正课:

1. RegExp

2. Function

1. RegExp:

什么是: 封装一条正则表达式，并提供使用正则表达式执行验证和查找功能的API

何时: 只要在程序中存储正则表达式，或使用正则执行验证和查找时

如何:

创建: 2种:

1. 直接量: var reg=/正则/ig

问题: //中的正则是写死的，不能动态拼接生成

何时: 如果正则表达式可以写死，不需要动态生成时

2. 用new: var reg=new RegExp("正则","ig")

何时: 如果正则表达式需要动态生成！

API:

1. 字符串格式验证:

验证一个字符串的格式是否符合正则表达式要求

如何: var bool=reg.test(str)

用reg检查str的格式是否正确

问题: 默认在str中只要能找到符合reg要求的子内容，就算通过！

解决: 今后，凡是验证，必须前加^，后加$

2. 查找关键词:

查找4: 即查找每个关键词的内容，又查找每个关键词的位置

var arr=reg.exec(str)

在str中查找下一个符合reg要求的关键词的内容和位置

返回值: 只返回本次找到的一个关键词的内容和位置

arr:[ 0: "关键词", index: i ]

如果找不到，返回null！

聪明: 可用循环反复调用exec，不断找下一个！

exec()每次可自动从下一个开始找！

强调: reg必须加g，才能先后找！

2. Function

什么是: 封装一段可重用的代码段的对象

何时: 重用一段代码

如何:

创建: 3种:

1. 声明:

function 函数名(形参,...){

函数体;

return 返回值

}

何时定义参数: 函数本身缺少必须的值

何时定义返回值: 调用者需要获得执行结果时

本质: 函数其实也是一个引用类型的对象

函数名其实仅是一个普通的变量

函数名变量使用内存地址指向函数对象

问题: 声明提前——广受诟病的缺陷！

声明提前: 在程序开始执行前，引擎会将var声明的变量和function声明的函数，提前到当前作用域的顶部，集中创建。赋值留在原地

解决: 改为赋值方式创建函数:

2. 赋值方式:

var 函数名=function(...){ ... }

3. 用new: ——了解:

var 函数名=new Function("形参1","形参2",...,"函数体")

重载(overload):

什么是: 相同函数名不同参数列表的多个函数，在调用时，可根据传入参数的不同，自动选择匹配的函数执行

为什么: 减少API的个数，减轻调用者的负担

何时: 一件事，可根据传入参数的不同，执行不同的逻辑时

如何:

问题: js中不支持重载写法:

因为: js中不允许多个同名函数同时存在

结果: 只有最后一个函数会覆盖之前所有同名函数

解决: 靠arguments对象

什么是: 函数内专门接收所有传入函数的参数值的类数组对象

类数组对象: 长的像数组的对象:

vs 数组: 相同: 1. 下标, 2. .length, 3. for遍历

不同: 类型不同——API不通用！

如何:

创建: 不用创建！每个函数中自带！

实现重载: 根据arguments中的参数个数和参数值的不同，动态执行不同的逻辑

匿名函数:

什么是: 定义函数时，不指定函数名

为什么: 2个原因:

1. 节约内存: 因为没有函数名变量引用着，用完立刻释放！

2. 避免函数重名

何时: 只要一个函数仅使用一次，不会被重用

如何: 2种:

1. 回调函数:

自己定义函数后不调用，而是交给别的函数去调用

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

//sort()执行完，内部匿名函数自动释放！

str.replace(/正则/,function(kw){ return 新值; })

2. 匿名函数自调:

定义匿名函数后，立刻调用自己执行。调用后立刻释放。

(function(...){ ... })() //调用后，匿名函数立刻释放！

作用域和作用域链:

作用域(scope):

什么是: 一个变量的可用范围——使用角度

其实是一个保存变量的对象——存储角度

为什么: 避免作用域内外的变量间互相干扰

包括: 2个作用域

1. 全局作用域:

什么是: 保存全局变量/函数的对象

nodejs: global

浏览器: window 代替了 global

2. 函数作用域:

什么是: 保存函数内局部变量的对象

函数的生命周期:

1. 程序开始执行时:

调用浏览器主程序main()启动浏览器

创建执行环境栈, 将main()的调用保存其中

main()创建window对象——全局作用域对象:

保存了所有内置对象和方法  
 准备保存将来声明的所有全局变量

2. 定义函数时:

在window中添加函数名变量

在window外创建函数对象，保存函数定义

函数名变量引用函数对象

函数对象的scope属性指回自己诞生自的对象

3. 调用函数时:

在执行环境栈中添加本次函数调用的记录

为本次函数调用创建临时作用域对象——函数作用域对象

在函数对象中添加函数中声明的所有局部变量

调用时变量使用顺序:

只要自己有，就用自己的

局部没有，才去全局

4. 函数调用后:

执行环境栈中本次函数调用的记录出栈

->导致函数作用域被释放

->导致局部变量一同释放！——局部变量不可重用

作用域链(scope chain):

由多级作用域逐级引用，形成的链式结构

两大作用: 1. 保存所有全局和局部变量

2. 控制着变量的使用顺序: 先局部，后全局

\*\*\*\*\*闭包(closure)

什么是: 即重用变量，又保护变量不被污染的一种机制

为什么: 全局变量和局部变量都有不可兼得的优缺点:

全局变量: 优: 可重复使用

缺: 随处可用——极容易被污染

今后尽量少用或不主动使用全局变量

局部变量: 优: 仅函数内可用，不会被污染

缺: 不可重用

何时: 即重用变量，又保护变量不被污染

如何:

1. 用外层函数包裹要保护的变量和使用变量的内层函数

2. 外层函数将内层函数返回到外部

3. 使用者调用外层函数，获得内层函数对象，保存在变量中反复使用

闭包形成的原因: 外层函数的作用域对象无法释放

主动使用闭包:

为一个函数，绑定一个专属的变量

课后任务:

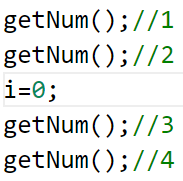
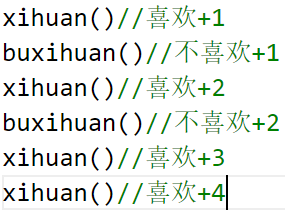
(1)复习: 闭包的使用方法

(2)作业: 完成课后练习:  
 题目要求:

1. 使用闭包实现"喜欢+1"和"不喜欢+1"效果

2. 使用闭包实现取号机程序

运行效果:

提示: 无

(3)项目:

题目要求:

运行效果: