# 1.初识es6

## ECMAScript6也就是ECMAScript2015

# 2.用let声明变量

## 好处：

### 1.块级作用域的限制，用var在一个代码块定义的变量，在代码块的外面也能访问到，用let就不会

|  |
| --- |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**let声明变量**</title> </head> <body> <script>  {  var *x* = 100  }  *alert*(*x*) </script> </body> </html> |
|  |
|  |

#### 可见用let在代码块里面定义的变量外面拿不到，同理在if，else和for循环等等语句里面也是一样的

|  |
| --- |
|  |
|  |
|  |
|  |

#### 注意.函数的情况和上面不一样，函数是局部作用域的

#### 小案例：我们创建一个ul里面有3个li我们想给每一个li添加点击事件，当对应的li被点击，就在控制台输出他的index，我们先用var，发现是不行的

|  |
| --- |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**let声明变量**</title> </head> <body> <ul>  <li>**11**</li>  <li>**22**</li>  <li>**33**</li> </ul> <script>  var *lis* = *document*.querySelectorAll("li")  for(var *i*=0;*i*<*lis*.length;*i*++){  *lis*[*i*].onclick = function () {  *console*.log(i)  }  }  </script> </body> </html> |
|  |
|  |
|  |

##### 原因是for循环执行快，而onclick是异步的，所以得不到正确效果、

#### 可以这么解决

|  |
| --- |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**let声明变量**</title> </head> <body> <ul>  <li>**11**</li>  <li>**22**</li>  <li>**33**</li> </ul> <script>  var *lis* = *document*.querySelectorAll("li")  for(var *i*=0;*i*<*lis*.length;*i*++){  *lis*[*i*].index = *i  lis*[*i*].onclick = function () {  *console*.log( this.index)  }  }  </script> </body> </html> |
|  |

##### 可以解决这个问题

#### 我们把代码改为let

|  |
| --- |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**let声明变量**</title> </head> <body> <ul>  <li>**11**</li>  <li>**22**</li>  <li>**33**</li> </ul> <script>  // {  // // var x = 100  // let x = 100  // }  // alert(x)  var *lis* = *document*.querySelectorAll("li")  for(let i=0;i<*lis*.length;i++){  *lis*[i].onclick = function () {  *console*.log(i)  }  }  </script> </body> </html> |
|  |

##### 发现可以实现这个效果

### 2.不允许用let声明同一个变量2次，而var是可以的

|  |
| --- |
|  |
|  |
|  |
|  |

#### 即使前面用var定义了一个变量a，后面也不能用let定义a

|  |
| --- |
|  |
|  |

### 3.不允许变量提升

#### 什么是变量提升？

|  |
| --- |
| **通俗来说，变量提升是指在 JavaScript 代码执行过程中，JavaScript 引擎把变量的声明部分和函数的声明部分提升到代码开头的行为。变量被提升后，会给变量设置默认值为 undefined。** 正是由于 JavaScript 存在变量提升这种特性，导致了很多与直觉不太相符的代码，这也是 JavaScript 的一个设计缺陷。虽然 ECMAScript6 已经通过引入块级作用域并配合使用 let、const 关键字，避开了这种设计缺陷，但是由于 JavaScript 需要向下兼容，所以变量提升在很长时间内还会继续存在。也就是变量提升允许先使用变量再定义变量。 |
| **这样子会报错** |
|  |
| **只要有变量定义，即使在后面也不会报错** |
|  |

#### 使用let就不允许这种情况发生

|  |
| --- |
|  |
|  |

#### 下面的代码也是会报错的

|  |
| --- |
|  |
|  |

### 4.不与顶层对象挂钩

#### 用var定义的变量可以使用window对象来获取

|  |
| --- |
|  |
|  |

#### 使用let后

|  |
| --- |
|  |
|  |

# 3.const声明常量

## 1）声明一个常量必须同时给他赋值，否则报错，

|  |
| --- |
| 上面的const myname的写法是错误的，因为它没有赋值 |
|  |

## 2）声明一个常量字符串然后尝试修改它的值就会报错，如下图

|  |
| --- |
|  |
|  |

## 3）和let一样。用const定义的常量不能重复

## 4）和let一样const定义的常量的块作用域的，在代码块外面访问不到

|  |
| --- |
|  |
|  |

## 5）和let一样，const也是没有变量提升功能的，会有一个暂存性死区

|  |
| --- |
|  |
|  |

## 6）和let一样const也是不与顶层对象如window挂钩

|  |
| --- |
|  |
|  |

## 7）必须要注意的是：如果一个指向对象的变量声明为const，只是他不能指向别的对象内存地址，它所指向的对象的属性还是可以修改的

|  |
| --- |
|  |
|  |
| const *obj* ={  name:"kenny",  age:45,  money:999999999999999999999999999999999.00  }  // obj = { //obj不能再指向别的对象  // name:"Benny",  // age:45,  // money:0  // }  //但是obj的属性可以修改  *obj*.name = "Kenny guo cai"  *console*.log(*obj*) </script> |
|  |

### 同理，const修饰的变量指向一个数组，只是他以后不能再指向别的地址，这个数组是可以修改的

## 8）如果想连const变量指向的对象的属性都不允许修改，必须使用Object.freeze方法

|  |
| --- |
|  |
| **不报错但是没有修改** |

### 注意：如果对象的属性也是一个复杂类型如对象或者数组，那freeze就无能为力了，它只能冻结一层

# 4.es6变量解构赋值

## 1）数组的解构

|  |
| --- |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**解构表达式**</title> </head> <body> <script>  //数组的解构使用[]  let *objs* = [  {  name:"Timmy",  age:18,  },  {  name:"Gimmy",  age:19,  },  {  name:"Jammy",  age:17,  }  ]  let [*o1*,*o2*,*o3*] = *objs  console*.log(*o1*)  *console*.log(*o2*)  *console*.log(*o3*) </script> </body> </html> |
|  |

### 小案例，利用解构表达式来交换两个变量的值，需要分号

|  |
| --- |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**解构表达式**</title> </head> <body> <script>  let *obj* = {  name:"Jimmy",  age:18,  gender:'Male',  email:"Jimmy123@gmail.com"  }   // //解构表达式  // let {name,age,gender,email} = obj ////注意：js的对象解构表达式用{}，数组的解构用[]，kotlin的解构表达式用()  // console.log(name)  // console.log(age)  // console.log(gender)  // console.log(email)   // //用数组来存储对象的属性值  // let arr = []  // for(key in obj){  // arr.push(obj[key])  // }  // console.log(arr)  // //方式2  // let arr2 = []  // for(let key in obj){  // let obj2={}  // obj2[key] = obj[key]  // arr2.push(obj2)  // }  // console.log(arr2)   // let objs = [  // {  // name:"Timmy",  // age:18,  // },  // {  // name:"Gimmy",  // age:19,  // },  // {  // name:"Jammy",  // age:17,  // }  // ]  // let [o1,o2,o3] = objs  // console.log(o1)  // console.log(o2)  // console.log(o3)   //利用解构表达式来交换两个变量的值，需要分号  let *a* = 10;  let *b* = 20; //需要分号，不然报错  *console*.log(*a*,*b*);//需要分号，不然报错   [*b*,*a*] = [*a*,*b*];//需要分号，不然报错   *console*.log(*a*,*b*)  </script> </body> </html> |
|  |

### 如果只需要一个值怎么解构，不需要的直接留空，但是逗号不能少

|  |
| --- |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**解构表达式**</title> </head> <body> <script>  let *obj* = {  name:"Jimmy",  age:18,  gender:'Male',  email:"Jimmy123@gmail.com"  }   // //解构表达式  // let {name,age,gender,email} = obj ////注意：js的对象解构表达式用{}，数组的解构用[]，kotlin的解构表达式用()  // console.log(name)  // console.log(age)  // console.log(gender)  // console.log(email)   // //用数组来存储对象的属性值  // let arr = []  // for(key in obj){  // arr.push(obj[key])  // }  // console.log(arr)  // //方式2  // let arr2 = []  // for(let key in obj){  // let obj2={}  // obj2[key] = obj[key]  // arr2.push(obj2)  // }  // console.log(arr2)   // let objs = [  // {  // name:"Timmy",  // age:18,  // },  // {  // name:"Gimmy",  // age:19,  // },  // {  // name:"Jammy",  // age:17,  // }  // ]  // let [o1,o2,o3] = objs  // console.log(o1)  // console.log(o2)  // console.log(o3)   // //利用解构表达式来交换两个变量的值，需要分号  // let a = 10;  // let b = 20; //需要分号，不然报错  // console.log(a,b);//需要分号，不然报错  //  // [b,a] = [a,b];//需要分号，不然报错  //  // console.log(a,b)   //只想用解构表达式获取一个值,不需要的直接留空，但是逗号不能少  let *nums* = [50,60,100]  let [,,*a*] = *nums  console*.log(*a*)  </script> </body> </html> |
|  |

### 嵌套数组的解构

|  |
| --- |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**解构表达式**</title> </head> <body> <script>  // //只想用解构表达式获取一个值,不需要的直接留空，但是逗号不能少  // let nums = [50,60,100]  // let [,,a] = nums  // console.log(a) //嵌套数组的解构  let *myarr*=[1,[11,22,33],2]  let [*a*,[*x*,*y*,*z*],*b*] = *myarr  console*.log(*x*) //11  </script> </body> </html> |
|  |

### 可以给解构表达式的变量赋默认值以防万一拿不到数据返回undefined。

|  |
| --- |
|  |
|  |
|  |
|  |

|  |
| --- |
|  |
|  |

## 2）对象的解构赋值

|  |
| --- |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**解构表达式**</title> </head> <body> <script>  let *obj* = {  name:"Jimmy",  age:18,  gender:'Male',  email:"Jimmy123@gmail.com"  }   //对象解构表达式  let {*name*,*age*,*gender*,*email*} = *obj* //注意：js的对象解构表达式用{}，数组的解构用[],kotlin的解构表达式用()  *console*.log(*name*)  *console*.log(*age*)  *console*.log(*gender*)  *console*.log(*email*)  </script> </body> </html> |
|  |

### 注意：由于对象的属性是key：value的键值对所以，解构表达式的变量的顺序可以和对象属性顺序不一样

|  |
| --- |
|  |
|  |

### 对象的解构更加灵活，你不一定非得获取需要属性，你需要那个属性就取那个属性

|  |
| --- |
| <script>  let *obj* = {  name:"Jimmy",  age:18,  gender:'Male',  email:"Jimmy123@gmail.com"  }   // //解构表达式  // let {email,age,gender,name} = obj  let {*email*} = *obj*   *console*.log(*email*)  </script> |
|  |
| <script>  let *obj* = {  name:"Jimmy",  age:18,  gender:'Male',  email:"Jimmy123@gmail.com"  }   // //解构表达式  let {*age*} = *obj*  *console*.log(*age*)  </script> |
|  |

### 给对象解构的时候还可以给属性取别名，需要使用冒号

|  |
| --- |
| <script>  let *obj* = {  name:"Jimmy",  age:18,  gender:'Male',  email:"Jimmy123@gmail.com"  }   // //解构表达式  // let {email,age,gender,name} = obj  // let {email} = obj  let {*age*} = *obj* let {gender:*sex*} = *obj//解构时候给属性取别名*   *console*.log(*age*,*sex*)  </script> |
|  |

### 这种方法可以解决命名冲突问题，

### 也可以给解构表达式的变量赋初值，以防万一拿不到值

|  |
| --- |
| <script>  let *obj* = {  name:"Jimmy",  age:18,  gender:'Male',  email:"Jimmy123@gmail.com"  }   // //解构表达式  let {*age*} = *obj* let {gender:*sex*} = *obj* let {*phone*="没有该属性"} = *obj*   *console*.log(*age*,*sex*,*phone*)  </script> |
|  |

## 3）嵌套对象的解构

|  |
| --- |
|  |
|  |

### 注意：建议不要对嵌套超过2级的对象或者数组进行直接解构，这样子很复杂，容易出错

## 4）也可以在以对象或者数组作为参数的函数的实参里面进行结构

|  |
| --- |
|  |

### 也可以嵌套结构

|  |
| --- |
|  |

### 小扩展，用数组来保存对象的属性

|  |
| --- |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**解构表达式**</title> </head> <body> <script>  let *obj* = {  name:"Jimmy",  age:18,  gender:'Male',  email:"Jimmy123@gmail.com"  }   //用数组来存储对象的属性值  let *arr* = []  for(key in *obj*){  *arr*.push(*obj*[key])  }  *console*.log(*arr*)  //方式2  let *arr2* = []  for(let key in *obj*){  let obj2={}  obj2[key] = *obj*[key]  *arr2*.push(obj2)  }  *console*.log(*arr2*) </script> </body> </html> |
|  |

## 5）字符串的解构赋值，用的很少

|  |
| --- |
|  |

### 可以用解构表达式来获取字符串的长度

|  |
| --- |
|  |
|  |

### 还可以取别名

|  |
| --- |
|  |
|  |

# 5.es6模板字符串

|  |
| --- |
|  |

### 小案例

|  |
| --- |
|  |

## 自己写的案例

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <title>Title</title>  <style>  .active:hover{  background: #ccc;  color: deeppink;  border: 1px solid green;  }  </style>  </head>  <body>  <ul>  </ul>  <script>  function doClick(item) {  // if(item.className === "active"){  // item.className = ''  // } else{  // item.className = "active"  // }  // alert(item.innerHTML)  }  let oul = document.querySelector("ul")  let arr = ["Java","cpp","kotlin","python"]  let newArr = arr.map(function (item,index) {  return `<li class=”active”>  <b>${item}</b>  </li>`  })  oul.innerHTML = newArr.join(" ")  // arr.forEach(item=>{  // oul.appendChild(document.createElement(item))  // })  </script>  </body>  </html> |

## 还可以这么写

|  |
| --- |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**Title**</title>  <style>  .active{  background: #ccc;  color: **deeppink**;  border: 1**px solid green**;  }   </style> </head> <body> <ul>  </ul> <script>  let *selected* = []  function *doClick*(item) {  if(*selected*.length === 0){  item.className = 'active'  *selected*.push(item)  } else {  *selected*.forEach(item=>item.className ='')  *selected* = []  item.className = 'active'  *selected*.push(item)  }  }  let *oul* = *document*.querySelector("ul")  let *arr* = ["Java","cpp","kotlin","python"]  let *newArr* = *arr*.map(function (item,index) {  return `<li onclick="doClick(this)" >  <b>${item}</b>  </li>`  })  *oul*.innerHTML = *newArr*.join(" ")  // arr.forEach(item=>{  // oul.appendChild(document.createElement(item))  // }) </script> </body> </html> |

## 实现效果是只有当前点击的那一项会有样式

|  |
| --- |
|  |
|  |

# 6.es6字符串与数值扩展

## 字符串扩展

|  |
| --- |
|  |

### 注意上面的函数还有第二个参数，表示从那个索引开始作为源字符串

|  |
| --- |
| let *myname* = "kenny"  //有第二个参数的用法 *console*.log(*myname*.includes("k",1)) //false,表示从索引为1的位置往后看 *console*.log(*myname*.startsWith("k",1)) //false,表示从索引为1的位置往后看 *console*.log(*myname*.endsWith("e",2)) //true，和上面不一样，表示从索引为2的位置往前看 |

### 关于repeat函数的特殊用法

|  |
| --- |
| let *myname* = "kenny"  *console*.log(*myname*.repeat(3)) //kennykennykenny *console*.log(*myname*.repeat(0)) //空,这个比较特别，repeat0个就是一个都不输出 *console*.log(*myname*.repeat(2.2))//kennykenny，遇到小数会自动截取他的整数部分抛弃小数部分 *console*.log(*myname*.repeat('3'))//kennykennykenny 字符串数字也有效  *console*.log(*myname*.repeat('aaa')) //不会输出，因为这不是有效的数字 |

## 数字扩展

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

### 注意：Number的这些方法window顶层对象也有（也就是不需要Number直接使用方法名来调用），他们和这些方法的不同之处的他们会尝试就字符串转化为数字而Number的方法不会这么做

# 7.数组的扩展

|  |
| --- |
|  |
|  |
|  |

## 1.扩展运算符...

### 以前，我们是这样子复制数组的

|  |
| --- |
|  |

### concat函数是深拷贝，修改副本不会改变原来的数组

|  |
| --- |
| let *nums* = [10,20,30] let *nums2* = *nums*.concat() *console*.log(*nums2*) |
|  |

### 现在可以这么做

|  |
| --- |
|  |

使用let *arr3* = [...*arr1*]这种方法进行的也是深拷贝，修改副本不会影响原来的数组，也可以合并拷贝，如上面的let arr = [...arr1,...arr2]

#### 注意：这种方法只适合只有一层的数组，如果有嵌套数组，里面的嵌套数组是浅拷贝！！！

### 扩展运算符还可以和解构表达式一起使用，可以表示打包赋值的意思

|  |
| --- |
|  |

### 注意：这种写法，...只能放在最后

## 2.Array.from()方法，是Array类的静态方法。把一个类数组对象转换为数组，例如函数里面的arguments对象

|  |
| --- |
|  |

### 使用document的querySelectorAll方法获取的元素集合他也不是数组，他是一个节点列表，没有数组的方法，

|  |
| --- |
|  |
|  |

### 此时如果你想对这个列表做一些数组才有的方法，就需要使用Array.from方法把这个列表转化为数组

|  |
| --- |
| <ul>  <li>**Java**</li>  <li>**Cpp**</li>  <li>**Go**</li>  <li>**Kotlin**</li> </ul>  <script>  let *lis* = *document*.querySelectorAll("li")  *console*.log(*lis*)  //转换  let *arrLis* = *Array*.from(*lis*);  *console*.log(*arrLis*)  *arrLis*.map(item=> item.style.color="pink") </script> |
|  |

## 3.Array.of,将数值或者类数组对象转换为数组，是Array类的静态方法

|  |
| --- |
|  |
| //3.Array.of,将数值转化为数组,其实就是新建数组 let *arrnum* = *Array*.of(1,2,3,4,5) *console*.log(*arrnum*); // [1,2,3,4,5] //将前面的lis集合转化为数组 let *lisArr* = *Array*.of(...*lis*) *console*.log(*lisArr*) // [li, li, li, li] |

## 4.find ,findindex,是数组的实例对象方法，只返回符合条件的第一个结果

|  |
| --- |
| //4.find ,findindex,是数组的实例对象方法 let *texts* = ['Java',"javascript","python","golang","kotlin"] let *res* = *texts*.find(item=>item.startsWith('j')) //find方法是区分大小写的 *console*.log(*res*) //javascript *res* = *texts*.find(item=>!item.startsWith('j')) //find方法是区分大小写的 *console*.log(*res*) //Java,find方法只要找到一个就会返回 *res* = *texts*.findIndex(item=>item.startsWith('j')) *console*.log(*res*);//1 *res* = *texts*.findIndex(item=>!item.startsWith('j')) *console*.log(*res*);//0,findIndex方法和find方法类似，只是他返回一个元素的索引 |

## 5.findLast，findLastIndex，也查找只不过从后往前找，用法类似，es2022才有

|  |
| --- |
| let *num* = [11,12,13,14,15] *console*.log(*num*.findLast(item=>item > 13)) //15 从后往前，第一个符合条件的是15 *console*.log(*num*.findLastIndex(item=>item > 13)) //4 第一个符合条件的是下标为4的元素 |
|  |

## 6.fill方法是数组的实例方法，用来填充一个数组，用处不大。了解即可

|  |
| --- |
|  |
| //6.fill let *data* = *Array*(3).fill("hello") *console*.log(*data*);// ['hello', 'hello', 'hello'] *data*.fill("kenny",1,2) *console*.log(*data*)// ['hello', 'kenny', 'hello'] |

## 7.flat 和flatMap，数组扁平化，是数组的实例方法

|  |
| --- |
| ////7.扁平化flat和flatMap let *sj* = [1,2,3,[4,5,6]] let *flatSj* = *sj*.flat() *console*.log(*flatSj*);//[1, 2, 3, 4, 5, 6]  let *citiesGroup* = [  {  type:"A",  cities:['安庆','安阳','鞍山']  },  {  type:"B",  cities: ['北京','包头','保定']  }  ]  *console*.log(*citiesGroup*.flat()) //对于一个复杂的对象数组使用flat方法没有什么效果 //像上面的情况，需要使用flatMap方法里面传递一个函数作为参数 // res = citiesGroup.flatMap(function (item) { // return item.cities // }) *res* = *citiesGroup*.flatMap(item => item.cities) *console*.log(*res*) //['安庆', '安阳', '鞍山', '北京', '包头', '保定'] //可见flatMap可以把一个对象数组变为一个普通数组 |

### 本节小结：

#### 重要方法是：...扩展运算符，Array.from()和flat和flatMap，剩下的方法了解即可

### 本节完整实例代码

|  |
| --- |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**数组扩展**</title> </head> <body> <ul>  <li>**Java**</li>  <li>**Cpp**</li>  <li>**Go**</li>  <li>**Kotlin**</li> </ul>  <script>  //数组复制可以使用concat函数，是深拷贝，修改副本不会影响原来的数组  let *nums* = [10,20,30]  let *nums2* = *nums*.concat()  *console*.log(*nums2*)  //1.扩展运算符...  //1).扩展运算符可以用来复制数组  let *arr1* = [7,8,9]  let *arr3* = [...*arr1*]  *console*.log(*arr3*)  *arr3*.pop()  *console*.log(*arr3*)  *console*.log(*arr1*)  // 2),数组扩展符...,可以用来合并数组  let *arr1* = [1,2,3]  let *arr2* = [4,5,6]  let *arr* = [...*arr1*,...*arr2*] //[1, 2, 3, 4, 5, 6]  *console*.log(*arr*)  //3)...和解构表达式一起使用  let *myarr* = [1,2,3,4,5,6,7,8]  let [*a*,*b*,*c*] = *myarr  console*.log(*a*,*b*,*c*) //1,2,3  let [*x*,*y*,...*z*] = *myarr* //把第一个赋值给x,把第二个赋值给y,剩下的全部打包给z，也就是z变为一个数组:他的值是[3,4,5,6,7,8]  *console*.log(*x*,*y*,*z*)  //2.Array.from,将类数组对象转换为数组,是Array类的静态方法  function *test*(){  *console*.log(arguments) //arguments和数组有类似的地方可以用下标来获取元素，也可以用for循环遍历但是它没有数组的map，filter等等方法  //转换  let args = *Array*.from(arguments)  *console*.log(args); //[1, 2, 3, 5]  //然后这个args就是一个数组，可以使用数组的方法如map  *console*.log(args.map(item=>item \* 2)) //[2, 4, 6, 10]  //调用reduce方法  let sum = 0  sum = args.reduce((sum,current)=>sum += current,0) //rdeuce方法需要接收返回值  *console*.log("sum=",sum)  }  *test*(1,2,3,5)   let *lis* = *document*.querySelectorAll("li")  *console*.log(*lis*)  //转换  let *arrLis* = *Array*.from(*lis*);  *console*.log(*arrLis*)  *arrLis*.map(item=> item.style.color="pink")  //3.Array.of,将数值转化为数组,其实就是新建数组,是Array类的静态方法  let *arrnum* = *Array*.of(1,2,3,4,5)  *console*.log(*arrnum*); // [1,2,3,4,5]  //将前面的lis集合转化为数组  // let lisArr = Array.of(...lis) //直接生成一个有4个li是数组，有4个元素的  // console.log(lisArr) // [li, li, li, li]  let *lisArr* = *Array*.of(*lis*) //这个和上面用...lis是不一样的，上面直接生成一个有4个li是数组，有4个元素的  *console*.log(*lisArr*) //这里生成的是注意一个NodeList集合作为成员的数组，数组只有一个元素，Nodelist对象有4个孩子，下面我们测试一下  *console*.log(*lisArr*[0])//NodeList(4)[li, li, li, li]  *console*.log(*lisArr*[1]) //undefine,声明这个数组的确只有一个集合作为元素   let *texts* = ['Java',"javascript","python","golang","kotlin"]  let *res* //4.find ,findindex,是数组的实例对象方法  *res* = *texts*.find(item=>item.startsWith('j')) //find方法是区分大小写的  *console*.log(*res*) //javascript  *res* = *texts*.find(item=>!item.startsWith('j')) //find方法是区分大小写的  *console*.log(*res*) //Java,find方法只要找到一个就会返回  *res* = *texts*.findIndex(item=>item.startsWith('j'))  *console*.log(*res*);//1  *res* = *texts*.findIndex(item=>!item.startsWith('j'))  *console*.log(*res*);//0,findIndex方法和find方法类似，只是他返回一个元素的索引  //5 findLast,findLastIndex,数组的实例对象方法，从后往前查找,es2022才有，es6没有  let *num* = [11,12,13,14,15]  *console*.log(*num*.findLast(item=>item > 13)) //15 从后往前，第一个符合条件的是15  *console*.log(*num*.findLastIndex(item=>item > 13)) //4 第一个符合条件的是下标为4的元素  //6.fill  let *data* = *Array*(3).fill("hello")  *console*.log(*data*);  *data*.fill("kenny",1,2)  *console*.log(*data*)  ////7.扁平化flat和flatMap  let *sj* = [1,2,3,[4,5,6]]  let *flatSj* = *sj*.flat()  *console*.log(*flatSj*);//[1, 2, 3, 4, 5, 6]   let *citiesGroup* = [  {  type:"A",  cities:['安庆','安阳','鞍山']  },  {  type:"B",  cities: ['北京','包头','保定']  }   ]   *console*.log(*citiesGroup*.flat()) //对于一个复杂的对象数组使用flat方法没有什么效果  //像上面的情况，需要使用flatMap方法里面传递一个函数作为参数  // res = citiesGroup.flatMap(function (item) {  // return item.cities  // })  *res* = *citiesGroup*.flatMap(item => item.cities)  *console*.log(*res*) //['安庆', '安阳', '鞍山', '北京', '包头', '保定']  //可见flatMap可以把一个对象数组变为一个普通数组  </script>  </body> </html> |

# 8.对象扩展

## 1）对象简写，假如我们有一个对象

|  |
| --- |
| let *name* = "My Object" let *obj* = {  name:*name*,  test1:function () {  *console*.log("inside function1")  },  test2:function () {  *console*.log("inside function2")  } } |

### 我们可以把它简写为

|  |
| --- |
| let *obj2* = {  *name*,  test1(){  *console*.log("inside function1")  },  test2(){  *console*.log("inside function2")  } } |

### 把他们输出

|  |
| --- |
| **console.log(obj,obj2)** |

### 效果是一模一样的

|  |
| --- |
|  |

## 2.对象属性表达式，可以用来动态修改对象属性

|  |
| --- |
|  |
|  |

## 3.对象扩展运算符...

|  |
| --- |
| let *o1* = {  name:"Betty",  gender:"Female",  age:18 } let *o2* = {  ...*o1*, //把o1的属性一个一个拆开然后作为o2对象的属性，下面是o2对象的特有属性  email:"betty1234@hotmail.com",  address:"3 orange street,downtown" } *console*.log(*o2*) |
|  |

### 也是深拷贝，修改副本不影响原来的对象

### ...扩展符号也可以用来合并对象

|  |
| --- |
| let *oj1*= {  name:"Brandy",  gender:"Female",  age:19 } let *oj2* = {  email:"Brandysexy@hotmail.com",  address:"5 orange street,downtown" } let *oj* = {...*oj1*,...*oj2*} *console*.log(*oj*) |
|  |

#### 注意：如果后面的对象有属性和前面的对象的属性一样，后面的属性会覆盖前面的同名属性

### 在es6中也有Object.assign方法也可以完成上面功能

|  |
| --- |
| let *oj1*= {  name:"Brandy",  gender:"Female",  age:19 } let *oj2* = {  email:"Brandysexy@hotmail.com",  address:"5 orange street,downtown" }  let *oj* = *Object*.assign(*oj1*,*oj2*) *console*.log(*oj*) |
|  |

### 这个方法有一个问题，就是会改变第一个参数的值，解决办法也很简单，传递一个空对象给第一个参数即可

|  |
| --- |
| let *oj1*= {  name:"Brandy",  gender:"Female",  age:19 } let *oj2* = {  email:"Brandysexy@hotmail.com",  address:"5 orange street,downtown" }  let *oj* = *Object*.assign({},*oj1*,*oj2*) *console*.log(*oj*) |
|  |

### 这样子就不会改变第一个参数的值

### 其实有了...就不需要使用这个方法了

### Object还有一个is方法他可以判断用==或者===不能判断的情况

#### 如：

|  |
| --- |
| *console*.log(*NaN* == *NaN*) //false *console*.log(*NaN* === *NaN*) //false *console*.log(*Object*.is(*NaN*, *NaN*)); //true |

#### 这个方法的好处是，万一的的项目符合了NaN你用==或者是===无法得知，用Object.is方法可以得知,这在实际开发中很有用，Object.is认为+0不等于-0，但是===认为他们相等

### 本节完整代码

|  |
| --- |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**对象扩展**</title> </head> <body> <script>  //对象扩展  //1.对象简写  let *name* = "My Object"  let *obj* = {  name:*name*,  test1:function () {  *console*.log("inside function1")  },  test2:function () {  *console*.log("inside function2")  }  } // //简写  let *obj2* = {  *name*,  test1(){  *console*.log("inside function1")  },  test2(){  *console*.log("inside function2")  }  } // console.log(obj,obj2)  //2.对象属性表达式，用外面的变量来替换对象的key  let *hername*="Name"  let *obj1* = {  [*hername*]:"Jackline" //会变为 Name:"Jackline",也就是在对象里面，可以使用[]来获取外面变量的值作为对象的key  }  *console*.log(*obj1*); //{Name: 'Jackline'} //3.对象扩展运算符...  let *o1* = {  name:"Betty",  gender:"Female",  age:18  }  let *o2* = {  ...*o1*, //把o1的属性一个一个拆开然后作为o2对象的属性，下面是o2对象的特有属性  email:"betty1234@hotmail.com",  address:"3 orange street,downtown"  }  *console*.log(*o2*)  //...也可以用来合并对象  let *oj1*= {  name:"Brandy",  gender:"Female",  age:19  }  let *oj2* = {  email:"Brandysexy@hotmail.com",  address:"5 orange street,downtown"  }  // let oj = {...oj1,...oj2}  let *oj* = *Object*.assign({},*oj1*,*oj2*)  *console*.log(*oj*)  *console*.log(*NaN* == *NaN*) //false  *console*.log(*NaN* === *NaN*) //false  *console*.log(*Object*.is(*NaN*, *NaN*)); //true  *console*.log(*Object*.is(*parseInt*("ken"), *NaN*)); //true,不能将字母字符串转化为数字，他一定是NaN  </script> </body> </html> |

# 9.函数扩展

## 1）函数的默认参数

|  |
| --- |
| //1.参数默认值 function *myfun*(name="kerwin",age=19) {  //Hello,my name is kerwin,i am 19 years old  *console*.log(`Hello,my name is ${name},i am ${age} years old`)  } *myfun*() *myfun*("Jackline",30) //Hello,my name is Jackline,i am 30 years old |
|  |

## 2）函数的剩余参数rest，也是用...打包接收参数

|  |
| --- |
| //2.函数的剩余参数...打包 function *argsFun*(...args) {  *console*.log(args) //此时args=[1, 2, 3, 4, 5] } *argsFun*(1,2,3,4,5) //[1, 2, 3, 4, 5],他其实是打包成为一个数组 function *argsFun2*(a,b,...args) { //注意可变参数一定要写最后  *console*.log(args) //此时a=1,b=2,args=[3,4,5] } *argsFun2*(1,2,3,4,5) //[3, 4, 5] |
|  |

## 3）函数的name属性

|  |
| --- |
| function *argsFun2*(a,b,...args) { //注意可变参数一定要写最后  *console*.log(args) //此时a=1,b=2,args=[3,4,5] } *argsFun2*(1,2,3,4,5) //[3, 4, 5] //3.函数的name属性 *console*.log(*argsFun2*.name) //argsFun2 |
|  |

## 4）箭头函数

|  |
| --- |
| <script>  //4.箭头函数  let *objArr*=[  {  name:"Tracy",  age:16,  gender:'female'  },  {  name:"Mark",  age:18,  gender:'male'  },  {  name:"Lily",  age:19,  gender:'female'  },  {  name:"Tony",  age:15,  gender:'male'  },  {  name:"Bobby",  age:30,  gender:'male'  },  ]  let *getYoungest* = ()=>{  let min = *objArr*[0].age  for (let i = 0; i < *objArr*.length; i++) {  if(*objArr*[i].age < min){  min = *objArr*[i].age  }  }  return *objArr*.filter(item=>item.age == min)  }  *console*.log(*getYoungest*()); </script> |
|  |

### 如果箭头函数只有一行，可以省去{}和return语句，他一样可以返回数值

|  |
| --- |
|  |

### 注意，如果箭头函数多于一行，返回值的时候一定要使用return语句

### 还有，如果一个箭头函数需要返回一个对象，需要用一对()把对象包裹起来，否则return和{}都不能省略

|  |
| --- |
|  |

### 如果箭头函数只有一个参数，可以省略小括号

|  |
| --- |
|  |

### 注意，箭头函数不能访问普通函数的内建对象arguments还有箭头函数本身没有this对象

|  |
| --- |
|  |
|  |

### 也无法作为构造函数来new对象

|  |
| --- |
|  |
|  |

### 箭头函数的this其实是执行父作用域的

#### 先看一个小案例

|  |
| --- |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**箭头函数的this**</title> </head> <body> **模糊搜索：** <input type**="text"** id**="search"**> <script>  let *searchInput* = *document*.querySelector("#search")  // console.log(searchInput)  *searchInput*.oninput = function() {  *console*.log(this.value)  } </script> </body> </html> |
|  |

#### 可以看到此时程序可以正常运行

### 我们把程序修改一下

|  |
| --- |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**箭头函数的this**</title> </head> <body> **模糊搜索：** <input type**="text"** id**="search"**> <script>  let *searchInput* = *document*.querySelector("#search")  // console.log(searchInput)  *searchInput*.oninput = function() {  // console.log(this.value)  *setTimeout*(function() {  *console*.log(`发送${this.value}到后端，获取数据列表`)  },1000)  } </script> </body> </html> |
|  |

#### 我们发现，拿不到数据，因为此时this的指向有问题，他指向window顶层对象而不是当前的input对象

### 解决办法

|  |
| --- |
| <script>  let *searchInput* = *document*.querySelector("#search")  // console.log(searchInput)  *searchInput*.oninput = function() {  // console.log(this.value)  let \_this = this //把input对象保存起来  *setTimeout*(function() {  *console*.log(`发送${\_this.value}到后端，获取数据列表`)  },1000)  } </script> |
|  |

### 这个问题也可以用箭头函数来解决

|  |
| --- |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**箭头函数的this**</title> </head> <body> **模糊搜索：** <input type**="text"** id**="search"**> <script>  let *searchInput* = *document*.querySelector("#search")  // console.log(searchInput)  *searchInput*.oninput = function() {  //用箭头函数  *setTimeout*(() =>{  *console*.log(`发送${this.value}到后端，获取数据列表`)  },1000)   } </script> </body> </html> |
|  |

### 注意：箭头函数还是不能乱改，一定要注意，改错了地方，也是不能拿到正确结果的

### 结论：对于一些事件绑定函数还是使用普通函数好，因为普通函数的this是谁调用我，我就指向谁

### 本节代码

|  |
| --- |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**函数扩展**</title> </head> <body> <script>  //1.参数默认值  function *myfun*(name="kerwin",age=19) {  *console*.log(`Hello,my name is ${name},i am ${age} years old`) //Hello,my name is kerwin,i am 19 years old  }  *myfun*()  *myfun*("Jackline",30) //Hello,my name is Jackline,i am 30 years old  //2.函数的剩余参数...打包  function *argsFun*(...args) {  *console*.log(args) //此时args=[1, 2, 3, 4, 5]  }  *argsFun*(1,2,3,4,5) //[1, 2, 3, 4, 5],他其实是打包成为一个数组  function *argsFun2*(a,b,...args) { //注意可变参数一定要写最后  *console*.log(args) //此时a=1,b=2,args=[3,4,5]  }  *argsFun2*(1,2,3,4,5) //[3, 4, 5]  //3.函数的name属性  *console*.log(*argsFun2*.name) //argsFun2  //4.箭头函数  let *objArr*=[  {  name:"Tracy",  age:16,  gender:'female'  },  {  name:"Mark",  age:18,  gender:'male'  },  {  name:"Lily",  age:19,  gender:'female'  },  {  name:"Tony",  age:15,  gender:'male'  },  {  name:"Bobby",  age:30,  gender:'male'  },  ]  let *getYoungest* = ()=>{  let min = *objArr*[0].age  for (let i = 0; i < *objArr*.length; i++) {  if(*objArr*[i].age < min){  min = *objArr*[i].age  }  }  return *objArr*.filter(item=>item.age == min)  }  *console*.log(*getYoungest*()); </script> </body> </html> |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**箭头函数的this**</title> </head> <body> **模糊搜索：** <input type**="text"** id**="search"**> <script>  let *searchInput* = *document*.querySelector("#search")  // console.log(searchInput)  *searchInput*.oninput = function() {  // console.log(this.value)  // let \_this = this //把input对象保存起来  // setTimeout(function() {  // console.log(`发送${\_this.value}到后端，获取数据列表`)  // },1000)  //用箭头函数  *setTimeout*(() =>{  *console*.log(`发送${this.value}到后端，获取数据列表`)  },1000)   } </script> </body> </html> |

# 10 .es6的Symbol

|  |
| --- |
|  |

## 课堂笔记

## 1.Symbol没有两个Symbol对象是相等的

## 2.Symbol不能进行运算

## 3.Symbol有toString方法 返回的是“Symbol()“

## 4.Symbol可以隐式转化为Boolean

|  |
| --- |
| <!DOCTYPE html> <html lang**="en"**> <head>  <meta charset**="UTF-8"**>  <title>**es6的Symbol数据类型**</title> </head> <body> <script>  //基本使用  let *name* = *Symbol*("name")  let *age*= *Symbol*("age")  let *obj* = {  [*name*]:"kenny",  [*age*]:30  }  *console*.log(*obj*)  *console*.log(*obj*[*name*]) //使用Symbol的对象获取属性值的方法，不能使用obj.key的方法，会报错  *obj*[*age*] = 20 //使用Symbol的对象修改属性值的方法  *console*.log(*obj*)  let *s1* = *Symbol*()  //Symbol不能进行运算  // console.log(s1+"who") //报错：不能把Symbol的值转化为字符串，  //Symbol有一个toString方法,但是没有什么意义  *console*.log(*s1*.toString()) //Symbol()  //Symbol可以转换为bool  if(*s1*){ //此时s1就是true  *console*.log("that is right")  }  </script> </body> </html> |

### Symbol还能这么用

|  |
| --- |
|  |
|  |

### 需要这么调用

|  |
| --- |
|  |
|  |

### 给Symbol对象添加说明文字

|  |
| --- |
|  |
|  |

### 注意：使用Symbol后，不能进行for in循环遍历

|  |
| --- |
|  |
|  |

#### for in获取不到Symbol的数据

### 注意：使用Symbol作为对象的key，不能直接用[Symbol()]:值，因为这样子是无法访问 到这个属性的，必须借助于变量

### 遍历Symbol需要使用Object的getOwnPropertySymbols(*obj*)方法

|  |
| --- |
| Object.getOwnPropertySymbols(obj).forEach(item=>console.log(obj[item])) |

### 还可以使用Reflact.ownkeys方法来获取

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |

#### 这个方法可以返回对象的所有属性，包括字符串属性和Symbol属性

### 注意：Symbol有一个属性很特殊，就是Symbol.iterator,如果一个对象有这个属性而且他是方法，可以这么使用obj[Symbol.iterator]()

|  |
| --- |
| let obj = {          name:"Jade",          age:18,          [Symbol.iterator]:function(){              console.log(`hello my name is ${this.name},i am ${this.age} years old`);          }      }      obj[Symbol.iterator]() |
|  |

## Symbol还有一个作用，就是作为常量

|  |
| --- |
|  |

### 这样子可以统一接口调用

## Symbol的参考文档1

|  |
| --- |
| Symbol ES6 引入了一种新的原始数据类型 Symbol，表示独一无二的值  javascript  复制代码  let name = Symbol();  typeof name  // "symbol" 作用  1. 表示一个独一无二的变量，任何两个symbol都是不相等的   ini  复制代码  let a=Symbol('lucy')  let b=Symbol('lucy')  a===b  //false   1. symbol变量可以作为对象属性名，用来避免**对象属性名冲突**   **只有symbol和string类型的值才可以作为对象的属性名**  css  复制代码  let name = Symbol();  let a = {  age:23 };  ​  a[name] = 'lucy';  --------------------------  a = {  [name]: 'lucy'  };  --------------------------  Object.defineProperty(a, name, { value: 'xu' });  ​  // 以上写法都可以获得属性名为symbol/string的属性值  console.log(a[name])  //xu 在对象中查找Symbol类型的属性名 symbol的变量不会出现在 Object.keys()的结果中，但是Object.getOwnPropertySymbols()可以  css  复制代码  Object.keys(a);  //["age"]  Object.getOwnPropertySymbols(a)  //name Symbol.for(key) 使用现有的key来搜索现有的symbol变量，找到则返回这个变量，如果不存在则新建一个值。  ini  复制代码  const s1 = Symbol.for('foo');  const s2 = Symbol.for('foo');  console.log(s1 === s2); // true Symbol.for( )与Symbol( )区别 这两种写法，都会生成新的 Symbol。  Symbol.for()生成的变量是全局的，不会每次调用就返回一个新的 Symbol 类型的值，而是会先检查给定的key是否已经存在，如果不存在才会新建一个值。  比如，如果你调用Symbol.for("cat")30 次，每次都会返回同一个 Symbol 值，但是调用Symbol("cat")30 次，会返回 30 个不同的 Symbol 值。 Symbol.keyFor( ) 返回一个已登记的**全局** Symbol 类型值的key。由Symbol.for()创建  javascript  复制代码  let s1 = Symbol.for("foo");  Symbol.keyFor(s1) // "foo"  ​  let s2 = Symbol("foo");  Symbol.keyFor(s2) // undefined  上面代码中，变量s2属于未登记的 Symbol 值，所以返回undefined。 为什么 ES6 要提出 Symbol？ ES5 的对象属性名都是字符串，这容易造成**属性名的冲突**。比如，你使用了一个他人提供的对象，但又想为这个对象添加新的方法，新方法的名字就有可能与现有方法产生冲突，如果能保证每个属性的名字都是独一无二的就好了，这样就从根本上防止属性名的冲突。 |

## Symbol的参考文档2

|  |
| --- |
| 由于每一个 Symbol 的值都是不相等的，所以 Symbol 作为对象的属性名，可以保证属性不重名。  let sy = Symbol("key1");  // 写法1  let syObject = {};  syObject[sy] = "kk";  console.log(syObject); // {Symbol(key1): "kk"}  // 写法2  let syObject = { [sy]: "kk" };  console.log(syObject); // {Symbol(key1): "kk"}  // 写法3  let syObject = {};  Object.defineProperty(syObject, sy, {value: "kk"});  console.log(syObject); // {Symbol(key1): "kk"}  Symbol 作为对象属性名时不能用.运算符，要用方括号。因为.运算符后面是字符串，所以取到的是字符串 sy 属性，而不是 Symbol 值 sy 属性。  let syObject = {};  syObject[sy] = "kk";  syObject[sy]; // "kk"  syObject.sy; // undefined 注意点 Symbol 值作为属性名时，该属性是公有属性不是私有属性，可以在类的外部访问。但是不会出现在 for...in 、 for...of 的循环中，也不会被 Object.keys() 、 Object.getOwnPropertyNames() 返回。如果要读取到一个对象的 Symbol 属性，可以通过 Object.getOwnPropertySymbols() 和 Reflect.ownKeys() 取到。  let syObject = {};  syObject[sy] = "kk";  console.log(syObject);  for (let i in syObject) { console.log(i); } // 无输出  Object.keys(syObject); // []  Object.getOwnPropertySymbols(syObject); // [Symbol(key1)]  Reflect.ownKeys(syObject); // [Symbol(key1)] 定义常量 在 ES5 使用字符串表示常量。例如：  const COLOR\_RED = "red";  const COLOR\_YELLOW = "yellow";  const COLOR\_BLUE = "blue";  但是用字符串不能保证常量是独特的，这样会引起一些问题：  const COLOR\_RED = "red";  const COLOR\_YELLOW = "yellow";  const COLOR\_BLUE = "blue";  const MY\_BLUE = "blue";  function ColorException(message) {  this.message = message; this.name = "ColorException";  }  function getConstantName(color) {  switch (color) {  case COLOR\_RED : return "COLOR\_RED";  case COLOR\_YELLOW : return "COLOR\_YELLOW ";  case COLOR\_BLUE: return "COLOR\_BLUE";  case MY\_BLUE: return "MY\_BLUE";  default: throw new ColorException("Can't find this color");  }  }  try {  var color = "green"; // green 引发异常  var colorName = getConstantName(color);  } catch (e) {  var colorName = "unknown"; console.log(e.message, e.name); // 传递异常对象到错误处理  }  但是使用 Symbol 定义常量，这样就可以保证这一组常量的值都不相等。用 Symbol 来修改上面的例子。  const COLOR\_RED = Symbol("red");  const COLOR\_YELLOW = Symbol("yellow");  const COLOR\_BLUE = Symbol("blue");  function ColorException(message) {  this.message = message; this.name = "ColorException";  }  function getConstantName(color) {  switch (color) {  case COLOR\_RED : return "COLOR\_RED";  case COLOR\_YELLOW : return "COLOR\_YELLOW ";  case COLOR\_BLUE: return "COLOR\_BLUE";  default: throw new ColorException("Can't find this color");  }  }  try {  var color = "green"; // green 引发异常  var colorName = getConstantName(color);  } catch (e) {  var colorName = "unknown";  console.log(e.message, e.name); // 传递异常对象到错误处理  }  Symbol 的值是唯一的，所以不会出现相同值得常量，即可以保证 switch 按照代码预想的方式执行。 Symbol.for() Symbol.for() 类似单例模式，首先会在全局搜索被登记的 Symbol 中是否有该字符串参数作为名称的 Symbol 值，如果有即返回该 Symbol 值，若没有则新建并返回一个以该字符串参数为名称的 Symbol 值，并登记在全局环境中供搜索。  let yellow = Symbol("Yellow");  let yellow1 = Symbol.for("Yellow");  yellow === yellow1; // false  let yellow2 = Symbol.for("Yellow");  yellow1 === yellow2; // true Symbol.keyFor() Symbol.keyFor() 返回一个已登记的 Symbol 类型值的 key ，用来检测该字符串参数作为名称的 Symbol 值是否已被登记。  let yellow1 = Symbol.for("Yellow"); Symbol.keyFor(yellow1); // "Yellow" |

## Symbol的参考文档3： [Symbol - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Symbol)