**第一天**

**HTML**

html:结构 页面骨架 约束结构布局

骨架:固定 html>head&body

html:根标签 界定当前页面的范围

<head>标签内写的是 设置性标签

<meta charset="utf-8"> 字符集设置

<body>标签内写的是 显示性标签

Meta meta:功能标签 字符集设置 页面初始化 搜索引擎优化 网页重定向

页面初始化

<meta name="viewport" content="width=device-width,initial-scale=1,maximum-scale=1,minimum-scale=1,user-scalable=no">

搜索引擎优化

description:网页描述

<meta name="description" content="描述内容">

keywords:关键字 英文逗号隔开

<meta name="keywords" content="key1,k2,k3,k4,k5">

author:作者

<meta name="author" content="ckk">

<!-- 网页重定向 -->

<meta http-equiv="refresh" content="3;http://www.baidu.com">

a标签

target:指定超链接打开的位置 \_self \_blank

<a href="http://www.baidu.com" target="\_blank"><img src="resouce/meng.jpg" alt="图片加载失败" title="鼠标悬停"></a>

a标签的锚点 <a href="需要跳转的哪个元素的ID">第一章</a>

a调用JS函数 <a href="javascript:函数名();">调用JS</a>

图片地图（图片局部链接）



Transition属性 复合属性

过渡:属性从一个值到另一个值之间的变化添加了一个缓动效果

transition: 0.35s;

过渡持续时长（必须有）

transition-duration: 0.75s;

过渡的属性

需要过渡的属性（通常是all，所有）

transition-property: width;

transition-property: all;

过渡延迟时间

transition-delay: 1s;

缓动效果

transition-timing-function: linear;

animation 关键帧动画

动画名称

animation-name: donghua;

动画执行时间

animation-duration: 0.75s;

动画执行次数

animation-iteration-count: 2;

动画执行结束后的填充模式 forwards停留在结束时的状态

animation-fill-mode: forwards;

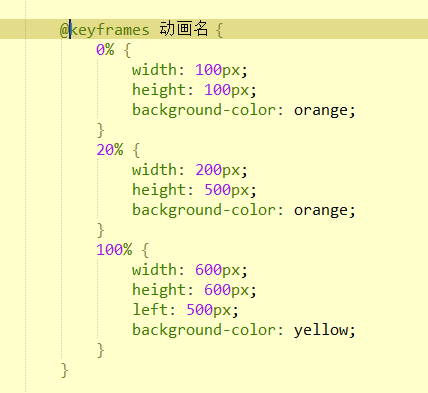
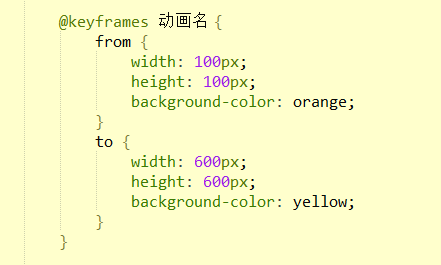
animation-fill-mode: backwards;

缓动效果

animation-timing-function:

动画延迟时间

animation-delay:



Css兼容

IE firefox chrome Safari opera

内核: trident gecko webkit webkit presto

兼容: ::-ms- ::-moz- ::-webkit- ::-webkit- ::-o-

**第二天**

盒模型

标准盒模型: css设置的width和height是content的尺寸 padding border往外加

box-sizing: content-box;

box-sizing: padding-box;

IE老版本中的盒模型就是怪异盒模型: css设置的width和height是content+padding+border的尺寸

怪异模式的应用: 应用于移动端页面

box-sizing: border-box;

移动端布局

移动端页面的初始化

<meta name="viewport" content="width=device-width,initial-scale=1,maximum-scale=1,minimum-scale=1,user-scalable=no">

适配:页面布局适配各种屏幕尺寸 JS适配各种浏览器的版本

rem:相对单位 root 1rem相当于是根元素的一倍字体大小

em:相对单位 1em相当于是当前元素的一倍字体大小

形变

形变的原理:本质是修改元素自身的参考坐标系

位移

transform: translate(100px,100px);

旋转

transform: rotate(45deg);

斜切，倾斜

transform: skew();

缩放

transform: scale();

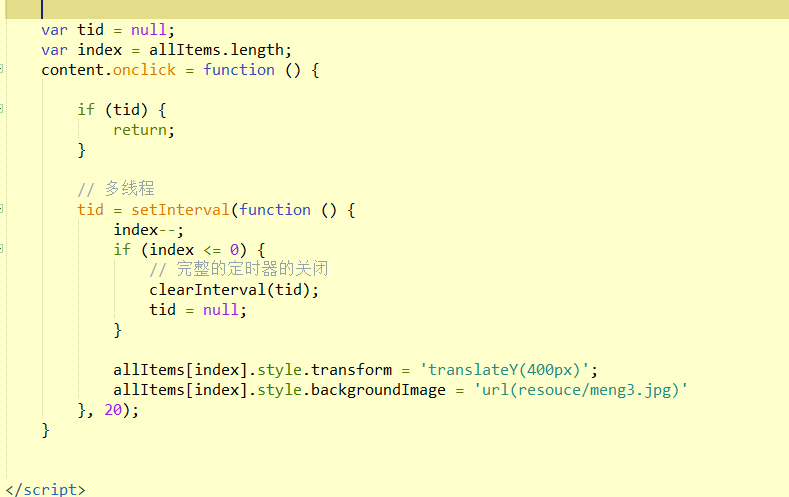
**第三天**

定时器

seTtimeout 单次定时器

setInterval 多次定时器

清除定时器：事件触发后先判断定时器是否存在，存在就return,定时器执行完后再设置为空



Img

不管是img标签还是background-image 图片自带3px的底部边距

解决：vertical-align: middle;

Dispaly:inline-block

父级：font-size:0;

浮动

浮动:设置了浮动的元素 会脱离标准流(脱标) 浮在标准流的上方 在标准流中不占位置 不受display限制 显示类似于inline-block

浮动的影响: 脱标的元素不占位置 父元素的高度撑不开 影响浮动元素下方的布局

清除浮动:清除浮动造成的影响

清浮动的方法

1. 给父元素添加高度
2. 给父元素设置溢出隐藏 overflow: hidden;
3. 在浮动元素下方添加一个空盒子 设置clear:both 隔墙法 影响dom结构
4. 伪元素清除法 给浮动元素的父元素添加一个类名clearfix

（1）.clearfix:before,.clearfix:after {

content: '';

clear: both;

width: 0;

display: block;

visibility: hidden;

}

.clearfix {

zoom: 1;

}

（2）.clearfix:before,.clearfix:after {

content: '';

display: table;

}

.clearfix:after {

clear: both;

}

.clearfix {

zoom: 1;

}

定位 position

static:静态定位 默认定位

absolute:绝对定位 脱标(浮动) 以父元素中第一个非静态定位的元素作为参照 如果父元素都是静态定位 则以浏览器作为参照

relative:相对定位 不脱标 以自身的位置作为参照

fixed:固定定位 脱标 始终以浏览器作为参照

子绝父相:父元素设置相对定位 子元素设置绝对定位 保证子元素始终以父元素作为参照

数据类型:

基本数据类型:number string boolean undefiend

引用数据类型:object null Array Date Function RegExp Math...

**第四天**

伪元素 before after

必须有content属性 before after伪元素才生效 content: '';

类型转换:

基本数据类型:number string boolean undefined

引用数据类型:object null

undefined: 未初始化 有变量 没有初始值

is not defiend:未定义 变量都没有

null:空对象

隐式(自动)转换

任何类型的数据和字符串做+运算都是字符串的拼接

做除了加法以外的运算都要将其他类型转换为数字

console.log(10+'1');//101

console.log(10-'1');//9

console.log(true+'1');//'true1'

console.log(true-'true');//NaN

console.log(true-'0');//1

console.log(10 \* '20');//200

console.log(10 \* '-20');//-200

console.log(10 \* '-true');// NaN

console.log(10 \* '2+3');//NaN

console.log(10 + 5);// 15

其他类型-->布尔值: 非空(零)即为真

if ('') {

console.log(1);

} //false

if ('true') {

console.log(1);

} //true

if ('true'-true) {

console.log(1);

} //false

if (false-1) {

console.log(1);

} //true

if ({}) {

console.log(1);

} //true

if (undefined == null) {

console.log(1);

} //true

if (undefined === null) {

console.log(1);

} //false

if ({}==[]) {

console.log(1);

} //false

if ('length' in []) {

console.log(1);

} //true

if (' ') {

console.log(1);

} //true

强制转换

其他类型转字符串

String

toString(进制)

var a = 10;

console.log(a.toString(16));

其他类型转数字

Number()

parseInt

parseFloat

console.log(Number('10'));

console.log(Number('10a'));

console.log(parseInt('10a'));

console.log(parseFloat('10.1a'));//10.1

console.log(parseFloat('10.a1a'));//10

其他类型转布尔值

Boolean();

**第五天**

浅拷贝: 复制内存地址

var o1 = {num:10};

var o2 = o1;

深拷贝: 复制出来一个一摸一样的新的对象

var o3 = {num:10};

var newO = {};

newO.num = o1.num;

for (var key in o1) {

newO[key] = o1[key];

}

console.log(newO)

function deepCopy(obj) {

var newO = {};

for (var key in obj) {

newO[key] = typeof obj[key] == 'object'?deepCopy(obj[key]):obj[key];

}

return newO;

}

字符串: String '' "" ``

如果需要字符串的嵌套: '""' `""` 不能使用"''"

'{name:"zhang3"}';

`{name:"zhang3"}`;

es6新增的语法

var s1 = `aaa${表达式}`; //模板字符串

console.log(s1.length);

var s1 = `aaa${{}?`dsd${111}`:2}`;

字符串的常用属性和方法:

length: 字符串的长度

在JS中 有长度 有下标 就是一个类数组的结构

类数组: 只有数组的结构 不能使用数组的方法

var s2 = 'hello';

console.log(s2.charAt(2));

console.log(s2[3]);

var s1 = 'hello world';

split(字符);//根据给定的字符将字符串分割成数组

console.log(s1.split('o'));

indexOf(查找的字符, [起始的位置]);//查找给定的字符的下标 如果没有返回-1

lastIndexOf(查找的字符, [起始的位置]);//根据给定的字符查找对应的下标 如果没有返回-1 第二个参数可以指定开始查找的位置

console.log(s1.indexOf('lo', 5));

toUpperCase();//将字符串转为大写

toLowerCase();//将字符串转为小写 、=-【ololkkkjghvb]

slice(起始的下标, 结束的下标);//截取字符串

substring(起始的下标, 结束的下标);

substr(起始的下标, 截取的长度);// 截取字符串 返回截取出来的字符串 不影响原字符串

trim();//去除字符串的首尾的空格

var s2 = ' hello wolrd ';

console.log(s2.trim());

search();//查找

match();//字符串的匹配 返回值是一个信息数组

console.log(s2.search('s'));

console.log(s2.match('h').index);

charAt(下标);//根据下标查找对应的字符

concat();//字符串的拼接

repeat(次数);//字符串重复

console.log('明天我要过节了'.repeat(3));

replace(被替换的内容, 新的内容);//字符串的内容替换

var s1 = 'sty是个zz';

var rel = s1.replace('zz', 'cc');

console.log(rel);

console.log(s1);

数组: JS中的数据容器 用来批量处理数据

Array : 在内存上以一种链式结构存储 长度 下标

数组中的数据类型没有要求

var arr = [1, '2', true, {}, [1,2,3], function fn () {

console.log(111);

}];

console.log(arr);

数组获取元素的方式: 通过下标

for-of

es6新增的遍历数组的方法

for (var item of arr) {

console.log(item);

}

数组常用属性和方法:

属性: length

方法:

var arr = [1,2,3,4,5];

shift();//从数组的头部删除 返回删除的元素

var rel = arr.shift();

console.log(arr);

console.log(rel);

unshift();//从数组的头部添加 可以一次添加多个 返回数组的新长度

var rel = arr.unshift('a', 'b');

console.log(arr);

console.log(rel);

push();//数组末尾添加 可以一次添加多个 返回数组的新长度

var rel = arr.push('a', 'b');

console.log(arr);

console.log(rel);

pop();//删除数组尾部的元素 返回删除的元素

var rel = arr.pop();

console.log(arr);

console.log(rel);

sort();//排序

var arr = [232, 44, 12, 654, 3254, 1111];

如果直接使用 按照每一个元素的第一个字符对应的ascii值进行排序

arr.sort();

console.log(arr);

可以传入函数 通过函数将元素按照从大到小/从小到大的顺序排列

arr.sort(function (a,b) {

return a - b;

});

arr.sort(function (张俊x是狗,梁一x不是人) {

return 梁一x不是人 - 张俊x是狗;

});

console.log(arr);

reverse();//倒序

slice(起始的下标, 结束的下标);//截取 不影响原数组 返回截取出来的新数组

var arr = [1,2,3,4,5,6];

var rel = arr.slice(2,3);

console.log(arr);

console.log(rel);

如果查找不到返回-1

indexOf(元素, [起始位置]);//根据元素查找下标

lastIndexOf(元素, [起始位置]);//从后往前查找

join(char);//将数组中的元素根据给定的字符拼接成字符串

var arr = ['a', 'b', 'c'];

console.log(arr.toString());

console.log(arr.join('&'));

concat();//数组的拼接

var rel = arr.concat([1,2,3]);

console.log(arr);

console.log(rel);

splice(起始的位置, [删除的长度, 替换的内容]);//删除 替换 添加

var arr = [1,2,3,4,5];

只给一个参数: 删除/截取

var rel = arr.splice(2);

console.log(arr);

console.log(rel);

只给两个个参数: 删除/截取 从给定的下标开始删除 删掉指定的长度

var rel = arr.splice(2, 2);

console.log(arr);

console.log(rel);

给定三个个参数: 替换 如果是0表示插入

var rel = arr.splice(2, 2, 2);

console.log(arr);

console.log(rel);

数组的遍历

map();

findIndex();

find();

forEach(function(item, index, arr){});

数组的过滤

filter();

every();

some();

var arr = [{name:'zhang3', age:18},{name:'zhang5', age:20},{name:'zhao3', age:21},{name:'wang3', age:18},{name:'zhaoxx', age:22},{name:'zhangjunx', age:39},{name:'gouzhang', age:39}];

var rel = arr.filter(function(item, index) {

return item.name.indexOf('zhang') == 0&& item.age>20;

});

console.log(rel);

reduce();//数组的累加

var arr = [1,2,3,4,5];

var rel = arr.reduce(function (a, b) {

console.log(a, b);

return a + b;

});

console.log(rel);//15

**第六天**

c3新增一个盒子的背景图可以设置多张 先写的在上

background-image: url(images/ground.png),url(images/bg.png);

background-size: 100% 50px, 100% 100%;

json: 对象 Object 存储形式是以键值对(key-value)的形式存储数据

json数据在任何语言都是互通

1. 访问json中数据 用点语法访问 json.属性名 点语法访问属性只能点具体的属性名

console.log(data.img);

1. 通过[] 也可以访问json中的数据json['属性名'] json[变量]

**i**

in:关键字 布尔值 判断对象是否包含某个属性

console.log('img' in data);

console.log('poster' in data);

for-in

for(var key in data){

// key:取到json中每一个属性名

console.log(key);

console.log(data[key]);

// console.log(data.key);

}

JSON

两个方法:

JSON.parse(): 将json字符串 --> json对象

JSON.stringify(): 将json对象 --> json字符串

JSON.parse只能解析标准json字符串 标准json:属性名，属性值都要加引号

如：var jsonStr = '{"name":"zhang3", "age":18}';

不标准的解析方法 如：var jsonStr1 = '{name:"li4", age:20}'; 用 eval();

eval: 可以解析json字符串

使用eval解析json字符串 需要使用()将jsonStr包起来

使用eval解析json字符串相对不安全 存在注入攻击的风险

console.log(eval('('+jsonStr1+')'));

console.log(eval(`(${jsonStr1})`));

eval也可以: 计算字符串

console.log("1+2");

console.log(eval("1+2"));

正则表达式:字符串模式

ES: RegExp() 正则的构造函数 对象

正则表达式创建

1.构造函数创建

var r1 = new RegExp(规则字符串, 权限修饰符);

i:忽略大小写

g:全局匹配

m:多行匹配

var r1 = new RegExp('box', 'igm');

2.字面量创建

var r2 = /box/im;

new Array();//[]

new Object();//{}

正则表达式提供了两个方法:

test():返回值是一个布尔值 如果字符串符合正则 就返回true 不符合就返回false

exec():信息数组 index 匹配到的下标

var s1 = 'this is a boX, that is another BOX';

var rel = r2.test(s1);

console.log(rel)

var rel2 = r2.exec(s1);

console.log(rel2)

字符串提供了4个方法可以配合正则使用

split(字符);//根据给定的字符将字符串分割成数组

replace(被替换的内容, 新的内容);//字符串的内容替换

search();//查找

match();//字符串的匹配 返回值是一个信息数组

var s1 = '138.32&44,25?2-4';

var r = /\D/igm;

var rel = s1.split(r);

console.log(rel.join(''));

var rel = s1.replace(r, '');

console.log(rel);

var r = /^[1-9][0-9]{5}([18|19])|(20)[0-9]{2} {0|1}[1-9][0|1|2|3][0-9]7332/igm;

DOM对象:通过DOM操作获取的节点对象 就是DOM对象

JQ对象:$(选择器)获取到的DOM对象通过merge方法转化的类数组 就是JQ对象

console.log(content);

console.log($('#content'));

JQ对象 --> DOM对象: 从类数组中通过标取出

$('#content')[0].innerHTML = '';

DOM对象 --> JQ对象: 通过$(dom对象)

$(content).css({});

条件结构

if (条件表达式) {

条件语句1;

}else {

条件语句2;

}

if (条件表达式) {

条件语句1;

}else if(条件表达式2){

条件语句2;

}else {

}

弹出框:暂停页面的运行

alert('alert');

confirm: 带有确定键和取消键 通过返回值可以拿到用户点击的按钮 true false

var rel = confirm('confim');

console.log(rel);

var rel = prompt('prompt');

console.log(rel);

switch结构: 只能判断变量的多种取值情况

var a = prompt('请输入一个成绩');

if (a >= 90) {

console.log('优秀');

}else if (a >= 80) {

console.log('良好');

}else if(a >= 60){

console.log('及格');

}else {

console.log('不及格')

}

var a = 1;

switch (a) {

// case只能列举具体的取值

case 1:

console.log('晴天');

break;

case 2:

console.log('阴天');

break;

case 3:

console.log('雨天');

break;

default:

console.log('aaaaa');

break;

}

**第八天**

函数: 拥有特定功能的代码片段 function 引用类型数据

函数名的本质: 函数的内存地址

function 函数名(形参列表) { 形参就相当于是函数内部的局部变量

函数体;

return;设置函数的返回值 函数运行到return 就跳出函数体

}

函数名(实参); 函数调用时将实参的值传递给形参 按照传参顺序一一对应

function定义的函数内部有一个属性叫arguments 该属性是一个类数组 存储了所有的实参的值

函数内部是一个局部的执行环境

执行环境被加载时 优先使用当前环境的变量

作用域:界定变量的使用范围

作用域链: 当作用域(函数)存在嵌套时 使用变量优先当前作用域中的变量 如果当前作用域中没有就去上一级作用域中查找 一直找到全局环境为止

在JS中通过函数划分变量的作用域: 全局作用域&局部作用域

全局作用域:函数外部就是全局环境

局部作用域:函数内部就是局部环境

全局变量:定义函数外部的变量就是全局变量 任何作用域都可以使用

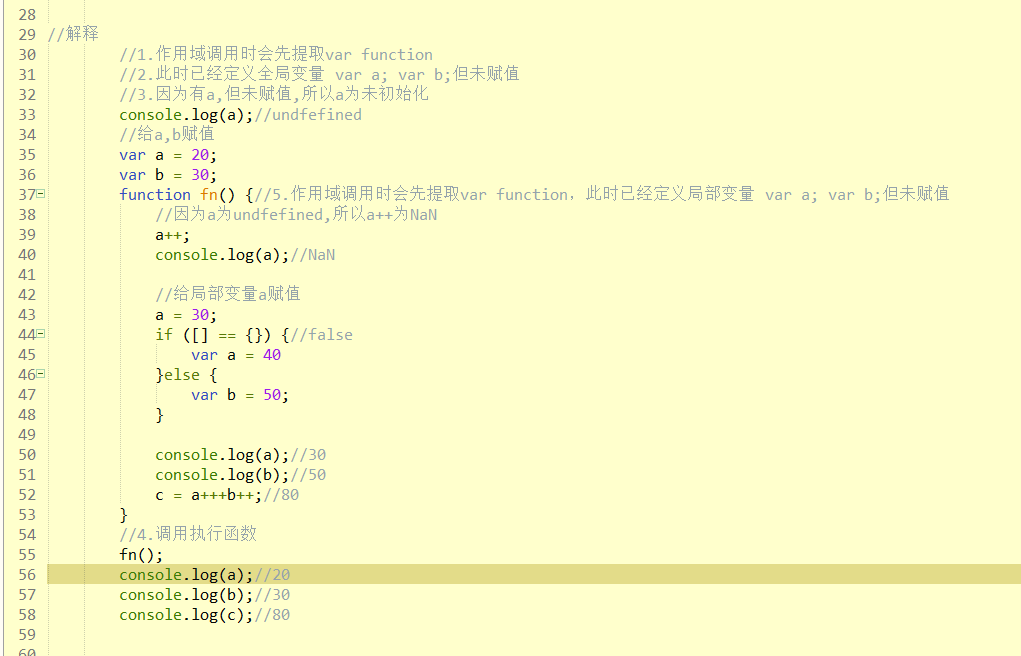
局部变量:定义函数内部的变量就是局部变量 局部变量仅限于函数内部使用

变量的生命周期: 全局变量的生命周期伴随整个页面 页面加载开始创建全局变量 页面关闭 全局表里销毁 局部变量的生命周期是当函数被调用时 局部变量创建 函数执行完 局部变量就销毁

当局部变量和全局变量重名时 优先使用函数内部的局部变量

变量提升: 预加载/预解析 在一个执行环境被加载时 先将该环境中var(定义变量) function(定义函数)定义的变量和函数定义出来

如果省略var 那这个变量就会成为全局变量



匿名函数:没有名字的函数 本身是个语法错误

匿名函数无法直接用

1.定义一个变量 指向 匿名函数

var fn = function () {

console.log(1);

}

fn();

2.函数自执行

(function (a, b) {

console.log(a + b);

})(10, 20);

闭包:

function fn() {

var a = 20;

var b = 30;

f就是闭包:

function f() {

a++;

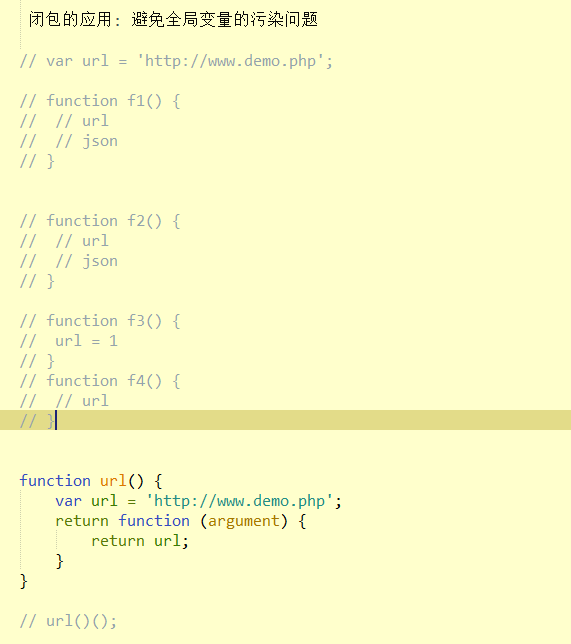
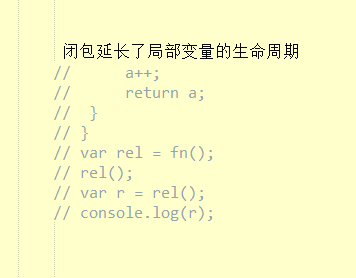
b--;

}

}

闭包延长了局部变量的生命周期

闭包的应用: 避免全局变量的污染问题

闭包的优点 : 使用闭包,可以将局部变量驻留在内存中,因此在闭包函数中仍然可以访问局部变量,有效的避免了全局变量的污染问题(全局变量污染会引起程序出现不可预测的问题,所以推荐使用封装的私有的局部变量)

闭包的缺点 : 由于闭包作用域中返回的局部变量被驻留在内存中,所占用的内存资源不会被立即被销毁回收,所以可能会占用更多的内存,要谨慎使用闭包

this: 函数中的一个指针 指代了一个对象

function函数中的this指向函数的调用者

定义在全局环境中的函数和变量会成为window 的属性和方法

直接调用的函数中 this都指向window

**第九天**

JS中var定义的变量根据函数划分作用域

es6新增块级作用域: let const定义变量 是根据代码块划分作用域 代码块内就是局部变量 代码块外就是全局变量

var:根据函数划分作用域 可以变量提升 允许重复定义

let/const:根据代码块划分作用域 不能变量提升 先定义后使用 不允许重复定义

let定义的是变量

const定义的是常量 一经定义 就不能修改

es6新增箭头函数: 新增的函数的定义方法 arrow function

箭头函数本身是一个匿名函数

(形参) => {

函数体

}

var fn = ()=>{

console.log('hello');

}

// fn();

(()=>{

console.log('hello');

})();

1.如果箭头函数的函数体只有一个语句 可以省略{}

var sum = (a, b)=> console.log(a + b);

sum(10,20);

2.如果箭头函数的函数体只有一个语句 并且该语句是return 可以省略{}和return

var f2 = (a, b)=>a \* b;

var rel = f2(10,20);

console.log(rel);

3.如果箭头函数的参数只有一个 可以省略() 0个或者多个不能省略

var f3 = a=>a\*a;

console.log(f3(10));

4.如果箭头函数的函数体只有一个语句 并且该语句是return 可以省略{}和return 如果返回值是一个对象 需要用()将对象括起来

var f4 = ()=> ({

name:"zhang3"

})

console.log(f4());

箭头函数中的this跟定义时上下文中的this保持一致

var o = {

name:'zhang3',

fn:()=>{

console.log(this);

console.log(this.name)

}

}

**第十天**

编程思想

面向对象:

面向过程:

对象:JS中一种数据类型 object 对象包含了属性和方法 属性描述对象特征 方法描述对象的功能

以键值对的形式存储属性和方法

// var o1 = {

// }

对象的创建方式

1.构造函数创建

var o1 = new Object();

2.字面量创建

var o2 = {

name:'zhang3',

age:18,

eat:function () {

console.log(this.name+'吃饭');

}

}

访问对象的属性和方法 :

点语法

o2.name;

o2.age;

o2.eat();

// []

o2['name']

o2['eat']();

o2.name = 'li4';

console.log(o2);

console.log(window);

\_\_proto\_\_: 原型对象 通过对象访问原型

console.log(o2.\_\_proto\_\_ === Object.prototype);

constructor: 对象所属的构造函数

console.log(o2.\_\_proto\_\_.constructor);

JS中通过构造函数区分类别

工厂模式

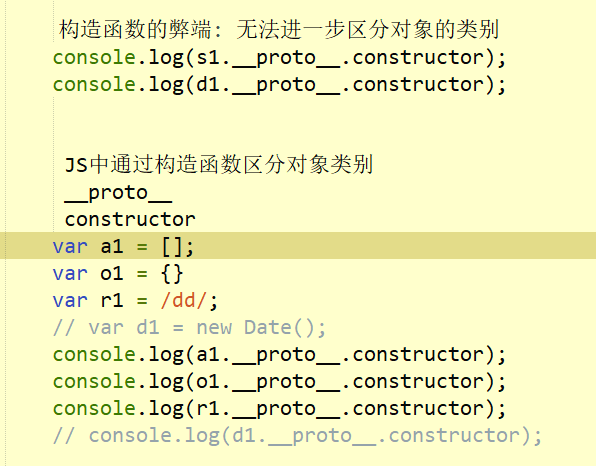
工厂模式:将给对象重复赋值的过程封装成一个函数

构造函数的弊端: 无法进一步区分对象的类别

JS中通过构造函数区分对象类别

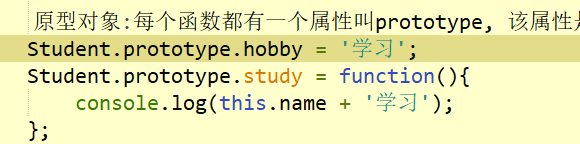
\_\_proto\_\_

constructor





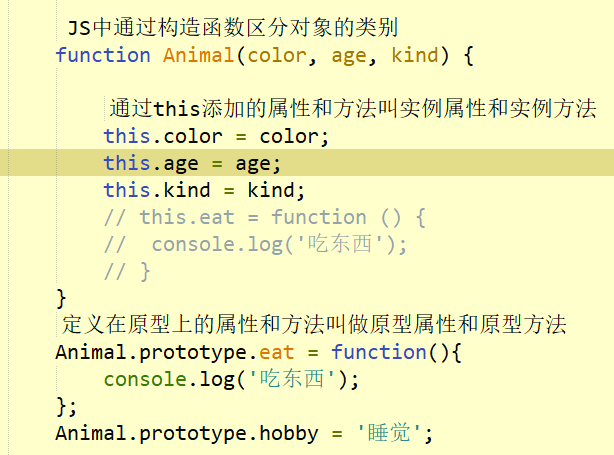
原型对象:每个函数都有一个属性叫prototype, 该属性是一个对象,定义在原型对象上的属性和方法可以被构造函数创建出来的对象共享



**第十一天**

对象 == 实例

实例化:使用构造函数创建对象



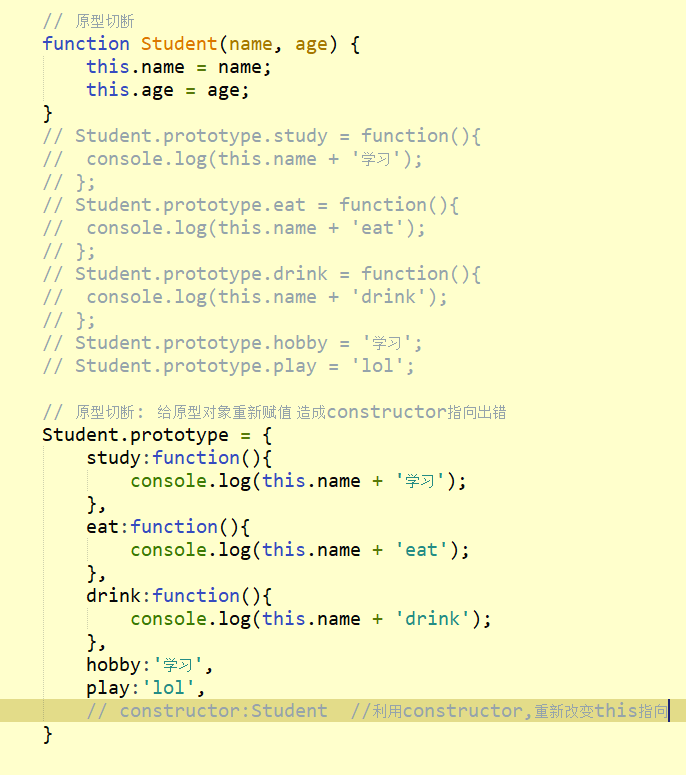
通过对象使用原型属性和原型方法:只读

a1.\_\_proto\_\_.hobby = '🐝'; \_\_proto\_\_可以改变原型属性

原型模式: 当一个对象使用属性和方法时 优先使用实例属性\方法, 如果没有该实例属性\方法, 就去原型上查找 如果原型上也没有 就去父类的原型的原型查找 只到找到Object的原型为止 如果还没有 属性获取到undefined 方法就会报错

原型切断: 给原型对象重新赋值 造成constructor指向出错

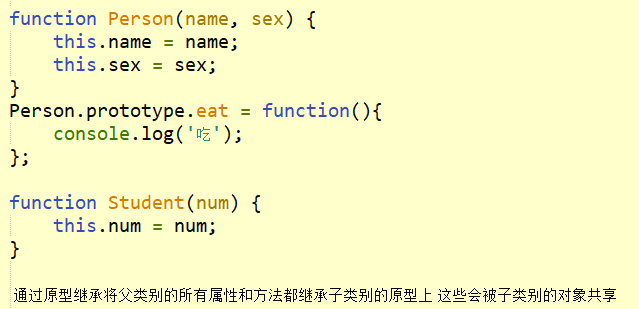
constructor:Student 利用constructor,重新改变this指向



继承:子类别拥有父类别的属性和方法

继承方式: 原型继承 对象冒充继承 组合继承 寄生继承 寄生组合继承 组合寄生继承...

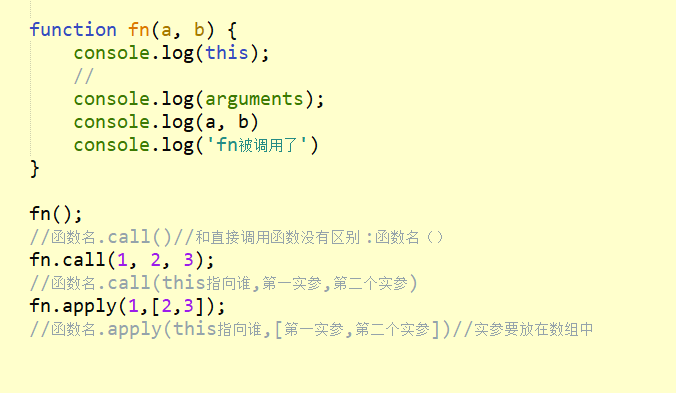
原型继承: 子类别的原型等于父类别的一个实例

通过原型继承将父类别的所有属性和方法都继承子类别的原型上 这些会被子类别的对象共享

call&apply:都是函数的方法

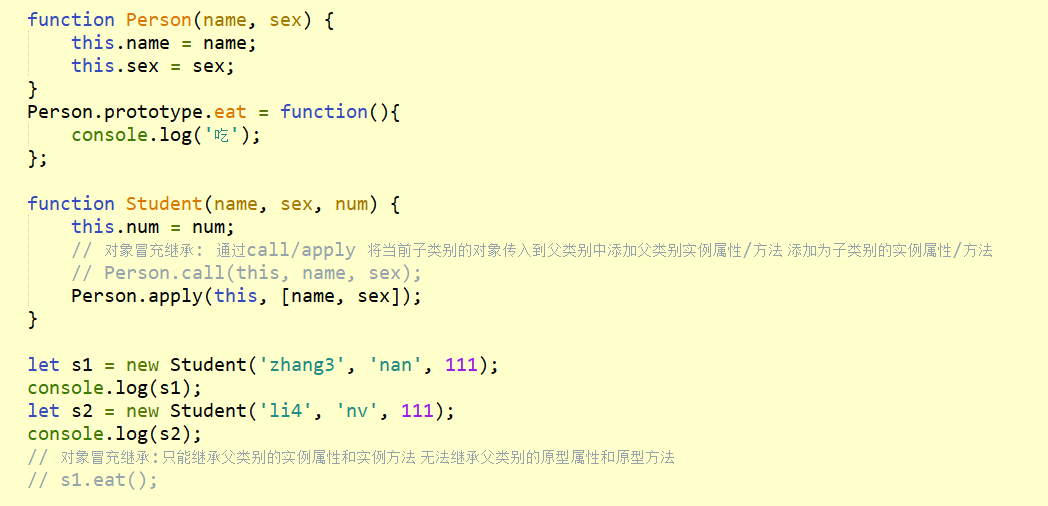
如果直接使用就和直接调用函数没有区别

如果传参可以修改函数中的this指向

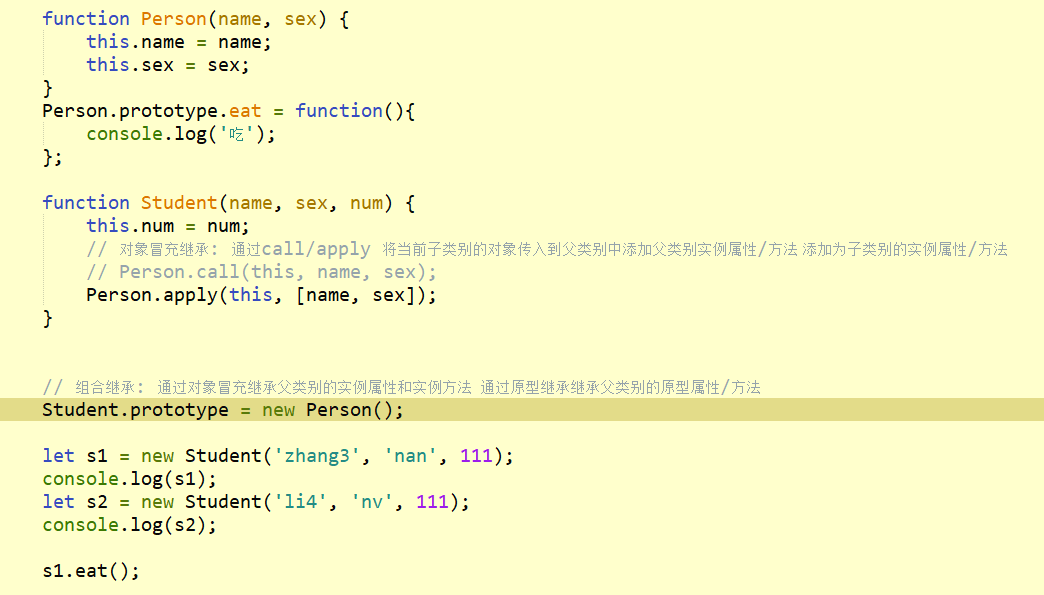


对象冒充继承: 通过call\apply 将当前子类别的对象传入到父类别中添加父类别实例属性\方法 添加为子类别的实例属性\方法

对象冒充继承:只能继承父类别的实例属性和实例方法 无法继承父类别的原型属性和原型方法



组合继承: 通过对象冒充继承父类别的实例属性和实例方法 通过原型继承继承父类别的原型属性/方法



es6之前: JS中没有类的概念 使用构造函数

es6之后 新增类的概念 关键字class

class 类名 [extends 继承的类] {

构造器­­­­­

constructor(参数1,参数2,参数3) {

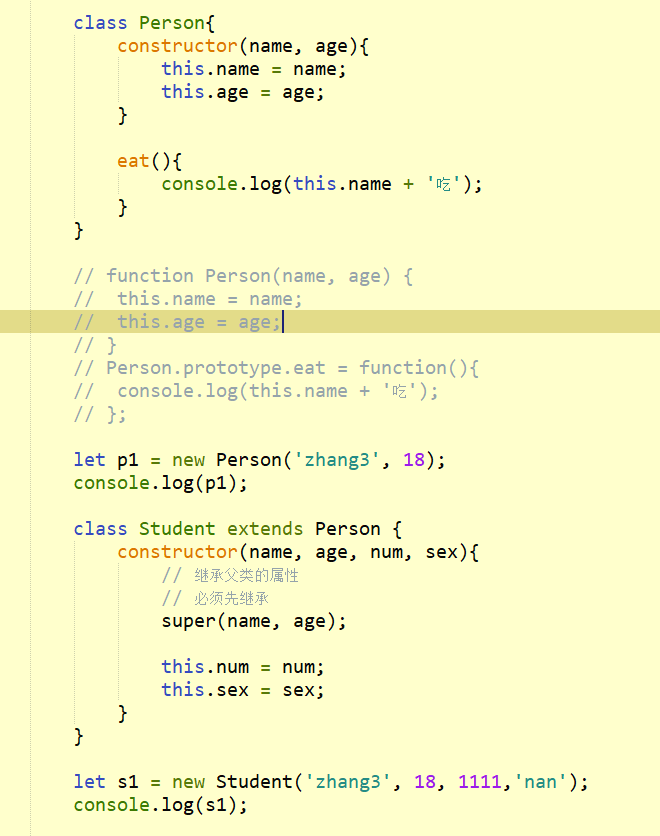
super(继承的属性1,继承的属性2,继承的属性3);

实例属性和实例方法

this.属性名 = 参数;

}

}



**第十二天**

BOM: browser object model 操作浏览器和页

window: 浏览器的一个实例 每个窗口都有对应的window对象

暂停页面加载

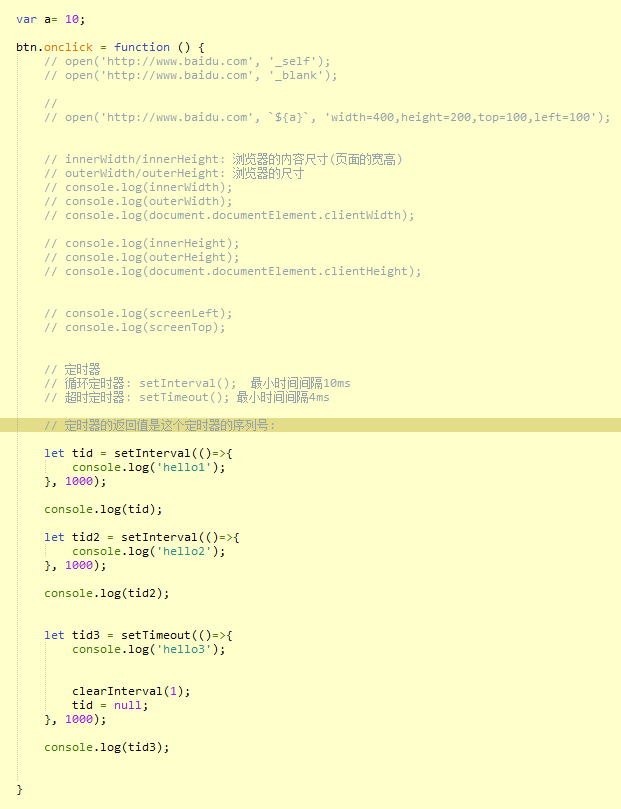
alert();

confirm();

prompt();

使用window的属性和方法可以省略window.可以通过属性名或者方法名直接使用

open(url, [name / target, 特效字符串]): 打开一个新窗口



href: 可读可写 当前页面的完整路径

location的属性和方法:

hash: 锚点

host: 主机名: 端口号

hostname: 主机名

pathname: 路径

port: 端口号

protocol: 协议 http file ftp https files ftps

search: 当前页面的参数

assign(): 替换

reload(): 刷新

replace(): 替换

页面跳转方法

setTimeout(() => {

location.replace('http://www.baidu.com')

location.assign('http://www.baidu.com', '\_blank')

location.reload(true);刷新

open(url, [name / target, 特效字符串]): 打开一个新窗口

location.href = 'http://www.baidu.com'

}, 2000)

**DOM操作**

节点操作

querySelector():返回满足条件的第一个元素

var box = document.querySelector('#box');

querySelectorAll():返回值是一个类数组

var rel = document.querySelectorAll('#box11');

console.log(rel)

document.getElementById();

document.getElementsByClassName();

document.getElementsByTagName();

var rel = document.getElementsByName('pear');

console.log(rel);

innerHTML/innerText: 都是非w3c标准属性 是浏览器提供的

console.log(wei.innerHTML);

console.log(wei.innerText);

wei.innerHTML = '<a href="#">hhhhh</a>';

wei.innerText = '<a href="#">hhhhh</a>';

// 节点创建

var newLi = document.createElement('li');

// newLi.innerHTML = '曹植';

var textNode = document.createTextNode('曹植');

newLi.appendChild(textNode);

// 追加节点

wei.appendChild(newLi);

// 节点删除

wei.removeChild(xia);

//节点替换

父元素.replaceChild(新节点, 被替换的节点);

可以使用已有的节点进行替换

wei.replaceChild(guan, xia);

// 节点插入

// 父元素.insertBefore(新节点,被插入的节点);

shu.insertBefore(xia, guan);

// 节点复制 cloneNode(isDeep);

var copyGuan = guan.cloneNode(true);

console.log(copyGuan)

常用属性

dom对象 dom元素 dom节点 节点对象

console.log(box.id);

// box.id = 'box1';

// console.log(box1);

console.log(box.className);

// box.style: 行内样式

console.log(box.style.width);

计算样式: 元素最终生效的样式

获取计算样式: 浏览器兼容

获取计算样式对象

var rel = getComputedStyle(box,false) || box.currentStyle;

console.log(rel.width);

console.log(getStyle(box, 'fontSize'));

function getStyle(dom, stylePro) {

return (getComputedStyle(dom,false) || dom.currentStyle)[stylePro];

}

// tagName/nodeName:标签名 大写

console.log(box.tagName, box.nodeName);

box.innerHTML;

box.innerText;

无法通过点语法访问对象的自定义属性

console.log(box.address);

//获取元素属性的通用方法

setAttribute();//设置/修改/添加

getAttribute();//获取

removeAttr();//删除

console.log(box.getAttribute('id'))

console.log(box.getAttribute('address'))

console.log(box.getAttribute('class'))

box.setAttribute('address', 'shanghai')

box.removeAttribute('class');

box.setAttribute('class', box.getAttribute('class').replace('pear', ''));

层次属性

// 所有子元素

// children:所有子元素节点

// childNodes:元素节点和文本节点

console.log(box.children);

console.log(box.childNodes);

// firstChild:第一个节点 包含文本节点

// firstElementChild

console.log(box.firstChild);

console.log(box.firstElementChild);//第一个元素节点

// lastChild

// lastElementChild

// previousSibling

// previousElementSibling

// nextSibling

// nextElementSibling

console.log(para.previousSibling);//上一个文本节点

console.log(para.previousElementSibling);//上一个元素节点

console.log(para.nextSibling);//下一个文本节点

console.log(para.nextElementSibling);//下一个元素节点

// 获取元素的父元素

console.log(box.parentNode);

// write: 修改body的内容

window.onload = function () {

document.write('hello');

}

Dom 方位属性&方位对象

box.onclick = function () {

// clientWidth/Height: 元素可视区的大小 content+padding

console.log(this.clientWidth);

// offsetWidth/offsetHeight:元素在页面上占据的尺寸(盒模型的尺寸): contetn+padding+border

console.log(this.offsetWidth);

// scrollWidth/scrollHeight:元素内容可滚动的区域大小

console.log(this.scrollWidth);

console.log(this.scrollHeight);

// clientLeft/Top:边框的宽度

console.log(this.clientLeft);

// offsetLeft/Top:距离有定位属性的父元素的距离 如果父元素都是静态定位 就是距离浏览器可视区左上角的位置

console.log(this.offsetLeft);

// scrollLeft/Top:元素内容滚动过的距离

console.log(this.scrollLeft);

console.log(this.scrollTop);

}

方位对象

// 获取元素的方位对象:width/height top/left/right/bottom x y

console.log(box.getBoundingClientRect().top);

console.log(box.getBoundingClientRect().left);

console.log(box.getBoundingClientRect().right);

console.log(box.getBoundingClientRect().bottom);

console.log(box.getBoundingClientRect().x);=

console.log(box.getBoundingClientRect().y);

console.log(box.offsetLeft);

事件类型

事件: 用户在浏览器上的操作

只要用户在浏览上有操作 就一定会有事件产生

事件类型:

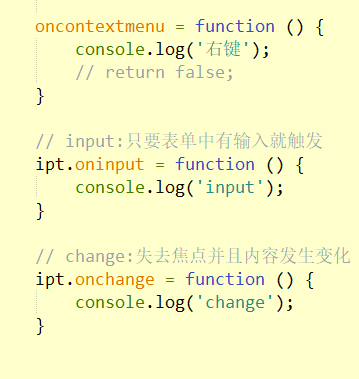
鼠标事件:click dblclick mousedown mouseup mousemove mouseover mouseout mouseenter mouseleave wheel contextmunu

键盘事件:keydown keypress keyup

表单事件:submit reset input change select focus blur

浏览器事件:load resize scroll error online offline readystatechange

移动端事件:touchstart touchmove touchend ...



mainDiv.onclick = function () {  
console.log('click  只有鼠标左键触发');  
}  
mainDiv.onmousedown = function (e) {  
console.log('mousedown ');  
  
let evt = e || event;  
console.log(evt.buttons);  
console.log(evt.button);  
}

事件绑定

事件模型:元素响应事件的方式

原始事件模型/行内事件: 将事件绑定作为标签属性

传统事件模型/事件绑定/dom一级事件:dom.on+事件类型

事件绑定的缺陷: 同一个元素的同一个事件类型只能绑定一个回调函数

box.onclick = function(){

this.style.backgroundColor = 'red';

}

// console.log(box.onclick);

事件监听

事件绑定/dom一级事件/传统事件模型: dom.on+事件类型

事件监听/dom二级事件/现代事件模型:w3c提供了一组事件监听的方法 addEventListener()/removeEventListenter();

addEventListener(事件类型, 回调函数, 是否捕获(false表示冒泡))

// 监听匿名回调函数无法被移除

box.addEventListener('click', fn, false);

function fn() {

this.style.backgroundColor = 'red';

console.log('red');

}

box.addEventListener('click', function () {

this.style.width = '400px';

}, false);

setTimeout(()=>{

box.removeEventListener('click', fn, false);

},2000);

w3c标准:addEventListener()/removeEventListenter()

IE老版本: attachEvent(on+事件类型,回调函数)/detachEvent(on+事件类型,回调函数)

// 事件监听兼容

if (addEventListener) {

box.addEventListener('click', function () {

console.log('click');

}, false);

}else {

box.attachEvent('onclick', function () {

console.log('click');

});

}

**第十三天**

事件: 用户在浏览器上的操作

只要用户在浏览器上有操作 就一定会有事件产生 事件产生之后还会沿着dom结构传递

事件流指的是元素获得事件的方式

事件传递: 从产生事件的元素到父元素之间传递

事件冒泡: 事件传递的默认方式 从里往外

事件捕获: 子元素有事件产生 设置了捕获的父元素优先响应

通过事件冒泡将多个子元素的同一种事件类型的回调函数都绑定给父元素统一处理

只要用户在浏览器上有操作就一定会有事件产生 只要事件产生会沿着dom结构向父元素传递 浏览器会将事件相关的信息封装成一个对象 就是事件对象

获取事件对象: w3c标准 通过事件的回调函数的参数传递

非w3c标准: 将事件对象添加为winodw的event属性

MouseEvent

// clientX/Y:事件产生时鼠标到浏览器左上角的坐标

// offsetX/Y:事件产生时鼠标到产生事件的元素的左上角的坐标

// pageX/Y:事件产生时鼠标到页面的左上角的坐标

// screenX/Y:事件产生时鼠标到屏幕左上角的坐标

// type:事件类型

// target/srcElement:产生事件的元素

// timeStamp:事件产生时距离页面打开的时间间隔

// path:事件传递的路径

// 功能键: 产生时 是否按下功能键

metaKey

shiftKey

altKey

ctrlKey

fromElement

toElement

阻止事件冒泡 无法阻止捕获

evt.cancelBubble = true;

evt.stopPropagation();

// 阻止默认事件

evt.preventDefault();

return false;

keyDown keyUp keyPress

键盘事件只能绑定给body html document window

KeyboardEvent

onkeydown = function (e) {

let evt =e || event;

// console.log(evt.target);//body

// console.log(evt.path);

console.log(evt);

// 键码 : ascii

// charCode:只有keypress事件才有

// keyCode

// key:区分大小写

// 判断组合键

// metaKey

// shiftKey

// altKey

// ctrlKey

}

**D:\专业\老师发的\第七个月\15、16、17、18**