前端JS笔试面试题

前端JS笔试面试题

笔试部分

Q： 下面有关javascript内部对象的描述，正确的有？（多选）

Q：以下哪个是错误的（单选）

Q： 写出以下程序的输出结果，为什么？

Q：有完成以下代码， 使其输出结果与注释中的一样（忽略时区信息）

Q：以下程序输出什么，为什么？

Q：请写出以下程序的输出结果

Q：在 javascript 中，用于阻止默认事件的默认操作的方法是？

Q：在标准的 JavaScript 中， Ajax 异步执行调用基于下面哪一项机制才能实现(单选)，并写出选择该项的理由

Q：请写出可以匹配以下16进制颜色值中任意一个的正则表达式

Q：请写出正则表达式，匹配帐号是否合法(字母开头，允许5-16字节，允许字母数字下划线)

Q：请输出以下程序的执行结果

Q：请写出以下程序的执行结果

Q：请写出以下程序的输出结果

Q：以下程序输出什么，为什么？

Q：请写出以下程序的执行结果，为什么？

Q：请在下方列出var与let的区别

Q：假设当前网址是 `http://foouser:barpassword@www.wrox.com:80/WileyCDA/?q=javascript#contents` 请写出以下属性的值：

Q：请写一个方法来判断一个任意值是否为文本节点

Q 请写一个方法来判断一个对象是否为可迭代对象

Q： 请使用正则写一个可以过滤文本中所有的 `script` 标签的函数

Q：请写出以下程序的输出结果

Q： 请编写一个方法，实现给出任意一个数组，可以去重后返回

Q： 请完成 `flatten` 方法

Q： 请实现一个可兼容所有浏览器的DOM事件监听函数

Q：请完成以下程序：

Q：请实现 `getCommonAncestor` 方法，返回任意两个DOM节点的最近的公共祖先节点

Q：请实现 `Person` 方法，使其可以通过如下方式被调用

Q 请使用高阶函数分贝完成以下需求

Q： 列出JS中常见的错误类型，其中哪些是早期错误，并写一个示例程序，抛出一个错误并捕获

Q： 写出以下程序的执行结果

Q： `<script>` 标签的 `async` 属性与 `defer` 属性之间的异同

Q：请写出以下程序的执行结果，为什么？

Q：请写出以下程序的执行结果，为什么

Q： 请完成 `parseQuery` 方法，要求必须使用正则

Q：请完成 `concatWithSort` 方法：

Q：以下代码，请使用至少两种不同方法实现 `a` 与 `b` 的值互换，要求不使用临时变量，并给出解题思路

面试部分

Q：请简述前后端分离架构的优缺点

Q： 请简述前端都有哪些安全问题，怎么解决

Q： 浏览器缓存读取规则

Q 简述浏览器的工作流程

Q： **cookie 和 token 都存放在 header 中，为什么不会劫持 token？**

Q：介绍模块化发展历程

笔试部分

Q： 下面有关javascript内部对象的描述，正确的有？（多选）

1. History 对象包含用户（在浏览器窗口中）访问过的 URL
2. Location 对象包含有关当前 URL 的信息
3. Window 对象表示浏览器中打开的窗口
4. Navigator 对象包含有关浏览器的信息

> 1、2、3、4
>
> 考察： JS基础

Q：以下哪个是错误的（单选）

1. iframe是用来在网页中插入第三方页面，早期的页面使用iframe主要是用于导航栏这种很多页面都相同的部分，这样在切换页面的时候避免重复下载
2. iframe的创建比一般的DOM元素慢了1-2个数量级
3. iframe标签会阻塞页面的的加载
4. iframe本质是动态语言的Incude机制和利用ajax动态填充内容

4

解析：关于IFRAME的局限：

1. 创建比一般的 DOM 元素慢了 1-2 个数量级

iframe 的创建比其它包括 scripts 和 css 的 DOM 元素的创建慢了 1-2 个数量级，使用 iframe 的页面一般不会包含太多 iframe，所以创建 DOM 节点所花费的时间不会占很大的比重。但带来一些其它的问题：onload 事件以及连接池（connection pool）

2、阻塞页面加载

及时触发 window 的 onload 事件是非常重要的。onload 事件触发使浏览器的 “忙” 指示器停止，告诉用户当前网页已经加载完毕。当 onload 事件加载延迟后，它给用户的感觉就是这个网页非常慢。

window 的 onload 事件需要在所有 iframe 加载完毕后（包含里面的元素）才会触发。在 Safari 和 Chrome 里，通过 JavaScript 动态设置 iframe 的 SRC 可以避免这种阻塞情况

3、唯一的连接池

浏览器只能开少量的连接到 web 服务器。比较老的浏览器，包含 Internet Explorer 6 & 7 和 Firefox 2，只能对一个域名（hostname）同时打开两个连接。这个数量的限制在新版本的浏览器中有所提高。Safari 3+ 和 Opera 9+ 可同时对一个域名打开 4 个连接，Chrome 1+, IE 8 以及 Firefox 3 可以同时打开 6 个

绝大部分浏览器，主页面和其中的 iframe 是共享这些连接的。这意味着 iframe 在加载资源时可能用光了所有的可用连接，从而阻塞了主页面资源的加载。如果 iframe 中的内容比主页面的内容更重要，这当然是很好的。但通常情况下，iframe 里的内容是没有主页面的内容重要的。这时 iframe 中用光了可用的连接就是不值得的了。一种解决办法是，在主页面上重要的元素加载完毕后，再动态设置 iframe 的 SRC。

4、不利于 SEO

搜索引擎的检索程序无法解读 iframe。另外，iframe 本身不是动态语言，样式和脚本都需要额外导入。综上，iframe 应谨慎使用。

Q： 写出以下程序的输出结果，为什么？

```
(function() {

    var x=foo();

    var foo=function foo() {
        return "foobar"
    };
    return x;
})();
```

类型错误

原因： 函数表达式和变量声明一样，只有定义提升，赋值并不会提升，当执行 `var x = foo()` 时， `foo` 的值还是 `undefined` ，所以它不是一个函数，当做函数调用时会报类型错误。

考察： JS基础-变量提升

Q：有完成以下代码， 使其输出结果与注释中的一样（忽略时区信息）

```
var d = new Date('2018-05-09');
//在下面编写代码
d.setDate(40);

console.log(d)//   Sat Jun 09 2018 xxxxx (xxxx)

```

参考实现1：

```
d.setMonth(5);
```

参考实现2：

```
d.setDate(40);
```

考察： 日期相关方法使用。

Q：以下程序输出什么，为什么？

```
function Foo(){
  'use strict'
  console.log(this.location);
}
Foo();
```

> 类型错误（TypeError）
>
> 原因： 严格模式下this禁止指向全局对象
>
> 考察：严格模式

Q：请写出以下程序的输出结果

```
console.log(typeof Date.now());
```

> number
>
> 考察： JS基础-数据类型判断

Q：在 javascript 中，用于阻止默认事件的默认操作的方法是？

```
preventDefault()
```

> 考察： JS基础-事件处理

Q：在标准的 JavaScript 中， Ajax 异步执行调用基于下面哪一项机制才能实现(单选)，并写出选择该项的理由

1. Event和callback
2. 多线程操作
3. 多CPU核
4. Deferral和promise

> 1
>
> 理由：

JavaScript是单线程的，浏览器通过事件循环实现了异步的操作，所以js程序是事件驱动的，每个事件都会绑定相应的回调函数

考察： JS异步-事件循环

Q：请写出可以匹配以下16进制颜色值中任意一个的正则表达式

```
#ffbbad
#Fc01DF
#FFF
#ffE
```

参考实现：

```
/#([0-9a-fA-F]{6}|[0-9a-fA-F]{3})/g
```

考察：JS基础-正则表达式基础

Q：请写出正则表达式，匹配帐号是否合法(字母开头，允许5-16字节，允许字母数字下划线)

```
/^[a-zA-Z][a-zA-Z0-9_]{4,15}$/
```

考察： JS正则表达式应用

Q：请输出以下程序的执行结果

```
console.log(Function.prototype.__proto__.__proto__ === null);
```

true

考察：JS原型链

Q：请写出以下程序的执行结果

```
var a =[1,2,3];
var b = a.slice(0,1);
b.push(4);
console.log(a, b)
```

[1,2,3] [1, 4]

考察： 数组为引用类型；slice方法不会改变原数组，返回的是截取到的新数组

Q：请写出以下程序的输出结果

```
var a = 10;
var obj = {
  a: 20,
  fn: function(a){
    this.a += 5;
    let b = 3;
    function fn(a) {
      this.a += 2;
      a += b;
      console.log(this.a + a);
    };
    return fn;
  }
};

var fn = obj.fn(a);
fn(3);
fn(5);
fn(obj.a);
```

输出： 18 22 44

考察： 1. 闭包； 2. 局部作用域与全局作用域； 3. this指向

Q：以下程序输出什么，为什么？

```
var tmp = new Date();

function f() {
  console.log(tmp);
  if (false) {
    var tmp = "hello world";
    console.log(tmp);
  }
  console.log(tmp);
}

f();
```

输出：undefined undefiend

1. 因为函数f作用域中的局部变量 `tmp` 覆盖了外层全局作用域的全局变量 `tmp` ;

2. 条件声明（if）并不影响变量提升
3. 由于变量提升，所以第一个 `console` 输出 `tmp` 的时候它的值还未初始化，所以输出 undefined
4. 由于条件声明条件为假，所以 `tmp = 'hello word'` 与 第二个 `consile` 语句都不会执行；
5. 由于赋值语句没有执行，所以第三个 `console` 输出的还是 `undefined`

考察：JS基础

Q：请写出以下程序的执行结果，为什么？

```
a = 1;
console.log(window.a);

let b = 1;
console.log(window.b);
```

1 undefiend1

原因：

- a是全局变量，全局变量与window对象存在映射，执行 `a=1` 时，同时会给window增加一个属性 `a` ，赋值为1；
- let声明的是块级作用域，与window对象不存在映射关系，所以并不会给window添加属性，所以访问 `window.b` 为 `undefiend` ;

考察：JS基础

Q：请在下方列出var与let的区别

- ES6可以用let定义块级作用域变量，var不能
- let存在暂时死区，而var没有
- let没有变量提升，而var有；
- let变量不能重复声明，而var 可以
- 全局作用域中，var声明的对象会映射到window对象属性上，let声明的不会

考察： JS基础：变量定义

Q：假设当前网址是 `http://foouser:barpassword@www.wrox.com:80/WileyCDA/?q=javascript#contents` 请写出以下属性的值：

- `localtion.hash`
- `location.search`
- `location.pathname`
- `location.origin`
- `location.username`

> 答案：
>
> - `location.hash` : `#contents`
> - `location.search` : `?q=javascript`
> - `location.pathname` : `/wileyCDA`
> - `location.origin` : `http://www.wrox.com`
> - `location.username` : `foouser`

Q：请写一个方法来判断一个任意值是否为文本节点

> 参考实现：
>
> ```
> var isTextNode = function (val) {
> return  val.nodeType === "3";
> }
> ```
>
> 考察： DOM节点类型 nodeType

Q 请写一个方法来判断一个对象是否为可迭代对象

> 参考实现：
>
> ```
> var isIterablle = function (val) {
>  if(val == null){
>      return false;
>  }
>  return typeof val[Symbol.iterator] === 'function';
> }
> ```
>
> 考察： ES6的Symbol相关知识

Q： 请使用正则写一个可以过滤文本中所有的script标签的函数

参考实现：

```
var stripScripts = function (val){
return val.replace(new RegExp('<script[^>]*>([\\S\\s]*?)<\/script\\s*>', 'img'),
'');

}
```

考察： 正则表达式

Q： 请写出以下程序的输出结果

```
let [foo, , , ,[[bar], ,baz]] = [1,2,3,4,[['a','b'], 'e', {a:1}]];
console.log(foo);
console.log(bar);
console.log(baz);
```

1 a {a:1}

考察： ES6数组结构赋值

Q： 请编写一个方法，实现给出任意一个数组，可以去重后返回

参考实现一：

```
let a = [1,1,2,3,4,5,5,6];

function arrayNoRepeat(arr) {
let newArr = [];
for(let i=0;i<arr.length;i++) {
if (newArr.indexOf(arr[i]) === -1) {
 newArr.push(arr[i])
}
}
return newArr;
}

a = arrayNoRepeat(a);
```

参考实现二：

```
let a = [1,1,2,3,4,5,5,6];

function arrayNoRepeat(arr) {
return Array.from(new Set(arr));
}

a = arrayNoRepeat(a);
```

Q： 请完成 `flatten` 方法

```
function flatten(arr){

}
var arr = flatten([1,2,[3,4,[5,6,[7,[8]]]]]);
console.log(arr);
// 输出：  [ 1, 2, 3, 4, 5, 6, 7, 8 ]
```

参考实现：

```
function flatten(arr){
var newArray = [];
for(var i=0;i<arr.length;i++){
if(arr[i] instanceof Array){
 newArray = newArray.concat(flatten(arr[i]));
}else{
 newArray.push(arr[i]);
}
}
return newArray;
}

var arr = flatten([1,2,[3,4,[5,6,[7,[8]]]]]);
console.log(arr);

```

考察： 数组类型判断； 递归

Q： 请实现一个可兼容所有浏览器的**DOM**事件监听函数

参考实现：

```
function addListener(element, type, handler){
if (element.addEventListener) {
  element.addEventListener(type, handler, false);
} else if (element.attachEvent) {
  element.attachEvent("on" + type, handler);
} else {
  element["on" + type] = handler;
}
}
```

> 考察： DOM事件监听

Q：请完成以下程序：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="initial-scale=1, maximum-scale=3, minimum-scale=1, user-scalable=no">
    <title>canvas存为图片</title>
    <script>
        window.onload = function() {
            draw();
            var dlButton = document.getElementById("downloadImageBtn");
            bindButtonEvent(dlButton, "click", saveAsLocalImage);
        };
        function draw(){
            var canvas = document.getElementById("thecanvas");
            var ctx = canvas.getContext("2d");
            // 绘制一个边长为100的正方形，并填充为透明度50%的黑色,位置随意


            ctx.fillText("canvas图片下载", 50, 50);
        }

        function bindButtonEvent(element, type, handler)
        {
            if(element.addEventListener) {
                element.addEventListener(type, handler, false);
            } else {
                element.attachEvent('on'+type, handler);
            }
        }

        function saveAsLocalImage () {
            var myCanvas = document.getElementById("thecanvas");
            // 完成该方法，使其可以将canvas内容下载到本地
        }
    </script>
</head>
<body bgcolor="#E6E6FA">
<div>
    <canvas width=200 height=200 id="thecanvas"></canvas>
    <button id="downloadImageBtn">下载为图片</button>
</div>
</body>
</html>
```

> 参考实现：

```
function draw(){
       var canvas = document.getElementById("thecanvas");
       var ctx = canvas.getContext("2d");
       ctx.fillStyle = "rgba( 0, 0, 0, .5)";
       ctx.fillRect(25,25,100,100);
   }

function saveAsLocalImage () {
       var myCanvas = document.getElementById("thecanvas");
       var image = myCanvas.toDataURL("image/png").replace("image/png",
"image/octet-stream");
       window.location.href=image;
   }
```

考察： canvas相关；

Q：请实现getCommonAncestor方法，返回任意两个DOM节点的最近的公共祖先节点

```
function getCommonAncestor (nodeA, nodeB) {

}
```

参考实现：

```
function getCommonAncestor (nodeA, nodeB) {
   if (nodeA === nodeB)
       return nodeA;
   var parentsA = [nodeA] , parentsB = [nodeB], parent = nodeA, i = -1;
   while (parent = parent.parentNode) {
       if (parent === nodeB) {
           return parent;
       }
       parentsA.push(parent);
   }
   parent = nodeB;
   while (parent = parent.parentNode) {
       if (parent === nodeA)
           return parent;
       parentsB.push(parent);
   }
   parentsA.reverse();
   parentsB.reverse();
   while (i++, parentsA[i] === parentsB[i]) {
   }
   return i == 0 ? null : parentsA[i - 1];

}
```

考察：1. DOM基础； 2. 复杂问题的算法设计

基本思路： 如果两个节点是同一个，则直接返回该节点。如果不同，则将两个节点的所有祖先节点分别存入两个数组，然后数组反转后逐个对比相同索引位置的机电，如果相同，则返回该节点。

Q：请实现Person方法，使其可以通过如下方式被调用

```
var xiaoming  = Person('xiaoming', 23);
xiaoming.name().age();
// 输出:
// xiaoming
// 23
```

参考实现一：简单版

```
var Person = function(pName,pAge){
//构造函数
return {
    personName: pName,
    personAge: pAge,
    name: function(){
      console.log(this.personName);
      return this;
    },
    age: function() {
      console.log(this.personAge);
      return this;
    }
}
}
var xiaoming  = Person('xiaoming', 23);
xiaoming.name().age();
```

参考实现二：ES5高级版

```
var Person = function(name,age){
//构造函数
return  new Person.prototype.init(name,age);
}

Person.prototype = {
// 工厂方法
init:function(name,age){
    this.personName = name;
    this.personAge = age;
    return this; // 这里的this是Person.prototype自身的一个引用，而非copy
},
//原型
name: function(){console.log(this.personName); return this;},
age: function(){console.log(this.personAge); return this;}
}

```

```
Person.prototype.init.prototype = Person.prototype;
var xiaoming  = Person('xiaoming', 23);
xiaoming.name().age();
```

参考实现三： ES6版

```
class PersonClass {
  constructor(name,age) {
    this.personName = name;
    this.personAge = age;
  }

  name() {
    console.log(this.personName);
    return this;
  }
  age() {
    console.log(this.personAge);
    return this;
  }
}

function Person(name,age){
  return new PersonClass(name,age);
}
var xiaoming  = Person('xiaoming', 23);
xiaoming.name().age();
```

考察： 面向对象基础

Q 请使用高阶函数分贝完成以下需求

```
var arr = [
  {
    id:1,
    name: '张三',
    score: 80
  },
  {
    id:2,
    name: '李四',
    score: 60
  },
  {
    id:3,
    name: '王五',
    score: 75
  },
  {
    id:4,
    name: '陈六',
    score: 80
  },
  {
```

```
    id:5,
    name: '鬼脚七',
    score: 80
  },
];


```

1. 返回一个字符串，字符串中是所有score为80的人的ID和名称，如 `1-张三` ，用逗号分隔。
2. 计算所有人得分的总和

> 参考实现：
>
> ```
> // 返回一个字符串，字符串中是所有score为80的人的名字，用逗号分隔。
>
> var names = arr.filter(item=>item.score === 80).map(item => item.id + '-' + item.name).join(',');
>
> //计算所有人得分的总和
> var totalScores = arr.map(item => item.score).reduce((total, score) => total + score);
> ```
>
> 考察：高阶函数的掌握情况

Q： 列出JS中常见的错误类型，其中哪些是早期错误，并写一个示例程序，抛出一个错误并捕获

> 错误类型：(至少能答出前三个)
>
> - ReferenceError
> - SyntaxError
> - TypeErrorError
> - InternalError
> - EvalError
> - RangeError
> - URIError
>
> 大多数 `SyntaxError` 为早期错误，是指程序运行前进行词法分析和语法分析时抛出的错误，其它都是运行时错误。
>
> 抛出/捕获错误示例：
>
> ```
> var Car = function(){}
> Car.prototype = {
>     getPrice:function(){
>         throw new Error('抽象方法不能调用');
>     },
> ```

```
    getSpeed:function(){
        throw new Error('抽象方法不能调用');
    }
}
function Bus(name,price,speed){
    this.name = name;
    var price = price
    this.getPrice = function(){
        return price;
    }
}
Bus.prototype = new Car()

var bus = new Bus("公交车","30万","120公里");
try {
  console.log(bus.getPrice()); // 30万
  var speed = bus.getSpeed();
  console.log('车的速度是: ', speed);
} catch(e) {
  console.log(e.message); // 抽象方法不能调用
}

```

考察： 错误处理

Q： 写出以下程序的执行结果

```
async function async1() {
  console.log('async1 start');
  await async2();
  console.log('async1 end');
}
async function async2() {
  console.log('async2');
}
console.log('script start');
setTimeout(function() {
  console.log('setTimeout');
}, 0)
async1();
new Promise(function(resolve) {
  console.log('promise1');
  resolve();
}).then(function() {
  console.log('promise2');
});
console.log('script end');
```

输出结果：

```
script start
async1 start
async2;
promise1;
script end
async1 end;
promise2
setTimeout
```

考察： js中的异步： `setTimeout`，`promise`， `async/await`，微任务与宏任务

```
1. 同步代码先执行，执行完会执行该同步任务的异步微任务，执行完后再执行异步的宏任务，所以代码会跳过前面的函数定义，先执行第9行，输出`script start`
2. `setTimeout` 执行，在事件队列末尾加一个异步的宏任务
3. 在没有 await 的情况下执行` async `函数，它会立即执行，返回一个`Promise` 对象，并且，绝不会阻塞后面的语句。所以此处执行`async1`，输出`async1 start`，
4. 此时在`async1`执行上下文中，执行 `await async2`，await 等待的是一个表达式，这个表达式的计算结果是` Promise` 对象或者其它值，这里表达式是一个带有`async`标志的函数，它的计算结果是一个`promise`，而`promise`会在定义时立即执行，所以输出`async2`，由于`await `会造成阻塞，所以程序会先执行`async1`外面的同步代码，执行完同步代码后才会继续执行`await`后面的代码；
5. 在全局上下文执行`new Promise()` 函数，`promise`被定义时立即执行，输出`promise1`， 但它的`resolve`是异步的，所以`then`方法里的函数被作为异步微任务加入到全局上下文宏任务的微任务队列中。
6. 同步任务继续执行，输出`script end`；
7. 同步任务执行完毕，回到之前因`await`而阻塞的地方，输出`async1 end`， 然后查看异步微任务队列，输出`promise2`；
8. 微任务队列执行完毕，查看异步宏任务队列，执行延时回调，输出`setTimeout`
```

Q：<script>标签的async属性与defer属性之间的异同

- 同：
  - 两者都只针对外部脚本(即使用src属性设置路径的脚本)
- asycn属性规定脚本立即下载但不阻塞其它操作，defer属性规定脚本立即下载但延迟到文档完全被解析和显示之后再执行
- 设置async属性脚本不可以修改DOM，但设置defer的可以
- 设置async属性的脚本可能会在DOMContentLoaded之前或之后执行，而设置defer属性的一定在DOMContentLoaded之后执行。

考察： 页面加载性能优化

Q：请写出以下程序的执行结果，为什么？

```
var  a = ['a1','a2']
var b = a
[0,1].slice(1)
console.log(b)
```

输出：2

原因： 由于程序语句省略了分号，JS会自动插入分号，程序变为：

```
var a = ['a1','a2'];
var b = a[0,1].slice(1);
console.log(b);
```

这里 a[0,1]里面的逗号不是作为数组元素分割符，而是作为逗号运算符，它会先运算左面的表达式，再运算右面的表达式，然后返回右面表达式的运算结果

> 考察： 自动分号插入；逗号操作符

Q：请写出以下程序的执行结果，为什么

```
{ // 外层块
  let x = "outer";
  { // 内层块
    console.log(x);
    var refX1 = function () { return x };
    console.log(refX1());
    const x = "inner";
    console.log(x);
    var refX2 = function () { return x };
    console.log(refX2());
  }
}
```

结果：ReferenceError: Cannot access 'x' before initialization（回答引用错误就算正确）

原因：

```
1. 外层块let形成外层块作用域;
```

2. 内层const 形成内层块作用域

3. const声明的变量在未初始化时会存在于暂时性死区
4. 所以第4行代码方位未初始化的x会发成引用错误。

> 考察：块级作用域；暂时性死区；

Q： 请完成 parseQuery 方法，要求必须使用正则

```
var parseQuery = function (query){

}

var obj = parseQuery("name=1&age=2");
console.log(obj); //{ name: '1', age: '2' }
```

参考实现：

```
var parseQuery = function (query){
var reg = /([^=&\s]+)[=\s]([^=&\s])/g;
var obj = {};
while(reg.exec(query)){
    obj[RegExp.$1] = RegExp.$2;
}
return obj;
}

var obj = parseQuery("name=1&age=2");
console.log(obj);
```

Q： 请完成 concatWithSort 方法：

```
var arr1 = ["A1", "A2", "B1", "B2", "C1", "C2", "D1", "D2"];
var arr2 = ["A", "B", "C", "D"];

function concatWithSort(arr1, arr2) {

}

var arr = concatWithSort(arr1, arr2);
console.log(arr); //  [ 'A1', 'A2', 'A', 'B1', 'B2', 'B', 'C1', 'C2', 'C', 'D1', 'D2',
'D' ]
```

参考实现：

```
var arr1 = ["A1", "A2", "B1", "B2", "C1", "C2", "D1", "D2"];
var arr2 = ["A", "B", "C", "D"];

function concatWithSort(arr1, arr2) {
```

```
return arr1.concat(arr2).sort(function(a, b) {
return (
 a.codePointAt(0) - b.codePointAt(0) ||
 b.length - a.length ||
 a.codePointAt(1) - b.codePointAt(1)
);
});
}

var arr = concatWithSort(arr1, arr2);
console.log(arr);
```

考察：1. concat(); 2. sort(); 3. codePointAt();

Q：以下代码，请使用至少两种不同方法实现a与b的值互换，要求不使用临时变量，并给出解题思路

```
var a = 1;
var b = 2;
```

解法一：

```
a = [b, b=a][0];
```

> 考察：对运算符的掌握，赋值运算符会先计算右面的表达式，然后将结果赋给左面的操作数，在右面的表达式计算中，由于逗号运算符的优先级是所有运算中最低的，所以，先会获取到 `[b, b=a][0]` 的值（ `b` ），然后运算 `b=a` 使得 `b` 获得 `a` 的值，最后再把先前获取到的 `b` 的值赋值给 `a`

解法二：

```
a = (b^=a^=b)^a;
```

> 考察：对位运算的掌握，利用异或操作两个操作数二进制对应位真价值不同威震，相同为假，通过两次交换真假值实现变量值的交换

解法三：

```
a = a + b;
b = a - b;
a = a - b;
```

解法四：

```
[a,b] = [b, a]
```

> 考察： 对ES6解构赋值的掌握

> 本题整体考察：js编程算法能力，能否利用所掌握的知识点解决复杂问题，能否找到最优算法。

面试部分

Q：请简述前后端分离架构的优缺点

> 优点：
>
> - 减轻服务器的请求和渲染压力
> - 解决后端工程师无法专注后端业务逻辑问题（因为要将静态html转换为动态页面，涉及css和js问题需要与前端协同开发）
> - 解决了应用前端无法利用更高效的静态服务器的问题；
> - 前后端可以使用各自专业的IDE，提升开发效率；
> - 解决了前后端的流程依赖问题（即后端工程师必须等待前端工程师的html页面做好后才能将其转化为动态页面，前端如果发生样式或逻辑改变，后端也必须返工），可以并行开发
> - 可以更快速和精确地定位问题，不会出现互相踢皮球的现象。
> - 后端某个服务出问题不会影响到前端整体呈现，只会有部分数据无法显示而已
> - 前端独立出来后更好地实现模块化、组件化和工程化。
> - 可以实现后台一套接口支持前端多种终端共用
>
> 缺点：
>
> - 加重了客户端的压力
> - 异步请求的增多容易造成用户体验的下降
> - SEO问题
> - 小项目浪费人员配置成本

Q： 请简述前端都有哪些安全问题，怎么解决

> - XSS-脚本攻击漏洞
> - 如果使用HTML进行内容转换，则使用innerText而不用innerHTML,或者把script、iframe标签过滤替换掉

- 对一些标签的字符串进行转移
- CSRF-跨站请求伪造
- 增加token验证；
- Referer页面来源验证（后端）；
- iframe安全隐患
- 使用安全的网站进行嵌入
- 在iframe上添加 sandbox 属性
- 本地存储数据问题
- 对本地存储的信息进行加密
- 第三方依赖安全隐患
- 利用自动化工具扫描第三方插件（如NSP）
- HTTP安全隐患
- 对服务器加HTTPS

Q： 浏览器缓存读取规则

缓存位置：

- service worker
- 浏览器独立线程，因涉及请求拦截必须使用HTTPS
- 可自由定制缓存哪些文件、如何匹配缓存、如何读取缓存，且缓存可持续
- 工作原理：注册--监听--请求拦截判断是否有缓存
- memory cache
- 存储在内存中，缓存脚本与图片
- 读取高效，持续时间短，容量小
- 随页面关闭而被释放
- 可利用 preloader 相关指令预下载资源
- disk cache
- 存储在硬盘中，缓存样式文件
- 读取较慢，但容量大，持续时间长
- push cache
- HTTP/2的内容，只有上述三种缓存都未命中，才会生效
- 只在会话中存在，会话结束后被释放缓存
- 只能被使用一次
- 缓存策略：
- 强缓存：expires和cache-control，由于前者存在时间准确性问题，一般使用后者
- 协商缓存：last-modified/if-modified-since 或者 Etag/if-none-match，Etag比前者精确度更高，但性能略差，服务器校验优先使用Etag
- 强缓存先于协商缓存进行，协商缓存失效，返回200，重新返回资源与缓存标识，协商缓存生效则返回304，继续使用缓存
- 用户行为
- 地址栏输入地址：查找disk cache中是否有匹配，没有匹配则发起网络请求
- 普通刷新页面（F5）：优先使用memory cache，其次才是disk cache

- 强制刷新（Ctrl+F5）: 不使用缓存

Q 简述浏览器的工作流程

1.解析HTML以构建DOM树：渲染引擎开始解析HTML文档，转换树中的html标签或js生成的标签到DOM节点，它被称为 – 内容树。

2. 构建渲染树：解析CSS（包括外部CSS文件和样式元素以及js生成的样式）成样式结构体，根据CSS选择器计算出节点的样式，创建另一个树 —- 渲染树（render tree）。*注：在解析的过程中会去掉浏览器不能识别的样式，比如IE会去掉-moz开头的样式，而firefox会去掉_开头的样式。*

3.布局渲染树: 从根节点递归调用，计算每一个元素的大小、位置等，给每个节点所应该出现在屏幕上的精确坐标。

4.绘制渲染树: 遍历渲染树，每个节点将使用UI后端层来绘制。

Q： cookie 和 token 都存放在 header 中，为什么不会劫持 token？

- token是防止跨站请求伪造的
- 因为浏览器进行信息提交时会自动带上cookie，给了伪造请求者可乘之机
- 而浏览器不会自动带上token，所以即使伪造者伪造了请求，也无法通过后端验证

Q：介绍模块化发展历程

- 一开始，全部代码都写在全局作用域中，于是出现了命名冲突的问题；
- 然后开始开发人员将各自的模块代码放入立即执行函数中，解决了命名冲突问题。但随着模块增多，全局的模块命名变量还是有可能冲突
- jquery风格的匿名自执行函数，解决了依赖模块的传递问题，虽然灵活，但未从根本上解决以上问题
- CommonJS出现，通过全局的require和exports来管理模块及其依赖，解决了全局作用域变量污染问题和依赖传递问题，但它是同步的，无法用于浏览器端；
- AMD出现（requirejs），实现了模块的异步加载和对浏览器的支持。但出现了模块代码被预先执行的问题。
- CMD（sea.js）,定义模块时无需再罗列依赖数组，代码会预先下载但不会预先执行
- ES6模块：实现了编译时确定依赖关系，并且可以指定加载模块中用export暴露的接口，没指定的就不会加载