整理：

Ps:var声明的变量是全局或者整个函数块的而let、const声明的变量是块级的变量。var声明的变量存在变量提升，let、const声明的变量不存在变量提升。

let声明的变量允许重新赋值，const声明的变量不允许重新赋值。

Es6:

1. 两种声明（let和const）：

Let的特点：

①.不存在变量提升，只能在声明的位置后面使用

②.在相同的作用域内，不允许重复声明同一个变量

③.块级作用域

Const的特点：

1. 声明一个只读的常量，一旦声明，常量的值就不能被改变，意味着

const一旦声明变量就必须立即初始化，

② 只能声明的块级作用域内有效

③.不存在变量提升，只能在声明的位置后面使用

暂时性死区：

var声明变量可以在声明之前使用，相当于认为其声明其值为undefined，但是let声明的变量一旦用用let声明，那么在声明之前的都不能用,术语称为暂时性死区

所以我们要养成先声明再使用的好习惯

解构赋值：

es6允许按照一定模式，从数组和对象中提取值对变量进行赋值，这被称为解构赋值

1. 对象的解构赋值：

规律：对象根据属性名称一 一对应，这是一个无序的对应关系。

对象的解构赋值内部机制：

是先找到同名属性，然后再赋给对应的变量，真正赋值的是后者

// const props={

// className='tiger-button',

// loading:false,

// clicked:true,

// disabled:'disabled',

// }

// es5:

// var loading=props.loading;

// var clicked=props.clicked;

// es6:

const{loading,clicked}=props;

//给一个默认值

const{loading=false,clicked}=props;

// 对象的解构赋值

// 规律：对象根据属性名称一一对应，

var {foo,bar}={foo:"222",bar:'555'};

console.log(bar)

//如果变量名与属性名不一致必须写成

var {foo:baz，bar}={foo:"222",bar:'555'};

// 证明了对象的解构赋值是下面的简写形式

var {foo：foo，bar：bar}={foo:"222",bar:'555'};

var {foo,bar}={foo:"222",bar:'555'};

//对象的解构赋值的内部机制，是先找到同名属性，然后再给对应的变量，真正赋值的是后者，而不是前者，

// 我们管后面的叫变量（baz），前面的叫模式（foo）

var {foo:baz}={foo:"222",bar:'555'};

// 真正被赋值的是变量 baz 而不是模式foo

1. 数组的解构赋值

数组的解构赋值规律：数组以序号一 一对应，这是一个有序的对应关系

let[foo,[[bar],baz]] = [1,[[2],3]];

console.log(foo); //1

console.log(bar); //2

console.log(baz); //3

let[head,...tail]=[1,2,3,4,5]

console.log(...tail); 、//2,3,4,5

1. 字符串的解构赋值

字符串被转化成一个类似数组的对象

length也可以解构赋值

const [a,b,c,d,e]='hello';

a,b,c,d,e对应的分别就是hello的每个字母

function add([x,y]){

return x+y;

}

console.log( add([1,2])) //3

//函数add的参数表面上是一个数组，但在传入参数的那一刻，数组参数就被解构成x和y

//对于函数内部的代码来说，它们能感受到的参数就是x和y

function move({x=0,y=0} = {}){

return [x,y];

}

console.log(move({x:3,y:8})) ;// 3,8

console.log(move({x:3})) ; //3,0

console.log(move({})); //0,0

console.log(move()) //0,0

模板字符串：

1. 模板字符串采用反引号(``)标识，并且模板字符串中的空格，换行将在输出时有所保留

字符串拼接，将表达式嵌入字符串中进行拼接，用${}来定义。

es6提供了很多字符串的方法：

includes :判断是否找到了参数字符串，返回布尔值，找到返回true 否则返回false

repeat(n):返回一个新字符串，将原字符串重复n次(n是次数)

startsWith(参数)：返回布尔值，参数字符串是否在源字符串的头部

endsWith(参数)：返回布尔值，参数字符串是否在源字符串的尾部

includes,startsWith, endWith,这三个方法，都支持第二个参数，指定第几个位置

startWith，includes：从第几个位置开始到结束

endWith:针对当前n个位置

let x='hello world!';

console.log(x.startsWith('h'))

console.log(x.endsWith('r'))

console.log(x.startsWith('world',6)) //true 从第六个位置开始到结束

console.log(x.endsWith('hello',5)) //true 针对前n个值

x.includes('hello',6) //false 从第几个位置开始到结束

/\* repeat 参数为0；返回空字符串

参数为小数：会被取整

参数为负数:会报错，不能写负数

参数为nan：等同于参数为0

参数为字符串：会被转化成数字

参数为0到-1之间的小数

因为：先会取整 -0 ， repeat视同于

\*/

console.log('na'.repeat(0));

console.log('na'.repeat(1.3));

// console.log('na'.repeat(-5));

console.log('na'.repeat(NaN));

console.log('na'.repeat('2'));

console.log('na'.repeat(-0.5));

模板字符串的好处

1.当做普通字符串使用

var str=`hello world`;

1. 多行字符串

`

hello world!

hello xiaoming!

`

1. 嵌入变量，表达式等

箭头函数：

箭头函数：Arrow Function

为什么叫Arrow Function?

因为他的定义就是一个箭头来表示一个函数

普通函数和普通函数的区别

①更简洁的代码语法

②不绑定this

3.语法

将原函数的‘function’关键字和函数名称都去掉，并且使用‘=>’将函数体和参数列表连接起来

写法规则：

1.当函数只有一个参数的时候，括号是可选的，可以写可以不写

var fn2=function(a){

return a;

}

var fn2=a => a;

2.如果函数没有参数时，括号不可以省略，必须写

var fn1=function(){

}

var fn1 = () =>{

}

3.如果多个参数，必须要写小括号，并且用逗号分隔

var fn3=function(a,b){

}

var fn3=(a,b)=>{}

4.只有一个参数，一个表达式,可以省大括号和小括号

function(x){

return x\*x;

}

(x) => x\*x ;或者 x=>x\*x

5.如果包含多条语句的话，不能省略大括号和return

x =>{

if(x>0){

return x\*x;

}else{

return -x\*x;

}

}

//如果要返回一个对象，需要注意，return后面返回的对象必须加上括号，不写括号会报错，因为和函数整体的大括号有语法冲突

// var getHash=arr=>{

// return{

// name:'lily',

// age:"23"

// }

// }

// // console.log(getHash())

不绑定this：箭头函数的this其实就是定义的时候确定的this，

以后不管怎么调用这个箭头函数，箭头函数this始终为定义时候的this

以下的情况不可以用箭头函数

1.箭头函数不能用作构造器和new一起，会报错

2.箭头函数没有prototype属性

Es6新增数组方法：

1.es6新增数组方法

1.find:用于找出第一个符合条件的数组成员,参数是一个回调函数，数组的每一个成员一次执行这个回调函数，直到找到第一个返回值为true的成员，返回这个成员，如果没有找到，返回undefined

2.includes(): 返回一个布尔值，用来判断某个数组是否包含给定的值

includes可以指定位置，放在方法的第二个参数上，默认搜索的起始位置为0，

如果第二个参数为负数，就是从倒数位置开始搜索

3.every():有一项条件不成立就返回false，所有项都满足才返回true，返回布尔值

4.some():有一项成立就返回true，返回布尔值

5.map:映射，把一个数组映射成一个新数组，不会改变原来的数组

6.fill():使用一个给定值，填充数组，一般用于数组的初始化，数组的已有元素被全部覆盖

fill还可以指定填充的位置：第二个(开始位置)和第三个参数(不包括结束位置)

7.from():类似数组的对象和可遍历的对象转为真正的数组

from的使用方法：Array.from(类似数组的对象，可选参数，可选参数)

类似数组的对象必须要指定一个长度length，返回的是新数组

8.filter：过滤，过滤掉数组中不满足条件的值，返回一个新数组,不改变原数组的值

1.for...in:一般用于遍历对象的属性，也可以用来遍历数组，但是不推荐  
      遍历数组的时候，遍历的是索引遍历对象的时候，获取的是对象的属性

2.for...of:一般用来遍历数组，Array，Map，Set，arguments ,string  
         遍历的是数组的每一项,不能遍历对象，可以使用break,continue,return 终止循环

3.forEach:接收一个回调函数  
        forEach(function(currentValue,index,arr){  
        })  
         currentValue:当前循环的值，必选  
           index：当前循环值的索引，可选参数  
           arr:所在数组，可选  
           不能使用break,continue,return这些关键字对循环造成干扰

var arr=['a','b','c'];

arr.forEach((value,index,arr)=>{

console.log(value);

console.log(index);

console.log(arr);

Set方法：set 本身是一个构造函数，类似于数组，但成员都是唯一的，不重复 使用 Set(); 可以接收数组作为参数，也可以接收对象作为参数

Set实例的属性和方法：  
Set.prototype.constructor  //构造函数，默认就是set函数  
Set.prototype.size   // 大小，实例的成员总数

Set实例方法：两大类（操作方法，遍历方法）

操作方法：  
add(value): 结构本身添加成员  
delete(value): 删除某个结构成员，返回布尔值，成功 true 表示是否删除成功  
 has(value) ： 返回一个布尔值，表示是否是set的成员  
 clear(value) ： 清除所有成员，没有返回值

遍历方法：

keys():返回键名  
values()：返回键值  
 entries()：返回键值对  
 forEach()：使用回调函数遍历成员

var s3 = new Set(['apple','orange','pear'])

for(let i of s3.keys()){

console.log(i); //apple,orange,pear

}

for(let val of s3.values()){

console.log(val); //apple,orange,pear

}

for(let ent of s3.entries()){

console.log(ent); //['apple','apple','orange','orange','pear','pear']

}

s3.forEach((value,key) => {

console.log(key); //apple,orange,pear

console.log(value); //apple,orange,pear

})

map实例的方法：

map出现的意义：我们的意愿是把一个DOM节点添加到对象({})里面作为键值，但是对象值接收字符串类型的键值，map的出现解决了这个问题，它会把对象转化成字符串  
map的定义：是一种数据结构，数据的存在可以是 键——值  
键不局限于字符串，可以是对象等任意数据类型

属性：

1. size属性返回 Map 结构的成员总数  
   set()：设置键和值，返回的是当前map对象  
   往map里面设置值的时候添加重复的后面的会覆盖前面的  
   get()：获取键所对应的值读取的时候读取不存在的值，会返回undefined。  
   delete(key)：删除全部成员，没有返回值  
   has(key)：判断是不是拥有，返回布尔值  
   clear()清除所有数据

遍历方法：

keys():返回键名  
values()：返回键值  
 entries()：返回键值对  
 forEach()：使用回调函数遍历成员

forEach()接受两个参数，第一个是个函数，第二个是reporter

Promise

Promise：解决异步编程的方式，比传统方式更强大

Promise的用法：

promise的用法：根据es6的规定，promise是一个构造函数，用来实例化一个promise对象

promise函数的参数需要接收一个函数，这个函数叫立即执行器：excutor

函数的参数(立即执行器)j接收两个参数，resolve,reject,这两个参数是函数，由js引擎提供，不需要我们自己部署

resolve：异步成功的函数，

reject：异步失败的函数

resolve的作用：将promise的状态从未完成改变成已完成

成功(从pending变成resolved)

reject的作用：将promise的状态从‘未完成’改变成失败

失败(从pending变成rejected)

promise有三种状态：

pending(进行中)

resolved(成功)

rejected(失败)

状态的特点:要么成功，要么失败，两种状态，是互斥的

例子：支付宝付钱

var promise=new Promise((resolve,reject)=>{

if('异步成功'){

resolve()

/\*

resolve在异步请成功时调用，并将请求成功的结果作为参数传递出去

写法：resolve(值)值可以是任意值

\*/

}else{

reject()

// reject在异步请求失败时调用，并将请求的失败的原因作为reject()函数的参数传递出去

// 写法：reject(值) 值可以是任意值

}

}); //实例化一个promise对象

console.log(promise)

/\*

PromiseStatus:当前promise的状态

PromiseValue：promise的返回值

then 的解释：每一个promise都有一个then方法，这个函数接收两个函数作为参数第一个参数在promise的状态由pending变成resolved的时候调用

第二个回调函数：promise的状态由pending变成rejectd时调用

每个then方法会返回一个全新的promise

var newPromise=promise.then((data)=>{

console.log(data)

// return new Promise((resolve,reject)=>{

// resolve('这是一个新的')

// })

// resolve('aaaaa')

}).then((res)=>{

console.log(res);

})

promise.catch((error)=>{

//失败

console.log(error)

}).finally(()=>{

console.log('不管成功还是失败都会去执行')

})