

# CSS

## 行内元素和块级元素有什么区别

行内元素指的是书写完成后不会自动换行，并且元素没有宽和高，margin padding无效。块级元素写完后会自动换行，有宽高可以修改。行内元素可以display:inline-block;变成行内块元素 可以设置宽高 margin padding

块级元素有：address article aside audio blockquote dd div dl fieldset figcaption figure footer form h1-h6 header hgroup hr noscript ol output p section canvas table tfoot ul video


行内元素有：b, big, i, small, tt abbr, acronym, cite, code, dfn, em, kbd, strong, samp, var a, bdo, br, img, map, object, q, script, span, sub, sup button, input, label, select, textarea

## 盒子模型

dom采用的布局模型，可以通过box-sizing进行设置 1、content-box (w3c标准盒模型) width 与 height 只包括内容的宽和高，不包括边框（border），内边距（padding），外边距（margin）。2、border-box (ie盒模型) width = border + padding + 内容的宽度 3、padding-box padding计算入 width内 只有Firefox实现了这个值，它在Firefox 50中被删除。4、margin-box (浏览器未实现)

## 层叠上下文

关于will-change cpu gpu 硬件加速问题 <https://www.w3cplus.com/css3/introduction-css-will-change-property.html>

元素提升为一个比较特殊的图层，在三维空间中(z轴) 高出普通元素一等。触发条件： 1、根层叠上下文 (html) 2、position 3、css属性 flex、transform、opacity、will-change、overflow-scroll 层叠等级 

## 水平居中、垂直居中、水平垂直居中

## 选择器优先级

!important > 行内样式 > #id > .class > tag > \* > 继承 > 默认 选择器从右往左解析

## 清除浮动

额外标签、伪类元素、bfc overflow:hidden、父元素也设置浮动

## link和@import区别

1、link功能较多，可以定义rss,定义rel等作用，而@import只能用于加载css 2、当解析到link时，页面会同步加载css，@import所引用的css会等到页面加载完才能加载 3、@import需要ie5以上才能使用 4、link可以js动态引用，@import不行

# JS

## js

```
function Foo() {
    getName = function () { alert (1); };
    return this;
}
Foo.getName = function () { alert (2);};
Foo.prototype.getName = function () { alert (3);};
var getName = function () { alert (4);};
function getName() { alert (5);}
```

//答案:

```
Foo.getName();//2
getName();//4
Foo().getName();//1
getName();//1
new Foo.getName();//2
new Foo().getName();//3
new new Foo().getName();//3
```

----- 第二问 -----

```
function Foo() {
    getName = function () { alert (1); };
    return this;
}
var getName; //只提升变量声明
function getName() { alert (5); } //提升函数声明, 覆盖var的声明

Foo.getName = function () { alert (2);};
Foo.prototype.getName = function () { alert (3);};
getName = function () { alert (4); }; //最终的赋值再次覆盖function getName声明

getName(); //最终输出4
```

----- 第三问 ----- 第三问的 Foo().getName(); 先执行了Foo函数, 然后调用Foo函数的返回值对象的 getName属性函数。Foo函数的第一句 getName = function () { alert (1); }; 是一句函数赋值语句, 注意它没有var声明, 所以先向当前Foo函数作用域内寻找getName变量, 没有。再向当前函数作用域上层, 即外层作用域内寻找是否含有getName变量, 找到了, 也就是第二问中的alert(4)函数, 将此变量的值赋值为 function(){alert(1)}。

此处实际上是将外层作用域内的getName函数修改了。注意: 此处若依然没有找到会一直向上查找到window对象, 若window对象中也没有getName属性, 就在window对象中创建一个getName变量。之后Foo函数的返回值是this, 而JS的this问题博客园中已经有非常多的文章介绍, 这里不再多说。简单的讲, this的指向是由所在函数的调用方式决定的。而此处的直接调用方式, this指向window对象。遂

Foo函数返回的是window对象，相当于执行 window.getName()，而window中的getName已经被修改为alert(1)，所以最终会输出1 此处考察了两个知识点，一个是变量作用域问题，一个是this指向问题。

----- 第四问 ----- 直接调用getName函数，相当于 window.getName()，因为这个变量已经被Foo函数执行时修改了，遂结果与第三问相同，为1

----- 第五问 ----- new Foo.getName();,此处考察的是js的运算符优先级问题。 [https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Operator\\_Precedence](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Operator_Precedence) 通过查上表可以得知点 (.) 的优先级高于new操作，遂相当于是：

```
new (Foo.getName)();
```

所以实际上将getName函数作为了构造函数来执行，遂弹出2。

----- 第六问 ----- new Foo().getName()，首先看运算符优先级括号高于new，实际执行为 (new Foo()).getName() 遂先执行Foo函数，而Foo此时作为构造函数却有返回值，所以这里需要说明下js中的构造函数返回值问题。构造函数的返回值 在传统语言中，构造函数不应该有返回值，实际执行的返回值就是此构造函数的实例化对象。而在js中构造函数可以有返回值也可以没有。1、没有返回值则按照其他语言一样返回实例化对象

```
function F() {}  
//undefined  
new F()  
//F{}
```

2、若有返回值则检查其返回值是否为引用类型。如果是非引用类型，如基本类型 (string,number,boolean,null,undefined) 则与无返回值相同，实际返回其实例化对象。

```
function F() {return true}  
//undefined  
new F()  
//F{}
```

3、若返回值是引用类型，则实际返回值为这个引用类型。

```
function F() {return {a:1}}  
//undefined  
new F()  
//Object {a:1}
```

原题中，返回的是this，而this在构造函数中本来就代表当前实例化对象，遂最终Foo函数返回实例化对象。之后调用实例化对象的getName函数，因为在Foo构造函数中没有为实例化对象添加任何属性，遂到当前对象的原型对象 (prototype) 中寻找getName，找到了。遂最终输出3。

----- 第七问 ----- new new Foo().getName(); 同样是运算符优先级问题。最终实际执行为： new ((new Foo()).getName()); 先初始化Foo的实例化对象，然后将其原型上的getName函数作为构造函数再次 new。遂最终结果为3

用js实现随机选取10-100之间的10个且不重复的数字，存入一个数组。还要排序

```
let numS=new Set();
while (numS.size<10) {
    numS.add(parseInt(10 + Math.random() * 90));
}
console.log([...numS].sort((a,b)=>a-b));
```

请写出弹出值，并解释为什么

```
alert(a) //f a() {console.log(10)}
a();
var a = 3;
function a() {
    alert(10) //10
}
alert(a) //3
a = 6;
a();//Uncaught TypeError: a is not a function
```

请写出如下输出值，并写出把注释掉的代码取消注释的值，并解释为什么

```
var a = function people(num) {
    people = num;
    console.log(typeof people);//function
}
a(1)

var a = function people(num) {
    people = num;
    console.log(typeof people);//function
}
a(1);
console.log(typeof people());//people is not defined
```

这种函数表达式，**people()**只读只能内部访问

```
let fn = {
    a:function() {
        console.log(1);
    },
    b() {
        console.log(2);
    }
}
```

```

    },
    c:()=>{
        console.log(3);
    }
}
var n1 = new fn.a();
var n2 = new fn.b();//报错
fn.c()

```

## es6写法的不允许被new

```

var s = {
    a: function () {
        console.log(this);//a{}
    }
}

var f = s.a.bind(this);
new f();

var s = {
    a: function () {
        console.log(this);//window{}
    }
}

var f = s.a.bind(this);
f();

```

## new过后bind没乱用，new过后this变成函数的实例

```

this.a = 20;
var test = {
    a: 40,
    init: () => {
        console.log(this.a);
        function go() {
            // this.a = 60;
            console.log(this.a);
        }
        go.prototype.a = 50;
        return go;
    }
};
//var p = test.init();
//p();
new (test.init())();

```

## 请写出以下代码执行结果

```
function C1(name) {
    if (name) this.name = name;
}
function C2(name) {
    this.name = name;
}
function C3(name) {
    this.name = name || 'fe';
}
C1.prototype.name = "yideng";
C2.prototype.name = "lao";
C3.prototype.name = "yuan";
console.log((new C1()).name) + (new C2()).name + (new C3()).name);
```

```
function test(a) {
    this.a = a; //选择是否注释看效果
}
test.prototype.a = 3;
test.prototype.init = function () {
    console.log(this.a); //3
}
var s = new test();
s.init()
```

## 对象和闭包不能在一起 必须有分号

```
var obj = {
    name: 'xiao'
}
(function() {
    console.log('haha');
})();
```

## 请写出如下点击li的输出值，并用三种办法正确输出li里的数字

```
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
  <li>4</li>
  <li>5</li>
  <li>6</li>
</ul>
```

```

var liList = document.getElementsByTagName('li');
for (var i = 0; i < liList.length; i++) {
    liList[i].onclick = (function () {
        var index = i;
        // 返回了一个匿名函数
        return function () {
            console.log('我的索引是:' + index);
        };
    })()
    // liList[i].onclick = function() {
    //     console.log(this.innerHTML);
    // }
}

```

## 写出输出值，并解释为什么

考察点 按值传递 按引用传递

```

function test(m) {
    m = { v: 5 }
}
var m = { k: 30 };
test(m);
alert(m.v); //undefined

```

**\*\*函数参数是按值传递的\*\***

## 请写出代码执行结果，并解释为什么

```

function yideng() {
    console.log(1);
}
(function () {
    if (false) {
        function yideng() {
            console.log(2);
        }
    }
    console.log(typeof yideng) //undefined
    yideng();
})();

```

分3种情况 ie 老版本浏览器整个函数体提升 function yideng() {console.log(2)} 现在版本浏览器 函数名提升 yideng

[,,] // 求它的长度 不同浏览器长度不同

## es5、es6的继承

### es5

```
function Person(color) {
    this.color = color;
}
Person.prototype.say = function () {
    console.log('Hello world!');
}

function chinese(color) {
    Person.call(this, color)
}

var __prototype = Object.create(Person.prototype);
__prototype.constructor = chinese;
chinese.prototype = __prototype;
chinese.prototype.kungfu = function () {
    console.log('yongchun!');
}
var xiaoming = new chinese('yellow');
console.log(xiaoming);
```

### es6

```
class Person {
    constructor(color) {
        this.color = color;
    }
    say() {
        console.log(this.color);
    }
}

class chinese extends Person {
    constructor(color) {
        super(color)
    }
    kungfu() {
        console.log('yongchun!');
    }
}

let xiaoming = new chinese('yellow');
console.log(xiaoming.say());
console.log(xiaoming);
```



## 写出如下代码执行结果，并解释为什么

```
function fn() {
    console.log(this.length);
}
var person = {
    length: 5,
    method: function (fn) {
        fn()
    }
};
person.method(fn, 1)
```

window.length输出的iframe的个数

```
function fn() {
    console.log(this); //arguments
    console.log(this.length); //2
}
var person = {
    length: 5,
    method: function (fn) {
        // fn()
        console.log(arguments[0]); //fn(){}
        arguments[0]();
    }
};
person.method(fn, 1)
```

## 手写bind

<https://blog.csdn.net/u010552788/article/details/50850453>

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function (oThis) {
        if (typeof this !== 'function') {
            throw new TypeError('Function.prototype.bind - what is trying to be
            bound is not callable');
        }

        var aArgs = Array.prototype.slice.call(arguments, 1),
            fToBind = this,
            fNOP = function () {},
            fBound = function () {
                // this instanceof fBound === true时,说明返回的fBound被当做new的构造函数调用
                return fToBind.apply(this instanceof fBound
                    ? this
```

```

        : oThis,
        // 获取调用时(fBound)的传参.bind 返回的函数入参往往是这么传递的
        aArgs.concat(Array.prototype.slice.call(arguments)));
    };

    // 维护原型关系
    if (this.prototype) {
        // Function.prototype doesn't have a prototype property
        fNOP.prototype = this.prototype;
    }
    // 下行的代码使fBound.prototype是fNOP的实例,因此
    // 返回的fBound若作为new的构造函数,new生成的新对象作为this传入fBound,新对象的
    __proto__ 就是fNOP的实例
    fBound.prototype = new fNOP();

    return fBound;
};

}

function foo() {
    this.b = 100;
    return this.a;
}

var func = foo.bind({ a: 1 })
func();//1
new func();//foo {b: 100}

```

## 如何找到数组最大值

```

const findMax = (arr) => (arr.reduce((prev, next) => next > prev ? next :
prev));

```

给定一个大小为  $n$  的数组，找到其中的众数。众数是指在数组中出现次数大于  $\lfloor n/2 \rfloor$  的元素

```

let arr = [5, 2, 2, 3]
const majorityElement = arr => {
    arr.sort((a, b) => a - b)
    return Math.floor(arr.length / 2)
}
console.log(majorityElement(arr));

```

## 原型、构造函数、实例

**原型** 一个简单的对象，用来实现对象的继承。可以简单理解成对象的爹。在js中每个对象都包含一个**proto**的属性指向它的爹也就是原型。**构造函数** 可以通过new来新建一个对象的函数 **实例** 通过通过函数和new创建出来的对象，便是实例。实例通过**proto**指向原型，通过**constructor**指向构造函数 实例.**proto** === 原型 构造函数.prototype === 原型 原型.constructor === 构造函数 实例.constructor === 构造函数



**原型链** 原型链是由原型对象组成，每个对象都有**proto**属性，指向了创建对象的构造函数的原型，**\_\_proto\_\_**将对象连接起来形成了原型链。是一个用来实现继承和共享属性的有限的对象链。

- 属性查找机制 当查找对象的属性时，如果实例对象自身不存在该属性，则沿着原型链往上一级查找，找到时则输出，不存在时，则继续沿着原型链往上一级查找，直至最顶层的原型对象 Object.prototype,如果还没有找到,则输出undefined.
- 属性修改机制 只会修改实例对象本身的属性，如果不存在，则进行添加该属性，如果需要修改原型的属性时，则可以用:b.prototype.x = 2;但是这样会造成所有继承于该对象的实例的属性发生改变。

## js

```
var x = 1;
if (function f() { }) { //把f 放到 () 中，变成表达式立即执行 变成true 然后消失
    x += typeof f
}
// console.log(x); //1undefined
```

## try catch

不会让try后面的代码终止，但是会让try里面的代码终止

```
try {
    console.log('a')
    console.log(b)
    console.log('c')
} catch (error) { //error.name error.message
    console.log('d')
    console.log(error)
    console.log(error.name);
}
console.log('e')
```

**Error.name**的六种值对应的信息 1、EvalError:eval()的使用与定义不一致 2、RangeError:数值越界 3、ReferenceError:非法或不能识别的引用数值 4、SyntaxError:发生语法解析错误 5、TypeError:操作数据类型错误 6、URIError:URL处理函数使用不当

## 克隆

## 浅克隆

- 传统

```
var obj = {
  name: 'xiaoming',
  age: 18,
  male: 'man',
  card: ['first', 'second']
}
var obj1 = {}

function clone(origin, target) {
  var target = target || {};
  for (var prop in origin) {
    target[prop] = origin[prop]
  }
  return target;
}

clone(obj, obj1)
console.log(obj1);
```

- Object.assign()
- 展开运算符(...)

深克隆 1、判断是不是原始类型 2、判断是数组还是对象 3、建立相应的数组或对象

- 传统 递归进行逐一赋值

```
var obj = {
  name: 'xiaoming',
  age: 18,
  male: 'man',
  card: ['first', 'second'],
  wife: {
    name: 'abc',
    son: {
      name: 'def',
    }
  }
}
var obj1 = {}

function deepClone(origin, target) {
  var target = target || {},
      toStr = Object.prototype.toString,
      arrStr = "[object Array]"
  for (var prop in origin) {
    if (origin.hasOwnProperty(prop)) {
```

```

        if (origin[prop] !== 'null' && typeof (origin[prop]) == 'object')
        {
            target[prop] = (toStr.call(origin[prop]) == arrStr ? [] : {});
            deepClone(origin[prop], target[prop]);
        } else {
            target[prop] = origin[prop];
        }
    }
}
return target
}

deepClone(obj, obj1)

```

- JSON.parse(JSON.stringify(obj)) 性能最快 具有循环引用的对象时报错 当为函数、undefined、symbol 无法拷贝

## js

```

var test = {
    bar: function () {
        return this.baz;
    },
    baz: 1
};
(function () {
    console.log(arguments[0]()); //bar() {} 这里打印看一下 this
    console.log(typeof arguments[0]()); //undefined
})(test.bar)

```

```

var x = [typeof x, typeof y][1]; //y undefined
console.log(typeof x); //string

```

```

function f() {
    return f; //如果是引用类型 return出去的就是实例
}
console.log(new f() instanceof f); //false

```

```

Object.prototype.a = '1';
Function.prototype.a = '2';
function Person() {};
var test = new Person();
console.log(test);

```

```

var test = [0];
if(test) {
    console.log(test);//[0]
    console.log(test == false);//true
}else{
    console.log('fail');
}

```

```

<script>
    test;//报错不往下执行
    console.log(1);
</script>
<script>
    console.log(2);//2
</script>

```

## js

```

console.log(a);
console.log(typeof yideng(a));
var flag = true;
if (!flag) {
    var a = 1;
}
if (flag) {
    function yideng(a) {
        yideng = a;
        console.log("yideng1");
    }
} else {
    function yideng(a) {
        yideng = a;
        console.log("yideng2");
    }
}

```

## 根据tbl\_id的值去重

```

let arr=[
    {tbl_id: 1, tbl_name: 'tbl1'},
    {tbl_id: 2, tbl_name: 'tbl2'},
    {tbl_id: 2, tbl_name: 'tbl2'},
    {tbl_id: 3, tbl_name: 'tbl3'}
];

const fn = (arr) => {

```

```
const obj = {};  
arr.forEach(item => {  
  if (!obj[item['tbl_id']]) {  
    obj[item['tbl_id']] = item  
  }  
});  
console.log(Object.values(obj));  
return Object.values(obj)  
}  
  
fn(arr)
```