正课:

1. ES6

1. ES6: 不改变原理的基础上，尽量简化代码

let: 代替var，用来声明变量:

为什么: js有两个广受诟病的缺陷:

1. 声明提前

2. 没有块级作用域

参数增强:

1. 默认参数: 即使不提供实参，形参也有备用的参数值

如何: function fun(形参1, ..., 形参n=默认值)

执行效果: fun(实参1,.../\*没有提供实参n\*/)

fun中的形参n的值就采用默认值

强调: 带默认值的参数必须定义在参数列表的末尾

2. 剩余参数: 代替arguments，接收不确定个数的参数值

为什么: arguments的问题:

1. 不是纯正的数组类型，无法使用数组的API

2. 只能获得全部参数值，无法有选择的获得部分

如何: 定义函数时: function fun(形参1,...数组名){ ... }

执行时: ...可自动获得除形参1之外的剩余参数,保存在一个数组中。

优点: 1. ...后的数组名获得的是一个纯正的数组

2. 可有选择的获得部分参数值

3. 散播: 代替apply，专门执行打散数组的操作:

为什么: apply的本职是替换this，顺便打散数组

如何: 调用函数时: fun(...数组)

执行时: ...会先将数组打散为单个元素，再分别传入fun中

箭头函数: 用于简化回调函数和匿名函数的

1. 去function，改为=>

2. 如果函数体只有一句话，则可省略{}

如果仅有的一句话还是return，则必须省略return

3. 如果只有一个形参，则可省略()

箭头函数特点: 一旦改为箭头函数后，内外this一致

今后如果不希望内外this相同时，就不能用箭头函数

解构: 从一个大的对象中，提取出想要的成员，单独使用

3种:

1. 数组解构: 将一个大的数组中的元素提取出来单独使用:

如何: 下标对下标:

var [变量1, 变量2, ...]=[元素1, 元素2, ...]

0 1 0 1

结果: 变量1=元素1;

变量2=元素2;

2. 对象解构: 从一个对象中提取出想要的成员单独使用

如何: 属性对属性:

var {属性1:变量1, 属性2:变量2, ...}=

{属性1:值1, 属性2:值2, ...}

结果: 变量1=值1; 变量2=值2;

3. 参数解构: 其实是对象解构在函数传参时的应用

何时: 多个参数都可选时

如何: 2步:

1. 定义时: 将参数列表定义为对象语法:

//function fun({属性1:形参1,属性2:形参2,...})

function fun({形参1,形参2,...}){ ... }

2. 调用时: 将传入的参数放在一个对象中整体传入

fun({

属性1: 值1, ...

})

执行: fun将整个实参对象传给形参对象，形参对象通过解构，从实参对象中抽取对应的参数值。

如果找不到对应的，则形参值默认为undefined

for...of...: 最简化的遍历数组和类数组对象的方法

1. for循环:

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

var elem=arr[i];

}

最灵活的，万能的

2. forEach:

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

局限: 无法控制循环的方向和步调，只能顺序依次遍历每个元素

3. for of:

for(var elem of arr){

//of会依次获得arr中每个元素的值

}

局限: 只能获得元素值，无法获得当前位置

vs for in: for(var key in obj/关联数组)——自定义下标名

for(var elem of arr/类数组对象)——数字下标

class: ES6对整个面向对象语法的简化:

1. 对象直接量的简化:

var sname="Li Lei";

var sage=11;

var lilei={

sname, //sname: sname,

sage, //sage: sage,

intr(){ ... } //:function(){ ... }

}

2. 对创建一种类型的简化:

//1. 用class{}包裹构造函数和原型对象

class Student{

//2. 构造函数名提升为类型名，构造函数统一更名为 constructor

constructor(sname,sage){

this.sname=sname;

this.sage=sage;

}

//3. 所有原型对象方法可直接写在class中

intr(){

console.log(`I'm ${this.sname}, I'm ${this.sage}`);

}

}

3. 继承:

//1.让子类型继承父类型: Plane extends Enemy

//不再需要Object.setPrototypeOf

class Plane extends Enemy{

constructor(fname,speed,score){

//2.用super(fname,speed)调用父类型构造

//不再需要call，不再需要传this

super(fname,speed);

this.score=score;

}

4. 静态方法:

class 类名{

constructor(){

}

方法(){ ... }

static 静态方法(){ ... }

}

5. 访问器属性:

class Emp{

constructor(...){ ... }

get age(){return this.\_age}

set age(value){

if(value>=18&&value<=65)

this.\_age=value;

else throw Error("年龄超范围");

}

}

Promise:

问题: 多个异步调用的函数必须依次排队执行

错误的做法: 仅顺序编写:

task1();

task2();

传统的做法: 利用回调函数: 2步:

1. 在定义异步调用的函数时，就要定义回调函数参数，并决定何时调用回调函数。

function task1(callback){//callback准备提前接收下一个任务，但暂不执行

异步调用: 定时器、ajax

在异步调用的程序最后一句话之后，调用callback()

}

2. 调用前一个任务task1时，应提前将下一个任务封装在一个函数中，传入task1，相当于交给callback

task1(

function(){

task2()

}

)

问题: 回调地狱 callback hell

多级函数调用的嵌套

根源: 回调函数要求，必须在前一个任务开始时，就要提前将后一个任务传入。

解决: Promise:

如何: 2步:

1. 在定义异步调用的函数时:

1. 不要callback参数

2. 用return new Promise(function(open){

包裹原函数的所有函数体

})

3. 决定在原异步函数最后执行的一句话之后，打开开关:open()

2. 在调用函数时:

task1().then(task2)

如果task2也支持Promise

.then(task3)