个数组
    //args是一个数组
    console.log(args)
    //遍历args数组进行计算
    return args.reduce((prev,current)=>prev+current)
}
console.log(sum(1,2,3,4,5,1,2,1,2,1))
//数组分割符 , es5的
var arr = [,,,,,]
console.log(arr)
```

- 打开对应的数组

```
// 利用...打开对应的数组
var arr = [{name:'张三'},{name:'李四'}]
var newArr = [...arr]
console.log(arr,newArr,newArr == arr)
//俩个数组组成一个数组
var arr1 = [1,2,3]
var arr2 = [4,5,6]
var newArr = arr1.concat(arr2)
console.log(newArr)
var newArr = [...arr1,...arr2] //对应的方法传参里面自动添加,
console.log(...arr1)
```

- 打开对应的对象

```
//利用...来打开对象
var obj = {name:'jack',age:18}
var newObj = {...obj}
console.log(obj,newObj,newObj == obj)
```

generator函数

他是一个解决异步问题的一个函数 他可以将异步代码同步执行

声明方式

```
function* fn(){
    yield 代码片段
}
```

基础示例

```
function* fn(){
    //分成一个段 断点
    yield console.log(1)
    yield console.log(2)
    yield console.log(3)
    yield console.log(4)
    yield console.log(5)
}
var g = fn() //返回一个generator
var iter = g.return('a') //结束
console.log(iter) //done 是否完成 value返回的值
// g.throw() //抛出异常
// //next下一个
// g.next()
// g.next()
// g.next()
// g.next()
// g.next()
```

promise （es7）

他是一个解决异步问题的一个类 他可以将异步代码同步执行

ES6的模块化

模块化技术指代的是将对应的功能代码拆分为一个个的模块，完成对应的复用。

require.js的模块化（一个内容要导入必须先导出）

- export 导出

```
//基础导出 一个文件只有一个 默认导出 第一种
export default {
  name:'jack',
  sayHi(){
    console.log('hello')
  }
}
//导出变量的形式 第二种
export const name = 'tom'
export const sayHi = ()=>{console.log('hello')}
```

- **import 导入**

```
<!-- 支持es6的写法 -->
<script type="module">
    // import 名字 from 路径地址 默认导出的 导入的名字随便写
    // import a from './export.js' //第一种导入
    // a.sayHi()
    // console.log(a.name)
    // 导出多个的导入 第二种导入
    import {sayHi,name} from './export.js'
    sayHi()
    console.log(name)
</script>
```

重点： AMD和CMD的区别