：一．正则表达式

二．String类型提供的正则表达式相关函数：3件事

1.查找敏感词：4种情况

（1）.查找一个固定的敏感词出现的位置：

a. var i=str.indexOf(“要找的敏感词”，fromi)

位置的

b. 意思：在字符串str中，从fromi位置开始，查找下一个指定的敏感词出现的位置

c. 简写：其实大部分情况都是藏开始位置开始查找（fromi=0）,所以fromi可省略

f.问题：只能找一种固定的敏感词，不能写正则

(2).查找多种敏感词的位置：

a.var i=str.search(/正则表达式/)

查找

b.意思：在str中从开始位置处向后查找第一个符合正则表达式要求的铭感词的位置

C.返回值：同indexOf是完全一样的

1）.如果找到，返回敏感词第一个在字符转中的位置

2）.如果没找到。返回-1

D.坑

1）正则表达式默认区分大小写

2）想要忽略大小写，要在第二个斜线后加i

f.问题：只能获得敏感词的下标位置i,无法获得敏感词的内容

（3）.获得敏感词的内容：2种

a.只查找第一个敏感词的内容和位置

1）.var arr=str.match(/正则/i)

2).意思：在字符串str中查找一个符合正则要求的敏感词的内容和位置，并将内容和位置放在一个数组中返回

3）.返回值：

1.如果找到敏感词，就会将敏感词的内容和位置放入数组中返回

arr:[ 0:”敏感词内容”,index:敏感词位置下标i]

1. 如果找不到返回null

b.查找所有的敏感词，但是无法获得位置了

要查找所有的敏感词，在i后面加g /正则/ig

c.既查找每个敏感词的内容，有查找每个敏感词的位置

do{

Var arr=reg.exec(str);

If(arr!=null){

....

}

}while(arr!=null);

数组底层都是关联数组，两者存储方式相同，

都可以用 ：数组名[“数字/字符串”] 来找到某一个元素值

索引数组（下标是数字），本质其实也是关联数组：

数组名[数字]

关联数组（下标是字符串）:

数组名.字符串

2.替换敏感词：2中

（1）.简单替换：将所有敏感词都替换为统一的新值

a. str=str.replace(/正则/ig,”新值”)

替换

b.意思：查找字符串str中所有符合要求的敏感词，全部替换为“新值”

c.

js中字符串是不可变类型：一旦创建不可以改变只能整体替换

所以必须要用变量接住

数组是可变类型：数组类型提供的大多数函数，都是可以直接修改原数组，而不用变量接住

1. .高级替换：根据每个敏感词的不同，动态选择不同的新值替换
2. str=str.replace(/正则/ig,function(keyword){

//replace 每次找到一个敏感词就自动调用一次回调函数

//keyword:自动接收到本次找到的敏感词内容

return 根据本次找到的敏感词,动态返回不同的新值

}

1. .衍生操作：删除敏感词，其实就是将敏感词替换为“”；
2. str=str.repalce(/正则/ig,””)
3. 实例：去掉开头和结尾的空字符，其实就是ES5、6的新标准中已经包含了内置的trim函数
4. var arr1=str.trimLeft();删除开头
5. var arr2=str.trimRight();删除结尾
6. var arr3=str.trim();删除开头可结尾
7. 切割字符串
8. 简单切割，切割符是固定不变的，将字符串切割为多段字符串

Var arr=dtr.aplit(“切割符”);

1. .强调：在切割之后的结果中，不在包含切割符
2. 固定套路：将字符串打散位字符数组：按“”切割
3. 示例: 用固定的切割符切割字符串
4. 复杂切割：切割符是变化的

a. var arr=str.split(/正则/)

b. 意为: 用str中所有符合正则要求的敏感词当做切割符，切割字符串

c. 说明: split中的正则不加g，照样可以找所有敏感词——单独记忆

d. 示例: 用正则切割字符串

二. RegExp对象

1. 什么是: 在js程序中专门保存一条正则表达式的对象

2. 为什么: 正则表达式的语法是独立于所有编程语言之外的一种全新的语法，js语言不认识正则表达式语法。所以，才需要一种专门的对象，将正则表达式翻译为js语言能明白的格式。

3. 何时: 只要在js语言中使用正则表达式都要创建RegExp对象保存一条正则表达式

4. 如何创建: 2种:

(1). 用//创建: （首选）

a. var reg=/正则/ig

b. 问题: //之中只能写死一个固定不变的正则表达式，//之中不支持用js语句动态生成正则表达式

c. 何时: 如果正则表达式是固定不变的，比如: 手机号，电子邮件地址，身份证号

(2). 用new创建:

a. var reg=new RegExp("正则表达式" , "ig")

b. 强调: 用new创建正则表达式，不要再写//了！

c. 何时: 如果正则表达式不是固定的，可能根据其他变量的值动态生成正则表达式，就只能用new 来创建。因为new RegExp()的参数是字符串，而再js中有无数种办法动态拼接出我们想要的任何字符串——灵活。

d. 示例: 动态生成正则表达式;

)

5. RegExp对象提供的函数: 2个函数:

(1). 验证字符串的格式:

a. var bool=正则表达式对象.test(要检查的字符串)

b. 意为: 用test前的正则表达式，检查()中的字符串是否符合格式要求

c. 返回值:

1). 如果()中的字符串符合test前的正则表达式的要求，则返回true

2). 如果()中的字符串不符合test前的正则表达式的要求，则返回false

d. 坑: test默认只要在字符串中找到部分内容符合正则的要求，就返回true，不要求从头到尾都完整匹配的！

e. 解决: 今后只要验证格式，都要同时前加^，后加$，表示从头到尾必须完整匹配！

f. 示例: 验证手机号格式:

(2). 既查找每个敏感词的内容，又查找每个敏感词的位置

a. var arr=正则表达式.exec(可能包含敏感词的字符串)

强调: 如果希望exec()可以继续找下一个敏感词，最终找到所有敏感词，则正则表达式必须加g

b. 意为: 用正则表达式在可能包含敏感词的字符串中查找下一个敏感词的内容和位置

c. 问题: 每执行一次exec()也只能找一个敏感词——同match()不加时相同

d. 解决: 用循环反复调用exec()，exec()函数自动知道要找下一个敏感词——比match()聪明！

e. 返回值:

1). 如果本次调用函数时，找到了敏感词，就返回一个数组：同match()不加g的情况: arr:[ 0:"本次找到的敏感词内容", index:本次找到的敏感词位置 ]

2). 如果本次调用函数时，没找到敏感词，就返回null

f. 示例: 查找一个字符串中所有敏感词的内容和位置

三. Function:

1. 什么是函数: 内存中保存一段可重用的代码片段的存储空间，再起一个名字

2. 何时: 当程序中一段代码片段可能被反复使用时，都要将这段代码片段定义为一个函数，然后反复调用函数。——重用

3. 如何: (回顾)

(1). 先定义函数: 创建一个函数对象，其中保存一段代码片段，再起一个名字

a. function 函数名(形参列表){

函数体;

return 返回值;

}

b. 形参: 专门保存函数调用时必须的数据的局部变量

1). 何时: 当函数必须提供某些数据才能正常执行时，就要定义形参

c. 返回值: 一个函数的执行结果数据，通过return返回到函数外部，被外部变量接住

1). 何时: 当调用者需要获得函数的执行结果时，函数就要定义返回值

(2). 调用函数

a. var 变量=函数名(实参值列表)

b. 过程:

1). Js引擎在内存中找到指定函数名的函数

2). 将实参值列表中的实参值赋值给函数的形参变量

3). 执行函数体中定义的代码片段

4). 如果有返回值，则将返回保存到=左边的变量中

(3). 示例: 定义函数计算任意两个数的和

4. 创建函数共有3种方法:

(1). 用声明方式创建函数:

a. function 函数名(...){

... ... 声明

}

b. 会被声明提前: 打乱程序正常的执行顺序

(2). 用赋值方式创建函数:

a. var 函数名= function (...){

... ... 不是声明

}

b. 不会被声明提前:

c. 示例: 判断程序的输出结果

d. 揭示了: 函数的本质

1). 其实js中函数也是一个对象，只不过对象中保存的不是数据，而是代码片段而已

2). 其实js中的函数名只是一个普通的变量而已。

3). 函数名变量中通过函数对象的内存地址值，引用着函数对象。

4). 当我们调用函数时，都是先找到函数名变量，再通过函数名变量中保存的内存地址，找到函数对象，并执行函数对象中保存的代码片段。

(3). 用new创建函数:

a. var 函数名=new Function("形参1", "形参2", ..., "函数体;return 返回值");

b. 基本不用。

c. 揭示了函数的本质: 其实前两种方法的小写function，都是new Function()的简写

5. 重载（overload）:

(1). 什么是: 相同函数名，不同形参列表的多个函数，在调用时可自动根据传入的实参值不同，自动选择匹配的函数执行

(2). 何时: 只要一件事，可能根据传入的实参值不同，执行不同的操作逻辑时，都应该用重载.

(3). 为什么: 减少函数名的数量，减轻调用者的负担。

(4). 如何:

a. 问题: JS语言不支持其他语言那样的重载语法

因为: JS语言不允许多个同名的函数同时存在。

结果: 如果多个同名函数同时存在，最后一个同名函数会覆盖之前的所有同名函数

b. 解决: JS中借助于arguments对象来变通实现重载效果

1). 什么是arguments:

i. 每个函数中都自带的：无需创建，在函数内可直接使用

ii. 专门自动接收所有传入函数的实参值:

iii. 类数组对象: 长得像数组的对象。

和数组相比:

相同点: 1. 下标， 2. Length

不同点: 类型不同！类数组对象是Object家的孩子，不是Array家的孩子

导致类数组对象无法使用数组家的函数

2). 何时使用: 只要在js中使用重载时，都要借助于arguments对象

3). 如何使用:

i. 只创建一个函数，不要定义形参，但是要包含多种不同的逻辑:

ii. 函数内通过判断arguments对象的长度或arguments对象中的实参值内容来决定该执行哪种逻辑

(5). 将来普通js函数还要不要写形参？

a. arguments名字太长！不好用！自定义形参都普遍简短！好用！

b. arguments中的下标0、1、等没有具体意义！自定义形参！都是见名知义的！

结论: 普通函数该用形参，还用形参。只有在不确定参数个数的重载时，才被迫用arguments。

1. . 示例: 使用重载实现三种付款方式:
2. Function:
3. 匿名函数：
4. 概念：定义函数时不指定函数名的函数
5. 为什么：2大优点
6. 节约内存
7. 避免全局变量，造成全局污染
8. 何时：2种情况
9. 几乎所有的回调函数都要定义匿名函数
10. 匿名函数自调用

（4）

1. 回调函数：
2. :概念：自己定义了函数，但是不由自己调用，而是交给了另一个函数，有另一个函数按需自动调用

比如：将一个数字内容的数组升序排列

arr.sort(function(a,b){return a-b;})

1. 为什么回调函数都要定义匿名函数：节约内存
2. 匿名函数自调
3. .概念：定义一个匿名函数后，立刻调用该函数，调用后立刻释放
4. 用意：避免全局污染
5. 何时：今后一切js代码都应该放在一个大的匿名函数自调内
6. 如何：

(function(){ ... })();

创建新函数，并且立刻调用执行，而且该函数没有名字，所以调用后立刻释放

或者:

+function(){ ... }();

!Function(){ ... }();

1. 作用域(scope)和作用域链
2. .概念：a.作用：作用就是一个变量的可用范围
3. 本质：
4. .目的：为了避免不同范围之间的变量互相干扰
5. js中包含2中作用域
6. 全局作用域
7. 概念：在程序中任何位置都可以随意访问到，，专门保存全局变量的在浏览器中，全局作用域由window担当的存储空间
8. 当网页打开的一刹那，浏览器就会自动提前创建window对象，等待着保存后续创建的所有全局变量
9. 我们所创建的所有全局变量都默认保存在window对象中
10. 保存在window对象中的变量，可以再程序任何位置被访问到
11. 函数作用域
12. .概念：专门保存仅函数内可用的存储空间
13. js中一个函数的作用域不调用是不存在的，只有在调用函数时，才临时创建该函数的作用域，函数调用后，函数作用域对象连同创建改函数的变量一同释放
14. 函数作用域对象创建和释放的过程
15. 当定义一个函数时，都会创建一个函数对象，但是每个函数对象上都会保存一个“好友列表”，
16. 当调用函数时①会临时创建一个“函数作用域对象”

②在函数作用域对象中添加一个函数中所需的所有局部变量（局部变量有2种情况：在函数内var出和函数的形参变量）

③并将函数作用域对象的地址，临时保存到“好友列表”中离函数近的格子中

1. 当函数执行过程中：①函数会先按照局部，再全局的顺序，使用变量

②除非函数作用域中没有变量，就在window中找，

1. 调用函数后 ①.函数会释放函数作用域对象，导致函数作用域对象中的所有变量一同释放掉，所以，所有的局部变量都不可重用
2. .作用域链（scope chain）
3. 概念：由多级作用域对象连接起来组成的数据结构，就叫做作用域链，其实就是上例中的“好友列表”
4. 保存着一个函数的所有变量
5. 控制着变量的使用顺序
6. 闭包
7. .概念：
8. 从作用上说：既重用一个变量，有保护变量不被污染的一种方式
9. 本质：外层函数的作用域对象，被内层函数引用着二无法释放，就形成了闭包对象
10. .为什么：因为js中，全局变量和局部变量都有不可兼得的优缺点
11. 全局变量：优点：可重用！缺点：极易被污染和篡改

----将来开发中近乎禁止使用全局变量

1. 局部变量：优点：因为只能在函数内使用，绝对不会被篡改

缺点：不可重用

1. 何时：今后只要想重用一个变量，而且还想保护这个变量不被被人篡改时，都要使用闭包
2. .如何：3步
3. 用外层函数包裹要保护的变量和内层函数
4. 外层函数将内层函数返回到外部
5. 想使用内层函数，就调用外层函数，即可获得返回出来的内层函数的对象，再保护在变量中，就可以反复使用内层函数对象了
6. .结果：外层函数与内层函数之间的被保护的变量，既可以反复使用，又不会被篡改
7. .闭包的原理：鄙视题：一句话概括闭包是如何形成的

外层函数调用后，由于

外层函数的作用域对象被内层函数对象引用着，无法释放就形成了闭包

1. 闭包的缺点：比一般的函数多占用一块内存空间--外层函数的作用域对象
2. 解决：当闭包不再使用时，尽早释放闭包

外层函数名=null//外层函数和外层函数作用域对象呗一起释放

1. 笔试题：如何判断闭包问题的输出：找2样东西：
2. 找处于外层函数和内层函数之间的变量
3. 找外层函数共向外抛出了几个内层函数的对象：3种情况：
4. 通过return返回一个内层函数
5. 通过给全局变量赋值，将内层函数，赋值到外部的变量中
6. 含使用return或者赋值的方式，返回一个数组或者对象，但是数组或者对象中包含了一个或者多个函数
7. 结论：妈妈生的多个孩子都共用一个受保护的变量
8. 如果第二次调用外层函数，会重新创建一个全新的闭包对象和变量，重新生成一组内层函数对象，以第一次调用外层函数生成的闭包毫无关系！

1. 什么是面向对象
2. 概念：程序中先用对象保存一个事物的属性和功能，然后按需使用事物的属性和功能，这种编程的方法，就叫做面向对象编程
3. 为什么要面向对象：便于大量数据的管理和维护
4. 何时：今后所有项目都用面向对象思想开发
5. 如何：3步/三大特点：封装、继承、多态
6. 封装
7. 概念：创建一个对象，集中保存显示中一个事物的属性和方法
8. 什么是：便于大量数据的管理和维护
9. 何时：今后只要使用面向对象的编程方式，都要先创建所需的所有对象，备用、
10. 如何：3种
11. 用{}创建一个对象
12. var 对象名={

名 值

属性名：属性值，

...:...,

方法名：function（）{},

}

1. 如何访问对象中的成员：（属性名和方法统称为对象的成员）
2. 访问对象中的一个方法：对象.属性名
3. 调用对象中的一个方法：对象.方法名()
4. This
5. 问题：对象自己的方法中，想要使用对象自己的属性，报错，自己的属性名未定义
6. 原因：任何情况下不加任何前缀的普通变量，默认只能在函数作用域对象和全局作用域对象window中查找变量使用。无权擅自闯入一个对象中寻找属性使用
7. 不好的解决：在变量前写死当前对象的对象名
8. 问题：对象名仅仅是一个普通的变量而已，很可能随时发生变化。如果写死旧的对象名，在执行时会报错，被迫也要修改方法中写死的对象名---紧耦合
9. 好的解决：this关键字
10. 什么是this：每个函数自带的--不用创建，可直接使用，自动指向正在调用函数前的那个对象---作用
11. 何时：今后只要在对象的方法中，想用当前对象的另一个属性时，都必须加this.属性名
12. 优点：松耦合。即使对象名发生变化，也不用修改this,因为This会在调用方法时，自动获得.前当前正在使用的对象名
13. . 用new 创建

a.2步：

1. 先用new创建一个空对象：var 对象名=new Object();了解：new和()可以省略，但是至少保留一个
2. 再强行给空对象赋值新属性和新方法

对象名.新属性名=属性名

对象名.新方法名=function（）{}

b.揭示了js中对象最底层的本质：js底层都是关联数组

1)存储结构完全相同，：都是名值对的集合

2）都可用2种方式访问成员

i:标准：数组名[“下标名”] 对象名[“属性名”]

ii:简写：数组名.下标名 对象名.属性名

(3).用构造函数反复创建多个相同函数的对象

a问题： 前两种的创建对象的方式，一次只能创建一个对象，如果要反复创建多个相同结构的对象是，代码就会繁琐

b.何时：今后只要反复创建多个相同结构的对象时，都用构造函数来创建对象

1. 如何:2步：
2. 定义一个构造函数，描述多个相同结构的对象的统一结构

Function 类型名（形参变量列表）{

this.属性名=形参

... ...

this.方法名=function(){...}

...

}

2）用new调用构造函数，反复创建多个相同结构的对象

i:new做了4件事

①new先创建了一个空对象

②让孩子对象继承构造函数的原型对象

③调用构造函数，new自动将构造函数中的this指向正在创建的新对象

所以构造函数中的this都指向要创建的新对象

构造函数中的每个this.属性名=形参，都在给新对象强行赋值新属性的方式，添加新属性  
 所以构造函数中所有要加入到新对象中的属性前都要写this

构造函数中所有写this.属性名的属性，最终都会添加到新对象中

④返回新创建的对象地址保存到=左边的对象名变量中

1. 面向对象
2. .封装
3. 当只想创建一个新对象，且已经知道新对象中所有成员时，可用{}

var 属性名={

属性名：属性值,

... ...

方法名：function(){

...this.属性名...

}

}

1. 如果只想要创建一个新对象，但是暂时不知道新对象的成员，后续知道了成员，再补充时，用2步
2. 先创建空对象：var 对象名={}//其实就是new Obiect的简写
3. 在强行为新对象添加新属性

i：对象名=属性名

ii:对象名=function(){

...this.属性名...

}

1. 如果希望反复创建多个相同结构的对象时，用构造函数：2步
2. 先定义构造函数：

Function 类型名（形参变量列表）{

This.属性名=属性值

This.方法名=function(){

This.属性名...

}

}

2）用new调用构造函数

var 对象名=new 类型名（实参值列表）

|  |
| --- |
| 今天的this:4种 判断this指谁，看被谁调用   1. obj.fun() fun中的this->.前的obj 2. new fun() fun中的this->new正在创建 的新对象 3. 构造函数   .prototype.方法=function(){..${this.sname},${this.sage}}  原型对象方法中的this->将来调用这个方法的.前的某个子对象   1. 匿名函数自调或者普通函数调用fun() 在没有.没有new时,   this->window   1. 回调函数中的this指window,因为在回调函数被调用时，没有.   没有指明是谁调用 |

1. 继承
2. 问题：构造函数的优点虽然是可重用的结构代码，便于维护，但是构造函数那会却无法节约内存，反而浪费内存。因为，放在构造函数中的方法，会为每个子对象都反复创建函数对象的副本

所以：将来在构造函数中就不应该包含方法定义

解决：用继承来解决

1. 什么是继承：父对象中的成员，子对象无需再反复创建，就可直接使用，就像自己的成员一样
2. 何时：只要多个子对象，希望共用一个方法或属性，就可以用继承的方式来定义一次，反复只用=使用--解决内存
3. 如何：js中继承都是借助于原型对象实现的
4. 原型对象（prototype）：
5. .其实在创建每个构造函数时，都会附赠一个空对象，名为原型对象
6. 通过构造函数.prototype属性，获得这个构造函数对应的一个原型对象

比如：想获得.prototype属性：Student.prototype

1. 当用构造函数创建对象时，new的第二步司东伟新对象天使”\_\_proto\_\_”,将“\_\_proto\_\_”属性指向当前构造函数的原型对象

比如: 如果var lilei=new Student("Li Lei",11)

则new会自动: lilei.\_ \_proto\_ \_=Student.prototype。

(4). 结果:

a. 凡是这个构造函数创建出的新对象，都是原型对象的孩子(子对象)

b. 放在原型对象中的属性值或方法，所有子对象无需重复创建，就可直接使用

6. 如何为空的原型对象中添加共有的属性或方法: 强行赋值

构造函数.prototype.方法名=function(){

... ...

}

7. 原型对象中的方法如何使用: 子对象可以直接调用，像调用自己的方法一样

(1). 子对象.原型对象中的方法名()

(2). 原理: js引擎会先在当前子对象中查找是否包含该方法。

a. 如果找到，就优先使用当前子对象自己的方法

b. 如果在当前子对象中没找到想用的方法，则自动延"\_ \_proto\_ \_"去父对象(原型对象)中查找方法使用。

8. 总结:

(1). 今后，构造函数中应该只包含属性结构的定义

(2). 所有的方法，都应该强行添加到原型对象中，所有子对象共用一份——节约内存

1. 面向对象
2. 继承
3. 自有属性和公有属性

a. 什么是自有属性: 保存在子对象中，只归子对象独有的属性

b. 什么是共有属性: 保存在原型对象中，归当前类型下所有子对象共有的属性

c. 获取属性值时: 无论获取自有属性值，还是获得共有属性值，都可用"子对象.属性名"，无差别

d. 修改属性值时:

1). 如果修改一个子对象的自有属性，才可以“子对象.自有属性=新值”

2). 如果修改多个子对象共有的属性，必须使用原型对象亲自修改:

构造函数.prototype.共有属性=新值

3）. 错误做法！ 如果强行用子对象直接修改共有属性: 结果，原型队形中的共有属性保持不变，而是只给当前这一个子对象添加一个新的同名的自有属性。从此，这个子对象，因为已经有了同名的自有属性，就不会再使用共有属性。从此，共有属性变化，当前子对象的这个同名自有属性也不会跟随变化。从此，这个子对象和其他子对象，在这个属性的使用上，分道扬镳。

1. 内置类型的原型对象：
2. 什么是内置类型/对象:ES新标准中已经规定的，浏览器已经设置好的，我们可以直接使用的类型/对象
3. 11种：

String Number Boolean--包装类型（扩展） Array Date RegExp Math(不是类型，是一个对象，不能new)

Error Function Object global(不是类型，是一个对象，不能new,且在浏览器中被window替代)

1. 今后凡是可以new的，都是构造函数。只要是构造函数，都会牵扯出一个大家庭，至少包含2个人：

1）.构造函数（妈妈）：i.负责反复创建多个相同结构的子对象

ii.构造函数中的属性，会成为将来子对象中的自有属性

2）.原型对象（爸爸）：i.负责替该类型集中保管公有的方法

ii.爸爸有什么功能，孩子也有

d.比如:内置对象Array:包含2部分：

1）.function Array(){...内部代码 看不见...}

2).所有数组共用的函数，都放在Array.prototype中！

3）.如果想用的对象中没有想要的函数功能，可以自己在原型对象中自己添加一个函数，以后要将来这个调用这个对象时，都可以使用添加的这个函数

|  |
| --- |
| 注意：  今后不能用for in来遍历索引数组，  因为in不仅会遍历当前数组中的所有对象，还会遍历原型对象中的\_\_proto\_\_中的深紫色的内容，忽略浅紫色的内容， |

1. 原型链（儿子可以继承原型链上的一切属性和方法，但是不可以继承构造函数的 属性和方法，也就是妈妈的属性和方法）
2. 什么是原型链：有多级父元素逐级继承形成的链式结构
3. 保存着一个对象可用的属性和方法
4. 控制着成员的继承顺序：先自有，再共有
5. 多态
6. .什么是：同一个函数，在不同情况下表现出不同的状态
7. 包括2种情况：重载和重写（override）

推翻//覆盖

1. 重写：子对象中定义了和父对象中同名的成员
2. 为什么重写：因为虫父对象继承来的东西不总是好用的
3. 何时重写：只要从父对象继承来的某个东西不好用，就要重写
4. 如何：只要在子对象中定义和父对象名称相同的成员，从此在子对象中使用这个成员时，都会优先使用自己定义的成员，而不再使用父对象中的成员
5. 自定义继承：如果觉得整个父对象都不好用，可以换爹：2种
6. .只修改一个子对象的父对象
7. 不推荐的做法：子对象.\_\_proto\_\_=新对象 问题：不是所有浏览器都开放\_\_proto\_\_
8. 推荐做法：Object.setPrototypeOf(子对象，父对象)

设置 子对象 的原型为 父对象

(2).同时修改对个子对象的爹

a.只要更换构造函数的prototype属性，指向新的原型对象即可，

构造函数.prototype=新的原型对象

1. 时机：应该再创建子对象之前就要更换，这样好处就是，子对象就会自动继承新的原型对象
2. ES5:ECMAScript语言（js语言的核心语法）标准的第五个升级版本

(学习ES5和ES6就是不系统的，都是给js语言打补丁)

1. 严格模式：
2. .什么是严格模式：比普通js运行机制更严格的模式，
3. 为什么：就得js语言存在着很多广受诟病的缺陷
4. 何时：今后所有的js程序都要运行在严格模式下
5. 如何启用严格模式：只要在当亲代码段的顶部添加字符串“use strict”。 “ 用 严格”
6. 新要求：4个
7. 禁止给未声明的变量赋值：
8. .旧的js中：强行给未声明的变量赋值，会自动在全局创建该变量----容易造成全局污染
9. 静默失败升级为错误
10. 静默失败：执行不成功，但是还不报错
11. 缺点：极其不便于调试
12. 严格模式下，已经将所有的静默失败都升级为错误
13. 及其便于调试，避免歧义
14. 普通函数调用和匿名函数自调用中的this值默认指向undefined，而不再指window
15. 旧的js:普通函数调用和匿名函数自调用中的this值默认指向window
16. 增加了造成全局污染的可能性
17. 严格模式下this指向undefined
18. 禁止使用arguments.callee:
19. 什么是：在函数内部获得当前函数本身的一种关键词
20. 何时：递归
21. 问题：如果在函数内，递归调用时写死当前函数的函数名，则一旦当前函数名改变，必须要记得同时修改递归内写死的函数名，一旦忘改就会发生----紧耦合
22. 解决：在函数内部用arguments.callee自动获得当前函数对象本身，直接用当前函数对象递归调用，与函数名无关了
23. 为什么严格模式下要禁用：因为递归调用的效率极低（重复计算量太大）
24. 所以严格模式强烈不建议使用递归调用，会报错
25. 解决：今后多数的递归调用都可用循环来解决，但是难度极高
26. 总结：今后工作中，改用递归还是首选递归。除非影响效率了，才使用循环
27. 保护对象
28. 问题：旧的js中的对象毫无自保能力，对象的结构和属性值可以被任意修改
29. 解决：其实ES5中，已经提供了保护对象属性和结构的新方法
30. 保护对象的属性
31. .什么是：阻止对对象的属性值进行不符合规定的篡改
32. 如何：
33. 其实，ES5已经把每个对象的属性值，变成一个微缩的小对象
34. 获得对象中一个属性的描述小对象: (了解)

var 属性的描述对象=Object.getOwnPropertyDescriptor(对象, "属性名")

获得自己的属性的描述信息

c.如何修改对象属性的开关:

1. 强调：禁止使用.直接访问属性的开关，不许用专门的函数
2. 如果只修改一个对象的属性开关：

（重新）定义 属性

Object.defineProperty(对象，“属性名”，{

开关名：true或false,

... : ....

})

3)问题：defineProperty方法只能一次修改一个对象中的一个属性的开关。如果对个属性都要修改开关，代码就会很繁琐

1. 用一个函数批量修改一个对象中的多个属性的开关;

Object.defineProperties(对象，{

属性名：{

开关：true或者false,

... : ...

}

属性名：{

开关：true或者false,

... : ...

}

})

1. 问题：开关的保护功能很弱，不灵活，无法使用自定义的规则保护属性值
2. 解决：给程序的属性请保镖--访问器属性
3. 什么是访问器属性:不实际存储属性值,仅提供对另一个保存数据的属性的保护
4. 为什么：属性的开关不够灵活，无法自定义一规则灵活保护属性值
5. 何时：今后只要用自定义规则，灵活保护属性值时，都用访问器属性
6. 如何:2步：
7. 定义访问器属性保护另一个保存数据的属性：2步

第一步：先要将保护的属性，隐姓埋名，半隐藏

第二步：为受保护的属性，请保镖：

\* 保镖就是访问器属性，但是为大学添加访问器属性，不能直接在对象的{}内添加。只能通过Object.defineProperty()或者Object.Propertise()添加

\*保镖要冒名顶替原属性名：

访问器属性的名称，应该和想要保护的那个数据属性名一致，才能起到替身的作用

\*保镖一请就是一对：

get:function(){ return this.受保护的变量}

set:function(value){...}

其中get和set不能改变，必须折磨写

1. 通过访问器属性来操作受保护的数据属性：

1）用访问器属性保护属性不是为了组织大家使用，而是为了保护大家合理的范围内使用属性

1. 保护对象的结构：3中级别
2. .防扩展：阻止为对象添加新属性
3. 旧的js中：可以随时为对象添加新属性
4. 如何禁止为对象添加新属性

Object.preventExtensions(对象名);

1. .密封：
2. 什么是：既阻止给对象添加新属性，又阻止删除对象中的现有属性
3. 为什么：因为对象中的属性都应该是禁止删除的，所以要给每个属性写configurable:false,太麻烦
4. 何时：今后几乎所有的对象都要密封
5. 如何：Object.seal(对象名)

密封

1. seal()做了2件事：
2. 自动添加了Object.preventExtensions()阻止对当前对象的扩展
3. 自动为每个属性添加configurable:false,从此不再手动为每个属性添加configurbable:false,
4. .冻结：
5. 既不能添加现有属性，又不能修改属性值
6. 何时：如果多个模块共用的对象，就不应该让某一个模块擅自修改对象的属性值，
7. 如何：Object.freeze(对象)

冻结

1. 原理：做了3件事：

禁止给对象添加新的属性：Object.preventExtensions()

禁止删除对象中的属性：configurable:false,

禁止修改对象中的属性：wriable:false,

1. Object.create()
2. 什么是:基于一个现有对象，创建新的子对象，来继承这个父对象
3. 简单说：就是没有构造函数，也能创建子对象，来继承父对象
4. 何时：没有构造函数时，也想创建子对象
5. 如何：
6. 只创建子对象
7. var 子对象=Object.create(新对象)
8. 2件事：
9. 创建一个新的空对象
10. 自动设置新的空对象\_\_proto\_\_来继承父对象
11. 既创建子对象，尤为子对象添加自有属性
12. var 子对象=Object.create(父对象，{

自有属性名{

value:属性值，

writable：false,

enumerable:false,

configurable:false

}

自有属性名{

value:属性值，

writable：false,

enumerable:false,

configurable:false

}

})

1. 做了三件事

1)创建一个新的空对象

2)自动设置新的空对象\_\_proto\_\_来继承父对象

3)

1. call/apply/bind:都能替换函数中不想要的this为想要的对象
2. 何时：如果一个函数中的this 不是自己想要的，就可以用call/apply/bind来替换为自己想要的对象
3. 如何：2种情况
4. 只在调用函数时，临时替换一次this为指定的对象
5. 要调用的函数.call(替换this的对象)
6. 结果：
7. 会调用一次该函数
8. 在本次调用时，临时替换this为自己想要的对象
9. 如果要调用的函数需要调用多个实参值，但是实参值却放在一个数组中
10. 解决;可用apply代替call,既可以替换this,又能打散数组
11. 如何：要调用的函数.apply(要替换this的对象，数组名)
12. 原理：
13. apply拥有和call一样的功能，都要先调用函数，并替换this为指定对象
14. 但是apply比call多一个功能，先将数组打散，后再传参
15. 基于原函数，创建一个新函数副本，，并永久的替换this
16. 问题：需要反复的替换this后的函数，每次要写call，很繁琐
17. 解决：用原函数.bind()替代.call(),创建一个指定对象专属的函数副本，永久的绑定函数副本中的this为指定的对象
18. 如何：var 新函数名=原函数.bind(替换this的对象)
19. 原理：
20. .bind()会创建一个和原函数一摸一样的新函数副本，原函数保持不变
21. 只不过.bind()将新函数中的this,永久的指向替换的对象
22. 结果：将来反复调用函数，即使不传入替换this的对象，也可以保障this为指定的对象
23. 如果部分实参值也可以永久固定不变，也可以用.bind()提前绑定到形参变量上

Var 新函数名=原函数名.bind(替换this的对象，一个固定不变的或多个实参值);

1. 原理：
2. .bind()会创建一个和原函数一模一样的新函数副本
3. 永久的绑定新函数副本中的this为指定对象
4. 而且永久绑定一个或多个实参值到形参值变量上。(可以绑定多个实参值，但是要按顺序绑定)
5. 强调：已经被.bind()永久绑定的实参值，将来在调用实参值时。无需重复传入，只要从后续未绑定的实参值继续传入有即可
6. 何时：今后只要希望反复适用替换this后的函数时，不止一次使用，都要用.bind()代替.call()
7. 数组新增的函数
8. 判断：2个
9. .判断数组中是否包含符合条件的元素
10. var bool=arr.some(function(){

Retuen 判断条件

}

1. 原理：
2. .some中封装了for循环，自动遍历数组中每个元素
3. 每次遍历一个元素，就会自动调用一次回调函数，在调用回调函数时

i.自动将当前正在便利的元素传递给回调函数的形参val

ii.回调函数内判断当前元素是否符合要求，并且返回判断结果

3）.some函数会根据回调函数的判断结果，决定有没有必要向后执行

i.如果当前元素结果返回true,说明当前元素符合条件，则循环不在继续，整体返回true，说明找到了符合要求的元素

ii.如果当前元素判断结果返回false,活命当前元素不符合条件，则循环继续执行，如果循环结束返回false,则当前元素没有符合条件的元素

(2). 判断数组中是否所有元素都符合条件

a. var bool=arr.every(function(val){

return 判断条件

})

b. 原理:

1). every中封装了for循环，自动遍历数组中每个元素

2). 每遍历一个元素，就自动调用一次回调函数，在调用回调函数时

i. 自动将当前正在遍历的元素值传递给回调函数的形参val

ii. 回调函数内判断当前元素值是否符合要求，并且返回判断结果

3). every函数会根据回调函数返回的判断结果，决定有没有必要继续向后执行

i. 如果当前元素判断结果返回false，说明当前元素不符合条件，则循环结束，并整体返回false，说明当前数组不是所有元素都符合要求

ii. 如果当前元素判断结果返回true，说明当前元素符合条件，则循环继续执行。如果可以走到最后一个元素，而中途不退出循环，则说明所有元素都符合要求，则整体返回true。

一．ES5:新增数组函数

1.遍历：2个：

（1）.仅单纯遍历原数组中每个元素

a. arr.forEach(function(val){

对val的操作

})

b.原理

1）forEach中也自带for循环，自动遍历数组中每个数组的元素

2). 每遍历一个元素就自动调用一次回调函数，并自动将当前元素值传入回调函数的形参val

3). 在回调函数中对当前元素执行操作

d. forEach vs 普通for循环

1). 是否可更简化:

i. for: 是固定写法，几乎不可简化

ii. forEach可用箭头函数简化

2). 灵活性:

i. for: 极其灵活: 可改变遍历顺序(反向遍历)，可改变遍历的步调(2,4,6...)，可随时break退出循环！

ii. forEach: 不够灵活: 只能从前向后挨个遍历，且不支持break随时退出。

3). 可遍历的对象类型来说:

i. for可遍历一切数字下标的东西: 索引数组和类数组对象都行(都是数字下标)

ii. forEach是数组家函数，只能被索引数组使用。类数组对象无法使用

4). 结论: 因为绝大多数循环都不需要改变遍历的顺序和步调，所以，能用forEach，当然用forEach简写。除非特殊循环，才被迫改用for循环。

1. .遍历数组中每个元素，修改后，放入新数组中返回
2. 过滤和汇总
3. .过滤(filter)：
4. 什么是：复制出原数组汇总符合条件的元素组成新数组返回，原数组保持不变
5. 何时：只要想仅仅获得数组中符合条件的元素，都用过滤
6. 如何：

Var 新数组=arr.filter(function(val){

Return 判断条件

})

1. 原理：
2. arr.filter()先创建一个空数组等着
3. 然后arr.filter()才开始遍历数组中的每个元素
4. 每遍历一个元素就调用一次回调函数，并将当前元素自动传入到回调函数形参val中
5. 回调函数内判断当前元素是否符合条件，返回判断结果
6. .汇总
7. 什么是：遍历数组中每个元素，并对所有元素进行统计，最终得到一个统计结果
8. 何时：今后只要希望根据数组中的内容统计出一个结果时，就用汇总
9. 如何：(数组元素求和)：

var 统计结果=arr.reduce(function(prev,val){

prev+=val;

return prev;

},

起始值

)

1. 原理：
2. reduce开始执行时，会将起始值保存在变量prev中
3. 然后才开始遍历数组中的每个元素
4. 每次遍历一个元素，就会自动调用一次回调函数，并传入2个值，变量prev中的临时汇总值，会交给prev，当前元素值交给val

4). 回调函数中应该将当前元素值累加到临时汇总值prev变量上。并返回新的临时汇总值

1. 5). Reduce会将回调函数返回的新的汇总值保存到变量prev上，供下一轮汇总使用。
2. 6). 当循环结束后，reduce会返回变量prev中保存的最终的汇总值

\*\*\*es6\*\*\*

二. 模板字符串——重要，同将来vue框架中绑定语法的要求！

1. 什么是: 支持换行和动态生成内容的特殊字符串

2. 为什么: 因为传统的用+实现的字符串拼接和算数计算的加法极容易混淆

3. 何时: 今后，只要动态拼接字符串，都要用模板字符串代替普通的+拼接

4. 如何:

(1). 整个模板字符串必须用一对儿反引号包裹`——键盘左上角ESC正下发那个按键

(2). 在模板字符串中支持换行

(3). 模板字符串中用js变量和表达式动态生成的部分，必须用${}包裹

5. 强调: 模板字符串最终也是一个普通的字符串，只不过需要动态拼接内容而已。所以，能用字符串的地方，都能用模板字符串。

6. 模板字符串中${}里都能放什么：

(1). 能放: 变量、各种计算/比较等js表达式、三目、访问数组元素、调用方法，可以创建对象——凡是有返回值的合法的js表达式，都可放在${}中

(2). 不能放: 程序结构(分支和循环)以及没有返回值的js表达式

三. let

1. 什么是: 专门代替var来声明变量的新的关键词

2. 为什么: var有问题:

(1). var会被声明提前——打乱程序正常的执行顺序

(2). var没有块级作用域——代码块(分支和循环)内的变量很有可能影响块外的代码

3. 何时用let: 今后几乎所有的var，都会被let代替！

4. let的优点:

(1). 不会被声明提前

(2). 为js程序增加了块级作用域(分支和循环的{}从此也变成了一级作用域)

5. let的原理:

(1). let会被自动翻译为匿名函数自调！形成一级临时的作用域，就阻止了变量被声明提前

(2). 双保险: let在底层根据需要自动改变变量名

总结: let=匿名函数自调+自动修改变量名

7. let的小脾气:

(1). 在全局let一个变量，这个变量也不会保存在window中！

(2). 不能在let创建一个变量前，提前使用该变量，会报错

(3). 同一作用域内不允许重复let两个同名的变量

8. const:

(1). const原本是专门声明常量(值不能改变的变量)的关键词——第一阶段讲过

(2). 随着es6标准的提出，let具有的特征，const也具有了，只不过const的变量值不能改变而已。

(3). 但是const的效率稍微比let高一点儿，所以，如果将来一个变量的值不会再改变，则首选用const声明。——也不是必须！

四．箭头函数

1.什么是：专门简化所有的function定义函数的语法

2.为什么：写function，太繁琐

3.何时：今后几乎程序中是看不到任何function的

4.如何：3步：

（1）.去掉function,在()和{}之间加=>;

(2).如果只有一个形参变量，则可以省略（）,但是没有形参时，不可以省略()

(3),如果函数体只有一句话，可省略{}，和结尾的；

如果仅有的一句话是return,则必须省略return

1. 特点：箭头函数的内部的this和外部的this保持一致
2. .今后希望函数中的this与函数外的this保持一致时，或函数根本不涉及this时，则可以随意改为箭头函数
3. 但是，如果今后函数中的this刚好不希望与外部this保持一致时，就不能用箭头函数，

6.ES6新规定：所有对象的方法可以省略“:function”

 但是，省略了：function的方法，绝不等于箭头函数，只是单纯的简写而已

箭头函数会影响this的指向，当变成箭头函数时，箭头函数就不是一个作用域了，就不能规定this单单指向.前的对象了

五．for of

1.什么是：专门对普通for循环的简写

2.何时：只有那些从头到尾正向遍历所有元素的标准for循环才能用for of简写

3.如何： 三个都是数字下标的东西

for(var 变量 of 索引数组或类数组对象或字符串){

... : ...

}

4.fo会自动遍历后边数组或类数组对象中的每个元素的值，一次保存到of前的变量中。但是of无法获得下标位置！所以操作中涉及下标位置的循环，也不能用for of简写

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | for循环 | for of | arr.forEach | for in |
|  | 索引数组 | √ | √ | √ |  |
|  | 类数组对象 | √ | √ |  |  |
|  | 字符串 | √ | √ |  |  |
| 自定义下标，没有规律 | 关联数组/对象 |  |  |  | √ |

1. for of 的局限
2. .只能从前到尾正向遍历，无法改变遍历顺序
3. 只能卓哥遍历数组中的每个元素，无法隔着遍历
4. 无法在遍历时获得正在遍历到的下标位置
5. 参数增强:ES6对函数参数的使用提供了一些新的方法
6. 参数默认值：
7. .何时：即使调用函数时，没有传入实参值，也希望形参变量有默认值使用时
8. 如何：function 函数名(形参变量=默认值，...,...){....}
9. 结果：
10. 如果将来调用这个函数时，提供了实参值，则新参变量采用迂回提供的实参值
11. 如果将来调用这个函数时，没有提供实参值，则新参变量设置的默认值采用提亲准备好的默认值

（4）.局限：通常只能采用最后一个形参值有没有的情况下,取默认值的情况

2.剩余参数(rest):专门代替arguments来处理不确定实参值个数的情况

(1).为什么：arguments的问题：

a.箭头函数不支持arguments

b.是类数组对象，不是数组家的孩子

c.只能获取所有实参值，不能有选择的获取部分实参值

d.名字太长，且下标没有意义

(2).何时：今后只要不确定传入函数的实参值个数时，都要用剩余参数语法代替arguments

(3).如何：在定义函数时，形参变量列表中：

function 函数名(形参1，形参2，...数组名){

//...专门负责收集除前几个形参已经匹配的实参值之外的所有多余的实参值。

//如果没有之前的形参，则...也可以获得所有实参值

//会将收集到的实参值，放入一个自己定义名称的标准的数组中

}

1. .优点：

a.箭头函数不支持剩余参数语法

b.是标准的数组，可以使用数组家的孩子

c.只能获取所有实参值，不能有选择的获取部分实参值

d.名字太长，且下标没有意义

3.打散（spread）:

(1).什么是：先将数组打散为多个值，然后再分别传给函数的形参变量

（2）.何时：当调用函数时，函数本来需要多个实参值，但是，给定的实参值却是放入一个数组中给的，发生了不一致。就要先打散数组为多个值，在分别传给函数的形参值

（3）.如何：调用函数时：

函数名（...数组），

1. .执行过程：先将数组打散为多个值，然后再将打散后的多个值分别一一对应传给函数的形参值的过程
2. 总结：
3. 定义函数时，形参变量列表结尾的...arr是收集剩余实参值的意思
4. 调用函数时，实参值列表结尾的...arr是打散数组再传参的意思
5. 解构：
6. 什么是：从一个大的对象或数组中仅仅提取除个别属性或者方法单独使用---简化程序
7. 为什么：
8. .访问数组中的一个元素值，比如：数组名[下标],但是这种语法很麻烦，且下标是数字，没有意义
9. 访问对象中的属性或者方法时，总要带着”对象名.”,前缀，很麻烦

3. 何时: 今后如果只想使用一个大的对象或数组中的个别成员时，就用解构

4. 如何: 分为数组解构、对象解构、参数解构

5. 数组解构: 从一个大的数组中提取出个别元素值单独使用

(1). 如何:

//先将要接收元素值的所有变量，装扮成一个数组的样式

var [变量1, 变量2, ...]=数组

// 0 1

(2). 结果: 相同下标位置的变量，自动收到数组中相同下标位置的元素值

变量1=数组[0]

变量2=数组[1]

... = ...

从此，数组中某个元素值，就可以脱离开数组，以变量方式单独使用

6.对象解构：

(1).什么是：从一个大的对象中提取出个别成员，单独使用

(2).何时：今后如果一个大的对象中只想使用个别成员时，就可以那个对象解构

(3).如何：

//先把要接收的对象成员的变量装扮成和对象相同的结构

var {属性名：变量1，属性名：变量2，...:...}=对象

1. .结果：等号左边{}中的相同属性名的变量会自动获得等号右边相同属性名的属性值
2. 解构不成功如果等号左边写的属性名，等号右边的对象中没有，则不会报错，而是返回undefined而已。
3. 简写：如果对象中原来的属性名或者方法名已经很好了，不需要改名，则解构时，等号左边的{}中只写一个名字就够了

var {成员名1，成员名2}=对象;

{}中的成员名，一个名字两用：既可以当属性名去对象中配对要结构的属性，又用来接收将来属性值单独使用的变量名，ES6规定，只写一个就够了

7.参数解构

(1).什么是：其实就是对象解构在函数内会传参时的应用

(2).何时：如果多个形参变量不确定将来没有值，而且又要求传入的实参值，必须给指定的形参变量，不能错位

(3).如何;步：

a.定义函数时，做鱼的形参变量都要放在一个对象结构中,将形参列表装扮成对象的样子

function 函数名（{

属性名1：形参名1，

属性名2：形参名2，

... : ...

}）{

//函数体

}

b.调用函数时：实参列表也要装扮成对象，且实参列表中的对象结构的属性名，应该与形参列表中的属性名一致

函数名（{

属性名1：实参值1，

属性名2：实参值2，

... : ...

}）

1. .结果：.
2. Class
3. 什么是:集中保存的一个类型的构造函数和原型对象方法的存储空间
4. 为什么：因为旧js中的构造函数和原型对象虽然同属于同一类型，但是写法上是分开的，不和封装的要求
5. 何时：今后只要创建一种新类型，想反复创建相同结构的对象时，都要用class代替原来的构造函数和原型对象分散的写法
6. 如何：
7. .用class()来包裹原来的分散写的构造函数和原型对象的方法
8. 所有构造函数的类型名都提升为整个class的类型名，所有构造函数从此统一 用constructor关键词创建
9. 所有直接放在class中的方法，默认自动会保存到原型对象中，不用再加“构造函数.prototype”前缀
10. class间继承：
11. .问题：程序中两个class之间包含部分相同的属性结构定义和定义方法

(2). 解决: 再额外定义一个父级class，统一保管两个子级class相同部分的内容

(3). 如何: 2大步

a. 第一步: 先额外定义一个父类型: 2小步

1). 在父类型的构造函数中，集中定义所有子类型相同部分的属性结构

2). 在父类型的class中，集中定义所有子类型相同的方法定义

b. 第二步: 让子类型class，继承父类型class: 2小步

1). 使用extends关键词，让子类型继承父类型

结果: 子类型.prototype.\_ \_proto\_ \_ = 父类型.prototype

结果: 将来子类型new出来的孙子一级对象，即可用子类型的原型对象方法，又可用父类型的原型对象方法

2). 在子类型构造函数内，先用super关键词，调用父类型的构造函数。

(super关键字是extends赠送的，专门指向父类型构造函数的关键词)

结果: 父类型的构造函数和子类型的构造函数共同协作创造出将来的子类型的子对象。

结果: 将来子类型的子对象身体里，包含着来自于父类型构造哈数和子类型构造函数，两个构造函数中规定的所有属性结果。

1. promise:
2. 什么是：专门解决回调地狱问题的新编程的方法
3. 问题：回调地狱(callback hell)
4. .何时：多个异步函数，却要求顺序执行
5. 如何：

a.错误做法：单纯顺序调用多个异步函数，无法保证多个异步函数顺序执行，因为异步函数之间默认谁也不等谁

b. 正确但不好的做法: 用回调函数做: 2步:

1). 在前一个异步函数定义时，增加一个形参变量"回调函数"

在前一个异步函数内，当异步任务最后一句话执行完之后，自动调用形参中传入的回调函数

2). 在调用前一个异步函数时，提前将调用下一个异步函数的语句，包裹在一个匿名函数function()内，提前保存到前一个异步函数的回调函数形参变量上

c. 结果: 当前一个异步函数执行完自己的最后一句话之后，自动调用回调函数，则回调函数中提前保存的下一个异步函数才开始执行

d. 问题: 如果要求多个一个函数必须顺序执行，则回调函数的嵌套会很深！——回调地狱

3. 解决: 今后只要多个异步任务需要顺序执行，都可用Promise技术来代替传统的回调地狱

(1). 如果打算用promise，就不要定义回调函数参数

(2). 如何: 2步

a. 先改造原异步函数，变成一个格子间

function 前一项任务函数(不要写回调函数参数了){

return new Promise( //创建一个格子间，包裹原异步任务的代码

function(door){ //赠送了一个门

//原异步任务代码

//异步任务最后一句话执行完

//手动调用door()开门——结果: 才开始执行.then()链接的下一项任务

}

)

}

b. 调用时:

前一项异步任务.then(下一项任务函数).then(...)

//如果下一项任务函数也是一个格子间

强调: .then()中的下一项任务函数因为不是立刻执行，所以不要加()！