正课:

1. ES6

1.ES6:

ECMAScript的第6个版本: 在不改变原理的基础上，尽量简化了代码。

let:

什么是: 专门代替var，用于声明变量用的关键词

何时: 今后所有的var，都用let代替

为什么:

var的两个缺点:

1. 声明提前: 打乱了程序正常执行的顺序

2. 没有块级作用域: 块内的变量很有可能提前到块外部，影响原本正确的外部的程序。

块: if(){ ... }else if(){ ... }else{ ... }

for(){ ... }

while(){ ... } do{ ... }while();

都是块，但都不是作用域

其内部的变量，都会被提前到块外对外部造成不可预期的影响。

比如:

var t=0;

function fun(){

t+=10;//原本就是想对全局t+10

}

fun();

console.log(t); //10

过了一段时间/其它人给函数中添加了一段不会执行的代码:

var t=0;

function fun(){

var t//undefined;

t+=10;//原本就是想对全局t+10

//因为有了局部变量t，就无法修改全局了

var err=false;

if(err){ //if根本没执行

//if不是作用域，拦不住内部的t被提前到外部函数的顶部

var t=new Date();

... ...

}

}

fun();

console.log(t); //0

结论: 今后所有的var都用let代替

let的两个作用:

1. 阻止声明提前: 在let a之前，不允许使用变量a

在程序中，相同范围内，不允许重复声明两个let a;

2. 让if while for do while块都变成作用域，块内部的变量无法提前到外部，也就不会影响外部。

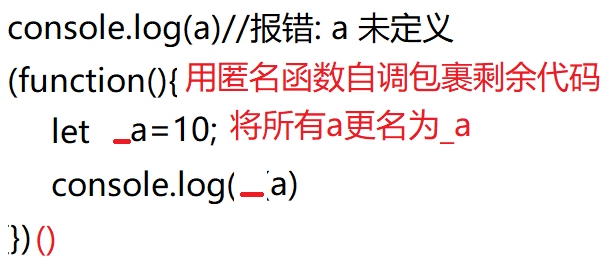
let的原理: 其实let底层就是一个匿名函数自调

let相当于做了两件事:

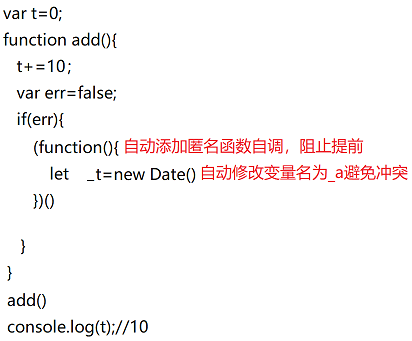
1. 用匿名函数自调包裹当前范围内的剩余代码

2. 将let后的变量a，更名为\_a

比如:



再比如:



for+let: 会形成闭包

for(let i=0;i<3;i++){

arr[i]=function(){console.log(i)}

//(function(\_i){

//arr[\_i]=function(){console.log(\_i)}

//})(i)

}

箭头函数: 对所有函数定义的简写:

function 函数名(形参列表){ 函数体; return 返回值 }

var 函数名= function (形参列表){ 函数体; return 返回值 }

3句话:

1. 去function，在()和{}之间加=>

(形参列表)=>{ 函数体; return 返回值 }

2. 如果形参列表中只有一个形参，可省略()

形参1=>{ 函数体; return 返回值 }

3. 如果函数体只有一句话，可省略{}

形参1=>一句话

如果仅剩的一句话，还是return，必须去return

省略{}后，结尾的分号，必须省略

箭头函数的双刃剑: 箭头函数中的this，是内外通用的

比如:

var lilei={

sname:"Li Lei",

friends:["tom","jerry","jack","rose"],

intr:function(){

//this->lilei

this.friends.forEach(

function(name){//回调函数

//不加bind，回调函数中的this->window

console.log(`${this.sname} 认识 ${name}`);

}.bind(this) //加上bind才能将函数内部错误的this换成外部正确的this

)

}

}

如果觉得bind不直观，可改为箭头函数:

var lilei={//对象的{不是作用域！}

sname:"Li Lei",

friends:["tom","jerry","jack","rose"],

intr:function(){//这里的function不能改箭头，一旦改为箭头，this->window

//this->lilei

this.friends.forEach(

//内部的function可改为箭头函数，因为希望和外部的this通用！this->lilei

name=>console.log(`${this.sname} 认识 ${name}`)

)

}

}

总结: 如果希望内外this一致时，就可改为箭头函数，代替bind

如果不希望内外this相同时，就不可改为箭头函数。比如: 对象中的方法不能改箭头；事件处理函数不能改箭头

for of: 快速遍历一个数组的内容的最简化写法

总结: 遍历一个数组:

1. for循环

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

arr[i] //获得当前数组元素

}

最灵活的遍历，即可控制遍历的顺序，又可控制遍历的步调

2. forEach简化:

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

elem //当前数组元素

})

问题: 只能从头到尾依次遍历每个元素，不能控制顺序（从后向前或从前向后）和步调(2,4,6,... 5,10,15,... )

何时: 如果既关系内容，又关心位置，但不会改变遍历的顺序或步调时

3. for of更简化:

for(var elem of arr){

//of会依次取出索引数组中的每个元素值

elem //当前数组元素

}

问题：只能获得元素内容，无法获得元素位置

所以: 仅关心元素内容时，才用for of

vs for in

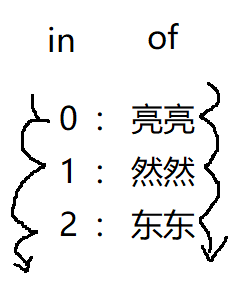
1. 使用场合: for of专门遍历索引数组

for in专门遍历关联数组

2. of vs in:

of只取元素值

in 只取属性



比如: var names=["亮亮","然然","东东"]

for(var name of names){

//of依次取出names数组中每个人名保存到前边的变量name中

console.log(name + " - 到!");

}

参数增强:

1. 默认值:

ES6中可以对形参列表中的最后一个形参设置默认值

何时: 如果只有最后一个形参不确定时

如何:

//定义函数find，在arr中查找指定元素出现的位置

//如果不给第三个参数，则默认从0位置开始找

//如果给第三个参数，则从给定的位置向后找

function find(arr,elem,starti){

/\*方法一: if判断

if(starti===undefined){

starti=0;

}\*/

//方法二: 短路

//starti=starti||0;//0是starti的备胎

//||会自动将前后转为boolean

//如果starti转换后不是false，就首选starti使用。如果starti转换后为false，就用0代替starti；

console.log(`在arr中,从${starti}位置开始，找下一个${elem}出现的位置...`);

}

//方法三: ES6的默认值

function find(arr,elem,starti=0){

//如果没有给starti传参，则自动用0代替！

console.log(`在arr中,从${starti}位置开始，找下一个${elem}出现的位置...`);

}

var arr=[];

find(arr,3);//从0位置开始

find(arr,3,5);//从5位置开始找

2. 剩余参数: 代替arguments

什么是: 让函数可以自动收集结尾多个不确定的参数值

何时: 如果在函数形参的结尾，有不确定个数的参数值传入，则可用剩余参数方式收集

为什么:

arguments: 2大缺点:

1. 不是纯正的数组类型对象，数组家的API一个都不能用

2. 只能收集所有参数值，无法有选择的收集部分参数值

如何: 2步:

1. 在定义函数时，在可以确定的形参后添加"...数组名"

2. 在函数内，"数组名"数组中，就自动收集了除已经确定的形参之外的剩余实参值。

比如：

定义函数时:

function calc(ename,...arr){

调用时:

calc("Li lei",10000,2000,3000);

//...arr[10000,2000,3000]

calc("Han Meimei",5000,1000,2000,3000);

//...arr[5000,1000,2000,3000]

优点: 1. 可有选择的获得部分实参值

2. arr是纯正的数组，数组家的API可以随便用！

3. 打散数组: 代替apply，专门用于打散数组类型的参数

何时: 只要函数需要多个零散的参数值，但给定的参数值却是放在一个数组中传入的，发生不一致，就要打散数组

apply一定有问题:

apply的主要目的是替换this，顺便打散数组

如果只希望打散数组参数，和this无关时，用apply就会很别扭.

比如: Math.max(arr)不支持数组中查找最大的元素

传统: 可用: apply打散数组类型参数为单个值

console.log(Math.max.apply(null,nums));

console.log(Math.max.apply(Math,nums));

console.log(Math.max.apply(nums,nums));

apply的第一个参数即没用，又必须写

解决: 不用apply,用...

如何: console.log(Math.max(...nums))

// 拆

...先将nums数组打散为单个值，再陆续传给max

总结:

1. 如果在定义函数时，形参列表中...arr，是收集剩余参数保存到数组的意思

2. 如果在调用函数时，在实参列表中...arr，作用是相反的，是打散数组为单个值，再传入。

解构:

什么是: 仅提取出一个大对象中的部分成员单独使用

何时: 如果只使用一个大对象中的一小部分成员时

如何: 3种情况:

1. 数组解构: 下标对下标:

var date=[2019,5,5];

/\*var y=data[0]

var m=data[1]

var d=data[2]\*/

var [y,m,d]=date;

y m d

2. 对象解构: 属性对属性:

比如:

var user={

uid:1001,

uname:"dingding",

set:1,

signin:function(){

console.log("登录...");

},

signout:function(){

console.log("注销...");

},

signup:function(){

console.log("注册...");

}

}

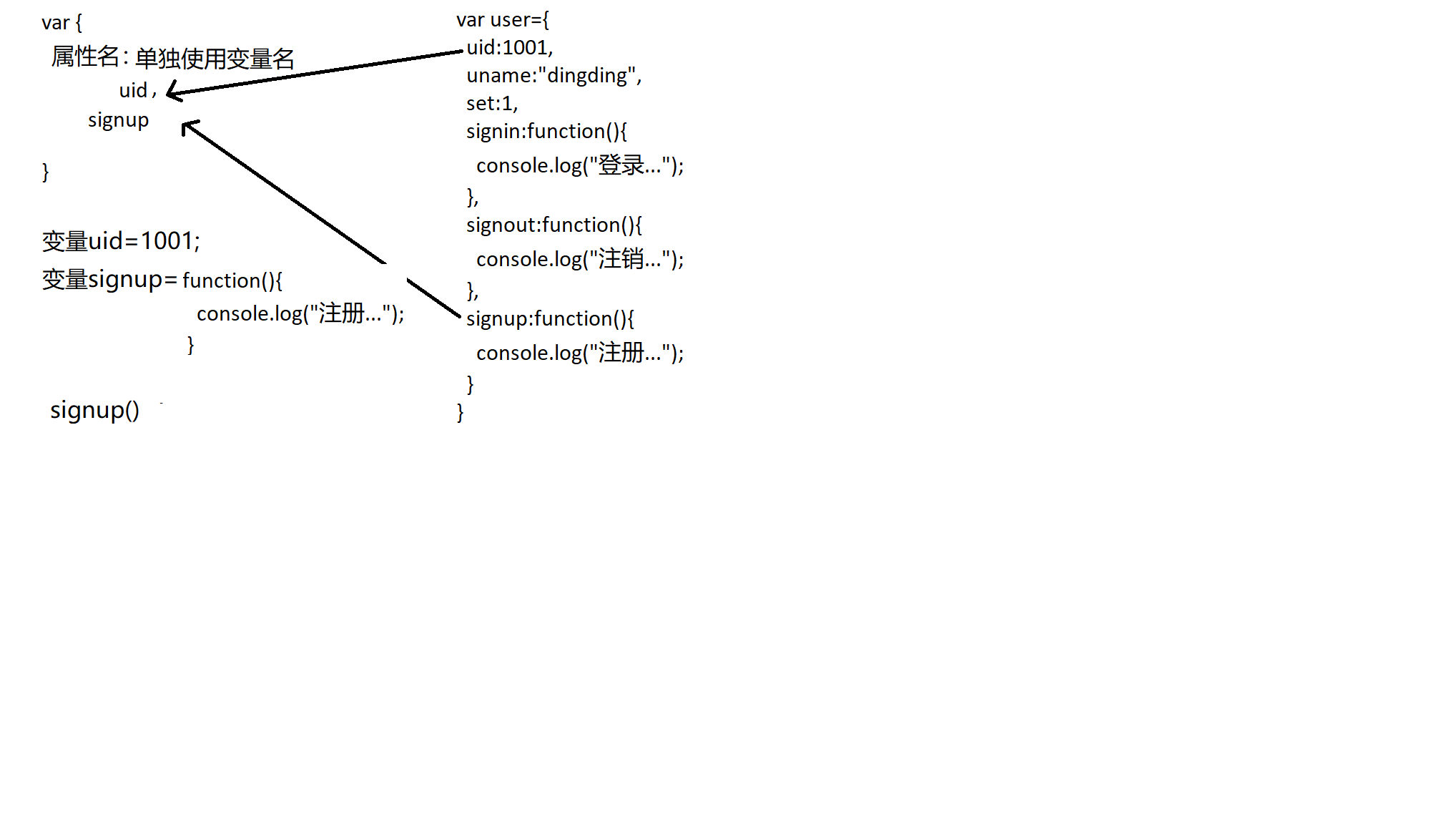
//var {uid:uid,signup:signup}=user;

var {uid,signup}=user;

console.log(uid);

signup();

原理:



3. 参数解构:

什么是: 在向函数中传参时，将一个大的对象，打散后，传递给对应的形参变量.

何时: 多个形参不确定时，又要求按顺序传入时、

为什么:

1. 默认值: 只能应对结尾一个参数不确定时

2. 剩余参数: 虽然可应对多个参数不确定的情况

但无法规定传入参数的顺序

比如:

function calc(ename, base,饭补, 高温,房补)

calc(lilei, 10000, , 200, 600)

calc(hmm, 10000, , , 200)

calc(jack, 10000, , 300, 500)

总结: 当多个形参都不确定时，且每个实参值必须对位传给指定的形参变量时，单靠调整形参的个数和顺序，无法满足所有调用的情况

解决: 参数解构:

如何: 2步:

1. 定义形参时，所有的形参变量都要定义在一个对象结构中

比如:

//1. 定义形参列表时，就用对象结构定义

function ajax({

//与顺序无关

url:url,

type:type,

data:data,//不确定

dataType:dataType//不确定

}){

console.log(`向${url}发送${type}请求`);

if(data!=undefined&&type=="get"){

console.log(`在url结尾拼接参数?${data}`)

}

if(data!=undefined&&type=="post"){

console.log(`xhr.send(${data})`);

}

if(dataType=="json"){

console.log(`JSON.parse(返回结果)`);

}

}

2. 调用函数传参时，所有实参值，都要放在一个对象结构中整体传入。

比如:

ajax({

url:"http://localhost:3000/products/getProductsByKwords",

type:"get",

data:"kw=macbook i5",

dataType:"json"

});

总结: 参数解构，其实就是对象解构在传参时的应用而已