**JavaScript基础**

作者：Tom(QQ:76876201)

# JavaScript历史

## 1995年,兰登艾奇(Brendan Eich)开发出LiveScript,并首次运行在Netscape Navigator 2上,用来完成简单的表单验证

## 因为市场宣传需求,LiveScript与Sun公司合作,发布前将LiveScript改名为JavaScript

## 随后在Netscape Navigator 3中发布了JavaScript1.1,同时微软在IE3中加入JavaScript的实现JScript,ScriptEase中加入JavaScript的实现CEnvi,至此浏览器上有三种不同版本的脚本语言

## 1997年,为了统一,欧洲计算机制造商协会ECMA(European Computer Manufacturers Association）的39号技术委员会(TC39 Technical Committee#39)以JavaScript1.1为蓝本,完成了ECMA-262号标准的制定,即定义了一种名为ECMAScript的新的脚本语言标准,届时ES1诞生

## 1998年,ES2发布,但是ES2没有任何改动,该版本主要是为了和ISO/IEC-16262保持一致

## 1999年,ES3发布,ES3新增了正则,异常处理,控制语句等,自此ES成了一门正真的语言

## 2000年,ES4开始准备,但是这个版本最后没有通过,但是它的大部分内容被ES6继承了

## 2007年,ES4草案发布,ES4做了一次全面的修改,包括强类型变量,类和继承,但是由于改动太大,ECMA和各大浏览器厂商发生了严重分歧,最后ECMA开会决定,中止ES4的开发

## 2009年12月3号,ES5发布,新增JSON对象,严格模式等,到了2011年,ES5成为了ISO国际标准

## 2015年6月,ES6发布

## 2016年,ES7发布

## 2017年6月,ES8发布

# 介绍

## JS的组成

### ECMAScript

### DOM(Document Object Model)

### BOM(Browser Object Model)

## JS运行环境

### JS是一门解释性语言,运行在浏览器中,由浏览器的JS引擎负责解释运行JS代码。

### 注意点

* 解释型语言：程序执行之前，不需要编译就可以直接运行, 运行时再边解析边执行的语言
* 编译型语言：程序执行之前，需要一个专门的编译过程，把程序编译成为机器语言的文件，比如exe文件

## JS学什么

### 程序共同

* 常量
* 变量
* 表达式
* 控制结构
* 数据类型
* 算法
* ...

### JS语言独有

* 内置对象
* 操作 DOM
* 操作 BOM

# 使用

## 写在元素标签的起始标签中

### 用法:把JS代码写在元素起始标签的事件属性中

### 缺点:重复代码多,维护困难,使用场景有限

### 案例1:点击改颜色/免登陆提示框,通过案例总结

* JS可以修改元素的属性
* 修改元素属性的步骤

• HTML页面布局

• 确定修改哪些元素

• 确定触发修改元素的事件,并将该事件添加在触发元素上

• 编写JS代码来修改属性

* 注意点

• 任何属性都可以修改

• 任何标签都可以有ID属性,可以通过ID属性获取该元素

• 修改属性时,HTML当中怎么写,JS当中就怎么写,但class是特例,用className

• 修改样式的格式:obj.style.属性＝值

• JS用style修改属性时,如果CSS的属性有"-"的话,通常把"-"去掉,然后把"-"后面的单词首字母大写

## 写在script标签中

### 用法:把样js代码写在在script标签里面

### 缺点: 只适用于当前页面

### 注意点

* 传统的做法是把script标签写在head标签里面
* script标签中的type属性可以不用写, 默认就是type="text/javascript"

### 案例:修改多个样式,通过案例总结

* 可以把多行代码写在一起,然后给他起个名字,这就是函数
* 函数分为定义和执行,二者缺一不可
* 函数可以多次执行
* 变量相当于值的一个别名,变量出现的地方就相当于值本身

### 案例:网页换肤,通过案例总结

* 任何标签都可以加ID
* 任何标签的任何属性都可以修改

## 写在外部的单独文件中,用script标签引入

### 用法:把JS写在一个外部文件里面,这个外部文件用.js作为扩展名,在需要使用的页面通过script标签的src属性引用即可

### 注意点

* 传统的做法是把script标签写在head标签里面
* script标签中的type属性可以不用写, 默认就是type="text/javascript"
* 每次加载外部的js文件都会发送一次请求, 这样非常消耗性能, 所以在企业开发中推荐将多个JS文件合成为一个JS文件,以提升网页的性能和加载速度
* script代码块中不能编写js代码, 即便写了也不会执行

### 案例:提取事件案例,通过案例总结

* 用JS可以给元素添加事件
* 可以把事件看做是一种特殊的属性

## 脚本加载顺序

### 渲染机制

* 浏览器渲染过程

•

• 浏览器HTML Parser将HTML文件解析成树形的数据结构(DOM tree)

• 浏览器CSS Parser将CSS文件解析成树形的数据结构(CSSOM Tree)

• OM构建

•

• OM构建过程:Bytes → characters → tokens → nodes → object model

• DOM tree和CSS Rule Tree合并生成Render Tree,Render Tree构建完成后，每个节点都是可见节点并且都含有其内容和对应规则的样式

• 遍历Render Tree开始布局，计算每个节点的位置大小信息

• 将Render Tree每个节点绘制到屏幕

• 注意点

• 解析HTML遇到外部CSS立即请求,所以需要合并CSS文件,减少HTTP请求

• 新的CSS会修改CSSOM,会重新渲染页面所以CSS文件应放在头部,缩短首次渲染时间

• 遇到<img>会发出请求,但不会阻塞,服务器返回图片文件,由于图片占用了一定面积,影响了后面段落的排布,因此浏览器需要回过头来重新渲染这部分代码,所以最好为图片都设置尺寸，避免重新渲染

• 遇到<script> 标签,会立即执行js代码,阻塞渲染,所以通常将js代码放在body的最后

• js修改DOM会重新渲染,所以页面初始化样式不要使用js控制

• CSS和HTML 被视为阻塞渲染的资源

• HTML无论如何都要加载,所有的渲染都是从HTML加载后开始的

• CSS文件加载时DOM构造继续进行,当CSS文件返回一些内容后CSSOM构造就开始

• JS文件加载时DOM构造就暂停,直到JS解析执行完毕

• CSSOM没有构建完毕,浏览器将不会渲染任何已处理的内容,而JavaScript 不仅可以读取和修改 DOM 属性,还可以读取和修改 CSSOM 属性,所以CSS 资源先于 JavaScript 资源的引入

* 重排Reflow

• 元素的几何尺寸发生了变化,需要重新验证并计算渲染树,如增加删除了DOM,移动了DOM的位置等

* 重绘Repaint

• 屏幕的一部分重画,不影响整体布局,如改变某个元素的背景色、文字颜色、边框颜色等等

### <script>

* HTML是从上至下加载,HTML页面中出现<script>标签后,就会让HTML页面暂停解析等待脚本的解析和执行。
* 如果在JS中要使用页面中的元素,需要保证页面中的元素加载后再使用

• 方法一:通常将js代码放在body的最后(推荐)

• 方法二:放在head标签中,但需要把把代码写在window.onload函数中,window.onload指的是当页面加载完成后执行的函数

### <script defer>

* 如果script标签设置了defer属性,则浏览器会异步的下载该文件并且不会影响到后续HMTL解析
* defer脚本会在整个页面都解析完毕后,DOMContentLoaded事件调用前执行
* 如果有多个设置了defer的script标签存在，则会按照顺序执行所有的script
* 该属性对内部使用方法无效

### <script async>

* 如果script标签设置了async属性,则浏览器会异步的下载该文件并且不会影响到后续HMTL解析
* async脚本会在脚本下载完毕后执行,并且不一定在DOMContentLoaded事件调用前执行
* 如果有多个设置了async的script标签存在,并不会按着script在页面中的顺序来执行,而是谁先加载完谁执行。
* 该属性对内部使用方法无效

### <script><script defer><script async>比较图

### window.onload和DOMContentLoaded事件

* window.onload需要页面上所有的资源(图片,音频,视频等)被加载以后才会触发
* DOMContentLoaded的触发不需要等待所有的资源(图片,音频,视频等)加载完成

## noscript标签

### 作用:指定在不支持脚本的浏览器中显示替代内容

### 代替内容出现情况

* 浏览器自身不支持脚本
* 浏览器支持脚本,但是被禁用了

## JS中的常见输出方式

### 在浏览器弹窗

* alert("hello world");
* prompt("请输入内容：");
* confirm("你好吗?");

### 在页面中显示内容

* document.write("hello world");

### 在控制台中显示内容

* console.log("hello world");
* console.error("错误信息");
* console.warn("警告信息");

# 语法

## 区分大小写

## 关键字和保留字

### 关键字是被赋予特殊含义的单词

### 保留字是JavaScript预留的关键字

### 保留字虽然现在没有作为关键字,但在以后的升级版本中有可能作为关键字

## 标示符(变量,函数,属性,参数的名字)的规则

### 标示符是在程序中自己起的名称

### 第一个字符必须是字母,下划线(\_)或者美元符号($)

### 其他字符可以是字母,下划线(\_)或者美元符号($)或者数字

### 命名一般采用驼峰大小写格式,即第一个字母小写,剩下的每个单词首字母大写

### 不能用关键字,保留字做为标示符

## 每条语句以分号结尾

### 注意点:语句结尾的分号不是必须的,但建议都加上

## 注释

### 单行注释,举例:// 我是单行注释

### 多行注释,举例\*\* 我是多行注释\* \*/

## 严格模式(strict mode)

### 作用:告诉JS引擎切换到严格模式下解析代码,这种模式可以消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为,这是在ES5中新增的一种运行模式

### 使用

* 针对整个文件,将"use strict"放在脚本文件的第一行,则整个脚本都将以"严格模式"运行,如果这行语句不在第一行,则无效
* 针对单个函数,将"use strict"放在函数体的第一行,则整个函数以"严格模式"运行

### 注意点

* 同样的代码,在"严格模式"中,可能会有不一样的运行结果

• 举例:以下代码在两种运行模式下的返回值是不同的 function sayHi(){ alert(!this); } sayHi();

* 一些在"正常模式"下可以运行的语句,在"严格模式"下将不能运行

# 数据

## 常量

### 固定不能改变的数据

### 整型常量111,222等各种进制整数

### 小数常量3.14、6.66等所有小数

### 布尔常量true、false

### 字符串常量"kuazhu"等使用单引号(')或双引号(")括起来的一个或几个字符

## 变量

### 可以变化的数据

### 变量定义

* var 变量名称;
* JavaScript在定义变量时不需要通过数据类型指定变量的类型
* 变量的类型会在赋值时自动确定,并且随着值的变化而变化

### 变量初始化

* 先定义后初始化
* 定义的同时初始化

### 变量默认值

* JavaScript中变量没有初始化保存的是undefined

## 数据类型

### 为了方便数据的运算和操作, JavaScript对这些数据进行了分类

### 基本数据类型

* 分类

• Undefined 未定义

• 未定义类型只有一个值undefined,这个值的意思是变量没有定义

• 未定义有两种情况,一个是真的没有定义,一个是定义了但是没有赋值

• Null 空对象

• 空对象类型只有一个值null,代表一个空对象指针

• Number 数字类型

• 用来表示数字

• 可以表示整数和小数(也叫浮点数或者双精度数)

• String 字符串类型

• 用单引号或者双引号括起来的字符序列

• 单引号和双引号表示的字符串完全一样

• Boolean 布尔类型

• 布尔型也被称为逻辑值类型或者真假值类型

• 布尔型只能够取真（true）和假（false）两种数值

• 所有类型的值都可以表示布尔值

• 真:true,非零数字,非空字符,非空对象

• 假:false,数字零,空字符,null,undefined

* 基本数据类型的特点

• 基本类型访问的变量的实际值

• 基本类型不能动态的添加属性

• 基本类型变量复制时会新建一个变量并分配空间,将变量的值分配在新的空间上

• Undefined和Null是一种数据类型,而undefined和null是值,是一个常量

### 引用数据类型

* 分类

• Object 对象类型

• 对象是一组数据和功能的集合

• null代表了一个空对象指针,所以用 typeof 检测时会返回 'object'

• 函数在ES中是对象,但函数也有一些特殊的属性,所以用 typeof 检测时会返回 'function'

• 数组在ES中是对象

* 引用数据类型特点

• 引用类型主要指保存在内存中的对象,包括对象,数组,函数

• 引用类型访问的是引用,引用是指向内存的一个指针

• 引用类型可以动态的添加属性

• 引用类型复制时不会新建空间,只是复制了一个新的引用

### 内存

* 栈内存

• 栈内存就像一个线性排列的空间，每个小单元大小基本相等。

• 栈内存存储基本数据类型

• 栈内存中存储的变量一般都是已知大小或者有范围上限

• 栈内存线性有序存储，容量小，系统分配效率高

• 栈内存变量基本上用完就回收

* 堆内存

• 堆内存空间排列和大小都不固定。

• 堆内存存储引用数据类型

• 堆内存中存储的对象类型数的大小一般都是未知的

• 堆内存首先要在堆内存新分配存储区域，之后又要把指针存储到栈内存中，效率相对就要低

• 只有当所有引用它的变量不存在时才回收

## 数据类型转换

### 转数值

* 显示

• parseInt 和 parseFloat函数

• 用法:将被转换的数据传入parseInt ()或者parseFloat()函数中

• parseInt用来转换整数,parseFloat可以转换整数和小数

• 转换规则

• 忽略字符串前面的空格

• 如果第一个字符不是数字或者负号加数字,会返回NaN(Not a Number)

• 从左到右解析直到第一个非数字为止

• Number函数

• 用法:将被转换的数据传入Number()函数中

• 转换规则

• 如果是纯数字的字符串，则直接将其转换为数字

• 如果字符串中有非数字的内容，则转换为NaN

• 如果字符串是一个空串或者是一个全是空格的字符串，则转换为0

• true 转成 1,false 转成 0

• null 转成0

• undefined转成NaN

• isNaN

• isNaN函数来判断转换后的值是不是数字,不是数字返回ture,是数字返回false

• 注意NaN和NaN并不相等,所以要使用isNaN来判断

* 隐示

• 减法(-) 乘法(\*) 除法(/) 比较(==) 运算会

• 注意点

• undefined == null 返回true,而undefined === null 返回false

• 隐式转换实际上调用了Number函数

* 案例:求和

### 转字符串

* toString()方法

• 用法:调用被转换数据类型的toString()方法

• 注意点

• null和undefined这两个值没有toString()方法，如果调用他们的方法，会报错

• 该方法不会影响到原变量，它会将转换的结果返回

• 数值类型的toString()，可以携带一个参数，输出对应进制的值

* String函数

• 用法:将被转换的数据传入String()函数中

• 注意点

• 有些值没有toString()，这个时候可以使用String()。比如：undefined和null

• 对于Number和Boolean实际上就是调用的toString()方法

• 对于null和undefined，就不会调用toString()方法.而是在内部生成一个新的字符串

* +""连接

• 用法:将被转换的数据和+""连接到一起

• 注意点

• 任何数据和 +"" 连接到一起都会转换为字符串

• 内部实现原理和String()函数一样

## 变量名 = 类型简称 + 对象描述

### 类型简称指变量的数据类型

* array 数组a
* boolean 布尔值b
* float 浮点数l
* function 函数fn
* int 整型i
* object 对象o
* regular 正则r
* string 字符串s

### 对象描述指对象名字全称或名字的一部分，要求有明确含义，命名要容易记忆容易理解

# 运算

## 算数

### 加(+)

### 减(-)

### 乘(\*)

### 除(/)

### 模(%)

* 案例:秒变时间

### ++

* i++,先使用后自增
* ++i,先自增后使用

### - -

* i--,先使用后自减
* --i,先自减后使用

## 赋值

### =

### +=

### -=

### \*=

### /=

### %=

## 关系

### <

### >

### <=

### >=

### 等于/不等于

* ==
* !=
* 比较时不比较类型

### 全等/全不等

* ===
* !==
* 比较时同时比较类型

## 逻辑

### 非(!)

* 格式: ! 条件A;
* 运算结果:真变假,假变真
* 注意点

• 可以多次连续使用逻辑非运算符

• 逻辑非运算符返回布尔值

### 与(&&)

* 格式: 条件A && 条件B;
* 运算结果:一假则假
* 运算过程

• 如果条件A不成立, 则返回条件A的数值本身

• 如果条件A成立, 不管条件B成不成立都返回条件B数值本身

### 或()

* 格式:条件A 条件B;
* 运算结果:一真则真
* 运算过程

• 如果条件A不成立, 则不管条件B成不成立都返回条件B数值本身

• 如果条件A成立, 则返回条件A的数值本身

## 优先级

### 逻辑运算中的优先级顺序为:逻辑非>逻辑与>逻辑或

### 可以用括号()来改变优先级

# 流程控制

## 顺序结构

### 按照代码书写的先后顺序来执行,这是程序的默认执行方式

## 分支结构

### 对给定的条件进行判断，再根据判断结果来决定执行代码

### if/else

* 语法:if(条件表达式1){ 语句块1}else if(条件表达式2){ 语句块2}else if(条件表达式3){ 语句块3}else{ 语句块4}
* 注意点

• 条件是一个布尔值,布尔值只有真或者假

• else可以省略

* 案例:单击显示和隐藏

### switch

* 语法:switch(变量){ case 值1: 语句1; break; case 值2: 语句2; break; .... default: 语句n; }
* 注意点

• JavaScript中case判断是否相等时是全等于(===),而不是等于(==),也就是说既会判断类型是否相等又会判断值是否相等

• 每一个case语句后面需要添加break语句来跳出当前switch语句

• default后的语句是当前面所有的条件都不满足时执行

• case后可以是常量也可以是变量

* 案例:判断星期几

### 三元运算

* 语法:条件 ? 语句1 : 语句2
* 注意点

• 条件成立执行语句1,否则执行语句2

• 条件运算符?和:是一个整体,不能分开使用

* 案例:判断正负数

## 循环结构

### 在给定条件成立时,反复执行某程序段, 直到条件不成立为止

### while循环

* 语法:初始值;while(循环退出条件){ 循环体(语句) 循环退出条件的改变}

### for循环

* 语法:for(初始值;循环退出条件;循环退出条件的改变){ 循环体(语句)}
* 案例

• 改变所有元素

• 隔行变色

• 全选全不选反选

• 选项卡

• 乘法表

• 倒三角

• 月历

### for in 循环

* 语法:for(属性 in 对象){ 循环体(语句)}
* 注意点

• 主要用来遍历对象的属性

• 输出属性名的顺序可能会因浏览器不同而不同

### break和continue

* break 中止所有循环
* continue 中止本次循环

### 获取元素

* document.getElementById('id名称') 选择第一个匹配的节点
* dom元素.getElementsByTagName('标签名称') 选择所有匹配的节点
* dom元素.querySelector(选择器) 选择第一个匹配的节点
* dom元素.querySelectorAll(选择器) 选择所有匹配的节点
* 注意点

• dom元素可以是获取的节点也可以是document

• 选择器是css中的任何选择器

# 函数

## 函数的定义

### 函数声明function 函数名(){ 函数体;}

### 函数表达式var 函数名 = function(){ 函数体;}

### 注意点

* 函数声明和函数表达式两种方法都可以定义一个函数,区别

• 函数声明定义的函数解析器会做函数声明的提升(函数声明的提升是指在代码开始执行前,解析器会把所有用函数声明定义的函数放到代码的顶部)

• 函数表达式定义的函数没有函数声明的提升,只有变量声明提升(JS函数中用var声明的所有变量(不涉及赋值)都被提前到函数的顶部,不用var声明的变量会添加到全局环境中)

* 函数是对象,函数名是指向函数的指针
* 函数名也是变量名

## 函数的参数

### 函数的参数其实就是一个占位符号,只有在调用的时候才来赋值

### 函数中经常变的部分用参数

### 案例:修改样式函数 function css(obj,attr,value){ obj.style[attr] = value; }

* 在使用属性时,用'[]'可以代替所有的'.',但是'.'不一定能代替'[]'

## 函数的不定参数

### 每一个函数都有一个参数数组,数组的名字是arguments

### 参数数组中的值由调用时传递,按照传递参数的顺序存储在数组中

### 案例:求和 function sum(){ var result = 0; for(var i = 0;i<arguments.length;i++){ result += arguments[i]; } console.log(result); }

### 案例:获取和设置样式 function css(obj,attr,value){ if(arguments.length == 2){ return getComputedStyle(obj,false)[attr]; }else{ obj.style[attr] = value; } }

* 获取行间样式 obj.style[attr]
* 获取非行间样式 getComputedStyle(obj,false)[attr]

## 匿名函数

### 没有名字的函数就是匿名函数

## 函数的返回值

### 在函数体内通过return把函数的执行结果返回

### 一个函数只能有一个返回值

### 没有返回值或者只有return的函数返回值是undefined

### 函数的返回值是在调用的时候获取的

## 递归调用

### 自己调用自己的函数

### 案例:求阶乘 function fact(n){ if(n==1){ return 1; }else{ return n \* fact(n-1); } }

### 注意

* 递归调用必须要有终止条件,否则就会爆栈
* 递归调用中的函数可以使用arguments.callee来代替

# 作用域和执行环境

## 作用域

### 变量的作用域

* 全局变量

• 定义在所有函数外面,所有的函数都可以使用的变量

* 局部变量

• 定义在函数的内部,只能在函数内部或者函数的子函数中使用

* 作用域的决定

• 函数执行时依赖于变量的作用域,这个作用域是在定义的时候决定的,而不时在调用的时候决定

• 举例1: var age = 1; function getAge(){ var age = 2; return age; } alert(age);//1 alert(getAge());//2 函数执行时age的值是1,但在函数内部定义时age是2,所以结果是2

• 举例2: var a = 1; function fn(){ console.log(a);//定义时a的值时1 } function show(){ var a = 2; fn();//执行时a的值是2 } show();//show中的fn执行时a的值是2,fn在定义时a的值是1,所以结果是1

• 在函数内部用var声明的变量会添加到当前函数的作用域中

• 如果变量不用var关键字声明会被添加到全局作用域中,不同推荐使用,在严格模式下定义变量不写var会报错

### 闭包

* 子函数可以使用父函数中的变量

### 函数作用域

* 块级作用域:在代码块(花括号中的代码)中声明的变量在代码块外不可用
* 函数作用域:在函数内部声明的变量在函数内部以及子函数中都可以使用
* JS没有块级作用域,而是函数作用域
* JS中因为是函数作用域,所以有声明提升的现象,即JS函数中用var声明的所有变量(不涉及赋值)都被提前到函数的顶部

## 执行环境

### 全局执行环境

* 在WEB浏览器中全局执行环境是window对象
* 所有全局变量和函数都是做为window对象的属性和方法创建的
* 执行环境中的代码执行完毕后该环境被销毁,保存在其中的变量和函数也随之销毁
* 当应用程序退出(例如关闭页面)时全局执行环境销毁

### 局部执行环境

* 每个函数在执行时会创建自己的执行环境,称为局部执行环境
* 局部执行环境也叫函数执行环境,当函数执行完毕后局部执行环境销毁并把控制权返回给之前的执行环境

### 作用域链

* 内部执行环境可以通过一个链条向上搜索作用域中的变量或者函数,这个链条就是作用域链
* 任何执行环境不能沿着作用域链向下搜索作用域中的变量或者函数

# 数组

## 数组是数据的有序列表

## 定义数组

### 方法1 通过Array构造函数var arr = new Array(1,2,3);

### 方法2 通过数组字面量[]var arr = [1,2,3]

### 注意点

* 方法1和方法2的作用是一样的,方法2效率上更高一些
* 数组中可以存放不同类型的数据
* 可以通过下标来访问数组中的元素
* 数组的下标是从0开始的
* 可以通过数组的length属性获取到数组的长度
* 数组不需要在定义时确定长度, 后续可以动态增长
* 可以改变数组的length属性,改变后的值是数组的实际长度,可以通过这种方法清空数组

## 基本操作

### 数组.push(元素) 在数组末尾添加元素,返回添加后数组的长度

### 数组.pop() 删除数组的末尾元素,返回删除的元素

### 数组.shift() 删除数组的第一个元素,返回删除的元素

### 数组.unshift(元素) 把元素添加到数组的头部,返回添加后数组的长度

### 数组.splice(起始下标,删除个数,插入值1,插入值2...),返回包含从原数组中删除项的新数组,如果没有删除则返回空数组

* 删除,数组.splice(起始下标,删除个数)
* 插入,数组.splice(起始下标,0,插入值1,插入值2...)
* 替换,数组.splice(起始下标,删除个数,插入值1,插入值2...)

### 数组.join(字符) 用指定的字符把数组的每一项连接成字符串并返回,原数组不变

### 数组1.concat(数组2) 把数组1和数组2合并生成一个新的数组并返回,原数组不变

### 数组.slice(起始下标,结束下标) 基于当前数组创建一个新数组(包含起始数据,不包含结束数据 )并返回,原数组不变

### 数组.indexOf(查找项,起始下标) 从数组头部开始向后查找,返回查找到全等匹配的数组的下标,如果没有找到返回-1

### 数组.lastIndexOf(查找项,起始下标) 从数组未部开始向前查找,返回查找到全等匹配的数组的下标,如果没有找到返回-1

## 排序

### 数组.reverse()

* 反转数组的顺序,修改原数组并返回原数组

### 数组.sort(比较函数)

* sort方法默认升序排序数组,该方法会修改原数组并返回原数组
* sort方法会调用数组中的每一项的toString方法将其转换为字符串后再比较
* 比较函数可以定义数组的排序规则

## 去重

### 初始化目标数组为空数组用来存放去除重复后的值

### 遍历原数组

### 用原始数组中的每一项和目标数组中每一项比较

* 如果有相等的值存在说明在原始数组中是重复值,则不添加该值
* 如果整个目标数组都没有相等的值说明当前原数组的值不是重复值,则添加

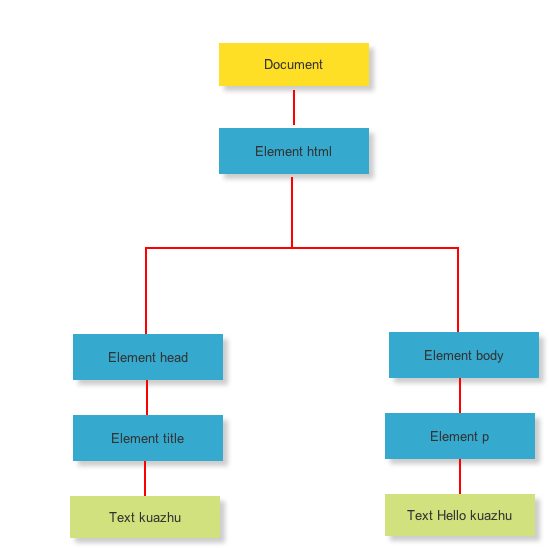
# DOM

## 什么是DOM？

### DOM(文档对象模型)是针对HTML的一个API(应用程序接口),DOM把HTML描绘成一个层次化的节点树,提供添加删除和修改页面的某一部分。

## 节点层次

### <!DOCTYPE html><html><head> <title>kuazhu</title></head><body> <p>Hello kuazhu</p></body></html>



### Document是文档节点,是整个文档的根节点

### html元素是文档元素,整个文档只能有一个文档元素

## 节点类型

### 元素节点

* nodeType属性的值为1
* 表示HTML元素

### 文本节点

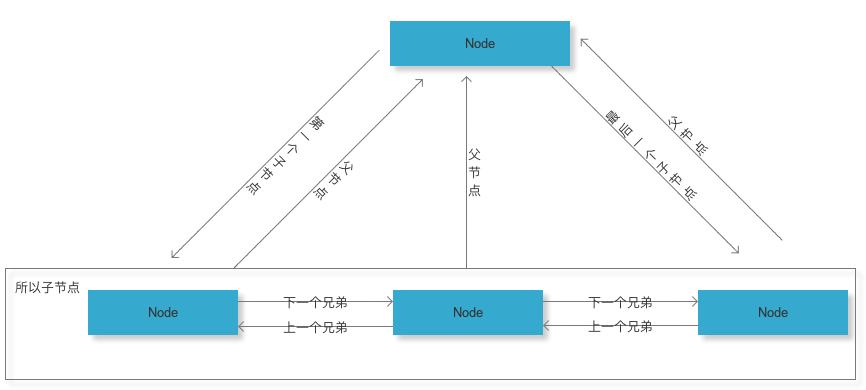
* nodeType属性的值为3
* 包含的纯文本内容

### 文档节点

* nodeType属性的值为9
* 代表了文档对象document,document是window对象的一个属性,表示整个HTML页面

## 节点关系

### 



### 获取子节点

* oParentNode.childNodes,返回一个数组,包含父节点中的所有文本节点和元素节点
* oParentNode.children,返回一个数组, 包含父节点中的所有元素节点
* oParentNode.firstChild和oParentNode.lastChild分别指向childNodes中的第一个和最后一个
* oParentNode.firstElementChild和oParentNode.lastElementChild分别指向children中的第一个和最后一个
* 案例

• 封装通过类名获取元素函数

• 无序列表鼠标移入高亮移出还原

### 获取父节点

* oNode.parentNode,子节点的父节点对象
* oNode.offsetParent,子节点绝对定位时,获取参考的父节点对象
* 案例:隐藏父元素案例

### 获取兄弟节点

* oNode.nextSibling/oNode.previousSibling,下一个/上一个文本节点或者元素节点
* oNode.nextElementSibling/oNode.previousElementSibling,下一个/上一个元素节点

## 属性操作

### 获取属性的值

* oNode.getAttribute(名称),有属性获取值,没有属性返回null
* oNode.属性名称
* oNode["属性名称"]

### 设置属性

* oNode.setAttribute(属性名称,值)
* oNode.属性名称 ＝ 值
* oNode["属性名称"] ＝ 值

### 删除属性

* oNode.removeAttribute(名称)

### 注意点

* getAttribute()和setAttribute()方法可以获取和设置自定义属性,而.和[]不可以

### 案例:输入文本并添加到文本域中

## 节点操作

### 生成元素 document.createElement('标签名')

* 该方法会返回一个DOM节点
* 创建的元素并不会插入到页面
* 需要用追加或者插入方法来将新创建的元素插入到页面

### 追加元素 parentNode.appendChild(node)

* 将指定节点追加到父节点的最后面
* appendChild会把要追加的节点从原父节点中删除
* 案例

• 获取输入框的值插入到无序列表的后面

• 按无序列表项中值对列表排序

### 插入元素 parentNode.insertBefore(node1,node2)

* 在父节点中的node2之前插入node1
* 案例:获取输入框的值插入到无序列表的最前面

### 删除元素 parentNode.removeChild(node)

* 在父元素中删除指定的子节点
* 案例

• 删除父元素

• 获取输入框的值插入到无序列表的后面并且可以删除该列表项

### 克隆节点 oNode.cloneNode([true])

* 参数true表示克隆节点的同时克隆所有后代
* 不带参数true只克隆节点
* 案例:动态九宫格

# BOM

## 什么是BOM？

### BOM(浏览器对象模型),提供用于访问浏览器的对象,这些对象和网页内容无关.

## window对象

### 什么是window对象?

* window对象代表了浏览器的一个实例
* window对象是全局对象

### 窗口位置和大小

* 窗口位置

• window.screenLeft/window.screenTop 窗口相对于屏幕左边/顶部的距离(IE Chrome Safari Opera)

• window.screenX/window.screenY 窗口相对于屏幕左边/顶部的距离(FireFox)

* 窗口大小

• 浏览器视口宽高(不包含浏览器的边框,如果有滚动条则包括)

• window.innerWidth

• window.innerHeight

• 浏览器窗口整个自身的宽高(包含浏览器边框)

• window.outerWidth

• window.outerHeight

• 边框包括工具栏,侧边栏,窗口镶边和调正窗口大小的边框。

### 打开关闭窗口

* window.open()

• 作用:打开一个新窗口并返回新打开的窗口对象

• 用法

• window.open(地址,打开方式);

• 注意点

• 地址的取值

• 完整的url

• 本地页面

• "about:blank",表示空白页面

• 打开方式取值

• "\_blank",空白页面打开,默认值

• "\_self",当前窗口打开

* window.close()

• 作用:关闭当前打开的窗口

• 注意,该方法不能关闭用户打开的窗口,只能关闭程序打开的窗口

## location对象

### document.location和window.location是同一个对象,代表当前页面的相关信息对象,可以读取和赋值

### window.location.href 属性代表当前页面的完整url

### window.location.assign(url),assign方法会打开url,如果把location.href或者location设置为一个url值,也会调用assign方法

### location的其他属性

* host 服务器名称和端口号
* hostname 服务器名称
* pathname 地址中的路径部分
* port 端口号
* search 查询字符串,指?后面的字符串

## navigator对象

### window.navigator.userAgent 用于获取当前的浏览器的信息

## history对象

### history对象代表用户上网的记录,从窗口被打开的那一刻算起

### 用法

* history.go(跳转页面数),跳转页面数是一个整数,负数表示回退,整数表示前进
* history.back(),回退一页,等价于history.go(-1)
* history.forward(),前进一页,等价于history.go(1)

# 定时器和JS运行机制

## 定时器

### setInterval(函数或者代码字符串,时间) 开启循环型定时器,返回一个唯一的整型数字代表定时器的ID

### setTimeout(函数或者代码字符串,时间) 开启延迟型定时器,返回一个唯一的整型数字代表定时器的ID

### clearInterval(定时器的ID) 关闭循环型定时器

### clearTimeout(定时器的ID) 关闭延迟型定时器

### 注意点

* 循环型定时器在定时器关闭前会一直按照设定的间隔时间执行,延迟型定时器只执行一次
* 定时器中指定的时间指的是多久后把代码加入到任务队列中,单位是毫秒
* HTML5规定定时器中的时间不得低于4毫秒，如果低于这个值(如0)，就会自动增加到4毫秒。

### 案例

* 数码时钟
* 倒计时
* 延迟菜单
* 轮播图

## JS运行机制

### 单线程和多线程

* 进程和线程

• 对于操作系统来说，一个任务就是一个进程,比如打开一个浏览器就是启动一个浏览器进程，打开一个记事本就启动了一个记事本进程

• 在一个进程内部，要同时干多件事，就需要同时运行多个“子任务”，我们把进程内的这些“子任务”称为线程（Thread）,比如Word，它可以同时进行打字、拼写检查、打印等事情。

• 线程是最小的执行单元，而进程由至少一个线程组成。

* 什么是单线程?

• 同一个时间只能处理一个指令,也就是说，同一个时间只能做一件事。

* 什么是多线程?

• 同一个时间能处理多个指令,也就是说，同一个时间能做多件事。

* 为什么JavaScript是单线程？

• JavaScript的主要用途是与用户互动，以及操作DOM,这个用途决定了它只能是单线程，否则会带来很复杂的同步问题。

• 如果JavaScript是多线程，假设一个线程在某个DOM节点上添加内容，同时另一个线程删除了这个节点，这时的处理就变得非常复杂,为了避免复杂性，从一诞生，JavaScript就是单线程。

### 同步和异步

* 什么同步任务?

• 主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务

* 什么异步任务?

• 不进入主线程,而进入"任务队列"的任务，只有"任务队列"通知主线程，某个异步任务可以执行了，该任务才会进入主线程执行。

* 为什么要有异步

• 单线程就意味着,所有任务需要排队,前一个任务结束,才会执行后一个任务。如果前一个任务耗时很长(多数是IO操作,也叫IO阻塞),后一个任务就不得不一直等着。

• 为了避免IO阻塞,主线程完全可以不管IO设备，挂起处于等待中的任务，先运行排在后面的任务。等到IO设备返回了结果，再回过头，把挂起的任务继续执行下去。这样就出现了异步

* JavaScript中的常见异步任务

• Ajax

• DOM事件

• 定时器

• Promise

### 任务队列和事件循环

* 任务队列

• 1)所有同步任务都在主线程上执行，形成一个执行栈

• 2)主线程之外，还存在一个"任务队列",只要异步任务有了运行结果,就在"任务队列"之中放置一个事件。

• 3)一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，对应的回调函数进入执行栈，开始执行。

• 4)主线程不断重复上面的第三步。这个过程是循环不断的,所以整个的这种运行机制又称为Event Loop（事件循环）

### 练习

* console.log(1); setTimeout(function () { console.log(2); }, 0); console.log(3);
* console.log('A'); setTimeout(function () { console.log('B'); }, 0); while (true) { }
* for (var i = 0; i < 3; i++) { setTimeout(function () { console.log(i); }, 1000); }

## Date对象

### Date对象是JavaScript的标准内置对象(JavaScript的标准内置对象有很多,Date只是其中的一个),他的实例是用来处理日期和时间的

### Date 实例

* 创建实例

• var today = new Date(); 依据系统设置的当前时间来创建一个Date对象。

• var date = new Date(日期时间字符串); 依据指定的日期时间字符串来创建一个Date对象。如:var date = new Date('2018/10/20 19:00:00');

* 实例方法

• getFullYear() 返回指定日期对象的年份。

• getMonth()返回指定日期对象的月份，从0开始（0表示一年中的第一月）。

• getDate()返回指定日期对象在月中的第几天

• getHours() 返回指定日期对象的小时。

• getMinutes() 返回指定日期对象的分钟数。

• getSeconds() 返回指定日期对象的秒数。

• getMilliseconds() 返回指定日期对象的毫秒数。

• getTime() 返回值一个数值，表示从1970-1-1 00:00:00 UTC (世界标准时间)距离该日期对象所代表时间的毫秒数。

• getDay() 返回指定日期对象在具体日期中一周的第几天，0 表示星期天。

• toLocaleDateString() 返回指定日期对象的年月日字符串

• toLocaleString() 返回指定日期对象的年月日和时间字符串

• toLocaleTimeString() 返回指定日期对象的时间字符串

### Date.now()方法返回自 1970-1-1 00:00:00 UTC (世界标准时间)至今所经过的毫秒数。

# 动画

## Math对象

### Math 是一个内置对象， 提供数学常数和方法

### 常见方法

* Math.abs(x) 返回x的绝对值
* Math.ceil(x) 返回x向上取整后的值
* Math.floor(x) 返回x向下取整后的值
* Math.max([x[,y[,…]]]) 返回0个到多个数值中最大值.
* Math.min([x[,y[,…]]]) 返回0个到多个数值中最小值.
* Math.random() 返回0到1之间的伪随机数.包括0但是不包括1
* Math.round(x) 返回四舍五入后的整数.

### 常见常数

* Math.PI 圆周率

## document相关值

### document.compatMode,当前浏览器的渲染模式

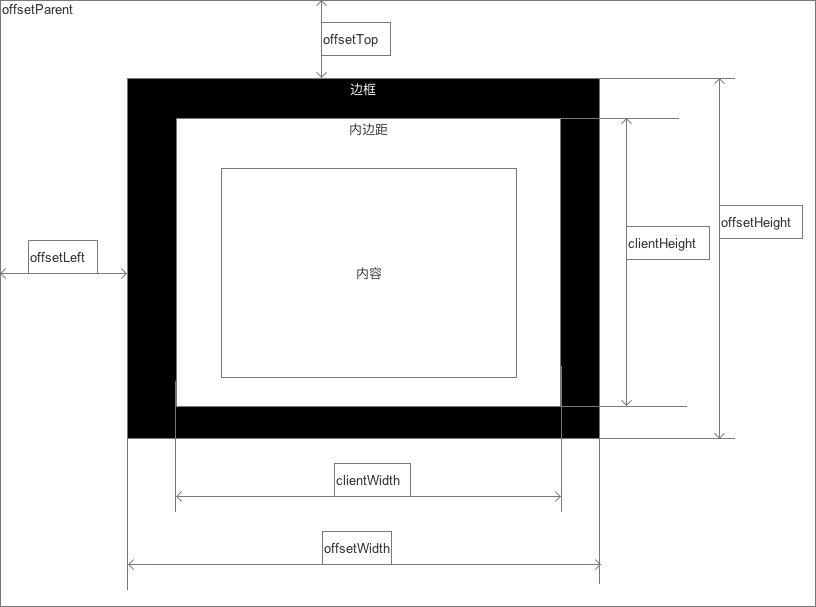
* 标准模式,写DOCTYPE的页面按照标准模式解析,document.compatMode的返回值是CSS1Compat
* 混杂模式(也叫怪异模式),不写DOCTYPE的页面按照混杂模式解析,document.compatMode的返回值是BackCompat

### document.documentElement,返回文档对象（document）的根元素的只读属性（通常指HTML文档的 <html> 元素）

### document.body,返回当前文档中的<body>元素

## offset和client相关值

### 



### offset相关值

* offsetLeft,元素的左外边框和参考父元素左内边框的距离
* offsetTop,元素的上外边框和参考父元素上内边框的距离
* offsetWidth,元素的宽(水平方向上边框+水平方向上内边距+内容宽+垂直滚动条)
* offsetHeight,元素的高(垂直方向上边框+垂直方向上内边距+内容高+水平滚动条)

### client相关值

* clientWidth,元素的客户区宽(水平方向上内边距+内容宽),不包括滚动条
* clientHeight,元素的客户区高(垂直方向上内边距+内容高),不包括滚动条
* 注意点

• client相关值通常用来确定视口的宽高,但需要注意兼容性问题

• 确定浏览器视口宽高(不包括滚动条)

• 标准模式用document.documentElement.clientWidth/clientHeight

• 混杂模式用 document.body.clientWidth/clientHeight

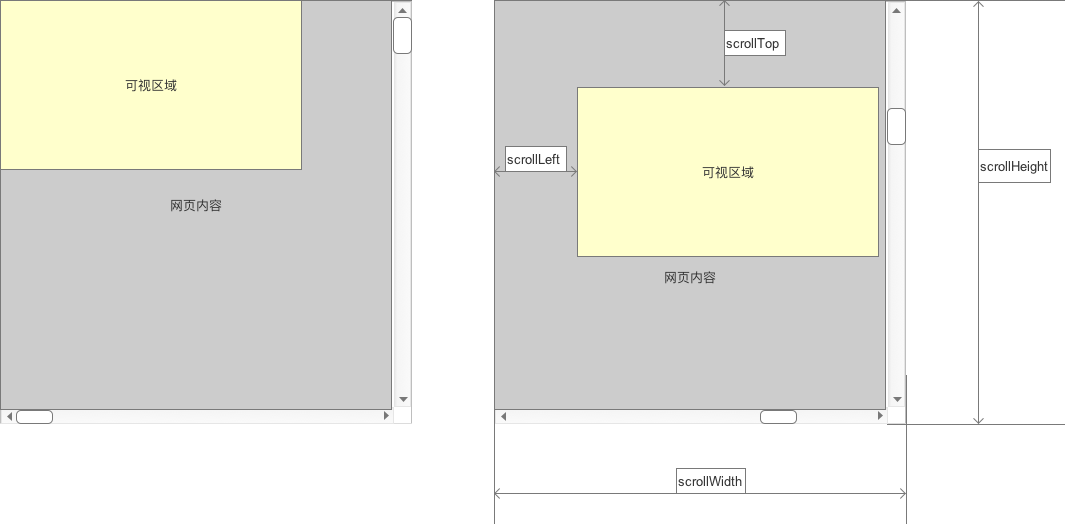
• var w =0,h=0; if(document.compatMode == 'CSS1Compat'){ w = document.documentElement.clientWidth; h = document.documentElement.clientHeight; }else{ w = document.body.clientWidth; h = document.body.clientHeight; }

### 注意点

* offset相关值和client相关值都是只读属性
* offset相关值和client相关值每次访问都会从新计算
* offset相关值和client相关值都是获取计算后的值

## scroll相关值

### 



### scrollLeft,读写属性,表示隐藏在内容区域左侧的宽度,一般指水平滚动距离

### scrollTop,读写属性,表示隐藏在内容区域上侧的高度,一般指垂直滚动距离

### scrollWidth,只读属性,表示元素内容的宽度,包括由于滚动而未显示在屏幕中内容

### scrollHeight,只读属性,表示元素内容的高度,包括由于滚动而未显示在屏幕中内容

### 注意点

* scrollLeft/scrollTop 通常用来确定页面的水平／垂直滚动距离,但需要注意兼容问题,其中scrollTop是最常用的一个属性
* 确定页面的水平／垂直滚动距离

• 标准模式用document.documentElement.scrollLeft/scrollTop

• 混杂模式用 document.body.scrollLeft/scrollTop

• IE9以上和高级浏览器可以用window.pageXOffset/window.pageYOffset

• var l = window.pageXOffset document.documentElement.scrollLeft document.body.scrollLeft; var t = window.pageYOffset document.documentElement.scrollTop document.body.scrollTop;

## window相关事件

### window.onresize 窗口大小发生变化触发

### window.onscroll 拖动滚动条触发

### window.onload 页面加载完毕后触发

## 动画

### 匀速动画

* 核心原理

• 位置移动一般需要元素绝对定位

• 在循环定时器中按一定的值(速度)改变元素的状态直到到达目标值为止(清除循环定时器)

* 注意点

• 防止同时开启多个定时器,在设置定时器前需要清除定时器

• 速度正负的决定由当前值和目标值的大小关系来决定,当前值小于目标值的话为正速度,当前值大于目标值的话为负速度

• 动画结束的条件

• 速度一般需要根据需求调整,通常不能保证运动结束时恰好到达目标值,所以不能用当前值等于目标值来做为动画结束的条件

• 动画结束的条件应当是当目标值和当前值的差不够一次速度的改变时,直接把元素改为目标值并且结束动画

• 透明值的获取需要用getComputedStyle方法,透明值的设置不需单位,因此封装函数时需要分别处理

• 多物体同时运动时需要把定时器ID保存在元素对象上避免各个物体动画之间的干扰

* 案例

• 隐藏伸缩菜单

• 底部固定

• 元素相对于body绝对定位

• 元素的top值=浏览器视口的高(window.innerHeight) - 元素自身的高度 + 垂直滚动距离

• 侧边中部固定

• 元素相对于body绝对定位

• 元素的top值=(浏览器视口的高(window.innerHeight) - 元素自身的高度)/2 + 垂直滚动距离

• 顶部固定

### 减速动画

* 核心原理

• 执行速度和目标值减去当前值的差成正比,即离目标的越远速度越快,因此执行速度可以是(目标值 - 当前值) / 系数;

* 注意点

• 由于减速动画的速度有除法,所以会产生小数,需要取整

• 由于减速动画最终的速度会变为0,所以可以用速度作为结束条件

### 综合动画

* 通过参数决定执行匀速动画还是减速动画
* 不同的动画区别在于速度和动画的介绍条件不同

### 链式动画

* 通过回调函数实现链式调用

### 多值动画

* 将属性和目标值以对象的形式传递到函数
* 循环遍历每一个属性执行动画
* 动画结束的条件是如果有在执行的动画就不结束

# 事件

## 基本概念

### 什么是事件?

* 事件就是用户或者浏览器自身执行的某种动作

### 什么是事件名?

* 浏览器预设的一些特定动作名称,如点击click,鼠标移入mouseover,加载load等

### 什么是事件处理程序?

* 响应某个事件的函数,事件处理程序的名字为"on"加上事件名,如onclick

## 事件处理程序

### DOM0级

* 添加事件处理程序: elem.事件处理程序属性 = 执行函数,即将一个函数赋给元素的事件处理程序属性,如elem.onclick = function(){}
* 注意点

• DOM0级指定事件一个事件只能指定一个执行程序

### DOM2级

* 方法

• 添加事件处理程序:elem.addEventListener(事件名,执行函数,那个阶段调用)

• 移除事件处理程序:elem.removeEventListener(事件名,执行函数,那个阶段调用)

* 注意点

• 添加/移除事件处理程序第一个参数为要处理的事件名,如'click'

• 添加/移除事件处理程序第二个参数为要事件处理程序的函数

• 添加/移除事件处理程序第三个参数为一个布尔值,false表示在冒泡阶段执行,true表示在捕获阶段执行

• DOM2事件可以为一个事件指定多个执行程序

• 移除事件监听只能是移除有函数名的，不能移除匿名函数的

### DOM3级

* DOM3事件处理程序的方法和DOM2级一样,不同的是DOM3级事件模型在DOM2级事件模型的基础上重新定义了一些事件也添加了一些新的事件(如keyup事件)

## DOM事件流

### 事件流指的是页面中接收事件的顺序

### DOM事件流模型

* 事件捕获

• document最先接收到事件,然后沿DOM树依次向下传播到具体的元素

* 事件冒泡

• 事件由具体元素接收,然后逐级向上传播

• 阻止事件冒泡 event.stopPropagation()

### 完整DOM事件流

* 事件捕获阶段

• 处于目标阶段

• 事件冒泡阶段

### 案例

* 下拉菜单
* 蒙板效果

## 事件类型

### 鼠标事件

* onclick 点击鼠标
* onmousedown 鼠标按下
* onmouseup 鼠标抬起
* onmousemove 鼠标移动
* onmousewheel 滚动滚轮
* onmouseover/onmouseenter

• 鼠标穿过事件源对象或其子元素时触发onmouseover

• 鼠标穿过事件源对象时触发onmouseenter

* onmouseout/onmouseleave

• 鼠标离开事件源对象或其子元素时触发onmouseout

• 鼠标离开事件源对象时触发onmouseleave

* oncontextmenu 右键菜单
* 案例

• 随着鼠标移动的物体

• 随着鼠标移动的多个物体

• 拖拽

• 基础拖拽

• 限制在浏览器中

• 限制在父容器中

• 磁性吸附

• 带框的拖拽

• 自定义滚动条(控制大小,控制透明度,控制文字)

• 碰撞检测

• 放大镜效果

### 键盘事件

* onkeydown 键盘按下
* onkeyup 键盘抬起
* onkeypress 键按下去触发,键不抬起会一直触发
* 案例

• 左右键移动物体

• ctrl+回车提交

• 愤怒的小鸟

### 表单事件

* onfocus 表单获取焦点触发
* onblur 表单失去焦点触发
* onchange 页面内容有改动,失去焦点执行
* oninput 输入后触发,不包括功能键
* onsubmit form表单提交事件
* onreset form表单重置事件
* 案例:简单提交验证和输入提示

## 默认行为

### 默认行为是浏览器的自带行为

### 阻止默认行为

* 方法一 return false
* 方法二 event.preventDefault()

### 案例

* 自定义右键菜单
* 只能输入数字的输入框

## 事件委托

### 事件委托是利用了事件冒泡原理,在父元素上监听事件,来处理子元素

### 事件委托的好处

* 减少绑定DOM事件
* 动态添加的DOM元素也可以有事件处理

### 案例:删除父元素

## 事件对象

### 什么是事件对象?

* 触发DOM事件时,会产生一个对象,这个对象包含了所有与事件相关的信息(属性和方法),这个对象就是事件对象event

### 共通事件属性

* event.currentTarget 绑定事件的对象
* event.target 事件源对象,即触发事件的对象
* event.type 现在的事件类型是什么

### 共通事件方法

* event.preventDefault() 阻止默认行为
* event.stopPropagation() 阻止冒泡

### 鼠标事件属性

* event.clientY/event.clientX 可视区(浏览器)的内沿到鼠标的距离
* event.pageY/event.pageX 可视区(浏览器)的内沿加滚动距离到鼠标的距离
* event.screenY/event.screenX 显示器屏幕内沿到鼠标的距离
* event.wheelDelta 鼠标滚轮距离

### 键盘属性

* event.keyCode 键盘编码
* event.ctrlKey Ctrl键
* event.shiftKey Shift键
* event.altKey Alt键

### 注意点

* 触发事件的类型不一样,可用的属性和方法也不一样

## 自定义事件

### 自定义事件步骤

* 1.创建事件对象: var 事件对象 = new Event(自定义事件类型)
* 2.监听事件 dom.addEventListener(自定义事件类型,function(){})
* 3.触发事件dom.dispatchEvent(事件对象)

### 自定义事件举例

* var dom = document.getElementById('box'); var ev = new Event('test'); dom.addEventListener('test',function(){ console.log('aa'); }) dom.dispatchEvent(ev);

## 兼容处理

### 事件对象,在IE9以下需要用window.event对象

### 添加事件,在IE9以下用attachEvent(事件处理程序名,事件处理函数)

### 移除事件,在IE9以下用detachEvent(事件处理程序名,事件处理函数)

# 面向对象基础

## 什么是对象?

### 对象就是一个可以提供特定功能的黑盒子,使用者不用关心功能的具体实现,只要知道怎么用就可以了

### 举例:电视就是一个对象,通过遥控器或者电视面板上的按钮操作电视

## 什么是面向对象的程序设计?

### 开发具有特定功能的黑盒子

## 对象分类

### JS内置对象,如:Date/Array/Math

### 用户自定义对象

## 对象的特点

### 封装: 把具体的功能实现包起来,对外不公开

### 抽象: 根据具体的业务需求把需要的属性和方法抽取出来

### 继承: 子类具有父类的方法和属性,然后子类再开发自己特有的方法和属性

### 多态: 把子类赋给父类,赋值后的父类具有子类的特性

## 对象的组成

### 属性(变量)

### 方法(函数)

## 创建自定义对象

### 字面量创建

* 单体对象,适合整个系统只有一个对象的场景,一般用于数据的传递
* 举例

• var person = { name:"Tom", age:18, friends:['Leo','Andy'], sayHi:function(){ console.log("Hi,my name is "+this.name); } }

• 注意:函数调用时属于谁,函数中的this就是谁

* 对象属性的特征

• 无序唯一

• 对象属性特征的应用,去除重复数组

* 什么是JSON?

• JSON(JavaScript Object Notation, JS 对象简谱),JSON是js对象的一个子集,主要用来做数据的传递

* JSON和JS对象的区别

• json是一种数据传输格式,主要用来数据的传递,js对象是类的实例,不能直接传输

• json的键必须添加双引号,js对象键不用加双引号,但是如果js对象的属性不是一个合法的变量名时需要加双引号

• json的值不能是函数,js对象的值可以是任意类型

* JSON和JS对象互转

• json转js对象:JSON.parse(json)

• js对象转json:JSON.stringify(js对象)

### 构造函数创建

* 基本概念

• 构造函数和类,用来创建对象的函数叫构造函数,也叫类

• 对象实例,由构造函数创建的一个具体的对象

• 静态方法和实例方法

• 添加在构造函数上的方法是静态方法,静态方法的调用方法是 构造函数.方法名()

• 添加在构造函数的原型对象上的方法是实例方法,实例方法的调用方法是 对象的实例.方法名()

* 用JS内置的构造函数创建对象的缺点

• 举例:

• var person = new Array(); person.name = 'Tom'; person.age = 18; person.sayHi = function() { console.log("Hello"); }

• 缺点

• 内置构造函数都是有特定功能的

• 用内置构造函数来创建自定义的对象,会存在把内置构造函数生成的对象自带的方法和属性覆盖掉的风险

* 用Object对象来创建对象

• 举例:

• var obj = new Object(); obj.name = 'Tom'; obj.age = 18; obj.sayHi = function(){ console.log(this.name + ' say Hello'); }

• 缺点

• Object是所有对象的父类

• 创建的对象方法和属性不能共用

* 用工厂模式的函数创建对象

• 举例:

• function createPerson(name,age){ var obj = new Object(); obj.name = name; obj.age = age; obj.showName = function(){ console.log("I am "+this.name); } obj.showAge = function(){ console.log("I am "+this.age + " years old"); } return obj; }

• 缺点

• 不能用new来创建对象

• 方法不能被实例共享,即每个对象实例都生成一个相同的方法

* 用new关键字和工厂函数创建对象

• new 的作用

• 函数会在一开始创建一个对象，并且这个对象就是this

• 在函数的最后会把这个this对象返回出去

• new +工厂函数

• 举例

• function Person(name,age){ this.name = name; this.age = age; this.showName = function(){ console.log("I am "+this.name); } this.showAge = function(){ console.log("I am "+this.age + " years old"); } }

• 注意点

• 工厂模式函数中创建对象和返回对象省略,用this代替

• 用new后,把工厂模式函数可以看做是类,类名一般首字母大写

• 缺点

• 方法不能被实例共享

• new +工厂函数+全局函数

• 举例

• function Person(name,age){ this.name = name; this.age = age; this.showName = showName; this.showAge = showAge; } function showName(){ console.log("I am "+this.name); } function showAge(){ console.log("I am "+this.age + " years old"); }

• 注意点

• 将构造函数里面的方法指向全局函数,这样所有的实例都共享这个全局的函数

• 缺点

• 全局函数有可能会在其他地方调用,这时会有风险

• 如果方法太多,就需要创建大量的全局函数

• 违背面向对象的封装特性

* 原型模式

• 原型 prototype 是存在于构造函数上的一个对象,所有的实例对象都可以访问原型对象

• 构造原型模式

• 举例

• function Person(name,age){ this.name = name; this.age = age; } Person.prototype.showName = function(){ console.log("I am "+this.name+" , I have "+this.LEGS + " legs"); } Person.prototype.showAge = function(){ console.log("I am "+this.age + " years old"); }

• 可以在原型对象prototype添加共用的属性或者方法

• 变化的属性和方法写在构造器中

• 动态原型模式

• 举例

• function Person(name,age){ this.name = name; this.age = age; if(!this.showName){ Person.prototype.showName = function(){ console.log("I am "+this.name); } } if(!this.showAge){ Person.prototype.showAge = function(){ console.log("I am "+this.age + " years old"); } } }

• 把原型对象prototype上的定义写在构造函数中

• 定义之前需要唯一执行判断

## 案例

### 选项卡

### 轮播图

# 正则表达式和字符处理

## 什么是正则？

### 匹配字符串的规则

### 体验案例:从字符串中找出数字,原始方法和正则的比较,正则为:str.match(/\d+/g)

### 正则测试网站:https://regexper.com/

## 创建正则对象

### 方法一(构造函数): new RegExp(规则,修正模式) 如:var reg = new RegExp('a','g')

### 方法二(字面量): /规则/修正模式 如:var reg = /a/g

## 正则中的概念

### 原子(正则中的最小匹配单位)

* 可见原子

• 键盘输出后可以看得见的字符,如符号,英文,数字等

• 注意点

• 以下字符具有特殊含义,如果要使用原本的字符意思,需要在前面添加\来转义. \* + ? $ ^ |:! \ /() {} []

* 不可见原子

• 键盘输出后看不见的字符,如换行(\n),回车(\r),制表符(\t),垂直制表符(\v) 换页符(\f)

### 原子筛选

* | 竖线两边任意匹配
* [] 括号中任意一个原子,可以写区间(闭区间),如[a-z0-9]
* [^] 括号中任意一个原子之外的原子
* 注意点

• 一般情况下,正则中的一个字符对应字符串中的一个字符, 例如:/ab\t/表示字符串中连续出现字符 ab制表符

* 案例

• 过滤多个关键字:str.replace(/淘宝|京东/g,'\*')

• 匹配汉字并且获取长度:str.replace(/[\u4e00-\u9fa5]/g,'bb').length

• 过滤HTML标签:str.replace(/<[^<>]+>/g,'\*')

### 原子集合

* . 除了回车和换行外的任意字符 等价[^\r\n]
* \d 任意数字,等价[0-9], \D 任意非数字 等价[^0-9]
* \w 任意数字,字母,下划线 等价[0-9a-zA-Z\_],\W 任意非数字,字母,下划线 等价[^0-9a-zA-Z\_]
* \s 空白符 等价于[\t\n\f\r],\S 非空白符 等价于[^\t\n\f\r]

### 量词(限定前面原子出现的次数)

* {n} 前面原子恰好出现n次
* {n,} 前面原子最少出现n次
* {n,m} 前面原子最少出现n次,最多出现m次
* + 前面原子最少出现1次 等价{1,}
* ? 前面原子最少出现0次,最多出现一次 等价{0,1}
* \* 前面原子最少出现0次 {0,}
* 案例

• 合法年份(公元1000年至公元2999年):/^[1-2]\d{3}$/.test(str)

• 验证手机号:/^1[35789]\d{9}$/.test(str)

• 密码(以字母开头,长度在6~12之间,只能包含字母、数字和下划线)：/^[a-zA-Z]\w{5,11}$/.test(str)

### 匹配模式

* 贪婪模式

• 尽可能多的匹配(默认)

• 如'12345'.replace(/\d{2,4}/,'\*')的结果是"\*5"

* 非贪婪模式

• 尽可能少的匹配(在量词后添加?)

• 如'12345'.replace(/\d{2,4}?/,'\*');的结果是"\*345"

### 边界控制

* ^ 匹配字符串开始的位置,限制必须以其后面的原子开头,前面不能有任何东西
* $ 匹配字符串结尾的位置,限制必须以其前面的原子结束,后面不能有任何东西
* \b 单词边界
* \B 非单词边界
* () 把括号里面的正常表达匹配的结果当作一个原子来看,起到分组的效果,分组后可以通过反向引用来取得分组后的值,反向引用从$1开始
* 案例

• 邮箱校验:/^\w+@\w+\.[a-z]{2,5}$/.test(str)

• 替换单词:str.replace(/\bis\b/gi,'\*')

• url解析: str.match(/(\w+):\/\/(\w+)\.([0-9a-z-]+)\.(\w+)/)

• 将年月日替换成月日年:'2018-05-09'.replace(/(\d{4})-(\d{2})-(\d{2})/,'$2/$3/$1')

### 修正模式

* i(ignoreCase) 忽略大小写,默认区分大小写,可以通过正则对象上的ignoreCase属性查看
* g(global) 匹配所有的,默认只找到第一个,可以通过正则对象上的global属性查看

### 前瞻

* 什么是前瞻?

• 正则从文本头部向文本尾部解析,文本尾部成为"前",前瞻就是向前(文本尾部)检查是否符合断言

* 正向前瞻 exp(?=assert)

• 匹配到exp的内容后再看前面的内容符合不符合assert,符合的话就匹配exp的内容

• 举例:字母后面是数字的字母,'a1b2c\*d&'.replace(/[a-z](?=\d)/g,'$'),结果为:"$1$2c\*d&"

* 负向前瞻 exp(?!assert)

• 匹配到exp的内容后再看前面的内容符合不符合assert,不符合的话就匹配exp的内容

• 举例:字母后面不是数字的字母,'a1b2c\*d&'.replace(/[a-z](?!\d)/g,'$'),结果为:"a1b2$\*$&"

## 正则对象的属性和方法

### 正则对象的属性

* lastIndex 下一个开始匹配的索引值
* source 正则的文本字符串

### 正则对象的方法

* reg.test(str)

• test方法测试字符串是否有匹配到正则的字符串,有返回true,没有返回false

• 全局匹配

• 每一次匹配后正则对象上的lastIndex属性会更改,改为匹配内容的最后一个字符的下一个位置,所以结果会不可靠,一般使用test方法时建议不要全局匹配

• 举例: var reg = /[a-z]/g; console.log(reg.lastIndex);//0 console.log(reg.test('a1b2'));//true 从下标0开始匹配到a console.log(reg.lastIndex);//a的下一个位置是1 console.log(reg.test('a1b2'));//true 从下标1开始匹配到b console.log(reg.lastIndex);//b的下一个位置是3 console.log(reg.test('a1b2'));//false 从下标3开始匹配,没有字符 console.log(reg.lastIndex);//匹配结束,lastIndex重置为0

• 非全局匹配

• lastIndex始终是0,也就是说始终从第0个元素开始查找

* reg.exec(str)

• exec方法对字符串执行搜索,并更新正则对象上的lastIndex属性,如果没有匹配的返回null,有的话返回一个数组

• 数组的第0个元素是匹配的内容,如果有分组,从下标是1开始是对应分组的内容

• index 是匹配内容第一个字符的位置

• input 是被搜索的字符串

• groups 存放着每个命名分组的名称以及它们匹配到的值,如果没有命名分组则为undefined,命名捕获分组自身的语法是 (?<name>...)，比普通的分组多了一个 ?<name>

• 全局匹配

• 每一次匹配后正则对象上的lastIndex属性会更改,改为匹配内容的最后一个字符的下一个位置

• 举例: var reg = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/g; var str = 'a2018-05-10b'; console.log(reg.lastIndex);//0 console.log(reg.exec(str));//["2018-05-10", "2018", "05", "10", index: 1, input: "a2018-05-10b", groups: {year: "2018", month: "05", day: "10"}] console.log(reg.lastIndex);//11 console.log(reg.exec(str));//null console.log(reg.lastIndex);//0 console.log(reg.exec(str));//["2018-05-10", "2018", "05", "10", index: 1, input: "a2018-05-10b", groups: {year: "2018", month: "05", day: "10"}]

• 非全局匹配

• lastIndex始终是0,也就是说始终从第0个元素开始查找

## 字符串处理方法

### str.search(reg)

* 从字符中检索与正则相匹配的字符串,返回第一个匹配结果的index,查不到返回-1
* search方法不执行全局匹配,它会忽略g,并且总是从字符串开始检索
* 举例: var str = 'good baby,have a good day'; console.log(str.search(/good/));//0 console.log(str.search(/good/g));//0 console.log(str.search(/good/g));//0

### str.math(reg)

* 全局匹配

• 找到字符串中的所有匹配的字符串,如果没有匹配的话返回null,有的话返回一个数组

• 数组中的每一项就是一个匹配结果,数组中没有index,input和groups等属性

• 全局匹配不需要分组

• 举例: var reg = /\d{4}-\d{2}-\d{2}/g; var str = 'a2018-05-10b2019-06-10c'; console.log(str.match(reg));//["2018-05-10", "2019-06-10"]

* 非全局匹配

• 和reg.exec(str)非全局匹配的结果一样

• 非全局匹配可以分组

• 举例: var reg = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/; var str = 'a2018-05-10b2019-06-10c'; console.log(str.match(reg));//["2018-05-10", "2018", "05", "10", index: 1, input: "a2018-05-10b2019-06-10c", groups:{year: "2018", month: "05", day: "10"}

### str.split(reg)

* 根据正则匹配的内容把字符串分隔成数组
* 举例:以空白符分隔字符串为数组,'ab c d ef fgh'.split(/\s+/)的结果是["ab", "c", "d", "ef", "fgh"]
* 举例:判断是否回文字符串:str === str.split('').reverse().join('')

### str.replace(reg,str2)

* 把字符串str中匹配到正则的子串用str2进行替换

### str.replace(reg,function(匹配结果,index,原值))

* 把字符串中匹配到正则的子串传递到回调函数进一步处理,用函数的返回值来替换匹配项
* 举例: var str = 'a1b2c3d4'.replace(/\d/g,function(match,index,origin){ return parseInt(match)\*10; }) console.log(str);//a10b20c30d40

### str.charAt(index)

* 根据位置返回字符
* 位置在0和str.length-1之间,如果不在返回空字符
* 一个汉字是一个字节

### str.charCodeAt(index)

* 根据位置返回字符的编码
* 位置在0和str.length-1之间,如果不在返回空字符
* 一个汉字是一个字节
* charCodeAt返回的是字符的Unicode码
* Unicode码是统一的编码,Unicode码包含了ASCII码(0-127)

### str.indexOf(str1)

* 根据字符返回位置,从前向后找,找不到返回-1

### str.lastIndexOf(str1)

* 根据字符返回位置,从后向前找,找不到返回-1

### encodeURIComponent(str)

* URI 编码
* 编码除了字母、数字、(、)、.、!、~、\*、'、-和\_之外的所有字符

### decodeURIComponent(str)

* URI 解码

### str1.concat(str2)

* 连接字符串

### str.slice(起始下标,结束下标)

* 截取字符串,下标可以是负数,负数表示从尾部开始计算,包括起始下标但不包括结束下标

### str.substr(起始下标,长度)

* 截取字符串,下标可以是负数,负数表示从尾部开始计算

### str.substring(起始下标,结束下标)

* 截取字符串,下标不可以是负数

### str.toLowerCase()

* 把字符串转换为小写

### str.toUpperCase()

* 把字符串转换为大写

### str.trim()

* 去除字符串两边的空白

### 案例:

# 表单脚本