1. HTML语义化：

HTML中自带一些标签，这些标签有特定含义，比如H1-H6代表标题、p代表段落、img代表图片、HTML5里面也推出了一些新的标签：header、footer、nav这些标签都有特定的含义，使用的时候应该尽可能的按照它的语义来使用。

好处：有利于程序员阅读、便于团队开发和维护，浏览器读取方便，有利于搜索引擎优化。

1. 行级块级元素

行级:不能设置宽高（内容撑的） 水平排列

span a I b strong em

块级元素：可以设置宽高 垂直排列

div p ul li h1-h6 ol

display:blockI(块级) inline（行级）inline-box(既可以设置宽高 又可以水平分布)

Img button text既可以设置宽高 又水平排列

1. CSS常见选择器：

id class 标签

div p 后代选择器

div>p 子代选择器

div,p 分组选择器

伪类选择器：hover{}

伪元素选择器：：after{}

属性选择器 div[a]

E+F E后面紧邻的兄弟 E~F E后面所有的兄弟

1. css优先级

！import 最高

第一等级：代表内联样式，如style=””,权值为1000

第二等级：代表id选择器。如#content，权值为100

第三等级：代表类，伪类和属性选择器，如.content 权值为10

第四等级：代表标签选择器和伪元素选择器，如div p

id

class

1. float

left right

清除浮动方式：

1 父元素加overflow：hidden

2 父元素 float

3 已知父元素高 设置高

4 clearfix::after{

Content:’’;

Display:block;

Clear:both;

}

1. 定位 position

Static 默认值

Absolute 绝对定位 相对于最近已定位的祖先元素 定位后空间释放

Relative 相对对位 相对于自己初始位置 定位后空间不释放

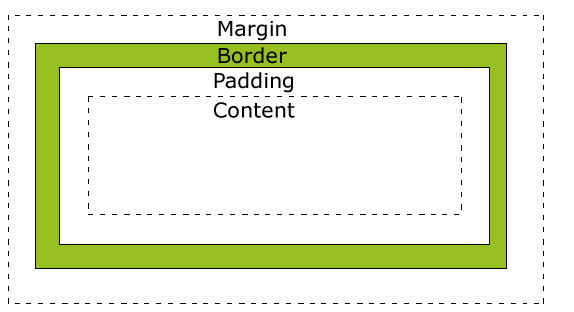
Fixed 固定定位 相对于浏览器位置 定位后空间释放

Sticky 粘性定位 相当于吸顶效果

1. 盒模型

标准盒模型

width属性 content的宽度



怪异盒模型（ie6 及ie6一下浏览器 不设置DOCTYPE）

Width 属性 content+padding+border 宽度

Box-sizing:content-box 标准

Border-box 怪异

1. 外边距合并

外边距合并

1. 父元素overflow：hidden；
2. 父元素浮动或定位 float：left position：absolute
3. 子元素浮动或定位
4. 父元素加边框
5. BFC（Block formatting context）块级格式化上下文

它是一个独立的渲染区域，只有Block-level box 参与。

它规定了内部Block-level box 如何布局，并且与这个区域外部毫不相干。

BFC布局规则：

1. 内部的Box会在垂直方向，一个接一个地放置。
2. Box垂直方向的距离由margin决定。属于同一个BFC地两个相邻Box的margin会发生重叠。
3. 每个元素的margin box的左边，与包含块border box的左边接触（对于从左往右的格式化）
4. BFC的区域不会与float box重叠。
5. BFC就是页面上的一个隔离的独立容器，容器里的子元素不会影响到外面的元素，反之亦然。
6. 计算BFC高度时，浮动元素也参与计算。

哪些元素会生成BFC？

根元素 html

Float属性不为none

Position为absolute或fixed

Display为inline-block, table-cell,table-caption,flex,inline-flex

Overflow不为visible

1. 重排 重绘
2. 重绘就是在一个元素的外观被改变，但没有改变布局（宽高）的情况下发生，如改变outline、背景色等等。
3. 重排就是DOM的变化影响到了元素的几何属性（宽和高），浏览器会重新计算元素的几何属性。如：改变窗口大小、改变文字大小、内容的改变、浏览器窗口变化等等。
4. reflow（回流）：当浏览器发现某个部分发生了点变化影响到了布局，需要倒回去重新渲染，内行称这个回退的过程叫reflow
5. display ： inline block inline-block flex table cell
6. 显示隐藏：display：none block; 不占空间 没有事件

opacity:0 1 占空间 有事件

visibility: hidden visible 占空间 没有事件

1. JS数据类型

基本数据类型： number sting boolean null undefined symbol

Null 空对象 document.getElementById(‘div1’)

Undefined 定义了没有赋值 数组越界 访问对象没有属性 函数默认返回undefined

Typeof ‘abc’ string 判断数据类型

‘null’ object 返回空对象

Array [1，2，3] object 返回对象

{name:’zs’} object

引用数据类型： array object （对象和数组）

判断对象还是数组：

var arr=[1,2,3];

console.log( arr.constructor == Array) 返回 true or false

var obj={name:’zs’}

console.log( obj.constructor == Object) 返回 true or false

console.log( arr instanceof Array)// 判断arr是否为Array实例化对象

console.log(Array.isArray(arr));

1. 对象\_类

<script>

var obj = {

name: 'zs',

say:function(){}

}

//类 具有相同属性和方法的对象的集合

//属性写在构造函数中 方法写在原型对象下

//属性写在原型对象下 1.不能传参数 2.引用数据类型 有一个改另一个也改了

function Person(name,age) { //构造函数 类

this.name = name;

this.age = age;

//this.eat = function () {

// console.log('....is eating')

//}

}

//prototype 原型对象 定义在原型对象下的所有属性和方法能被所有实例化对象共享

Person.prototype.eat = function () {

console.log(this.name+'...is eating')

}

var person1 = new Person('ls',20);

console.log(person1)

person1.eat();

</script>

1. 栈\_堆

**栈(stack)**：栈会自动分配内存空间，会自动释放，存放**基本类型**，简单的数据段，占据固定大小的空间。

javascript的引用数据类型是保存在堆内存中的对象。

1. 继承

//JS高级程序设计

//原型继承

function Animal(name, age) {

this.name = name;

this.age = age;

}

Animal.prototype.eat = function () {

console.log(this.name+'...is eating');

}

//继承属性用 在构造函数中.call() 继承属性

function Cat(name,age) {

Animal.call(this, name, age);

}

//原型对象=new 父类的实例化对象

Cat.prototype = new Animal();

//缺点就是这个构造函数有问题，需要指回去。

Cat.prototype.constructor = Cat.constructor;

var cat1 = new Cat('xx',5);

console.log(cat1);

1. this指向

//1.函数中 指向window

function a() {

console.log(this);

}

window.a();

//2.定时器中 指向window

setInterval(function () { console.log(this) }, 100)

//3.对象

obj = { name: 'zs', say: function () { console.log(this) } }

obj.say();

//4. 事件中 this指向点击的元素

//5. 类中 this 指向实例化对象

//改变this指向

function a(a,b) {

console.log(this,a,b)

}

objj = { name:'zs'

}

//call() apply() bind()

a.call(obj, '1', '2');

a.apply(obj, ['1', '2']);

a.bind(obj, '1', '2')();//需要自己调用

1. 事件模型

冒泡 捕获

事件冒泡 从里往外

事件捕获 从外往里

先捕获后冒泡 先从外往里 后从里往外

var odiv1 = document.getElementById('div1')

odiv1.onclick = function () { console.log(111) };//onclick绑定事件，后面的事件覆盖前面的事件

//给元素绑多个事件 addEventListener

odiv1.addEventListener('click', function () { console.log(111) }, false);

odiv1.addEventListener('click', function () { console.log(22) }, false);

1. 事件源

e.target 点谁触发的谁就是事件源

谁的事件 this就是谁

e．currenttarget 和this 一样

event.stopPropagation() 方法阻止事件冒泡到父元素 e.cancelBubble=true

e.preventDefault()阻止默认行为 或 return false e.returnValue=false;

1. 事件委托

<button id="btn">click</button>

<ul>

<li>1</li>

<li>2</li>

</ul>

//事件委托 解决后生成元素的事件绑定问题

<script>

var oBtn=document.getElementById('btn');

var aLi = document.getElementsByTagName('li');

var oUl = document.getElementsByTagName('ul')[0];

//for (var i = 0; i < aLi.length; i++)

//{

// aLi[i].onclick = function () { console.log(this.innerHTML)}

//}

oBtn.onclick = function () {

var oLi = document.createElement('li')

oLi.innerHTML = Math.random();

oUl.appendChild(oLi);

}

//点击子元素的时候 通过事件冒泡 会触发父元素的点击事件 通过e.target判断

oUl.onclick = function (e) {

if (e.target.tagName == 'LI') {

console.log(e.target.innerHTML)

}

}

//jQuery

$('li').on('click', function () { console.log($(this).html())})

$('ul').on('click', 'li', function () {console.log($(this).html() })

</script>

1. 闭包

<%-- 闭包是指外部函数里声明内部函数，内部函数引用外部函数里的局部变量，

这样当外部函数调用完毕以后，局部变量不被释放，可以一直使用。--%>

<ul>

<li>11</li>

<li>22</li>

<li>33</li>

<li>44</li>

</ul>

<script>

var oUl = document.getElementById('ul');

var aLi = document.getElementsByTagName('li');

for (var i = 0; i < aLi.length; i++) { //i=4

(function () {

aLi[idx].onclick = function () {

console.log(idx);

}

})(i)

}

for (let i = 0; i < aLi.length; i++) {

aLi[i].onclick = function () { console.log(i)}

}

1. CSS3常用语法

//常用属性

border-raduis 给div元素添加圆角的边框

box-shadow background-size cover/contain

动画

transform

rotate(30deg)

translate()

scale()

skew

//过渡 需要改变属性 触发

transition:width 2s ease 3s;

//动画 可以直接执行 可以执行更复杂的动画

animation: run 2s ease 3s forwards

@keyframes run{}

弹性盒模型

父元素：

display:flex

flex-direction:row row-reverse column column-reverse

横向从左到右排列 横向相反 即纵向从上往下排列 纵向相反

align-items:flex-start flex-end center baseline stretch

纵轴垂直对齐方式

justify-content:flex-start flex-end center space-between

横轴水平对齐方式

子元素：

flex：<'flex-grow'><'flex-shrink'><'flex-basis'>

flex-group:份数

<ul> display:flex

<li> flex-group:2

</li>

</ul>

水平垂直全部居中

利用vertical-align,text-align,inline-block实现

.parent{display:cable-cell;vertical-align:middle;text-align:center;}

.child{display:inline-block;}

利用绝对定位实现

.parent{position:relative}

.child{position:absolute;top:50%;left:50%;transform:translate(-50%,-50%);}

利用flex实现

.parent{display:flex;justify-content:center;align-items:center;}

<div id="div1"> </div>

1. HTML5

//h5新增的语义化元素，以及标签

<!-- header footer nav aside canvas input type="email/num/data/color" -->

JSON.parse() //将字符串转化成json对象

JSON.stringify()// json对象转换成字符串

{

"name":"zhangsan",

"age":20

}

'{"name":"zhangsan","age":20}'

<script>

function clone(obj) { //深克隆 如果属性还是对象的话需要递归再克隆一次

var newObj = {};

for (key in obj)

{

if (typeof obj[key] == "object") {

newObj[key] = clone(obj[key]);

}

else {

newObj[key] = obj[key];

}

return newObj;

}

}

//浅克隆

var person = { name: 'zs', age: 20 };

function clone(obj) { //key 属性 obj对象

var result = {}

for (key in obj)

{

result[key]=obj[key]

}

return result;

}

var person2 = clone(person)

person.name='lisi'

console.log(person2)

var str = JSON.stringify(person);

var person3 = JSON.parse(str);

</script>

1. Session

var date=new Date();

date.setDate(29);

document.cookie = "name=11 expires=" + date;

console.log(document.cookie);

function getCoolie(key) {

"name='zs';age=20"

var arr = document.coo4.split(":");//["name='zs';age=20"]

for (var i = 0; i < arr.length; i++)

{

var arr2 = arr[i].split("="); //["bbb","aaa"] ["name","zs"]

if (key == arr2[0])

{

return arr2[1];

}

}

}

console.log(getCookie('age'))

本地存储。

sessionStorage.setItem('name', 'zs');

console.log(sessionStorage.getItem('name'));

sessionStorage.removeItem('name');

sessionStorage.clear('name');

localStorage.setItem('age', '111');

//sessionStorage、localStorage cookie session

共同点：都是保存在浏览器端，且同源的

区别：1、cookie数据始终在同源的http请求中携带 se loc 仅在本地保存

2、存储大小限制也不同

3、生命周期不同

4、作用域不同

1. Jquery原理

JQ最核心的是$，他是一个函数，函数中返回一个自定义的JQ对象，Jquery它是一个类，

在这个类的原型下有许多JQ自定义的方法，每次调用$返回一个新的JQ对象，

同时在Jquery类中根据传入参数的类型，函数或者字符串，把选中的元素存在数组中，

遍历这个数组，取出每个元素进行dom操作。

每个方法中，return this,this就是在$函数中返回的JQ对象..链式操作

$('#div1')

$('#div1').html()

document.getElementById("#div1").innerHTML

$('#div1')[0] 原生对象

$('#div1').get(0).innerHTML

$(document.getElementById("#div1"))

//$('#div1') $('.aa')

function $(selector){return new Jquery(selector)}

function Jquery(selector){

//'#div1'

if(selector.charAt(0)=="#")

{

this.elements=[];

this.elements.push(document.getElementById(selector.substr(1)))

}

if(selector.charAt(0)==".")

{

this.elements=document.getElementByClassName(selector.substr(1))

}

if(typeof selector == 'function')

{

document.addEventListener('DomContentLoad',function(){

selector();

})

}

}

Jquery.prototype.css()=function(attr,val){

if(typeof attr == "string"&&typeof val =='string'){

for(var i=0;i<this.elements.length;i++){

this.elements[i].style[attr]=val;

}

}

}

Jquery.prototype.on = function (eventName, fn) { //绑定事件

for (var i = 0; i < this.elements.length; i++) {

this.elements[i].addEventListener(eventName, fn);

}

return this;

}

$('.aa').css('background', 'red')

$('.aa').on('click', function () { console.log('111'); })

$('.aa').css('background', 'red').on('click', function () { console.log('111'); })

//想实现上述，$返回JQ对象，在CSS属性函数里 return this 在on 函数里 return this。

$(function(){ //DomContentLoad 文档就绪函数

console.log(111)//不会覆盖

})

$(function(){ //DomContentLoad 文档就绪函数

console.log(22)

})

window.onload=function(){console.log(1111)}

window.onload=function(){console.log(2222)}//页面加载完覆盖第一个，显示第二个

</script>

1. 移动端

响应式布局：

c3 media query

百分比

弹性盒模型

rem

点透

1px transform scale

<script>

//rem原理

; (function (win) {

var doc = win.document;

var docEl = doc.documentElement;

var tid;

function refreshRem() {

var width = docEl.getBoundingClientRect().width;//获取屏幕宽度

var rem = width / 6.4;//将屏幕宽度分成6.4份，1份为1rem

docEl.style.fontSize = rem + 'px';

console.log(0.14\*rem);

}

win.addEventListener('resize',function(){

clearTimeout(tid);

tid = setTimeout(refreshRem, 10);

},false);

win.addEventListener('pageshow',function(e){

if (e.persisted) {

clearTimeout(tid);

tid = setTimeout(refreshRem, 10);

}

})

})

1. ES6

es6 新特性

let(定义变量) const(定义常量) 块级作用域(只对let const 有效)

arrow(箭头函数) class map set 数组解构 对象解构 数组对象字符串新加了一些函数

promise 模版字符串

let var

<script>

var a; //相当于在上面定义了一个a但是没有赋值，所以输出 undefined

console.log(a);

var a = 5;

//1、let 没有变量提升

console.log(b); //会报错误，没有定义b

let b = 5;

//2、块级作用域

{

let c = 10;

}

console.log(c);

//3、不能重复声明

let x = 40;

let x = 20;

//4、临时失效区 在当前作用域 不允许你同名的变量进来

let v=10;

function d(){

console.log(v);

let v=20;

}

d();

//const 常量

const e=10;

e=20;

console.log(e);

const arr =[1,2,3]; //不想让他改呢

冻结

Object.freeze(arr);

arr[0]=0;

console.log(arr);

//解构

[a,b]=[1,2]

console.log(a);

//let{ f,g } = { f:1,g:2 };

//console.log(f,g);

let arrr=[1,2,3,2]

//console.log(new Set(arrr));

let arr2=new Set(arr);

//类数组转换成数组

let arr3=[...arr2];

</script>

<ul>

<li>1</li>

<li>2</li>

<li>3</li>

</ul>

<script>

//箭头函数： 找父作用域中的this

var aLi=document.getElementsByTagName('li');

for(var i=0;i<aLi.length;i++)

aLi[i].onclick=function(){

// setTimeout(function(){console.log(this);},500)//this 指向window

// setTimeout(()=>{console.log(this)},500) //箭头函数中this指向声明时的this 而不是执行时的this

console.log(this);//li0

setTimeout(function(){console.log(this);}.bind(this),500) //bind() 把this指向改了，但没有调用

}

</script>

<script>

//es6 class

//constructor里声明属性，下面声明方法

class Person{

constructor(name){

this.name=name;

}

say(){console.log(this.name+"say....")}

static sleep(){console.log(this.name+"say....")}//静态方法，不能被子类继承。

}

class Coder extends Person{

constructor(name,age){

super(name);

this.age=age

}

eat(){

console.log(this.name+"is eating...")

}

}

var person1=new Person("zs")

person1.say();

var coder=new Coder("lsi",20)

console.log(coder);

coder.say();

</script>

<%-- promise 就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。

一个promise可能有三种状态：等待（pending）、已完成（resolved）、已拒绝（rejected）

一个promise的状态只能从“等待”转到“完成”或者“已拒绝”，不能逆向转换。

promise必须实现then方法，（可以说，then是promise的核心），而且then必须返回一个promise

--%>

<script>

//ajax(function(res){

// if(res.a){

// ajax(function(res){

// if(res.id){

// ajax(function(){})

// }

// })

// }

//})

let p1=new Promise(function(resolve,reject){

setTimeout(function(){

let res ={code:1};

if(res.code==1){

resolve()

}else{

reject()

}

},500)

}).then(function(){

console.log(111);

return new Promise(function(resolve,reject){

setTimeout(function () {

console.log(22222);

resolve()

},500)

}).then(function(){console.log('成功')})

}).catch(function(){

console.log('失败')

})

let P1 = new Promise(function (resolve, reject) {

setTimeout(function () {

console.log(111);

resolve();

},500)

})

let P2 = new Promise(function (resolve, reject) {

setTimeout(function () {

console.log(555);

resolve();

},800)

})

let P3 = new Promise(function (resolve, reject) {

setTimeout(function () {

console.log(777);

resolve();

},300)

})

Promise.all([P1, P2, P3]).then(function () { //all是集体完成，race选中三者中最快的一个

console.log('over ok');

}).catch(function () {

console.log('over no');

})

</script>

<%-- 事件循环

除了广义的同步任务和异步任务，我们对任务有更精细的定义：

macro-task(宏任务)：包括整体代码script，setTimeout,setInterval

micro-task(微任务)：Promise，process.nextTick--%>

<%-- ajax请求 定时器属于异步。

先执行同步的任务，执行完之后去任务队列里找，先执行所有的微任务，接着

再从宏任务里开始，找到其中一个任务队列执行完毕，再执行所有的微任务。--%>

<script>

setTimeout(function () { //最后找宏任务，定时器。

console.log('定时器开始啦')

}, 0);

new Promise(function (resolve) {

console.log('马上执行for循环啦');

for (var i = 0; i < 10000; i++) {//先执行promise同步任务，以及console.log('代码执行结束')

i == 99 && resolve();

}

}).then(function () { console.log('执行then函数啦'); })//然后去任务队列中找异步的微任务，.then

console.log('代码执行结束')

</script>

<script>

//先同步任务，后异步任务。先微任务，后宏任务。

//1 7 6 8 2 4 3 5

//先执行1，定时器，process.nextTick放到任务队列,执行第一个new promise

//.then()放到任务队列里，然后去任务队列中找微任务 执行 process.nextTick（）6

//然后执行.then(8)，接着执行宏任务 输出2 process.nextTick（3）放到任务队列

//执行new promise 输出4 .then()放到任务队列里 然后执行process.nextTick（3）then.(5)

console.log('1');

setTimeout(function() {

console.log('2');

process.nextTick(function() {

console.log('3');

})

new Promise(function(resolve) {

console.log('4');

resolve();

}).then(function() {

console.log('5')

})

})

process.nextTick(function() {

console.log('6');

})

new Promise(function(resolve) {

console.log('7');

resolve();

}).then(function() {

console.log('8')

})

</script>

1. VUE

常用指令

v-if v-show v-on @ v-for v-model v-bind:

v-if v-show 区别

双向数据绑定原理

VueJS 利用ES5提供的访问器属性Object.defineProperty()方法，监控对数据的操作，

Object.defineProperty()方法第一个参数是要设置的对象，第二个参数是要对象的属性，第三个

参数是一个对象，里面有set和get方法，当设置属性值绘制获取属性值的时候就会执行这个set，get

方法。

vue就是通过set方法，做到vue数据变更时，同时渲染页面数据。

vue也会给dom对象绑定事件，这样的页面数据变更时，也会对应的修改data

订阅者模式

1、实现一个数据监听器Observer，能够对数据对象的所有属性进行监听，如有变动可拿到最新值并通知订阅者

2、实现一个指令解析器Compile，对每个元素节点的指令进行扫描和解析，根据指令模板替换数据，以及绑定相应的更新函数

3、实现一个Watcher，作为连接Observer和Compile的桥梁，能够订阅并收到每个属性变动的通知，执行指令绑定的相应回调函数，从而更新视图

4、mvvm入口函数，整合以上三者

<img src="https://segmentfault.com/img/bVBQYu?w=730&amp;h=390" alt="图片描述"/>

Object.defineProperty(obj,attr,{

get:function(){},

set:function(){}

})

<script>

data: {

}

</script>

<script src="VUE/vue.js"></script>

<%-- 3、生命周期-- 查资料！ --%>

<div id="app">

**{{**msg**}}**

</div>

<script>

var vm = new Vue({

el: "#app",

data: {

msg: 'aaa'

},

beforeCreate() {

console.log('beforeCreate...')

console.log(this.$data);

},

created() {

console.log('created...')

console.log(this.$data, this.$el);

},

beforeMount() {

console.log('beforeMount...')

console.log(this.$el);

},

mounted() {

console.log('mounted...')

console.log(this.$el);

},

beforeUpdate() {

console.log('beforeUpdate...')

console.log(this.msg);

},

updated() {

console.log('updated...')

console.log(this.msg);

},

beforeDestory() {

console.log('beforeDestory');

},

destoryed() {

//vm.$destory()

console.log('destoryed...');

}

});

</script>

4.组件传值

父组件到子组件 props

子组件到父组件 this.$emit()