正课:

\*\*\*\*ES5

1. call/apply/bind

2. 数组API

ES6

\*\*\*\*ES5:

1. call/apply/bind

替换this！

何时: 只要this不是想要的，就用call/apply/bind替换

鄙视: call vs apply vs bind

call/apply : 立刻调用函数，并临时替换函数中的this为指定对象

何时: 如果希望立刻调用函数时

如何: fun.call(obj,参数值,......)

立刻调用fun函数，并临时替换fun中this为obj

fun.apply(obj,数组)

立刻调用fun函数，并临时替换fun中this为obj

同时，打散数组参数为单个参数值，再传给fun

bind: 基于原函数，创建一个新函数，永久绑定this为指定对象

何时: 如果希望基于原函数创建一个新函数时

或者说，不是立刻调用时

如何: fun.bind(obj,参数值,...)

创建一个和fun完全一样的新函数

永久绑定fun中的this为obj

永久绑定部分参数值

2. 数组API:

3组:

1. 判断:

every(): 判断数组中是否所有元素都符合条件

如何: var bool=arr.every(function(val,i,arr){

//val自动获得当前元素值

//i获得当前位置

//arr 指当前数组对象本身

return 判断条件

})

原理: every用function去每个元素上执行一次

只有所有元素都返回true时，整体才返回true

只要有一个返回false，则整体返回false

some(): 判断数组中是否包含符合条件的元素

如何: var bool=arr.some(function(val,i,arr){

同every

return 判断条件;

})

原理: some用function去每个元素上执行一次

只要有一个返回true，则整体返回true

除非所有元素都返回false时，整体才返回false

2. 遍历API:

forEach(): 对原数组中每个元素执行相同的操作

如何: arr.forEach(function(val,i,arr){

arr[i]=新值;

})

map(): 依次取出原数组中的值，执行相同操作后，放入新数组返回。

如何: var newArr=arr.map(function(val,i,arr){

return新值;

})

3. 过滤和汇总:

filter(): 选取数组中符合条件的元素，组成新数组。原数组不变

如何: var subArr=arr.filter(function(val,i,arr){

return判断条件;

})

reduce(): 遍历数组中每个元素值，汇总出一个最终结果

如何: var result=arr.reduce(function(prev,val,i,arr){

//prev 截止目前的临时汇总值

return prev+val;

},prev)

ES6:

1. let和块作用域

问题1: hoist，打乱了程序的正常执行顺序

解决: 今后用let代替var声明变量

原理: let通过禁止在当前作用域内提前使用未声明的变量，来避免声明提前

强调: 如果已经用let声明的变量，不能再用var重复声明

问题2: js没有块级作用域, 块内的变量即使不执行，也会影响块外的变量:

块级作用域: 程序中的一对儿{}之间，称为代码块

js vs java: Java 三级作用域: 全局，函数，块

js 两级作用域: 全局，函数，没有块级作用

解决: 用let代替var

规定: let声明的变量仅在当前块内有效！

原理: 自动在块作用域位置创建匿名函数自调，划分临时作用域。

问题3: 原来在for/while/if...内声明的变量，出了结构，就不能使用

解决: 今后，凡是出了结构，还要使用的变量，必须在结构外提前声明。

总结: 今后所有变量都要用let声明，而不var

2. 箭头函数: 简化回调函数

何时: 今后所有回调函数都可用箭头函数简化！

如何:

1. 去function改箭头

更简化:

2. 如果只有一个参数变量，可省略()

3. 如果函数体只有一句话，可省略{}

如果仅有一句return，则可省略return

箭头函数特点: 内容this通用！

3.参数增强:

Default:(默认值)

function fun(参数1,参数2,参数3=默认值,参数4=默认值){}

调用时，如果传给参数的值有效，就用传入的值

如果没有传入参数，或传入的参数无效，则用提前指定的默认值

REST:(剩余参数列表) 为了代替arguments

arguments的问题:

1. 不是真正的数组类型，无法使用数组类型的API

2. 默认只能获得所有参数值，无法仅选取部分参数值

解决: REST

何时: 今后，都是用REST语法代替arguments

如何:

定义函数: function fun(参数1,参数2,...剩余参数的统称)

调用时: ...剩余参数的统称 会获得一个数组

SPREAD:(散播) 为了代替apply打散数组类型的参数

apply的问题: 1. 必须传入第一个对象参数，代替this

2. 必须将所有参数都放入数组中

解决: spread

何时: 今后只要打散数组类型参数时，首选spread

如何: 定义函数时，参数变量是分开定义的

调用函数时，可用fun(值1,...arr)，打散arr数组

4. 模板字符串: 简化复杂格式的字符串拼接

何时: 今后所有字符串拼接，都用模板字符串代替+

如何:

1. 用反引号`包裹整个字符串

2. 在模板字符串内，自动支持换行

3. 在模板字符串内，支持动态生成表达式的值

要执行的表达式，必须用${}包裹

5. 解构: 简化复杂对象的打散和批量赋值

1. 数组解构: 等号左右两边都是数组格式

解构时，用下标对应值和变量

2. 对象解构: 等号左右两边都是对象格式

解构时，用属性名匹配值和变量

3. 参数解构: 其实，在调用函数时，如果传入的参数值是数组或对象，也可被解构为单个参数值

用法和数组结构和对象结构完全一样

6: class: 为了简化面向对象:

封装，继承，多态，访问器属性，静态方法

class: 集中定义构造函数和原型对象方法的程序结构

为什么: 更像封装

何时: 只要创建一种新类型，必须先定义class

如何: class 类型名{

constructor(...){

this.xxx=xxx;

}

原型对象方法(){//类型名.prototype.xxx()

... this.xxx ...

}

}

继承: 2步:

1. 子类型构造函数中借用父类型构造函数:

super(参数值)

super自动指代父类型的构造函数

2. 让子类型继承父类型

class 子类型 extends 父类型

不用再Object.setPrototypeOf

访问器属性:

class 类型名{

constructor(...){

this.\_age=age

}

get age(){return this.\_age}

set age(val){

... ...

}

}