day1

张东

18301092802

1. 正则表达式: 见ftp中regexp.pdf

2. String的正则API

查找: 4个场景:

1. 查找一个固定的敏感词出现的位置:

回顾: str.indexOf()

var i=str.indexOf("敏感词",starti)

在str中，从starti位置开始，查找下一个“敏感词”的位置

省略第二个参数starti，默认从0位置开始查找

返回值: 如果找到，返回敏感词第一个字的位置

如果找不到，返回-1

问题: 只能查找一种固定的敏感词

2. 模糊查找符合正则表达式要求的敏感词:

var i=str.search(/正则/);

在str中查找第一个符合正则要求的敏感词的位置

返回值: 如果找到，返回敏感词第一个字的位置

如果找不到，返回-1

问题1: 正则默认都是区分大小写的

解决: 在第二个/后加后缀i, ignore

问题2: 只能返回位置i，无法返回敏感词的内容

3. 查询敏感词的内容: 2种情况:

1. 只查看第一个敏感词的内容和位置:

var arr=str.match(/正则/i);

在str中查找第一个符合正则要求的敏感词的内容和位置

返回值: 如果找到: arr:[ 0:"敏感词", index: 位置i ]

补充: js中其实一切都是关联数组

如果找不到: 返回null

强调: 如果一个函数有可能返回null，都要先验证不是null，再使用。

问题: 正则表达式默认只找到第一个就退出

解决: 在第二个/后加后缀g, global(全部)

2. 查找所有敏感词的内容:

var arr=str.match(/正则/ig)

在str中查找所有符合正则要求的敏感词的内容

返回值: 如果找到: arr: [ 敏感词1, 敏感词2,... ]

如果没找到，返回null

问题: 只能获得内容，无法获得位置

4. 查找每个敏感词的内容和位置: reg.exec()

替换: 2种:

1. 简单替换: 将所有敏感词都替换为统一的新值:

str=str.replace(/正则/ig,"新值")

查找str中符合正则要求的所有敏感词，替换为统一的新值

强调: 所有字符串API是无权直接修改原字符串，总是返回新字符串作为处理结果，所以必须用变量接住返回的新字符串。

2. 高级替换: 可根据每次找到的敏感词不同，动态生成替换的新值

str=str.replace(/正则/ig,function(kword){

return 新值

});

查找str中每个符合正则要求的敏感词，每找到一个敏感词，就自动调用一次回调函数。每次调用时，都自动传入本次的敏感词作为参数kword，再将返回的新值，替换回字符串中。

衍生操作: 删除找到的敏感词，其实就是替换为""

切割: 将一个字符串按指定的分隔符切割为多段子字符串:

2种:

1. 简单切割: 切割符是固定的:

回顾: var arr=str.split("切割符")

2. 复杂切割: 切割符不是固定的

var arr=str.split(/正则/)

day2

1. RegExp对象: 正则表达式(Regular Expression)

什么是: 在程序中专门保存一条正则表达式，并提供用正则表达式执行验证和查找功能的API 的对象

何时: 只要在程序中保存和使用正则表达式时，都要先创建正则表达式对象

如何:

创建正则表达式对象: 2种

1. 用//直接量方式创建: var reg=/正则/ig;

何时: 如果正则表达式不需要动态生成时

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

何时: 如果正则表达式需要用程序动态生成时

API:

1. 验证: var bool=reg.test(str)

用正则表达式reg检查一个字符串str的格式是否符合要求。

返回值: true/false

问题: 正则表达式，默认只要找到符合规则的内容，就返回true，不要求从头到尾完整匹配——所以，也有人用reg.test()做查找关键词

解决: 今后凡是验证必须前加^后加$，表示从头到尾完整匹配

2. 查找敏感词: reg.exec(str)

什么是: 查找str中每个符合reg要求的敏感词的内容和位置

何时: 只要既查找每个关键词的内容，又查找每个关键词的位置

如何:

var arr=reg.exec(str);

执行过程:

每次只能查找一个敏感词，

返回: arr: [ 0: "敏感词", index: 位置i ] ——同match不加g的情况

反复调用reg.exec可自动向后找！

找不到，返回null

2. Function:

什么是: 专门保存一段可重用的代码段的程序结构，再起一个名字

何时: 只要一段代码可能被反复使用，都要定义为函数

如何:

创建函数: 3种:

1. 声明方式: function 函数名(形参列表){

函数体;

return 返回值;

}

问题: 会被声明提前

解决: 改声明方式为赋值方式

2. 赋值方式: var函数名=function (形参列表){

函数体;

return 返回值;

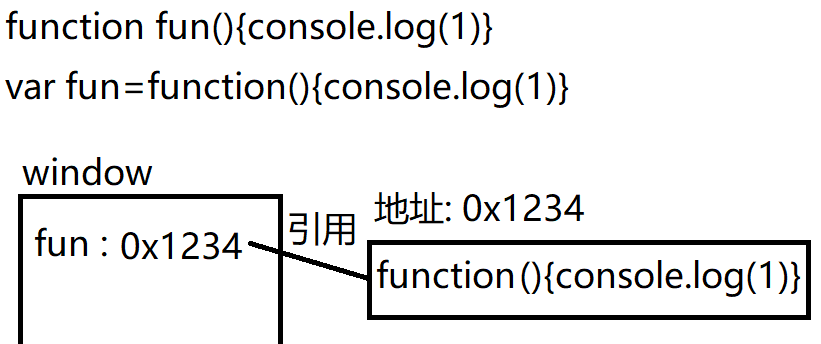
}

优点: 因为是赋值方式，不会被声明提前

揭示了js中函数的本质:

1. 函数其实也是一个保存一段代码的对象

2. 函数名其实只是一个普通的变量而已。函数名变量通过函数对象的地址引用着函数对象。所以使用函数名变量等效于使用函数对象本身。



3. 用new Function构造函数创建函数对象:

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，可根据传入实参的不同，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，可能根据传入参数的不同，执行不同的操作时

比如: arr.splice():

删除元素: arr.splice(i,n)

插入元素: arr.splice(i,0,新值1,...)

替换元素: arr.splice(i,n,新值1,...)

如何:

问题: js语法默认不支持重载

因为js中不允许多个同名函数同时存在

如果多个同名函数同时存在，则只有最后一个生效

解决: 借助arguments

arguments: 每个函数中自带的

自动接收所有传入函数的参数值的

类数组对象

类数组对象: 长得像数组的对象

vs 数组: 相同: 1. 下标, 2. length, 3. 遍历

不同: 类型不同！API不通用！

如何: 2步:

1. 仅定义一个函数，且不要定义形参

2. 在函数内判断arguments的length或内容，根据不同的参数，执行不同的操作

匿名函数:

什么是: 定义函数时，不指定函数名的函数

为什么: 节约内存

何时: 如果一个函数只使用一次，不会重用时，都要用匿名函数

如何: 2种；

1. 回调函数: 自己定义的函数，但是不是自己亲自调用，而是给别的函数去调用。

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

setInterval/setTimeout(function(){...},ms)

str.replace(/正则/ig,function(kword){return 新值})

2. 匿名函数自调:

什么是: 定义函数后，立刻执行！

为什么: 避免使用全局变量，造成全局污染

何时: 几乎所有的js代码都要装在匿名函数自调中

如何: (function(){ ... })()

作用域(scope)/作用域链: 参考2\_function\_4.pdf

闭包: 参考2\_function\_4.pptx

day3

1. OOP: 面向对象编程

什么是对象: 程序中描述现实中一个具体事物的属性和功能的程序结构

什么是面向对象编程: 程序都是先将数据和功能定义在对象中，再按需使用对象的数据和功能

为什么: 便于大量数据和功能的维护和使用

如何: 三大特点:封装，继承，多态

封装:

什么是: 创建对象集中保存一个事物的属性和功能

为什么: 便于大量数据和功能的管理和使用

何时: 要面向对象编程，都要先封装对象，再按需使用对象的属性和功能

如何: 3种:

1. 用{}直接量:

var 对象名={

属性名:属性值,

... : ... ,

方法名:function(){

... this.成员 ...

}

}

问题: 对象的方法要访问对象自己的属性或方法，无法直接使用

为什么: 所有对象的属性和方法默认都不在作用域链当中。任何程序无权自动进入对象中获取成员。

不好的解决: 在方法中: 对象名.成员名

为什么: 紧耦合，如果对象名发生变化，方法的内容必须同时修改才行。

好的解决办法: this

什么是this: 自动指向正在调用当前函数的.前的对象

何时: 只要对象自己的方法，要使用自己的成员，都要加this.

强调: 鄙视时，this与定义时无关，只和调用时有关

访问对象的成员: (属性和方法)

对象.属性名

对象.方法名()

2. 用new创建: 2步:

1. 先创建空对象: var obj=new Object();

2. 向空对象中添加新属性: obj.新成员=值/function(){...}

js中对象的特点:

1. 对象创建完成后，依然可继续添加新成员

2. 访问对象中不存在的属性，不报错！而是返回undefined

3. 访问对象的成员，既可用.，又可用["成员名"]

选择: 如果成员名是变化的，就必须用[变量]

如果成员名可以写死，就可用.简化

揭示: 其实js底层，所有对象都是关联数组！

所以，对象也可用for in遍历其中的属性

前两种方式的问题: 一次只能创建一个对象

如果反复创建多个相同结构的对象时，代码重复很多

解决:

3. 用构造函数反复创建统一类型的多个对象:

构造函数: 描述一类对象统一结构的函数

为什么: 重用创建对象的代码！

何时: 要反复创建多个相同结构的对象时

如何: 2步:

1. 定义构造函数:

function 类型名(){

//this->new->{}

this.属性名=值; //将来的对象中一定要有指定的"属性"

this.xxx = 值;

this.方法名=function(){//将来的对象中必须有指定的"方法"

... ...

}

}

2. 调用构造函数反复创建多个对象:

var obj=new Student(属性值,...)

new:

1. 创建一个新的空对象:{}

2. ?

3. 调用构造函数:

自动将构造函数中的this->new->{}

构造函数中的每一句话，都通过强行赋值的方式，为新对象{}添加规定的新属性

4. 返回新对象地址

问题: 放在构造函数中的方法定义，会在每个子对象中重复创建函数对象，浪费内存！

解决:

继承:

什么是: 父对象的成员，子对象无需重复创建，就可直接使用

为什么: 代码重用，节约内存

何时: 只要多个子对象需要相同的成员时

如何:

原型对象: 专门保存所有子对象共有成员的父对象

不用创建，定义构造函数时，附赠的

构造函数.prototype -> 当前构造函数的原型对象

如何继承: new的第2步，自动设置本次创建的子对象，继承构造函数的原型对象。

每个子对象都有\_ \_proto\_ \_属性，指向自己的父级对象

向原型对象中添加共有成员:

构造函数.prototype.成员=值/function(){...}

自有属性和共有属性:

获取属性值, 没有任何差别

差别在: 修改时: 自有属性，可用子对象修改

共有属性，不能用某一个子对象修改

只能用原型对象修改

内置对象的构造函数和原型对象:

内置对象: ES标准中规定的，浏览器已经定义好的可直接使用的对象

包括: 11个

String Number Boolean

Array RegExp Date Math

Error

Function Object

global

除了Math和global外，其余都是一种类型:

都包含2部分组成:

1. 构造函数: 创建该类型的子对象

2. 原型对象: 保存该类型的所有子对象共有的成员

原型链: prototype chain

什么是: 由多级父对象逐级继承，形成的链式结构

2大作用:

1. 保存了当前对象可用的所有属性:

2. 控制着属性的使用顺序

多态:

什么是: 一个函数在不同情况下表现出不同的状态

包括: 2种:

1. 重载:

2. 重写:override

什么是: 在子对象中定义和父对象同名的成员

何时: 只要从父对象继承来的成员不好用

如何:

day4

1. 自定义继承: 见PDF: 1\_自定义继承.pdf

2. 静态方法:

什么是: 不需要创建子对象，就可用构造函数直接调用的方法

问题: 原型对象中的方法，只能被当前类型的子对象使用

何时: 2种:

1. 如果一个函数，不确定将来使用它的对象的类型时

2. 如果一个函数在调用时，不需要提前创建子对象

如何: 构造函数.静态方法=function(){ ... }

调用时: 构造函数.静态方法()

强调: 调用静态方法之前不需要创建子对象

3. ES5:

1. 严格模式:

什么是: 比普通js运行机制要求更严格的模式

为什么: js语言有广受诟病的很多缺陷

何时: 今后所有js，都要运行在严格模式下.

如何: 在当前代码段的顶部: "use strict";

严格模式要求:

1. 禁止给未声明的变量赋值:

2. 静默失败升级为错误:

静默失败: 执行不成功，也不报错

3. 禁用了arguments.callee:

arguments.callee专门在调用函数时，获得当前函数本身

何时: 专门用于递归调用中获得当前函数

为什么禁用: 不再建议使用递归

因为: 递归重复计算量太大！效率极低

解决: 几乎所有的递归都可用循环代替

4. 普通函数调用中的this不再指向window

而是undefined

2. 保护对象:

问题: JS中对象毫无自保能力

解决: ES5提供了一系列措施保护对象

如何:

1. 保护对象的单个属性

ES5将对象的属性，分为2大类:

1. 命名属性: 可用.直接访问的属性

又分为: 2类:

1. 数据属性: 实际存储属性值的属性

2. 访问器属性: 不实际存储属性值，专门提供对另一个数据属性的自定义保护！

2. 内部属性: 实际存在，但不能用.访问到的属性

保护数据属性:

ES5将一个数据属性，又定义为一个缩微的小对象

属性: {

value: 实际存储属性值,

writable: true/false, //控制是否可修改

enumerable: true/false, //控制是否可被for in遍历

//只能防住for in，但是用.依然可访问

configurable: true/false //控制是否可删除该属性

//还控制是否可修改另外两个特性

//一旦改为false，不可逆！

//所以，修改其它两个开关时都会同时修改configurable:false,作为双保险。

}

获取一个属性的四大特性: 了解

var miniObj=

Object.getOwnPropertyDescriptor(obj, "属性名");

修改一个属性的四大特性:

Object.defineProperty(obj,"属性名",{

开关: true/false,

... : ...

})

同时修改多个属性的四大特性:

Object.defineProperties(obj,{

属性名:{

开关:true/false,

... : ...

},

... : {

}

})

问题: 开关不能用自定义规则保护属性

解决: 用访问器属性:

访问器属性:

何时: 只要用自定义规则保护对象属性时

如何:

前提: 先有一个隐藏的数据属性实际存储数据

再添加一个访问器属性来保护隐藏的数据属性:

Object.defineProperty(obj,"访问器属性名",{

get:function(){ return this.受保护的数据属性 },

set:function(value){

//value可接住将来要赋的新值

//如果value符合条件

//才this.受保护的数据属性=value

//否则

//报错

},

enumerable:true, //让访问器属性代替数据属性抛头露面

configurable:false

})

访问器属性何时起作用:

访问器属性的用法和普通属性是一样的

只不过:

1. 当试图获取访问器属性的值时，自动调用get()

2. 当试图修改访问器属性的值时，自动调用set()，并传入新值作为参数value

2. 保护对象的结构

day5

1. ES5

保护对象:

1. 保护属性:

2. 保护对象结构: 3个级别:

1. 防扩展: 禁止给当前对象添加新属性

Object.preventExtensions(obj)

2. 密封: 在兼具防扩展的同时，又禁止删除现有属性

Object.seal(obj)

原理: 1. Object.preventExtensions(obj)

2. 自动将所有属性的configurable都设置为false

3. 冻结: 在兼具密封的基础上，又禁止修改属性值

Object.freeze(obj)

原理: 1. Object.seal(obj)

2. 又自动将所有属性的writable都设置为false

Object.create(): 仅有父对象，也可创建子对象

何时: 如果没有构造函数也想创建子对象时

如何: var child=Object.create(father,{

//为child添加新自有属性

//语法相当于defineProperties

属性名:{

value:属性值,

开关: true/false,

... : ...

}

})

原理: 1. 创建一个空对象

2. 让空对象继承father

3. 为新对象添加自有属性

替换this:

何时: 如果一个函数调用时，其中的this不是想要的，就可替换为想要的对象.

如何: 3种:

1. 调用函数，并临时替换一次this:

call(): 要求传入函数的参数，是分别传入

apply(): 要求传入函数的参数，必须放在一个数组中整体传入。

何时: 如果参数值是放在一个数组中的

原理: 先打散数组为单个值，再注射进函数中

2. 基于原函数，创建一个一模一样的新函数，并永久绑定this:

var 新函数=原函数.bind(替换this的对象, 其他参数...)

1. 创建一个和原函数一模一样的新函数

2. 永久替换函数中的this为指定对象

何时: 替换回调函数中的this

数组API:

1. 查找元素: 查找一个指定元素在数组中的位置

var i=arr.indexOf(elem[, starti ])

用法同str.indexOf()

2. 判断: 判断数组中的元素是否符合要求:

1. 判断数组中的元素是否都符合要求:

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

//val->当前正在遍历到的元素值

//i->当前正在遍历的位置

//arr->当前正在遍历的数组对象

return 判断条件;

})

依次判断arr中每个元素，是否都符合“判断条件”的要求

执行过程: every自动拿着回调函数，在每个元素上执行一次

每次执行时，都自动传入: 当前元素值，当前位置, 当前数组对象

每次执行回调函数后，都返回对当前元素的判断结果。

只有所有元素经过判断都返回true时，every才返回true

2. 判断数组中是否包含符合要求的元素:

var bool=arr.some(function(elem,i,arr){

return 判断条件

})

3. 遍历: 依次对数组中每个元素执行相同的操作

1. 对原数组中的元素执行相同的操作：

arr.forEach(function(elem,i,arr){

操作

})

会自动拿着回调函数，在每个元素上执行一次！

2. 复制出原数组中的元素，执行相同操作后，放入新数组中，原数组保持不变。

var 新数组=arr.map(function(elem, i, arr){

return 新元素值;

})

map: 3件事:

1. 创建一个新的空数组

2. 自动在每个元素上调用一次回调函数，将返回的新值，放入新数组中相同位置

3. 返回新数组

4. 过滤和汇总:

1. 过滤: 复制出原数组中符合条件的元素，组成新数组返回

var 新数组=arr.filter(function(elem,i,arr){

return 判断条件;

})

执行过程: filter会自动拿着回调函数在每个元素上执行一次

每次执行时，如果当前元素符合“判断条件”要求，就复制到新数组中。

2. 汇总: 将数组中的所有元素，经过统计，得出一个最终结论

var result=arr.reduce(function(prev,elem,i,arr){

return 将当前元素的内容汇总到prev中

},start);

//start: 开始的基础值

//prev: 截止目前位置，之前的元素的临时汇总值

day6

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:

day7

1. 什么是DOM: Document Object Model

DOM就是专门操作网页内容的API标准——W3C

为什么:

何时: 只要用js操作网页的内容

包括: 增删改查+事件绑定

2. DOM Tree:

什么是: 在内存中存储一个网页中所有内容的树形结构

为什么: 只有树形结构才能灵活保存上下级包含关系

何时: 浏览器每加载一个网页，就会读取网页内容，并在内存中创建DOM数。

网页中每项内容，都是DOM树上的节点对象:

整个DOM树只有唯一一个根节点: document对象

每个节点对象上，都有三个属性:

.nodeType: 节点类型, 值为数字

何时: 当获得一个节点对象，需要判断节点的类型时

包括: document 9 根节点

element 1 元素节点

attribute 2 属性节点

text 3 文本节点

问题: 只能判断类型，不能进一步判断元素名

.nodeName: 节点名称

何时: 希望进一步判断节点/元素的名称时

包括: document #document

element 全大写的标签名

attribute 属性名

text #text

.nodeValue: 节点值

何时: 几乎不用

包括: document null

element null

attribute 属性值

text 文本的内容

3. 查找: 4种:

何时: 修改/删除元素前都要先找到要操作的元素

1. 不需要查找就可直接获得的重要元素:

html document.documentElement

head document.head

body document.body

form document.forms[i/id]

2. 按节点间关系查找  
 节点树: 包含所有网页内容（元素，文本，属性）的树结构

2大类关系:

1. 父子:

elem.parentNode 找elem的父节点

elem.childNodes 找elem的所有直接子节点

elem.firstChild 找elem的第一个直接子节点

elem.lastChild 找elem的最后一个直接子节点

2. 兄弟:

elem.nextSibling 找elem的下一个兄弟

elem.previousSibling 找elem的前一个兄弟

问题: 受看不见的空字符的影响

解决: 元素树: 仅包含元素节点的树结构

元素树不是一棵新树，而是节点树中的一个子集

优点: 不受看不见的空字符的干扰！

2大类关系:

1. 父子:

elem.parentElement 找elem的父元素

elem.children 找elem的所有直接子元素

elem.firstElementChild找elem的第一个直接子元素

elem.lastElementChild找elem的最后一个直接子元素

2. 兄弟:

elem.nextElementSibling 找elem的下一个兄弟元素

elem.previousElementSibling 找elem的前一个兄弟元素

递归遍历一个父元素下所有后代元素:

2步:

1. 定义函数，仅遍历直接子元素

2. 在遍历直接子元素时，对每个直接子元素调用和父元素完全相同的操作

问题: 必须先获得一个元素，才能按节点间关系查找

解决: 用HTML特征查找

3. 按HTML特征查找:

何时: 在没有获得任何元素情况下，可作为首次查找之用

包括: 4种

1. 按id查找一个元素:

var elem=document.getElementById("id");

强调:

1. 按id查找，必须用document调用

2. 只能找到一个元素

2. 按标签名查找:

var elems=任意父元素.getElementsByTagName("标签名");

强调:

1. 可用任意父元素调用:

比如: 在ul下找li; 在table下找td; 在select下找option

2. 可找到多个，所以返回多个元素组成的类数组对象

3. 按标签名查找不仅查找直接子元素，且在所有后代中查找符合条件的

3. 按name属性查找:

var elems=document.getElementsByName("name")

强调: 1. 只能用document调用

2. 返回多个元素组成的类数组对象

4. 按class查找:

var elems=任意父元素.getElementsByClassName("class")

强调: 1. 可用任意父元素调用

2. 返回多个元素组成的类数组对象

3. 不仅查找直接子元素，且在所有后代中查找。

4. 如果一个元素同时被多个class修饰，那么只要用其中一个class，就可找到该元素。

day8

1. 查找:

4. 按选择器查找: 2种情况:

1. 只查找一个符合条件的元素:

var elem=任意父元素.querySelector("选择器")

2. 查找多个符合条件的元素:

var elems=任意父元素.querySelectorAll("选择器")

\*\*\*鄙视: 优化: 按HTML查找 vs 按选择器查找

1. 返回值:

按HTML查找返回动态集合: 不实际存储属性值，每次访问集合内容，都重新查找DOM树

按选择器查找返回非动态集合: 实际存储所有属性值，每次访问集合内容，不需要重新查找DOM树

2. 首次查找效率:

按HTML查找快

按选择器查找慢

3. 易用性:

按选择器查找，简单又强大

按HTML查找，相对繁琐

总结: 1. 当仅靠一个条件就可找到元素时，首选按HTML查找

2. 当查找条件复杂时，首选按选择器查找，简单易用

2. 修改:

内容: 获取或修改元素的内容: 3种:

1. 获取或修改元素的原始HTML片段内容: elem.innerHTML

2. 获取或修改元素的纯文本内容: elem.textContent

vs innerHTML: 1. 将特殊符号翻译为正文

2. 去掉了内嵌标签

3. 获取或修改表单元素的内容: elem.value

属性: 3种:

1. HTML标准属性: HTML标准中规定的元素具有的属性

2种:

1. 核心DOM API: 最初DOM标准中指定的，可以访问所有结构化文档的API

4个:

1. 获取属性值: elem.getAttribute("属性名")

2. 修改属性值: elem.setAttribute("属性名","新属性值")

3. 判断是否包含指定属性: elem.hasAttribute("属性名")

4. 移除属性: elem.removeAttribute("属性名")

2. HTML DOM API: 对常用DOM API提供了简化版本

对于标准属性: HTML DOM已经将所有标准属性都提前内置在了元素对象中，可用.直接访问。如果没有出现在页面上的标准属性值，默认为""

elem.标准属性名

特例: html元素的class属性:

应该换成: elem.className => <elem class="">

因为class是ES中的保留字

2. 四大状态属性:

enabled disabled checked selected

值都不是字符串，而是bool类型

不能用核心DOM 4个API修改

只能用.直接访问。

补: CSS3: 状态伪类选择器:

:selected :checked :disabled :enabled

选中处于某种状态的元素

3. 自定义扩展属性:

什么是: HTML标准中没有规定的，程序员自行添加的属性

何时: 2种:

1. 在客户端临时缓存业务数据，减少向服务器端发送请求的次数

2. 代替其他选择器(id,class,元素)，为元素绑定行为

id 只能选一个元素

元素 限制太死板

class 是样式专用，不稳定

解决: 今后只要为元素添加行为，都应该用自定义扩展属性

bootstrap:

1. 为想要添加行为的元素，指定自定义扩展属性

<a data-toggle="dropdown"

2. 用属性选择器找到有自定义扩展属性的元素绑定行为:

var elems=

document.querySelectorAll("[data-toggle=dropdown]")

for(var elem of elems)

elem.on事件名=function(){ ... }

如何:

1. 定义时: <ANY 自定义属性名="值"

2. 获取或设置自定义属性的值:

不能用.修改，因为内存中的元素对象中不包含自定义扩展属性

用核心DOM API可以访问自定义扩展属性:

elem.getAttribute("属性名")

elem.setAttribute("属性名","新值")

elem.hasAttribute("属性名")

elem.removeAttribute("属性名")

HTML5:

1. 如果在定义自定义扩展属性时加了data-前缀

2. 在程序中: elem.dataset.属性名

其中: dataset可自动收集data-\*开头的属性

样式:

1. 修改一个css样式:

elem.style.css属性名="新值"

其中: style代表元素开始标签中的内联样式style属性

强调: 所有css属性名都要去-，变驼峰命名

2. 获取一个css样式:

错误: elem.style 只能获得内联样式

解决: 今后只要获取css属性值，都用计算后的样式

什么是计算后的样式: 最终应用到元素上的所有css属性的集合。

还将相对单位的属性值计算为绝对单位(px)

如何: 2步:

1. 先获得计算后的完整样式对象

var style=getComputedStyle(elem)

2. 从style中获得想要的css属性值

style.css属性名

强调: 计算后的所有样式，只读。因为计算后的样式来源不确定，不知道多少地方正在同时使用。

总结: 今后只要修改css属性，都用elem.style.css属性=值

今后只要获取css属性，都用getComputedStyle

问题: elem.style，一次只能修改一个css属性

如果批量修改多个css属性时，代码繁琐

解决: 今后只要批量应用样式，都用class

day9

1. 添加/删除:

添加: 3步:

1. 先创建一个空元素对象:

var a=document.createElement("a");

相当于: <a></a>

2. 设置关键属性:

a.innerHTML="go to tmooc"; => <a> go to tmooc </a>

a.href="http://tmooc.cn"

=> <a href="http://tmooc.cn"> go to tmooc </a>

3. 将新元素添加到DOM树上指定位置

3种:

1. 父元素.appendChild(a) 将a追加到父元素下最后一个孩子之后

2. 父元素.insertBefore(a, child) 将a插入在父元素下一个现有child元素之前

3. 父元素.replaceChild(a, child) 用a代替父元素下现有的child子元素

优化: 尽量减少操作DOM树的次数:

为什么: 每修改一次DOM树，就会导致重排重绘，降低页面加载的效率。

原理:

html->DOM Tree ←修改

↓

Render Tree -> 重排->重绘paint

↑

css -> CSS Model

解决: 2种:

1. 当同时添加父元素和子元素时，应该先在内存中将所有字元素加入父元素下，最后再一次性将父元素添加到DOM树

2. 如果父元素已经在页面上了，要同时添加多个平级子元素时，应该找一个临时的父元素，先将所有子元素加入临时的父元素中，最后，再将临时的父元素添加到DOM树

临时的父元素: 文档片段 document fragment

什么是: 内存中，临时存储多个子元素的虚拟父元素

何时: 要同时添加多个平级子元素到DOM树时

如何: 3步:

1. 创建文档片段对象:

var frag=document.createDocumentFragment();

2. 将子元素添加到文档片段中:

frag.appendChild(child)

3. 将frag整体一次性添加到DOM树

父元素.appendChild(frag)

删除: 父元素.removeChild(child)

child.parentNode.removeChild(child)

2. HTML DOM 常用对象:

Image: 唯一的简化: var img=new Image();

Select/Option

Select:

属性: .selectedIndex 自动获得当前选中的option的下标i

.options 快速获得当前select下所有option的集合

.options.length 表示select下option的个数

.length 简化.options.length

方法: .add(opt) 简化.appendChild(opt)

问题: 不支持文档片段

.remove(i) 移除i位置的一个option

Option:

最大的简化: var opt=new Option(text, value)

新建一个option元素，同时设置其内容和值

Table/...:

逐级管理:

table管着行分组:

创建(行分组): 创建行分组，同时将行分组立刻就添加到table中

var thead=table.createTHead()

//createElement(thead);table.appendChild(thead)

var tfoot=table.createTFoot()

var tbody=table.createTBody()

删除(行分组):

table.deleteTHead();

table.deleteTFoot();

获得(行分组):

table.tHead

table.tFoot

table.tBodies[i]

行分组管着行:

创建一个新行:

var tr=行分组.insertRow(i);

在行分组的第i行位置前插入一个新行，并返回新行对象。原i位置的行及其之后的行被向后挤压顺移一位

固定套路:

1. 在行分组末尾追加一个新行:

var tr=行分组.insertRow()

//createElement(tr) 行分组.appendChild(tr)

2. 在行分组第一行之前插入一个新行:

var tr=行分组.insertRow(0)

删除现有行:

不好: 行分组.deleteRow(i)

问题: 因为.前的主语是行分组，所以i要求是行在行分组内的相对下标位置。而tr有一个rowIndex属性，虽然可获得行的下标位置，但是是相对于整个表格的位置。

好的: table.deleteRow(tr.rowIndex)

获取某一个行: 行分组.rows[i]

行管着格:

添加: var td=tr.insertCell(i) //省略i，默认在末尾追加

局限: 只能添加td，不能添加th

删除: tr.deleteCell(i)

获取: tr.cells[i]

Form/表单元素

day10

1. HTML DOM常用对象:

Form:

获取: var form=document.forms[i/id]

属性: .elements 获得表单中所有表单元素(input, button, textarea, select)的集合

.elements.length 获得表单中表单元素的个数

.length => .elements.length

方法: form.submit() 代替提交按钮手动提交表单

自定义表单提交:   
 <input type="button"

.onclick=function(){

先验证所有表单元素的值是否符合要求

只有所有表单都符合要求

才手动调用form.submit()

}

表单元素对象:

获取表单元素: var elem=form.elements[i/name/id]

如果表单元素有name属性: form.name

方法: elem.focus()

elem.blur()

2. 什么是BOM: Browser Object Model

BOM是专门操作浏览器窗口或软件的API——没有标准

何时: 操作浏览器窗口，或访问浏览器软件的信息时

包含:

window: history

location

navigator

document

screen

event

3. window对象:

2个角色:

1. 代替ES标准中的global充当全局作用域对象

2. 封装所有原生API的大的对象，包括:

ES+DOM+BOM

属性:

窗口大小: 2组:

1. 完整窗口大小: [window.]outerWidth/outerHeight

2. 文档显示区大小: [window.]innerWidth/innerHeight

4. 打开和关闭窗口

[window.]open() 打开一个新窗口

close() 关闭当前窗口

5. history:

什么是: 当前窗口中，保存当前窗口打开后成功访问过的url的历史记录栈

何时: 用程序实现前进和后退时

如何: history.go(i)

前进一步: history.go(1)

后退一步: history.go(-1)

后退两步: history.go(-2)

刷新: history.go(0)

6. location:

什么是: 保存地址栏中正在打开的url的信息的对象

还提供了操作页面跳转的API

何时: 2件事:

1. 获得当前url相关的信息

2. 页面跳转  
 如何:

属性: .href 获得当前完整的url信息

.protocol 获得url中的协议部分

.host 获得url中的主机名+端口号部分

.hostname 获得url中的主机名

.port 获得url中的端口号部分

.pathname 获得url中的相对路径部分

.hash 获得url中的#锚点地址部分

.search 获得url中的?查询字符串部分

方法:

1. 在当前窗口打开，可后退:

open("url","\_self")

location.href=url => location.assign("url")

2. 在当前窗口打开，进制后退:

location.replace("新url")

用"新url"代替history数组中当前旧的url，以此来阻止后退。

3. 刷新: 2种:

1. 普通刷新: 优先从本地浏览器缓存中获取资源，除非本地缓存没有或过期，才被迫从服务器重新下载

history.go(0)

F5

location.reload()

2. 强制刷新: 强制浏览器跳过本地缓存，总是直接从服务器下载新资源。

location.reload(true)

7. navigator:

什么是: 封装浏览器配置信息的对象

何时: 只要获取浏览器的配置信息

包括:

.cookieEnabled 判断当前浏览器是否启用cookie!

什么是cookie: 在客户端本地硬盘持久保存一个数据的小文件

.plugins 封装浏览器安装的插件信息的集合

什么是插件: 为浏览器添加新功能的小程序

何时: 判断浏览器是否安装了指定的插件:

navigator.plugins["完整插件名"]!==undefined 说明安装了

.userAgent 保存浏览器名称和版本号的字符串

何时: 只要精确判断浏览器的名称和版本号时

如何:

8. 定时器: —— 回顾:

周期性定时器:

什么是: 让程序每隔一段时间间隔，反复执行一项任务

如何: 3件事:

1. 先定义一个任务函数

2. 启动定时器: 将任务函数放入定时器中反复执行:

var n=setInterval(任务函数, 间隔ms)

强调:

1. 如果定时器可能被停止，则必须在启动是获得定时器的需要n。

2. 任务函数后，不要加()，因为不是立刻执行，且不止执行一次。

3. 停止定时器:

clearInterval(n) //停止n号定时器

一次性定时器:

什么是: 让程序先等待一段时间后，再延迟执行一项任务

如何: 3件事

1. 任务函数

2. 启动定时器: 开始等待，等待结束后，自动执行一次任务。任务执行后，自动停止。

var n=setTimeout(任务函数, 等待ms)

3. 关闭定时器:

clearTimeout(n)

day11

1. 事件:

什么是事件: 浏览器内容或元素状态的变化

什么是事件处理函数: 当事件发生时，自动调用的函数

何时: 如果希望事件发生时，能自动执行一项任务

如何绑定事件处理函数: 3种:

1. 在HTML中: <ANY on事件名="处理函数()"

js: function 处理函数(){ ... }

问题: 不符合内容与行为分离的原则，不便于维护

2. 在js中绑定，赋值方式:

elem.on事件名=处理函数;

function(){ ... }

问题: 一个事件只能绑定一个处理函数

3. 在js中绑定，添加事件监听对象的方式

elem.addEventListener("事件名",处理函数)

在elem元素的监听队列中添加一个监听"事件名"的对象，其中包含一个处理函数。

当elem元素上发生"事件名"时，会遍历监听队列，找到符合条件的处理函数，自动执行处理函数。

elem.removeEventListener("事件名",原处理函数)

强调: 如果一个处理函数有可能被移除时，则绑定时就要用有名称的函数作为处理函数。

移除时才能通过函数名找到绑定时使用的原函数

事件模型: 在发生事件时，浏览器的执行过程:

3个阶段:

1. 捕获: 由外向内，记录各级父元素上绑定的处理函数

2. 目标触发: 优先触发目标元素上的处理函数:

目标元素: 事件最初实际发生在的元素

3. 冒泡: 由内向外，依次触发捕获时记录的各级父元素上的处理函数

事件对象: 事件发生时，自动创建的，保存事件信息，并提供操作事件的API 的对象

何时: 获取事件信息，或修改事件的默认特性时

如何:

获取: 事件对象总是作为处理函数的第一个参数，默认传入

elem.on事件名=function(e){ e->event }

因为当事件发生时: elem.on事件名( event )

API:

1. 取消冒泡: e.stopPropagation();

//停止 蔓延

2. 利用冒泡:

优化: 尽量减少事件监听对象的个数

为什么: 浏览器触发事件是采用遍历的方式找到的监听对象。监听对象多，遍历就慢。

如何: 如果多个子元素需要绑定相同的事件处理函数时，其实只要在父元素上绑定一次即可。所有子元素共用。

两个难题:

1. 获得目标元素:

错误: this -> 父元素

正确: e.target 代替this

2. 判断目标元素是否想要的！

3. 阻止默认行为: e.preventDefault()

1. a当按钮用时: 如果href="#xxx", 会擅自修改地址栏中的url

2. 提交按钮: 也能实现自定义提交。

自定义提交: 2种:

1. input type=button

btn.onclick=function(){

先验证

如果都验证通过

才form.submit()

}

2. input type=submit

表单提交过程:

点submit按钮->触发form.onsubmit()->正式提交

先验证

如果有一项验证未通过

就阻止提交

3. HTML5的拖拽API: 可能和浏览器的一些快捷键发生冲突。通常都要先阻止默认的行为，再编写自己的拖拽。

4. 事件发生时，鼠标坐标:

3组:

1. 相对于屏幕左上角: e.screenX e.screenY

2. 相对于文档显示区左上角: e.clientX e.clientY

3. 相对于当前事件所在元素左上角的偏移量:

e.offsetX e.offsetY

5. 滚动事件:

//当窗口中的内容发生滚动时，自动执行

window.onscroll=function(){

//获得页面滚动过的距离: 页面顶部超出文档显示区的距离。

var scrollTop=document.body.scrollTop

||document.documentElement.scrollTop

}

day12

1. 什么是jQuery:

jQuery是第三方开发的执行DOM操作的极简化的函数库

第三方: 下载才能用

执行DOM操作: jQuery还是在执行DOM操作:

学jQuery还是在学DOM

极简化: jQuery是对DOM操作的终极简化

但是没有改变DOM的四步

仅是对每个API进行了简化

函数库: jQuery都是用函数解决一切问题！

为什么:

1. jQuery是对DOM操作的终极简化：

增删改查

事件绑定

动画

ajax

2. 解决了绝大部分兼容性问题

凡是jQuery让用的，都没有兼容性问题:

何时: PC端的大项目和框架都用jQuery开发

2. 如何使用:

官网: jquery.com

下载:

版本:

1.x 支持IE8

uncompressed development 未压缩版本

具有完备的注释，代码格式和见名知意的变量名

优: 可读性好

缺: 体积大，不便于下载

适合于学习和开发之用

compressed production 压缩版

1. 删除注释和代码格式

2. 极简化变量名

优: 体积小，便于下载

缺: 可读性差

适合于生产环境

2.x 不再支持IE8

3.x 不再支持IE8

引入jquery.js:

先引入jQuery.js，再编写自定义脚本

原理:

引入<script src="jquery.js"

其实是向全局添加一种新的类型: jQuery，包含2部分

1. 构造函数: function jQuery(){}

2. 原型对象: jQuery.prototype={ 所有简化版API }

限制: 只有jQuery创建的子对象，才能使用jQuery简化版API

所有，如果想用jQuery简化版API操作DOM元素，都要先创建一个jQuery对象，封装要操作的DOM元素

如何创建: 2种:

1. 先查找，再创建

本来: var $btn1=new jQuery("选择器")

但是: 因为new jQuery使用非常频繁，所以:

jQuery构造函数进行了改造:

function jQuery(){

return new jQuery();

}

$=jQuery()

所以，创建jQuery对象，只要: var $btn1=$("选择器")

执行: 先创建jQuery对象

再用选择器找到要操作的DOM元素对象，保存进jQuery对象中

调用API时: 对jQuery对象调用API，等效于对其内部封装的DOM元素调用对等的API.

2. 直接封装已经获得的DOM元素：var $btn=$(this)

this默认返回DOM元素，不能使用jQuery家的API

$(this)是将this返回的DOM元素封装成一个jQuery对象

创建jQuery对象的结果是:

jQuery对象其实是一个封装多个DOM元素的类数组对象

jQuery API的三大特点:

1. 一个函数两用: 没给新值，默认就读取内容；给了新值，就变成修改内容

2. jQuery函数都自带遍历功能: 对jQuery对象(类数组对象)调用一次API，等效于对jQuery内封装的多个DOM元素，每个都调用一次API.

3. 每个函数都自动返回正在使用的jQuery对象本身，就可使用链式操作，简化代码。

3. 查找:

按选择器查找：

jQuery支持所有CSS3的选择器

复习: CSS3选择器:

基本选择器: #id element .class \* select1,select2,

层次选择器: 父 后代 父>子 兄+弟 兄~弟

子元素过滤选择器:

:first-child :last-child :nth-child(i) :only-child

属性选择器:

[属性名]

[属性名=值]

[属性名^=开头]

[属性名$=结尾]

[属性名\*=部分]

[属性名!=值]

并扩展了部分jQuery新增的选择器

基本过滤选择器: 先将所有符合条件的元素放入一个集合中，再统一编号，然后选择集合中指定位置的元素

:first/last :even/odd :eq/gt/lt(i)

按节点间关系查找

4. 修改:

内容

属性

样式

5. 添加/删除:

6. 事件绑定:

day13

1. 查找:

jQuery新增的选择器:

可见性过滤: 2个

:visible 选择可见的元素

:hidden 选择隐藏的元素

只能匹配: display:none和type="hidden"的两种隐藏元素

内容过滤: 2种:

1. 根据元素内容中的文本关键字作为搜索条件

:contains(关键字)

2. 根据元素下的子元素的特征作为搜索条件

:has(selector)

选择子元素中有符合selector要求的 父元素

:empty 选择内容为空的元素

:parent 选择内容不为空的元素

状态过滤: CSS3中的:

:disabled :enabled :checked :selected

表单元素过滤:

:input 选择表单中的表单元素: button input select textarea

input 是元素选择器，只能选择input元素

每一种type属性都对应着一种input

每一种type属性都有专门的一个选择器跟它对应

:text :password :checkbox :radio :submit :reset...

2. 修改: 同DOM

内容:

1. 原始HTML片段: $().html([新HTML片段]) .innerHTML

2. 纯文本内容: $().text([新文本]) .textContent

3. 表单元素的值: $().val ([新值]) .value

属性:

1. HTML标准属性: 2种访问方式:

1.核心DOM API:

$().attr("属性名"[,新值]) .getAttribute()/.setAttribute()

2. HTML DOM API:

$().prop("属性名"[,新值]) elem.属性名

2. 状态属性:

$().prop("状态"[,bool]) elem.状态

3. 自定义扩展属性:

$().attr("自定义属性") .getAttribute()/.setAttribute()

其实attr和prop可一次修改多个属性:

$().attr/prop({

属性名:值,

属性名:值,

})

样式:

$().css("css属性"[,新值]) 修改: elem.style.css属性=值

获取: var style=getComputedStyle(elem)

style.css属性

$().css({

css属性:值, //jQuery中长度数值不用加单位！

... : ... ,

})

用class批量修改元素的样式:

$().addClass("class1 class2")

$().removeClass("class1")

$().hasClass("class1")

$().toggleClass("class1") 在有或没有class1之前切换

相当于: if(!$btn.hasClass("down"))

//为当前按钮添加down class

$btn.addClass("down");

else//否则(有)

//为当前按钮移除down class

$btn.removeClass("down");

3. 按节点间关系查找:

2大类:

1. 父子:

$().parent() elem.parentNode

$().children(["selector"]) elem.children

$().find("selector") 查找所有后代元素中符合selector的元素

$().children(":first-child") elem.firstElementChild

$().children(":last-child") elem.lastElementChild

2. 兄弟:

$().prev() elem.previousElementSibling

$().prevAll() 之前所有

$().next() elem.nextElementSibling

$().nextAll() 之后所有

$().siblings() 选除当前元素自己之外的所有

4. 添加/删除/替换/克隆

添加: 2步:

1. 用$("html片段")创建新元素，及其子元素

var $elem=$("html片段");

2. 将$()创建的新元素追加到DOM树上

$(父元素).append($elem) 父元素.appendChild(elem)

$(elem).appendTo(父元素)

返回值: $(父元素).append() return 父元素

$(elem).appendTo() return 子元素elem

$(父元素).prepend($elem) 在开头插入

$(elem).prependTo(父元素)

$( child).before(elem) 父元素.insertBefore(elem, child)

$(child).after(elem) 将elem插入到child之后

$(child).replaceWith(elem)父元素.replaceChild(elem, child)

$(elem).replaceAll(child) return 新子元素elem

删除: $().remove()