正课:

1. ES6

1. ES6

什么是: ECMAScript 第6个版本

优点: 在不改变原理的基础上，简化了js代码。

问题: 兼容性

包括:

1. let:

什么是: 专门代替var来声明变量用的

为什么: var的问题：

1. 声明提前

2. 没有块级作用域

何时: 只要声明变量都用let

优点: 1. 阻止了声明提前

2. 添加了块级作用域

原理: let其实就是一个匿名函数自调！

且let为了双保险，其实在底层悄悄给变量改了名字

let的小脾气:

1. 在相同作用域/块内，禁止同时let两个同名的变量

2. 在let a之前到当前作用域的顶部之间，不允许提前使用a变量

2. 箭头函数:

什么是: 对普通函数声明的简写

何时: 今后绝大多数匿名函数的创建，都用箭头函数。

如何: 3件事:

1. 去掉function 在()和{}之间加=>

2. 如果形参列表只有一个变量，可省略()

3. 如果函数体只有一句话，可省略{}

但是，如果仅有的这句话，还是return，则必须省略return

去掉{}时，还要注意去掉语句结尾的";"

双刃剑: 让函数内的this，与函数外的this连通，保持一致。都指向函数外的this。

结论: 如果希望函数内的this和函数外的this，不一致时！就不能用箭头函数！

比如: 对象的方法不能用箭头函数简写。

比如: DOM中的事件处理函数也不能用箭头函数简写。

问题: 一旦改为箭头函数，则不能再使用arguments

解决: 参数增强中的rest语法

3. for of

什么是: 在特定情况下专门简化for循环

总结: 遍历一个数组:

1. for(var i=0;i<arr.length;i++){

//当前元素arr[i]

}

优点: 即可控制遍历的方向，又可控制循环的步调。

2. arr.forEach((elem,i,arr)=>{

//当前元素elem

})

局限: 不能控制遍历的方向，也不能控制遍历的步调。

3. for of

如何: for(var elem of 数组){

//of会依次取出数组中每个元素值，保存在of前的变量elem中

}

局限: 只能获得元素值，无法获得当前元素的位置。

其实, for of不但可以用于数组，还可以用于字符串，类数组对象。

总结:

for of可以遍历数字下标的数组，类数组对象，字符串中的每个字符。

for in 专门遍历自定义下标的关联数组和对象。

4. 参数增强:

什么是: ES6中对函数的参数，提供了更强大，更方便的功能

包括:

1. 参数默认值(default):

function fun(形参1, ..., 形参n=默认值){

//将来，如果调用fun函数时，传入了最后一个参数，则用用户传入的。如果调用fun函数时，没有传入最后一个参数，则使用默认值作为备胎。

}

ES5的代替方式:

形参n=形参n||默认值

//如果"形参n"是有意义的值(不是false,"",undefined,null, NaN)，就用"形参n"的值继续后续执行。

//如果"形参n"是没有意义的值(没给值)，就用"默认值"作为备胎。

强调: 局限: 只能解决最后一个形参不确定的情况。

2. 剩余参数(rest):

什么是: 专门代替arugments

为什么: arguments的问题:

1. 不是纯正的数组类型，用不了数组家的函数。

2. 在箭头函数中不支持arguments了。

3. 只能获得全部实参值。不能有选择的获得部分实参值。

何时: 今后只要使用arguments的地方，都可用rest语法代替  
 如何:定义函数时:

function fun(形参1, 形参2,...数组名){

//...后的数组中会接住除之前形参外的其它所有剩余实参值。

//数组是纯正的数组类型，可随意使用数组家的函数

//即使将fun改为箭头函数，依然可以使用...数组名的rest语法

}

3. 打散数组(spead):

什么是: 将一个数组或对象打散成单个的变量值

为什么: 代替apply。

apply的问题是: 本职工作不是打散数组，而是替换this，顺便打散数组。

何时: 今后只要希望将一个数组整体，打散为单个值。

如何: 传递参数或赋值时

fun(...arr)

先将arr中的元素值打散为单个值，再分别传给fun()。

项目中:

1. 拼接两个数组:

var arr1=[1,2,3], arr2=[4,5,6];

arr=[...arr1,...arr2]

arr : [ 1,2,3,4,5,6 ]

2. 合并两个对象:

var obj1={x:1, y:2},

obj2={a:1,b:2};

var obj={...obj1, ...obj2}

obj:{ x:1, y:2, a:1,b:2 }

5. 解构:

什么是: 将一个大的对象或数组中的个别成员提取出来，单独使用。

何时: 如果给的是一个巨大的对象，而我们这个功能，只需要其中个别函数或属性时，就要将所需的成员解构出来，单独使用。

如何:

1. 数组解构:

什么是: 将大的数组中的个别元素提取出来单独使用。

如何: 下标对下标

[变量1,变量2,...]=数组

0 1 ...

结果: 变量1=数组[0]

变量2=数组[1]

2. 对象解构:

什么是: 将大的对象中的个别成员提取出来单独使用。

如何: 属性名对属性名

{属性名1:变量1, 属性名2:变量2, ... }=对象

结果: 变量1=对象.属性名1

变量2=对象.属性名2

简写: 如果新变量的名字延用成员在原对象中的属性名，则可以简写:

比如: 想从user对象中，解构出登陆方法和注册方法单独使用:

var {signin:signin,signup:signup}=user;

可简写为:

var {signin,signup}=user;

调用时:

signin();

signup();

3. 参数解构:

什么是: 在定义函数和调用函数时，采用对象结构传递参数。

何时: 多个形参值不确定有没有，而且又要求实参值与形参值之间必须对应。

为什么:

默认值: 局限: 只有最后一个形参不确定时，才能用默认值。

...rest和arguments: 局限：无法让形参值与实参值一一对应。

用对象结构传参: 问题: 定义函数时，只有一个对象形参，将来调用者不知道具体形参的个数和名称！

如何: 2步:

1. 定义函数时，形参就写成对象语法！

2. 调用函数时，所有实参必须放在一个和形参对象结构相同的对象中传入

比如:

//定义制作西红柿鸡蛋面的函数

/\*function cook({

//配对 变量

dan:dan,

cai:cai,

mian:mian

}){\*/

function cook({dan="鸡蛋",cai="西红柿",mian="面"}){

//如果一个形参变量没有传值，则给一个默认值顶替。

//dan=dan||"鸡蛋";

//cai=cai||"西红柿";

//mian=mian||"面";

console.log(`炒${dan}`)

console.log(`炒${cai}`)

console.log(`炒${cai}和${dan}`)

console.log(`煮${mian}`)

console.log(`把${mian}和${cai}${dan}拌在一起`)

return `热腾腾的${cai}${dan}${mian}`;

}

//周一:

var wan=cook({});

console.log(`获得一碗:${wan}`);

//周二:

wan=cook({

dan:"辣条",

cai:"黄瓜",

mian:"米线"

});

console.log(`获得一碗:${wan}`);

//周三:

wan=cook({

dan:"火腿肠"

})

console.log(wan);

//周四:

wan=cook({

cai:"黄瓜"

})

console.log(wan);

//周五:

wan=cook({

cai:"榴莲",

mian:"米饭"

})

console.log(wan);

作业:

1. 不声明第三个变量，交换两个变量中的值，共有几种方法

a=3, b=5

... ...

//最后一种: [a,b]=[b,a]

console.log(a, b)//5 3