# 易忘，但常问的面试题

## 11、 map、forEach 和 for 的区别

### map方法

`map` 方法用于对数组中的每个元素进行操作，并返回一个新的数组，该数组包含原数组中每个元素操作后的结果。它不会改变原数组，而是返回一个新的数组。

### forEach方法

`forEach` 方法用于对数组中的每个元素执行一个函数，但不会返回任何值。它会遍历数组中的每个元素，并对其执行提供的函数。与 `map` 方法不同， `forEach` 方法不会创建新的数组，而是直接在原数组上进行操作。

### for 循环

`for` 循环是一种通用的迭代结构，它可以用于迭代任何可迭代对象（如数组、字符串等）。它通过指定迭代的起始索引、结束索引和迭代步骤来遍历可迭代对象中的每个元素。

> 三者的性能比较：`for > forEach > map`

`for`：for循环没有额外的函数调用栈和上下文，所以它的实现最为简单。
`forEach`：对于forEach来说，它的函数签名中包含了参数和上下文，所以性能会低于 `for` 循环。
`map`： `map` 最慢的原因是因为 `map` 会返回一个新的数组，数组的创建和赋值会导致分配内存空间，因此会带来较大的性能开销。

`for` 循环是js提出时就有的循环方法。 `forEach` 和 `Map` 是ES5提出的，挂载在可迭代对象原型上的方法。

## 12、ES6模块化和CommonJS的区别

1. **语法差异：**

- **ES6 模块化：** 使用 `import` 和 `export` 语法。

```
1 js
2 复制代码
3 // 导入import { myFunction } from "./myModule";// 导出export const pi =
  3.14;export function square(x) {  return x * x;}
```

- **CommonJS：** 使用 `require` 和 `module.exports` 语法。

```
1 js
```

```
复制代码
// 导入const myModule = require("./myModule");// 导出exports.pi = 3.14;exports.square = function(x) {  return x * x;};
```

1. **加载时机：**
   - **ES6 模块化：** 在编译阶段静态分析，加载时可以进行优化。
   - **CommonJS：** 在运行时加载，是同步加载模块的方式。
2. **模块值的拷贝：**
   - **ES6 模块化：** 模块值是动态映射关系，只读视图，通过实时的映射关系来获取值。
   - **CommonJS：** 是值的拷贝，一旦加载，被加载模块的值变化不会影响导入模块。
3. **导出方式：**
   - **ES6 模块化：** 导出的是值的引用，修改导入值会影响到导出值。
   - **CommonJS：** 导出的是值的拷贝，修改导入值不会影响到导出值。
4. **适用场景：**
   - **ES6 模块化：** 主要用于浏览器端和现代前端开发，支持异步加载。
   - **CommonJS：** 主要用于服务器端，是同步加载模块的一种较为简单的方式。
5. **ES6 模块化的静态特性：**
   - ES6 模块化在编译时可以静态分析模块之间的依赖关系，这使得一些工具（如 tree shaking）能够更好地优化代码，去除未使用的部分。

总的来说，ES6 模块化在语法上更加优雅，支持静态分析，异步加载等特性，适用于现代前端开发；而 CommonJS 更适用于服务器端的模块化开发。

## 13、 深浅拷贝的区别

> 浅拷贝复制对象的引用，深拷贝创建一个新对象并递归复制所有嵌套的对象

**浅拷贝：**

- 浅拷贝仅复制对象或数组的第一层结构，对于嵌套的对象或数组，仅复制引用而不是实际的对象或数组。
- 常见的浅拷贝方法有 `Object.assign()`、扩展运算符（`...`）、`Array.slice()` 等。

```
js
复制代码
   const originalArray = [1, 2, { a: 3 }];   const shallowCopy = [...originalArray];      originalArray[0] = 100;   originalArray[2].a = 99;    console.log(originalArray); // [100, 2, { a: 99 }]  console.log(shallowCopy); // [1, 2, { a: 99 }]
```

上面代码中，说明 `shallowCopy` 仅复制 `originalArray数组` 的第一层结构，其他深层次结构则复制的是引用，对 `shallowCopy` 中数据进行修改时， `shallowCopy 数组` 中除第一层结构以外的数据，都会被修改（通俗的说，就是除开第一层结构是自己的，其他层都是别人的，别人想改就改）。

**深拷贝：**

- 深拷贝会递归地复制对象或数组及其所有嵌套的对象或数组，生成一份完全独立的副本。
- 深拷贝能够解决浅拷贝中引用传递的问题，确保复制的对象和原对象互不影响。
- 常见的深拷贝方法有使用递归、 `JSON.parse(JSON.stringify())` 、第三方库如 lodash 的 `_.cloneDeep()` 等。

```
1 js
2 复制代码
3    const originalArray = [1, 2, { a: 3 }];   const deepCopy =
JSON.parse(JSON.stringify(originalArray));   originalArray[2].a = 99;
console.log(deepCopy); // [1, 2, { a: 3 }]
```

深拷贝开辟了一个新的空间，将数据复制进去，进行处理操作时，与原数据互不影响。

**浅拷贝实现方式：**

- **Object.assign()**

```
1 js
2 复制代码
3 const shallowCopy = Object.assign({}, originalObject);
```

- **扩展运算符（...）**

```
1 js
2 复制代码
3 const shallowCopy = [...originalArray];
```

- **Array.slice()**

```
1 js
2 复制代码
```

```
3 const shallowCopy = originalArray.slice();
```

**深拷贝实现方式：**

- **递归实现**（递归实现深拷贝可能会因为循环引用而陷入死循环）

```
1 js
2 复制代码
3 function deepClone(obj) {  if (obj === null || typeof obj !== 'object') {
  return obj;  }  const clone = Array.isArray(obj) ? [] : {};  for (let key in
  obj) {    if (obj.hasOwnProperty(key)) {      clone[key] =
  deepClone(obj[key]);    }  }  return clone;}
```

- **JSON.parse(JSON.stringify())**（不能复制函数、正则表达式等特殊对象）

```
1 js
2 复制代码
3 const deepCopy = JSON.parse(JSON.stringify(originalObject));
```

- **第三方库，如 lodash**

```
1 js
2 复制代码
3 const deepCopy = _.cloneDeep(originalObject);
```

## 14、 var、let、const的区别

`var`、`let` 和 `const` 是 JavaScript 中用来声明变量的关键字，它们的区别如下：

1. 作用域：

- `var`：函数作用域，在函数内部声明的 `var` 变量可以在整个函数内部使用。
- `let`：块级作用域，在代码块（花括号）内部声明的 `let` 变量只能在该代码块内部使用。
- `const`：块级作用域，与 `let` 类似，但声明的变量的值不能被修改。

1. 变量提升：

- `var`：存在变量提升，即在声明变量之前就可以使用该变量，但是值为 `undefined`。
- `let` 和 `const`：不存在变量提升，必须先声明再使用，否则会抛出 `ReferenceError` 异常。

1. 可修改性：

- `var` 和 `let`：声明的变量的值可以被修改。
- `const`：声明的变量的值不能被修改，必须在声明时或在构造函数中进行初始化。

1. 暂时性死区（Temporal Dead Zone，TDZ）：

- `let` 和 `const`：在它们所声明的块级作用域内，存在暂时性死区。在 TDZ 中，不能访问该变量，也不能修改该变量的值。

1. 全局对象属性：

- 使用 `var` 声明的变量，会成为全局对象的属性，前提是没有在函数内部或块级作用域中声明该变量。
- 使用 `let` 声明的变量不会成为全局对象的属性。
- 使用 `const` 声明的变量不会成为全局对象的属性，但如果声明同时进行初始化，会在全局对象上创建一个只读属性。

## 15、 介绍下 BFC 及其应用

BFC(Blcok formatting context) 直译为“块级格式化上下文”。他是一个独立的渲染区域，只有块级元素参与，它规定了内部块级元素的布局，并且与这个区域外部毫不相关，外部元素也不会影响这个渲染区域的元素。

> 简单说：BFC 就是页面上的一个隔离的独立渲染区域，区域里边的子元素不会影响到外面的元素。外边的元素也不会影响到区域里面的子元素。

以下是一些常见的创建 BFC 的方法：

1. 浮动元素：将一个元素设置为浮动（`float: left` 或 `float: right`）会创建一个 BFC。
2. 绝对定位元素：将一个元素设置为绝对定位（`position: absolute`）会创建一个 BFC。
3. 固定定位元素：将一个元素设置为固定定位（`position: fixed`）会创建一个 BFC。
4. 具有 `overflow` 属性的元素：将一个元素的 `overflow` 属性设置为非 `visible` 的值（例如 `overflow: hidden`、`overflow: auto` 或 `overflow: scroll`）会创建一个 BFC。
5. 具有 `display` 属性为 `inline-block`、`table-cell`、`table-caption` 或 `flex` 的元素：这些元素会自动创建一个 BFC。
6. 根元素（`html` 元素）：根元素始终是一个 BFC。

**BFC 可以解决那些问题**

1. 避免垂直方向的 `margin` 合并。问题：垂直方向上的，两个元素 `margin` 相遇，两元素间的距离并不等于两个 `margin` 之和。而是等于最大的 `margin`。小的 `margin` 会被大的 `margin` 吞并。
2. 清除浮动元素的影响。问题：如果父元素包含一个浮动元素，那么其他元素可能会受到浮动元素的影响，导致布局混乱。但是给父元素变成BFC，浮动元素对其他元素的布局不再产生影响。

3. 防止高度塌陷。问题：父元素不写高度时，子元素浮动后，导致父元素会发生高度塌陷（造成父元素高度为0）。但是将父元素变成BFC，就不会造成高度塌陷，最简单的方法是，给父元素设置 `overflow: hidden` 属性。
4. 垂直布局。问题：父元素包含多个子元素，并且这些子元素的高度不同，那么在没有创建 BFC 的情况下，这些子元素可能会在垂直方向上重叠。但是将父元素变成BFC，子元素在垂直方向上就能正确排列。

## 16、 什么是作用域链?

- 作用域可以视为一套规则，这套规则用来管理引擎如何在当前作用域以及嵌套的子作用域根据标识符名称进行变量查找。
- 简单来说作用域就是变量的有效范围。在一定的空间里可以对变量数据进行读写操作，这个空间就是变量的作用域。

> 作用域链的本质，就是底层变量查找机制

**过程：** 当在 `js` 中使用一个变量的时候，首先 `js` 引擎会尝试在当前作用域下去寻找该变量，如果没找到，再到它的上层作用域寻找，以此类推直到找到该变量或是已经到了全局作用域。

**作用：** 保证对执行环境有权访问的所有变量和函数的有序访问，通过作用域链，可以访问到外层环境的变量和函数。

## 17、 浏览器的垃圾回收机制

### （1）内存的生命周期

JS 环境中分配的内存, 一般有如下生命周期：

1. 内存分配：当我们声明变量、函数、对象的时候，系统会自动为他们分配内存
2. 内存使用：即读写内存，也就是使用变量、函数等
3. 内存回收：使用完毕，由垃圾回收自动回收不再使用的内存

全局变量一般不会回收, 一般局部变量的的值, 不用了, 会被自动回收掉

### （2）垃圾回收的概念

**垃圾回收**：JavaScript代码运行时，需要分配内存空间来储存变量和值。当变量不在参与运行时，就需要系统收回被占用的内存空间，这就是垃圾回收。

**回收机制**：

- Javascript 具有自动垃圾回收机制，会定期对那些不再使用的变量、对象所占用的内存进行释放，原理就是找到不再使用的变量，然后释放掉其占用的内存。
- JavaScript中存在两种变量：局部变量和全局变量。全局变量的生命周期会持续要页面卸载；而局部变量声明在函数中，它的生命周期从函数执行开始，直到函数执行结束，在这个过程中，局部变量会在堆或栈中存储它们的值，当函数执行结束后，这些局部变量不再被使用，它们所占有的空间就会被释放。

- 不过，当局部变量被外部函数使用时，其中一种情况就是闭包，在函数执行结束后，函数外部的变量依然指向函数内部的局部变量，此时局部变量依然在被使用，所以不会回收。

**（3）垃圾回收的方式**

**1.引用计数法**

- 这个用的相对较少，IE采用的引用计数算法。引用计数就是跟踪记录每个值被引用的次数。当声明了一个变量并将一个引用类型赋值给该变量时，则这个值的引用次数就是1。相反，如果包含对这个值引用的变量又取得了另外一个值，则这个值的引用次数就减1。当这个引用次数变为0时，说明这个变量已经没有价值，因此，在在机回收期下次再运行时，这个变量所占有的内存空间就会被释放出来。
- 这种方法会引起**循环引用**的问题：例如：`obj1`和`obj2`通过属性进行相互引用，两个对象的引用次数都是2。当使用循环计数时，由于函数执行完后，两个对象都离开作用域，函数执行结束，`obj1`和`obj2`还将会继续存在，因此它们的引用次数永远不会是0，就会引起循环引用。

**2.标记清除法**

现代的浏览器已经不再使用引用计数算法了。

现代浏览器通用的大多是基于标记清除算法的某些改进算法，总体思想都是一致的。

- 标记清除是浏览器常见的垃圾回收方式，当变量进入执行环境时，就标记这个变量“进入环境”，被标记为“进入环境”的变量是不能被回收的，因为他们正在被使用。当变量离开环境时，就会被标记为“离开环境”，被标记为“离开环境”的变量会被内存释放。
- 垃圾收集器在运行的时候会给存储在内存中的所有变量都加上标记。然后，它会去掉环境中的变量以及被环境中的变量引用的标记。而在此之后再被加上标记的变量将被视为准备删除的变量，原因是环境中的变量已经无法访问到这些变量了。最后。垃圾收集器完成内存清除工作，销毁那些带标记的值，并回收他们所占用的内存空间。

**（4）如何减少垃圾回收**

虽然浏览器可以进行垃圾自动回收，但是当代码比较复杂时，垃圾回收所带来的代价比较大，所以应该尽量减少垃圾回收。

- **对数组进行优化：** 在清空一个数组时，最简单的方法就是给其赋值为[]，但是与此同时会创建一个新的空对象，可以将数组的长度设置为0，以此来达到清空数组的目的。
- **对`object`进行优化：** 对象尽量复用，对于不再使用的对象，就将其设置为null，尽快被回收。
- **对函数进行优化：** 在循环中的函数表达式，如果可以复用，尽量放在函数的外面。

**（5）内存泄漏是什么**

是指由于疏忽或错误造成程序未能释放已经不再使用的内存

**（6）哪些情况会导致内存泄漏**

以下四种情况会造成内存的泄漏：

1. **意外的全局变量：** 由于使用未声明的变量，而意外的创建了一个全局变量，而使这个变量一直留在内存中无法被回收。
2. **被遗忘的计时器或回调函数：** 设置了 setInterval 定时器，而忘记取消它，如果循环函数有对外部变量的引用的话，那么这个变量会被一直留在内存中，而无法被回收。
3. **脱离 DOM 的引用：** 获取一个 DOM 元素的引用，而后面这个元素被删除，由于一直保留了对这个元素的引用，所以它也无法被回收。
4. **闭包：** 不合理的使用闭包，从而导致某些变量一直被留在内存当中。

## 18、什么是函数式编程、

> 主要的编程范式有三种：命令式编程，声明式编程和函数式编程
>
> 函数式编程强调程序执行的结果而非执行的过程，倡导利用若干简单的执行单元让计算结果不断渐进，逐层推导复杂的运算，而非设计一个复杂的执行过程

**优点**

- 更好的管理状态：因为它的宗旨是无状态，或者说更少的状态，能最大化的减少这些未知、优化代码、减少出错情况
- 更简单的复用：固定输入->固定输出，没有其他外部变量影响，并且无副作用。这样代码复用时，完全不需要考虑它的内部实现和外部影响
- 更优雅的组合：往大的说，网页是由各个组件组成的。往小的说，一个函数也可能是由多个小函数组成的。更强的复用性，带来更强大的组合性
- 隐性好处。减少代码量，提高维护性

**缺点**

- 性能：函数式编程相对于指令式编程，性能绝对是一个短板，因为它往往会对一个方法进行过度包装，从而产生上下文切换的性能开销
- 资源占用：在 JS 中为了实现对象状态的不可变，往往会创建新的对象，因此，它对垃圾回收所产生的压力远远超过其他编程方式
- 递归陷阱：在函数式编程中，为了实现迭代，通常会采用递归操作

## 19、 事件循环Event Loop

> 事件循环指的是js代码所在运行环境（浏览器、nodejs）编译器的一种解析执行规则。

**（1） JS的执行机制(同步任务、异步任务)**

单线程是为了避免UI操作混乱，所有和UI操作相关的开发语言都应该是单线程。

JS是一门单线程语言，单线程就意味着，所有的任务需要排队，前一个任务结束，才会执行下一个任务。这样所导致的问题是：如果JS执行的时间过长，这样就会造成页面的渲染不连贯，导致页面渲染加载阻塞的觉。为了解决这个问题，JS中出现了同步和异步。

**同步任务**：即主线程上的任务，按照顺序由上至下依次执行，当前一个任务执行完毕后，才能执行下一个任务。

**异步任务**：不进入主线程，而是进入任务队列的任务，执行完毕之后会产生一个回调函数,并且通知主线程。当主线程上的任务执行完后，就会调取最早通知自己的回调函数，使其进入主线程中执行。

**（2） 宏任务与微任务的概念与区别**

为了让任务有条不紊地在主线程上执行，页面进程引入了 **消息队列** 和 **事件循环机制**，渲染进程内部也会维护多个消息队列，比如延迟执行队列和普通的消息队列。然后主线程采用一个 `for` 循环，不断地从这些任务队列中取出任务并执行任务。这些消息队列中的任务就称为 **宏任务**。

**微任务**是一个需要异步执行的回调函数，执行时机是在主函数执行结束之后、当前宏任务结束之前。当 JS 执行一段脚本（一个宏任务）的时候，V8 会为其创建一个全局执行上下文，在创建全局执行上下文的同时，V8 引擎也会在内部创建一个 **微任务队列**。也就是说 **每个宏任务都关联了一个微任务队列**。

**（3） 事件循环Event Loop执行机制**

1.进入到script标签,就进入到了第一次事件循环.

2.遇到同步代码，立即执行

3.遇到宏任务,放入到宏任务队列里.

4.遇到微任务,放入到微任务队列里.

5.执行完所有同步代码

6.执行微任务代码

7.微任务代码执行完毕，本次队列清空

8.寻找下一个宏任务，重复步骤1

**（4） 常见的宏任务与微任务分别有哪些**

常见的宏任务有：

1. setTimeout 函数：用于在指定的时间后执行一个函数。
2. setInterval 函数：用于按照指定的时间间隔重复执行一个函数。
3. requestAnimationFrame 函数：用于在浏览器的每一帧中执行一个函数。
4. I/O 操作：例如文件读取、网络请求等。
5. UI 事件：例如鼠标点击、键盘按下等。

常见的微任务有：

1. Promise 对象的回调函数：当 Promise 对象的状态变为 resolved 或 rejected 时，会调用相应的回调函数。
2. MutationObserver 的回调函数：当监听的 DOM 元素发生变化时，会调用相应的回调函数。

3. Async/Await 中的异步函数：Async/Await 是基于 Promise 的语法糖，异步函数内部的代码会在微任务队列中执行。
4. process.nextTick 函数（Node.js 环境）：用于在当前事件循环的下一个滴答中执行一个函数。

## 20、 浏览器缓存机制

### 缓存行为

- 浏览器每次发起请求，都会先在浏览器缓存中查找该请求的结果以及缓存标识
- 浏览器每次拿到返回的请求结果都会将该结果和缓存标识存入浏览器缓存中

### 缓存位置

- `Service Worker`：是运行在浏览器背后的独立线程，无法直接访问`DOM`，但可以用来做`离线缓存`、`消息推送`和`网络代理`。传输协议必须为`HTTPS`。
- `Memory Cache`：内存中的缓存
- `Disk Cache`：存储在硬盘中的缓存
- `Push Cache`：（`推送缓存`）是`HTTP/2`中的内容；

### 缓存过程

1. 浏览器第一次加载资源，服务器返回`200`，浏览器将资源文件从服务器上请求下载下来，并把`response header`及该请求的返回时间一并缓存；
2. 下一次加载资源时，先比较当前时间和上一次返回`200`时的时间差，如果没有超过`cache-control`设置的`max-age`，则没有过期，命中强缓存，不发请求直接从本地缓存读取该文件（如果浏览器不支持`HTTP1.1`，则用`expires`判断是否过期）；如果时间过期，则向服务器发送`header`带有`If-None-Match`和`If-Modified-Since`的请求；
3. 服务器收到请求后，优先根据`Etag`的值判断被请求的文件有没有做修改，`Etag`值一致则没有修改，命中协商缓存，返回`304`；如果不一致则有改动，直接返回新的资源文件带上新的Etag值并返回`200`；
4. 如果服务器收到的请求没有`Etag`值，则将`If-Modified-Since`和被请求文件的最后修改时间做比对，一致则命中协商缓存，返回`304`；不一致则返回新的`last-modified`和文件并返回`200`；

### 强缓存

强缓存表示在缓存期间不需要发送请求。

`强缓存`通过设置两种`HTTP Header`来实现，分别是`Expires`和`Cache-Control`。

- `Expires`是`HTTP/1.0`的产物。值代表的是服务端的时间，并且`Expires`受限于本地时间，如果修改了本地时间，可能会造成缓存失效。

- `Cache-Control` 出现于 `HTTP/1.1`，优先级高于 `Expires` 。该属性值表示资源会在 `多少秒` 后过期，需要再次请求。其中的 `max-age` 指令，记录了缓存过期的相对时间（相对请求的时间）。

> 注意：
>
> 如果 `Cache-Control` 属性值为 no-store ，表示不进行任何缓存；属性值为 no-cache ，表示强制使用协商缓存。

**协商缓存**

如果是 `首次请求` 或者 `Cache-Control` 的属性设置为 `no-cache` 时，又或者如果 `缓存过期了` ，都会向服务器发送请求，并且请求头中携带 `If-Modified-Since` 和 `If-None-Match` 来判断是否命中协商缓存，如果命中，则返回 `304` 状态码并且更新浏览器缓存有效期。

| 字段 | Header类型 | 协议版本 | 缓存类型 |
| --- | --- | --- | --- |
| Last-Modified | Response（响应头） | HTTP1.0 | 协商缓存 |
| If-Modified-Since | Request（请求头） | HTTP1.0 | 协商缓存 |
| ETag | Response（响应头） | HTTP1.1 | 协商缓存 |
| If-None-Match | Request（请求头） | HTTP1.1 | 协商缓存 |

- `Last-Modified` 表示本地文件最后修改时间，发送请求时，会将当前的 `Last-Modified` 值作为 `If-Modified-Since` 这个字段的内容，放在请求头中发送给服务器，去询问服务器在该时间后资源是否有更新，有更新的话就会将新的资源发送回来，否则返回 `304` 状态码。
- `ETag` 类似于文件指纹，请求时会将当前 `ETag` 作为 `If-None-Match` 这个字段的内容，并放到请求头中发送给服务器，服务器接收到 **`If-None-Match`** 后会跟服务器上该资源的 **`ETag`** 进行比对，有变动的话就将新的资源发送回来，否则返回 **`304`** 状态码。