# **2017前端面试题及答案总结|掘金技术征文**

作者： Yeaseon  
Blog：[yeaseonzhang.github.io](https://link.juejin.im/?target=https://yeaseonzhang.github.io" \t "https://juejin.im/post/_blank)  
[原文链接](https://link.juejin.im/?target=http://yeaseonzhang.github.io/2017/09/17/2017%E5%89%8D%E7%AB%AF%E9%9D%A2%E8%AF%95%E9%A2%98%E5%8F%8A%E7%AD%94%E6%A1%88%E6%80%BB%E7%BB%93/" \t "https://juejin.im/post/_blank)

"金三银四，金九银十"，用来形容求职最好的几个月。但是随着行业的饱和，初中级前端er就业形势不容乐观。

行业状态不可控，我们能做的当然只是让自己变得更加具有竞争力。

今年自己也用了几个月的时间来准备笔记面试，巩固基础知识。特此将自己在这个过程总结的题目分享出来，希望对于求职和准备求职的同学有所帮助。

## CSS

### 列举不同的清除浮动的技巧

/\* 1.添加新元素 \*/

<div class="outer">

<div class="div1"></div>

<div class="div2"></div>

<div class="div3"></div>

<div class="clearfix"></div>

</div>

.clearfix {

clear: both;

}

/\* 2.为父元素增加样式 \*/

.clearfix {

overflow: auto;

zoom: 1; // 处理兼容性

}

/\* 3.:after 伪元素方法 (作用于父元素) \*/

.outer {

zoom: 1;

&:after {

display: block;

height: 0;

clear: both;

content: '.';

visibillity: hidden;

}

}

### 一像素边框

使用sass语法。

/\* 定义 \*/

@mixin border-1px ($color) {

position: relative;

&:after {

display: block;

position: absolute;

left: 0;

bottom: 0;

width: 100%;

border-top: 1px solid $color;

context: '';

}

}

@media (-webkit-min-device-pixel-radio: 1.5), (min-device-pixel-radio: 1.5) {

border-1px {

&:after {

-webkit-transform: scaleY(0.7);

transform: scaleY(0.7);

}

}

}

@media (-webkit-min-device-pixel-radio: 2), (min-device-pixel-radio: 2) {

border-1px {

&:after {

-webkit-transform: scaleY(0.5);

transform: scaleY(0.5);

}

}

}

/\* 使用方式 \*/

@inclue border-1px(rgba(7, 17, 27, .1));

### 形成****BFC****(Block Formatting Context)的几种方式

BFC全称”Block Formatting Context”, 中文为“块级格式化上下文”。BFC元素特性表现原则就是，内部子元素再怎么翻江倒海，翻云覆雨都不会影响外部的元素。

* float为 left|right
* overflow为 hidden|auto|scroll
* display为 table-cell|table-caption|inline-block
* position为 absolute|fixed

### 布局

**[圣杯布局](https://link.juejin.im/?target=https://yeaseonzhang.github.io/2017/07/05/CSS%E5%B8%83%E5%B1%80/" \t "https://juejin.im/post/_blank)**[和](https://link.juejin.im/?target=https://yeaseonzhang.github.io/2017/07/05/CSS%E5%B8%83%E5%B1%80/" \t "https://juejin.im/post/_blank)**[双飞翼布局](https://link.juejin.im/?target=https://yeaseonzhang.github.io/2017/07/05/CSS%E5%B8%83%E5%B1%80/" \t "https://juejin.im/post/_blank)**

左定宽右自适应宽度，并且等高布局(最小高度200px)

/\* HTML \*/

<div class="container">

<div class="left">Left silder</div>

<div class="content">Main content</div>

</div>

/\* CSS \*/

.container {

overflow: hidden;

}

.left {

float: left；

width: 200px;

margin-bottom: -9999px;

padding-bottom: 9999px;

background-color: #eee;

}

.content {

margin-left: 200px;

margin-bottom: -9999px;

padding-bottom: 9999px;

background-color: #ccc;

}

.left, .content {

min-height: 200px;

height: auto !important;

}

## JS

### async与defer区别

异步(async) 脚本将在其加载完成后立即执行，而 延迟(defer) 脚本将等待 HTML 解析完成后，并按加载顺序执行。

### location.replace()与location.assign()区别

location.replace()的url不会出现在history中

### new操作符

* 创建一个空对象，并且this变量引用该对象，同时还继承了 该函数的原型
* 属性和方法被加入到this引用的对象中
* 新创建的对象由this所引用，并且最后隐式的返回this

### AMD CMD CommonJS

/\* AMD是RequireJS对模块化的定义

\* CMD是seaJS对模块化的定义

\* CommonJS是Node对模块化的规范

\*\*/

/\* AMD 依赖关系前置 \*/

define(['./a', './b'], **function** (a, b) {

a.something();

b.something();

})

/\* CMD 按需加载，依赖就近 \*/

define(**function** (require, exports, module) {

var a = require('./a');

a.something();

var b = require('./b');

b.something();

})

#### DOM 操作

// 创建节点

createDocumentFragment()

createElement()

createTextNode()

// 添加 移除 替换 插入

appendChild()

removeChild()

replaceChild()

insertBefore()

// 查找

getElementsByTagName()

getElementsByName()

getElementsByClassName()

getElementById()

querySelector()

querySelectorAll()

### JS设置css样式的几种方式

/\* 1.直接设置style属性 \*/

element.style.height = '100px';

/\* 2.直接设置属性 \*/

element.setAttribute('height', '100px');

/\* 3.使用setAttribute设置style属性 \*/

element.setAttribute('style', 'height: 100px !important');

/\* 4.使用setProperty设置属性，通过第三个参数设置important \*/

element.style.setProperty('height', '300px', 'important');

/\* 5.设置cssText \*/

element.style.cssText += 'height: 100px !important';

### 阻止默认行为

**function** stopDefault( e ) {

// 阻止默认浏览器动作(W3C)

**if** ( e && e.preventDefault ) {

e.preventDefault();

} **else** {

// IE中阻止函数器默认动作的方式

window.event.returnValue = false;

}

return false;

}

### 阻止冒泡

**function** stopBubble(e) {

// 如果提供了事件对象，则这是一个非IE浏览器

**if** ( e && e.stopPropagation ) {

// 因此它支持W3C的stopPropagation()方法

e.stopPropagation();

} **else** {

// 否则，我们需要使用IE的方式来取消事件冒泡

window.event.cancelBubble = true;

}

}

### Ajax交互过程

* 创建XMLHttpRequest对象,也就是创建一个异步调用对象.
* 创建一个新的HTTP请求,并指定该HTTP请求的方法、URL及验证信息.
* 设置响应HTTP请求状态变化的函数.
* 发送HTTP请求.
* 获取异步调用返回的数据.
* 使用JavaScript和DOM实现局部刷新.

### 考察知识点最广的JS面试题

**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); }

/\* 写出输出 \*/

Foo.getName();

getName();

Foo().getName();

getName();

new Foo.getName();

new Foo().getName();

new new Foo().getName();

具体讲解参见[一道常被人轻视的前端JS面试题](https://link.juejin.im/?target=http://www.cnblogs.com/xxcanghai/p/5189353.html" \t "https://juejin.im/post/_blank)

### JS数组深浅拷贝

* slice实现

var arr = ['old', 1, true, null, undefined];

var new\_arr = arr.slice();

new\_arr[0] = 'new';

console.log(arr) // ["old", 1, true, null, undefined]

console.log(new\_arr) // ["new", 1, true, null, undefined]

* concat实现

var arr = ['old', 1, true, null, undefined];

var new\_arr = arr.concat();

new\_arr[0] = 'new';

console.log(arr) // ["old", 1, true, null, undefined]

console.log(new\_arr) // ["new", 1, true, null, undefined]

以上两种方法只是浅拷贝，如果数组元素是基本类型，就会拷贝一份新的；但是如果数组元素是对象或者数组，就只会拷贝引用（类似指针），修改其中一个就会影响另外一个。

var arr = ['old', 1, true, ['old1', 'old2'], {old: 1}];

var new\_arr = arr.concat();

new\_arr[0] = 'new';

new\_arr[3][0] = 'new1';

console.log(arr) // ["old", 1, true, ['new1', 'old2'], {old: 1}]

console.log(new\_arr) // ["new", 1, true, ['new1', 'old2'], {old: 1}]

* JSON.stringify实现数组深拷贝

var arr = ['old', 1, true, ['old1', 'old2'], {old: 1}];

var new\_arr = JSON.parse(JSON.stringify(arr));

new\_arr[0] = 'new';

new\_arr[3][0] = 'new1';

console.log(arr) // ["old", 1, true, ['old1', 'old2'], {old: 1}]

console.log(new\_arr) // ["new", 1, true, ['new1', 'old2'], {old: 1}]

简单粗暴，但是问题是不能拷贝函数，不推荐。

然后我们来手动实现深浅拷贝。

* 浅拷贝

var shallowCopy = **function** (obj) {

// 判断是否是数组或者对象

**if** (typeof obj !== 'object') {

return

}

var newObj = obj instanceof Array ? [] : {};

**for** (var key **in** obj) {

**if** (obj.hasOwnProperty(key)) {

newObj[key] = obj[key];

}

}

return newObj;

}

* 深拷贝

var deepCopy = **function** (obj) {

**if** (typeof obj !== 'object') {

return

}

var newObj = obj instanceof Array ? [] : {};

**for** (var key **in** obj) {

**if** (obj.hasOwnProperty(key)) {

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

}

}

return newObj

}

### 数组去重

* filter + indexOf

**function** unique (arr) {

var res = arr.filter(**function** (item, index, array) {

return array.indexOf(item) === index;

})

return res;

}

* filter + sort

**function** unique (arr) {

return arr.concat().sort().filter(**function** (item, index, array) {

return !index || item !== array[index - 1];

})

}

* ES6

**function** uniqu3 (arr) {

return [... new Set(arr)];

}

### 找出数组中的最大值

* reduce

var arr = [6, 4, 1, 8, 2, 11, 3];

**function** max (prev, next) {

return Math.max(prev, next)

}

console.log(arr.reduce(max));

* apply

var arr = [6, 4, 1, 8, 2, 11, 3];

console.log(Math.max.apply(null, arr));

* ES6

var arr = [6, 4, 1, 8, 2, 11, 3];

**function** max (arr) {

return Math.max(...arr);

}

console.log(max(arr));

### 打乱数组的方法

var arr = [];**for** (var i = 0; i < 100; i++) {

arr[i] = i;

}

arr.sort(**function** () {

return 0.5 - Math.random();

});

### 数组扁平化

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

**function** flatten(arr) {

**while** (arr.some(item => Array.isArray(item))) {

arr = [].concat(...arr);

}

return arr;

}

console.log(flatten(arr))

### 排序

// 冒泡**function** bubbleSort2(arr) {

var len = arr.length;

**for** (var i = 0; i <= len - 1; i++) {

**for** (var j = 0; j <= len - i; j++) {

**if** (arr[j + 1] < arr[j]) {

var temp;

temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

return arr;

}

// 快速排序**function** quickSort(arr) {

**if**(arr.length == 0) {

return []; // 返回空数组

}

var cIndex = Math.floor(arr.length / 2);

var c = arr.splice(cIndex, 1);

var l = [];

var r = [];

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

**if**(arr[i] < c) {

l.push(arr[i]);

} **else** {

r.push(arr[i]);

}

}

return quickSort(l).concat(c, quickSort(r));

}

### 数字格式化 1234567890 -> 1,234,567,890

**function** formatNum (num) {

return num.replace(/\B(?=(\d{3})+(?!\d))/g, ',');

}

var num = '1234567890';

var res = formatNum(num);

console.log(res);

### 打乱数组的方法

var arr = [];**for** (var i = 0; i < 100; i++) {

arr[i] = i;

}

arr.sort(**function** () {

return 0.5 - Math.random();

})

### 尾调用优化

即只保存内层函数的调用帧(只有开启严格模式，才会生效),只有不再用到外层函数的内部变量，内层函数的调用帧才会取代外层函数的调用帧，否则无法进行“尾调用优化。

**function** factorial(n) {

**if** (n === 1) return 1;

return n \* factorial(n-1);

}

factorial(5)

/\* 优化尾递归 \*/**function** factorial(n, total) {

**if** (n === 1) return total;

return factorial(n - 1, n \* total);

}

factorial(5, 1)

### 柯里化

实现add(1,2)和add(1)(2)均输出3

**function** **add** () {

let sum = 0;

Array.prototype.forEach.call(arguments, **function** (item, index){

**if** (typeof item !== 'number') {

return false;

} **else** {

sum += item;

}

})

var tmp = **function** () {

Array.prototype.forEach.call(arguments, **function** (item, index){

**if** (typeof item !== 'number') {

return false;

} **else** {

sum += item;

}

})

return tmp;

}

tmp.toString = **function** () {

return sum

}

return tmp;

}

add(1, 2); // 3

add(1)(2); // 3

add(1, 2, 3)(1, 4)(2, 2)(1) // 16

### ES8 新特性

* 字符串填充

str.padStart(targetLength [, padString])

str.padEnd(targetLength [, padString])

* values和entries函数

Object.values(obj)

Object.entries(obj)

* getOwnPropertyDescriptors函数

Object.getOwnPropertyDescriptors(obj)

* 函数参数逗号结尾

**function** es8(var1, var2, var3,) {}

* 异步函数  
  由async关键词定义的函数声明了一个可以异步执行的函数，返回一个AsyncFunction类型的对象。

fucntion **fetchTextByPromise** () {

return new Promise(resolve => {

setTimeout(() => {

resolve('es8');

}, 2000);

});

}

async **function** **sayHello** () {

const externalFetchedText = await fetchTextByPromise();

console.log(`Hello, ${externalFetchedText}`);

}

sayHello();

### 数据类型判断

var class2type = {};

'Boolean Number String Function Array Date RegExp Object Error Null Undefined'.split(' ').map((item, index) => {

class2type['[object ' + item + ']'] = item.toLowerCase();

})

**function** type (obj) {

return typeof obj === 'object' || typeof obj === 'function' ?

class2type[{}.toString.call(obj)] || 'object' : typeof obj;

}

### 防抖

/\*

\* func：需要调用的函数

\* wait: 防抖时间

\* immediate：布尔值，是否立即执行

\*\*/

var debounce = **function** (func, wait, immediate) {

var timeout;

return **function** () {

var context = this;

var args = arguments;

**if** (timeout) clearTimeout(timeout);

**if** (immediate) { // 是否立即执行func

var callNow = !timeout;

timeout = setTimeout(**function** () {

timeout = null;

}, wait)；

**if** (callNow) {

func.apply(context, args);

}

} **else** {

timeout = setTimeout(**function** () {

func.apply(context, args);

}, wait);

}

}

}

### 简单的字符串模板

var TemplateEngine = **function**(tpl, data) {

var re = /<%([^%>]+)?%>/g, match;

**while**(match = re.exec(tpl)) {

tpl = tpl.replace(match[0], data[match[1]])

}

return tpl;

}

var template = '<p>Hello, my name is <%name%>. I\'m <%age%> years old.</p>';

console.log(TemplateEngine(template, {

name: "Yeaseon",

age: 24

}));

// '<p>Hello, my name is Yeaseon. I\'m 24 years old.</p>';

### apply、call和bind

* apply和call

在严格模式下，未指定环境对象而调用函数，则this 值不会转型为window。除非明确把函数添加到某个对象或者调用apply()或call()，否则this 值将是undefined。

在非严格模式下，call、apply的第一个参数传递为null或undefined时，函数体内的this会指向默认的宿主对象，在浏览器中则是window。

* apply、call和bind比较

var obj = {

x: 81

};

var foo = {

getX: **function** () {

return this.x;

}

}

console.log(foo.getX.bind(obj)());

console.log(foo.getX.apply(obj));

console.log(foo.getX.call(obj));

很明显，bind方法后面多了一对括号。也就是说，当你希望改变上下文环境之后并且立即执行，而是回调执行的时候（多用于事件监听器函数），使用bind()方法，而apply/call则会立即执行函数。

* 定义一个 log 方法，让它可以代理 console.log 方法。

**function** **log**(){

console.log.apply(console, arguments);

};log(1); //1log(1,2); //1 2

* 给每一个 log 消息添加一个"(app)"的前辍，比如：

log("hello world"); //(app)hello world

**function** **log**(){

var args = Array.prototype.slice.call(arguments);

args.unshift('(app)');

console.log.apply(console, args);

};

* apply实现bind

**function** bind (fn, context) {

return **function** () {

return fn.apply(context, argument);

}

}

### 创建对象

* 工厂模式

**function** createPerson(name, age, job){

var o = new Object();

o.name = name;

o.age = age;

o.job = job;

o.sayName = **function**(){

alert(this.name);

};

return o;

}

var person1 = createPerson("Nicholas", 29, "Software Engineer");

var person2 = createPerson("Greg", 27, "Doctor");

* 构造函数模式

**function** Person(name, age, job){

this.name = name;

this.age = age;

this.job = job;

this.sayName = **function**(){

alert(this.name);

};

}

var person1 = new Person("Nicholas", 29, "Software Engineer");

var person2 = new Person("Greg", 27, "Doctor");

* 原型模式

**function** **Person**(){

}

Person.prototype.name = "Nicholas";

Person.prototype.age = 29;

Person.prototype.job = "Software Engineer";

Person.prototype.sayName = **function**(){

alert(this.name);

};

var person1 = new Person();

person1.sayName(); //"Nicholas"

* 构造函数 + 原型模式

**function** Person(name, age, job){

this.name = name;

this.age = age;

this.job = job;

this.friends = ["Shelby", "Court"];

}

Person.prototype = {

constructor : Person,

sayName : **function**(){

alert(this.name);

}

}

var person1 = new Person("Nicholas", 29, "Software Engineer");

var person2 = new Person("Greg", 27, "Doctor");

person1.friends.push("Van");

alert(person1.friends); //"Shelby,Count,Van"

alert(person2.friends); //"Shelby,Count"

alert(person1.friends === person2.friends); //false

alert(person1.sayName === person2.sayName); //true

### JS实现Jquery的extend()方法

**function** **extend**() {

// 默认不进行深拷贝

var deep = false;

var name, options, src, copy, clone, copyIsArray;

var length = arguments.length;

// 记录要复制的对象的下标

var i = 1;

// 第一个参数不传布尔值的情况下，target 默认是第一个参数

var target = arguments[0] || {};

// 如果第一个参数是布尔值，第二个参数是 target

**if** (typeof target == 'boolean') {

deep = target;

target = arguments[i] || {};

i++;

}

// 如果target不是对象，我们是无法进行复制的，所以设为 {}

**if** (typeof target !== "object" && !isFunction(target)) {

target = {};

}

// 循环遍历要复制的对象们

**for** (; i < length; i++) {

// 获取当前对象

options = arguments[i];

// 要求不能为空 避免 extend(a,,b) 这种情况

**if** (options != null) {

**for** (name **in** options) {

// 目标属性值

src = target[name];

// 要复制的对象的属性值

copy = options[name];

// 解决循环引用

**if** (target === copy) {

continue;

}

// 要递归的对象必须是 plainObject 或者数组

**if** (deep && copy && (isPlainObject(copy) ||

(copyIsArray = Array.isArray(copy)))) {

// 要复制的对象属性值类型需要与目标属性值相同

**if** (copyIsArray) {

copyIsArray = false;

clone = src && Array.isArray(src) ? src : [];

} **else** {

clone = src && isPlainObject(src) ? src : {};

}

target[name] = extend(deep, clone, copy);

} **else** **if** (copy !== undefined) {

target[name] = copy;

}

}

}

}

return target;

};

### 自定义事件（通过观察者模式）

**function** **EventTarget** () {

this.handlers = {};

}

EventTarget.prototype = {

constructor: EventTarget,

addHandler: **function** (type, handler) {

**if** (typeof this.handlers[type] == 'undefined') {

this.handlers[type] = [];

}

this.handlers[type].push(handler)

},

fire: **function** (event) {

**if** (!event.target) {

event.target = this;

}

**if** (this.handlers[event.type] instanceof Array) {

var handlers = this.handlers[event.type];

**for** (var i = 0, len = handlers.length; i < len; i++) {

handlers[i](event);

}

}

},

removeHandler: **function** (type, handler) {

**if** (this.handlers[type] instanceof Array) {

var handlers = this.handlers[type];

**for** (var i = 0, len = handlers.length; i < len; i++) {

**if** (handlers[i] === handler) {

break;

}

}

handlers.splice(i, 1);

}

}

}

## 安全

### 跨域的几种方法

1. 主域相同的跨域

document.domain

window.postMessage

JSONP跨域(只支持GET)

**function** todo(data){

console.log('The author is: '+ data.name);

}

var script = document.createElement('script');

/\* 向服务器www.yeaseonzhang.com发出请求。注意，该请求的查询字符串有一个callback参数，用来指定回调函数的名字。 \*/

script.src = 'http://www.yeaseonzhang.com/author?callback=todo';

document.body.appendChild(script);

/\* 服务器收到这个请求以后，会将数据放在回调函数的参数位置返回。 \*/

todo({"name": "fewjq"});

/\* 由于<script>元素请求的脚本，直接作为代码运行。这时，只要浏览器定义了todo函数，该函数就会立即调用。作为参数的JSON数据被视为JavaScript对象。\*/

1. websocket跨域

### XSS 和 CSRF

* [XSS](https://link.juejin.im/?target=http://blog.csdn.net/ghsau/article/details/17027893" \t "https://juejin.im/post/_blank)
* [CSRF](https://link.juejin.im/?target=http://www.cnblogs.com/hyddd/archive/2009/04/09/1432744.html" \t "https://juejin.im/post/_blank)

## 性能

### CSS 优化

* 正确的时机调用CSS
* 使用媒体查询标记<link>，选择性加载
* 减少css文件数量
* 压缩css代码

### 渲染

* [浏览器渲染原理](https://link.juejin.im/?target=http://coolshell.cn/articles/9666.html" \t "https://juejin.im/post/_blank)  
  [掘金秋招征文大赛](https://link.juejin.im/?target=https://juejin.im/post/59af60acf265da24701ee7da" \t "https://juejin.im/post/_blank)

**关注下面的标签，发现更多相似文章**