大厂高级前端工程师面试题汇总

在 JavaScript 中，`0.1 + 0.2 === 0.3` 吗？请阐述原因并给出解决方案

这道题在网上有很多答案，解决方法也大相径庭，不过我在工作中就曾经遇到过，在前端对订单的各种数额进行计算，并与后端的结果进行对比，保证计算结果精度正确。

当然，问这道题，答案肯定是否，为什么呢？难道 `0.1 + 0.2` 不等于 `0.3` 吗？是的，在 JS 中，这道题的答案确实是 `false`，而 `0.2 + 0.3` 的结果却是 `0.5`，原因在于 JS 采用 IEEE 754 标准定义的 64 位浮点格式表示数字，所以 JS 中的所有数字都是浮点数。按照 JS 的数字格式，整数有的范围是 `-2^53 ~ 2^53`，而且只能表示有限个浮点数，能表示的个数为 `2^64 - 2^53 + 3` 个，浮点数的个数是无限的，这就导致了 JS 不能精确表达所有的浮点数，而只能是一个近似值。并且所有采用 IEEE 754 标准的语言都会有这个问题，只是它们已经在其标准库中解决了这个问题。而很遗憾的是 JS 却没有。下面我们来分析一下运算过程：

- 0.1 的二进制表示为 `1.1001100110011001100110011001100110011001100110011001 1(0011)+ * 2^-4`；
- 当 64bit 的存储空间无法存储完整的无限循环小数，而 IEEE 754 Floating-point(双精度) 采用 `round to nearest, tie to even` 的舍入模式，因此 0.1 实际存储时的位模式是 `0-01111111011-1001100110011001100110011001100110011001100110011010`；
- 0.2 的二进制表示为 `1.1001100110011001100110011001100110011001100110011001 1(0011)+ * 2^-3`；
- 当 64bit 的存储空间无法存储完整的无限循环小数，而 IEEE 754 Floating-point 采用 `round to nearest, tie to even` 的舍入模式，因此 0.2 实际存储时的位模式是 `0-01111111100-1001100110011001100110011001100110011001100110011010`；
- 实际存储的位模式作为操作数进行浮点数加法，得到 `0-01111111101-0011001100110011001100110011001100110011001100110100`。转换为十进制即为 `0.30000000000000004`。

那如何来解决这个问题呢？？原生的解决方法如下：

```
parseFloat((0.1 + 0.2).toFixed(10))
```

更精确的解决方案如下：

```
function accAdd (arg1, arg2) {
  var r1, r2, m;
  try{
    r1 = arg1.toString().split(".")[1].length
  } catch (e) {
    r1 = 0
  }

  try{
    r2 = arg2.toString().split(".")[1].length
  } catch (e) {
    r2 = 0
  }

  m = Math.pow(10, Math.max(r1,r2))
  return (parseInt(arg1*m, 10) + parseInt(arg2*m, 10)) / m
}
```

可以按此原理抽象成自己的标准计算库，在各个项目中使用。

详细说明 Event Loop

先说说概念吧，JS 是一种单线程语言，所谓单线程，意思就是一次只能执行一个任务，如果有多个任务，那么就排队，执行完一个再执行下一个 (还有其他方案，比如多线程或多进程)。这样的模式势必会造成资源浪费，也就是说，下一个任务必须等待，造成一种“假死” 的情况，从而无法响应用户的行为。那为什么 JS 从一开始不设计为一个多线程语言呢? 这是历史原因造成的，JS 本身被创造出来就是为了解决一些简单问题的，并且 JS 没有锁机制，如果存在多线程，DOM 操作将会变得复杂且不可控。当然，现在可以使用 `Web Worker API` 来实现多线程。

当这种等待机制运行时，会造成阻塞，这也就是同步机制，Event Loop 就是为了解决这个问题而生的。

> Event Loop 是一个程序结构，用于等待和发送消息和事件

简单说，就是在程序中设置两个线程：一个负责程序本身的运行，称为 “主线程”；另一个负责主线程与其他进程（主要是各种 I/O 操作）的通信，被称为 “Event Loop 线程”（可以译为 “消息线程”）。

每当遇到`I/O`的时候，主线程就让`Event Loop`线程去通知相应的`I/O`程序，然后接着往后运行, 等到`I/O`程序完成操作，`Event Loop`线程再把结果返回主线程。主线程就调用事先设定的回调函数，完成整个任务。

上面介绍的这种运行模式，就被称为 “异步模式”，或者 “非阻塞模式”。

下面我们再用几个例子来说明`Event Loop`中的几个概念。

JS 在执行的过程中会产生执行环境，这些执行环境会被顺序的加入到执行栈中。如果遇到异步的代码，会被挂起并加入到 `Task`（有多种 task） 队列中。一旦执行栈为空，`Event Loop` 就会从 `Task` 队列中拿出需要执行的代码并放入执行栈中执行，所以本质上来说 JS 中的异步还是同步行为。

```
console.log('script start');

setTimeout(function() {
  console.log('setTimeout');
}, 0);

console.log('script end');
// script start -> script end -> setTimeout
```

不同的任务源会被分配到不同的 `Task` 队列中，任务源可以分为微任务（`microtask`）和宏任务（`macrotask`）。在 ES6 规范中，`microtask` 称为 `jobs`，`macrotask` 称为 `task`。

```
console.log('script start');

setTimeout(function() {
  console.log('setTimeout');
}, 0);

new Promise((resolve) => {
    console.log('Promise')
    resolve()
}).then(function() {
  console.log('promise1');
}).then(function() {
  console.log('promise2');
});

console.log('script end');
// script start => Promise => script end => promise1 => promise2 => setTimeout
```

微任务包括 `process.nextTick` ， `promise` ， `Object.observe` ， `MutationObserver`

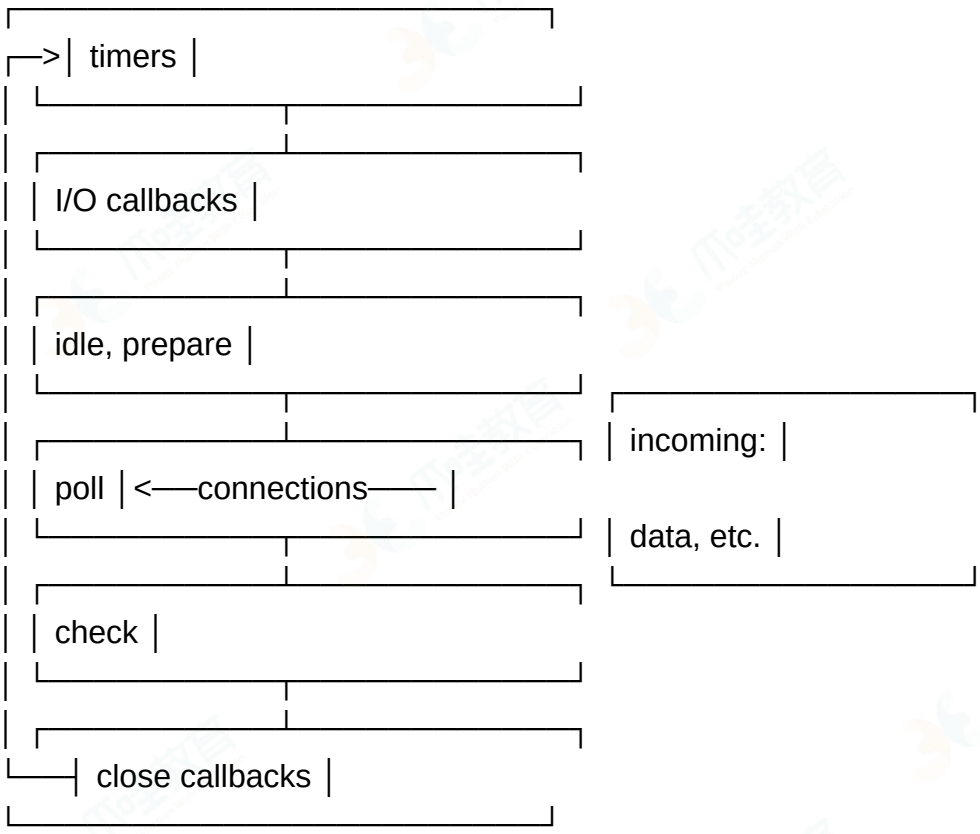
宏任务包括 `script` ， `setTimeout` ， `setInterval` ， `setImmediate` ， `I/O` ， `UI rendering`

所以正确的一次 Event loop 顺序是这样的

- 执行同步代码，这属于宏任务
- 执行栈为空，查询是否有微任务需要执行
- 执行所有微任务
- 必要的话渲染 UI
- 然后开始下一轮 Event loop，执行宏任务中的异步代码

除此之外，这里不得不提一下，Node 中也存在`Event Loop`机制，并且与浏览器中的不一样。

Node 的 Event loop 分为 6 个阶段，它们会按照顺序反复运行



timer

`timers` 阶段会执行 `setTimeout` 和 `setInterval`

一个 `timer` 指定的时间并不是准确时间，而是在达到这个时间后尽快执行回调，可能会因为系统正在执行别的事务而延迟。

下限的时间有一个范围：[1, 2147483647] ，如果设定的时间不在这个范围，将被设置为 1。

I/O

`I/O` 阶段会执行除了 `close` 事件，定时器和 `setImmediate` 的回调

idle, prepare

`idle`，`prepare` 阶段内部实现

poll

`poll` 阶段很重要，这一阶段中，系统会做两件事情

- 执行到点的定时器
- 执行 `poll` 队列中的事件

并且当 `poll` 中没有定时器的情况下，会发现以下两件事情

- 如果 `poll` 队列不为空，会遍历回调队列并同步执行，直到队列为空或者系统限制
- 如果 `poll` 队列为空，会有两件事发生

如果有 `setImmediate` 需要执行，`poll` 阶段会停止并且进入到 `check` 阶段执行 `setImmediate`
如果没有 `setImmediate` 需要执行，会等待回调被加入到队列中并立即执行回调
如果有别的定时器需要被执行，会回到 `timer` 阶段执行回调。

check

`check` 阶段执行 `setImmediate`

close callbacks

`close callbacks` 阶段执行 `close` 事件

并且在 Node 中，有些情况下的定时器执行顺序是随机的

```
setTimeout(() => {
    console.log('setTimeout');
```

```
}, 0);
setImmediate(() => {
    console.log('setImmediate');
})

// 这里可能会输出 setTimeout，setImmediate
// 可能也会相反的输出，这取决于性能
// 因为可能进入 event loop 用了不到 1 毫秒，这时候会执行 setImmediate
// 否则会执行 setTimeout
```

当然在这种情况下，执行顺序是相同的

```
var fs = require('fs')

fs.readFile(__filename, () => {
    setTimeout(() => {
        console.log('timeout');
    }, 0);
    setImmediate(() => {
        console.log('immediate');
    });
});
// 因为 readFile 的回调在 poll 中执行
// 发现有 setImmediate ，所以会立即跳到 check 阶段执行回调
// 再去 timer 阶段执行 setTimeout
// 所以以上输出一定是 setImmediate，setTimeout
```

上面介绍的都是 `macrotask` 的执行情况，`microtask` 会在以上每个阶段完成后立即执行。

```
setTimeout(()=>{
    console.log('timer1')

    Promise.resolve().then(function() {
        console.log('promise1')
    })
}, 0)

setTimeout(()=>{
    console.log('timer2')

    Promise.resolve().then(function() {
        console.log('promise2')
    })
}, 0)

// 以上代码在浏览器和 node 中打印情况是不同的
// 浏览器中打印 timer1, promise1, timer2, promise2
// node 中打印 timer1, timer2, promise1, promise2
```

Node 中的 `process.nextTick` 会先于其他 `microtask` 执行。

```
setTimeout(() => {
  console.log("timer1");

  Promise.resolve().then(function() {
    console.log("promise1");
  });
}, 0);

process.nextTick(() => {
  console.log("nextTick");
});
// nextTick, timer1, promise1
```

从输入 URL 到页面加载发生了什么？

这个问题很经典，也是很多面试高级前端时必问的问题。我也在面试时遇到过，只不过不同的是，面试官在这之前还问了一个问题，那就是从打开一个浏览器标签页开始，发生了什么。

也就是说，考察的是面试者对浏览器进程与线程的认知程度。下面是浏览器中进程的相关概念：

- 浏览器是多进程的
- 浏览器之所以能够运行，是因为系统给它的进程分配了资源（cpu、内存）
- 简单点理解，每打开一个 Tab 页，就相当于创建了一个独立的浏览器进程。

也就是说，新打开一个 TAB 页，实际上就创建了一个浏览器进程，但是有时候会有不同，为了性能考虑，浏览器的优化策略会将多个空的 TAB 页进程合并成一个，在有输入内容之后才分离出来创建另一个新的浏览器进程。

下面来说说当输入 url 之后，到底发生了什么。总体来说分为以下几个过程:

- DNS 解析
- TCP 连接
- 发送 HTTP 请求
- 服务器处理请求并返回 HTTP 报文

- 浏览器解析渲染页面
- 连接结束

DNS 解析

这个过程实际上是浏览器将输入的 url 发送到 DNS 服务器进行查询，DNS 服务器会返回当前查询 url 的 IP 地址。它实际上充当了一个翻译的角色，实现了网址到 IP 地址的转换

DNS 解析是一个递归查询的过程。

上图中演示的过程经历了 8 个步骤，如果每次都是这样，必然会损耗大量的资源，所以我们必须对 DNS 解析进行优化。

- DNS 缓存：DNS 存在着多级缓存，从离浏览器的距离排序的话，有以下几种: 浏览器缓存，系统缓存，路由器缓存，IPS 服务器缓存，根域名服务器缓存，顶级域名服务器缓存，主域名服务器缓存
- DNS 负载均衡：DNS 可以返回一个合适的机器的 IP 给用户，例如可以根据每台机器的负载量，该机器离用户地理位置的距离等等，这种过程就是 DNS 负载均衡，又叫做 DNS 重定向。大家耳熟能详的 CDN(Content Delivery Network) 就是利用 DNS 的重定向技术，DNS 服务器会返回一个跟用户最接近的点的 IP 地址给用户，CDN 节点的服务器负责响应用户的请求，提供所需的内容

TCP 连接

HTTP 协议是使用 TCP 作为其传输层协议的，当 TCP 出现瓶颈时，HTTP 也会受到影响。HTTP 报文是包裹在 TCP 报文中发送的，服务器端收到 TCP 报文时会解包提取出 HTTP 报文。但是这个过程中存在一定的风险，HTTP 报文是明文，如果中间被截取的话会存在一些信息泄露的风险。那么在进入 TCP 报文之前对 HTTP 做一次加密就可以解决这个问题了。HTTPS 协议的本质就是 HTTP + SSL(or TLS)。在 HTTP 报文进入 TCP 报文之前，先使用 SSL 对 HTTP 报文进行加密。从网络的层级结构看它位于 HTTP 协议与 TCP 协议之间。

HTTPS 在传输数据之前需要客户端与服务器进行一个握手 (TLS/SSL 握手)，在握手过程中将确立双方加密传输数据的密码信息。TLS/SSL 使用了非对称加密，对称加密以及 hash 等

HTTP 请求

发送 HTTP 请求的过程就是构建 HTTP 请求报文并通过 TCP 协议中发送到服务器指定端口 (HTTP 协议 80/8080, HTTPS 协议 443)。HTTP 请求报文是由三部分组成: 请求行, 请求报头和请求正文。

请求行的格式如下：

```
Method Request-URL HTTP-Version CRLF
```

例如：

```
eg: GET index.html HTTP/1.1
```

常用的方法有: GET, POST, PUT, DELETE, OPTIONS, HEAD。

请求报头：请求报头允许客户端向服务器传递请求的附加信息和客户端自身的信息。常见的请求报头有: `Accept`, `Accept-Charset`, `Accept-Encoding`, `Accept-Language`, `Content-Type`, `Authorization`, `Cookie`, `User-Agent` 等。

`Accept` 用于指定客户端用于接受哪些类型的信息， `Accept-Encoding` 与 `Accept` 类似，它用于指定接受的编码方式。 `Connection` 设置为 `Keep-alive` 用于告诉客户端本次 HTTP 请求结束之后并不需要关闭 TCP 连接，这样可以使下次 HTTP 请求使用相同的 TCP 通道，节省 TCP 连接建立的时间。

请求正文： 当使用 POST, PUT 等方法时，通常需要客户端向服务器传递数据。这些数据就储存在请求正文中。在请求包头中有一些与请求正文相关的信息，例如: 现在的 Web 应用通常采用 Rest 架构，请求的数据格式一般为 json。这时就需要设置 `Content-Type: application/json`。

服务器处理请求并返回 HTTP 报文

后端从在固定的端口接收到 TCP 报文开始，这一部分对应于编程语言中的 socket。它会对 TCP 连接进行处理，对 HTTP 协议进行解析，并按照报文格式进一步封装成 HTTP

Request 对象，供上层使用。这一部分工作一般是由 Web 服务器去进行

HTTP 响应报文也是由三部分组成: 状态码, 响应报头和响应报文。

状态码是由 3 位数组成，第一个数字定义了响应的类别，且有五种可能取值:

- 1xx：指示信息–表示请求已接收，继续处理。
- 2xx：成功–表示请求已被成功接收、理解、接受。
- 3xx：重定向–要完成请求必须进行更进一步的操作。
- 4xx：客户端错误–请求有语法错误或请求无法实现。
- 5xx：服务器端错误–服务器未能实现合法的请求。

平时遇到比较常见的状态码有: 200, 204, 301, 302, 304, 400, 401, 403, 404, 422, 500

响应报头: 常见的响应报头字段有: Server, Connection…。

响应报文: 服务器返回给浏览器的文本信息，通常 HTML, CSS, JS, 图片等文件就放在这一部分

浏览器解析渲染页面

浏览器在收到 HTML,CSS,JS 文件后，它是如何把页面呈现到屏幕上的

浏览器是一个边解析边渲染的过程。首先浏览器解析 HTML 文件构建 DOM 树，然后解析 CSS 文件构建渲染树，等到渲染树构建完成后，浏览器开始布局渲染树并将其绘制到屏幕上, 这个过程比较复杂，涉及到两个概念: `reflow`(回流) 和 `repain`(重绘)。

- `reflow` ： DOM 节点中的各个元素都是以盒模型的形式存在，这些都需要浏览器去计算其位置和大小等，这个过程称为 reflow
- `repain`当盒模型的位置, 大小以及其他属性，如颜色, 字体, 等确定下来之后，浏览器便开始绘制内容，这个过程称为 repain

页面在首次加载时必然会经历 `reflow` 和 `repain`。 `reflow` 和 `repain` 过程是非常消耗性能的，尤其是在移动设备上，它会破坏用户体验，有时会造成页面卡顿。所以我们应该尽可能少的减少 `reflow` 和 `repain`。

JS 的解析是由浏览器中的 JS 解析引擎完成的。JS 是单线程运行，也就是说，在同一个时间内只能做一件事，所有的任务都需要排队，前一个任务结束，后一个任务才能开始。但是又存在某些任务比较耗时，如 IO 读写等，所以需要一种机制可以先执行排在后面的任务，这就是：同步任务 (synchronous) 和异步任务(asynchronous)。

JS 的执行机制就可以看做是一个主线程加上一个任务队列 (task queue)。同步任务就是放在主线程上执行的任务，异步任务是放在任务队列中的任务。所有的同步任务在主线程上执行，形成一个执行栈; 异步任务有了运行结果就会在任务队列中放置一个事件；脚本运行时先依次运行执行栈，然后会从任务队列里提取事件，运行任务队列中的任务，这个过程是不断重复的，所以又叫做事件循环 (Event loop)。

浏览器在解析过程中，如果遇到请求外部资源时，如图像, iconfont,JS 等。浏览器将重复上面的过程下载该资源。请求过程是异步的，并不会影响 HTML 文档进行加载，但是当文档加载过程中遇到 JS 文件，HTML 文档会挂起渲染过程，不仅要等到文档中 JS 文件加载完毕还要等待解析执行完毕，才会继续 HTML 的渲染过程。原因是因为 JS 有可能修改 DOM 结构，这就意味着 JS 执行完成前，后续所有资源的下载是没有必要的，这就是 JS 阻塞后续资源下载的根本原因。CSS 文件的加载不影响 JS 文件的加载，但是却影响 JS 文件的执行。JS 代码执行前浏览器必须保证 CSS 文件已经下载并加载完毕

TCP、UDP 和 HTTP 的区别

TCP/IP 协议栈主要分为四层: 应用层、传输层、网络层、数据链路层, 每层都有相应的协议

- IP：网络层协议；（类似于高速公路）
- TCP 和 UDP：传输层协议；（类似于卡车）
- HTTP：应用层协议；（类似于货物）。HTTP(超文本传输协议)是利用 TCP 在两台电脑 (通常是 Web 服务器和客户端) 之间传输信息的协议。客户端使用 Web 浏览器发起 HTTP 请求给 Web 服务器，Web 服务器发送被请求的信息给客户端。

其实重要的在 TCP 和 UDP，那它们有什么区别呢？

TCP（传输控制协议，Transmission Control Protocol）：(类似打电话)
面向连接、传输可靠（保证数据正确性）、有序（保证数据顺序）、传输大量数据（流模

式）、速度慢、对系统资源的要求多，程序结构较复杂，每一条 TCP 连接只能是点到点的，TCP 首部开销 20 字节。

UDP(用户数据报协议，User Data Protocol)：（类似发短信）
面向非连接 、传输不可靠（可能丢包）、无序、传输少量数据（数据报模式）、速度快，对系统资源的要求少，程序结构较简单 ， UDP 支持一对一，一对多，多对一和多对多的交互通信，UDP 的首部开销小，只有 8 个字节。

TCP 建立连接需要三次握手：

- 第一次握手：客户端发送 syn 包 (seq=x) 到服务器，并进入 SYN_SEND 状态，等待服务器确认；
- 第二次握手：服务器收到 syn 包，必须确认客户的 SYN（ack=x+1），同时自己也发送一个 SYN 包（seq=y），即 SYN+ACK 包，此时服务器进入 SYN_RECV 状态；
- 第三次握手：客户端收到服务器的 SYN＋ACK 包，向服务器发送确认包 ACK(ack=y+1)，此包发送完毕，客户端和服务器进入 ESTABLISHED 状态，完成三次握手。

握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP 连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP 连接都将被一直保持下去

结论：

HTTP 协议是建立在请求 / 响应模型上的。首先由客户建立一条与服务器的 TCP 链接，并发送一个请求到服务器，请求中包含请求方法、URI、协议版本以及相关的 MIME 样式的消息。服务器响应一个状态行，包含消息的协议版本、一个成功和失败码以及相关的 MIME 式样的消息

虽然 HTTP 本身是一个协议，但其最终还是基于 TCP 的

HTTP 与 HTTPS 的区别

超文本传输协议 HTTP 协议被用于在 Web 浏览器和网站服务器之间传递信息，HTTP 协议以明文方式发送内容，不提供任何方式的数据加密，如果攻击者截取了 Web 浏览器和网站服务器之间的传输报文，就可以直接读懂其中的信息

为了解决 HTTP 协议的这一缺陷，需要使用另一种协议：安全套接字层超文本传输协议 HTTPS，为了数据传输的安全，HTTPS 在 HTTP 的基础上加入了 SSL 协议，SSL 依靠

证书来验证服务器的身份，并为浏览器和服务器之间的通信加密。

HTTP 和 HTTPS 的基本概念

HTTP：是互联网上应用最为广泛的一种网络协议，是一个客户端和服务器端请求和应答的标准（TCP），用于从 WWW 服务器传输超文本到本地浏览器的传输协议，它可以使浏览器更加高效，使网络传输减少。

HTTPS：是以安全为目标的 HTTP 通道，简单讲是 HTTP 的安全版，即 HTTP 下加入 SSL 层，HTTPS 的安全基础是 SSL，因此加密的详细内容就需要 SSL。

HTTPS 协议的主要作用可以分为两种：一种是建立一个信息安全通道，来保证数据传输的安全；另一种就是确认网站的真实性。

HTTP 与 HTTPS 有什么区别？

HTTP 协议传输的数据都是未加密的，也就是明文的，因此使用 HTTP 协议传输隐私信息非常不安全，为了保证这些隐私数据能加密传输，于是网景公司设计了 SSL（Secure Sockets Layer）协议用于对 HTTP 协议传输的数据进行加密，从而就诞生了 HTTPS。

HTTPS 加密、加密、及验证过程

简单来说，HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，要比 http 协议安全。

HTTPS 和 HTTP 的区别主要如下：

- https 协议需要到 ca 申请证书，一般免费证书较少，因而需要一定费用。
- http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议。
- http 和 https 使用的是完全不同的连接方式，用的端口也不一样，前者是 80，后者是 443。
- http 的连接很简单，是无状态的；HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 http 协议安全。

最不起眼的循环打印题

问题

先看下面的第一问，下列代码打印出什么？

```
for (var i = 0; i < 10; i++) {
  setTimeout(function() {
    console.log(i)
  }, 1000);
}
```

好了，是不是很经典？很多人在面试的时候都会被问到这个问题，OK，可能 80% 的同学都知道，它会打印出 `10` 个 `10` 来，为什么？

这就考到作用域的问题了。实际上，`var` 定义的变量在 `for` 循环之外是可以访问到的，也就是说，在执行 `setTimeout` 这个类似异步的操作之前，循环就已经结束了。这时的 `i` 已经为 `10`，所以最后打印出来的也就是 `10` 个 `10` 了

这就完了？呵呵，刚刚开始。

解决方案

下面来接着问，如何解决这个问题呢？ 有经验的同学都会想到使用 `IIFE`，这也考察了面试者对闭包的理解。

```
for (var i = 0; i < 10; i++) {
  (function(i) {
    setTimeout(function() {
      console.log(i);
    })
  })(i)
}
```

这样，就可以顺利的打印出 `0` ~ `9` 的结果了。如果还想考察，可以进一步问还有其他的解决方案吗？ 可以提示：让每次循环的代码块都能正常的拿到 `i` 值即可。

```
var output = function(i) {
  setTimeout(function() {
    console.log(i);
  }, 1000);
}

for (var i = 0; i < 10; i++) {
  output(i);
}
```

当然，最快的解决方案，有面试者可能会直接说出，利用 ES6 的 `let` 就可以，因为 `let` 可以定义一个块级作用域嘛。

```
for (let i = 0; i < 10; i++) {
  setTimeout(function() {
    console.log(i)
  }, 1000)
}
```

Promise

进一步，问利用 ES6 的 `Promise` 如何解决这个呢？ 下面给出答案，如果面试者能大概写出来，那么是一个大大的加分。

```
const tasks = [];
for (var i = 0; i < 10; i++) {
  (function(i) {
    tasks.push(new Promise((resolve) => {
        setTimeout(() => {
            console.log(i);
            resolve()
        })
    }))
  })(i)
}

Promise.all(tasks).then(() => {
  setTimeout(() => {
    console.log(i)
  }, 1000)
})
```

这里实际上考察了面试者对 ES6 的 `Promise` 的理解程度。下面是一种更简洁的写法：

```
const tasks = [];
const output = (i) => new Promise(resolve => {
  setTimeout(() => {
    console.log(i);
    resolve()
  }, 1000)
})

for (var i = 0; i < 10; i++) {
  tasks.push(output(i))
}
```

```
Promise.all(tasks).then(() => {
  setTimeout(() => {
    console.log(i)
  }, 1000)
})

// 最简洁写法：
new Promise((resolve) => {
  for (var i = 0; i < 10; i++) {
    console.log(i);
    resolve()
  }
})
```

async/await

最后再来看下，更 “变态” 的，可以进一步使用 ES7 的 `async/await` 来解决，下面的代码摘自网上，不得不说，太牛掰了。

```
const sleep = (timeout) => new Promise((resolve) => {
    setTimeout(resolve, timeout);
});

(async () => {
    for (var i = 0; i < 10; i++) {
        await sleep(1000);
        console.log(i);
    };
})();

// 这里也有个最简洁写法：
(async () => {
  for (var i = 0; i < 10; i++) {
    console.log(i)
  }
})();
```

防抖和节流

防抖

什么是防抖？

如果我们页面上有一个事件会被用户操作触发，而如果用户反复操作，就会被反复触发，这样带来的后果是性能低下和资源浪费。比如，页面有一个鼠标移入则会发送 `ajax` 请求

的事件，如果用户反复的操作，就会浪费网络资源，不停的去请求。这时，我们就需要防抖。

防抖的原理就是：当执行一个事件函数时，会等待一个阈 (yu) 值，可以设置为 `n` 秒，只有在 `n` 秒之后不再有操作，事件才会真正被触发。这样就不会引起页面抖动。

```
function debounce(func, wait) {
  var timeout;

  return function () {
    var context = this;
    var args = arguments;

    clearTimeout(timeout)
    timeout = setTimeout(function(){
      func.apply(context, args)
    }, wait);
  }
}
```

节流

节流的概念比较简单，就是当触发某个事件，每隔一段时间，只执行一次该事件。实现方式有两种：时间戳和定时器

```
// 时间戳方式实现：
function throttle(func, wait) {
  var context, args;
  var previous = 0;

  return function() {
    var now = +new Date();
    context = this;
    args = arguments;
    if (now - previous > wait) {
      func.apply(context, args);
      previous = now;
    }
  }
}

// 定时器方式：
function throttle(func, wait) {
  var timeout;
  var previous = 0;

  return function() {
```

```
    context = this;
    args = arguments;
    if (!timeout) {
      timeout = setTimeout(function(){
        timeout = null;
        func.apply(context, args)
      }, wait)
    }
  }
}
```

JavaScript 中的设计模式

说说你了解并掌握的设计模式在 JavaScript 中的实现。

单例模式 Singleton Pattern

保证一个类只有一个实例，并提供一个访问它的全局访问点（调用一个类，任何时候返回的都是同一个实例）

实现方法：使用一个变量来标志当前是否已经为某个类创建过对象，如果创建了，则在下一次获取该类的实例时，直接返回之前创建的对象，否则就创建一个对象。

```
class Singleton {
  constructor(name) {
    this.name = name
    this.instance = null
  }
  getName() {
    alert(this.name)
  }
  static getInstance(name) {
    if (!this.instance) {
      this.instance = new Singleton(name)
    }
    return this.instance
  }
}
```

工厂模式 Factory Pattern

工厂模式定义一个用于创建对象的接口，这个接口由子类决定实例化哪一个类。该模式使一个类的实例化延迟到了子类。而子类可以重写接口方法以便创建的时候指定自己的对象类型。

简单说：假如我们想在网页面里插入一些元素，而这些元素类型不固定，可能是图片、链接、文本，根据工厂模式的定义，在工厂模式下，工厂函数只需接受我们要创建的元素的类型，其他的工厂函数帮我们处理。

```
// 文本工厂
class Text {
  constructor(text) {
    this.text = text
  }
  insert(where) {
    const txt = document.createTextNode(this.text)
    where.appendChild(txt)
  }
}

// 链接工厂
class Link {
  constructor(url) {
    this.url = url
  }
  insert(where) {
    const link = document.createElement('a')
    link.href = this.url
    link.appendChild(document.createTextNode(this.url))
    where.appendChild(link)
  }
}

// 图片工厂
class Image {
  constructor(url) {
    this.url = url
  }
  insert(where) {
    const img = document.createElement('img')
    img.src = this.url
    where.appendChild(img)
  }
}

// DOM工厂
class DomFactory {
  constructor(type) {
    return new (this[type]())
  }

  // 各流水线
  link() { return Link }
  text() { return Text }
  image() { return Image }
}
```

```
// 创建工厂
const linkFactory = new DomFactory('link')
const textFactory = new DomFactory('text')

linkFactory.url = 'https://surmon.me'
linkFactory.insert(document.body)

textFactory.text = 'HI! I am surmon.'
textFactory.insert(document.body)
```

数据结构与算法题

统计字符串中出现次数最多的字符

```
let str = 'asdfghjklaqwertyuiopiaia';
const strChar = str => {
  let string = [...str],
      maxValue = '',
      obj = {},
      max = 0;

  string.forEach(value => {
      obj[value] = obj[value] == undefined ? 1 : obj[value] + 1
      if (obj[value] > max) {
          max = obj[value]
          maxValue = value
      }
  })
  return maxValue;
}
```

数组去重

```
// forEach
let arr = ['1', '2', '3', '1', 'a', 'b', 'b']
const unique = arr => {
    let obj = {}
    arr.forEach(value => {
        obj[value] = 0
    })
    return Object.keys(obj)
}

// filter
let arr = ['1', '2', '3', '1', 'a', 'b', 'b']
```

```
const unique = arr => {
    return arr.filter((ele, index, array) => {
        return index === array.indexOf(ele)
    })
}

// set
let arr = ['1', '2', '3', '1', 'a', 'b', 'b']
const unique = arr => {
    return [...new Set(arr)]
}
```

作用域

作用域是可访问变量的集合，在JavaScript中对象和函数同样是变量，作用域为可访问变量，对象，函数的集合。作用域可以分为全局作用域和局部作用域。

全局作用域： 变量在函数外定义，即为全局变量，全局变量有全局作用域，网页中所有脚本和函数都可以使用。如果变量在函数内没有声明，也是全局变量。

```
var name = "hello World";
// 此处可调用 name 变量
function myFunction() {
    // 函数内可调用 name 变量
}
```

```
// 此处可调用 name 变量
function myFunction() {
    name = "hello World";
    // 此处可调用 name 变量
}
```

局部作用域：变量在函数内声明，变量为局部作用域，只能在函数内部访问。

```
// 此处不能调用 name 变量
function myFunction() {
    var name = "hello World";
    // 函数内可调用 name 变量
}
```

局部变量只作用于函数内，所以不同的函数可以使用相同名称的变量。局部变量在函数执行时创建，函数执行完毕后局部变量就会自动销毁。

JavaScript变量生命周期，局部变量函数执行完毕后销毁，全局变量在页面关闭后销毁。函数参数只在函数内起作用，属于局部变量。

闭包

闭包：函数A内部有函数B，函数B可以访问函数A的变量，那么函数B就是闭包。本质上，闭包就是将函数内部和函数外部连接起来的一座桥梁。

```
function A(){
    var a = 123;
    function B(){
        console.log(a) //123
    }
    return B()
}
A()();
```

闭包有3大特性：

- 函数嵌套函数
- 函数内部可以引用函数外部的参数和变量
- 参数和变量不会被垃圾回收机制回收

闭包优点：

1. 可读取函数内部的变量
2. 局部变量可以保存在内存中，实现数据共享
3. 执行过程所有变量都匿名在函数内部

闭包缺点：

1. 使函数内部变量存在内存中，内存消耗大
2. 滥用闭包可能会导致内存泄漏
3. 闭包可以在父函数外部改变父函数内部的值，慎操作

使用场景：

1. 模拟私有方法
2. setTimeout的循环

3. 匿名自执行函数
4. 结果要缓存场景
5. 实现类和继承

this指向

this是在函数运行时，在函数体内部自动生成的一个对象，只能在函数体内部使用。通过捣鼓这么多代码，无非就是几种情况，在不同的环境下会有不同的值。发现网上很多关于this的文章都会让人觉得很难以理解，讲解一大堆例子但是没有讲到点上。

首先我们来看一下代码

```
var a = 1
function foo() {
    console.log(this.a)
}
foo()

const obj = {
 a: 2,
 foo: foo
}
obj.foo()

const c = new foo()
```

- 对于直接调用函数来说，不管foo函数被放在了什么地方，this的指向一定是window
- 对于obj.foo()来说，谁调用了函数那么谁就是this
- 对于new操作实例化来说，this就会绑定在实例化对象上面且不会被改变
- 箭头函数this只取决于包裹箭头函数的第一个普通函数的this

PS：箭头函数是没有this的，只会从自己的作用域链的上一层继承this。箭头函数的this在它被定义的时候就确定了，之后永远不会改变。

跨域怎么解决

跨域解决主要有以下几种：

- JSONP
- CORS
- Nginx代理
- document.domain
- window.name
- postMessage+iframe

1、JSONP

我们知道写HTML代码的时候，加入图片链接就不会有获取不到图片的问题。这是因为图片资源并没有进行ajax请求，而且script标签是没有同源策略的，可以进行资源请求，可以说是一个前端设计的漏洞。

```
// 1.回调函数
function handleResponse(data){
    console.log(data);
}
// 2.动态创建 script
var script = document.createElement('script');
script.src = 'http://test.com/json?callback=handleResponse';
document.body.insertBefore(script,document.body.firstChild);
```

利用script标签立即下载并执行的特性，我们就可以在回调函数中拿到返回来的数据。那么是不是所有的情况都可以呢？显然不是的。虽然实现是比较简单的操作，但是有缺点：

1. 仅限于GET请求
2. 有安全问题，万一有恶意代码返回，前端无法阻止
3. 无法检测请求是否成功

2、CORS

CORS是跨域资源共享(Cross-origin resource sharing)

要想利用这个技术关键是在于服务端，设置返回的Access-Control-Allow-Origin响应头允许跨域操作，发送请求时有两种情况：

- 简单请求
- 复杂请求

①简单请求

当使用以下方法之一：

- GET
- HEAD
- POST

Content-Type的值为以下之一：

- text/plain
- multipart/form-data
- application/x-www-form-urlencoded

才会发起简单请求，浏览器判断是简单请求的话就会在请求头添加origin字段，值是发起请求的所在的源。服务端收到请求后会判断origin是否在自己的许可范围，如果不在就拒绝，如果在就会有以下的响应头添加：

- Access-Control-Allow-Origin（必选）：告诉客户端我接受请求，值为origin的值，若允许所有源请求就会返回 *。
- Access-Control-Allow-Credentials（可选）：告诉浏览器发送请求时携带Cookie，true表示允许false表示禁止。
- Access-Control-Expose-Headers（可选）：额外给客户端返回的头部字段。

②复杂请求

复杂请求会有两次，第一次是发送一个预检请求，使用的方法是options，询问服务器是否允许我进行跨域请求资源。并且允许客户端自定义请求头的类型，询问服务器是否允许。

```
OPTIONS /cors HTTP/1.1
Origin: http://test.com
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: Custom-Header1,Custom-Header2
Host: target.com
Accept-Language: en-US
Connection: keep-alive
User-Agent: Mozilla/5.0...
```

然后服务器会进行验证，还会在响应头进行说明允许你的请求。

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://test.com
Access-Control-Allow-Methods: GET, POST, PUT
Access-Control-Allow-Headers: Custom-Header1,Custom-Header2
Access-Control-Max-Age: 1728000
Content-type: text/html; charset=utf-8
Content-Encoding: gzip
Content-Length: 0
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain
```

- Access-Control-Allow-Origin：告诉客户端，允许你这个源的请求
- Access-Control-Allow-Methods：告诉客户端，服务端允许的跨域 AJAX 请求的类型，也可进行 GET 或者 POST 请求
- Access-Control-Allow-Headers：告诉客户端，服务端允许的发送请求时的自定义请求头
- Access-Control-Max-Age: 告诉客户端预检请求的有效期，省去了多次的预检请求。也就是说，1728000 秒内你可以直接发送真正的 AJAX 请求，不用每次询问

3、Nginx代理

将nginx目录下的nginx.conf修改，通过反向代理的方式来实现跨域请求。

```
# /所有以apis开头发起的请求会被分发到myserver
location ^~ /apis/ {
     proxy_pass http://myserver; # 负载均衡名，写你请求的服务器地址
     proxy_set_header X-real-ip $remote_addr;
     proxy_set_header Host $http_host;
}
```

4、document.domain

该方式只能用于二级域名相同的情况下，比如 a.test.com 和 b.test.com 适用于该方式。

只需要给页面添加 document.domain = 'test.com' 表示二级域名都相同就可以实现跨域

5、window.name

window.name有一个奇妙的性质，页面如果设置了window.name，那么在不关闭页面的情况下，即使进行了页面跳转location.href=...，这个window.name还是会保留。

```
// 打开必应 https://www.bing.com/
// 打开控制台
> window.name
""
> window.name='test';
"test"
> location.href='http://www.google.com';
"http://www.google.com"
Navigated to https://www.google.com/> window.name
"test"
```

6、postMessage+iframe

这种方式通常用于获取嵌入页面中的第三方页面数据。一个页面发送消息，另一个页面判断来源并接收消息

```
// 发送消息端
<div>
   <div id="color">Frame Color</div>
</div>
<div>
   <iframe id="child" src="http://b.com/b.html"></iframe>
</div>

window.onload=function(){
   window.frames[0].postMessage('getcolor','http://b.com');
}
// 接收消息端
window.addEventListener('message',function(e){
      console.log(e)
},false);
```

postMessage(data,origin)方法接受两个参数：

- data:要传递的数据，html5规范中提到该参数可以是JavaScript的任意基本类型或可复制的对象，然而并不是所有浏览器都做到了这点儿，部分浏览器只能处理字符串参数，所以我们在传递参数的时候需要使用JSON.stringify()方法对对象参数序列化
- origin：字符串参数，指明目标窗口的源，协议+主机+端口号[+URL]，URL会被忽略，所以可以不写，这个参数是为了安全考虑，postMessage()方法只会将message传递给指定窗口，当然如果愿意也可以把参数设置为"*"，这样可以传递给任意窗口，如果要指定和当前窗口同源的话设置为"/"。

Vue过滤器原理

过滤器实质不改变原始数据，只是对数据进行加工处理后返回过滤后的数据再进行调用处理。我们看一下官方的定义：

> Vue.js 允许你自定义过滤器，可被用于一些常见的文本格式化。过滤器可以用在两个地方：双花括号插值和 v-bind 表达式 (后者从 2.1.0+

> 开始支持)。过滤器应该被添加在 JavaScript 表达式的尾部，由“管道”符号指示:

```
<!-- 在双花括号中 -->
{{ message | capitalize }}

<!-- 在 `v-bind` 中 -->
<div v-bind:id="rawId | formatId"></div>
```

你可以在一个组件的选项中定义本地的过滤器：

```
filters: {
  capitalize: function (value) {
    if (!value) return ''
    value = value.toString()
    return value.charAt(0).toUpperCase() + value.slice(1)
  }
}
```

或者在创建 Vue 实例之前全局定义过滤器：

```
Vue.filter('capitalize', function (value) {
  if (!value) return ''
  value = value.toString()
  return value.charAt(0).toUpperCase() + value.slice(1)
})

new Vue({
  // ...
})
```

过滤器函数总接收表达式的值 (之前的操作链的结果) 作为第一个参数。在上述例子中，capitalize 过滤器函数将会收到 message 的值作为第一个参数。过滤器可以串联：

```
{{ message | filterA | filterB }}
```

在这个例子中，filterA 被定义为接收单个参数的过滤器函数，表达式 message 的值将作为参数传入到函数中。然后继续调用同样被定义为接收单个参数的过滤器函数 filterB，将 filterA 的结果传递到 filterB 中。

过滤器是 JavaScript 函数，因此可以接收参数：

```
{{ message | filterA('arg1', arg2) }}
```

这里，filterA 被定义为接收三个参数的过滤器函数。其中 message 的值作为第一个参数，普通字符串 'arg1' 作为第二个参数，表达式 arg2 的值作为第三个参数。

过滤器原理

```
{{ message | capitalize }}
```

上面的过滤器经过一顿操作之后就会变成：`_s(_f("capitalize")(message))`。

- _f：该函数其实就是resolveFilter的别名，作用是从`_this.$options.filter`找到过滤器并返回
- _s：该函数就是toString函数的别名，作用是拿到过滤之后的结果并传递给toString()函数，结果会保存到VNode中的text属性，返回结果直接渲染视图

串联过滤器

```
{{ message | filterA | filterB }}
```

上面的串联过滤器经过一顿操作之后就会变成：

`_s(_f("filterB")(_f("filterA")(message)))`

这里的意思就是message作为第一个参数传进filterA当中，然后经过filterA的处理结果就传进filterB当中。`即filterA过滤器的结果就是filterB过滤器的输入`。

过滤器参数接收

```
{{ message | filterA | filterB("param") }}
```

以上的过滤器的编译结果就是：

`_s(_f("filterB")(_f("filterA")(message),"param"))`

这里有一点注意的是：这个param参数是filterB的第二个参数，它的第一个参数是经过filterA处理的结果。

_f函数的原理

_f函数其实就是寻找过滤器的，如果找到过滤器就返回过滤器，找不到就返回与参数相同的值。它的代码其实很简单：

```
import {identity, resolveAssets} from 'core/util/index'

export function resolveFilter(id){
  return resolveAssets(this.$options, 'filters', id, true) || identity
}
```

我们重点来看一下resolveAssets到底做了什么事情。

```
export function resolveAsset (options, type, id, warnMissing){
  if(typeof(id) !== 'string'){
    return
  }

  const assets = options[type]
  if(hasOwn(assets, id)) return assets[id]
  const camelizedId = camelize(id)
  if(hasOwn(assets, camelizedId)) return assets[camelizedId]
  const PascalCaseId = capitlize(camelizedId)
  if(hasOwn(assets, PascalCaseId)) return assets[PascalCaseId]

  //检查原型链
```

```
  const res assets[id] || assets[camelizedId] || PascalCaseId
  if(process.env.NODE_ENV!=='production'&& warnMissing&&!res){
    warn('Fail to resolve' + type.slice(0,-1)+':'+id, options)
  }
  return res
}
```

其实它的寻找过程也很简单，主要是做了以下的操作（id是过滤器id）：

- 判断过滤器id是否为字符串，不是则终止
- 用assets存储过滤器
- hasOwn函数检查assets自身是否存在id属性，存在则返回
- hasOwn函数检查assets自身是否存在`驼峰化后的`id属性，存在则返回
- hasOwn函数检查assets自身是否存在将`首字母大写后的`id属性，存在则返回
- 如果还是没有，就是去原型链找，找不到就会打印警告

过滤器解析原理

我们想一下，解析器是怎么解析过滤器的语法？其实在vue内部专门有这么一个函数用来解析过滤器语法：`parseFilters`

它的原理就是解析过滤器列表，然后`循环过滤器列表`并`拼接字符串`。

vue中的路由模式

history模式

- HTML5中的两个API：pushState和replaceState，改变url之后页面不会重新刷新，也不会带有#号，页面地址美观，url的改变会触发popState事件，监听该事件也可以实现根据不同的url渲染对应的页面内容但是因为没有#会导致用户在刷新页面的时候，还会发送请求到服务端，为避免这种情况，需要每次url改变的时候，都将所有的路由重新定位到跟路由下

hash模式

- url hash: http ://foo.com/#help#后面hash值的改变，并不会重新加载页面，同时hash值的变化会触发hashchange事件，该事件可以监听，可根据不同的哈希值渲染不同

的页面内容

vue 3.0中proxy数据双向绑定

- Proxy 可以直接监听对象而非属性；
- Proxy 可以直接监听数组的变化；
- Proxy 有多达 13 种拦截方法,不限于 apply、ownKeys、deleteProperty、has 等等是 Object.defineProperty 不具备的；
- Proxy 返回的是一个新对象,我们可以只操作新的对象达到目的,而 Object.defineProperty 只能遍历对象属性直接修改；
- Proxy 作为新标准将受到浏览器厂商重点持续的性能优化，也就是传说中的新标准的性能红利；

ajax/axios/fetch区别

ajax

- 不符合现在前端MVVM的浪潮
- 基于原生的XHR开发，XHR本身的架构不清晰
- jQuery整个项目太大，单纯使用ajax却要引入整个jQuery

axios

- 从 node.js 创建 http 请求
- 支持 Promise API
- 客户端支持防止CSRF
- 提供了一些并发请求的接口

fetch

- 更加底层，提供的API丰富（request, response）
- 脱离了XHR，是ES规范里新的实现方式
- fetch只对网络请求报错，对400，500都当做成功的请求，需要封装去处理
- fetch默认不会带cookie，需要添加配置项

- fetch没有办法原生监测请求的进度，而XHR可以

webSocket通信原理

- 客户端会先发送一个HTTP请求，包含一个Upgrade请求头来告诉服务端要升级为WebSocket协议
- 服务器就会返回101状态码并切换为WebSocket协议建立全双工连接，后续信息将会通过这个协议进行传输

有几个头信息需要注意一下：

Sec-WebSocket-Key：客户端随机生成的一个base64编码
Sec-WebSocket-Accept：服务端经过算法处理后回传给客户端
Connection和Upgrade字段告诉服务器，客户端发起的是WebSocket协议请求

如何优化webpack配置

缩小文件查找范围

- 优化loader
- 优化resolve.modules
- 优化resolve.mainFields
- 优化resolve.alias
- 优化resolve.extensions
- 优化module.noPaese

使用DllPlugin

- 基础模块抽离，打包到动态链接库
- 需要使用模块，直接去动态链接库查找

使用HappyPack

- 单线程变多进程

使用ParallelUglifyPlugin

- 开启多进程压缩代码，并行执行

使用CDN加速

- 静态资源放到CDN服务器上面

Tree Shaking

- 剔除无用的代码

提取公共代码

- 防止相同资源重复加载
- 减少网络流量及服务器成本

使用prepack

- 编译代码时提前计算结果放到编译后的结果中，而不是在代码运行才求值

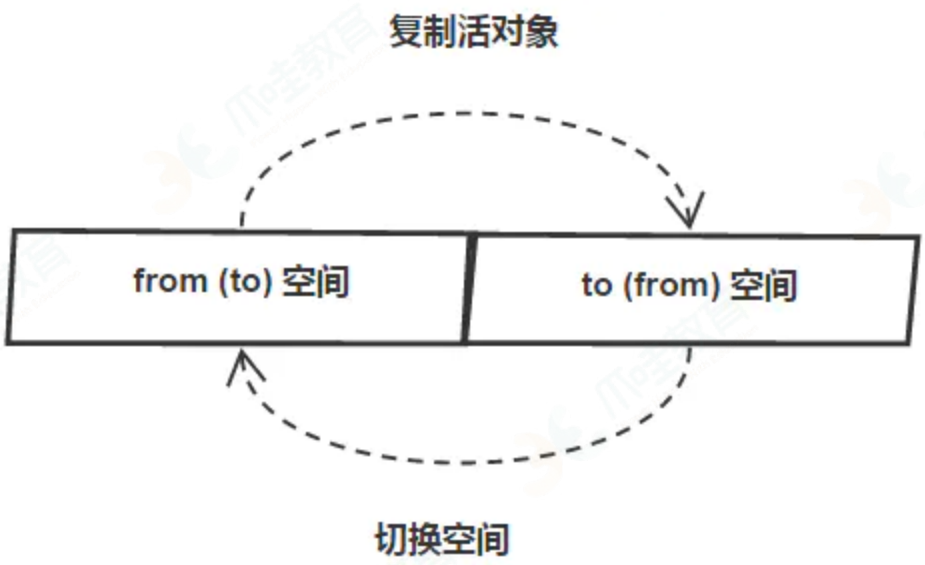
V8垃圾回收机制

新生代

新生代中的对象主要通过 `Scavenge` 算法进行垃圾回收。在 `Scavenge` 的具体实现中，主要采用了 `Cheney` 算法。

> Cheney 算法是一种采用复制的方式实现的垃圾回收算法。它将堆内存一分为二，每一部分空间成为 semispace。在这两个 semispace 空间中，只有一个处于使用中，另一个处于闲置中。处于使用中的 semispace 空间成为 From 空间，处于闲置状态的空间成为 To 空间。当我们分配对象时，先是在 From 空间中进行分配。当开始进行垃圾回收时，会检查 From 空间中的存活对象，这些存活对象将被复制到 To 空间中，而非存活对象占用的空间将被释放。完成复制后，From 空间和 To 空间的角色发生对换。

Scavenge 的缺点是只能使用堆内存的一半，但 Scavenge 由于只复制存活的对象，并且对于生命周期短的场景存活对象只占少部分，所以它在时间效率上表现优异。Scavenge 是典型的牺牲空间换取时间的算法，无法大规模地应用到所有的垃圾回收中，但非常适合应用在新生代中。



对象是如何释放的呢？

有个叫可达性分析算法的概念，即通过一系列的称为“GC ROOT”的对象作为起始点。从这些节点开始向下搜索。搜索走过的路径称为引用链。当一个对象到GC ROOT没有任何引用链时，则证明此对象是不可用的。当然在虚拟机判断要被释放的对象里面，即使在可达性分析算法中不可达的对象，也并非是立即释放的。如果对象在进行可达性分析后发现没有与GC ROOTS相连接的引用链。将会对它进行一次标记，并进行刷选。它会放进一个队列中依次进行回收。如果这时又有对象引用到它，它就不会被回收。

晋升

对象从新生代中移动到老生代中的过程称为晋升。

From 空间中的存活对象在复制到 To 空间之前需要进行检查，在一定条件下，需要将存活周期长的对象移动到老生代中，也就是完成对象的晋升。

晋升条件主要有两个：

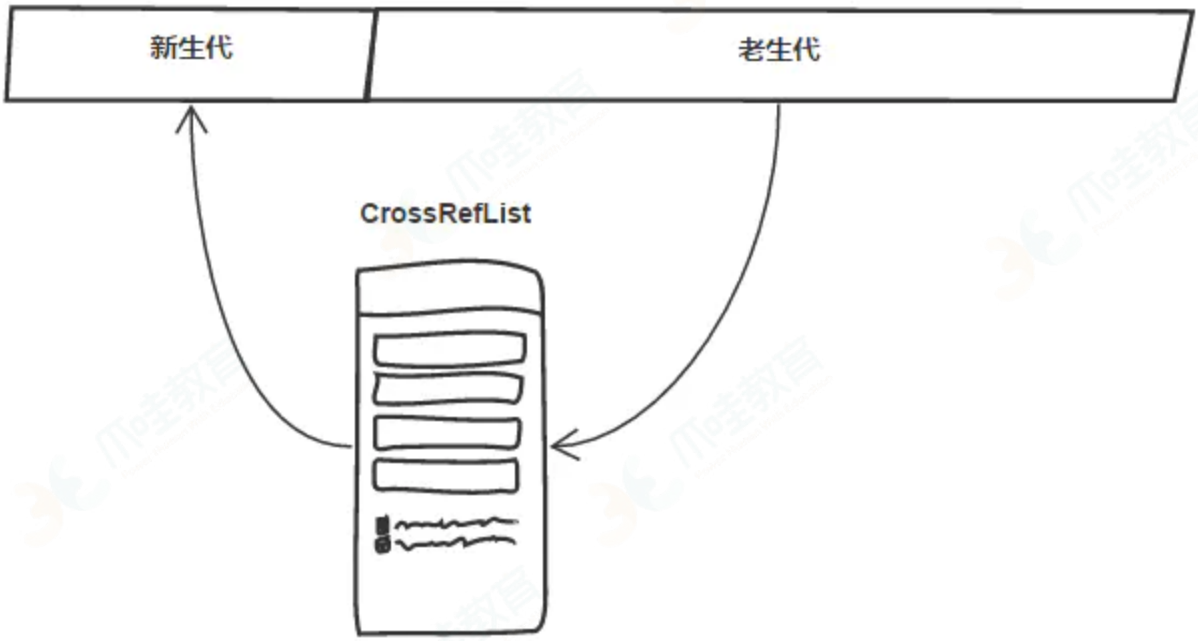
- 对象是否经历过一次 Scavenge 回收，是的话，则移动到老生代
- To 空间已经使用超过 25%，To 空间对象移动到老生代

设置 25% 这个限制值得原因是当这次 Scavenge 回收完成后，这个 To 空间将变成 From 空间，接下来的内存分配将在这个空间中进行，如果占比过高，会影响后续的内存分配。

写屏障

上面有一个细节被忽略了：如果新生区中某个对象，只有一个指向它的指针，而这个指针恰好是在老生区的对象当中，我们如何才能知道新生区中那个对象是活跃的呢？显然我们并不希望将老生区再遍历一次，因为老生区中的对象很多，这样做一次消耗太大。

为了解决这个问题，实际上在写缓冲区中有一个列表(我们称之为`CrossRefList`)，列表中记录了所有老生区对象指向新生区的情况。新对象诞生的时候，并不会有指向它的指针，而当有老生区中的对象出现指向新生区对象的指针时，我们便记录下来这样的跨区指向。由于这种记录行为总是发生在写操作时，它被称为`写屏障`——因为每个写操作都要经历这样一关。



老生代

老生代的内存空间较大且存活对象较多，因此其垃圾回收算法也就没有新生代那么简单了。为此V8使用了标记-清除算法 (Mark-Sweep)进行垃圾回收，并使用标记-压缩算法(Mark-Compact)整理内存碎片，提高内存的利用率。老生代的垃圾回收算法步骤如下：

```
1. 对老生代进行第一遍扫描，标记存活的对象
2. 对老生代进行第二次扫描，清除未被标记的对象
3. 将存活对象往内存的一端移动
4. 清除掉存活对象边界外的内存
```

Mark-Sweep

Mark-Sweep 是标记清除的意思，它分为两个阶段，标记和清理。Mark-Sweep 在标记阶段遍历堆中的所有对象，并标记活着的对象，在随后的清除阶段中，只清除未被标记的对象。

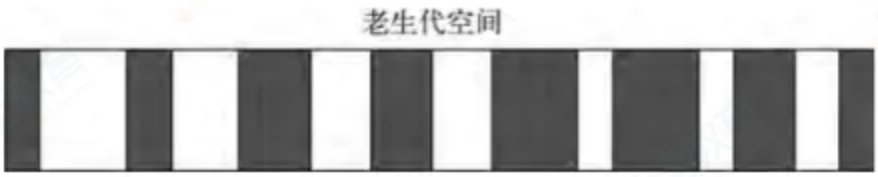


图5-6　Mark-Sweep在老生代空间中标记后的示意图

算法机制

在标记阶段，所有堆上的活跃对象都会被标记。每个页（注意，V8的内存页是1MB的连续内存块，与虚拟内存页不同）都会包含一个用来标记的位图，位图中的每一位对应页中的一字。这个标记非常有必要，因为指针可能会在任何字对齐的地方出现。显然，这样的位图要占据一定的空间（32位系统上占据3.1%，64位系统上占据1.6%），但所有的内存管理机制都需要这样占用，因此这种做法并不过分。除此之外，另有2位来表示标记对象的状态。由于对象至少有2字长，因此这些位不会重叠。

状态一共有三种：如果一个对象的状态为白，那么它尚未被垃圾回收器发现；如果一个对象的状态为灰，那么它已被垃圾回收器发现，但它的邻接对象仍未全部处理完毕；如果一个对象的状态为黑，则它不仅被垃圾回收器发现，而且其所有邻接对象也都处理完毕。

如果将堆中的对象看作由指针相互联系的有向图，标记算法的核心实际是深度优先搜索。在标记的初期，位图是空的，所有对象也都是白的。从根可达的对象会被染色为灰色，并被放入标记用的一个单独分配的双端队列。标记阶段的每次循环，GC会将一个对象从双端队列中取出，染色为黑，然后将它的邻接对象染色为灰，并把邻接对象放入双端队列。这一过程在双端队列为空且所有对象都变黑时结束。

特别大的对象，如长数组，可能会在处理时分片，以防溢出双端队列。如果双端队列溢出了，则对象仍然会被染为灰色，但不会再被放入队列（这样他们的邻接对象就没有机会再染色了）。因此当双端队列为空时，GC仍然需要扫描一次，确保所有的灰对象都成为了黑对象。对于未被染黑的灰对象，GC会将其再次放入队列，再度处理。

标记算法结束时，所有的活跃对象都被染为了黑色，而所有的死对象则仍是白的。这一结果正是清理和紧缩两个阶段所期望的。

清理阶段，清理算法扫描连续存放的死对象，将其变为空闲空间，并将其添加到空闲内存链表中。每一页都包含数个空闲内存链表，其分别代表小内存区（<256字）、中内存区（<2048字）、大内存区（<16384字）和超大内存区（其它更大的内存）。

清理算法非常简单，只需遍历页的位图，搜索连续的白对象。空闲内存链表大量被scavenge算法用于分配存活下来的活跃对象，但也被紧缩算法用于移动对象。有些类型的对象只能被分配在老生区，因此空闲内存链表也被它们使用。

Mark-Compact

Mark-Sweep 最大的问题是在进行一次标记清除回收后，内存空间会出现不连续的状态。这种内存碎片会对后续的内存分配造成问题，因为很可能出现需要分配一个大对象的情况，这时所有的碎片空间都无法完成此次分配，就会提前触发垃圾回收，而这次回收是不必要的。

为了解决 Mark-Sweep 的内存碎片问题，Mark-Compact被提出来。Mark-Compact是标记整理的意思，是在 Mark-Sweep的基础上演进而来的。它们的差别在于对象在标记为死亡后，在整理过程中，将活着的对象往一端移动，移动完成后，直接清理掉边界外的内存。

算法机制

紧缩算法会尝试将对象从碎片页（包含大量小空闲内存的页）中迁移整合在一起，来释放内存。这些对象会被迁移到另外的页上，因此也可能会新分配一些页。而迁出后的碎片页就可以返还给操作系统了。

迁移整合的过程非常复杂，大概过程是这样的。对目标碎片页中的每个活跃对象，在空闲内存链表中分配一块其它页的区域，将该对象复制至新页，并在碎片页中的该对象上写上转发地址。迁出过程中，对象中的旧地址会被记录下来，这样在迁出结束后V8会遍历它所记录的地址，将其更新为新的地址。由于标记过程中也记录了不同页之间的指针，此时也会更新这些指针的指向。注意，如果一个页非常“活跃”，比如其中有过多需要记录的指针，则地址记录会跳过它，等到下一轮垃圾回收再进行处理。