

目录

3.JavaScript 面试知识点总结.....	11
3.1. 介绍 js 的基本数据类型。.....	11
3.2. JavaScript 有几种类型的值？你能画一下他们的内存图吗？.....	11
3.3. 什么是堆？什么是栈？它们之间有什么区别和联系？.....	11
3. 4. 内部属性 [[Class]] 是什么？.....	12
3.5. 介绍 js 有哪些内置对象？.....	12
3.6. undefined 与 undeclared 的区别？.....	13
3.7. null 和 undefined 的区别？.....	14
3. 8. 如何获取安全的 undefined 值？.....	14
3. 9. 说几条写 JavaScript 的基本规范？.....	14
在平常项目开发中，我们遵守一些这样的基本规范，比如说：.....	14
3.10. JavaScript 原型，原型链？有什么特点？.....	14
3.11. js 获取原型的方法？.....	15
3.12. 在 js 中不同进制数字的表示方式.....	15
3.13. js 中整数的安全范围是多少？.....	16
3.14. typeof NaN 的结果是什么？.....	16
3.15. isNaN 和 Number.isNaN 函数的区别？.....	16
3.16. Array 构造函数只有一个参数值时的表现？.....	16
3.17. 其他值到字符串的转换规则？.....	17
3.18. 其他值到数字值的转换规则？.....	17
3.19. 其他值到布尔类型的值的转换规则？.....	18
3.20. {} 和 [] 的 valueOf 和 toString 的结果是什么？.....	18
3.21. 什么是假值对象？.....	18
3.22. ~ 操作符的作用？.....	18
3.23. 解析字符串中的数字和将字符串强制类型转换为数字的返回结果都是数字，它们之间的区别是什么？.....	18
3.24. + 操作符什么时候用于字符串的拼接？.....	19
3.25. 什么情况下会发生布尔值的隐式强制类型转换？.....	19
3.26. 和 && 操作符的返回值？.....	19
3.28. == 操作符的强制类型转换规则？.....	20
3.29. 如何将字符串转化为数字，例如 '12.3b'?.....	20
3.30. 如何将浮点数点左边的数每三位添加一个逗号，如 12000000.11 转化为 『12,000,000.11』？.....	20
3.31. 常用正则表达式.....	21
3.32. 生成随机数的各种方法？.....	21
3.33. 如何实现数组的随机排序？.....	21
3.34. javascript 创建对象的几种方式？.....	23
3.35. JavaScript 继承的几种实现方式？.....	24
3.36. 寄生式组合继承的实现？.....	24
3.37. Javascript 的作用域链？.....	25

3.38. 谈谈 This 对象的理解。	25
3.39. eval 是做什么的？	26
3.40. 什么是 DOM 和 BOM？	27
3.41. 写一个通用的事件监听器函数。	27
3.42. 事件是什么？IE 与火狐的事件机制有什么区别？如何阻止冒泡？	29
3.43. 三种事件模型是什么？	30
4.44. 事件委托是什么？	30
3.45. ["1", "2", "3"].map(parseInt) 答案是多少？	30
3.46. 什么是闭包，为什么要用它？	31
3.47. javascript 代码中的 "use strict"; 是什么意思？使用它区别是什么？	31
3.48. 如何判断一个对象是否属于某个类？	32
3.49. instanceof 的作用？	32
3.50. new 操作符具体干了什么呢？如何实现？	33
3.51. Javascript 中，有一个函数，执行时对象查找时，永远不会去查找原型，这个函数是？	34
3.52. 对于 JSON 的了解？	34
3.53. [].forEach.call(\$\$("")),function(a){a.style.outline="1px solid #"+(~(Math.random()*(1<<24))).toString(16)} 能解释一下这段代码的意思吗？	35
3.54. js 延迟加载的方式有哪些？	35
3.55. Ajax 是什么？如何创建一个 Ajax？	36
3.56. 谈一谈浏览器的缓存机制？	40
3.57. Ajax 解决浏览器缓存问题？	41
3.58. 同步和异步的区别？	42
3.59. 什么是浏览器的同源政策？	42
3.60. 如何解决跨域问题？	43
3.61. 服务器代理转发时，该如何处理 cookie？	45
3.62. 简单谈一下 cookie ？	45
3.63. 模块化开发怎么做？	45
3.64. js 的几种模块规范？	46
3.65. AMD 和 CMD 规范的区别？	46
3.66. ES6 模块与 CommonJS 模块、AMD、CMD 的差异。	47
3.67. requireJS 的核心原理是什么？（如何动态加载的？如何避免多次加载的？如何 缓存的？）	48
3.68. JS 模块加载器的轮子怎么造，也就是如何实现一个模块加载器？	48
3.69. ECMAScript6 怎么写 class，为什么会出现 class 这种东西？.....	48
3.70. document.write 和 innerHTML 的区别？	49
3.71. DOM 操作——怎样添加、移除、移动、复制、创建和查找节点？	49
(1) 创建新节点.....	49
(2) 添加、移除、替换、插入.....	49
(3) 查找.....	49
(4) 属性操作.....	49
3.72. innerHTML 与 outerHTML 的区别？	50
3.73. .call() 和 .apply() 的区别？	50
3.74. JavaScript 类数组对象的定义？	50

(1) 通过 call 调用数组的 slice 方法来实现转换.....	50
(2) 通过 call 调用数组的 splice 方法来实现转换.....	50
(3) 通过 apply 调用数组的 concat 方法来实现转换.....	51
(4) 通过 Array.from 方法来实现转换.....	51
3.75. 数组和对象有哪些原生方法，列举一下？	51
3.76. 数组的 fill 方法？	51
3.77. [,,] 的长度？	52
3.78. JavaScript 中的作用域与变量声明提升？	52
3.79. 如何编写高性能的 Javascript ?	52
3.80. 简单介绍一下 V8 引擎的垃圾回收机制.....	53
3.81. 哪些操作会造成内存泄漏？	54
3.82. 需求：实现一个页面操作不会整页刷新的网站，并且能在浏览器前进、后退时正确响应。给出你的技术实现方案？	55
3.83. 如何判断当前脚本运行在浏览器还是 node 环境中？（阿里）	55
3.84. 把 script 标签放在页面的最底部的 body 封闭之前和封闭之后有什么区别？浏览器会如何解析它们？	55
3.85. 移动端的点击事件的有延迟，时间是多久，为什么会有？ 怎么解决这个延时？	56
3.86. 什么是“前端路由”？什么时候适合使用“前端路由”？“前端路由”有哪些优点和缺点？	56
(1) 什么是前端路由？	56
(2) 什么时候使用前端路由？	56
(3) 前端路由有什么优点和缺点？	57
3.87. 如何测试前端代码么？ 知道 BDD, TDD, Unit Test 么？ 知道怎么测试你的前端工程么(mocha, sinon, jasmin, qUnit..)？	57
3.88. 检测浏览器版本版本有哪些方式？	57
3.89. 什么是 Polyfill ？	58
3.90. 使用 JS 实现获取文件扩展名？	58
3.91. 介绍一下 js 的节流与防抖？	58
3.92. Object.is() 与原来的比较操作符 “==”、“==” 的区别？	60
3.93. escape,encodeURI,encodeURIComponent 有什么区别？	61
3.94. Unicode 和 UTF-8 之间的关系？	61
3.95. js 的事件循环是什么？	62
3.96. js 中的深浅拷贝实现？	62
3.97. 手写 call、apply 及 bind 函数.....	64
3.98. 函数柯里化的实现.....	68
3.99. 为什么 0.1 + 0.2 != 0.3？如何解决这个问题？	70
3.100. 原码、反码和补码的介绍.....	70
3.101. toPrecision 和 toFixed 和 Math.round 的区别？	71
3.102. 什么是 XSS 攻击？如何防范 XSS 攻击？	71
3.103. 什么是 CSP？	72
3.104. 什么是 CSRF 攻击？如何防范 CSRF 攻击？	73
CSRF 攻击指的是跨站请求伪造攻击，攻击者诱导用户进入一个第三方网站，然后该网站向被攻击网站发送跨站请求。如果用户在被.....	73
3.105. 什么是 Samesite Cookie 属性？	74
3.106. 什么是点击劫持？如何防范点击劫持？	75

3.107. SQL 注入攻击？	75
3.108. 什么是 MVVM？比之 MVC 有什么区别？什么又是 MVP ？	75
3.109. vue 双向数据绑定原理？	77
3.110. Object.defineProperty 介绍？	77
3.111. 使用 Object.defineProperty() 来进行数据劫持有什么缺点？	78
3.112. 什么是 Virtual DOM？为什么 Virtual DOM 比原生 DOM 快？	78
3.113. 如何比较两个 DOM 树的差异？	79
3.114. 什么是 requestAnimationFrame ?	79
3.115. 谈谈你对 webpack 的看法.....	79
3.116. offsetWidth/offsetHeight,clientWidth/clientHeight 与 scrollWidth/scrollHeight 的区别？	80
3.117. 谈一谈你理解的函数式编程？	81
3.118. 异步编程的实现方式？	81
3.119. Js 动画与 CSS 动画区别及相应实现.....	83
3.120. get 请求传参长度的误区.....	83
3.121. URL 和 URI 的区别？	84
3.122. get 和 post 请求在缓存方面的区别.....	84
3.123. 图片的懒加载和预加载.....	85
3.124. mouseover 和 mouseenter 的区别？	86
3.125. js 拖拽功能的实现.....	86
3.126. 为什么使用 setTimeout 实现 setInterval？怎么模拟？	87
3.127. let 和 const 的注意点？	88
3.128. 什么是 rest 参数？	88
3.129. 什么是尾调用，使用尾调用有什么好处？	88
3.130. Symbol 类型的注意点？	89
3.131. Set 和 WeakSet 结构？	89
3.132. Map 和 WeakMap 结构？	90
3.133. 什么是 Proxy ？	90
3.134. Reflect 对象创建目的？	90
3.135. require 模块引入的查找方式？	91
3.136. 什么是 Promise 对象，什么是 Promises/A+ 规范？	92
3.137. 手写一个 Promise.....	92
3.138. 如何检测浏览器所支持的最小字体大小？	96
3.139. 怎么做 JS 代码 Error 统计？	96
3.140. 单例模式模式是什么？	97
3.141. 策略模式是什么？	97
3.142. 代理模式是什么？	97
4.143. 中介者模式是什么？	97
4.144. 适配器模式是什么？	97
3.145. 观察者模式和发布订阅模式有什么不同？	97
3.146. Vue 的生命周期是什么？	98
3.147. Vue 的各个生命阶段是什么？	98
3.148. Vue 组件间的参数传递方式？	99
(1) 父子组件间通信.....	99

(2) 兄弟组件间通信.....	100
(3) 任意组件之间.....	100
3.149. computed 和 watch 的差异?	100
3.150. vue-router 中的导航钩子函数.....	101
3.151. \$route 和 \$router 的区别?	101
3.152. vue 常用的修饰符?	101
3.153. vue 中 key 值的作用?	102
3.154. computed 和 watch 区别?	102
3.155. keep-alive 组件有什么作用?	102
3.156. vue 中 mixin 和 mixins 区别?	102
3.157. 开发中常用的几种 Content-Type ?	103
3.158. 如何封装一个 javascript 的类型判断函数?	104
3.159. 如何判断一个对象是否为空对象?	104
3.160. 使用闭包实现每隔一秒打印 1,2,3,4.....	105
3.161. 手写一个 jsonp.....	105
3.162. 手写一个观察者模式?	107
3.163. EventEmitter 实现.....	109
3.164. 一道常被人轻视的前端 JS 面试题.....	110
3.165. 如何确定页面的可用性时间, 什么是 Performance API?	111
3.166. js 中的命名规则.....	112
3.167. js 语句末尾分号是否可以省略?	112
3.168. Object.assign().....	112
3.169. Math.ceil 和 Math.floor.....	112
3.170. js for 循环注意点.....	112
3.171. 一个列表, 假设有 100000 个数据, 这个该怎么办?	113
3.172. js 中倒计时的纠偏实现?	113
3.173. 进程间通信的方式?	114
3.174. 如何查找一篇英文文章中出现频率最高的单词?	114
4. 算法知识总结.....	115
4.1 常用算法和数据结构总结.....	116
4.1.1 排序.....	116
4.1.2 树.....	137
4.1.3 链表.....	157
4.1.4 动态规划.....	158
4.1.3 经典笔试题.....	161
(1).js 实现一个函数, 完成超过范围的两个大整数相加功能.....	161
(2).js 如何实现数组扁平化?	162
(3).js 如何实现数组去重?	163
(4). 如何求数组的最大值和最小值?	164
(5). 如何求两个数的最大公约数?	164
(6). 如何求两个数的最小公倍数?	165
(7). 实现 IndexOf 方法?	165
(8). 判断一个字符串是否为回文字符串?	165
(9). 实现一个累加函数的功能比如 sum(1,2,3)(2).valueOf().....	166

(10). 使用 reduce 方法实现 forEach、map、filter.....	166
(11). 设计一个简单的任务队列，要求分别在 1,3,4 秒后打印出 "1", "2", "3"	168
(12). 如何查找一篇英文文章中出现频率最高的单词？	169
4.2 常见面试智力题总结.....	170
4.3 剑指 offer 思路总结.....	176
4.3.1. 二维数组中的查找.....	176
4.3.2. 替换空格.....	177
4.3.3. 从尾到头打印链表.....	177
4.3.4. 重建二叉树.....	178
4.3.5. 用两个栈实现队列.....	179
4.3.6. 旋转数组的最小数字.....	179
4.3.7. 斐波那契数列.....	180
4.3.8. 跳台阶.....	180
4.3.9. 变态跳台阶.....	181
4.3.10. 矩形覆盖.....	182
4.3.11. 二进制中 1 的个数.....	182
4.3.12. 数值的整数次方.....	183
4.3.13. 调整数组顺序使奇数位于偶数前面.....	183
4.3.14. 链表中倒数第 k 个节点.....	184
4.3.15. 反转链表.....	184
4.3.16. 合并两个排序的链表.....	185
4.3.17. 树的子结构.....	185
4.3.18. 二叉树的镜像.....	186
4.3.19. 顺时针打印矩阵.....	186
4.3.20. 定义一个栈，实现 min 函数.....	187
4.3.21. 栈的压入弹出.....	187
4.3.22. 从上往下打印二叉树.....	188
4.3.23. 二叉搜索树的后序遍历.....	188
4.3.24. 二叉树中和为某一值路径.....	189
4.3.25. 复杂链表的复制.....	189
4.3.26. 二叉搜索树与双向链表.....	190
4.3.27. 字符串的排列.....	191
4.3.28. 数组中出现次数超过一半的数字.....	192
4.3.29. 最小的 K 个数.....	193
4.3.30. 连续子数组的最大和.....	194
4.3.31. 整数中 1 出现的次数（待深入理解）	195
4.3.32. 把数组排成最小的数.....	196
4.3.33. 丑数（待深入理解）	196
4.3.34. 第一个只出现一次的字符.....	197
4.3.35. 数组中的逆序对.....	197
4.3.36. 两个链表的第一个公共结点.....	198
4.3.37. 数字在排序数组中出现的次数.....	199
4.3.38. 二叉树的深度.....	200
4.3.39. 平衡二叉树.....	201

4.3.40. 数组中只出现一次的数字.....	201
4.3.41. 和为 S 的连续正数序列.....	202
4.3.42. 和为 S 的两个数字.....	203
4.3.43. 左旋转字符串.....	204
4.3.44. 翻转单词顺序列.....	204
4.3.45. 扑克牌的顺子.....	205
4.3.46. 圆圈中最后剩下的数字（约瑟夫环问题）.....	206
4.3.47. $1+2+3+\dots+n$	206
4.3.48. 不用加减乘除做加法.....	207
4.3.49. 把字符串转换成整数。.....	207
4.3.50. 数组中重复的数字.....	208
4.3.51. 构建乘积数组.....	208
4.3.52. 正则表达式的匹配.....	209
4.3.53. 表示数值的字符串.....	210
4.3.54. 字符流中第一个不重复的字符.....	210
4.3.55. 链表中环的入口结点.....	211
4.3.56. 删除链表中重复的结点.....	211
4.3.57. 二叉树的下一个结点.....	212
4.3.58. 对称二叉树.....	213
4.3.59. 按之字形顺序打印二叉树（待深入理解）.....	214
4.3.60. 从上到下按层打印二叉树，同一层结点从左至右输出。每一层输出一行。.....	214
4.3.61. 序列化二叉树（待深入理解）.....	215
4.3.62. 二叉搜索树的第 K 个节点.....	215
4.3.63. 数据流中的中位数（待深入理解）.....	215
4.3.64. 滑动窗口中的最大值（待深入理解）.....	216
4.3.65. 矩阵中的路径（待深入理解）.....	216
4.3.66. 机器人的运动范围（待深入理解）.....	217
4.3.67 相关算法题.....	217
(1). 明星问题.....	217
(2). 正负数组求和.....	218
5.计算机网络知识总结.....	218
5.1 应用层.....	219
5.1.1HTTP 协议.....	219
(1)概况.....	219
(2)HTTP 请求报文.....	219
(3)HTTP 响应报文.....	220
(4)首部行.....	221
(5) HTTP/1.1 协议缺点.....	222
5.1.2 HTTP/2 协议.....	222
(1) 二进制协议.....	223
(2) 多路复用.....	223
(3) 数据流.....	223
(4) 头信息压缩.....	223
(5) 服务器推送.....	224

(6) HTTP/2 协议缺点.....	224
(7) HTTP/3 协议.....	224
5.1.3 HTTPS 协议.....	225
(1) HTTP 存在的问题.....	225
(2) HTTPS 简介.....	225
(3) TLS 握手过程.....	225
(4) 实现原理.....	226
5.1.4 DNS 协议.....	227
(1) 概况.....	227
(2) 域名的层级结构.....	228
(3) 查询过程.....	228
(4) DNS 记录和报文.....	229
(5) 递归查询和迭代查询.....	230
(6) DNS 缓存.....	230
(7) DNS 实现负载平衡.....	231
5.2 传输层.....	231
5.2.1 多路复用与多路分解.....	231
5.2.2 UDP 协议.....	232
(1) UDP 报文段结构.....	233
5.2.3 TCP 协议.....	233
(1) TCP 报文段结构.....	234
(2) TCP 三次握手的过程.....	235
(3) TCP 四次挥手的过程.....	236
(4) ARQ 协议.....	237
(5) TCP 的可靠运输机制.....	238
(6) TCP 的流量控制机制.....	239
(7) TCP 的拥塞控制机制.....	239
5.2.4 网络层.....	240
5.2.5 数据链路层.....	241
5.2.6 物理层.....	241
5.3 常考面试题.....	241
5.3.1. Post 和 Get 的区别？	241
5.3.2. TLS/SSL 中什么一定要用三个随机数，来生成"会话密钥"？	242
5.3.3. SSL 连接断开后如何恢复？	242
5.3.4. RSA 算法的安全性保障？	243
5.3.5. DNS 为什么使用 UDP 协议作为传输层协议？	243
5.3.6. 当你在浏览器中输入 Google.com 并且按下回车之后发生了什么？	244
5.3.7. 谈谈 CDN 服务？	246
5.3.8. 什么是正向代理和反向代理？	246
5.3.9. 负载平衡的两种实现方式？	247
5.3.10. http 请求方法 options 方法有什么用？	247
5.3.11. http1.1 和 http1.0 之间有哪些区别？	248
5.3.12. 网站域名加 www 与不加 www 的区别？	249
5.3.13. 即时通讯的实现，短轮询、长轮询、SSE 和 WebSocket 间的区别？	249

5.3.14. 怎么实现多个网站之间共享登录状态.....	250
6.常用工具知识总结.....	251
6.1. git 与 svn 的区别在哪里？	251
6.2. 经常使用的 git 命令？	251
6.3. git pull 和 git fetch 的区别.....	252
6.4. git rebase 和 git merge 的区别.....	252
7.面试记录总结.....	252
7.1.阿里巴巴（获得 OFFER）	252
7.1.1. 2019-3-25 阿里巴巴（淘宝）一面.....	252
7.1.2. 2019-3-28 阿里巴巴（淘宝）二面.....	253
7.1.3. 2019-4-1 阿里巴巴（淘宝）三面.....	254
7.1.4. 2019-4-3 阿里巴巴（淘宝）四面（hr）	254
7.1.5. 2019-4-29 阿里巴巴（阿里云）一面.....	255
7.1.6. 2019-5-27 阿里巴巴（阿里云）二面.....	255
7.1.7. 2019-5-29 阿里巴巴（淘宝二轮）一面.....	255
7.1.8. 2019-5-31 阿里巴巴（淘宝二轮）二面.....	256
7.1.9. 2019-5-31 阿里巴巴（淘宝二轮）三面（hr）	256
7.2 腾讯（获得 OFFER）	256
7.2.1. 2019-4-26 腾讯（TEG）一面.....	257
7.2.2. 2019-4-29 腾讯（TEG）二面.....	257
7.2.3. 2019-5-9 腾讯（TEG）三面.....	258
7.2.4. 2019-5-17 腾讯（TEG）四面（hr）	258
7.3 网易互娱（获得 OFFER）	258
7.3.1. 2019-4-15 网易互娱一面.....	259
7.3.2. 2019-4-18 网易互娱二面.....	259
7.4 字节跳动.....	259
7.4.1. 2019-3-23 字节跳动一面.....	259
7.5 微众银行.....	260
7.5.1. 2019-4-1 微众银行一面.....	260
7.6 酷家乐（获得 OFFER）	260
7.6.1. 2019-4-18 酷家乐一面.....	261
7.6.2. 2019-4-22 酷家乐二面.....	261
7.6.3. 2019-4-25 酷家乐三面.....	262
7.7 京东.....	262
7.7.1. 2019-4-22 京东一面.....	262
7.8 亿联网络.....	263
7.8.1. 2019-4-24 亿联网络一面.....	263
7.9 OPPO（获得 OFFER）	263
7.9.1. 2019-4-26 OPPO 一面.....	263
7.9.2. 2019-5-8 OPPO 二面（hr）	264
7.10 华为（获得 OFFER）	264
7.10.1. 2019-4-28 华为一面.....	264
7.10.2. 2019-4-28 华为二面.....	264

3. JavaScript 面试知识点总结

3.1. 介绍 js 的基本数据类型。

js 一共有六种基本数据类型，分别是 `Undefined`、`Null`、`Boolean`、`Number`、`String`，还有在 ES6 中新增的 `Symbol` 类型，代表创建后独一无二且不可变的数据类型，它的出现我认为主要是为了解决可能出现的全局变量冲突的问题。

3.2. JavaScript 有几种类型的值？你能画一下他们的内存图吗？

涉及知识点：

栈：原始数据类型（`Undefined`、`Null`、`Boolean`、`Number`、`String`）

堆：引用数据类型（对象、数组和函数）

两种类型的区别是：存储位置不同。原始数据类型直接存储在栈（`stack`）中的简单数据段，占据空间小、大小固定，属于被频繁使用数据，所以放入栈中存储。引用数据类型存储在堆（`heap`）中的对象，占据空间大、大小不固定。如果存储在栈中，将会影响程序运行的性能；引用数据类型在栈中存储了指针，该指针指向堆中该实体的起始地址。当解释器寻找引用值时，会首先检索其在栈中的地址，取得地址后从堆中获得实体。

回答：

js 可以分为两种类型的值，一种是基本数据类型，一种是复杂数据类型。

基本数据类型.... (参考 1)

复杂数据类型指的是 `Object` 类型，所有其他的如 `Array`、`Date` 等数据类型都可以理解为 `Object` 类型的子类。两种类型间的主要区别是它们的存储位置不同，基本数据类型的值直接保存在栈中，而复杂数据类型的值保存在堆中，通过使用在栈中保存对应的指针来获取堆中的值。

详细资料可以参考：[《JavaScript 有几种类型的值？》](#) [《JavaScript 有几种类型的值？能否画一下它们的内存图？》](#)

3.3. 什么是堆？什么是栈？它们之间有什么区别和联系？

堆和栈的概念存在于数据结构中和操作系统内存中。在数据结构中，栈中数据的存取方式为先进后出。而堆是一个优先队列，是按优先级来进行排序的，优先级可以按照大小来规定。完全二叉树是堆的一种实现方式。在操作系统中，内存被分为栈区和堆区。栈区内由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。堆区内由程序员分配释放，若程序员不释放，程序结束时可能由垃圾回收机制回收。

详细资料可以参考： [《什么是堆？什么是栈？他们之间有什么区别和联系？》](#)

3.4. 内部属性 `[[Class]]` 是什么？

所有 `typeof` 返回值为 "object" 的对象（如数组）都包含一个内部属性 `[[Class]]`（我们可以把它看作一个内部的分类，而非传统的面向对象意义上的类）。这个属性无法直接访问，一般通过 `Object.prototype.toString(..)` 来查看。例如：

```
Object.prototype.toString.call( [1,2,3] );
// "[object Array]

Object.prototype.toString.call( /regex-literal/i );
// "[object RegExp]"
```

3.5. 介绍 js 有哪些内置对象？

涉及知识点：

全局的对象（`global objects`）或称标准内置对象，不要和“全局对象（`global object`）”混淆。这里说的全局的对象是说在全局作用域里的对象。全局作用域中的其他对象可以由用户的脚本创建或由宿主程序提供。

标准内置对象的分类

（1）值属性，这些全局属性返回一个简单值，这些值没有自己的属性和方法。例如 `Infinity`、`NaN`、`undefined`、`null` 字面量

（2）函数属性，全局函数可以直接调用，不需要在调用时指定所属对象，执行结束后会将结果直接返回给调用者。例如 `eval()`、`parseFloat()`、`parseInt()` 等

（3）基本对象，基本对象是定义或使用其他对象的基础。基本对象包括一般对象、函数对象和错误对象。例如 `Object`、`Function`、`Boolean`、`Symbol`、`Error` 等

（4）数字和日期对象，用来表示数字、日期和执行数学计算的对象。例如 `Number`、`Math`、`Date`

(5) 字符串，用来表示和操作字符串的对象。例如 `String`、`RegExp`

(6) 可索引的集合对象，这些对象表示按照索引值来排序的数据集合，包括数组和类型数组，以及类数组结构的对象。例如 `Array`

(7) 使用键的集合对象，这些集合对象在存储数据时会使用到键，支持按照插入顺序来迭代元素。例如 `Map`、`Set`、`WeakMap`、`WeakSet`

(8) 矢量集合，`SIMD` 矢量集合中的数据会被组织为一个数据序列。例如 `SIMD` 等

(9) 结构化数据，这些对象用来表示和操作结构化的缓冲区数据，或使用 `JSON` 编码的数据。例如 `JSON` 等

(10) 控制抽象对象例如 `Promise`、`Generator` 等

(11) 反射，例如 `Reflect`、`Proxy`

(12) 国际化，为了支持多语言处理而加入 `ECMAScript` 的对象。例如 `Intl`、`Intl.Collator` 等

(13) `WebAssembly`

(14) 其他

例如 `arguments`

回答：

`js` 中的内置对象主要指的是在程序执行前存在全局作用域里的由 `js` 定义的一些全局值属性、函数和用来实例化其他对象的构造函数对象。一般我们经常用到的如全局变量值 `Nan`、`undefined`，全局函数如 `parseInt()`、`parseFloat()` 用来实例化对象的构造函数如 `Date`、`Object` 等，还有提供数学计算的单体内置对象如 `Math` 对象。

详细资料可以参考： [《标准内置对象的分类》](#) [《JS 所有内置对象属性和方法汇总》](#)

3.6. `undefined` 与 `undeclared` 的区别？

已在作用域中声明但还没有赋值的变量，是 `undefined` 的。相反，还没有在作用域中声明过的变量，是 `undeclared` 的。对于 `undeclared` 变量的引用，浏览器会报引用错误，如 `ReferenceError: b is not defined`。但是我们可以使用 `typeof` 的安全防范机制来避免报错，因为对于 `undeclared`（或者 `not defined`）变量，`typeof` 会返回 "`undefined`"。

3.7. null 和 undefined 的区别？

首先 `Undefined` 和 `Null` 都是基本数据类型，这两个基本数据类型分别都只有一个值，就是 `undefined` 和 `null`。`undefined` 代表的含义是未定义，`null` 代表的含义是空对象。一般变量声明了但还没有定义的时候会返回 `undefined`，`null` 主要用于赋值给一些可能会返回对象的变量，作为初始化。`undefined` 在 js 中不是一个保留字，这意味着我们可以使用 `undefined` 来作为一个变量名，这样的做法是非常危险的，它会影响我们对 `undefined` 值的判断。但是我们可以通过一些方法获得安全的 `undefined` 值，比如说 `void 0`。当我们对两种类型使用 `typeof` 进行判断的时候，`Null` 类型化会返回 “object”，这是一个历史遗留的问题。当我们使用双等号对两种类型的值进行比较时会返回 `true`，使用三个等号时会返回 `false`。

详细资料可以参考： [《JavaScript 深入理解之 undefined 与 null》](#)



3.8. 如何获取安全的 `undefined` 值？

因为 `undefined` 是一个标识符，所以可以被当作变量来使用和赋值，但是这样会影响 `undefined` 的正常判断。表达式 `void __` 没有返回值，因此返回结果是 `undefined`。`void` 并不改变表达式的结果，只是让表达式不返回值。按惯例我们用 `void 0` 来获得 `undefined`。

3.9. 说几条写 JavaScript 的基本规范？

在平常项目开发中，我们遵守一些这样的基本规范，比如说：

- (1) 一个函数作用域中所有的变量声明应该尽量提到函数首部，用一个 `var` 声明，不允许出现两个连续的 `var` 声明，声明时如果变量没有值，应该给该变量赋值对应类型的初始值，便于他人阅读代码时，能够一目了然的知道变量对应的类型值。
- (2) 代码中出现地址、时间等字符串时需要使用常量代替。
- (3) 在进行比较的时候吧，尽量使用 '`====`'，'`!==`' 代替 '`==`'，'`!=`'。
- (4) 不要在内置对象的原型上添加方法，如 `Array`, `Date`。
- (5) `switch` 语句必须带有 `default` 分支。
- (6) `for` 循环必须使用大括号。
- (7) `if` 语句必须使用大括号。

3.10. JavaScript 原型，原型链？有什么特点？

在 js 中我们是使用构造函数来新建一个对象的，每一个构造函数的内部都有一个 `prototype` 属性值，这个属性值是一个对象，这个对象包含了可以由该构造函数的所有实例共享的属性和方法。当我们使用构造函数新建一个对象后，在这个对象的内部将包含一个指针，这个指针指向构造函数的 `prototype` 属性对应的值，在 ES5 中这个指针被称为对象的原型。一般来说我们是不应该能够获取到这个值的，但是现在浏览器中都实现了 `__proto__` 属性来让我们访问这个属性，但是我们最好不要使用这个属性，因为它不是规范中规定的。ES5 中新增了一个 `Object.getPrototypeOf()` 方法，我们可以通过这个方法来获取对象的原型。当我们访问一个对象的属性时，如果这个对象内部不存在这个属性，那么它就会去它的原型对象里找这个属性，这个原型对象又会有自己的原型，于是就这样一直找下去，也就是原型链的概念。原型链的尽头一般来说都是 `Object.prototype` 所以这就是我们新建的对象为什么能够使用 `toString()` 等方法的原因。

特点：

JavaScript 对象是通过引用来传递的，我们创建的每个新对象实体中并没有一份属于自己的原型副本。当我们修改原型时，与之相关的对象也会继承这一改变。

详细资料可以参考： [《JavaScript 深入理解之原型与原型链》](#)

3.11. js 获取原型的方法？

- `p.__proto__`
- `p.constructor.prototype`
- `Object.getPrototypeOf(p)`

3.12. 在 js 中不同进制数字的表示方式

-

以 `0X`、`0x` 开头的表示为十六进制。

-

-

以 `0`、`0O`、`0o` 开头的表示为八进制。

-

以 0B、0b 开头的表示为二进制格式。

•

3.13. js 中整数的安全范围是多少？

安全整数指的是，在这个范围内的整数转化为二进制存储的时候不会出现精度丢失，能够被“安全”呈现的最大整数是 $2^{53} - 1$ ，即 9007199254740991，在 ES6 中被定义为 Number.MAX_SAFE_INTEGER。最小整数是 -9007199254740991，在 ES6 中被定义为 Number.MIN_SAFE_INTEGER。

如果某次计算的结果得到了一个超过 JavaScript 数值范围的值，那么这个值会被自动转换为特殊的 Infinity 值。如果某次计算返回了正或负的 Infinity 值，那么该值将无法参与下一次的计算。判断一个数是不是有穷的，可以使用 isFinite 函数来判断。

3.14. typeof NaN 的结果是什么？

NaN 意指“不是一个数字”（not a number），NaN 是一个“警戒值”（sentinel value，有特殊用途的常规值），用于指出数字类型中的错误情况，即“执行数学运算没有成功，这是失败后返回的结果”。

```
typeof NaN; // "number"
```

NaN 是一个特殊值，它和自身不相等，是唯一一个非自反（自反，reflexive，即 $x === x$ 不成立）的值。而 $\text{NaN} \neq \text{NaN}$ 为 true。

3.15. isNaN 和 Number.isNaN 函数的区别？

函数 isNaN 接收参数后，会尝试将这个参数转换为数值，任何不能被转换为数值的值都会返回 true，因此非数字值传入也会返回 true，会影响 NaN 的判断。

函数 Number.isNaN 会首先判断传入参数是否为数字，如果是数字再继续判断是否为 NaN，这种方法对于 NaN 的判断更为准确。

3.16. Array 构造函数只有一个参数值时的表现？

Array 构造函数只带一个数字参数的时候，该参数会被作为数组的预设长度（length），而非只充当数组中的一个元素。这样创建出来的只是一个空数组，只不过它的 length 属性被

设置成了指定的值。构造函数 `Array(..)` 不要求必须带 `new` 关键字。不带时，它会被自动补上。

3.17. 其他值到字符串的转换规则？

规范的 9.8 节中定义了抽象操作 `ToString`，它负责处理非字符串到字符串的强制类型转换。

- (1) `Null` 和 `Undefined` 类型，`null` 转换为 "`null`"，`undefined` 转换为 "`undefined`"，
- (2) `Boolean` 类型，`true` 转换为 "`true`"，`false` 转换为 "`false`"。
- (3) `Number` 类型的值直接转换，不过那些极小和极大的数字会使用指数形式。
- (4) `Symbol` 类型的值直接转换，但是只允许显式强制类型转换，使用隐式强制类型转换会产生错误。
- (3) 对普通对象来说，除非自行定义 `toString()` 方法，否则会调用 `toString()` (`Object.prototype.toString()`) 来返回内部属性 `[[Class]]` 的值，如"`[object Object]`"。如果对象有自己的 `toString()` 方法，字符串化时就会调用该方法并使用其返回值。

3.18. 其他值到数字值的转换规则？

有时我们需要将非数字值当作数字来使用，比如数学运算。为此 ES5 规范在 9.3 节定义了抽象操作 `ToNumber`。

- (1) `Undefined` 类型的值转换为 `Nan`。
- (2) `Null` 类型的值转换为 `0`。
- (3) `Boolean` 类型的值，`true` 转换为 `1`，`false` 转换为 `0`。
- (4) `String` 类型的值转换如同使用 `Number()` 函数进行转换，如果包含非数字值则转换为 `Nan`，空字符串为 `0`。
- (5) `Symbol` 类型的值不能转换为数字，会报错。
- (6) 对象（包括数组）会首先被转换为相应的基本类型值，如果返回的是非数字的基本类型值，则再遵循以上规则将其强制转换为数字。为了将值转换为相应的基本类型值，抽象操作 `ToPrimitive` 会首先（通过内部操作 `DefaultValue`）检查该值是否有 `valueOf()` 方法。如果有并且返回基本类型值，就使用该值进行强制类型转换。如果没有就使用 `toString()` 的返回值（如果存在）来进行强制类型转换。如果 `valueOf()` 和 `toString()` 均不返回基本类型值，会产生 `TypeError` 错误。

3.19. 其他值到布尔类型的值的转换规则？

ES5 规范 9.2 节中定义了抽象操作 `ToBoolean`，列举了布尔强制类型转换所有可能出现的结果。

以下这些是假值：

- `undefined`
- `null`
- `false`
- `+0、-0 和 NaN`
- `""`

假值的布尔强制类型转换结果为 `false`。从逻辑上说，假值列表以外的都应该是真值。

3.20. {} 和 [] 的 `valueOf` 和 `toString` 的结果是什么？

`{}` 的 `valueOf` 结果为 `{}`，`toString` 的结果为 "`[object Object]`"

`[]` 的 `valueOf` 结果为 `[]`，`toString` 的结果为 "`"`"

3.21. 什么是假值对象？

浏览器在某些特定情况下，在常规 JavaScript 语法基础上自己创建了一些外来值，这些就是“假值对象”。假值对象看起来和普通对象并无二致（都有属性，等等），但将它们强制类型转换为布尔值时结果为 `false` 最常见的例子是 `document.all`，它是一个类数组对象，包含了页面上的所有元素，由 DOM（而不是 JavaScript 引擎）提供给 JavaScript 程序使用。

3.22. ~ 操作符的作用？

`~` 返回 2 的补码，并且 `~` 会将数字转换为 32 位整数，因此我们可以使用 `~` 来进行取整操作。

`~x` 大致等同于 `-(x+1)`。

3.23. 解析字符串中的数字和将字符串强制类型转换为数字的返回结果都是数字，它们之间的区别是什么？

解析允许字符串（如 `parseInt()`）中含有非数字字符，解析按从左到右的顺序，如果遇到非数字字符就停止。而转换（如 `Number()`）不允许出现非数字字符，否则会失败并返回 `NaN`。

3.24. + 操作符什么时候用于字符串的拼接？

根据 ES5 规范 11.6.1 节，如果某个操作数是字符串或者能够通过以下步骤转换为字符串的话，`+` 将进行拼接操作。如果其中一个操作数是对象（包括数组），则首先对其调用 `ToPrimitive` 抽象操作，该抽象操作再调用 `[[DefaultValue]]`，以数字作为上下文。如果不能转换为字符串，则会将其转换为数字类型来进行计算。简单来说就是，如果 `+` 的其中一个操作数是字符串（或者通过以上步骤最终得到字符串），则执行字符串拼接，否则执行数字加法。那么对于除了加法的运算符来说，只要其中一方是数字，那么另一方就会被转为数字。

3.25. 什么情况下会发生布尔值的隐式强制类型转换？

- (1) `if (..)` 语句中的条件判断表达式。
- (2) `for (.. ; .. ; ..)` 语句中的条件判断表达式（第二个）。
- (3) `while (..)` 和 `do..while(..)` 循环中的条件判断表达式。
- (4) `? :` 中的条件判断表达式。
- (5) 逻辑运算符 `||`（逻辑或）和 `&&`（逻辑与）左边的操作数（作为条件判断表达式）。

3.26. || 和 && 操作符的返回值？

`||` 和 `&&` 首先会对第一个操作数执行条件判断，如果其不是布尔值就先进行 `ToBoolean` 强制类型转换，然后再执行条件判断。

对于 `||` 来说，如果条件判断结果为 `true` 就返回第一个操作数的值，如果为 `false` 就返回第二个操作数的值。

`&&` 则相反，如果条件判断结果为 `true` 就返回第二个操作数的值，如果为 `false` 就返回第一个操作数的值。

`||` 和 `&&` 返回它们其中一个操作数的值，而非条件判断的结果。

3.27. Symbol 值的强制类型转换？

ES6 允许从符号到字符串的显式强制类型转换，然而隐式强制类型转换会产生错误。

`Symbol` 值不能够被强制类型转换为数字（显式和隐式都会产生错误），但可以被强制类型转换为布尔值（显式和隐式结果都是 `true`）。

3.28. == 操作符的强制类型转换规则？

- (1) 字符串和数字之间的相等比较，将字符串转换为数字之后再进行比较。
- (2) 其他类型和布尔类型之间的相等比较，先将布尔值转换为数字后，再应用其他规则进行比较。
- (3) `null` 和 `undefined` 之间的相等比较，结果为真。其他值和它们进行比较都返回假值。
- (4) 对象和非对象之间的相等比较，对象先调用 `ToPrimitive` 抽象操作后，再进行比较。
- (5) 如果一个操作值为 `Nan`，则相等比较返回 `false`（`Nan` 本身也不等于 `Nan`）。
- (6) 如果两个操作值都是对象，则比较它们是不是指向同一个对象。如果两个操作数都指向同一个对象，则相等操作符返回 `true`，否则，返回 `false`。

详细资料可以参考： [《JavaScript 字符串间的比较》](#)

3.29. 如何将字符串转化为数字，例如 '12.3b'？

- (1) 使用 `Number()` 方法，前提是所包含的字符串不包含不合法字符。
- (2) 使用 `parseInt()` 方法，`parseInt()` 函数可解析一个字符串，并返回一个整数。还可以设置要解析的数字的基数。当基数的值为 `0`，或没有设置该参数时，`parseInt()` 会根据 `string` 来判断数字的基数。
- (3) 使用 `parseFloat()` 方法，该函数解析一个字符串参数并返回一个浮点数。
- (4) 使用 `+` 操作符的隐式转换。

详细资料可以参考： [《详解 JS 中 `Number\(\)`、`parseInt\(\)` 和 `parseFloat\(\)` 的区别》](#)

3.30. 如何将浮点数点左边的数每三位添加一个逗号，如 `12000000.11` 转化为「`12,000,000.11`」？

```
function format(number) {  
    return number && number.replace(/(?!^)(?=(\d{3})+\.)/g, ",");}
```

3.31. 常用正则表达式

```
// (1) 匹配 16 进制颜色值 var regex = /#[[0-9a-fA-F]{6}|[0-9a-fA-F]{3}])/g;  
  
// (2) 匹配日期, 如 yyyy-mm-dd 格式 var regex =  
/^[[0-9]{4}-(0[1-9]|1[0-2])-([0-9]{1}|([12][0-9]|3[01]))$/;  
  
// (3) 匹配 qq 号 var regex = /^[1-9][0-9]{4,10}$/g;  
  
// (4) 手机号码正则 var regex = /^1[34578]\d{9}$/g;  
  
// (5) 用户名正则 var regex = /^[a-zA-Z\$][a-zA-Z0-9_\$]{4,16}\$/;
```

详细资料可以参考： [《前端表单验证常用的 15 个 JS 正则表达式》](#) [《JS 常用正则汇总》](#)

3.32. 生成随机数的各种方法？

[《JS - 生成随机数的方法汇总（不同范围、类型的随机数）》](#)

3.33. 如何实现数组的随机排序？

```
// (1) 使用数组 sort 方法对数组元素随机排序, 让 Math.random() 出来的数与 0.5 比较, 如果大于就返回 1 交换位置, 如果小于就返回 -1, 不交换位置。  
  
function randomSort(a, b) {  
  
    return Math.random() > 0.5 ? -1 : 1;  
  
// 缺点: 每个元素被派到新数组的位置不是随机的, 原因是 sort() 方法是依次比较的。  
  
// (2) 随机从原数组抽取一个元素, 加入到新数组  
  
function randomSort(arr) {  
  
    var result = [];  
  
    while (arr.length > 0) {  
  
        var randomIndex = Math.floor(Math.random() * arr.length);
```

```
        result.push(arr[randomIndex]);

        arr.splice(randomIndex, 1);

    }

    return result;
}

// (3) 随机交换数组内的元素（洗牌算法类似）

function randomSort(arr) {

    var index,
        randomIndex,
        temp,
        len = arr.length;

    for (index = 0; index < len; index++) {

        randomIndex = Math.floor(Math.random() * (len - index)) + index;

        temp = arr[index];

        arr[index] = arr[randomIndex];
        arr[randomIndex] = temp;
    }

    return arr;
}

// es6
function randomSort(array) {

    let length = array.length;

    if (!Array.isArray(array) || length <= 1) return;

    for (let index = 0; index < length - 1; index++) {

        let randomIndex = Math.floor(Math.random() * (length - index)) + index;
        [array[index], array[randomIndex]] = [array[randomIndex], array[index]];
    }
}
```

```
    return array;}
```

详细资料可以参考： [《Fisher and Yates 的原始版》](#) [《javascript 实现数组随机排序?》](#) [《JavaScript 学习笔记：数组随机排序》](#)

3.34. javascript 创建对象的几种方式？

我们一般使用字面量的形式直接创建对象，但是这种创建方式对于创建大量相似对象的时候，会产生大量的重复代码。但 js 和一般的面向对象的语言不同，在 ES6 之前它没有类的概念。但是我们可以使用函数来进行模拟，从而产生出可复用的对象创建方式，我了解到的方式有这么几种：

(1) 第一种是工厂模式，工厂模式的主要工作原理是用函数来封装创建对象的细节，从而通过调用函数来达到复用的目的。但是它有一个很大的问题就是创建出来的对象无法和某个类型联系起来，它只是简单的封装了复用代码，而没有建立起对象和类型间的关系。

(2) 第二种是构造函数模式。js 中每一个函数都可以作为构造函数，只要一个函数是通过 new 来调用的，那么我们就可以把它称为构造函数。执行构造函数首先会创建一个对象，然后将对象的原型指向构造函数的 prototype 属性，然后将执行上下文中的 this 指向这个对象，最后再执行整个函数，如果返回值不是对象，则返回新建的对象。因为 this 的值指向了新建的对象，因此我们可以使用 this 给对象赋值。构造函数模式相对于工厂模式的优点是，所创建的对象和构造函数建立起了联系，因此我们可以通过原型来识别对象的类型。但是构造函数存在一个缺点就是，造成了不必要的函数对象的创建，因为在 js 中函数也是一个对象，因此如果对象属性中如果包含函数的话，那么每次我们都会新建一个函数对象，浪费了不必要的内存空间，因为函数是所有的实例都可以通用的。

(3) 第三种模式是原型模式，因为每一个函数都有一个 prototype 属性，这个属性是一个对象，它包含了通过构造函数创建的所有实例都能共享的属性和方法。因此我们可以使用原型对象来添加公用属性和方法，从而实现代码的复用。这种方式相对于构造函数模式来说，解决了函数对象的复用问题。但是这种模式也存在一些问题，一个是没有办法通过传入参数来初始化值，另一个是如果存在一个引用类型如 Array 这样的值，那么所有的实例将共享一个对象，一个实例对引用类型值的改变会影响所有的实例。

(4) 第四种模式是组合使用构造函数模式和原型模式，这是创建自定义类型的最常见方式。因为构造函数模式和原型模式分开使用都存在一些问题，因此我们可以组合使用这两种模式，通过构造函数来初始化对象的属性，通过原型对象来实现函数方法的复用。这种方法很好的解决了两种模式单独使用时的缺点，但是有一点不足的就是，因为使用了两种不同的模式，所以对于代码的封装性不够好。

(5) 第五种模式是动态原型模式，这一种模式将原型方法赋值的创建过程移动到了构造函数的内部，通过对属性是否存在的判断，可以实现仅在第一次调用函数时对原型对象赋值一次的效果。这一种方式很好地对上面的混合模式进行了封装。

(6) 第六种模式是寄生构造函数模式，这一种模式和工厂模式的实现基本相同，我对这个模式的理解是，它主要是基于一个已有的类型，在实例化时对实例化的对象进行扩展。这样既不用修改原来的构造函数，也达到了扩展对象的目的。它的一个缺点和工厂模式一样，无法实现对象的识别。详细资料可以参考： [《JavaScript 深入理解之对象创建》](#)

3.35. JavaScript 继承的几种实现方式？

我了解的 js 中实现继承的几种方式有：

(1) 第一种是以原型链的方式来实现继承，但是这种实现方式存在的缺点是，在包含有引用类型的数据时，会被所有的实例对象所共享，容易造成修改的混乱。还有就是在创建子类型的时候不能向超类型传递参数。

(2) 第二种方式是使用借用构造函数的方式，这种方式是通过在子类型的函数中调用超类型的构造函数来实现的，这一种方法解决了不能向超类型传递参数的缺点，但是它存在一个问题就是无法实现函数方法的复用，并且超类型原型定义的方法子类型也没有办法访问到。

(3) 第三种方式是组合继承，组合继承是将原型链和借用构造函数结合起来使用的一种方式。通过借用构造函数的方式来实现类型的属性的继承，通过将子类型的原型设置为超类型的实例来实现方法的继承。这种方式解决了上面的两种模式单独使用时的问题，但是由于我们是以超类型的实例来作为子类型的原型，所以调用了两次超类的构造函数，造成了子类型的原型中多了很多不必要的属性。

(4) 第四种方式是原型式继承，原型式继承的主要思路就是基于已有的对象来创建新的对象，实现的原理是，向函数中传入一个对象，然后返回一个以这个对象为原型的对象。这种继承的思路主要不是为了实现创造一种新的类型，只是对某个对象实现一种简单继承，ES5 中定义的 `Object.create()` 方法就是原型式继承的实现。缺点与原型链方式相同。

(5) 第五种方式是寄生式继承，寄生式继承的思路是创建一个用于封装继承过程的函数，通过传入一个对象，然后复制一个对象的副本，然后对象进行扩展，最后返回这个对象。这个扩展的过程就可以理解是一种继承。这种继承的优点就是对一个简单对象实现继承，如果这个对象不是我们的自定义类型时。缺点是没有办法实现函数的复用。

(6) 第六种方式是寄生式组合继承，组合继承的缺点就是使用超类型的实例做为子类型的原型，导致添加了不必要的原型属性。寄生式组合继承的方式是使用超类型的原型的副本作为子类型的原型，这样就避免了创建不必要的属性。

详细资料可以参考： [《JavaScript 深入理解之继承》](#)

3.36. 寄生式组合继承的实现？

```
function Person(name) {
```

```
this.name = name;

Person.prototype.sayName = function() {
    console.log("My name is " + this.name + ".");
}

function Student(name, grade) {
    Person.call(this, name);
    this.grade = grade;
}

Student.prototype =
Object.create(Person.prototype);Student.prototype.constructor = Student;

Student.prototype.sayMyGrade = function() {
    console.log("My grade is " + this.grade + ".");
}
```

3.37. Javascript 的作用域链？

作用域链的作用是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，我们可以访问到外层环境的变量和函数。

作用域链的本质上是一个指向变量对象的指针列表。变量对象是一个包含了执行环境中所有变量和函数的对象。作用域链的前端始终都是当前执行上下文的变量对象。全局执行上下文的变量对象（也就是全局对象）始终是作用域链的最后一个对象。

当我们查找一个变量时，如果当前执行环境中没有找到，我们可以沿着作用域链向后查找。

作用域链的创建过程跟执行上下文的建立有关....

详细资料可以参考： [《JavaScript 深入理解之作用域链》](#)

3.38. 谈谈 This 对象的理解。

`this` 是执行上下文中的一个属性，它指向最后一次调用这个方法的对象。在实际开发中，`this` 的指向可以通过四种调用模式来判断。



1. 第一种是函数调用模式，当一个函数不是一个对象的属性时，直接作为函数来调用时，`this` 指向全局对象。

•
•

2. 第二种是方法调用模式，如果一个函数作为一个对象的方法来调用时，
`this` 指向这个对象。

•
•

3. 第三种是构造器调用模式，如果一个函数用 `new` 调用时，函数执行前
会新创建一个对象，`this` 指向这个新创建的对象。

•
•

4. 第四种是 `apply`、`call` 和 `bind` 调用模式，这三个方法都可以显示的
指定调用函数的 `this` 指向。其中 `apply` 方法接收两个参数：一个是 `this`
绑定的对象，一个是参数数组。`call` 方法接收的参数，第一个是 `this` 绑
定的对象，后面的其余参数是传入函数执行的参数。也就是说，在使用
`call()` 方法时，传递给函数的参数必须逐个列举出来。`bind` 方法通过传入
一个对象，返回一个 `this` 绑定了传入对象的新函数。这个函数的 `this` 指
向除了使用 `new` 时会被改变，其他情况下都不会改变。

•

这四种方式，使用构造器调用模式的优先级最高，然后是 `apply`、`call` 和 `bind` 调用模式，
然后是方法调用模式，然后

是函数调用模式。

[《JavaScript 深入理解之 `this` 详解》](#)

3.39. `eval` 是做什么的？

它的功能是把对应的字符串解析成 JS 代码并运行。应该避免使用 eval，不安全，非常耗性能（2 次，一次解析成 js 语句，一次执行）。

详细资料可以参考： [《eval\(\)》](#)

3.40. 什么是 DOM 和 BOM？

DOM 指的是文档对象模型，它指的是把文档当做一个对象来对待，这个对象主要定义了处理网页内容的方法和接口。BOM 指的是浏览器对象模型，它指的是把浏览器当做一个对象来对待，这个对象主要定义了与浏览器进行交互的方法和接口。BOM 的核心是 window，而 window 对象具有双重角色，它既是通过 js 访问浏览器窗口的一个接口，又是一个 Global（全局）对象。这意味着在网页中定义的任何对象，变量和函数，都作为全局对象的一个属性或者方法存在。window 对象含有 location 对象、navigator 对象、screen 对象等子对象，并且 DOM 的最根本的对象 document 对象也是 BOM 的 window 对象的子对象。

详细资料可以参考： [《DOM, DOCUMENT, BOM, WINDOW 有什么区别?》](#)

[《Window 对象》](#) [《DOM 与 BOM 分别是什么，有何关联？》](#) [《JavaScript 学习总结（三）BOM 和 DOM 详解》](#)

3.41. 写一个通用的事件监听器函数。

```
const EventUtils = {

    // 视能力分别使用 dom0||dom2||IE 方式 来绑定事件

    // 添加事件

    addEvent: function(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;

        }
    }
}
```

```
},  
  
// 移除事件  
  
removeEvent: function(element, type, handler) {  
  
    if (element.removeEventListener) {  
  
        element.removeEventListener(type, handler, false);  
  
    } else if (element.detachEvent) {  
  
        element.detachEvent("on" + type, handler);  
  
    } else {  
  
        element["on" + type] = null;  
  
    }  
  
},  
  
// 获取事件目标  
  
getTarget: function(event) {  
  
    return event.target || event.srcElement;  
  
},  
  
// 获取 event 对象的引用，取到事件的所有信息，确保随时能使用 event  
  
getEvent: function(event) {  
  
    return event || window.event;  
  
},  
  
// 阻止事件（主要是事件冒泡，因为 IE 不支持事件捕获）  
  
stopPropagation: function(event) {  
  
    if (event.stopPropagation) {  
  
        event.stopPropagation();  
  
    } else {  
  
        event.cancelBubble = true;  
  
    }  
}
```

```
    }

    },
    // 取消事件的默认行为

    preventDefault: function(event) {

        if (event.preventDefault) {

            event.preventDefault();

        } else {

            event.returnValue = false;

        }
    });
}
```

详细资料可以参考： [《JS 事件模型》](#)

3.42. 事件是什么？IE 与火狐的事件机制有什么区别？如何阻止冒泡？

1.事件是用户操作网页时发生的交互动作，比如 click/move，事件除了用户触发的动作外，还可以是文档加载，窗口滚动和大小调整。事件被封装成一个 event 对象，包含了该事件发生时的所有相关信息（event 的属性）以及可以对事件进行的操作（event 的方法）。

2.事件处理机制：IE 支持事件冒泡、Firefox 同时支持两种事件模型，也就是：事件冒泡和事件捕获。

3.event.stopPropagation() 或者 ie 下的方法 event.cancelBubble = true;



详细资料可以参考： [《Javascript 事件模型系列（一）事件及事件的三种模型》](#)

[《Javascript 事件模型：事件捕获和事件冒泡》](#)

3.43. 三种事件模型是什么？

事件是用户操作网页时发生的交互动作或者网页本身的一些操作，现代浏览器一共有三种事件模型。

第一种事件模型是最早的 DOM0 级模型，这种模型不会传播，所以没有事件流的概念，但是现在有的浏览器支持以冒泡的方式实现，它可以在网页中直接定义监听函数，也可以通过 `js` 属性来指定监听函数。这种方式是所有浏览器都兼容的。

第二种事件模型是 IE 事件模型，在该事件模型中，一次事件共有两个过程，事件处理阶段，和事件冒泡阶段。事件处理阶段会首先执行目标元素绑定的监听事件。然后是事件冒泡阶段，冒泡指的是事件从目标元素冒泡到 `document`，依次检查经过的节点是否绑定了事件监听函数，如果有则执行。这种模型通过 `attachEvent` 来添加监听函数，可以添加多个监听函数，会按顺序依次执行。

第三种是 DOM2 级事件模型，在该事件模型中，一次事件共有三个过程，第一个过程是事件捕获阶段。捕获指的是事件从 `document` 一直向下传播到目标元素，依次检查经过的节点是否绑定了事件监听函数，如果有则执行。后面两个阶段和 IE 事件模型的两个阶段相同。这种事件模型，事件绑定的函数是 `addEventListener`，其中第三个参数可以指定事件是否在捕获阶段执行。

详细资料可以参考： [《一个 DOM 元素绑定多个事件时，先执行冒泡还是捕获》](#)



4. 44. 事件委托是什么？

事件委托本质上是利用了浏览器事件冒泡的机制。因为事件在冒泡过程中会上传到父节点，并且父节点可以通过事件对象获取到目标节点，因此可以把子节点的监听函数定义在父节点上，由父节点的监听函数统一处理多个子元素的事件，这种方式称为事件代理。使用事件代理我们可以不必要为每一个子元素都绑定一个监听事件，这样减少了内存上的消耗。并且使用事件代理我们还可以实现事件的动态绑定，比如说新增了一个子节点，我们并不需要单独地为它添加一个监听事件，它所发生的事件会交给父元素中的监听函数来处理。

详细资料可以参考： [《JavaScript 事件委托详解》](#)

3.45. ["1", "2", "3"].map(parseInt) 答案是多少？

`parseInt()` 函数能解析一个字符串，并返回一个整数，需要两个参数 (`val, radix`)，其中 `radix` 表示要解析的数字的基数。（该值介于 `2 ~ 36` 之间，并且字符串中的数字不能大于 `radix` 才能正确返回数字结果值）。此处 `map` 传了 3 个参数 (`element, index, array`)，默认第三个参数被忽略掉，因此三次传入的参数分别为 `"1-0"`, `"2-1"`, `"3-2"`

因为字符串的值不能大于基数，因此后面两次调用均失败，返回 `Nan`，第一次基数为 `0`，按十进制解析返回 `1`。

详细资料可以参考：[《为什么 `\["1", "2", "3"\].map\(parseInt\)` 返回 `\[1,Nan,Nan\]`？》](#)

3. 46. 什么是闭包，为什么要用它？

闭包是指有权访问另一个函数作用域中变量的函数，创建闭包的最常见的方式就在一个函数内创建另一个函数，创建的函数可以访问到当前函数的局部变量。闭包有两个常用的用途。闭包的第一个用途是使我们在函数外部能够访问到函数内部的变量。通过使用闭包，我们可以通过在外部调用闭包函数，从而在外部访问到函数内部的变量，可以使用这种方法来创建私有变量。函数的另一个用途是使已经运行结束的函数上下文中的变量对象继续留在内存中，因为闭包函数保留了这个变量对象的引用，所以这个变量对象不会被回收。其实闭包的本质就是作用域链的一个特殊的应用，只要了解了作用域链的创建过程，就能够理解闭包的实现原理。

详细资料可以参考：[《JavaScript 深入理解之闭包》](#)

3.47. javascript 代码中的 "`use strict`"; 是什么意思？使用它区别是什么？

相关知识点：

`use strict` 是一种 `ECMAScript5` 添加的（严格）运行模式，这种模式使得 `Javascript` 在更严格的条件下运行。设立“严格模式”的目的，主要有以下几个：

- 消除 `Javascript` 语法的一些不合理、不严谨之处，减少一些怪异行为；
- 消除代码运行的一些不安全之处，保证代码运行的安全；
- 提高编译器效率，增加运行速度；
- 为未来新版本的 `Javascript` 做好铺垫。

区别：

- 1. 禁止使用 `with` 语句。
- 2. 禁止 `this` 关键字指向全局对象。

- 3.对象不能有重名的属性。

回答：

`use strict` 指的是严格运行模式，在这种模式对 `js` 的使用添加了一些限制。比如说禁止 `this` 指向全局对象，还有禁止使用 `with` 语句等。设立严格模式的目的，主要是为了消除代码使用中的一些不安全的使用方式，也是为了消除 `js` 语法本身的一些不合理的地方，以此来减少一些运行时的怪异的行为。同时使用严格运行模式也能够提高编译的效率，从而提高代码的运行速度。我认为严格模式代表了 `js` 一种更合理、更安全、更严谨的发展方向。

详细资料可以参考： [《Javascript 严格模式详解》](#)

3.48. 如何判断一个对象是否属于某个类？

第一种方式是使用 `instanceof` 运算符来判断构造函数的 `prototype` 属性是否出现在对象的原型链中的任何位置。

第二种方式可以通过对象的 `constructor` 属性来判断，对象的 `constructor` 属性指向该对象的构造函数，但是这种方式不是很安全，因为 `constructor` 属性可以被改写。

第三种方式，如果需要判断的是某个内置的引用类型的话，可以使用 `Object.prototype.toString()` 方法来打印对象的

`[[Class]]` 属性来进行判断。

详细资料可以参考： [《js 判断一个对象是否属于某一类》](#)

3.49. `instanceof` 的作用？

// `instanceof` 运算符用于判断构造函数的 `prototype` 属性是否出现在对象的原型链中的任何位置。// 实现：

```
function myInstanceof(left, right) {
  let proto = Object.getPrototypeOf(left), // 获取对象的原型
      prototype = right.prototype; // 获取构造函数的 prototype 对象
  // 判断构造函数的 prototype 对象是否在对象的原型链上
  while (true) {
    if (!proto) return false;
```

```
    if (proto === prototype) return true;

    proto = Object.getPrototypeOf(proto);

}
```

详细资料可以参考： [《instanceof》](#)

3.50. new 操作符具体干了什么呢？如何实现？

```
// (1) 首先创建了一个新的空对象 // (2) 设置原型，将对象的原型设置为函数的 prototype 对象。
// (3) 让函数的 this 指向这个对象，执行构造函数的代码（为这个新对象添加属性）
// (4) 判断函数的返回值类型，如果是值类型，返回创建的对象。如果是引用类型，就返回这个引用类型的对象。
```

// 实现：

```
function objectFactory() {

    let newObject = null,
        constructor = Array.prototype.shift.call(arguments),
        result = null;

    // 参数判断

    if (typeof constructor !== "function") {

        console.error("type error");

        return;
    }

}
```

```
// 新建一个空对象，对象的原型为构造函数的 prototype 对象
```

```
newObject = Object.create(constructor.prototype);
```

```
// 将 this 指向新建对象，并执行函数
```

```
result = constructor.apply(newObject, arguments);
```

```
// 判断返回对象

let flag =

  result && (typeof result === "object" || typeof result === "function");

// 判断返回结果

return flag ? result : newObject;

// 使用方法// objectFactory(构造函数, 初始化参数);
```

详细资料可以参考： [《new 操作符具体干了什么？》](#) [《JavaScript 深入之 new 的模拟实现》](#)

3.51. Javascript 中，有一个函数，执行时对象查找时，永远不会去查找原型，这个函数是？

hasOwnProperty

所有继承了 `Object` 的对象都会继承到 `hasOwnProperty` 方法。这个方法可以用来检测一个对象是否含有特定的自身属性，和

`in` 运算符不同，该方法会忽略掉那些从原型链上继承到的属性。

详细资料可以参考： [《Object.prototype.hasOwnProperty\(\)》](#)

3.52. 对于 JSON 的了解？

相关知识点：

JSON 是一种数据交换格式，基于文本，优于轻量，用于交换数据。

JSON 可以表示数字、布尔值、字符串、`null`、数组（值的有序序列），以及由这些值（或数组、对象）所组成的对象（字符串与值的映射）。

JSON 使用 `JavaScript` 语法，但是 JSON 格式仅仅是一个文本。文本可以被任何编程语言读取及作为数据格式传递。

回答：

`JSON` 是一种基于文本的轻量级的数据交换格式。它可以被任何的编程语言读取和作为数据格式来传递。在项目开发中，我们使用 `JSON` 作为前后端数据交换的方式。在前端我们通过将一个符合 `JSON` 格式的数据结构序列化为 `JSON` 字符串，然后将它传递到后端，后端通过 `JSON` 格式的字符串解析后生成对应的数据结构，以此来实现前后端数据的一个传递。因为 `JSON` 的语法是基于 `js` 的，因此很容易将 `JSON` 和 `js` 中的对象弄混，但是我们应该注意的是 `JSON` 和 `js` 中的对象不是一回事，`JSON` 中对象格式更加严格，比如说在 `JSON` 中属性值不能为函数，不能出现 `NaN` 这样的属性值等，因此大多数的 `js` 对象是不符合 `JSON` 对象的格式的。

在 `js` 中提供了两个函数来实现 `js` 数据结构和 `JSON` 格式的转换处理，一个是 `JSON.stringify` 函数，通过传入一个符合 `JSON` 格式的数据结构，将其转换为一个 `JSON` 字符串。如果传入的数据结构不符合 `JSON` 格式，那么在序列化的时候会对这些值进行对应的特殊处理，使其符合规范。在前端向后端发送数据时，我们可以调用这个函数将数据对象转化为 `JSON` 格式的字符串。

另一个函数 `JSON.parse()` 函数，这个函数用来将 `JSON` 格式的字符串转换为一个 `js` 数据结构，如果传入的字符串不是标准的 `JSON` 格式的字符串的话，将会抛出错误。当我们从后端接收到 `JSON` 格式的字符串时，我们可以通过这个方法来将其解析为一个 `js` 数据结构，以此来进行数据的访问。

详细资料可以参考： [《深入理解 JavaScript 中的 JSON》](#)

3.53. `[] forEach.call($$(""),function(a){a.style.outline="1px solid "#"+(~~(Math.random()(1<<24))).toString(16)})` 能解释一下这段代码的意思吗？

(1) 选取页面所有 `DOM` 元素。在浏览器的控制台中可以使用`$$()`方法来获取页面中相应的元素，这是现代浏览器提供的一个命令行 API 相当于 `document.querySelectorAll` 方法。

(2) 循环遍历 `DOM` 元素

(3) 给元素添加 `outline`。由于渲染的 `outline` 是不在 `CSS` 盒模型中的，所以为元素添加 `outline` 并不会影响元素的大小和页面的布局。

(4) 生成随机颜色函数。`Math.random()*(1<<24)` 可以得到 $0 \sim 2^{24} - 1$ 之间的随机数，因为得到的是一个浮点数，但我们只需要整数部分，使用取反操作符 `~` 连续两次取反获得整数部分，然后再用 `toString(16)` 的方式，转换为一个十六进制的字符串。

详细资料可以参考： [《通过一行代码学 JavaScript》](#)

3.54. js 延迟加载的方式有哪些？

相关知识点：

`js` 延迟加载，也就是等页面加载完成之后再加载 `JavaScript` 文件。`js` 延迟加载有助于提高页面加载速度。

一般有以下几种方式：

- `defer` 属性
- `async` 属性
- 动态创建 DOM 方式
- 使用 `setTimeout` 延迟方法
- 让 `JS` 最后加载

回答：

`js` 的加载、解析和执行会阻塞页面的渲染过程，因此我们希望 `js` 脚本能够尽可能的延迟加载，提高页面的渲染速度。我了解到的几种方式是：

第一种方式是我们一般采用的是将 `js` 脚本放在文档的底部，来使 `js` 脚本尽可能的在最后加载执行。

第二种方式是给 `js` 脚本添加 `defer` 属性，这个属性会让脚本的加载与文档的解析同步解析，然后在文档解析完成后再执行这个脚本文件，这样的话就能使页面的渲染不被阻塞。多个设置了 `defer` 属性的脚本按规范来说最后是顺序执行的，但是在一些浏览器中可能不是这样。

第三种方式是给 `js` 脚本添加 `async` 属性，这个属性会使脚本异步加载，不会阻塞页面的解析过程，但是当脚本加载完成后立即执行 `js` 脚本，这个时候如果文档没有解析完成的话同样会阻塞。多个 `async` 属性的脚本的执行顺序是不可预测的，一般不会按照代码的顺序依次执行。

第四种方式是动态创建 DOM 标签的方式，我们可以对文档的加载事件进行监听，当文档加载完成后再动态的创建 `script` 标签来引入 `js` 脚本。

详细资料可以参考：[《JS 延迟加载的几种方式》](#) [《HTML 5 <script> async 属性》](#)

3.55. Ajax 是什么？如何创建一个 Ajax？

相关知识点：

2005 年 2 月，AJAX 这个词第一次正式提出，它是 Asynchronous JavaScript and XML 的缩写，指的是通过 JavaScript 的 异步通信，从服务器获取 XML 文档从中提取数据，再更新当前网页的对应部分，而不用刷新整个网页。

具体来说，AJAX 包括以下几个步骤。

- 1. 创建 XMLHttpRequest 对象，也就是创建一个异步调用对象
- 2. 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
- 3. 设置响应 HTTP 请求状态变化的函数
- 4. 发送 HTTP 请求
- 5. 获取异步调用返回的数据
- 6. 使用 JavaScript 和 DOM 实现局部刷新

一般实现：

```
const SERVER_URL = "/server";

let xhr = new XMLHttpRequest();

// 创建 Http 请求 xhr.open("GET", SERVER_URL, true);

// 设置状态监听函数 xhr.onreadystatechange = function() {

  if (this.readyState !== 4) return;

  // 当请求成功时

  if (this.status === 200) {

    handle(this.response);

  } else {
```

```
    console.error(this.statusText);

};

// 设置请求失败时的监听函数 xhr.onerror = function() {

    console.error(this.statusText);};

// 设置请求头信息 xhr.responseType = "json";xhr.setRequestHeader("Accept",
"application/json");

// 发送 Http 请求 xhr.send(null);

// promise 封装实现:

function getJSON(url) {

    // 创建一个 promise 对象

    let promise = new Promise(function(resolve, reject) {

        let xhr = new XMLHttpRequest();

        // 新建一个 http 请求

        xhr.open("GET", url, true);

        // 设置状态的监听函数

        xhr.onreadystatechange = function() {

            if (this.readyState !== 4) return;

            // 当请求成功或失败时，改变 promise 的状态

            if (this.status === 200) {

                resolve(this.response);

            } else {

                reject(new Error(this.statusText));

            }

        }

    });

}

getJSON("http://www.baidu.com").then(function(data) {
    console.log(data);
}).catch(function(error) {
    console.error(error);
});
```

```
};

// 设置错误监听函数

xhr.onerror = function() {

    reject(new Error(this.statusText));

};

// 设置响应的数据类型

xhr.responseType = "json";

// 设置请求头信息

xhr.setRequestHeader("Accept", "application/json");

// 发送 http 请求

xhr.send(null);

});

return promise;
}
```

回答：

我对 `ajax` 的理解是，它是一种异步通信的方法，通过直接由 `js` 脚本向服务器发起 `http` 通信，然后根据服务器返回的数据，更新网页的相应部分，而不用刷新整个页面的一种方法。创建一个 `ajax` 有这样几个步骤

首先是创建一个 `XMLHttpRequest` 对象。然后在这个对象上使用 `open` 方法创建一个 `http` 请求，`open` 方法所需要的参数是请求的方法、请求的地址、是否异步和用户的认证信息。在发起请求前，我们可以为这个对象添加一些信息和监听函数。比如说我们可以通过 `setRequestHeader` 方法来为请求添加头信息。我们还可以为这个对象添加一个状态监听函数。一个 `XMLHttpRequest` 对象一共有 5 个状态，当它的状态变化时会触发 `onreadystatechange` 事件，我们可以通过设置监听函数，来处理请求成功后的结果。当对象的 `readyState` 变为 4 的时候，代表服务器返回的数据接收完成，这个时候我们可以通过判断请求的状态，如果状态是 `2xx` 或者 `304` 的话则代表返回正常。这个时候我们就可以通过 `response` 中的数据来对页面进行更新了。当对象的属性和监听函数设置完成后，最后我们调用 `send` 方法来向服务器发起请求，可以传入参数作为发送的数据体。

详细资料可以参考：[《XMLHttpRequest 对象》](#) [《从 ajax 到 fetch、axios》](#) [《Fetch 入门》](#) [《传统 Ajax 已死，Fetch 永生》](#)

3.56. 谈一谈浏览器的缓存机制？

浏览器的缓存机制指的是通过在一段时间内保留已接收到的 web 资源的一个副本，如果在资源的有效时间内，发起了对这个资源的再一次请求，那么浏览器会直接使用缓存的副本，而不是向服务器发起请求。使用 web 缓存可以有效地提高页面的打开速度，减少不必要的网络带宽的消耗。web 资源的缓存策略一般由服务器来指定，可以分为两种，分别是强缓存策略和协商缓存策略。使用强缓存策略时，如果缓存资源有效，则直接使用缓存资源，不必再向服务器发起请求。强缓存策略可以通过两种方式来设置，分别是 http 头信息中的 **Expires** 属性和 **Cache-Control** 属性。服务器通过在响应头中添加 **Expires** 属性，来指定资源的过期时间。在过期时间以内，该资源可以被缓存使用，不必再向服务器发送请求。这个时间是一个绝对时间，它是服务器的时间，因此可能存在这样的问题，就是客户端的时间和服务器端的时间不一致，或者用户可以对客户端时间进行修改的情况，这样就可能会影响缓存命中的结果。**Expires** 是 http1.0 中的方式，因为它的一些缺点，在 http 1.1 中提出了一个新的头部属性就是 **Cache-Control** 属性，它提供了对资源的缓存的更精确的控制。它有很多不同的值，常用的比如我们可以通过设置 **max-age** 来指定资源能够被缓存的时间的大小，这是一个相对的时间，它会根据这个时间的大小和资源第一次请求时的时间来计算出资源过期的时间，因此相对于 **Expires** 来说，这种方式更加有效一些。常用的还有比如 **private**，用来规定资源只能被客户端缓存，不能够代理服务器所缓存。还有如 **no-store**，用来指定资源不能够被缓存，**no-cache** 代表该资源能够被缓存，但是立即失效，每次都需要向服务器发起请求。一般来说只需要设置其中一种方式就可以实现强缓存策略，当两种方式一起使用时，**Cache-Control** 的优先级要高于 **Expires**。

使用协商缓存策略时，会先向服务器发送一个请求，如果资源没有发生修改，则返回一个 304 状态，让浏览器使用本地的缓存副本。

如果资源发生了修改，则返回修改后的资源。协商缓存也可以通过两种方式来设置，分别是 http 头信息中的 **Etag** 和 **Last-Modified** 属性。

服务器通过在响应头中添加 **Last-Modified** 属性来指出资源最后一次修改的时间，当浏览器下一次发起请求时，会在请求头中添加一个 **If-Modified-Since** 的属性，属性值为上一次资源返回时的 **Last-Modified** 的值。当请求发送到服务器后服务器会通过这个属性来和资源的最后一次的修改时间来进行比较，以此来判断资源是否做了修改。如果资源没有修改，那么返回 304 状态，让客户端使用本地的缓存。如果资源已经被修改了，则返回修改后的资源。使用这种方法有一个缺点，就是 **Last-Modified** 标注的最后修改时间只能精确到秒级，如果某些文件在 1 秒钟以内，被修改多次的话，那么文件已将改变了但是 **Last-Modified** 却没有改变，这样会造成缓存命中的不准确。因为 **Last-Modified** 的这种可能发生的不准确性，http 中提供了另外一种方式，那就是 **Etag** 属性。服务器在返回资源的时候，在头信息中添加了 **Etag** 属性，这个属性是资源生成的唯一标识符，当资源发生改变的时候，这个值也会发生改变。在下一次资源请求时，浏览器会在请求头中添加一个 **If-None-Match** 属性，这个属性的值就是上次返回的资源的 **Etag** 的值。服务接收到请求后会根据这个值来和资源当前的 **Etag** 的值来进行比较，以此来判断资源是否发生改变，是否需要返回资源。通过这种方式，比 **Last-Modified** 的方式更加精确。

当 `Last-Modified` 和 `Etag` 属性同时出现的时候，`Etag` 的优先级更高。使用协商缓存的时候，服务器需要考虑负载平衡的问题，因此多个服务器上资源的 `Last-Modified` 应该保持一致，因为每个服务器上 `Etag` 的值都不一样，因此在考虑负载平衡时，最好不要设置 `Etag` 属性。强缓存策略和协商缓存策略在缓存命中时都会直接使用本地的缓存副本，区别只在于协商缓存会向服务器发送一次请求。它们缓存不命中时，都会向服务器发送请求来获取资源。在实际的缓存机制中，强缓存策略和协商缓存策略是一起合作使用的。浏览器首先会根据请求的信息判断，强缓存是否命中，如果命中则直接使用资源。如果不命中则根据头信息向服务器发起请求，使用协商缓存，如果协商缓存命中的话，则服务器不返回资源，浏览器直接使用本地资源的副本，如果协商缓存不命中，则浏览器返回最新的资源给浏览器。

详细资料可以参考：[《浅谈浏览器缓存》](#) [《前端优化：浏览器缓存技术介绍》](#) [《请求头中的 Cache-Control》](#) [《Cache-Control 字段值详解》](#)

3.57. Ajax 解决浏览器缓存问题？

•

1. 在 ajax 发送请求前加上

```
anyAjaxObj.setRequestHeader("If-Modified-Since","0");
```

•

•

2. 在 ajax 发送请求前加上

```
anyAjaxObj.setRequestHeader("Cache-Control","no-cache");
```

•

•

3. 在 URL 后面加上一个随机数： "fresh=" + Math.random();。

•

•

4. 在 URL 后面加上时间戳： "nowtime=" + new Date().getTime();。

•
•
5.如果是使用 `JQuery`, 直接这样就可以了`$.ajaxSetup({cache:false})`。这样页面的所有 `ajax` 都会执行这条语句就是不需要保存缓存记录。

•
详细资料可以参考: [《Ajax 中浏览器的缓存问题解决方法》](#) [《浅谈浏览器缓存》](#)

3.58. 同步和异步的区别?

相关知识点:

同步, 可以理解为在执行完一个函数或方法之后, 一直等待系统返回值或消息, 这时程序是处于阻塞的, 只有接收到返回的值或消息后才往下执行其他的命令。 异步, 执行完函数或方法后, 不必阻塞性地等待返回值或消息, 只需要向系统委托一个异步过程, 那么当系统接收到返回值或消息时, 系统会自动触发委托的异步过程, 从而完成一个完整的流程。

回答:

同步指的是当一个进程在执行某个请求的时候, 如果这个请求需要等待一段时间才能返回, 那么这个进程会一直等待下去, 直到消息返回为止再继续向下执行。

异步指的是当一个进程在执行某个请求的时候, 如果这个请求需要等待一段时间才能返回, 这个时候进程会继续往下执行, 不会阻塞等待消息的返回, 当消息返回时系统再通知进程进行处理。

详细资料可以参考: [《同步和异步的区别》](#)

3.59. 什么是浏览器的同源政策?

我对浏览器的同源政策的理解是, 一个域下的 `js` 脚本在未经允许的情况下, 不能够访问另一个域的内容。这里的同源的指的是两个域的协议、域名、端口号必须相同, 否则则不属于同一个域。同源政策主要限制了三个方面

第一个是当前域下的 `js` 脚本不能够访问其他域下的 `cookie`、`localStorage` 和 `indexDB`。

第二个是当前域下的 `js` 脚本不能够操作访问操作其他域下的 `DOM`。

第三个是当前域下 `ajax` 无法发送跨域请求。

同源政策的目的主要是为了保证用户的信息安全，它只是对 `js` 脚本的一种限制，并不是对浏览器的限制，对于一般的 `img`、或者

`script` 脚本请求都不会有跨域的限制，这是因为这些操作都不会通过响应结果来进行可能出现安全问题的操作。

3.60. 如何解决跨域问题？

相关知识点：

- i. 通过 `jsonp` 跨域
- i. `document.domain + iframe` 跨域
- i. `location.hash + iframe`
- i. `window.name + iframe` 跨域
- i. `postMessage` 跨域
- i. 跨域资源共享（CORS）
- i. `nginx` 代理跨域
- i. `nodejs` 中间件代理跨域
- i. `WebSocket` 协议跨域

回答：

解决跨域的方法我们可以根据我们想要实现的目的来划分。

首先我们如果只是想要实现主域名下的不同子域名的跨域操作，我们可以使用设置 `document.domain` 来解决。

(1) 将 `document.domain` 设置为主域名，来实现相同子域名的跨域操作，这个时候主域名下的 `cookie` 就能够被子域名所访问。同时如果文档中含有主域名相同，子域名不同的 `iframe` 的话，我们也可以对这个 `iframe` 进行操作。

如果是想要解决不同跨域窗口间的通信问题，比如说一个页面想要和页面中的不同源的 `iframe` 进行通信的问题，我们可以使用 `location.hash` 或者 `window.name` 或者 `postMessage` 来解决。

(2) 使用 `location.hash` 的方法，我们可以在主页面动态的修改 `iframe` 窗口的 `hash` 值，然后在 `iframe` 窗口里实现监听函数来实现这样一个单向的通信。因为在 `iframe` 是没有办法访问到不同源的父级窗口的，所以我们不能直接修改父级窗口的 `hash` 值来实现通信，我们可以在 `iframe` 中再加入一个 `iframe`，这个 `iframe` 的内容是和父级页面同源的，所以我们可以 `window.parent.parent` 来修改最顶级页面的 `src`，以此来实现双向通信。

(3) 使用 `window.name` 的方法，主要是基于同一个窗口中设置了 `window.name` 后不同源的页面也可以访问，所以不同源的子页面可以首先在 `window.name` 中写入数据，然后跳转到一个和父级同源的页面。这个时候父级页面就可以访问同源的子页面中 `window.name` 中的数据了，这种方式的好处是可以传输的数据量大。

(4) 使用 `postMessage` 来解决的方法，这是一个 h5 中新增的一个 api。通过它我们可以实现多窗口间的信息传递，通过获取到指定窗口的引用，然后调用 `postMessage` 来发送信息，在窗口中我们通过对 `message` 信息的监听来接收信息，以此来实现不同源间的信息交换。如果是像解决 `ajax` 无法提交跨域请求的问题，我们可以使用 `jsonp`、`cors`、`websocket` 协议、服务器代理来解决问题。

(5) 使用 `jsonp` 来实现跨域请求，它的主要原理是通过动态构建 `script` 标签来实现跨域请求，因为浏览器对 `script` 标签的引入没有跨域的访问限制。通过在请求的 `url` 后指定一个回调函数，然后服务器在返回数据的时候，构建一个 `json` 数据的包装，这个包装就是回调函数，然后返回给前端，前端接收到数据后，因为请求的是脚本文件，所以会直接执行，这样我们先前定义好的回调函数就可以被调用，从而实现了跨域请求的处理。这种方式只能用于 `get` 请求。

(6) 使用 `CORS` 的方式，`CORS` 是一个 W3C 标准，全称是"跨域资源共享"。`CORS` 需要浏览器和服务器同时支持。目前，所有浏览器都支持该功能，因此我们只需要在服务器端配置就行。浏览器将 `CORS` 请求分成两类：简单请求和非简单请求。对于简单请求，浏览器直接发出 `CORS` 请求。具体来说，就是在头信息之中，增加一个 `Origin` 字段。`Origin` 字段用来说明本次请求来自哪个源。服务器根据这个值，决定是否同意这次请求。对于如果 `Origin` 指定的源，不在许可范围内，服务器会返回一个正常的 `HTTP` 回应。浏览器发现，这个回应的头信息没有包含 `Access-Control-Allow-Origin` 字段，就知道出错了，从而抛出一个错误，`ajax` 不会收到响应信息。如果成功的话会包含一些以 `Access-Control-` 开头的字段。非简单请求，浏览器会先发出一次预检请求，来判断该域名是否在服务器的白名单中，如果收到肯定回复后才会发起请求。

(7) 使用 `websocket` 协议，这个协议没有同源限制。

(8) 使用服务器来代理跨域的访问请求，就是有跨域的请求操作时发送请求给后端，让后端代为请求，然后最后将获取的结果发返回。

详细资料可以参考： [《前端常见跨域解决方案（全）》](#) [《浏览器同源政策及其规避方法》](#) [《跨域，你需要知道的全在这里》](#) [《为什么 form 表单提交没有跨域问题，但 ajax 提交有跨域问题？》](#)

3.61. 服务器代理转发时，该如何处理 cookie？

详细资料可以参考： [《深入浅出 Nginx》](#)

3.62. 简单谈一下 cookie ?

我的理解是 `cookie` 是服务器提供的一种用于维护会话状态信息的数据，通过服务器发送到浏览器，浏览器保存在本地，当下一次有同源的请求时，将保存的 `cookie` 值添加到请求头部，发送给服务端。这可以用来实现记录用户登录状态等功能。`cookie` 一般可以存储 4k 大小的数据，并且只能被同源的网页所共享访问。

服务器端可以使用 `Set-Cookie` 的响应头部来配置 `cookie` 信息。一条 `cookie` 包括了 5 个属性值 `expires`、`domain`、`path`、`secure`、`HttpOnly`。其中 `expires` 指定了 `cookie` 失效的时间，`domain` 是域名、`path` 是路径，`domain` 和 `path` 一起限制了 `cookie` 能够被哪些 `url` 访问。`secure` 规定了 `cookie` 只能在确保安全的情况下传输，`HttpOnly` 规定了这个 `cookie` 只能被服务器访问，不能使用 `js` 脚本访问。在发生 `xhr` 的跨域请求的时候，即使是同源下的 `cookie`，也不会被自动添加到请求头部，除非显式地规定。

详细资料可以参考： [《HTTP cookies》](#) [《聊一聊 cookie》](#)

3.63. 模块化开发怎么做？

我对模块的理解是，一个模块是实现一个特定功能的一组方法。在最开始的时候，`js` 只实现一些简单的功能，所以并没有模块的概念，但随着程序越来越复杂，代码的模块化开发变得越来越重要。由于函数具有独立作用域的特点，最原始的写法是使用函数来作为模块，几个函数作为一个模块，但是这种方式容易造成全局变量的污染，并且模块间没有联系。

后面提出了对象写法，通过将函数作为一个对象的方法来实现，这样解决了直接使用函数作为模块的一些缺点，但是这种办法会暴露所有的所有的模块成员，外部代码可以修改内部属性的值。现在最常用的是立即执行函数的写法，通过利用闭包来实现模块私有作用域的建立，同时不会对全局作用域造成污染。

详细资料可以参考：[《浅谈模块化开发》](#) [《Javascript 模块化编程（一）：模块的写法》](#) [《前端模块化：CommonJS, AMD, CMD, ES6》](#) [《Module 的语法》](#)

3.64. js 的几种模块规范？

js 中现在比较成熟的有四种模块加载方案。

第一种是 CommonJS 方案，它通过 `require` 来引入模块，通过 `module.exports` 定义模块的输出接口。这种模块加载方案是服务器端的解决方案，它是以同步的方式来引入模块的，因为在服务端文件都存储在本地磁盘，所以读取非常快，所以以同步的方式加载没有问题。但如果是在浏览器端，由于模块的加载是使用网络请求，因此使用异步加载的方式更加合适。

第二种是 AMD 方案，这种方案采用异步加载的方式来加载模块，模块的加载不影响后面语句的执行，所有依赖这个模块的语句都定义在一个回调函数里，等到加载完成后再执行回调函数。`require.js` 实现了 AMD 规范。

第三种是 CMD 方案，这种方案和 AMD 方案都是为了解决异步模块加载的问题，`sea.js` 实现了 CMD 规范。它和 `require.js` 的区别在于模块定义时对依赖的处理不同和对依赖模块的执行时机的处理不同。参考 [60](#)

第四种方案是 ES6 提出的方案，使用 `import` 和 `export` 的形式来导入导出模块。这种方案和上面三种方案都不同。参考 [61](#)。

3.65. AMD 和 CMD 规范的区别？

它们之间的主要区别有两个方面。

(1) 第一个方面是在模块定义时对依赖的处理不同。AMD 推崇依赖前置，在定义模块的时候就要声明其依赖的模块。而 CMD 推崇就近依赖，只有在用到某个模块的时候再去 `require`。

(2) 第二个方面是对依赖模块的执行时机处理不同。首先 AMD 和 CMD 对于模块的加载方式都是异步加载，不过它们的区别在于 模块的执行时机，AMD 在依赖模块加载完成后就直接执行依赖模块，依赖模块的执行顺序和我们书写的顺序不一定一致。而 CMD 在依赖模块加载完成后并不执行，只是下载而已，等到所有的依赖模块都加载好后，进入回调函数逻辑，遇到 require 语句 的时候才执行对应的模块，这样模块的执行顺序就和我们书写的顺序保持一致了。

```
// CMD
define(function(require, exports, module) {
    var a = require("./a");
    a.doSomething();
    // 此处略去 100 行
    var b = require("./b"); // 依赖可以就近书写
    b.doSomething();
    // ...});

// AMD 默认推荐
define(["./a", "./b"], function(a, b) {
    // 依赖必须一开始就写好
    a.doSomething();
    // 此处略去 100 行
    b.doSomething();
    // ...});
```

详细资料可以参考： [《前端模块化，AMD 与 CMD 的区别》](#)

3.66. ES6 模块与 CommonJS 模块、AMD、CMD 的差异。



1.CommonJS 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用。

CommonJS 模块输出的是值的拷贝，也就是说，一旦输出一个值，模块内部的变化就影响不到这个值。ES6 模块的运行机制与 CommonJS 不一样。JS 引擎对脚本静态分析的时候，遇到模块加载命令 `import`，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。

•
•

2.CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。

CommonJS 模块就是对象，即在输入时是先加载整个模块，生成一个对象，然后再从这个对象上面读取方法，这种加载称为“运行时加载”。而 ES6 模块不是对象，它的对外接口只是一种静态定义，在代码静态解析阶段就会生成。

•

3.67. requireJS 的核心原理是什么？（如何动态加载的？如何避免多次加载的？如何 缓存的？）

`require.js` 的核心原理是通过动态创建 `script` 脚本来异步引入模块，然后对每个脚本的 `load` 事件进行监听，如果每个脚本都加载完成了，再调用回调函数。

详细资料可以参考： [《requireJS 的用法和原理分析》](#) [《requireJS 的核心原理是什么？》](#) [《从 RequireJs 源码剖析脚本加载原理》](#) [《requireJS 原理分析》](#)

3.68. JS 模块加载器的轮子怎么造，也就是如何实现一个模块加载器？

详细资料可以参考： [《JS 模块加载器加载原理是怎么样的？》](#)

3.69. ECMAScript6 怎么写 `class`，为什么会出现 `class` 这种东西？

在我看来 ES6 新添加的 `class` 只是为了补充 js 中缺少的一些面向对象语言的特性，但本质上来说它只是一种语法糖，不是一个新的东西，其背后还是原型继承的思想。通过加入 `class` 可以有利于我们更好的组织代码。在 `class` 中添加的方法，其实是添加在类的原型上的。

详细资料可以参考： [《ECMAScript 6 实现了 class，对 JavaScript 前端开发有什么意义？》](#) [《Class 的基本语法》](#)

3.70. `document.write` 和 `innerHTML` 的区别？

`document.write` 的内容会代替整个文档内容，会重写整个页面。

`innerHTML` 的内容只是替代指定元素的内容，只会重写页面中的部分内容。

详细资料可以参考： [《简述 `document.write` 和 `innerHTML` 的区别。》](#)

3.71. DOM 操作——怎样添加、移除、移动、复制、创建和查找节点？

(1) 创建新节点

```
createDocumentFragment(node);createElement(node);createTextNode(text);
```

(2) 添加、移除、替换、插入

```
appendChild(node)removeChild(node)replaceChild(new,old)insertBefore(new,old)
```

(3) 查找

```
getElementById();getElementsByName();getElementsByTagName();getElementsByClassName();querySelector();querySelectorAll();
```

(4) 属性操作

```
getAttribute(key);setAttribute(key,value);hasAttribute(key);removeAttribute(key);
```

详细资料可以参考：《DOM 概述》《原生 JavaScript 的 DOM 操作汇总》《原生 JS 中 DOM 节点相关 API 合集》

3.72. innerHTML 与 outerHTML 的区别？

对于这样一个 HTML 元素：`<div>content
</div>`。

`innerHTML`: 内部 HTML, `content
`;

`outerHTML`: 外部 HTML, `<div>content
</div>`;

`innerText`: 内部文本, `content` ;

`outerText`: 内部文本, `content` ;

3.73. `.call()` 和 `.apply()` 的区别？

它们的作用一模一样，区别仅在于传入参数的形式的不同。

`apply` 接受两个参数，第一个参数指定了函数体内 `this` 对象的指向，第二个参数为一个带下标的集合，这个集合可以为数组，也可以为类数组，`apply` 方法把这个集合中的元素作为参数传递给被调用的函数。`call` 传入的参数数量不固定，跟 `apply` 相同的是，第一个参数也是代表函数体内的 `this` 指向，从第二个参数开始往后，每个参数被依次传入函数。

详细资料可以参考：《[apply、call 的区别和用途](#)》

3.74. JavaScript 类数组对象的定义？

一个拥有 `length` 属性和若干索引属性的对象就可以被称为类数组对象，类数组对象和数组类似，但是不能调用数组的方法。常见的类数组对象有 `arguments` 和 DOM 方法的返回结果，还有一个函数也可以被看作是类数组对象，因为它含有 `length` 属性值，代表可接收的参数个数。

常见的类数组转换为数组的方法有这样几种：

(1) 通过 `call` 调用数组的 `slice` 方法来实现转换

```
Array.prototype.slice.call(arrayLike);
```

(2) 通过 `call` 调用数组的 `splice` 方法来实现转换

```
Array.prototype.splice.call(arrayLike, 0);
```

(3) 通过 `apply` 调用数组的 `concat` 方法来实现转换

```
Array.prototype.concat.apply([], arrayLike);
```

(4) 通过 `Array.from` 方法来实现转换

```
Array.from(arrayLike);
```

详细的资料可以参考： [《JavaScript 深入之类数组对象与 arguments》](#)

[《javascript 类数组》](#) [《深入理解 JavaScript 类数组》](#)

3.75. 数组和对象有哪些原生方法，列举一下？

数组和字符串的转换方法：`toString()`、`toLocaleString()`、`join()` 其中 `join()` 方法可以指定转换为字符串时的分隔符。数组尾部操作的方法 `pop()` 和 `push()`，`push` 方法可以传入多个参数。数组首部操作的方法 `shift()` 和 `unshift()` 重排序的方法 `reverse()` 和 `sort()`，`sort()` 方法可以传入一个函数来进行比较，传入前后两个值，如果返回值为正数，则交换两个参数的位置。

数组连接的方法 `concat()`，返回的是拼接好的数组，不影响原数组。

数组截取办法 `slice()`，用于截取数组中的一部分返回，不影响原数组。

数组插入方法 `splice()`，影响原数组查找特定项的索引的方法，`indexOf()` 和 `lastIndexOf()` 迭代方法 `every()`、`some()`、`filter()`、`map()` 和 `forEach()` 方法

数组归并方法 `reduce()` 和 `reduceRight()` 方法

详细资料可以参考： [《JavaScript 深入理解之 Array 类型详解》](#)

3.76. 数组的 `fill` 方法？

`fill()` 方法用一个固定值填充一个数组中从起始索引到终止索引内的全部元素。不包括终止索引。

`fill` 方法接受三个参数 `value`，`start` 以及 `end`，`start` 和 `end` 参数是可选的，其默认值分别为 `0` 和 `this` 对象的 `length` 属性值。

详细资料可以参考： [《Array.prototype.fill\(\)》](#)

3.77. [...] 的长度？

尾后逗号（有时叫做“终止逗号”）在向 `JavaScript` 代码添加元素、参数、属性时十分有用。如果你想要添加新的属性，并且上一行已经使用了尾后逗号，你可以仅仅添加新的一行，而不需要修改上一行。这使得版本控制更加清晰，以及代码维护麻烦更少。

`JavaScript` 一开始就支持数组字面值中的尾后逗号，随后向对象字面值（`ECMAScript 5`）中添加了尾后逗号。最近（`ECMAScript 2017`），又将其添加到函数参数中。但是 `JSON` 不支持尾后逗号。

如果使用了多于一个尾后逗号，会产生间隙。带有间隙的数组叫做稀疏数组（密致数组没有间隙）。稀疏数组的长度为逗号的数量。

详细资料可以参考： [《尾后逗号》](#)

3.78. `JavaScript` 中的作用域与变量声明提升？

变量提升的表现是，无论我们在函数中何处位置声明的变量，好像都被提升到了函数的首部，我们可以在变量声明前访问到而不会报错。造成变量声明提升的本质原因是 `js` 引擎在代码执行前有一个解析的过程，创建了执行上下文，初始化了一些代码执行时需要用到的对象。当我们访问一个变量时，我们会到当前执行上下文中的作用域链中去查找，而作用域链的前端指向的是当前执行上下文的变量对象，这个变量对象是执行上下文的一个属性，它包含了函数的形参、所有的函数和变量声明，这个对象的是在代码解析的时候创建的。这就是会出现变量声明提升的根本原因。

详细资料可以参考： [《JavaScript 深入理解之变量对象》](#)

3.79. 如何编写高性能的 `Javascript` ？

- 1. 使用位运算代替一些简单的四则运算。

- 2. 避免使用过深的嵌套循环。
- 3. 不要使用未定义的变量。
- 4. 当需要多次访问数组长度时，可以用变量保存起来，避免每次都会去进行属性查找。

详细资料可以参考：《如何编写高性能的 Javascript?》

3.80. 简单介绍一下 V8 引擎的垃圾回收机制

v8 的垃圾回收机制基于分代回收机制，这个机制又基于世代假说，这个假说有两个特点，一是新生的对象容易早死，另一个是不死的对象会活得更久。基于这个假说，v8 引擎将内存分为了新生代和老生代。新创建的对象或者只经历过一次的垃圾回收的对象被称为新生代。经历过多次垃圾回收的对象被称为老生代。

新生代被分为 `From` 和 `To` 两个空间，`To` 一般是闲置的。当 `From` 空间满了的时候会执行 `Scavenge` 算法进行垃圾回收。当我们执行垃圾回收算法的时候应用逻辑将会停止，等垃圾回收结束后再继续执行。这个算法分为三步：

- (1) 首先检查 `From` 空间的存活对象，如果对象存活则判断对象是否满足晋升到老生代的条件，如果满足条件则晋升到老生代。如果不满足条件则移动 `To` 空间。
- (2) 如果对象不存活，则释放对象的空间。
- (3) 最后将 `From` 空间和 `To` 空间角色进行交换。

新生代对象晋升到老生代有两个条件：

- (1) 第一个是判断对象是否已经历过一次 `Scavenge` 回收。若经历过，则将对象从 `From` 空间复制到老生代中；若没有经历，则复制到 `To` 空间。
- (2) 第二个是 `To` 空间的内存使用占比是否超过限制。当对象从 `From` 空间复制到 `To` 空间时，若 `To` 空间使用超过 25%，则对象直接晋升到老生代中。设置 25% 的原因主要是因为算法结束后，两个空间结束后会交换位置，如果 `To` 空间的内存太小，会影响后续的内存分配。

老生代采用了标记清除法和标记压缩法。标记清除法首先会对内存中存活的对象进行标记，标记结束后清除掉那些没有标记的对象。由于标记清除后会造成很多的内存碎片，不利于后面的内存分配。所以为了解决内存碎片的问题引入了标记压缩法。

由于在进行垃圾回收的时候会暂停应用的逻辑，对于新生代方法由于内存小，每次停顿的时间不会太长，但对于老生代来说每次垃圾回收的时间长，停顿会造成很大的影响。为了解决这个问题 V8 引入了增量标记的方法，将一次停顿进行的过程分为了多步，每次执行完一小步就让运行逻辑执行一会，就这样交替运行。

详细资料可以参考：[《深入理解 V8 的垃圾回收原理》](#) [《JavaScript 中的垃圾回收》](#)

3.81. 哪些操作会造成内存泄漏？

相关知识点：

- 1.意外的全局变量
- 2.被遗忘的计时器或回调函数
- 3.脱离 DOM 的引用
- 4.闭包

回答：

第一种情况是我们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。

第二种情况是我们设置了 `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。

第三种情况是我们获取一个 DOM 元素的引用，而后面这个元素被删除，由于我们一直保留了对这个元素的引用，所以它也无法被回收。

第四种情况是不合理的使用闭包，从而导致某些变量一直被留在内存当中。

详细资料可以参考：《JavaScript 内存泄漏教程》《4 类 JavaScript 内存泄漏及如何避免》《杜绝 js 中四种内存泄漏类型的发生》《javascript 典型内存泄漏及 chrome 的排查方法》

3.82. 需求：实现一个页面操作不会整页刷新的网站，并且能在浏览器前进、后退时正确响应。给出你的技术实现方案？

通过使用 `pushState + ajax` 实现浏览器无刷新前进后退，当一次 `ajax` 调用成功后我们将一条 `state` 记录加入到 `history`

对象中。一条 `state` 记录包含了 `url`、`title` 和 `content` 属性，在 `popstate` 事件中可以获取到这个 `state` 对象，我们可

以使用 `content` 来传递数据。最后我们通过对 `window.onpopstate` 事件监听来响应浏览器的前进后退操作。

使用 `pushState` 来实现有两个问题，一个是打开首页时没有记录，我们可以使用 `replaceState` 来将首页的记录替换，另一个问

题是当一个页面刷新的时候，仍然会向服务器端请求数据，因此如果请求的 `url` 需要后端的配合将其重定向到一个页面。

详细资料可以参考：《`pushState + ajax` 实现浏览器无刷新前进后退》

《Manipulating the browser history》

3.83. 如何判断当前脚本运行在浏览器还是 `node` 环境中？（阿里）

```
this === window ? 'browser' : 'node';
```

通过判断 `Global` 对象是否为 `window`，如果不为 `window`，当前脚本没有运行在浏览器中。

3.84. 把 `script` 标签放在页面的最底部的 `body` 封闭之前和封闭之后有什么区别？浏览器会如何解析它们？

详细资料可以参考：《为什么把 script 标签放在 body 结束标签之后 html 结束标签之前？》《从 Chrome 源码看浏览器如何加载资源》

3.85. 移动端的点击事件的有延迟，时间是多久，为什么会有？怎么解决这个延时？

移动端点击有 300ms 的延迟是因为移动端会有双击缩放的操作，因此浏览器在 click 之后要等待 300ms，看用户有没有下一次点击，来判断这次操作是不是双击。

有三种办法来解决这个问题：

- 1.通过 meta 标签禁用网页的缩放。
- 2.通过 meta 标签将网页的 viewport 设置为 ideal viewport。
- 3.调用一些 js 库，比如 FastClick

click 延时问题还可能引起点击穿透的问题，就是如果我们在一个元素上注册了 touchStart 的监听事件，这个事件会将这个元素隐藏掉，我们发现当这个元素隐藏后，触发了这个元素下的一个元素的点击事件，这就是点击穿透。

详细资料可以参考：《移动端 300ms 点击延迟和点击穿透》

3.86. 什么是“前端路由”？什么时候适合使用“前端路由”？“前端路由”有哪些优点和缺点？

(1) 什么是前端路由？

前端路由就是把不同路由对应不同的内容或页面的任务交给前端来做，之前是通过服务端根据 url 的不同返回不同的页面实现的。

(2) 什么时候使用前端路由？

在单页面应用，大部分页面结构不变，只改变部分内容的使用

(3) 前端路由有什么优点和缺点?

优点: 用户体验好, 不需要每次都从服务器全部获取, 快速展现给用户

缺点: 单页面无法记住之前滚动的位置, 无法在前进, 后退的时候记住滚动的位置

前端路由一共有两种实现方式, 一种是通过 `hash` 的方式, 一种是通过使用 `pushState` 的方式。

详细资料可以参考: [《什么是“前端路由”》](#) [《浅谈前端路由》](#) [《前端路由是什么东西?》](#)

3.87. 如何测试前端代码么? 知道 `BDD`, `TDD`, `Unit Test` 么? 知道怎么测试你的前端工程么(`mocha`, `sinon`, `jasmin`, `qUnit..`)?

详细资料可以参考: [《浅谈前端单元测试》](#)

3.88. 检测浏览器版本有哪些方式?

检测浏览器版本一共有两种方式:

一种是检测 `window.navigator.userAgent` 的值, 但这种方式很不可靠, 因为 `userAgent` 可以被改写, 并且早期的浏览器如 `ie`, 会通过伪装自己的 `userAgent` 的值为 `Mozilla` 来躲过服务器的检测。

第二种方式是功能检测, 根据每个浏览器独有的特性来进行判断, 如 `ie` 下独有的 `ActiveXObject`。

详细资料可以参考: [《JavaScript 判断浏览器类型》](#)

3.89. 什么是 Polyfill ?

Polyfill 指的是用于实现浏览器并不支持的原生 API 的代码。

比如说 `querySelectorAll` 是很多现代浏览器都支持的原生 Web API，但是有些古老的浏览器并不支持，那么假设有人写了一段代码来实现这个功能使这些浏览器也支持了这个功能，那么这就可以成为一个 Polyfill。

一个 `shim` 是一个库，有自己的 API，而不是单纯实现原生不支持的 API。

详细资料可以参考： [《Web 开发中的“黑话”》](#) [《Polyfill 为何物》](#)

3.90. 使用 JS 实现获取文件扩展名？

// `String.lastIndexOf()` 方法返回指定值（本例中的'.'）在调用该方法的字符串中最后出现的位置，如果没找到则返回 -1。

// 对于 '`filename`' 和 '`.hiddenfile`'，`lastIndexOf` 的返回值分别为 0 和 -1 无符号右移操作符(`>>>`) 将 -1 转换为 4294967295，将 -2 转换为 4294967294，这个方法可以保证边缘情况时文件名不变。

```
// String.prototype.slice() 从上面计算的索引处提取文件的扩展名。如果索引比文件名的长度大，结果为""。function getFileExtension(filename) {
  return filename.slice(((filename.lastIndexOf(".")) - 1) >>> 0) + 2);}
```

详细资料可以参考： [《如何更有效的获取文件扩展名》](#)

3.91. 介绍一下 js 的节流与防抖？

相关知识点：

// 函数防抖： 在事件被触发 n 秒后再执行回调，如果在这 n 秒内事件又被触发，则重新计时。

// 函数节流： 规定一个单位时间，在这个单位时间内，只能有一次触发事件的回调函数执行，如果在同一个单位时间内某事件被触发多次，只有一次能生效。

```
// 函数防抖的实现 function debounce(fn, wait) {
```

```
var timer = null;

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

    // 如果此时存在定时器的话，则取消之前的定时器重新记时
    if (timer) {
        clearTimeout(timer);
        timer = null;
    }

    // 设置定时器，使事件间隔指定事件后执行
    timer = setTimeout(() => {
        fn.apply(context, args);
    }, wait);
};

// 函数节流的实现;function throttle(fn, delay) {

var preTime = Date.now();

return function() {
    var context = this,
        args = arguments,
        nowTime = Date.now();
```

```
// 如果两次时间间隔超过了指定时间，则执行函数。  
  
if (nowTime - preTime >= delay) {  
  
    preTime = Date.now();  
  
    return fn.apply(context, args);  
  
}  
  
};}
```

回答：

函数防抖是指在事件被触发 n 秒后再执行回调，如果在这 n 秒内事件又被触发，则重新计时。这可以使用在一些点击请求的事件上，避免因为用户的多次点击向后端发送多次请求。

函数节流是指规定一个单位时间，在这个单位时间内，只能有一次触发事件的回调函数执行，如果在同一个单位时间内某事件被触发多次，只有一次能生效。节流可以使用在 `scroll` 函数的事件监听上，通过事件节流来降低事件调用的频率。

详细资料可以参考：[《轻松理解 JS 函数节流和函数防抖》](#) [《JavaScript 事件节流和事件防抖》](#) [《JS 的防抖与节流》](#)

3.92. `Object.is()` 与原来的比较操作符 “`====`”、“`==`” 的区别？

相关知识点：

两等号判等，会在比较时进行类型转换。

三等号判等（判断严格），比较时不进行隐式类型转换，（类型不同则会返回 `false`）。

`Object.is` 在三等号判等的基础上特别处理了 `NaN`、`-0` 和 `+0`，保证 `-0` 和 `+0` 不再相同，但 `Object.is(NaN, NaN)` 会返回 `true`.

`Object.is` 应被认为有其特殊的用途，而不能用它认为它比其它的相等对比更宽松或严格。

回答：

使用双等号进行相等判断时，如果两边的类型不一致，则会进行强制类型转化后再进行比较。

使用三等号进行相等判断时，如果两边的类型不一致时，不会做强制类型转换，直接返回 `false`。

使用 `Object.is` 来进行相等判断时，一般情况下和三等号的判断相同，它处理了一些特殊的情况，比如 `-0` 和 `+0` 不再相等，两个 `NaN` 认定为是相等的。

3.93. `escape`,`encodeURI`,`encodeURIComponent` 有什么区别？

相关知识点：

`escape` 和 `encodeURI` 都属于 Percent-encoding，基本功能都是把 URI 非法字符转化成合法字符，转化后形式类似「%*」。

它们的根本区别在于，`escape` 在处理 `0xff` 之外字符的时候，是直接使用字符的 `unicode` 在前面加上一个「%u」，而 `encodeURI` 则是先进行 `UTF-8`，再在 `UTF-8` 的每个字节码前加上一个「%」；在处理 `0xff` 以内字符时，编码方式是一样的（都是「%XX」，XX 为字符的 16 进制 `unicode`，同时也是字符的 `UTF-8`），只是范围（即哪些字符编码哪些字符不编码）不一样。

回答：

`encodeURI` 是对整个 URI 进行转义，将 URI 中的非法字符转换为合法字符，所以对于一些在 URI 中有特殊意义的字符不会进行转义。

`encodeURIComponent` 是对 URI 的组成部分进行转义，所以一些特殊字符也会得到转义。

`escape` 和 `encodeURI` 的作用相同，不过它们对于 `unicode` 编码为 `0xff` 之外字符的时候会有区别，`escape` 是直接在字符的 `unicode` 编码前加上 %u，而 `encodeURI` 首先会将字符转换为 `UTF-8` 的格式，再在每个字节前加上 %。

详细资料可以参考： [《escape,encodeURI,encodeURIComponent 有什么区别？》](#)

3.94. Unicode 和 UTF-8 之间的关系？

Unicode 是一种字符集合，现在可容纳 100 多万个字符。每个字符对应一个不同的 Unicode 编码，它只规定了字符的二进制代码，却没有规定这个二进制代码在计算机中如何编码传输。

UTF-8 是一种对 Unicode 的编码方式，它是一种变长的编码方式，可以用 1~4 个字节来表示一个字符。

详细资料可以参考：[《字符编码详解》](#) [《字符编码笔记: ASCII, Unicode 和 UTF-8》](#)

3.95. js 的事件循环是什么？

相关知识点：

事件队列是一个存储着待执行任务的队列，其中的任务严格按照时间先后顺序执行，排在队头的任务将会率先执行，而排在队尾的任务会最后执行。事件队列每次仅执行一个任务，在该任务执行完毕之后，再执行下一个任务。执行栈则是一个类似于函数调用栈的运行容器，当执行栈为空时，JS 引擎便检查事件队列，如果不为空的话，事件队列便将第一个任务压入执行栈中运行。

回答：

因为 js 是单线程运行的，在代码执行的时候，通过将不同函数的执行上下文压入执行栈中来保证代码的有序执行。在执行同步代码的时候，如果遇到了异步事件，js 引擎并不会一直等待其返回结果，而是会将这个事件挂起，继续执行执行栈中的其他任务。当异步事件执行完毕后，再将异步事件对应的回调加入到与当前执行栈中不同的另一个任务队列中等待执行。任务队列可以分为宏任务队列和微任务队列，当当前执行栈中的事件执行完毕后，js 引擎首先会判断微任务队列中是否有任务可以执行，如果有就将微任务队首的事件压入栈中执行。当微任务队列中的任务都执行完成后再去判断宏任务队列中的任务。

微任务包括了 promise 的回调、node 中的 process.nextTick 、对 Dom 变化监听的 MutationObserver。

宏任务包括了 script 脚本的执行、setTimeout , setInterval , setImmediate 一类的定时事件，还有如 I/O 操作、UI 渲染等。

详细资料可以参考：[《浏览器事件循环机制（event loop）》](#) [《详解 JavaScript 中的 Event Loop（事件循环）机制》](#) [《什么是 Event Loop?》](#) [《这一次，彻底弄懂 JavaScript 执行机制》](#)

3.96. js 中的深浅拷贝实现？

相关资料：

```
// 浅拷贝的实现；

function shallowCopy(object) {
    // 只拷贝对象

    if (!object || typeof object !== "object") return;

    // 根据 object 的类型判断是新建一个数组还是对象

    let newObject = Array.isArray(object) ? [] : {};

    // 遍历 object，并且判断是 object 的属性才拷贝

    for (let key in object) {
        if (object.hasOwnProperty(key)) {

            newObject[key] = object[key];
        }
    }

    return newObject;
}

// 深拷贝的实现；

function deepCopy(object) {
    if (!object || typeof object !== "object") return;

    let newObject = Array.isArray(object) ? [] : {};

    for (let key in object) {
        if (object.hasOwnProperty(key)) {

```

```
newObject[key] =  
  typeof object[key] === "object" ? deepCopy(object[key]) : object[key];  
}  
}  
  
return newObject;}
```

回答：

浅拷贝指的是将一个对象的属性值复制到另一个对象，如果有的属性的值为引用类型的话，那么会将这个引用的地址复制给对象，因此两个对象会有同一个引用类型的引用。浅拷贝可以使用 `Object.assign` 和展开运算符来实现。

深拷贝相对浅拷贝而言，如果遇到属性值为引用类型的时候，它新建一个引用类型并将对应的值复制给它，因此对象获得一个新的引用类型而不是一个原有类型的引用。深拷贝对于一些对象可以使用 `JSON` 的两个函数来实现，但是由于 `JSON` 的对象格式比 `js` 的对象格式更加严格，所以如果属性值里边出现函数或者 `Symbol` 类型的值时，会转换失败。

详细资料可以参考： [《JavaScript 专题之深浅拷贝》](#) [《前端面试之道》](#)

3.97. 手写 `call`、`apply` 及 `bind` 函数

相关资料：

```
// call 函数实现 Function.prototype.myCall = function(context) {  
  // 判断调用对象  
  if (typeof this !== "function") {  
    console.error("type error");  
  }  
  
  // 获取参数  
  let args = [...arguments].slice(1),  
    result = null;
```

```
// 判断 context 是否传入，如果未传入则设置为 window

context = context || window;

// 将调用函数设为对象的方法

context.fn = this;

// 调用函数

result = context.fn(...args);

// 将属性删除

delete context.fn;

return result;};

// apply 函数实现

Function.prototype.myApply = function(context) {

// 判断调用对象是否为函数

if (typeof this !== "function") {

throw new TypeError("Error");

}

let result = null;

// 判断 context 是否存在，如果未传入则为 window

context = context || window;
```

```
// 将函数设为对象的方法

context.fn = this;

// 调用方法

if (arguments[1]) {

    result = context.fn(...arguments[1]);

} else {

    result = context.fn();

}

// 将属性删除

delete context.fn;

return result;};

// bind 函数实现 Function.prototype.myBind = function(context) {

// 判断调用对象是否为函数

if (typeof this !== "function") {

    throw new TypeError("Error");

}

// 获取参数

var args = [...arguments].slice(1),

fn = this;
```

```
return function Fn() {  
    // 根据调用方式，传入不同绑定值  
  
    return fn.apply(  
        this instanceof Fn ? this : context,  
        args.concat(...arguments)  
    );  
};};
```

回答：

call 函数的实现步骤：

- 1. 判断调用对象是否为函数，即使我们是定义在函数的原型上的，但是可能出现使用 call 等方式调用的情况。
- 2. 判断传入上下文对象是否存在，如果不存在，则设置为 window。
- 3. 处理传入的参数，截取第一个参数后的所有参数。
- 4. 将函数作为上下文对象的一个属性。
- 5. 使用上下文对象来调用这个方法，并保存返回结果。
- 6. 删除刚才新增的属性。
- 7. 返回结果。

apply 函数的实现步骤：

- 1. 判断调用对象是否为函数，即使我们是定义在函数的原型上的，但是可能出现使用 call 等方式调用的情况。
- 2. 判断传入上下文对象是否存在，如果不存在，则设置为 window。

- 3.将函数作为上下文对象的一个属性。
- 4.判断参数值是否传入
- 4.使用上下文对象来调用这个方法，并保存返回结果。
- 5.删除刚才新增的属性
- 6.返回结果

bind 函数的实现步骤：

- 1.判断调用对象是否为函数，即使我们是定义在函数的原型上的，但是可能出現使用 call 等方式调用的情况。
- 2.保存当前函数的引用，获取其余传入参数值。
- 3.创建一个函数返回
- 4.函数内部使用 apply 来绑定函数调用，需要判断函数作为构造函数的情况，这个时候需要传入当前函数的 this 给 apply 调用，其余情况都传入指定的上下文对象。

详细资料可以参考：[《手写 call、apply 及 bind 函数》](#) [《JavaScript 深入之 call 和 apply 的模拟实现》](#)

3.98. 函数柯里化的实现

```
// 函数柯里化指的是一种将使用多个参数的一个函数转换成一系列使用一个参数的函数的技术。  
  
function curry(fn, args) {  
    // 获取函数需要的参数长度  
    let length = fn.length;
```

```
args = args || [];

return function() {
    let subArgs = args.slice(0);

    // 拼接得到现有的所有参数

    for (let i = 0; i < arguments.length; i++) {
        subArgs.push(arguments[i]);
    }

    // 判断参数的长度是否已经满足函数所需参数的长度

    if (subArgs.length >= length) {
        // 如果满足，执行函数

        return fn.apply(this, subArgs);
    } else {
        // 如果不满足，递归返回柯里化的函数，等待参数的传入

        return curry.call(this, fn, subArgs);
    }
};

// es6 实现 function curry(fn, ...args) {

    return fn.length <= args.length ? fn(...args) : curry.bind(null,
fn, ...args);
}
```

详细资料可以参考： 《JavaScript 专题之函数柯里化》

3.99. 为什么 $0.1 + 0.2 \neq 0.3$? 如何解决这个问题?

当计算机计算 $0.1+0.2$ 的时候，实际上计算的是这两个数字在计算机里所存储的二进制， 0.1 和 0.2 在转换为二进制表示的时候会出现位数无限循环的情况。`js` 中是以 64 位双精度格式来存储数字的，只有 53 位的有效数字，超过这个长度的位数会被截取掉这样就造成了精度丢失的问题。这是第一个会造成精度丢失的地方。在对两个以 64 位双精度格式的数据进行计算的时候，首先会进行对阶的处理，对阶指的是将阶码对齐，也就是将小数点的位置对齐后，再进行计算，一般是小阶向大阶对齐，因此小阶的数在对齐的过程中，有效数字会向右移动，移动后超过有效位数的位会被截取掉，这是第二个可能会出现精度丢失的地方。当两个数据阶码对齐后，进行相加运算后，得到的结果可能会超过 53 位有效数字，因此超过的位数也会被截取掉，这是可能发生精度丢失的第三个地方。

对于这样的情况，我们可以将其转换为整数后再进行运算，运算后再转换为对应的小数，以这种方式来解决这个问题。

我们还可以将两个数相加的结果和右边相减，如果相减的结果小于一个极小数，那么我们就可以认定结果是相等的，这个极小数可以

使用 es6 的 `Number.EPSILON`

详细资料可以参考：[《十进制的 0.1 为什么不能用二进制很好的表示？》](#) [《十进制浮点数转成二进制》](#) [《浮点数的二进制表示》](#) [《js 浮点数存储精度丢失原理》](#) [《浮点数精度之谜》](#) [《JavaScript 浮点数陷阱及解法》](#) [《 \$0.1+0.2 \neq 0.3\$?》](#) [《JavaScript 中奇特的~运算符》](#)

3.100. 原码、反码和补码的介绍

原码是计算机中对数字的二进制的定点表示方法，最高位表示符号位，其余位表示数值位。优点是易于分辨，缺点是不能够直接参与运算。

正数的反码和其原码一样；负数的反码，符号位为 1，数值部分按原码取反。

如 $[+7]_{\text{原}} = 00000111$, $[+7]_{\text{反}} = 00000111$; $[-7]_{\text{原}} = 10000111$, $[-7]_{\text{反}} = 11111000$ 。

正数的补码和其原码一样；负数的补码为其反码加 1。

例如 $[+7]_{\text{原}} = 00000111$, $[+7]_{\text{反}} = 00000111$, $[+7]_{\text{补}} = 00000111$;

$[-7]_{\text{原}} = 10000111$, $[-7]_{\text{反}} = 11111000$, $[-7]_{\text{补}} = 11111001$

之所以在计算机中使用补码来表示负数的原因是，这样可以将加法运算扩展到所有的数值计算上，因此在数字电路中我们只需要考虑加法器的设计就行了，而不用再为减法设置新的数字电路。

详细资料可以参考： [《关于 2 的补码》](#)

3.101. `toPrecision` 和 `toFixed` 和 `Math.round` 的区别？

`toPrecision` 用于处理精度，精度是从左至右第一个不为 0 的数开始数起。

`toFixed` 是对小数点后指定位数取整，从小数点开始数起。

`Math.round` 是将一个数字四舍五入到一个整数。

3.102. 什么是 XSS 攻击？如何防范 XSS 攻击？

XSS 攻击指的是跨站脚本攻击，是一种代码注入攻击。攻击者通过在网站注入恶意脚本，使之在用户的浏览器上运行，从而盗取用户的信息如 `cookie` 等。

XSS 的本质是因为网站没有对恶意代码进行过滤，与正常的代码混合在一起了，浏览器没有办法分辨哪些脚本是可信的，从而导致了恶意代码的执行。

XSS 一般分为存储型、反射型和 DOM 型。

存储型指的是恶意代码提交到了网站的数据库中，当用户请求数据的时候，服务器将其拼接为 `HTML` 后返回给了用户，从而导致了恶意代码的执行。

反射型指的是攻击者构建了特殊的 URL，当服务器接收到请求后，从 URL 中获取数据，拼接到 HTML 后返回，从而导致了恶意代码的执行。

DOM 型指的是攻击者构建了特殊的 URL，用户打开网站后，js 脚本从 URL 中获取数据，从而导致了恶意代码的执行。

XSS 攻击的预防可以从两个方面入手，一个是恶意代码提交的时候，一个是浏览器执行恶意代码的时候。

对于第一个方面，如果我们对存入数据库的数据都进行的转义处理，但是一个数据可能在多个地方使用，有的地方可能不需要转义，由于我们没有办法判断数据最后的使用场景，所以直接在输入端进行恶意代码的处理，其实是不太可靠的。

因此我们可以从浏览器的执行来进行预防，一种是使用纯前端的方式，不用服务器端拼接后返回。另一种是对需要插入到 HTML 中的代码做好充分的转义。对于 DOM 型的攻击，主要是前端脚本的不可靠而造成的，我们对于数据获取渲染和字符串拼接的时候应该对可能出现的恶意代码情况进行判断。

还有一些方式，比如使用 CSP，CSP 的本质是建立一个白名单，告诉浏览器哪些外部资源可以加载和执行，从而防止恶意代码的注入攻击。

还可以对一些敏感信息进行保护，比如 cookie 使用 http-only，使得脚本无法获取。也可以使用验证码，避免脚本伪装成用户执行一些操作。

详细资料可以参考：《前端安全系列（一）：如何防止 XSS 攻击？》

3.103. 什么是 CSP？

CSP 指的是内容安全策略，它的本质是建立一个白名单，告诉浏览器哪些外部资源可以加载和执行。我们只需要配置规则，如何拦截由浏览器自己来实现。

通常有两种方式来开启 CSP，一种是设置 HTTP 首部中的 Content-Security-Policy，一种是设置 meta 标签的方式 <meta

```
http-equiv="Content-Security-Policy">
```

详细资料可以参考： [《内容安全策略（CSP）》](#) [《前端面试之道》](#)

3.104. 什么是 CSRF 攻击？如何防范 CSRF 攻击？

CSRF 攻击指的是跨站请求伪造攻击，攻击者诱导用户进入一个第三方网站，然后该网站向被攻击网站发送跨站请求。如果用户在被

攻击网站中保存了登录状态，那么攻击者就可以利用这个登录状态，绕过后台的用户验证，冒充用户向服务器执行一些操作。

CSRF 攻击的本质是利用了 cookie 会在同源请求中携带发送给服务器的特点，以此来实现用户的冒充。

一般的 CSRF 攻击类型有三种：

第一种是 GET 类型的 CSRF 攻击，比如在网站中的一个 img 标签里构建一个请求，当用户打开这个网站的时候就会自动发起提交。

第二种是 POST 类型的 CSRF 攻击，比如说构建一个表单，然后隐藏它，当用户进入页面时，自动提交这个表单。

第三种是链接类型的 CSRF 攻击，比如说在 a 标签的 href 属性里构建一个请求，然后诱导用户去点击。

CSRF 可以用下面几种方法来防护：

第一种是同源检测的方法，服务器根据 `http` 请求头中 `origin` 或者 `referer` 信息来判断请求是否为允许访问的站点，从而对请求进行过滤。当 `origin` 或者 `referer` 信息都不存在的时候，直接阻止。这种方式的缺点是有些情况下 `referer` 可以被伪造。还有就是我们这种方法同时把搜索引擎的链接也给屏蔽了，所以一般网站会允许搜索引擎的页面请求，但是相应的页面请求这种请求方式也可能被攻击者给利用。

第二种方法是使用 `CSRF Token` 来进行验证，服务器向用户返回一个随机数 `Token`，当网站再次发起请求时，在请求参数中加入服务器端返回的 `token`，然后服务器对这个 `token` 进行验证。这种方法解决了使用 `cookie` 单一验证方式时，可能会被冒用的问题，但是这种方法存在一个缺点就是，我们需要给网站中的所有请求都添加上这个 `token`，操作比较繁琐。还有一个问题是如果只有一台网站服务器，如果我们的请求经过负载平衡转移到了其他的服务器，但是这个服务器的 `session` 中没有保留这个 `token` 的话，就没有办法验证了。这种情况我们可以通过改变 `token` 的构建方式来解决。

第三种方式使用双重 `Cookie` 验证的办法，服务器在用户访问网站页面时，向请求域名注入一个 `Cookie`，内容为随机字符串，然后当用户再次向服务器发送请求的时候，从 `cookie` 中取出这个字符串，添加到 `URL` 参数中，然后服务器通过对 `cookie` 中的数据和参数中的数据进行比较，来进行验证。使用这种方式是利用了攻击者只能利用 `cookie`，但是不能访问获取 `cookie` 的特点。并且这种方法比 `CSRF Token` 的方法更加方便，并且不涉及到分布式访问的问题。这种方法的缺点是如果网站存在 `XSS` 漏洞的，那么这种方式会失效。同时这种方式不能做到子域名的隔离。

第四种方式是使用在设置 `cookie` 属性的时候设置 `Samesite`，限制 `cookie` 不能作为被第三方使用，从而可以避免被攻击者利用。`Samesite` 一共有两种模式，一种是严格模式，在严格模式下 `cookie` 在任何情况下都不可能作为第三方 `Cookie` 使用，在宽松模式下，`cookie` 可以被请求是 `GET` 请求，且会发生页面跳转的请求所使用。

详细资料可以参考： [《前端安全系列之二：如何防止 CSRF 攻击？》](#) [《\[HTTP 趣谈\] origin, referer 和 host 区别》](#)

3.105. 什么是 `Samesite Cookie` 属性？

`Samesite Cookie` 表示同站 `cookie`，避免 `cookie` 被第三方所利用。

将 `Samesite` 设为 `strict`，这种称为严格模式，表示这个 `cookie` 在任何情况下都不可能作为第三方 `cookie`。

将 `Samesite` 设为 `Lax`，这种模式称为宽松模式，如果这个请求是个 `GET` 请求，并且这个请求改变了当前页面或者打开了新的页面，那么这个 `cookie` 可以作为第三方 `cookie`，其余情况下都不能作为第三方 `cookie`。

使用这种方法的缺点是，因为它不支持子域，所以子域没有办法与主域共享登录信息，每次转入子域的网站，都回重新登录。还有一个问题就是它的兼容性不够好。

3.106. 什么是点击劫持？如何防范点击劫持？

点击劫持是一种视觉欺骗的攻击手段，攻击者将需要攻击的网站通过 `iframe` 嵌套的方式嵌入自己的网页中，并将 `iframe` 设置为透明，在页面中透出一个按钮诱导用户点击。

我们可以在 `http` 相应头中设置 `X-FRAME-OPTIONS` 来防御用 `iframe` 嵌套的点击劫持攻击。通过不同的值，可以规定页面在特

定的一些情况才能作为 `iframe` 来使用。

详细资料可以参考： [《web 安全之--点击劫持攻击与防御技术简介》](#)

3.107. SQL 注入攻击？

SQL 注入攻击指的是攻击者在 `HTTP` 请求中注入恶意的 `SQL` 代码，服务器使用参数构建数据库 `SQL` 命令时，恶意 `SQL` 被一起构

造，破坏原有 `SQL` 结构，并在数据库中执行，达到编写程序时意料之外结果的攻击行为。

详细资料可以参考： [《Web 安全漏洞之 SQL 注入》](#) [《如何防范常见的 Web 攻击》](#)

3.108. 什么是 MVVM？比之 MVC 有什么区别？什么又是 MVP ？

MVC、MVP 和 MVVM 是三种常见的软件架构设计模式，主要通过分离关注点的方式来组织代码结构，优化我们的开发效率。

比如说我们实验室在以前项目开发的时候，使用单页应用时，往往一个路由页面对应了一个脚本文件，所有的页面逻辑都在一个脚本文件里。页面的渲染、数据的获取，对用户事件的响应所有的应用逻辑都混合在一起，这样在开发简单项目时，可能看不出什么问题，当时一旦项目变得复杂，那么整个文件就会变得冗长，混乱，这样对我们的项目开发和后期的项目维护是非常不利的。

MVC 通过分离 **Model**、**View** 和 **Controller** 的方式来组织代码结构。其中 **View** 负责页面的显示逻辑，**Model** 负责存储页面的业务数据，以及对相应数据的操作。并且 **View** 和 **Model** 应用了观察者模式，当 **Model** 层发生改变的时候它会通知有关 **View** 层更新页面。**Controller** 层是 **View** 层和 **Model** 层的纽带，它主要负责用户与应用的响应操作，当用户与页面产生交互的时候，**Co**

ntroller 中的事件触发器就开始工作了，通过调用 **Model** 层，来完成对 **Model** 的修改，然后 **Model** 层再去通知 **View** 层更新。

MVP 模式与 **MVC** 唯一不同的在于 **Presenter** 和 **Controller**。在 **MVC** 模式中我们使用观察者模式，来实现当 **Model** 层数据发生变化的时候，通知 **View** 层的更新。这样 **View** 层和 **Model** 层耦合在一起，当项目逻辑变得复杂的时候，可能会造成代码的混乱，并且可能会对代码的复用性造成一些问题。**MVP** 的模式通过使用 **Presenter** 来实现对 **View** 层和 **Model** 层的解耦。**MVC** 中的

Controller 只知道 **Model** 的接口，因此它没有办法控制 **View** 层的更新，**MVP** 模式中，**View** 层的接口暴露给了 **Presenter** 因此我们可以在 **Presenter** 中将 **Model** 的变化和 **View** 的变化绑定在一起，以此来实现 **View** 和 **Model** 的同步更新。这样就实现了对 **View** 和 **Model** 的解耦，**Presenter** 还包含了其他的响应逻辑。

MVVM 模式中的 **VM**，指的是 **ViewModel**，它和 **MVP** 的思想其实是相同的，不过它通过双向的数据绑定，将 **View** 和 **Model** 的同步更新给自动化了。当 **Model** 发生变化的时候，**ViewModel** 就会自动更新；**ViewModel** 变化了，**View** 也会更新。这样就将 **Presenter** 中的工作给自动化了。我了解过一点双向数据绑定的原理，比如 **vue** 是通过使用数据劫持和发布订阅者模式来实现的这一功

能。

详细资料可以参考：[《浅析前端开发中的 MVC/MVP/MVVM 模式》](#) [《MVC, MVP 和 MVVM 的图示》](#) [《MVVM》](#) [《一篇文章了解架构模式：MVC/MVP/MVVM》](#)

3.109. vue 双向数据绑定原理？

vue 通过使用双向数据绑定，来实现了 View 和 Model 的同步更新。vue 的双向数据绑定主要是通过使用数据劫持和发布订阅者模式来实现的。

首先我们通过 `Object.defineProperty()` 方法来对 Model 数据各个属性添加访问器属性，以此来实现数据的劫持，因此当 Model 中的数据发生变化的时候，我们可以通过配置的 `setter` 和 `getter` 方法来实现对 View 层数据更新的通知。

数据在 `html` 模板中一共有两种绑定情况，一种是使用 `v-model` 来对 `value` 值进行绑定，一种是作为文本绑定，在对模板引擎进行解析的过程中。

如果遇到元素节点，并且属性值包含 `v-model` 的话，我们就从 Model 中去获取 `v-model` 所对应的属性的值，并赋值给元素的 `value` 值。然后给这个元素设置一个监听事件，当 View 中元素的数据发生变化的时候触发该事件，通知 Model 中的对应的属性的值进行更新。

如果遇到了绑定的文本节点，我们使用 Model 中对应的属性的值来替换这个文本。对于文本节点的更新，我们使用了发布订阅者模式，属性作为一个主题，我们为这个节点设置一个订阅者对象，将这个订阅者对象加入这个属性主题的订阅者列表中。当 Model 层数据发生改变的时候，Model 作为发布者向主题发出通知，主题收到通知再向它的所有订阅者推送，订阅者收到通知后更改自己的数

据。

详细资料可以参考： [《Vue.js 双向绑定的实现原理》](#)

3.110. Object.defineProperty 介绍？

`Object.defineProperty` 函数一共有三个参数，第一个参数是需要定义属性的对象，第二个参数是需要定义的属性，第三个是该属性描述符。

一个属性的描述符有四个属性，分别是 `value` 属性的值，`writable` 属性是否可写，`enumerable` 属性是否可枚举，`configurable` 属性是否可配置修改。

详细资料可以参考： [《Object.defineProperty\(\)》](#)

3.111. 使用 `Object.defineProperty()` 来进行数据劫持有什么缺点？

有一些对属性的操作，使用这种方法无法拦截，比如说通过下标方式修改数组数据或者给对象新增属性，`vue` 内部通过重写函数解决了这个问题。在 `Vue3.0` 中已经不使用这种方式了，而是通过使用 `Proxy` 对对象进行代理，从而实现数据劫持。使用 `Proxy` 的好处是它可以完美的监听到任何方式的数据改变，唯一的缺点是兼容性的问题，因为这是 `ES6` 的语法。

3.112. 什么是 `Virtual DOM`？为什么 `Virtual DOM` 比原生 `DOM` 快？

我对 `Virtual DOM` 的理解是，

首先对我们将来要插入到文档中的 `DOM` 树结构进行分析，使用 `js` 对象将其表示出来，比如一个元素对象，包含 `TagName`、`props` 和 `Children` 这些属性。然后我们将这个 `js` 对象树给保存下来，最后再将 `DOM` 片段插入到文档中。

当页面的状态发生改变，我们需要对页面的 `DOM` 的结构进行调整的时候，我们首先根据变更的状态，重新构建起一棵对象树，然后将这棵新的对象树和旧的对象树进行比较，记录下两棵树的差异。

最后将记录的有差异的地方应用到真正的 `DOM` 树中去，这样视图就更新了。

我认为 `Virtual DOM` 这种方法对于我们需要有大量的 `DOM` 操作的时候，能够很好的提高我们的操作效率，通过在操作前确定需要做的最小修改，尽可能的减少 `DOM` 操作带来的重流和重绘的影响。其实 `Virtual DOM` 并不一定比我们真实的操作 `DOM` 要快，这种方法的目的是为了提高我们开发时的可维护性，在任意的情况下，都能保证一个尽量小的性能消耗去进行操作。

详细资料可以参考： [《Virtual DOM》](#) [《理解 Virtual DOM》](#) [《深度剖析：如何实现一个 Virtual DOM 算法》](#) [《网上都说操作真实 DOM 慢，但测试结果却比 React 更快，为什么？》](#)

3.113. 如何比较两个 DOM 树的差异?

两个树的完全 `diff` 算法的时间复杂度为 $O(n^3)$ ，但是在前端中，我们很少会跨层级的移动元素，所以我们只需要比较同一层级的元素进行比较，这样就可以将算法的时间复杂度降低为 $O(n)$ 。

算法首先会对新旧两棵树进行一个深度优先的遍历，这样每个节点都会有一个序号。在深度遍历的时候，每遍历到一个节点，我们就将这个节点和新的树中的节点进行比较，如果有差异，则将这个差异记录到一个对象中。

在对列表元素进行对比的时候，由于 `TagName` 是重复的，所以我们不能使用这个来对比。我们需要给每一个子节点加上一个 `key`，列表对比的时候使用 `key` 来进行比较，这样我们才能够复用老的 DOM 树上的节点。

3.114. 什么是 `requestAnimationFrame` ?

详细资料可以参考： [《你需要知道的 requestAnimationFrame》](#) [《CSS3 动画那么强，requestAnimationFrame 还有毛线用？》](#)

3.115. 谈谈你对 `webpack` 的看法

我当时使用 `webpack` 的一个最主要原因是为了简化页面依赖的管理，并且通过将其打包为一个文件来降低页面加载时请求的资源数。

我认为 `webpack` 的主要原理是，它将所有的资源都看成是一个模块，并且把页面逻辑当作一个整体，通过一个给定的入口文件，`webpack` 从这个文件开始，找到所有的依赖文件，将各个依赖文件模块通过 `loader` 和 `plugins` 处理后，然后打包在一起，最后输出一个浏览器可识别的 `JS` 文件。

`Webpack` 具有四个核心的概念，分别是 `Entry`（入口）、`Output`（输出）、`loader` 和 `Plugins`（插件）。

`Entry` 是 `webpack` 的入口起点，它指示 `webpack` 应该从哪个模块开始着手，来作为其构建内部依赖图的开始。

`Output` 属性告诉 `webpack` 在哪里输出它所创建的打包文件，也可指定打包文件的名称，默认位置为 `./dist`。

`loader` 可以理解为 `webpack` 的编译器，它使得 `webpack` 可以处理一些非 `JavaScript` 文件。在对 `loader` 进行配置的时候，`test` 属性，标志有哪些后缀的文件应该被处理，是一个正则表达式。`use` 属性，指定 `test` 类型的文件应该使用哪个 `loader` 进行预处理。常用的 `loader` 有 `css-loader`、`style-loader` 等。

插件可以用于执行范围更广的任务，包括打包、优化、压缩、搭建服务器等等，要使用一个插件，一般是先使用 `npm` 包管理器进行安装，然后在配置文件中引入，最后将其实例化后传递给 `plugins` 数组属性。

使用 `webpack` 的确能够提供我们对于项目的管理，但是它的缺点就是调试和配置起来太麻烦了。但现在 `webpack4.0` 的免配置一定程度上解决了这个问题。但是我感觉就是对我来说，就是一个黑盒，很多时候出现了问题，没有办法很好的定位。

详细资料可以参考：[《不聊 webpack 配置，来说说它的原理》](#) [《前端工程化——构建工具选型：grunt、gulp、webpack》](#) [《浅入浅出 webpack》](#) [《前端构建工具发展及其比较》](#)

3.116. `offsetWidth/offsetHeight,clientWidth/clientHeight` 与 `scrollWidth/scrollHeight` 的区别？

`clientWidth/clientHeight` 返回的是元素的内部宽度，它的值只包含 `content + padding`，如果有滚动条，不包含滚动条。

`clientTop` 返回的是上边框的宽度。

`clientLeft` 返回的左边框的宽度。

`offsetWidth/offsetHeight` 返回的是元素的布局宽度，它的值包含 `content + padding + border` 包含了滚动条。

`offsetTop` 返回的是当前元素相对于其 `offsetParent` 元素的顶部的距离。

`offsetLeft` 返回的是当前元素相对于其 `offsetParent` 元素的左部的距离。

`scrollWidth/scrollHeight` 返回值包含 `content + padding + 溢出内容的尺寸`。

`scrollTop` 属性返回的是一个元素的内容垂直滚动的像素数。

`scrollLeft` 属性返回的是元素滚动条到元素左边的距离。

详细资料可以参考： [《最全的获取元素宽高及位置的方法》](#) [《用 Javascript 获得页面元素的位置》](#)



3.117. 谈一谈你理解的函数式编程？

简单说，"函数式编程"是一种"编程范式"（programming paradigm），也就是如何编写程序的方法论。

它具有以下特性：闭包和高阶函数、惰性计算、递归、函数是"第一等公民"、只用"表达式"。

详细资料可以参考： [《函数式编程初探》](#)



3.118. 异步编程的实现方式？

相关资料：

回调函数

优点：简单、容易理解

缺点：不利于维护，代码耦合高

事件监听（采用时间驱动模式，取决于某个事件是否发生）：

优点：容易理解，可以绑定多个事件，每个事件可以指定多个回调函数

缺点：事件驱动型，流程不够清晰

发布/订阅（观察者模式）

类似于事件监听，但是可以通过‘消息中心’，了解现在有多少发布者，多少订阅者

Promise 对象

优点：可以利用 `then` 方法，进行链式写法；可以书写错误时的回调函数；

缺点：编写和理解，相对比较难

Generator 函数

优点：函数体内外的数据交换、错误处理机制

缺点：流程管理不方便

async 函数

优点：内置执行器、更好的语义、更广的适用性、返回的是 `Promise`、结构清晰。

缺点：错误处理机制

回答：

js 中的异步机制可以分为以下几种：

第一种最常见的是使用回调函数的方式，使用回调函数的方式有一个缺点是，多个回调函数嵌套的时候会造成回调函数地狱，上下两层的回调函数间的代码耦合度太高，不利于代码的可维护。

第二种是 `Promise` 的方式，使用 `Promise` 的方式可以将嵌套的回调函数作为链式调用。但是使用这种方法，有时会造成多个 `then` 的链式调用，可能会造成代码的语义不够明确。

第三种是使用 `generator` 的方式，它可以在函数的执行过程中，将函数的执行权转移出去，在函数外部我们还可以将执行权转移回来。当我们遇到异步函数执行的时候，将函数执行权转移出去，当异步函数执行完毕的时候我们再将执行权给转移回来。因此我们在 `generator` 内部对于异步操作的方式，可以以同步的顺序来书写。使用这种方式我们需要考虑的问题是何时将函数的控制权转移回来，因此我们需要有一个自动执行 `generator` 的机制，比如说 `co` 模块等方式来实现 `generator` 的自动执行。

第四种是使用 `async` 函数的形式，`async` 函数是 `generator` 和 `promise` 实现的一个自动执行的语法糖，它内部自带执行器，当函数内部执行到一个 `await` 语句的时候，如果语句返回一个 `promise` 对象，那么函数将会等待 `promise` 对象的状态变为 `resolve` 后再继续向下执行。因此我们可以将异步逻辑，转化为同步的顺序来书写，并且这个函数可以自动执行。

3.119. Js 动画与 CSS 动画区别及相应实现

CSS3 的动画的优点

在性能上会稍微好一些，浏览器会对 CSS3 的动画做一些优化

代码相对简单

缺点

在动画控制上不够灵活

兼容性不好

JavaScript 的动画正好弥补了这两个缺点，控制能力很强，可以单帧的控制、变换，同时写得好完全可以兼容 IE6，并且功能强大。对于一些复杂控制的动画，使用 javascript 会比较靠谱。而在实现一些小的交互动效的时候，就多考虑考虑 CSS 吧

3.120. get 请求传参长度的误区

误区：我们经常说 `get` 请求参数的大小存在限制，而 `post` 请求的参数大小是无限制的。

实际上 HTTP 协议从未规定 GET/POST 的请求长度限制是多少。对 `get` 请求参数的限制是来源与浏览器或 web 服务器，浏览器或 web 服务器限制了 url 的长度。为了明确这个概念，我们必须再次强调下面几点：

- 1.HTTP 协议未规定 GET 和 POST 的长度限制

- 2.GET 的最大长度显示是因为浏览器和 web 服务器限制了 URI 的长度
- 3.不同的浏览器和 WEB 服务器，限制的最大长度不一样
- 4.要支持 IE，则最大长度为 2083byte，若只支持 Chrome，则最大长度 8182byte

3.121. URL 和 URI 的区别？

URI: Uniform Resource Identifier 指的是统一资源标识符

URL: Uniform Resource Location 指的是统一资源定位符

URN: Universal Resource Name 指的是统一资源名称

URI 指的是统一资源标识符，用唯一的标识来确定一个资源，它是一种抽象的定义，也就是说，不管使用什么方法来定义，只要能唯一的标识一个资源，就可以称为 URI。

URL 指的是统一资源定位符，URN 指的是统一资源名称。URL 和 URN 是 URI 的子集，URL 可以理解为使用地址来标识资源，URN 可以理解为使用名称来标识资源。

详细资料可以参考：[《HTTP 协议中 URI 和 URL 有什么区别？》](#) [《你知道 URL、URI 和 URN 三者之间的区别吗？》](#) [《URI、URL 和 URN 的区别》](#)

3.122. get 和 post 请求在缓存方面的区别

相关知识点：

get 请求类似于查找的过程，用户获取数据，可以不用每次都与数据库连接，所以可以使用缓存。

post 不同，post 做的一般是修改和删除的工作，所以必须与数据库交互，所以不能使用缓存。因此 get 请求适合于请求缓存。

回答：

缓存一般只适用于那些不会更新服务端数据的请求。一般 `get` 请求都是查找请求，不会对服务器资源数据造成修改，而 `post` 请求一般都会对服务器数据造成修改，所以，一般会对 `get` 请求进行缓存，很少会对 `post` 请求进行缓存。

详细资料可以参考： [《HTML 关于 post 和 get 的区别以及缓存问题的理解》](#)

3.123. 图片的懒加载和预加载

相关知识点：

预加载： 提前加载图片，当用户需要查看时可直接从本地缓存中渲染。

懒加载： 懒加载的主要目的是作为服务器前端的优化，减少请求数或延迟请求数。

两种技术的本质：两者的行为是相反的，一个是提前加载，一个是迟缓甚至不加载。 懒加载对服务器前端有一定的缓解压力作用，预加载则会增加服务器前端压力。

回答：

懒加载也叫延迟加载，指的是在长网页中延迟加载图片的时机，当用户需要访问时，再去加载，这样可以提高网站的首屏加载速度，提升用户的体验，并且可以减少服务器的压力。它适用于图片很多，页面很长的电商网站的场景。懒加载的实现原理是，将页面上的图片的 `src` 属性设置为空字符串，将图片的真实路径保存在一个自定义属性中，当页面滚动的时候，进行判断，如果图片进入页面可视区域内，则从自定义属性中取出真实路径赋值给图片的 `src` 属性，以此来实现图片的延迟加载。

预加载指的是将所需的资源提前请求加载到本地，这样后面在需要用到时就直接从缓存取资源。通过预加载能够减少用户的等待时间，提高用户的体验。我了解的预加载的最常用的方式是使用 `js` 中的 `image` 对象，通过为 `image` 对象来设置 `scr` 属性，来实现图片的预加载。

这两种方式都是提高网页性能的方式，两者主要区别是一个是提前加载，一个是迟缓甚至不加载。 懒加载对服务器前端有一定的缓解压力作用，预加载则会增加服务器前端压力。

详细资料可以参考： [《懒加载和预加载》](#) [《网页图片加载优化方案》](#) [《基于用户行为的图片等资源预加载》](#)

3.124. mouseover 和 mouseenter 的区别?

当鼠标移动到元素上时就会触发 `mouseenter` 事件，类似 `mouseover`，它们两者之间的差别是 `mouseenter` 不会冒泡。

由于 `mouseenter` 不支持事件冒泡，导致在一个元素的子元素上进入或离开的时候会触发其 `mouseover` 和 `mouseout` 事件，但是却不会触发 `mouseenter` 和 `mouseleave` 事件。

详细资料可以参考： [《mouseenter 与 mouseover 为何这般纠缠不清？》](#)

3.125. js 拖拽功能的实现

相关知识点：

首先是三个事件，分别是 `mousedown`, `mousemove`, `mouseup`

当鼠标点击按下的时候，需要一个 `tag` 标识此时已经按下，可以执行 `mousemove` 里面的具体方法。

`clientX`, `clientY` 标识的是鼠标的坐标，分别标识横坐标和纵坐标，并且我们用 `offsetX` 和 `offsetY` 来表示

元素的元素的初始坐标，移动的举例应该是：

鼠标移动时候的坐标-鼠标按下去时候的坐标。

也就是说定位信息为：

鼠标移动时候的坐标-鼠标按下去时候的坐标+元素初始情况下的 `offsetLeft`.

回答：

一个元素的拖拽过程，我们可以分为三个步骤，第一步是鼠标按下目标元素，第二步是鼠标保持按下的状态移动鼠标，第三步是鼠标抬起，拖拽过程结束。

这三步分别对应了三个事件，`mousedown` 事件，`mousemove` 事件和 `mouseup` 事件。只有在鼠标按下的状态移动鼠标我们才会

执行拖拽事件，因此我们需要在 `mousedown` 事件中设置一个状态来标识鼠标已经按下，然后在 `mouseup` 事件中再取消这个状态。

在 `mousedown` 事件中我们首先应该判断，目标元素是否为拖拽元素，如果是拖拽元素，我们就设置状态并且保存这个时候鼠

标的位罝。然后在 `mousemove` 事件中，我们通过判断鼠标现在的位置和以前位置的相对移动，来确定拖拽元素在移动中的坐标。最后 `mouseup` 事件触发后，清除状态，结束拖拽事件。

详细资料可以参考： [《原生 js 实现拖拽功能基本思路》](#)

3.126. 为什么使用 `setTimeout` 实现 `setInterval`? 怎么模拟?

相关知识点：

```
// 思路是使用递归函数，不断地去执行 setTimeout 从而达到 setInterval 的效果

function mySetInterval(fn, timeout) {
    // 控制器，控制定时器是否继续执行

    var timer = {
        flag: true
    };

    // 设置递归函数，模拟定时器执行。
    function interval() {
        if (timer.flag) {
            fn();
            setTimeout(interval, timeout);
        }
    }

    // 启动定时器
    interval();
}
```

```
setTimeout(interval, timeout);

// 返回控制器

return timer;}
```

回答：

`setInterval` 的作用是每隔一段指定时间执行一个函数，但是这个执行不是真的到了时间立即执行，它真正的作用是每隔一段时间将事件加入事件队列中去，只有当当前的执行栈为空的时候，才能去从事件队列中取出事件执行。所以可能会出现这样的情况，就是当前执行栈执行的时间很长，导致事件队列里边积累多个定时器加入的事件，当执行栈结束的时候，这些事件会依次执行，因此就不能到间隔一段时间执行的效果。

针对 `setInterval` 的这个缺点，我们可以使用 `setTimeout` 递归调用来模拟 `setInterval`，这样我们就确保了只有一个事件结束了，我们才会触发下一个定时器事件，这样解决了 `setInterval` 的问题。

详细资料可以参考： [《用 setTimeout 实现 setInterval》](#) [《setInterval 有什么缺点？》](#)



3.127. `let` 和 `const` 的注意点？

- 1. 声明的变量只在声明时的代码块内有效
- 2. 不存在声明提升
- 3. 存在暂时性死区，如果在变量声明前使用，会报错
- 4. 不允许重复声明，重复声明会报错

3.128. 什么是 `rest` 参数？

`rest` 参数（形式为`...变量名`），用于获取函数的多余参数。

3.129. 什么是尾调用，使用尾调用有什么好处？

尾调用指的是函数的最后一步调用另一个函数。我们代码执行是基于执行栈的，所以当我们在一个函数里调用另一个函数时，我们会保留当前的执行上下文，然后再新建另外一个执行上下文加入栈中。使用尾调用的话，因为已经是函数的最后一步，所以这个时候我们可以不必再保留当前的执行上下文，从而节省了内存，这就是尾调用优化。但是 ES6 的尾调用优化只在严格模式下开启，正常模式是无效的。

3.130. Symbol 类型的注意点？

- 1.Symbol 函数前不能使用 new 命令，否则会报错。
- 2.Symbol 函数可以接受一个字符串作为参数，表示对 Symbol 实例的描述，主要是为了在控制台显示，或者转为字符串时，比较容易区分。
- 3.Symbol 作为属性名，该属性不会出现在 for...in、for...of 循环中，也不会被 Object.keys()、Object.getOwnPropertyNames()、JSON.stringify() 返回。
- 4.Object.getOwnPropertySymbols 方法返回一个数组，成员是当前对象的所有用作属性名的 Symbol 值。
- 5.Symbol.for 接受一个字符串作为参数，然后搜索有没有以该参数作为名称的 Symbol 值。如果有，就返回这个 Symbol 值，否则就新建并返回一个以该字符串为名称的 Symbol 值。
- 6.Symbol.keyFor 方法返回一个已登记的 Symbol 类型值的 key。

3.131. Set 和 WeakSet 结构？

- 1.ES6 提供了新的数据结构 Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。
- 2.WeakSet 结构与 Set 类似，也是不重复的值的集合。但是 WeakSet 的成员只能是对象，而不能是其他类型的值。WeakSet 中的对象都是弱引用，即垃圾回收机制不考虑 WeakSet 对该对象的引用，

3.132. Map 和 WeakMap 结构?

- 1.Map 数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。
- 2.WeakMap 结构与 Map 结构类似，也是用于生成键值对的集合。但是 WeakMap 只接受对象作为键名（`null` 除外），不接受其他类型的值作为键名。而且 WeakMap 的键名所指向的对象，不计入垃圾回收机制。

3.133. 什么是 Proxy ?

`Proxy` 用于修改某些操作的默认行为，等同于在语言层面做出修改，所以属于一种“元编程”，即对编程语言进行编程。

`Proxy` 可以理解成，在目标对象之前架设一层“拦截”，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。`Proxy` 这个词的原意是代理，用在这里表示由它来“代理”某些操作，可以译为“代理器”。

3.134. Reflect 对象创建目的?

- 1. 将 `Object` 对象的一些明显属于语言内部的方法（比如 `Object.defineProperty`，放到 `Reflect` 对象上。
- 2. 修改某些 `Object` 方法的返回结果，让其变得更合理。
- 3. 让 `Object` 操作都变成函数行为。
- 4. `Reflect` 对象的方法与 `Proxy` 对象的方法一一对应，只要是 `Proxy` 对象的方法，就能在 `Reflect` 对象上找到对应的方法。这就让 `Proxy` 对象可以方便地调用对应的 `Reflect` 方法，完成默认行为，作为修改行为的基础。

也就是说，不管 `Proxy` 怎么修改默认行为，你总可以在 `Reflect` 上获取默认行为。

3.135. `require` 模块引入的查找方式？

当 `Node` 遇到 `require(X)` 时，按下面的顺序处理。

(1) 如果 `X` 是内置模块（比如 `require('http')`）

- a. 返回该模块。
- b. 不再继续执行。

(2) 如果 `X` 以 `"./"` 或者 `"/"` 或者 `"../"` 开头

- a. 根据 `X` 所在的父模块，确定 `X` 的绝对路径。
- b. 将 `X` 当成文件，依次查找下面文件，只要其中有一个存在，就返回该文件，不再继续执行。

`X`
`X.js`
`X.json`
`X.node`

- c. 将 `X` 当成目录，依次查找下面文件，只要其中有一个存在，就返回该文件，不再继续执行。

`X/package.json` (`main` 字段)
`X/index.js`
`X/index.json`
`X/index.node`

(3) 如果 X 不带路径

- a. 根据 X 所在的父模块，确定 X 可能的安装目录。
- b. 依次在每个目录中，将 X 当成文件名或目录名加载。

(4) 抛出 "not found"

详细资料可以参考： [《require\(\) 源码解读》](#)

3.136. 什么是 Promise 对象，什么是 Promises/A+ 规范？

Promise 对象是异步编程的一种解决方案，最早由社区提出。Promises/A+ 规范是 JavaScript Promise 的标准，规定了一个 Promise 所必须具有的特性。

Promise 是一个构造函数，接收一个函数作为参数，返回一个 Promise 实例。一个 Promise 实例有三种状态，分别是 pending、resolved 和 rejected，分别代表了进行中、已成功和已失败。实例的状态只能由 pending 转变 resolved 或者 rejected 状态，并且状态一经改变，就凝固了，无法再被改变了。状态的改变是通过 resolve() 和 reject() 函数来实现的，我们

可以在异步操作结束后调用这两个函数改变 Promise 实例的状态，它的原型上定义了一个 then 方法，使用这个 then 方法可以为两个状态的改变注册回调函数。这个回调函数属于微任务，会在本轮事件循环的末尾执行。

详细资料可以参考： [《Promises/A+ 规范》](#) [《Promise》](#)

3.137. 手写一个 Promise

```
const PENDING = "pending";const RESOLVED = "resolved";const REJECTED = "rejected";

function MyPromise(fn) {
    // 保存初始化状态
    var self = this;
```

```
// 初始化状态

this.state = PENDING;

// 用于保存 resolve 或者 rejected 传入的值

this.value = null;

// 用于保存 resolve 的回调函数

this.resolvedCallbacks = [];

// 用于保存 reject 的回调函数

this.rejectedCallbacks = [];

// 状态转变为 resolved 方法

function resolve(value) {

    // 判断传入元素是否为 Promise 值，如果是，则状态改变必须等待前一个状态改变后再
    // 进行改变

    if (value instanceof MyPromise) {

        return value.then(resolve, reject);

    }

}

// 保证代码的执行顺序为本轮事件循环的末尾

setTimeout(() => {

    // 只有状态为 pending 时才能转变，

    if (self.state === PENDING) {

        // 修改状态

        self.state = RESOLVED;

    }

});
```

```
// 设置传入的值
self.value = value;

// 执行回调函数
self.resolvedCallbacks.forEach(callback => {
    callback(value);
});

}, 0);

}

// 状态转变为 rejected 方法
function reject(value) {
    // 保证代码的执行顺序为本轮事件循环的末尾
    setTimeout(() => {
        // 只有状态为 pending 时才能转变
        if (self.state === PENDING) {
            // 修改状态
            self.state = REJECTED;
        }
    });
}

// 设置传入的值
self.value = value;

// 执行回调函数
```

```
        self.rejectedCallbacks.forEach(callback => {
            callback(value);
        });
    }
}, 0);
}

// 将两个方法传入函数执行

try {
    fn(resolve, reject);
} catch (e) {
    // 遇到错误时，捕获错误，执行 reject 函数
    reject(e);
}

MyPromise.prototype.then = function(onResolved, onRejected) {
    // 首先判断两个参数是否为函数类型，因为这两个参数是可选参数
    onResolved =
        typeof onResolved === "function"
        ? onResolved
        : function(value) {
            return value;
        };
    onRejected =
        typeof onRejected === "function"
```

```
? onRejected

: function(error) {
    throw error;
};

// 如果是等待状态，则将函数加入对应列表中

if (this.state === PENDING) {
    this.resolvedCallbacks.push(onResolved);
    this.rejectedCallbacks.push(onRejected);
}

// 如果状态已经凝固，则直接执行对应状态的函数

if (this.state === RESOLVED) {
    onResolved(this.value);
}

if (this.state === REJECTED) {
    onRejected(this.value);
}
};
```

3.138. 如何检测浏览器所支持的最小字体大小？

用 JS 设置 DOM 的字体为某一个值，然后再取出来，如果值设置成功，就说明支持。

3.139. 怎么做 JS 代码 Error 统计？

error 统计使用浏览器的 window.error 事件。

3.140. 单例模式模式是什么？

单例模式保证了全局只有一个实例来被访问。比如说常用的如弹框组件的实现和全局状态的实现。

3.141. 策略模式是什么？

策略模式主要是用来将方法的实现和方法的调用分离开，外部通过不同的参数可以调用不同的策略。我主要在 **MVP** 模式解耦的时候

用来将视图层的方法定义和方法调用分离。

3.142. 代理模式是什么？

代理模式是为一个对象提供一个代用品或占位符，以便控制对它的访问。比如说常见的事件代理。

4.143. 中介者模式是什么？

中介者模式指的是，多个对象通过一个中介者进行交流，而不是直接进行交流，这样能够将通信的各个对象解耦。

4.144. 适配器模式是什么？

适配器用来解决两个接口不兼容的情况，不需要改变已有的接口，通过包装一层的方式实现两个接口的正常协作。假如我们需要一种

新的接口返回方式，但是老的接口由于在太多地方已经使用了，不能随意更改，这个时候就可以使用适配器模式。比如我们需要一种

自定义的时间返回格式，但是我们又不能对 `js` 时间格式化的接口进行修改，这个时候就可以使用适配器模式。

更多关于设计模式的资料可以参考： [《前端面试之道》](#) [《JavaScript 设计模式》](#)

[《JavaScript 中常见设计模式整理》](#)

3.145. 观察者模式和发布订阅模式有什么不同？

发布订阅模式其实属于广义上的观察者模式

在观察者模式中，观察者需要直接订阅目标事件。在目标发出内容改变的事件后，直接接收事件并作出响应。

而在发布订阅模式中，发布者和订阅者之间多了一个调度中心。调度中心一方面从发布者接收事件，另一方面向订阅者发布事件，订阅者需要在调度中心中订阅事件。通过调度中心实现了发布者和订阅者关系的解耦。使用发布订阅者模式更利于我们代码的可维护性。

详细资料可以参考： [《观察者模式和发布订阅模式有什么不同？》](#)

3.146. Vue 的生命周期是什么？

Vue 的生命周期指的是组件从创建到销毁的一系列的过程，被称为 Vue 的生命周期。通过提供的 Vue 在生命周期各个阶段的钩子函数，我们可以很好的在 Vue 的各个生命阶段实现一些操作。

3.147. Vue 的各个生命阶段是什么？

Vue 一共有 8 个生命阶段，分别是创建前、创建后、加载前、加载后、更新前、更新后、销毁前和销毁后，每个阶段对应了一个生命周期的钩子函数。

(1) `beforeCreate` 钩子函数，在实例初始化之后，在数据监听和事件配置之前触发。因此在这个事件中我们是获取不到 `data` 数据的。

(2) `created` 钩子函数，在实例创建完成后触发，此时可以访问 `data`、`methods` 等属性。但这个时候组件还没有被挂载到页面中去，所以这个时候访问不到 `$el` 属性。一般我们可以在这个函数中进行一些页面初始化的工作，比如通过 `ajax` 请求数据来对页面进行初始化。

(3) `beforeMount` 钩子函数，在组件被挂载到页面之前触发。在 `beforeMount` 之前，会找到对应的 `template`，并编译成 `render` 函数。

(4) `mounted` 钩子函数，在组件挂载到页面之后触发。此时可以通过 `DOM API` 获取到页面中的 `DOM` 元素。

(5) `beforeUpdate` 钩子函数，在响应式数据更新时触发，发生在虚拟 DOM 重新渲染和打补丁之前，这个时候我们可以对可能被移除的元素做一些操作，比如移除事件监听器。

(6) `updated` 钩子函数，虚拟 DOM 重新渲染和打补丁之后调用。

(7) `beforeDestroy` 钩子函数，在实例销毁之前调用。一般在这一步我们可以销毁定时器、解绑全局事件等。

(8) `destroyed` 钩子函数，在实例销毁之后调用，调用后，Vue 实例中的所有东西都会解除绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。

当我们使用 `keep-alive` 的时候，还有两个钩子函数，分别是 `activated` 和 `deactivated`。用 `keep-alive` 包裹的组件在切换时不会进行销毁，而是缓存到内存中并执行 `deactivated` 钩子函数，命中缓存渲染后会执行 `activated` 钩子函数。

详细资料可以参考： [《vue 生命周期深入》](#) [《Vue 实例》](#)

3.148. Vue 组件间的参数传递方式？

(1) 父子组件间通信

第一种方法是子组件通过 `props` 属性来接受父组件的数据，然后父组件在子组件上注册监听事件，子组件通过 `emit` 触发事

件来向父组件发送数据。

第二种是通过 `ref` 属性给子组件设置一个名字。父组件通过 `$refs` 组件名来获得子组件，子组件通过 `$parent` 获得父组

件，这样也可以实现通信。

第三种是使用 `provider/inject`，在父组件中通过 `provider` 提供变量，在子组件中通过 `inject` 来将变量注入到组件

中。不论子组件有多深，只要调用了 `inject` 那么就可以注入 `provider` 中的数据。

(2) 兄弟组件间通信

第一种是使用 `eventBus` 的方法，它的本质是通过创建一个空的 `Vue` 实例来作为消息传递的对象，通信的组件引入这个实

例，通信的组件通过在这个实例上监听和触发事件，来实现消息的传递。

第二种是通过 `$parent.$refs` 来获取到兄弟组件，也可以进行通信。

(3) 任意组件之间

使用 `eventBus`，其实就是创建一个事件中心，相当于中转站，可以用它来传递事件和接收事件。

如果业务逻辑复杂，很多组件之间需要同时处理一些公共的数据，这个时候采用上面这一些方法可能不利于项目的维护。这个时候

可以使用 `vuex`，`vuex` 的思想就是将这一些公共的数据抽离出来，将它作为一个全局的变量来管理，然后其他组件就可以对这个

公共数据进行读写操作，这样达到了解耦的目的。

详细资料可以参考： [《VUE 组件之间数据传递全集》](#)

3.149. `computed` 和 `watch` 的差异？

(1) `computed` 是计算一个新的属性，并将该属性挂载到 `Vue` 实例上，而 `watch` 是监听已经存在且已挂载到 `Vue` 实例上的数据，所以用 `watch` 同样可以监听 `computed` 计算属性的变化。

(2) `computed` 本质是一个惰性求值的观察者，具有缓存性，只有当依赖变化后，第一次访问 `computed` 属性，才会计算新的值。而 `watch` 则是当数据发生变化便会调用执行函数。

(3) 从使用场景上说，`computed` 适用一个数据被多个数据影响，而 `watch` 适用一个数据影响多个数据。

详细资料可以参考：[《做面试的不倒翁：浅谈 Vue 中 computed 实现原理》](#)[《深入理解 Vue 的 watch 实现原理及其实现方式》](#)



3.150. vue-router 中的导航钩子函数

(1) 全局的钩子函数 `beforeEach` 和 `afterEach`

`beforeEach` 有三个参数，`to` 代表要进入的路由对象，`from` 代表离开的路由对象。`next` 是一个必须要执行的函数，如果不传参数，那就执行下一个钩子函数，如果传入 `false`，则终止跳转，如果传入一个路径，则导航到对应的路由，如果传入 `error`，则导航终止，`error` 传入错误的监听函数。

(2) 单个路由独享的钩子函数 `beforeEnter`，它是在路由配置上直接进行定义的。

(3) 组件内的导航钩子主要有这三种：`beforeRouteEnter`、`beforeRouteUpdate`、`beforeRouteLeave`。它们是直接在路由组件内部直接进行定义的。

详细资料可以参考：[《导航守卫》](#)

3.151. \$route 和 \$router 的区别？

`$route` 是“路由信息对象”，包括 `path`, `params`, `hash`, `query`, `fullPath`, `matched`, `name` 等路由信息参数。而 `$router` 是“路由实例”对象包括了路由的跳转方法，钩子函数等。

3.152. vue 常用的修饰符？

.prevent: 提交事件不再重载页面; .stop: 阻止单击事件冒泡; .self: 当事件发生在该元素本身而不是子元素的时候会触发;

3.153. vue 中 key 值的作用?

vue 中 key 值的作用可以分为两种情况来考虑。

第一种情况是 v-if 中使用 key。由于 Vue 会尽可能高效地渲染元素，通常会复用已有元素而不是从头开始渲染。因此当我们使用 v-if 来实现元素切换的时候，如果切换前后含有相同类型的元素，那么这个元素就会被复用。如果是相同的 input 元素，那么切换前后用户的输入不会被清除掉，这样是不符合需求的。因此我们可以通过使用 key 来唯一的标识一个元素，这个情况下，使用 key 的元素不会被复用。这个时候 key 的作用是用来标识一个独立的元素。

第二种情况是 v-for 中使用 key。用 v-for 更新已渲染过的元素列表时，它默认使用“就地复用”的策略。如果数据项的顺序发生了改变，Vue 不会移动 DOM 元素来匹配数据项的顺序，而是简单复用此处的每个元素。因此通过为每个列表项提供一个 key 值，来以便 Vue 跟踪元素的身份，从而高效的实现复用。这个时候 key 的作用是为了高效的更新渲染虚拟 DOM。

详细资料可以参考： [《Vue 面试中，经常会被问到的面试题 Vue 知识点整理》](#)

[《Vue2.0 v-for 中 :key 到底有什么用？》](#) [《vue 中 key 的作用》](#)

3.154. computed 和 watch 区别?

computed 是计算属性，依赖其他属性计算值，并且 computed 的值有缓存，只有当计算值变化才会返回内容。

watch 监听到值的变化就会执行回调，在回调中可以进行一些逻辑操作。

3.155. keep-alive 组件有什么作用?

如果你需要在组件切换的时候，保存一些组件的状态防止多次渲染，就可以使用 keep-alive 组件包裹需要保存的组件。

3.156. vue 中 mixin 和 mixins 区别?

`mixin` 用于全局混入，会影响到每个组件实例。

`mixins` 应该是我们最常使用的扩展组件的方式了。如果多个组件中有相同的业务逻辑，就可以将这些逻辑剥离出来，通过 `mixins` 混入代码，比如上拉下拉加载数据这种逻辑等等。另外需要注意的是 `mixins` 混入的钩子函数会先于组件内的钩子函数执行，并且在遇到同名选项的时候也会有选择性的进行合并

详细资料可以参考： [《前端面试之道》](#) [《混入》](#)

3.157. 开发中常用的几种 Content-Type ?

(1) application/x-www-form-urlencoded

浏览器的原生 `form` 表单，如果不设置 `enctype` 属性，那么最终就会以 `application/x-www-form-urlencoded` 方式提交数据。该种方式提交的数据放在 `body` 里面，数据按照 `key1=val1&key2=val2` 的方式进行编码，`key` 和 `val` 都进行了 URL 转码。

(2) multipart/form-data

该种方式也是一个常见的 `POST` 提交方式，通常表单上传文件时使用该种方式。

(3) application/json

告诉服务器消息主体是序列化后的 `JSON` 字符串。

(4) text/xml

该种方式主要用来提交 `XML` 格式的数据。

详细资料可以参考：《常用的几种 Content-Type》

3.158. 如何封装一个 javascript 的类型判断函数？

```
function getType(value) {  
  
    // 判断数据是 null 的情况  
  
    if (value === null) {  
  
        return value + "";  
  
    }  
  
  
    // 判断数据是引用类型的情况  
  
    if (typeof value === "object") {  
  
        let valueClass = Object.prototype.toString.call(value),  
  
            type = valueClass.split(" ")[1].split("");  
  
  
        type.pop();  
  
  
        return type.join("").toLowerCase();  
    } else {  
  
        // 判断数据是基本数据类型的情况和函数的情况  
  
        return typeof value;  
    }  
}
```

详细资料可以参考：《JavaScript 专题之类型判断(上)》

3.159. 如何判断一个对象是否为空对象？

```
function checkNullObj(obj) {
```

```
return Object.keys(obj).length === 0;}
```

详细资料可以参考： [《js 判断一个 object 对象是否为空》](#)

3.160. 使用闭包实现每隔一秒打印 1,2,3,4

```
// 使用闭包实现 for (var i = 0; i < 5; i++) {  
  
    (function(i) {  
  
        setTimeout(function() {  
  
            console.log(i);  
  
        }, i * 1000);  
  
    })(i);}  
  
// 使用 let 块级作用域  
  
for (let i = 0; i < 5; i++) {  
  
    setTimeout(function() {  
  
        console.log(i);  
  
    }, i * 1000);}
```

3.161. 手写一个 jsonp

```
function jsonp(url, params, callback) {  
  
    // 判断是否含有参数  
  
    let queryString = url.indexOf "?" === "-1" ? "?" : "&";  
  
    // 添加参数  
  
    for (var k in params) {  
  
        if (params.hasOwnProperty(k)) {  
  
            queryString += k + "=" + params[k] + "&";  
        }  
    }  
    // 构造请求地址  
    let requestUrl = url + queryString;  
    // 将回调函数名作为参数  
    let script = document.createElement("script");  
    script.src = requestUrl;  
    document.body.appendChild(script);  
}
```

```
}

}

// 处理回调函数名

let random = Math.random()

    .toString()

    .replace(".", ""),
callbackName = "myJsonp" + random;

// 添加回调函数

queryString += "callback=" + callbackName;

// 构建请求

let scriptNode = document.createElement("script");

scriptNode.src = url + queryString;

window[callbackName] = function() {

    // 调用回调函数

    callback(...arguments);

    // 删除这个引入的脚本

    document.getElementsByTagName("head")[0].removeChild(scriptNode);

};

// 发起请求
```

```
document.getElementsByTagName("head")[0].appendChild(scriptNode);}
```

详细资料可以参考： [《原生 jsonp 具体实现》](#) [《jsonp 的原理与实现》](#)

3.162. 手写一个观察者模式？

```
var events = (function() {  
    var topics = {};  
  
    return {  
        // 注册监听函数  
  
        subscribe: function(topic, handler) {  
  
            if (!topics.hasOwnProperty(topic)) {  
  
                topics[topic] = [];  
  
            }  
  
            topics[topic].push(handler);  
  
        },  
  
        // 发布事件，触发观察者回调事件  
  
        publish: function(topic, info) {  
  
            if (topics.hasOwnProperty(topic)) {  
  
                topics[topic].forEach(function(handler) {  
  
                    handler(info);  
  
                });  
  
            }  
  
        },  
    };  
});
```

```
// 移除主题的一个观察者的回调事件

remove: function(topic, handler) {

    if (!topics.hasOwnProperty(topic)) return;

    var handlerIndex = -1;

    topics[topic].forEach(function(item, index) {

        if (item === handler) {

            handlerIndex = index;

        }

    });

    if (handlerIndex >= 0) {

        topics[topic].splice(handlerIndex, 1);

    }

},


// 移除主题的所有观察者的回调事件

removeAll: function(topic) {

    if (topics.hasOwnProperty(topic)) {

        topics[topic] = [];

    }

}

});})();
```

详细资料可以参考： [《JS 事件模型》](#)

3.163. EventEmitter 实现

```
class EventEmitter {  
  
    constructor() {  
  
        this.events = {};  
    }  
  
  
  
    on(event, callback) {  
  
        let callbacks = this.events[event] || [];  
  
        callbacks.push(callback);  
  
        this.events[event] = callbacks;  
  
  
  
        return this;  
    }  
  
  
  
    off(event, callback) {  
  
        let callbacks = this.events[event];  
  
        this.events[event] = callbacks && callbacks.filter(fn => fn !== callback);  
  
  
  
        return this;  
    }  
  
  
  
    emit(event, ...args) {  
  
        let callbacks = this.events[event];  
  
        callbacks.forEach(fn => {  
  
            fn(...args);  
        })  
    }  
}
```

```
});

return this;

}

once(event, callback) {

let wrapFun = function(...args) {

callback(...args);

this.off(event, wrapFun);

};

this.on(event, wrapFun);

return this;

} }
```

3.164. 一道常被人轻视的前端 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);}
```

```
//请写出以下输出结果: Foo.getName(); // 2
getName(); // 1
new Foo.getName(); // 2
new Foo().getName(); // 3
new new
Foo().getName(); // 3
```

详细资料可以参考： [《前端程序员经常忽视的一个 JavaScript 面试题》](#) [《一道考察运算符优先级的 JavaScript 面试题》](#) [《一道常被人轻视的前端 JS 面试题》](#)

3.165. 如何确定页面的可用性时间，什么是 Performance API？

Performance API 用于精确度量、控制、增强浏览器的性能表现。这个 API 为测量网站性能，提供以前没有办法做到的精度。

使用 `getTime` 来计算脚本耗时的缺点，首先，`getTime` 方法（以及 `Date` 对象的其他方法）都只能精确到毫秒级别（一秒的千分之一），想要得到更小的时间差别就无能为力了。其次，这种写法只能获取代码运行过程中的时间进度，无法知道一些后台事件的时间进度，比如浏览器用了多少时间从服务器加载网页。

为了解决这两个不足之处，ECMAScript 5 引入“高精度时间戳”这个 API，部署在 `performance` 对象上。它的精度可以达到 1 毫秒

的千分之一（1 秒的百万分之一）。

`navigationStart`: 当前浏览器窗口的前一个网页关闭，发生 `unload` 事件时的 Unix 毫秒时间戳。如果没有前一个网页，则等于 `fetchStart` 属性。

`loadEventEnd`: 返回当前网页 `load` 事件的回调函数运行结束时的 Unix 毫秒时间戳。如果该事件还没有发生，返回 `0`。

根据上面这些属性，可以计算出网页加载各个阶段的耗时。比如，网页加载整个过程的耗时的计算方法如下：

```
var t = performance.timing;
var pageLoadTime = t.loadEventEnd -
t.navigationStart;
```

详细资料可以参考： [《Performance API》](#)

3.166. js 中的命名规则

- (1) 第一个字符必须是字母、下划线（_）或美元符号（\$）
- (2) 余下的字符可以是下划线、美元符号或任何字母或数字字符

一般我们推荐使用驼峰法来对变量名进行命名，因为这样可以与 ECMAScript 内置的函数和对象命名格式保持一致。

详细资料可以参考： [《ECMAScript 变量》](#)

3.167. js 语句末尾分号是否可以省略？

在 ECMAScript 规范中，语句结尾的分号并不是必需的。但是我们一般最好不要省略分号，因为加上分号一方面有

利于我们代码的可维护性，另一方面也可以避免我们在对代码进行压缩时出现错误。

3.168. Object.assign()

`Object.assign()` 方法用于将所有可枚举属性的值从一个或多个源对象复制到目标对象。它将返回目标对象。

3.169. Math.ceil 和 Math.floor

`Math.ceil() ===` 向上取整，函数返回一个大于或等于给定数字的最小整数。

`Math.floor() ===` 向下取整，函数返回一个小于或等于给定数字的最大整数。

3.170. js for 循环注意点

```
for (var i = 0, j = 0; i < 5, j < 9; i++, j++) {  
    console.log(i, j);}  
  
// 当判断语句含有多个语句时，以最后一个判断语句的值为准，因此上面的代码会执行 10 次。  
// 当判断语句为空时，循环会一直进行。
```

3.171. 一个列表，假设有 100000 个数据，这个该怎么办？

我们需要思考的问题：该处理是否必须同步完成？数据是否必须按顺序完成？

解决办法：

(1) 将数据分页，利用分页的原理，每次服务器端只返回一定数目的数据，浏览器每次只对一部分进行加载。

(2) 使用懒加载的方法，每次加载一部分数据，其余数据当需要使用时再去加载。

(3) 使用数组分块技术，基本思路是为要处理的项目创建一个队列，然后设置定时器每过一段时间取出一部分数据，然后再使用定时器取出下一个要处理的项目进行处理，接着再设置另一个定时器。

3.172. js 中倒计时的纠偏实现？

在前端实现中我们一般通过 `setTimeout` 和 `setInterval` 方法来实现一个倒计时效果。但是使用这些方法会存在时间偏差的问题，这是由于 js 的程序执行机制造成的，`setTimeout` 和 `setInterval` 的作用是隔一段时间将回调事件加入到事件队列中，因此事件并不是立即执行的，它会等到当前执行栈为空的时候再取出事件执行，因此事件等待执行的时间就是造成误差的原因。

一般解决倒计时中的误差的有这样两种办法：

(1) 第一种是通过前端定时向服务器发送请求获取最新的时间差，以此来校准倒计时时间。

(2) 第二种方法是前端根据偏差时间来自动调整间隔时间的方式来实现的。这一种方式首先是以 `setTimeout` 递归的方式来实现倒计时，然后通过一个变量来记录已经倒计时的秒数。每一次函数调用的时候，首先将变量加一，然后根据这个变量和每次的间隔时间，我们就可以计算出此时无偏差时应该显示的时间。然后将当前的真实时间与这个时间相减，这样我们就可以得到时间的偏差大小，因此我们在设置下一个定时器的间隔大小的时候，我们就从间隔时间中减去这个偏差大小，以此来实现由于程序执行所造成的时间误差的纠正。

详细资料可以参考： [《JavaScript 前端倒计时纠偏实现》](#)

3.173. 进程间通信的方式？

- 1.管道通信
- 2.消息队列通信
- 3.信号量通信
- 4.信号通信
- 5.共享内存通信
- 6.套接字通信

详细资料可以参考： [《进程间 8 种通信方式详解》](#) [《进程与线程的一个简单解释》](#)

3.174. 如何查找一篇英文文章中出现频率最高的单词？

```
function findMostWord(article) {  
    // 合法性判断  
    if (!article) return;  
  
    // 参数处理  
    article = article.trim().toLowerCase();  
  
    let wordList = article.match(/[a-z]+/g),  
        visited = [],  
        maxNum = 0,
```

```
maxWord = "";  
  
article = " " + wordList.join(" ") + " ";  
  
// 遍历判断单词出现次数  
  
wordList.forEach(function(item) {  
  
    if (visited.indexOf(item) < 0) {  
  
        // 加入 visited  
  
        visited.push(item);  
  
        let word = new RegExp(" " + item + " ", "g"),  
            num = article.match(word).length;  
  
        if (num > maxNum) {  
  
            maxNum = num;  
  
            maxWord = item;  
  
        }  
    }  
});  
  
return maxWord + " " + maxNum;}
```

4. 算法知识总结

4.1 常用算法和数据结构总结

4.1.1 排序

冒泡排序

冒泡排序的基本思想是，对相邻的元素进行两两比较，顺序相反则进行交换，这样，每一趟会将最小或最大的元素“浮”到顶端，最终达到完全有序。

代码实现：

```
function bubbleSort(arr) {  
    if (!Array.isArray(arr) || arr.length <= 1) return;  
  
    let lastIndex = arr.length - 1;  
  
    while (lastIndex > 0) { // 当最后一个交换的元素为第一个时，说明后面全部排序完毕  
  
        let flag = true, k = lastIndex;  
  
        for (let j = 0; j < k; j++) {  
  
            if (arr[j] > arr[j + 1]) {  
  
                flag = false;  
  
                lastIndex = j; // 设置最后一次交换元素的位置  
  
                [arr[j], arr[j+1]] = [arr[j+1], arr[j]];  
            }  
        }  
  
        if (flag) break;  
    }  
}
```

冒泡排序有两种优化方式。

一种是外层循环的优化，我们可以记录当前循环中是否发生了交换，如果没有发生交换，则说明该序列已经为有序序列了。因此我们不需要再执行之后的外层循环，此时可以直接结束。

一种是内层循环的优化，我们可以记录当前循环中最后一次元素交换的位置，该位置以后的序列都是已排好的序列，因此下一轮循环中无需再去比较。

优化后的冒泡排序，当排序序列为已排序序列时，为最好的时间复杂度为 $O(n)$ 。

冒泡排序的平均时间复杂度为 $O(n^2)$ ，最坏时间复杂度为 $O(n^2)$ ，空间复杂度为 $O(1)$ ，是稳定排序。

详细资料可以参考：[《图解排序算法\(一\)》](#) [《常见排序算法 - 鸡尾酒排序》](#) [《前端笔试&面试爬坑系列---算法》](#) [《前端面试之道》](#)

选择排序

选择排序的基本思想为每一趟从待排序的数据元素中选择最小（或最大）的一个元素作为首元素，直到所有元素排完为止。

在算法实现时，每一趟确定最小元素的时候会通过不断地比较交换来使得首位置为当前最小，交换是个比较耗时的操作。其实 我们很容易发现，在还未完全确定当前最小元素之前，这些交换都是无意义的。我们可以通过设置一个变量 `min`，每一次比较 仅存储较小元素的数组下标，当轮循环结束之后，那这个变量存储的就是当前最小元素的下标，此时再执行交换操作即可。

代码实现：

```
function selectSort(array) {  
    let length = array.length;  
  
    // 如果不是数组或者数组长度小于等于 1，直接返回，不需要排序  
    if (!Array.isArray(array) || length <= 1) return;  
  
    for (let i = 0; i < length - 1; i++) {  
        let minIndex = i;  
        for (let j = i + 1; j < length; j++) {  
            if (array[j] < array[minIndex]) {  
                minIndex = j;  
            }  
        }  
        if (minIndex !== i) {  
            let temp = array[i];  
            array[i] = array[minIndex];  
            array[minIndex] = temp;  
        }  
    }  
}
```

```
let minIndex = i; // 设置当前循环最小元素索引

for (let j = i + 1; j < length; j++) {

    // 如果当前元素比最小元素索引，则更新最小元素索引

    if (array[minIndex] > array[j]) {

        minIndex = j;

    }

}

// 交换最小元素到当前位置

// [array[i], array[minIndex]] = [array[minIndex], array[i]];

swap(array, i, minIndex);

}

return array;

// 交换数组中两个元素的位置 function swap(array, left, right) {

var temp = array[left];

array[left] = array[right];

array[right] = temp;
}
```

选择排序不管初始序列是否有序，时间复杂度都为 $O(n^2)$ 。

选择排序的平均时间复杂度为 $O(n^2)$ ，最坏时间复杂度为 $O(n^2)$ ，空间复杂度为 $O(1)$ ，不是稳定排序。

详细资料可以参考： [《图解排序算法\(一\)》](#)

插入排序

直接插入排序基本思想是每一步将一个待排序的记录，插入到前面已经排好序的有序序列中去，直到插完所有元素为止。

插入排序核心--扑克牌思想： 就想着自己在打扑克牌，接起来一张，放哪里无所谓，再接起来一张，比第一张小，放左边，继续接，可能是中间数，就插在中间....依次

代码实现：

```
function insertSort(array) {  
    let length = array.length;  
  
    // 如果不是数组或者数组长度小于等于 1，直接返回，不需要排序  
    if (!Array.isArray(array) || length <= 1) return;  
  
    // 循环从 1 开始，0 位置为默认的已排序的序列  
    for (let i = 1; i < length; i++) {  
        let temp = array[i]; // 保存当前需要排序的元素  
        let j = i;  
  
        // 在当前已排序序列中比较，如果比需要排序的元素大，就依次往后移动位置  
        while (j - 1 >= 0 && array[j - 1] > temp) {  
            array[j] = array[j - 1];  
            j--;  
        }  
    }  
}
```

```
// 将找到的位置插入元素  
  
array[j] = temp;  
  
}  
  
  
return array;}
```

当排序序列为已排序序列时，为最好的时间复杂度 $O(n)$ 。

插入排序的平均时间复杂度为 $O(n^2)$ ，最坏时间复杂度为 $O(n^2)$ ，空间复杂度为 $O(1)$ ，是稳定排序。

详细资料可以参考：[《图解排序算法\(一\)》](#)

希尔排序

希尔排序的基本思想是把数组按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的元素越来越多，当增量减至 1 时，整个数组恰被分成一组，算法便终止。

代码实现：

```
function hillSort(array) {  
  
    let length = array.length;  
  
    // 如果不是数组或者数组长度小于等于 1，直接返回，不需要排序  
    if (!Array.isArray(array) || length <= 1) return;  
  
    // 第一层确定增量的大小，每次增量的大小减半  
    for (let gap = parseInt(length >> 1); gap >= 1; gap = parseInt(gap >> 1)) {  
  
        // 第二层遍历，从第 i+gap 个元素开始，步长为 gap  
        for (let i = gap; i < length; i += gap) {  
            let temp = array[i];  
            let j = i - gap;  
            while (j >= 0 && array[j] > temp) {  
                array[j + gap] = array[j];  
                j -= gap;  
            }  
            array[j + gap] = temp;  
        }  
    }  
}
```

```
// 对每个分组使用插入排序，相当于将插入排序的 1 换成了 n

for (let i = gap; i < length; i++) {

    let temp = array[i];

    let j = i;

    while (j - gap >= 0 && array[j - gap] > temp) {

        array[j] = array[j - gap];

        j -= gap;
    }

    array[j] = temp;
}

return array;
}
```

希尔排序是利用了插入排序对于已排序序列排序效果最好的特点，在一开始序列为无序序列时，将序列分为多个小的分组进行 基数排序，由于排序基数小，每次基数排序的效果较好，然后在逐步增大增量，将分组的大小增大，由于每一次都是基于上一次排序后的结果，所以每一次都可以看做是一个基本排序的序列，所以能够最大化插入排序的优点。

简单来说就是，由于开始时每组只有很少整数，所以排序很快。之后每组含有的整数越来越多，但是由于这些数也越来越有序， 所以排序速度也很快。

希尔排序的时间复杂度根据选择的增量序列不同而不同，但总的来说时间复杂度是小于 $O(n^2)$ 的。

插入排序是一个稳定排序，但是在希尔排序中，由于相同的元素可能在不同的分组中，所以可能会造成相同元素位置的变化， 所以希尔排序是一个不稳定的排序。

希尔排序的平均时间复杂度为 $O(n\log n)$ ，最坏时间复杂度为 $O(n^s)$ ，空间复杂度为 $O(1)$ ，不是稳定排序。

详细资料可以参考：《图解排序算法(二)之希尔排序》《数据结构基础 希尔排序 之 算法复杂度浅析》

归并排序

归并排序是利用归并的思想实现的排序方法，该算法采用经典的分治策略。递归的将数组两两分开直到只包含一个元素，然后 将数组排序合并，最终合并为排序好的数组。

代码实现：

```
function mergeSort(array) {  
  
    let length = array.length;  
  
    // 如果不是数组或者数组长度小于等于 0，直接返回，不需要排序  
    if (!Array.isArray(array) || length === 0) return;  
  
    if (length === 1) {  
        return array;  
    }  
  
    let mid = parseInt(length >> 1), // 找到中间索引值  
        left = array.slice(0, mid), // 截取左半部分  
        right = array.slice(mid, length); // 截取右半部分  
  
    return merge(mergeSort(left), mergeSort(right)); // 递归分解后，进行排序合并}
```

```
function merge(leftArray, rightArray) {  
  
    let result = [],  
        leftLength = leftArray.length,  
        rightLength = rightArray.length,  
        il = 0,  
        ir = 0;  
  
    // 左右两个数组的元素依次比较，将较小的元素加入结果数组中，直到其中一个数组的元素  
    // 全部加入完则停止  
  
    while (il < leftLength && ir < rightLength) {  
  
        if (leftArray[il] < rightArray[ir]) {  
  
            result.push(leftArray[il++]);  
        } else {  
  
            result.push(rightArray[ir++]);  
        }  
    }  
  
    // 如果是左边数组还有剩余，则把剩余的元素全部加入到结果数组中。  
  
    while (il < leftLength) {  
  
        result.push(leftArray[il++]);  
    }  
  
    // 如果是右边数组还有剩余，则把剩余的元素全部加入到结果数组中。  
  
    while (ir < rightLength) {
```

```
        result.push(rightArray[ir++]);  
    }  
  
    return result;}
```

归并排序将整个排序序列看成一个二叉树进行分解，首先将树分解到每一个子节点，树的每一层都是一个归并排序的过程，每一层归并的时间复杂度为 $O(n)$ ，

因为整个树的高度为 $\lg n$ ，所以归并排序的时间复杂度不管在什么情况下都为 $O(n \lg n)$ 。

归并排序的空间复杂度取决于递归的深度和用于归并时的临时数组，所以递归的深度为 $\lg n$ ，临时数组的大小为 n ，所以归并排序的空间复杂度为 $O(n)$ 。

归并排序的平均时间复杂度为 $O(n \lg n)$ ，最坏时间复杂度为 $O(n \lg n)$ ，空间复杂度为 $O(n)$ ，是稳定排序。

详细资料可以参考：[《图解排序算法\(四\)之归并排序》](#) [《归并排序的空间复杂度？》](#)

快速排序

快速排序的基本思想是通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

代码实现：

```
function quickSort(array, start, end) {  
  
    let length = array.length;  
  
    // 如果不是数组或者数组长度小于等于 1，直接返回，不需要排序
```

```
if (!Array.isArray(array) || length <= 1 || start >= end) return;

let index = partition(array, start, end); // 将数组划分为两部分，并返回右部分
的第一个元素的索引值

quickSort(array, start, index - 1); // 递归排序左半部分
quickSort(array, index + 1, end); // 递归排序右半部分}

function partition(array, start, end) {
    let pivot = array[start]; // 取第一个值为枢纽值，获取枢纽值的大小

    // 当 start 等于 end 指针时结束循环
    while (start < end) {

        // 当 end 指针指向的值大于枢纽值时，end 指针向前移动
        while (array[end] >= pivot && start < end) {
            end--;
        }

        // 将比枢纽值小的值交换到 start 位置
        array[start] = array[end];

        // 移动 start 值，当 start 指针指向的值小于枢纽值时，start 指针向后移动
        while (array[start] < pivot && start < end) {
```

```

    start++;
}

// 将比枢纽值大的值交换到 end 位置，进入下一次循环

array[end] = array[start];

}

// 将枢纽值交换到中间点

array[start] = pivot;

// 返回中间索引值

return start;
}

```

这一种方法是填空法，首先将第一个位置的数作为枢纽值，然后 `end` 指针向前移动，当遇到比枢纽值小的值或者 `end` 值 等于 `start` 值的时候停止，然后将这个值填入 `start` 的位置，然后 `start` 指针向后移动，当遇到比枢纽值大的值或者 `start` 值等于 `end` 值的时候停止，然后将这个值填入 `end` 的位置。反复循环这个过程，直到 `start` 的值等于 `end` 的 值为止。将一开始保留的枢纽值填入这个位置，此时枢纽值左边的值都比枢纽值小，枢纽值右边的值都比枢纽值大。然后在递 归左右两边的的序列。

当每次换分的结果为含 $\lfloor n/2 \rfloor$ 和 $\lceil n/2 \rceil - 1$ 个元素时，最好情况发生，此时递归的次数为 $\log n$ ，然后每次划分的时间复杂 度为 $O(n)$ ，所以最优的时间复杂度为 $O(n\log n)$ 。一般来说只要每次换分都是常比例的划分，时间复杂度都为 $O(n\log n)$ 。

当每次换分的结果为 $n-1$ 和 0 个元素时，最坏情况发生。划分操作的时间复杂度为 $O(n)$ ，递归的次数为 $n-1$ ，所以最坏的时间复杂度为 $O(n^2)$ 。所以当排序序列有序的时候，快速排序有可能被转换为冒泡排序。

快速排序的空间复杂度取决于递归的深度，所以最好的时候为 $O(\log n)$ ，最坏的时候为 $O(n)$ 。

快速排序的平均时间复杂度为 $O(n \log n)$ ，最坏时间复杂度为 $O(n^2)$ ，空间复杂度为 $O(\log n)$ ，不是稳定排序。

详细资料可以参考：[《图解排序算法\(五\)之快速排序——三数取中法》](#) [《关于快速排序的四种写法》](#) [《快速排序的时间和空间复杂度》](#) [《快速排序最好，最坏，平均复杂度分析》](#) [《快速排序算法的递归深度》](#)

堆排序

堆排序的基本思想是：将待排序序列构造成一个大顶堆，此时，整个序列的最大值就是堆顶的根节点。将其与末尾元素进行交换，此时末尾就为最大值。然后将剩余 $n-1$ 个元素重新构造成一个堆，这样会得到 n 个元素的次小值。如此反复执行，便能得到一个有序序列了。

```
function heapSort(array) {  
    let length = array.length;  
  
    // 如果不是数组或者数组长度小于等于 1，直接返回，不需要排序  
    if (!Array.isArray(array) || length <= 1) return;  
  
    buildMaxHeap(array); // 将传入的数组建立为大顶堆
```

```
// 每次循环，将最大的元素与末尾元素交换，然后剩下的元素重新构建为大顶堆

for (let i = length - 1; i > 0; i--) {

    swap(array, 0, i);

    adjustMaxHeap(array, 0, i); // 将剩下的元素重新构建为大顶堆

}

return array;

function adjustMaxHeap(array, index, heapSize) {

    let iMax,
        iLeft,
        iRight;

    while (true) {

        iMax = index; // 保存最大值的索引

        iLeft = 2 * index + 1; // 获取左子元素的索引

        iRight = 2 * index + 2; // 获取右子元素的索引

        // 如果左子元素存在，且左子元素大于最大值，则更新最大值索引

        if (iLeft < heapSize && array[iMax] < array[iLeft]) {

            iMax = iLeft;

        }

        // 如果右子元素存在，且右子元素大于最大值，则更新最大值索引

    }

}
```

```
if (iRight < heapSize && array[iMax] < array[iRight]) {

    iMax = iRight;

}

// 如果最大元素被更新了，则交换位置，使父节点大于它的子节点，同时将索引值跟新为被替换的值，继续检查它的子树

if (iMax !== index) {

    swap(array, index, iMax);

    index = iMax;

} else {

    // 如果未被更新，说明该子树满足大顶堆的要求，退出循环

    break;
}

// 构建大顶堆 function buildMaxHeap(array) {

let length = array.length,

iParent = parseInt(length >> 1) - 1; // 获取最后一个非叶子点的元素

for (let i = iParent; i >= 0; i--) {

    adjustMaxHeap(array, i, length); // 循环调整每一个子树，使其满足大顶堆的要求
}

// 交换数组中两个元素的位置 function swap(array, i, j) {

let temp = array[i];

array[i] = array[j];

array[j] = temp;
}
```

建立堆的时间复杂度为 $O(n)$ ，排序循环的次数为 $n-1$ ，每次调整堆的时间复杂度为 $O(\log n)$ ，因此堆排序的时间复杂度在 不管什么情况下都是 $O(n \log n)$ 。

堆排序的平均时间复杂度为 $O(n \log n)$ ，最坏时间复杂度为 $O(n \log n)$ ，空间复杂度为 $O(1)$ ，不是稳定排序。

详细资料可以参考：[《图解排序算法\(三\)之堆排序》](#) [《常见排序算法 - 堆排序\(Heap Sort\)》](#) [《堆排序中建堆过程时间复杂度 \$O\(n\)\$ 怎么来的？》](#) [《排序算法之 堆排序 及其时间复杂度和空间复杂度》](#) [《最小堆 构建、插入、删除的过程图解》](#)

基数排序

基数排序是一种非比较型整数排序算法，其原理是将整数按位数切割成不同的数字，然后按每个位数分别比较。排序过程：将 所有待比较数值（正整数）统一为同样的数位长度，数位较短的数前面补零。然后，从最低位开始，依次进行一次排序。这样 从最低位排序一直到最高位排序完成以后，数列就变成一个有序序列。

代码实现：

```
function radixSort(array) {  
  
    let length = array.length;  
  
    // 如果不是数组或者数组长度小于等于 1，直接返回，不需要排序  
    if (!Array.isArray(array) || length <= 1) return;  
  
    let bucket = [],  
        max = array[0],  
        loop;  
  
    ...  
}
```

```
// 确定排序数组中的最大值

for (let i = 1; i < length; i++) {

    if (array[i] > max) {

        max = array[i];

    }

}

// 确定最大值的位数

loop = (max + '').length;

// 初始化桶

for (let i = 0; i < 10; i++) {

    bucket[i] = [];

}

for (let i = 0; i < loop; i++) {

    for (let j = 0; j < length; j++) {

        let str = array[j] + '';



        if (str.length >= i + 1) {

            let k = parseInt(str[str.length - 1 - i]); // 获取当前位的值，作为插入的索引

            bucket[k].push(array[j]);

        } else {

            // 处理位数不够的情况，高位默认为 0
        }
    }
}
```

```
        bucket[0].push(array[j]);

    }

}

array.splice(0, length); // 清空旧的数组

// 使用桶重新初始化数组

for (let i = 0; i < 10; i++) {

    let t = bucket[i].length;

    for (let j = 0; j < t; j++) {

        array.push(bucket[i][j]);
    }

    bucket[i] = [];
}

return array;
}
```

基数排序的平均时间复杂度为 $O(nk)$, k 为最大元素的长度, 最坏时间复杂度为 $O(nk)$, 空间复杂度为 $O(n)$, 是稳定 排序。

详细资料可以参考: [《常见排序算法 - 基数排序》](#) [《排序算法之 基数排序 及其时间复杂度和空间复杂度》](#)

算法总结可以参考： 《算法的时间复杂度和空间复杂度-总结》 《十大经典排序算法（动图演示）》 《各类排序算法的对比及实现》

快速排序相对于其他排序效率更高的原因

上面一共提到了 8 种排序的方法，在实际使用中，应用最广泛的是快速排序。快速排序相对于其他排序算法的优势在于在相同 数据量的情况下，它的运算效率最高，并且它额外所需空间最小。

我们首先从时间复杂度来判断，由于前面几种方法的时间复杂度平均情况下基本趋向于 $O(n^2)$ ，因此只从时间复杂度上来看 的话，显然归并排序、堆排序和快速排序的时间复杂度最小。但是既然这几种方法的时间复杂度基本一致，并且快速排序在最 坏情况下时间的复杂度还会变为 $O(n^2)$ ，那么为什么它的效率反而更高呢？

首先在对大数据量排序的时候，由于归并排序的空间复杂度为 $O(n)$ ，因此归并排序在这种情况下会需要过多的额外内存，因 此归并排序首先就被排除掉了。

接下来就剩下了堆排序和快速排序的比较。我认为堆排序相对于快速排序的效率不高的原因有两个方面。

第一个方面是对于比较操作的有效性来说。对于快速排序来说，每一次元素的比较都会确定该元素在数组中的位置，也就是在 枢纽值的左边或者右边，快速排序的每一次比较操作都是有意义的结果。而对于堆排序来说，在每一次重新调整堆的时候，我 们在迭代时，已经知道上层的节点值一定比下层的节点值大，因此当我们每次为了打乱堆结构而将最后一个元素与堆顶元素互 换时，互换后的元素一定是比下层元素小的，因此我们知道比较结果却还要在堆结构调整时去进行再一次的比较，这样的比较 是没有意义的，以此在堆排序中会产生大量的没有意义的比较操作。

第二个方面是对于缓存局部性原理的利用上来考虑的，我认为这应该是造成堆排序的效率不如快速排序的主要原因。在计算机 中利用了多级缓存的机制，来解决 cpu 计算速度与存储器数据读取速度间差距过大的问题。缓存的原理主要是基于局部性原 理，局部性原理简单来说就是，当前被访问过的数据，很有可能在一段时间内被再次访问，这被称为时间局部性。还有就是当 前访问的数据，那么它相邻的数据，也有可能在一段时间内被访问到，这被称为空间局部性。计算机缓存利用了局部性的原理 来对数据进行缓存，来尽可能少的减少磁盘的 I/O 次数，以此来提高执行效率。对于堆排序来说，它最大的问题就是它对于 空

间局部性的违背，它在进行比较时，比较的并不是相邻的元素，而是与自己相隔很远的元素，这对于利用空间局部性来进行数据缓存的计算机来说，它的很多缓存都是无效的。并且对于大数据量的排序来说，缓存的命中率就会变得很低，因此会明显提高磁盘的 I/O 次数，并且由于堆排序的大量的无效比较，因此这样就造成了堆排序执行效率的低下。而相对来快速排序来说，它的排序每一次都是在相邻范围内的比较，并且比较的范围越来越小，它很好的利用了局部性原理，因此它的执行效率更高。简单来说就是在堆排序中获取一个元素的值所花费的时间比在快速排序获取一个元素的值所花费的时间要大。因此我们可以看出，时间复杂度类似的算法，在计算机中实际执行可能会有很大的差别，因为决定算法执行效率的还有内存读取这样的其他的因素。

相关资料可以参考：[《为什么在平均情况下快速排序比堆排序要优秀？》](#) [《为什么说快速排序是性能最好的排序算法？》](#)

系统自带排序实现

每个语言的排序内部实现都是不同的。

对于 JS 来说，数组长度大于 10 会采用快排，否则使用插入排序。选择插入排序是因为虽然时间复杂度很差，但是在数据量很小的情况下和 $O(N * \log N)$ 相差无几，然而插入排序需要的常数时间很小，所以相对别的排序来说更快。

稳定性

稳定性的意思就是对于相同值来说，相对顺序不能改变。通俗的讲有两个相同的数 A 和 B，在排序之前 A 在 B 的前面，而经过排序之后，B 跑到了 A 的前面，对于这种情况的发生，我们管他叫做排序的不稳定性。

稳定性有什么意义？个人理解对于前端来说，比如我们熟知框架中的虚拟 DOM 的比较，我们对一个

列表进行渲染，当数据改变后需要比较变化时，不稳定排序或操作将会使本身不需要变化的东西变化，导致重新渲染，带来性能的损耗。

排序面试题目总结

1.

快速排序在完全无序的情况下效果最好，时间复杂度为 $O(n \log n)$ ，在有序情况下效果最差，时间复杂度为 $O(n^2)$ 。

2.

3.

初始数据集的排列顺序对算法的性能无影响的有堆排序，直接选择排序，归并排序，基数排序。

4.

5.

合并 m 个长度为 n 的已排序数组的时间复杂度为 $O(nm \log m)$ 。

6.

7.

外部排序常用的算法是归并排序。

8.

9.

数组元素基本有序的情况下，插入排序效果最好，因为这样只需要比较大小，不需要移动，时间复杂度趋近于 $O(n)$ 。

10.

11.

如果只想得到 1000 个元素组成的序列中第 5 个最小元素之前的部分排序的序列，用堆排序方法最快。

12.

13.

插入排序和优化后的冒泡在最优情况（有序）都只用比较 $n-1$ 次。

14.

15.

对长度为 n 的线性表作快速排序，在最坏情况下，比较次数为 $n(n-1)/2$ 。

16.

17.

下标从 1 开始，在含有 n 个关键字的小根堆（堆顶元素最小）中，关键字最大的记录有可能存储在 $[n/2]+2$ 位置上。因为小根堆中最大的数一定是放在叶子节点上，堆本身是个完全二叉树，完全二叉树的叶子节点的位置大于 $[n/2]$ 。

18.

19.

拓扑排序的算法，每次都选择入度为 0 的结点从图中删去，并从图中删除该顶点和所有以它为起点的有向边。

20.

21.

任何一个基于“比较”的内部排序的算法，若对 n 个元素进行排序，则在最坏情况下所需的比较次数 k 满足 $2^k > n!$ ，时间下界为 $O(n \log n)$

22.

23.

m 个元素 k 路归并的归并趟数 $s = \log k(m)$ ，代入数据： $\log(100) \leq 3$

24.

25.

对 n 个记录的线性表进行快速排序为减少算法的递归深度，每次分区后，先处理较短的部分。

26.

27.

在用邻接表表示图时，拓扑排序算法时间复杂度为 $O(n+e)$

28.

4.1.2 树

二叉树相关性质

1.

节点的度：一个节点含有的子树的个数称为该节点的度；

2.

3.

叶节点或终端节点：度为零的节点；

4.

5.

节点的层次：从根开始定义起，根为第 1 层，根的子节点为第 2 层，以此类推。

6.

7.

树的高度或深度：树中节点的最大层次。

8.

9.

在非空二叉树中，第 i 层的结点总数不超过 2^{i-1} ， $i \geq 1$ 。

10.

11.

深度为 h 的二叉树最多有 $2^h - 1$ 个结点($h \geq 1$)，最少有 h 个结点。

12.

13.

对于任意一棵二叉树，如果其叶结点数为 N_0 ，而度数为 2 的结点总数为 N_2 ，则 $N_0 = N_2 + 1$ ；

14.

15.

给定 N 个节点，能构成 $h(N)$ 种不同的二叉树。 $h(N)$ 为卡特兰数的第 N 项。 $(2n)!/(n!(n+1)!)$ 。

16.

17.

二叉树的前序遍历，首先访问根结点，然后遍历左子树，最后遍历右子树。简记根-左-右。

18.

19.

二叉树的中序遍历，首先遍历左子树，然后访问根结点，最后遍历右子树。简记左-根-右。

20.

21.

二叉树的后序遍历，首先遍历左子树，然后遍历右子树，最后访问根结点。简记左-右-根。

22.

23.

二叉树是非线性数据结构，但是顺序存储结构和链式存储结构都能存储。

24.

25.

一个带权的无向连通图的最小生成树的权值之和是唯一的。

26.

27.

只有一个结点的二叉树的度为 0 。

28.

29.

二叉树的度是以节点的最大的度数定义的。

30.

31.

树的后序遍历序列等同于该树对应的二叉树的中序序列。

32.

33.

树的先序遍历序列等同于该树对应的二叉树的先序序列。

34.

35.

线索二叉树的线索实际上指向的是相应遍历序列特定结点的前驱结点和后继结点，所以先写出二叉树的中序遍历序列： debxac，中序遍历中在 x 左边和右边的字符，就是它在中序线索化的左、右线索，即 b、a 。

36.

37.

递归式的先序遍历一个 n 节点，深度为 d 的二叉树，需要栈空间的大

小为 $O(d)$ ，因为二叉树并不一定是平衡的，也就是深度 $d \neq \log n$ ，

有可能 $d > \log n$ 。所以栈大小应该是 $O(d)$

38.

39.

一棵具有 N 个结点的二叉树的前序序列和后序序列正好相反，则该二叉树一定满足该二叉树只有左子树或只有右子树，即该二叉树一定是一条链（二叉树的高度为 N ，高度等于结点数）。

40.

41.

引入二叉线索树的目的是加快查找结点的前驱或后继的速度。

42.

43.

二叉树线索化后，先序线索化与后序线索化最多有 1 个空指针域，而中序线索化最多有 2 个空指针域。

44.

45.

不管是几叉树，节点数等于=分叉数+1

46.

47.

任何一棵二叉树的叶子结点在先序、中序和后序遍历中的相对次序不发生改变。

48.

详细资料可以参考：[《n 个节点的二叉树有多少种形态》](#) [《数据结构二叉树知识点总结》](#) [《还原二叉树--已知先序中序或者后序中序》](#) [《树、森林与二叉树的转换》](#)

满二叉树

对于一棵二叉树，如果每一个非叶子节点都存在左右子树，并且二叉树中所有的叶子节点都在同一层中，这样的二叉树称为满二叉树。

完全二叉树

对于一棵具有 n 个节点的二叉树按照层次编号，同时，左右子树按照先左后右编号，如果编号为 i 的节点与同样深度的满二叉树中编号为 i 的节点在满二叉树中的位置完全相同，则这棵二叉树称为完全二叉树。

性质：

1.

具有 n 个结点的完全二叉树的深度为 $K = \lfloor \log_2 n \rfloor + 1$ (取下整数)

2.

3.

有 N 个结点的完全二叉树各结点如果用顺序方式存储，则结点之间有如下关系：若 i 为结点编号（从 1 开始编号）则 如果 $i > 1$ ，则其父结点的编号为 $i/2$ ；

4.

5.

完全二叉树，如果 $2 * i \leq N$ ，则其左儿子（即左子树的根结点）的编号为 $2 * i$ ；若 $2 * i > N$ ，则无左儿子；如果 $2 * i + 1 \leq N$ ，则其右儿子的结点编号为 $2 * i + 1$ ；若 $2 * i + 1 > N$ ，则无右儿子。

6.

平衡二叉查找树 (AVL)

平衡二叉查找树具有如下几个性质：

1. 可以是空树。
2. 假如不是空树，任何一个结点的左子树与右子树都是平衡二叉树，并且高度之差的绝对值不超过 1。

平衡二叉树是为了解决二叉查找树中出现链式结构（只有左子树或只有右子树）的情况，这样的情况出现后对我们的查找没有一点帮助，反而增加了维护的成本。

平衡因子使用两个字母来表示。第一个字母表示最小不平衡子树根结点的平衡因子，第二个字母表示最小不平衡子树较高子树的根结点的平衡因子。根据不同的情况使用不同的方法来调整失衡的子树。

详细资料可以参考： [《平衡二叉树，AVL树之图解篇》](#)

B-树

B-树主要用于文件系统以及部分数据库索引，如 MongoDB。使用 B-树来作为数据库的索引主要是为了减少查找是磁盘的 I/O 次数。试想，如果我们使用二叉查找树来作为索引，那么查找次数的最坏情况等于二叉查找树的高度，由于索引存储在磁盘中，我们每次都只能加载对应索引的磁盘页进入内存中比较，那么磁盘的 I/O 次数就等于索引树的高度。所以采用一种办法来减少索引树的高度是提高索引效率的关键。

B-树是一种多路平衡查找树，它的每一个节点最多包含 K 个子节点， K 被称为 B-树的阶， K 的大小取决于磁盘页的大小。每个节点中的元素从小到大排列，节点当中 $k-1$ 个元素正好是 k 个孩子包含的元素的值域分划。简单来说就是以前一个磁盘页只存储一个索引的值，但 B-树中一个磁盘页中存储了多个索引的值，因此在相同的比较范围内，B-树相对于一般的二叉查找树的高度更小。其实它的主要目的就是每次尽可能多的将索引值加载入内存中进行比较，以此来减少磁盘的 I/O 次数，其实就查找次数而言，和二叉查找树比较差不了多少，只是说这个比较过程是在内存中完成的，速度更快而已。

详细资料可以参考： [《漫画：什么是 B- 树？》](#)

B+树

B+ 树相对于 B-树有着更好的查找性能，根据 B-树我们可以知道，要想加快索引速度的方法就是尽量减少磁盘 I/O 的次数。B+ 树相对于 B-的主要变化是，每个中间节点中不再包含卫星数据，只有叶子节点包含卫星数据，每个父节点都出现在子节点 中，叶子节点依次相连，形成一个顺序链表。中间节点不包含卫星数据，只用来作为索引使用，这意味着每一个磁盘页中能够 包含更多的索引值。因此 B+ 树的高度相对于 B-来说更低，所以磁盘的 I/O 次数更少。由于叶子节点依次相连，并且包含了父节点，所以可以通过叶子节点来找到对应的值。同时 B+ 树所有查询都要查找到叶子节点，查询性能比 B-树稳定。

详细资料可以参考： [《漫画：什么是 B+ 树？》](#)

数据库索引

数据库以 B 树或者 B+ 树格式来储存的数据的，一张表是根据主键来构建的树的结构。因此如果想查找其他字段，就需要建立索引，我对于索引的理解是它就是以某个字段为关键字的 B 树文件，通过这个 B 树文件就能够提高数据查找的效率。但是 由于我们需要维护的是平衡树的结构，因此对于数据的写入、修改、删除就会变慢，因为这有可能会涉及到树的平衡调整。

相关资料可以参考： [《深入浅出数据库索引原理》](#) [《数据库的最简单实现》](#)

红黑树

红黑树是一种自平衡的二叉查找树，它主要是为了解决不平衡的二叉查找树的查找效率不高的缺点。红黑树保证了从根到叶子 节点的最长路径不会超过最短路径的两倍。

红黑树的有具体的规则：

1. 节点是红色或黑色。
2. 根节点是黑色。
3. 每个叶子节点都是黑色的空节点（NIL 节点）。

4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

5.从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

当红黑树发生删除和插入导致红黑树不满足这些规则时，需要通过处理，使其重新满足这些规则。

详细资料可以参考：《漫画：什么是红黑树？》《漫画算法等精选文章目录》

Huffman 树

给定 n 权值作为 n 个叶子节点，构造一棵二叉树，若这棵二叉树的带权路径长度达到最小，则称这样的二叉树为最优二叉树，也称为 Huffman 树。

利用 Huffman 树对每一个字符编码，该编码又称为 Huffman 编码，Huffman 编码是一种前缀编码，即一个字符的编码 不是另一个字符编码的前缀。

性质：

1.

对应一组权重构造出来的 Huffman 树一般不是唯一的

2.

3.

Huffman 树具有最小的带权路径长度

4.

5.

Huffman 树中没有度为 1 的结点

6.

7.

哈夫曼树是带权路径长度最短的树，路径上权值较大的结点离根较近

8.

9.

Huffman 树的带权路径长度 WPL 等于各叶子结点的带权路径长度之和

10.

详细资料可以参考：

[《数据结构和算法—— Huffman 树和 Huffman 编码》](#) [《详细图解哈夫曼 Huffman 编码树》](#)

二叉查找树

二叉查找树是一种特殊的二叉树，相对较小的值保存在左节点中，较大的值保存在右节点中，这一特性使得查找的效率很高，对于数值型和非数值型数据，比如字母和字符串，都是如此。

实现树节点类：

```
// 节点类，树的节点 class Node {  
  
    constructor(value) {  
  
        this.value = value;  
  
        this.left = null;  
  
        this.right = null;  
    }  
  
    show() {  
  
        console.log(this.value);  
    }  
}
```

实现二叉查找树类：

```
class BinarySearchTree {  
  
    constructor() {  
  
        this.root = null  
  
    }  
  
}
```

实现树的节点插入方法

节点插入的基本思想是将插入节点和当前节点做比较，如果比当前节点值小，并且没有左子树，那么将节点作为左叶子节点，否则继续和左子树进行比较。如果比当前节点值大，并且没有右子树，则将节点作为右叶子节点，否则继续和右子树进行比较。循环这个过程直到找到合适的插入位置。

```
insert(value) {  
  
    let newNode = new Node(value);  
  
    // 判断根节点是否为空，如果不为空则递归插入到树中  
  
    if (this.root === null) {  
  
        this.root = newNode;  
  
    } else {  
  
        this.insertNode(this.root, newNode);  
  
    }  
  
}  
  
insertNode(node, newNode) {  
  
    // 将插入节点的值与当前节点的进行比较，如果比当前节点小，则递归判断左子树，如果  
    // 比当前节点大，则递归判断右子树。  
}
```

```

    if (newNode.value < node.value) {

        if (node.left === null) {

            node.left = newNode;

        } else {

            this.insertNode(node.left, newNode);

        }

    } else {

        if (node.right === null) {

            node.right = newNode;

        } else {

            this.insertNode(node.right, newNode);

        }

    }

}

```

通过递归实现树的先序、中序、后序遍历

```

// 先序遍历通过递归实现

// 先序遍历则先打印当前节点，再递归打印左子节点和右子节点。

preOrderTraverse() {

    this.preOrderTraverseNode(this.root);

}

preOrderTraverseNode(node) {

    if (node !== null) {

        node.show();

    }
}

```

```
        this.preOrderTraverseNode(node.left);

        this.preOrderTraverseNode(node.right);

    }

}

// 中序遍历通过递归实现

// 中序遍历则先递归打印左子节点，再打印当前节点，最后再递归打印右子节点。

inOrderTraverse() {

    this.inOrderTraverseNode(this.root);

}

inOrderTraverseNode(node) {

    if (node !== null) {

        this.inOrderTraverseNode(node.left);

        node.show();

        this.inOrderTraverseNode(node.right);

    }

}

// 后序遍历通过递归实现

// 后序遍历则先递归打印左子节点和右子节点，最后再打印当前节点。

postOrderTraverse() {

    this.postOrderTraverseNode(this.root);

}
```

```
postOrderTraverseNode(node) {  
    if (node !== null) {  
        this.postOrderTraverseNode(node.left);  
        this.postOrderTraverseNode(node.right);  
        node.show();  
    }  
}
```

通过循环实现树的先序、中序、后序遍历

```
// 先序遍历通过循环实现  
  
// 通过栈来实现循环先序遍历，首先将根节点入栈。然后进入循环，每次循环开始，当前节点出栈，打印当前节点，然后将  
  
// 右子节点入栈，再将左子节点入栈，然后进入下一循环，直到栈为空结束循环。  
  
preOrderTraverseByStack() {  
    let stack = [];  
  
    // 现将根节点入栈，开始遍历  
    stack.push(this.root);  
  
    while (stack.length > 0) {  
        // 从栈中获取当前节点  
        let node = stack.pop();  
  
        // 执行节点操作  
        node.show();  
    }  
}
```

```
// 判断节点是否还有左右子节点，如果存在则加入栈中，注意，由于中序遍历先序遍历是先访问根

// 再访问左和右子节点，因此左右子节点的入栈顺序应该是反过来的

if (node.right) {

    stack.push(node.right);
}

if (node.left) {

    stack.push(node.left);
}

}

// 中序遍历通过循环实现

// 中序遍历先将所有的左子节点入栈，如果左子节点为 null 时，打印栈顶元素，然后判断该元素是否有右子树，如果有

// 则将右子树作为起点重复上面的过程，一直循环直到栈为空并且节点为空时。

inOrderTraverseByStack() {

    let stack = [],

    node = this.root;

}

// 中序遍历是先左再根最后右

// 所以首先应该先把最左边节点遍历到底依次 push 进栈

// 当左边没有节点时，就打印栈顶元素，然后寻找右节点

while (stack.length > 0 || node) {

    if (node) {
```

```
    stack.push(node);

    node = node.left;

} else {

    node = stack.pop();

    node.show();

    node = node.right;

}

}

}

// 后序遍历通过循环来实现

// 使用两个栈来实现，先将根节点放入栈 1 中，然后进入循环，每次循环将栈顶元素加入
// 栈 2，再依次将左节点和右节点依次

// 加入栈 1 中，然后进入下一次循环，直到栈 1 的长度为 0。最后再循环打印栈 2 的值。

postOrderTraverseByStack() {

let stack1 = [],
stack2 = [],
node = null;

// 后序遍历是先左再右最后根

// 所以对于一个栈来说，应该先 push 根节点

// 然后 push 右节点，最后 push 左节点

stack1.push(this.root);

while (stack1.length > 0) {
```

```
node = stack1.pop();

stack2.push(node);

if (node.left) {

    stack1.push(node.left);

}

if (node.right) {

    stack1.push(node.right);

}

}

while (stack2.length > 0) {

    node = stack2.pop();

    node.show();

}

}
```

实现寻找最大最小节点值

```
// 寻找最小值，在最左边的叶子节点上

findMinNode(root) {

    let node = root;

    while (node && node.left) {
```

```
        node = node.left;

    }

    return node;
}

// 寻找最大值，在最右边的叶子节点上

findMaxNode(root) {
    let node = root;

    while (node && node.right) {
        node = node.right;
    }

    return node;
}
```

实现寻找特定大小节点值

```
// 寻找特定值

find(value) {
    return this.findNode(this.root, value);
}

findNode(node, value) {
```

```
if (node === null) {  
  
    return node;  
  
}  
  
if (value < node.value) {  
  
    return this.findNode(node.left, value);  
  
} else if (value > node.value) {  
  
    return this.findNode(node.right, value);  
  
} else {  
  
    return node;  
  
}  
  
}
```

实现移除节点值

移除节点的基本思想是，首先找到需要移除的节点的位置，然后判断该节点是否有叶节点。如果没有叶节点，则直接删除，如果有两个叶子节点，则用这个叶子节点替换当前的位置。如果有两个叶子节点，则去右子树中找到最小的节点来替换当前节点。

```
// 移除指定值节点  
  
remove(value) {  
  
    this.removeNode(this.root, value);  
  
}  
  
removeNode(node, value) {  
  
    if (node === null) {  
  
        return node;  
  
}  
  
// 寻找指定节点
```

```
if (value < node.value) {

    node.left = this.removeNode(node.left, value);

    return node;

} else if (value > node.value) {

    node.right = this.removeNode(node.right, value);

    return node;

} else { // 找到节点

    // 第一种情况—没有叶节点

    if (node.left === null && node.right === null) {

        node = null;

        return node;

    }

    // 第二种情况—一个只有一个子节点的节点，将节点替换为节点的子节点

    if (node.left === null) {

        node = node.right;

        return node;

    } else if (node.right === null) {

        node = node.left;

    }

    // 第三种情况—一个有两个子节点的节点，去右子树中找到最小的节点，用它的值来替
    // 换当前节点

    // 的值，保持树的特性，然后将替换的节点去掉

    let aux = this.findMinNode(node.right);
```

```
    node.value = aux.value;

    node.right = this.removeNode(node.right, aux);

    return node;
}

}
```

求解二叉树中两个节点的最近公共祖先节点

求解二叉树中的两个节点的最近公共祖先节点可以分为三种情况来考虑

(1) 该二叉树为搜索二叉树

解决办法，首先从根节点开始遍历。如果根节点的值比两个节点的值都大的情况下，则说明两个节点的共同祖先存在于

根节点的左子树中，因此递归遍历左子树。反之，则遍历右子树。当当前节点的值比其中一个节点的值大，比其中一个

节点的值小时，该节点则为两个节点的最近公共祖先节点。

(2) 该二叉树为普通二叉树，但是每个节点含有指向父节点的指针。

通过指向父节点的指针，我们可以通过节点得到它的所有父节点，该父节点列表可以看做是一个链表，因此求两个节点

的最近公共祖先节点就可以看做是求两个链表的最近公共节点，以此来找到两个节点的最近公共祖先节点。

(3) 该二叉树为普通二叉树，节点不含有指向父节点的指针。

这种情况下，我们可以从根节点出发，分别得到根节点到两个节点的路径。然后遍历两条路径，直到遇到第一个不相同

的节点为止，这个时候该节点前面的那个节点则为两个节点的最近公共祖先节点。

详细资料可以参考： [《二叉树中两个节点的最近公共祖先节点》](#)

4.1.3 链表

反转单向链表

需要将一个单向链表反转。思路很简单，使用三个变量分别表示当前节点和当前节点的前后节点，虽然这题很简单，但是却是一道面试常考题。

思路是从头节点往后遍历，先获取下一个节点，然后将当前节点的 `next` 设置为前一个节点，然后再继续循环。

```
var reverseList = function(head) {  
    // 判断下变量边界问题  
  
    if (!head || !head.next) return head;  
  
    // 初始设置为空，因为第一个节点反转后就是尾部，尾部节点指向 null  
  
    let pre = null;  
  
    let current = head;  
  
    let next;  
  
    // 判断当前节点是否为空  
  
    // 不为空就先获取当前节点的下一节点  
  
    // 然后把当前节点的 next 设为上一个节点  
  
    // 然后把 current 设为下一个节点，pre 设为当前节点  
  
    while(current) {  
  
        next = current.next;  
  
        current.next = pre;  
  
        pre = current;  
  
        current = next;  
    }  
}
```

```
    }  
  
    return pre;};
```

4.1.4 动态规划

爬楼梯问题

有一座高度是 10 级台阶的楼梯，从下往上走，每跨一步只能向上 1 级或者 2 级台阶。要求用程序来求出一共有多少种走法？

递归方法分析

由分析可知，假设我们只差最后一步就能走上第 10 级阶梯，这个时候一共有两种情况，因为每一步只允许走 1 级或 2 级阶梯，因此分别为从 8 级阶梯和从 9 级阶梯走上去的情况。因此从 0 到 10 级阶梯的走法数量就等于从 0 到 9 级阶梯的走法数量加上 从 0 到 8 级阶梯的走法数量。依次类推，我们可以得到一个递归关系，递归结束的标志为从 0 到 1 级阶梯的走法数量和从 0 到 2 级阶梯的走法数量。

代码实现

```
function getClimbingWays(n) {  
  
    if (n < 1) {  
        return 0;  
    }  
  
    if (n === 1) {  
        return 1;  
    }  
}
```

```
if (n === 2) {  
    return 2;  
}  
  
return get ClimbingWays(n - 1) + get ClimbingWays(n - 2);}
```

使用这种方法时整个的递归过程是一个二叉树的结构，因此该方法的时间复杂度可以近似的看为 $O(2^n)$ ，空间复杂度 为递归的深度 $O(\log n)$ 。

备忘录方法

分析递归的方法我们可以发现，其实有很多的计算过程其实是重复的，因此我们可以使用一个数组，将已经计算出的值给 保存下来，每次计算时，先判断计算结果是否已经存在，如果已经存在就直接使用。

代码实现

```
let map = new Map();  
  
function get ClimbingWays(n) {  
  
    if (n < 1) {  
        return 0;  
    }  
  
    if (n === 1) {  
        return 1;  
    }  
  
    if (n === 2) {  
        return 2;  
    }  
  
    if (map.has(n)) {  
        return map.get(n);  
    }  
  
    let result = get ClimbingWays(n - 1) + get ClimbingWays(n - 2);  
    map.set(n, result);  
    return result;  
}
```

```
if (map.has(n)) {  
    return map.get(n);  
}  
else {  
    let value = get ClimbingWays(n - 1) + get ClimbingWays(n - 2);  
    map.set(n, value);  
    return value;  
}  
}
```

通过这种方式，我们将算法的时间复杂度降低为 $O(n)$ ，但是增加空间复杂度为

$O(n)$

迭代法

通过观察，我们可以发现每一个值其实都等于它的前面两个值的和，因此我们可以使用自底向上的方式来实现。

代码实现

```
function get ClimbingWays(n) {  
  
    if (n < 1) {  
        return 0;  
    }  
  
    if (n === 1) {  
        return 1;  
    }  
  
    if (n === 2) {  
        return 2;  
    }  
  
    let ways = [1, 2];  
    for (let i = 3; i <= n; i++) {  
        ways[i] = ways[i - 1] + ways[i - 2];  
    }  
    return ways[n];  
}
```

```
    return 2;

}

let a = 1,
b = 2,
temp = 0;

for (let i = 3; i <= n; i++) {
    temp = a + b;
    a = b;
    b = temp;
}

return temp;
```

通过这种方式我们可以将算法的时间复杂度降低为 $O(n)$ ，并且将算法的空间复杂度降低为 $O(1)$ 。

详细资料可以参考： [《漫画：什么是动态规划？（整合版）》](#)

4.1.3 经典笔试题

(1).js 实现一个函数，完成超过范围的两个大整数相加功能

主要思路是通过将数字转换为字符串，然后每个字符串在按位相加。

```
function bigNumberAdd(number1, number2) {
    let result = "", // 保存最后结果
```

```
carry = false; // 保留进位结果

// 将字符串转换为数组

number1 = number1.split("");
number2 = number2.split("");

// 当数组的长度都变为 0，并且最终不再进位时，结束循环

while (number1.length || number2.length || carry) {

    // 每次将最后的数字进行相加，使用~~的好处是，即使返回值为 undefined 也能转换为
    0

    carry += ~~number1.pop() + ~~number2.pop();

    // 取加法结果的个位加入最终结果

    result = carry % 10 + result;

    // 判断是否需要进位，true 和 false 的值在加法中会被转换为 1 和 0

    carry = carry > 9;

}

// 返回最终结果

return result;}
```

详细资料可以参考： [《JavaScript 实现超范围的数相加》](#) [《js 实现大整数加法》](#)

(2). js 如何实现数组扁平化？

```
// 这一种方法通过递归来实现，当元素为数组时递归调用，兼容性好 function
flattenArray(array) {

    if (!Array.isArray(array)) return;

    let result = [];

    result = array.reduce(function (pre, item) {
        // 判断元素是否为数组，如果为数组则递归调用，如果不是则加入结果数组中
        return pre.concat(Array.isArray(item) ? flattenArray(item) : item);
    }, []);

    return result;
}

// 这一种方法是利用了 toString 方法，它的一个缺点是改变了元素的类型，只适合于数组中
// 元素都是整数的情况 function flattenArray(array) {

    return array.toString().split(",").map(function (item) {
        return +item;
    })
}
```

详细资料可以参考： [《JavaScript 专题之数组扁平化》](#)



(3). js 如何实现数组去重？

```
function unique(array) {
    if (!Array.isArray(array) || array.length <= 1) return;

    var result = [];
    var seen = {};
```

```
array.forEach(function (item) {  
  if (result.indexOf(item) === -1) {  
    result.push(item);  
  }  
})  
  
return result;}  
  
  
function unique(array) {  
  if (!Array.isArray(array) || array.length <= 1) return;  
  
  
  
  return [...new Set(array)];}
```

详细资料可以参考： [《JavaScript 专题之数组去重》](#)



(4). 如何求数组的最大值和最小值？

```
var arr = [6, 4, 1, 8, 2, 11, 23]; console.log(Math.max.apply(null, arr))
```

详细资料可以参考： [《JavaScript 专题之如何求数组的最大值和最小值》](#)



(5). 如何求两个数的最大公约数？

基本思想是采用辗转相除的方法，用大的数去除以小的那个数，然后再用小的数去除以得到的余数，一直这样递归下去，直到余数为 0 时，最后的被除数就是两个数的最大公约数。

```
function getMaxCommonDivisor(a, b) {  
  if (b === 0) return a;  
  
  
  
  return getMaxCommonDivisor(b, a % b);}
```

(6). 如何求两个数的最小公倍数?

基本思想是采用将两个数相乘，然后除以它们的最大公约数

```
function getMinCommonMultiple(a, b){  
    return a * b / getMaxCommonDivisor(a, b);}
```

详细资料可以参考： [《百度 web 前端面试题之求两个数的最大公约数和最小公倍数》](#)

(7). 实现 `IndexOf` 方法?

```
function indexFun(array, val) {  
    if (!Array.isArray(array)) return;  
  
    let length = array.length;  
  
    for (let i = 0; i < length; i++) {  
        if (array[i] === val) {  
            return i;  
        }  
    }  
  
    return -1;
```

(8). 判断一个字符串是否为回文字符串?

```
function isPalindrome(str) {  
    let reg = /[^\w_]/g, // 匹配所有非单词的字符以及下划线
```

```
    newStr = str.replace(reg, "").toLowerCase(), // 替换为空字符并将大写字母转换为小写

    reverseStr = newStr.split("").reverse().join(""); // 将字符串反转

}

return reverseStr === newStr;}
```

(9). 实现一个累加函数的功能比如 `sum(1,2,3)(2).valueOf()`

```
function sum(...args) {

let result = 0;

result = args.reduce(function (pre, item) {

    return pre + item;}, 0);

let add = function (...args) {

    result = args.reduce(function (pre, item) {

        return pre + item;

    }, result);

    return add;};

add.valueOf = function () {

    console.log(result);}

return add;}
```

(10). 使用 `reduce` 方法实现 `forEach`、`map`、`filter`

```
// forEach

function forEachUseReduce(array, handler) {

    array.reduce(function (pre, item, index) {
```

```
    handler(item, index);

});

}

// map

function mapUseReduce(array, handler) {

let result = [];

array.reduce(function (pre, item, index) {

    let mapItem = handler(item, index);

    result.push(mapItem);

});

return result;

}

// filter

function filterUseReduce(array, handler) {

let result = [];

array.reduce(function (pre, item, index) {

    if (handler(item, index)) {

        result.push(item);

    }

});

}
```

```
    return result;  
}
```

(11). 设计一个简单的任务队列，要求分别在 1,3,4 秒后打印出 "1", "2", "3"

```
class Queue {  
  
    constructor() {  
  
        this.queue = [];  
  
        this.time = 0;  
  
    }  
  
  
  
    addTask(task, t) {  
  
        this.time += t;  
  
        this.queue.push([task, this.time]);  
  
        return this;  
  
    }  
  
  
  
    start() {  
  
        this.queue.forEach(item => {  
  
            setTimeout(() => {  
  
                item[0]();  
  
            }, item[1]);  
  
        })  
  
    }  
}
```

(12). 如何查找一篇英文文章中出现频率最高的单词?

```
function findMostWord(article) {  
  
    // 合法性判断  
  
    if (!article) return;  
  
    // 参数处理  
  
    article = article.trim().toLowerCase();  
  
  
  
    let wordList = article.match(/[a-z]+/g),  
        visited = [],  
        maxNum = 0,  
        maxWord = "";  
  
  
  
    article = " " + wordList.join(" ") + " ";  
  
  
  
    // 遍历判断单词出现次数  
  
    wordList.forEach(function (item) {  
  
        if (visited.indexOf(item) < 0) {  
  
            let word = new RegExp(" " + item + " ", "g"),  
                num = article.match(word).length;  
  
  
  
            if (num > maxNum) {  
  
                maxNum = num;  
  
                maxWord = item;  
            }  
        }  
    });  
}
```

```
    }  
}  
});  
  
return maxWord + " " + maxNum;  
  
}
```

4.2 常见面试智力题总结

4.2.1. 时针与分针夹角度数问题？

分析：

当时间为 m 点 n 分时，其时针与分针夹角的度数为多少？

我们可以这样考虑，分针每走一格为 6 度，分针每走一格对应的时针会走 0.5 度。

时针每走一格为 30 度。

因此，时针走过的度数为 $m * 30 + n * 0.5$ ，分针走过的度数为 $n * 6$ 。

因此时针与分针的夹角度数为 $|m * 30 + n * 0.5 - n * 6|$ ；

答案：

因此时针与分针的夹角度数为 $|m * 30 + n * 0.5 - n * 6|$ ；

详细资料参考： [《面试智力题 — 时针与分针夹角度数问题》](#)

4.2.2. 用 3 升，5 升杯子怎么量出 4 升水？

(1) 将 5 升杯子装满水，然后倒入 3 升杯子中，之后 5 升杯子还剩 2 升水。

(2) 将 3 升杯子的水倒出，然后将 5 升杯子中的 2 升水倒入 3 升杯子中。

(3) 将 5 升杯子装满水，然后向 3 升杯子中倒水，直到 3 升杯子装满为止，此时 5 升杯子中就还剩 4 升水。

4.2.3. 四个药罐中有一个浑浊的药罐，浑浊的每片药片都比其他三个干净的药罐多一克，如何只用一次天平找出浑浊的药罐？

由于浑浊的每片药片比正常药片都多出了一克，因此我认为可以通过控制药片的数量来实现判断。

(1) 首先将每个药罐进行编号，分别标记为 1、2、3、4 号药罐。

(2) 然后从 1 号药罐中取出 1 片药片，从 2 号药罐中取出 2 片药片，从 3 号药罐中取出 3 片药片，从 4 号药罐中取出 4 片药片。

(3) 将 10 片药片使用天平称重，药片的重量比正常重量多出几克，就是哪一号药罐的问题。

4.2.4. 四张卡片，卡片正面是数字，反面是字母。现在桌上四张卡片，状态为 **a 1 b 2** 现在我想要证明 **a** 的反面必然是 **1**，我只能翻两张牌，我翻哪两张？

我认为证明 **a** 的反面一定是 **1** 的充要条件为 **a** 的反面为 **1**，并且 **2** 的反面不能为 **a**，因此应该翻 **a** 和 **2** 两张牌。

4.2.5. 赛马问题，25 匹马，5 个赛道，最少几次能选出最快的三匹马？

我认为一共至少需要 7 次才能选出最快的三匹马。

(1) 首先，我们将 25 匹马分为 5 组，每组进行比赛，选出每组最快的三匹马，其余的马由于已经不可能成为前三了，因此可以直

接淘汰掉，那么我们现在还剩下了 15 匹马。

(2) 然后我们将 5 组中的第一名来进行一轮比赛，最终的结果能够确定最快的马一定是第一名，四五名的马以及它们对应组的其余

马就可以淘汰掉了，因为它们已经没有进入前三的机会了。并且第二名那一组的第三名和第三组的第二第三名都可以淘汰掉了，

它们也没有进入前三的机会了。因此我们最终剩下了第一名那一组的二三名和第二名那一组的一二名，以及第三名一共 5 匹马，

它们都有竞争最快第二第三的机会。

(3) 最后一次对最后的 5 匹马进行比赛，选择最快的一二名作为最终结果的二三名，因此就能够通过 7 次比较，选择出最快的马。

4.2.6. 五队夫妇参加聚会，每个人不能和自己的配偶握手，只能最多和他人握手一次。A 问了其他人，发现每个人的握手次数都不同，那么 A 的配偶握手了多少次？

(1) 由于每个人不能和自己的配偶握手，并且最多只能和他人握手一次，因此一个人最多能握 8 次手。

(2) 因为 A 问了除自己配偶的其他人，每个人的握手次数都不同。因此一共有九种握手的情况，由于一个人最多只能握 8 次手，因

此握手的情况分别为 0、1、2、3、4、5、6、7、8 这九种情况。

(3) 我们首先分析握了 8 次手的人，由于他和除了自己配偶的每一个人都握了一次手，因此其他人的握手次数都不为 0，因此只有

他的配偶握手次数为 0，由此我们可以知道握手次数为 8 的人和握手次数为 0 的人是配偶。

(4) 我们再来分析握了 7 次手的人，他和除了握了 0 次手以外的人都握了一次手，由于握了 8 次手的人和其余人也都握了一次手

，因此其他人的握手次数至少为 2，因此只有他的配偶的握手次数才能为 1。由此我们可以知道握手次数为 7 的人和握手次数

为 1 的人是配偶。

(5) 依次可以类推，握手次数为 6 的人和握手次数为 2 的人为配偶，握手次数为 5 的人和握手次数为 3 的人为配偶。

(6) 最终剩下了握手次数为 4 的人，按照规律我们可以得知他的配偶的握手次数也为 4。

(7) 由于 A 和其他人的握手次数都不同，因此我们可以得知握手次数为 4 的人就是 A。因此他的配偶的握手次数为 4。

4.2.7. 你只能带行走 60 公里的油，只能在起始点加油，如何穿过 80 公里的沙漠？

(1) 先走到离起点 20 公里的地方，然后放下 20 公里的油在这，然后返回起点加油。

(2) 当第二次到达这时，车还剩 40 公里的油，加上上一次放在这的 20 公里的油，一共就有 60 公里的油，能够走完剩下的路

程。

4.2.8. 烧一根不均匀的绳要用一个小时，如何用它来判断一个小时十五分钟？

一共需要三根绳子，假设分别为 1、2、3 号绳子，每个绳子一共有 A、B 两端。

(1) 首先点燃 1 号绳子的 A、B 两端，然后点燃 2 号绳子的 A 端。

(2) 当 1 号绳子燃尽时，此时过去了半小时，然后同时点燃 2 号绳子的 B 端。

(3) 当 2 号绳子燃尽时，此时又过去了 15 分钟，然后同时点燃 3 号绳子的 A、B 两端。

(4) 当 3 号绳子燃尽时，又过去了半小时，以此一共加起来过去了一个小时十五分钟。

4.2.9. 有 7 克、2 克砝码各一个，天平一只，如何只用这些物品三次将 140 克的盐分成 50、90 克各一份？

(1) 第一次用 7 克砝码和 2 克砝码称取 9 克盐。

(2) 第二次再用第一次称取的盐和砝码称取 16 克盐。

(3) 第三次再用前两次称取的盐和砝码称取 25 克盐，这样就总共称取了 50 克盐，剩下的就是 90 克。

4.2.10. 有一辆火车以每小时 15 公里的速度离开洛杉矶直奔纽约，另一辆火车以第 小时 20 公里的速度从纽约开往洛杉矶。如果有一只鸟，以外 30 公里每小时的速度和 两辆火车现时启动，从洛杉矶出发，碰到另辆车后返回，依次在两辆火车来回的飞行，直道两面辆火车相遇，请问，这只小鸟飞行了多长距离？

由于小鸟一直都在飞，直到两车相遇时才停下来。因此小鸟飞行的时间为两车相遇的时间，由于两车是相向而行，因此

两车相遇的时间为总路程除以两车的速度之和，然后再用飞行的时间去乘以小鸟的速度，就能够得出小鸟飞行的距离。

4.2.11. 你有两个罐子，50 个红色弹球，50 个蓝色弹球，随机选出一个罐子，随机选取一个弹球放入罐子，怎么给红色弹球最大的选中机会？在你的计划中，得到红球的准确几率是多少？

第一个罐子里放一个红球，第二个罐子里放剩余的球，这样概率接近 75%，这是概率最大的方法

4.2.12. 假设你有 8 个球，其中一个略微重一些，但是找出这个球的惟一方法是将两个球放在天平上对比。最少要称多少次才能找出这个较重的球？

最少两次可以称出。

首先将 8 个球分为 3 组，其中两组为 3 个球，一组为 2 个球。

第一次将两组三个的球进行比较，如果两边相等，则说明重的球在最后一组里。第二次将最后一组的球进行比较即可。如

果两边不等，则说明重的球在较重的一边，第二次只需从这一组中随机取两球出来比较即可判断。

4.2.13. 在房里有三盏灯，房外有三个开关，在房外看不见房内的情况，你只能进门一次，你用什么方法来区分那个开关控制那一盏灯？

(1) 首先打开一盏灯 10 分钟，然后打开第二盏。

(2) 进入房间，看看那盏灯亮，摸摸那盏灯热，热的是第一个开关打开的，亮的是第二个开关打开的，而剩下的就是第三个开关打开的。

4.2.14. 他们都各自买了两对黑袜和两对白袜，八对袜子的布质、大小完全相同，而每对袜子都有一张商标纸连着。两位盲人不小心将八对袜子混在一起。他们每人怎样才能取回黑袜和白袜各两对呢？

将每一对袜子分开，一人拿一只袜子，因为袜子不分左右脚的，因此最后每个人都能取回白袜和黑袜两对。

4.2.15. 有三筐水果，一筐装的全是苹果，第二筐装的全是橘子，第三筐是橘子与苹果混在一起。筐上的标签都是骗人的，（就是说筐上的标签都是错的）你的任务是拿出其中一筐，从里面只拿一只水果，然后正确写出三筐水果的标签。

从混合标签里取出一个水果，取出的是什么水果，就写上相应的标签。

对应水果标签的筐的标签改为另一种水果。

另一种水果标签的框改为混合。

4.2.16. 一个班级 60%喜欢足球，70%喜欢篮球，80%喜欢排球，问即三种球都喜欢占比有多少？

(1) 首先确定最多的一种情况，就是 60% 喜欢足球的人同时也喜欢篮球和排球，此时为三种球都喜欢的人的最大比例。

(2) 然后确定最小的一种情况，根据题目可以知道有 40% 的人不喜欢足球，30% 的人不喜欢篮球，20% 的人不喜欢排球，因此有最多

90% 的人三种球中有一种球不喜欢，因此三种球都喜欢人的最小比例为 10%。

因此三种球都喜欢的人占比为 10%-60%

4.2.17. 五只鸡五天能下五个蛋，一百天下一百个蛋需要多少只鸡？

五只鸡五天能下五个蛋，平均下来五只鸡每天能下一个蛋，因此五只鸡一百天就能够下一百个蛋。

更多的智力题可以参考： [《经典面试智力题 200+题和解答》](#)

4.3 剑指 offer 思路总结

本部分主要是笔者在练习剑指 offer 时所做的笔记，如果出现错误，希望大家指出！

4.3.1. 二维数组中的查找

题目：

在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的

一个二维数组和一个整数，判断数组中是否含有该整数。

思路：

(1) 第一种方式是使用两层循环依次遍历，判断是否含有该整数。这一种方式最坏情况下的时间复杂度为 $O(n^2)$ 。

(2) 第二种方式是利用递增序列的特点，我们可以从二维数组的右上角开始遍历。如果当前数值比所求的数要小，则将位置向下移动

，再进行判断。如果当前数值比所求的数要大，则将位置向左移动，再进行判断。这一种方式最坏情况下的时间复杂度为 $O(n)$ 。

4.3.2. 替换空格

题目：

请实现一个函数，将一个字符串中的空格替换成“%20”。例如，当字符串为 We Are Happy. 则经过替换之后的字符串为 We%20Happy

We%20Happy

思路：

使用正则表达式，结合字符串的 `replace` 方法将空格替换为 “%20”

```
str.replace(/\s/g, "%20")
```

4.3.3. 从尾到头打印链表

题目：

输入一个链表，从尾到头打印链表每个节点的值。

思路：

利用栈来实现，首先根据头结点以此遍历链表节点，将节点加入到栈中。当遍历完成后，再将栈中元素弹出并打印，以此来实现。栈的

实现可以利用 `Array` 的 `push` 和 `pop` 方法来模拟。

4.3.4. 重建二叉树

题目：

输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入

入前序遍历序列 `{1,2,4,7,3,5,6,8}` 和中序遍历序列 `{4,7,2,1,5,3,8,6}`，则重建二叉树并返回。

思路：

利用递归的思想来求解，首先先序序列中的第一个元素一定是根元素。然后我们去中序遍历中寻找该元素的位置，找到后该元素的左

边部分就是根节点的左子树，右边部分就是根节点的右子树。因此我们可以分别截取对应的部分进行子树的递归构建。使用这种方式的

时间复杂度为 $O(n)$ ，空间复杂度为 $O(\log n)$ 。

4.3.5. 用两个栈实现队列

题目：

用两个栈来实现一个队列，完成队列的 Push 和 Pop 操作。

思路：

队列的一个基本特点是，元素先进先出。通过两个栈来模拟时，首先我们将两个栈分为栈 1 和栈 2。当执行队列的 push 操作时，直接

将元素 push 进栈 1 中。当队列执行 pop 操作时，首先判断栈 2 是否为空，如果不为空则直接 pop 元素。如果栈 2 为空，则将栈 1 中

的所有元素 pop 然后 push 到栈 2 中，然后再执行栈 2 的 pop 操作。

扩展：

当使用两个长度不同的栈来模拟队列时，队列的最大长度为较短栈的长度的两倍。

4.3.6. 旋转数组的最小数字

题目：

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个非递减排序的数组的一个旋转，输出旋转数组的

最小元素。例如数组 {3, 4, 5, 1, 2} 为 {1, 2, 3, 4, 5} 的一个旋转，该数组的最小值为 1。NOTE：给出的所有元素都大于 0，若数组大

小为 0，请返回 0。

思路：

(1) 我们输入的是一个非递减排序的数组的一个旋转，因此原始数组的值递增或者有重复。旋转之后原始数组的值一定和一个值相邻，并且不满足递增关系。因此我们就可以进行遍历，找到不满足递增关系的一对值，后一个值就是旋转数组的最小数字。

(2) 二分法

相关资料可以参考： [《旋转数组的最小数字》](#)

4.3.7. 斐波那契数列

题目：

大家都知道斐波那契数列，现在要求输入一个整数 n ，请你输出斐波那契数列的第 n 项。 $n \leq 39$

思路：

斐波那契数列的规律是，第一项为 0，第二项为 1，第三项以后的值都等于前面两项的和，因此我们可以通过循环的方式，不断通过叠加来实现第 n 项值的构建。通过循环而不是递归的方式来实现，时间复杂度降为了 $O(n)$ ，空间复杂度为 $O(1)$ 。

4.3.8. 跳台阶

题目：

一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

思路：

跳台阶的问题是一个动态规划的问题，由于一次只能跳 1 级或者 2 级，因此跳上 n 级台阶一共有两种方案，一种是从 $n-1$ 跳上，一

种是从 $n-2$ 级跳上，因此 $f(n) = f(n-1) + f(n-2)$ 。

和斐波那契数列类似，不过初始两项的值变为了 1 和 2，后面每项的值等于前面两项的和。

4.3.9. 变态跳台阶

题目：

一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级.....它也可以跳上 n 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

思路：

变态跳台阶的问题同上一个问题的思考方案是一样的，我们可以得到一个结论是，每一项的值都等于前面所有项的值的和。

$$f(1) = 1$$

$$f(2) = f(2-1) + f(2-2) \quad // f(2-2) 表示 2 阶一次跳 2 阶的次数。$$

$$f(3) = f(3-1) + f(3-2) + f(3-3)$$

...

$$f(n) = f(n-1) + f(n-2) + f(n-3) + \dots + f(n-(n-1)) + f(n-n)$$

再次总结可得

$$f(n) = \begin{cases} 1 & , (n=0) \\ 1 & , (n=1) \\ 2*f(n-1), & (n>=2) \end{cases}$$

4.3.10. 矩形覆盖

题目：

我们可以用 $2*1$ 的小矩形横着或者竖着去覆盖更大的矩形。请问用 n 个 $2*1$ 的小矩形无重叠地覆盖一个 $2*n$ 的大矩形，总共

有多少种方法？

思路：

依旧是斐波那契数列的应用

4.3.11. 二进制中 1 的个数

题目：

输入一个整数，输出该数二进制表示中 1 的个数。其中负数用补码表示。

思路：

一个不为 0 的整数的二进制表示，一定会有一位为 1。我们找到最右边的一位 1，当我们把整数减去 1 时，最右边的一位 1 变为 0，它后

面的所有位都取反，因此将减一后的值与原值相与，我们就会能够消除最右边的一位 1。因此判断一个二进制中 1 的个数，我们可以判

断这个数可以经历多少次这样的过程。

如： $1100 \& 1011 = 1000$

4.3.12. 数值的整数次方

题目：

给定一个 `double` 类型的浮点数 `base` 和 `int` 类型的整数 `exponent`。求 `base` 的 `exponent` 次方。

思路：

首先我们需要判断 `exponent` 正负和零取值三种情况，根据不同的情况通过递归来实现。

4.3.13. 调整数组顺序使奇数位于偶数前面

题目：

输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有的奇数位于数组的前半部分，所有的偶数位于位于数组的后半

部分，并保证奇数和奇数，偶数和偶数之间的相对位置不变。

思路：

由于需要考虑到调整之后的稳定性，因此我们可以使用辅助数组的方式。首先对数组中的元素进行遍历，每遇到一个奇数就将它加入到

奇数辅助数组中，每遇到一个偶数，就将它加入到偶数辅助数组中。最后再将两个数组合并。这一种方法的时间复杂度为 $O(n)$ ，空间

复杂度为 $O(n)$ 。

4.3.14. 链表中倒数第 k 个节点

题目：

输入一个链表，输出该链表中倒数第 k 个结点。

思路：

使用两个指针，先让第一个和第二个指针都指向头结点，然后再让第二个指针走 $k-1$ 步，到达第 k 个节点。然后两个指针同时向后

移动，当第二个指针到达末尾时，第一个指针指向的就是倒数第 k 个节点了。

4.3.15. 反转链表

题目：

输入一个链表，反转链表后，输出链表的所有元素。

思路：

通过设置三个变量 `pre`、`current` 和 `next`，分别用来保存前继节点、当前节点和后继节点。从第一个节点开始向后遍历，首先将当

前节点的后继节点保存到 `next` 中，然后将当前节点的后继节点设置为 `pre`，然后再将 `pre` 设置为当前节点，`current` 设置为 `ne`

`xt` 节点，实现下一次循环。

4.3.16. 合并两个排序的链表

题目：

输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则。

思路：

通过递归的方式，依次将两个链表的元素递归进行对比。

4.3.17. 树的子结构

题目：

输入两棵二叉树 A、B，判断 B 是不是 A 的子结构。（ps：我们约定空树不是任意一个树的子结构）

思路：

通过递归的思想来解决

第一步首先从树 A 的根节点开始遍历，在左右子树中找到和树 B 根结点的值一样的结点 R。

第二步两棵树同时从 R 节点和根节点以相同的遍历方式进行遍历，依次比较对应的值是否相同，当树 B 遍历结束时，结束比较。

4.3.18. 二叉树的镜像

题目：

操作给定的二叉树，将其变换为源二叉树的镜像。

思路：

从根节点开始遍历，首先通过临时变量保存左子树的引用，然后将根节点的左右子树的引用交换。然后再递归左右节点的子树交换。

4.3.19. 顺时针打印矩阵

题目：

输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字，

例如，如果输入如下矩阵： 1 2 3 4

5 6 7 8

9 10 11

13 14 15

则依次打印出数字 1,2,3,4,8,12,16,15,14,13,9,5,6,7,11,10

思路：

(1) 根据左上角和右下角可以定位出一次要旋转打印的数据。一次旋转打印结束后，往对角分别前进和后退一个单位，可以确定下一次需要打印的数据范围。

(2) 使用模拟魔方逆时针解法，每打印一行，则将矩阵逆时针旋转 90 度，打印下一行，依次重复。

4.3.20. 定义一个栈，实现 `min` 函数

题目：

定义栈的数据结构，请在该类型中实现一个能够得到栈最小元素的 `min` 函数。

思路：

使用一个辅助栈，每次将数据压入数据栈时，就把当前栈里面最小的值压入辅助栈当中。这样辅助栈的栈顶数据一直是数据栈中最小

的值。

4.3.21. 栈的压入弹出

题目：

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如

序列 1,2,3,4,5 是某栈的压入顺序，序列 4,5,3,2,1 是该压栈序列对应的一个弹出序列，但 4,3,5,1,2 就不可能是该压栈序

列的弹出序列。（注意：这两个序列的长度是相等的）

思路：

我们可以使用一个辅助栈的方式来实现，首先遍历压栈顺序，依次将元素压入辅助栈中，每次压入元素后我们首先判断该元素是否与出

栈顺序中的此刻位置的元素相等，如果不相等，则将元素继续压栈，如果相等，则将辅助栈中的栈顶元素出栈，出栈后，将出栈顺序中

的位置后移一位继续比较。当压栈顺序遍历完成后，如果辅助栈不为空，则说明该出栈顺序不正确。

4.3.22. 从上往下打印二叉树

题目：

从上往下打印出二叉树的每个节点，同层节点从左至右打印。

思路：

本质上是二叉树的层序遍历，可以通过队列来实现。首先将根节点入队。然后对队列进行出队操作，每次出队时，将出队元素的左右子

节点依次加入到队列中，直到队列长度变为 0 时，结束遍历。

4.3.23. 二叉搜索树的后序遍历

题目：

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。如果是则输出 Yes，否则输出 No。假设输入的数组的任意两

个数字都互不相同。

思路：

对于一个合法而二叉树的后序遍历来说，最末尾的元素为根元素。该元素前面的元素可以划分为两个部分，一部分为该元素的左子树，

所有元素的值比根元素小，一部分为该元素的右子树，所有的元素的值比该根元素大。并且每一部分都是一个合法的后序序列，因此我

们可以利用这些特点来递归判断。

4.3.24. 二叉树中和为某一值路径

题目：

输入一颗二叉树和一个整数，打印出二叉树中结点值的和为输入整数的所有路径。路径定义为从树的根结点开始往下一直到叶结点所经

过的结点形成一条路径。

思路：

通过对树进行深度优先遍历，遍历时保存当前节点的值并判断是否和期望值相等，如果遍历到叶节点不符合要求则回退处理。

4.3.25. 复杂链表的复制

题目：

输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针指向任意一个节点），返回结果为

复制后复杂链表的 `head`。（注意，输出结果中请不要返回参数中的节点引用，否则判题程序会直接返回空）

思路：

(1) 第一种方式，首先对原有链表每个节点进行复制，通过 `next` 连接起来。然后当链表复制完成之后，再来设置每个节点的 `random`

`random` 指针，这个时候每个节点的 `random` 的设置都需要从头结点开始遍历，因此时间的复杂度为 $O(n^2)$ 。

(2) 第二种方式，首先对原有链表每个节点进行复制，并且使用 `Map` 以键值对的方式将原有节点和复制节点保存下来。当链表复

制完成之后，再来设置每个节点的 `random` 指针，这个时候我们通过 `Map` 中的键值关系就可以获取到对应的复制节点，因此

不必再从头结点遍历，将时间的复杂度降低为了 $O(n)$ ，但是空间复杂度变为了 $O(n)$ 。这是一种以空间换时间的做法。

(3) 第三种方式，首先对原有链表的每个节点进行复制，并将复制后的节点加入到原有节点的后面。当链表复制完成之后，再进行

`random` 指针的设置，由于每个节点后面都跟着自己的复制节点，因此我们可以很容易的获取到 `random` 指向对应的复制节点

。最后再将链表分离，通过这种方法我们也能够将时间复杂度降低为 $O(n)$ 。

4.3.26. 二叉搜索树与双向链表

题目：

输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。

思路：

需要生成一个排序的双向列表，那么我们应该通过中序遍历的方式来调整树结构，因为只有中序遍历，返回才是一个从小到大的排序

序列。

基本的思路是我们首先从根节点开始遍历，先将左子树调整为一个双向链表，并将左子树双向链表的末尾元素的指针指向根节点，并

将根节点的左节点指向末尾节点。再将右子树调整为一个双向链表，并将右子树双向链表的首部元素的指针指向根元素，再将根节点

的右节点指向首部节点。通过对左右子树递归调整，因此来实现排序的双向链表的构建。

4.3.27. 字符串的排列

题目：

输入一个字符串，按字典序打印出该字符串中字符的所有排列。例如输入字符串 `abc`，则打印出由字符 `a, b, c` 所能排列出来的所有

字符串 `abc, acb, bac, bca, cab` 和 `cba`。输入描述：输入一个字符串，长度不超过 9（可能有字符重复），字符只包括大小写字母。

思路：

我们可以把一个字符串看做是两个部分，第一部分为它的第一个字符，第二部分是它后面的所有字符。求整个字符串的一个全排列，可

以看做两步，第一步是求所有可能出现在第一个位置的字符，即把第一个字符和后面的所有字符交换。第二步就是求后面所有字符的一

个全排列。因此通过这种方式，我们可以以递归的思路来求出当前字符串的全排列。

详细资料可以参考： [《字符串的排列》](#)

4.3.28. 数组中出现次数超过一半的数字

题目：

数组中有一个数字出现的次数超过数组长度的一半。请找出这个数字。例如输入一个长度为 9 的数组{1,2,3,2,2,2,5,4,2}。由于数

字 2 在数组中出现了 5 次，超过数组长度的一半，因此输出 2。如果不存在则输出 0。

思路：

(1) 对数组进行排序，排序后的中位数就是所求数字。这种方法的时间复杂度取决于我们采用的排序方法的时间复杂度，因此最快为

$O(n \log n)$ 。

(2) 由于所求数字的数量超过了数组长度的一半，因此排序后的中位数就是所求数字。因此我们可以将问题简化为求一个数组的中

位数问题。其实数组并不需要全排序，只需要部分排序。我们通过利用快排中的 `partition` 函数来实现，我们现在数组中随

机选取一个数字，而后通过 `partition` 函数返回该数字在数组中的索引 `index`，如果 `index` 刚好等于 $n/2$ ，则这个数字

便是数组的中位数，也即是要求的数，如果 `index` 大于 $n/2$ ，则中位数肯定在 `index` 的左边，在左边继续寻找即可，反之

在右边寻找。这样可以只在 `index` 的一边寻找，而不用两边都排序，减少了一半排序时间，这种方法的时间复杂度为 $O(n)$ 。

(3) 由于该数字的出现次数比所有其他数字出现次数的和还要多，因此可以考虑在遍历数组时保存两个值：一个是数组中的一个数

字，一个是次数。当遍历到下一个数字时，如果下一个数字与之前保存的数字相同，则次数加 1，如果不同，则次数减 1，如果

次数为 0，则需要保存下一个数字，并把次数设定为 1。由于我们要找的数字出现的次数比其他所有数字的出现次数之和还要大，

则要找的数字肯定是最最后一次把次数设为 1 时对应的数字。该方法的时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$ 。

详细资料可以参考：[《出现次数超过一半的数字》](#)

4.3.29. 最小的 K 个数

题目：

输入 n 个整数，找出其中最小的 k 个数。例如输入 $4, 5, 1, 6, 2, 7, 3, 8$ 这 8 个数字，则最小的 4 个数字是 $1, 2, 3, 4$ 。

思路：

(1) 第一种思路是首先将数组排序，排序后再取最小的 k 个数。这一种方法的时间复杂度取决于我们选择的排序算法的时间复杂

度，最好的情况下为 $O(n \log n)$ 。

(2) 第二种思路是由于我们只需要获得最小的 k 个数，这 k 个数不一定是按序排序的。因此我们可以使用快速排序中的 `partition`

函数来实现。每一次选择一个枢纽值，将数组分为比枢纽值大和比枢纽值小的两个部分，判断枢纽值的位置，如果该枢

纽值的位置为 $k-1$ 的话，那么枢纽值和它前面的所有数字就是最小的 k 个数。如果枢纽值的位置小于 $k-1$ 的话，假设枢

纽值的位置为 $n-1$ ，那么我们已经找到了前 n 小的数字了，我们就还需要到后半部分去寻找后半部分 $k-n$ 小的值，进行划

分。当该枢纽值的位置比 $k-1$ 大时，说明最小的 k 个值还在左半部分，我们需要继续对左半部分进行划分。这一种方法的平

均时间复杂度为 $O(n)$ 。

(3) 第三种方法是维护一个容量为 k 的最大堆。对数组进行遍历时，如果堆的容量还没有达到 k ，则直接将元素加入到堆中，这

就相当于我们假设前 k 个数就是最小的 k 个数。对 k 以后的元素遍历时，我们将该元素与堆的最大值进行比较，如果比最

大值小，那么我们则将最大值与其交换，然后调整堆。如果大于等于堆的最大值，则继续向后遍历，直到数组遍历完成。这一

种方法的平均时间复杂度为 $O(n \log k)$ 。

详细资料可以参考：[《寻找最小的 \$k\$ 个数》](#)

4.3.30. 连续子数组的最大和

题目：

HZ 偶尔会拿些专业问题来忽悠那些非计算机专业的同学。今天测试组开完会后，他又发话了：在古老的一维模式识别中，常常需要计

算连续子向量的最大和，当向量全为正数的时候，问题很好解决。但是，如果向量中包含负数，是否应该包含某个负数，并期望旁边的

正数会弥补它呢？例如：{6, -3, -2, 7, -15, 1, 2, 2}，连续子向量的最大和为 8（从第 0 个开始，到第 3 个为止）。你会不会被他忽悠

住？（子向量的长度至少是 1）

思路：

(1) 第一种思路是直接暴力求解的方式，先以第一个数字为首往后开始叠加，叠加的过程中保存最大的值。然后再以第二个数字为首

往后开始叠加，并与先前保存的最大的值进行比较。这一种方法的时间复杂度为 $O(n^2)$ 。

(2) 第二种思路是，首先我们观察一个最大和的连续数组的规律，我们可以发现，子数组一定是以正数开头的，中间包含了正负数。

因此我们可以从第一个数开始向后叠加，每次保存最大的值。叠加的值如果为负数，则将叠加值初始化为 0，因为后面的数加上负

数只会更小，因此需要寻找下一个正数开始下一个子数组的判断。一直往后判断，直到这个数组遍历完成为止，得到最大的值。

使用这一种方法的时间复杂度为 $O(n)$ 。

详细资料可以参考： [《连续子数组的最大和》](#)

4.3.31. 整数中 1 出现的次数（待深入理解）

题目：

求出 1~13 的整数中 1 出现的次数，并算出 100~1300 的整数中 1 出现的次数？为此他特别数了一下 1~13 中包含 1 的数字有 1、10、11、

12、13 因此共出现 6 次，但是对于后面问题他就没辙了。ACMer 希望你们帮帮他，并把问题更加普遍化，可以很快的求出任意非负整

数区间中 1 出现的次数。

思路：

(1) 第一种思路是直接遍历每个数，然后将判断每个数中 1 的个数，一直叠加。

(2) 第二种思路是求出 1 出现在每位上的次数，然后进行叠加。

详细资料可以参考： [《从 1 到 n 整数中 1 出现的次数：O\(logn\)算法》](#)

4.3.32. 把数组排成最小的数

题目：

输入一个正整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。例如输入数组{3, 32, 321}

}，则打印出这三个数字能排成的最小数字为 321323。

思路：

(1) 求出数组的全排列，然后对每个排列结果进行比较。

(2) 利用排序算法实现，但是比较时，比较的并不是两个元素的大小，而是两个元素正序拼接和逆序拼接的大小，如果逆序拼接的

结果更小，则交换两个元素的位置。排序结束后，数组的顺序则为最小数的排列组合顺序。

详细资料可以参考： [《把数组排成最小的数》](#)

4.3.33. 丑数（待深入理解）

题目：

把只包含质因子 2、3 和 5 的数称作丑数。例如 6、8 都是丑数，但 14 不是，因为它包含因子 7。习惯上我们把 1 当做是第一个丑数。求

按从小到大的顺序的第 N 个丑数。

思路：

(1) 判断一个数是否为丑数，可以判断该数不断除以 2，最后余数是否为 1。判断该数不断除以 3，最后余数是否为 1。判断不断除以

5，最后余数是否为 1。在不考虑时间复杂度的情况下，可以依次遍历找到第 N 个丑数。

(2) 使用一个数组来保存已排序好的丑数，后面的丑数由前面生成。

4.3.34. 第一个只出现一次的字符

题目：

在一个字符串 ($1 \leq$ 字符串长度 ≤ 10000 , 全部由大写字母组成) 中找到第一个只出现一次的字符，并返回它的位置。

思路：

(1) 第一种思路是，从前往后遍历每一个字符。每遍历一个字符，则将字符与后边的所有字符依次比较，判断是否含有相同字符。这

一种方法的时间复杂度为 $O(n^2)$ 。

(2) 第二种思路是，首先对字符串进行一次遍历，将字符和字符出现的次数以键值对的形式存储在 Map 结构中。然后第二次遍历时

，去 Map 中获取对应字符出现的次数，找到第一个只出现一次的字符。这一种方法的时间复杂度为 $O(n)$ 。

4.3.35. 数组中的逆序对

题目：

在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对

的总数 P 。

思路：

(1) 第一种思路是直接求解的方式，顺序扫描整个数组。每扫描到一个数字的时候，逐个比较该数字和它后面的数字的大小。如果

后面的数字比它小，则这两个数字就组成了一个逆序对。假设数组中含有 n 个数字。由于每个数字都要和 $O(n)$ 个数字作比

较，因此这个算法的时间复杂度是 $O(n^2)$ 。

(2) 第二种方式是使用归并排序的方式，通过利用归并排序分解后进行合并排序时，来进行逆序对的统计，这一种方法的时间复杂

度为 $O(n \log n)$ 。

详细资料可以参考： [《数组中的逆序对》](#)

4.3.36. 两个链表的第一个公共结点

题目：

输入两个链表，找出它们的第一个公共结点。

思路：

(1) 第一种方法是在第一个链表上顺序遍历每个结点，每遍历到一个结点的时候，在第二个链表上顺序遍历每个结点。如果在第二

个链表上有一个结点和第一个链表上的结点一样，说明两个链表在这个结点上重合，于是就找到了它们的公共结点。如果第一

个链表的长度为 m ，第二个链表的长度为 n 。这一种方法的时间复杂度是 $O(mn)$ 。

(2) 第二种方式是利用栈的方式，通过观察我们可以发现两个链表的公共节点，都位于链表的尾部，以此我们可以分别使用两个栈

，依次将链表元素入栈。然后在两个栈同时将元素出栈，比较出栈的节点，最后一个相同的节点就是我们要找的公共节点。这

一种方法的时间复杂度为 $O(m+n)$ ，空间复杂度为 $O(m+n)$ 。

(3) 第三种方式是，首先分别遍历两个链表，得到两个链表的长度。然后得到较长的链表与较短的链表长度的差值。我们使用两个

指针来分别对两个链表进行遍历，首先将较长链表的指针移动 n 步， n 为两个链表长度的差值，然后两个指针再同时移动，

判断所指向节点是否为同一节点。这一种方法的时间复杂度为 $O(m+n)$ ，相同对于上一种方法不需要额外的空间。

详细资料可以参考： [《两个链表的第一个公共结点》](#)

4.3.37. 数字在排序数组中出现的次数

题目：

统计一个数字：在排序数组中出现的次数。例如输入排序数组 { 1, 2, 3, 3, 3, 3, 4, 5} 和数字 3，由于 3 在这个数组中出

现了 4 次，因此输出 4。

思路：

(1) 第一种方法是直接对数组顺序遍历的方式，通过这种方法来统计数字的出现次数。这种方法的时间复杂度为 $O(n)$ 。

(2) 第二种方法是使用二分查找的方法，由于数组是排序好的数组，因此相同数字是排列在一起的。统计数字出现的次数，我们需要

去找到该段数字开始和结束的位置，以此来确定数字出现的次数。因此我们可以使用二分查找的方式来确定该数字的开始和结束

位置。如果我们第一次我们数组的中间值为 k ，如果 k 值比所求值大的话，那么我们下一次只需要判断前面一部分就行了，如

果 k 值比所求值小的话，那么我们下一次就只需要判断后面一部分就行了。如果 k 值等于所求值的时候，我们则需要判断该值

是否为开始位置或者结束位置。如果是开始位置，那么我们下一次需要到后半部分去寻找结束位置。如果是结束位置，那么我们

下一次需要到前半部分去寻找开始位置。如果既不是开始位置也不是结束位置，那么我们就分别到前后两个部分去寻找开始和结

束位置。这一种方法的平均时间复杂度为 $O(\log n)$ 。

4.3.38. 二叉树的深度

题目：

输入一棵二叉树，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深

度。

思路：

根节点的深度等于左右深度较大值加一，因此可以通过递归遍历来实现。

4.3.39. 平衡二叉树

题目：

输入一棵二叉树，判断该二叉树是否是平衡二叉树。

思路：

(1) 在遍历树的每个结点的时候，调用函数得到它的左右子树的深度。如果每个结点的左右子树的深度相差都不超过 1，那么它

就是一棵平衡的二叉树。使用这种方法时，节点会被多次遍历，因此会造成效率不高的问题。

(2) 在求一个节点的深度时，同时判断它是否平衡。如果不平衡则直接返回 -1，否则返回树高度。如果一个节点的一个子树的深

度为 -1，那么就直接向上返回 -1，该树已经是不平衡的了。通过这种方式确保了节点只能够被访问一遍。

4.3.40. 数组中只出现一次的数字

题目：

一个整型数组里除了两个数字之外，其他的数字都出现了两次。请写程序找出这两个只出现一次的数字。

思路：

(1) 第一种方式是依次遍历数组，记录下数字出现的次数，从而找出两个只出现一次的数字。

(2) 第二种方式，根据位运算的异或的性质，我们可以知道两个相同的数字异或等于 0 ，一个数和 0 异或还是它本身。由于数组中

的其他数字都是成对出现的，因此我们可以将数组中的所有数依次进行异或运算。如果只有一个数出现一次的话，那么最后剩下

的就是落单的数字。如果是两个数只出现了一次的话，那么最后剩下的就是这两个数异或的结果。这个结果中的 **1** 表示的是 **A** 和

B 不同的位。我们取异或结果的第一个 **1** 所在的位数，假如是第 **3** 位，接着通过比较第三位来将数组分为两组，相同数字一定会

被分到同一组。分组完成后再按照依次异或的思路，求得剩余数字即为两个只出现一次的数字。

4.3.41. 和为 **S** 的连续正数序列

题目：

小明很喜欢数学，有一天他在做数学作业时，要求计算出 **9~16** 的和，他马上就写出了正确答案是 **100**。但是他并不满足于此，他在想究

竟有多少种连续的正数序列的和为 **100**（至少包括两个数）。没多久，他就得到另一组连续正数和为 **100** 的序列：**18,19,20,21,22**。

现在把问题交给你，你能不能也很快的找出所有和为 **S** 的连续正数序列？Good Luck!输出描述：输出所有和为 **S** 的连续正数序列。序

列内按照从小至大的顺序，序列间按照开始数字从小到大的顺序。

思路：

维护一个正数序列数组，数组中初始只含有值 1 和 2，然后从 3 依次往后遍历，每遍历到一个元素则将这个元素加入到序列数组中，然后

判断此时序列数组的和。如果序列数组的和大于所求值，则将第一个元素（最小的元素弹出）。如果序列数组的和小于所求值，则继续

往后遍历，将元素加入到序列中继续判断。当序列数组的和等于所求值时，打印出此时的正数序列，然后继续往后遍历，寻找下一个连续序列，直到数组遍历完成终止。

详细资料可以参考： [《和为 s 的连续正数序列》](#)

4.3.42. 和为 S 的两个数字

题目：

输入一个递增排序的数组和一个数字 S ，在数组中查找两个数，使的他们的和正好是 S ，如果有对数字的和等于 S ，输出两个数

的乘积最小的。输出描述：对应每个测试案例，输出两个数，小的先输出。

思路：

首先我们通过规律可以发现，和相同的两个数字，两个数字的差值越大，乘积越小。因此我们只需要从数组的首尾开始找到第一对和

为 S 的数字对进行了。因此我们可以使用双指针的方式，左指针初始指向数组的第一个元素，右指针初始指向数组的最后一个元素

。然后首先判断两个指针指向的数字的和是否为 S ，如果为 S ，两个指针指向的数字就是我们需要寻找的数字对。如果两数的和

比 S 小，则将左指针向左移动一位后继续判断。如果两数的和比 S 大，则将右指针向右移动一位后继续判断。

详细资料可以参考： [《和为 S 的字符串》](#)

4.3.43. 左旋转字符串

题目：

汇编语言中有一种移位指令叫做循环左移（ROL），现在有个简单的任务，就是用字符串模拟这个指令的运算结果。对于一个给定的

字符序列 S，请你把其循环左移 K 位后的序列输出。例如，字符序列 S=“abcXYZdef”，要求输出循环左移 3 位后的结果，即 “XYZdefabc”。

是不是很简单？OK，搞定它！

思路：

字符串裁剪后拼接

4.3.44. 翻转单词顺序列

题目：

牛客最近来了一个新员工 Fish，每天早晨总是会拿着一本英文杂志，写些句子在本子上。同事 Cat 对 Fish 写的内容颇感兴趣，有

一天他向 Fish 借来翻看，但却读不懂它的意思。例如，“student. a am I”。后来才意识到，这家伙原来把句子单词的顺序翻转了

，正确的句子应该是“**I am a student.**”。Cat 对一一的翻转这些单词顺序可不在行，你能帮助他么？

思路：

通过空格将单词分隔，然后将数组反序后，重新拼接为字符串。

4.3.45. 扑克牌的顺子

题目：

LL 今天心情特别好，因为他去买了一副扑克牌，发现里面居然有 2 个大王，2 个小王（一副牌原本是 54 张^_^）...他随机从中抽出

了 5 张牌，想测测自己的手气，看看能不能抽到顺子，如果抽到的话，他决定去买体育彩票，嘿嘿！！“红心 A，黑桃 3，小王，大王

，方片 5”，“Oh My God!”不是顺子..... LL 不高兴了，他想了想，决定大\小王可以看成任何数字，并且 A 看作 1，J 为 11，

Q 为 12，K 为 13。上面的 5 张牌就可以变成“1,2,3,4,5”（大小王分别看作 2 和 4），“So Lucky!”。LL 决定去买体育彩票啦。

现在，要求你使用这幅牌模拟上面的过程，然后告诉我们 LL 的运气如何。为了方便起见，你可以认为大小王是 0。

思路：

首先判断 5 个数字是不是连续的，最直观的方法是把数组排序。值得注意的是，由于 0 可以当成任意数字，我们可以用 0 去补满数

组中的空缺。如果排序之后的数组不是连续的，即相邻的两个数字相隔若干个数字，但只要我们有足够的。可以补满这两个数字的空

缺，这个数组实际上还是连续的。

于是我们需要做 3 件事情：首先把数组排序，再统计数组中 0 的个数，最后统计排序之后的数组中相邻数字之间的空缺总数。如

果空缺的总数小于或者等于 0 的个数，那么这个数组就是连续的；反之则不连续。最后，我们还需要注意一点：如果数组中的非 0

数字重复出现，则该数组不是连续的。换成扑克牌的描述方式就是如果一副牌里含有对子，则不可能是顺子。

详细资料可以参考： [《扑克牌的顺子》](#)

4.3.46. 圆圈中最后剩下的数字（约瑟夫环问题）

题目：

$0, 1, \dots, n-1$ 这 n 个数字排成一个圆圈，从数字 0 开始每次从圆圈里删除第 m 个数字。求出这个圆圈里剩下的最后一个数

字。

思路：

(1) 使用环形链表进行模拟。

(2) 根据规律得出（待深入理解）

详细资料可以参考： [《圆圈中最后剩下的数字》](#)

4.3.47. $1+2+3+\dots+n$

题目：

求 $1+2+3+\dots+n$ ，要求不能使用乘除法、`for`、`while`、`if`、`else`、`switch`、`case` 等关键字及条件判断语句 (`A?B:C`) 。

思路：

由于不能使用循环语句，因此我们可以通过递归来实现。并且由于不能够使用条件判断运算符，我们可以利用 `&&` 操作符的短路特性来实现。

4.3.48. 不用加减乘除做加法

题目：

写一个函数，求两个整数之和，要求在函数体内不得使用 `+`、`-`、`*`、`/` 四则运算符号。

思路：

通过位运算，递归来实现。

4.3.49. 把字符串转换成整数。

题目：

将一个字符串转换成一个整数，要求不能使用字符串转换整数的库函数。数值为 `0` 或者字符串不是一个合法的数值则返回 `0`。输入描述：

输入一个字符串，包括数字字母符号，可以为空。输出描述：如果是合法的数值表达则返回该数字，否则返回 `0`。

思路：

首先需要进行符号判断，其次我们根据字符串的每位通过减 0 运算转换为整数和，依次根据位数叠加。

4.3.50. 数组中重复的数字

题目：

在一个长度为 n 的数组里的所有数字都在 0 到 $n-1$ 的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知

道每个数字重复了几次。请找出数组中任意一个重复的数字。

思路：

(1) 首先将数组排序，排序后再进行判断。这一种方法的时间复杂度为 $O(n \log n)$ 。

(2) 使用 `Map` 结构的方式，依次记录下每一个数字出现的次数，从而可以判断是否出现重复数字。这一种方法的时间复杂度为 $O(n^2)$ 。

(3) 从数组首部开始遍历，每遍历一个数字，则将该数字和它的下标相比较，如果数字和下标不等，则将该数字和它对应下标的值

交换。如果对应的下标值上已经是正确的值了，那么说明当前元素是一个重复数字。这一种方法相对于上一种方法来说不需要

额外的内存空间。

4.3.51. 构建乘积数组

题目：

给定一个数组 $A[0, 1, \dots, n-1]$, 请构建一个数组 $B[0, 1, \dots, n-1]$, 其中 B 中的元素 $B[i] = A[0]*A[1]*\dots*A[i-1]*A$

$[i+1]*\dots*A[n-1]$ 。不能使用除法。

思路:

$$(1) C[i] = A[0]*A[1]*\dots*A[i-1] = C[i-1]*A[i-1]$$

$$D[i] = A[i+1]*\dots*A[n-1] = D[i+1]*A[i+1]$$

$$B[i] = C[i]*D[i]$$

将乘积分为前后两个部分, 分别循环求出后, 再进行相乘。

(2) 上面的方法需要额外的内存空间, 我们可以引入中间变量的方式, 来降低空间复杂度。
(待深入理解)

详细资料可以参考: [《构建乘积数组》](#)

4.3.52. 正则表达式的匹配

题目:

请实现一个函数用来匹配包括'.'和'*'的正则表达式。模式中的字符'.'表示任意一个字符, 而'*'表示它前面的字符可以出现任

意次(包含0次)。在本题中, 匹配是指字符串的所有字符匹配整个模式。例如, 字符串"aaa"与模式"a.a"和"ab*ac*a"匹配,

但是与"aa.a"和"ab*a"均不匹配。

思路：

(1) 状态机思路（待深入理解）

详细资料可以参考： [《正则表达式匹配》](#)

4.3.53. 表示数值的字符串

题目：

请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。例如，字符串 "+100", "5e2", "-123", "3.1416" 和 "-1E-

"16" 都表示数值。但是 "12e", "1a3.14", "1.2.3", "+-5" 和 "12e+4.3" 都不是。、

思路：

利用正则表达式实现

4.3.54. 字符流中第一个不重复的字符

题目：

请实现一个函数用来找出字符流中第一个只出现一次的字符。例如，当从字符流中只读出前两个字符 "go" 时，第一个只出现一次

的字符是 "g"。当从该字符流中读出前六个字符 "google" 时，第一个只出现一次的字符是 "l"。输出描述：如果当前字符流

没有存在出现一次的字符，返回#字符。

思路：

同第 34 题

4.3.55. 链表中环的入口结点

题目：

一个链表中包含环，如何找出环的入口结点？

思路：

首先使用快慢指针的方式我们可以判断链表中是否存在环，当快慢指针相遇时，说明链表中存在环。相遇点一定存在于环中，因此我

们可以从使用一个指针从这个点开始向前移动，每移动一个点，环的长度加一，当指针再次回到这个点的时候，指针走了一圈，因此

通过这个方法我们可以得到链表中的环的长度，我们将它记为 n 。

然后我们设置两个指针，首先分别指向头结点，然后将一个指针先移动 n 步，然后两个指针再同时移动，当两个指针相遇时，相遇

点就是环的入口节点。

详细资料可以参考： [《链表中环的入口结点》](#) [《《剑指 offer》——链表中环的入口结点》](#)

4.3.56. 删除链表中重复的结点

题目：

在一个排序的链表中，存在重复的结点，请删除该链表中重复的结点，重复的结点不保留，返回链表头指针。例如，链表 `1->2->3->3->4->4->5` 处理后为 `1->2->5`

思路：

解决这个问题的第一步是确定删除的参数。当然这个函数需要输入待删除链表的头结点。头结点可能与后面的结点重复，也就是说头

结点也可能被删除，所以在链表头额外添加一个结点。

接下来我们从头遍历整个链表。如果当前结点的值与下一个结点的值相同，那么它们就是重复的结点，都可以被删除。为了保证删除

之后的链表仍然是相连的而没有中间断开，我们要把当前的前一个结点和后面值比当前结点的值要大的结点相连。我们要确保 `prev`

要始终与下一个没有重复的结点连接在一起。

4.3.57. 二叉树的下一个结点

题目：

给定一棵二叉树和其中的一个结点，如何找出中序遍历顺序的下一个结点？树中的结点除了有两个分别指向左右子结点的指针以外，

还有一个指向父节点的指针。

思路：

这个问题我们可以分为三种情况来讨论。

第一种情况，当前节点含有右子树，这种情况下，中序遍历的下一个节点为该节点右子树的最左子节点。因此我们只要从右子节点

出发，一直沿着左子节点的指针，就能找到下一个节点。

第二种情况是，当前节点不含有右子树，并且当前节点为父节点的左子节点，这种情况下中序遍历的下一个节点为当前节点的父节

点。

第三种情况是，当前节点不含有右子树，并且当前节点为父节点的右子节点，这种情况下我们沿着父节点一直向上查找，直到找到

一个节点，该节点为父节点的左子节点。这个左子节点的父节点就是中序遍历的下一个节点。

4.3.58. 对称二叉树

题目：

请实现一个函数来判断一棵二叉树是不是对称的。如果一棵二叉树和它的镜像一样，那么它是对称的。

思路：

我们对一颗二叉树进行前序遍历的时候，是先访问左子节点，然后再访问右子节点。因此我们可以定义一种对称的前序遍历的方式

，就是先访问右子节点，然后再访问左子节点。通过比较两种遍历方式最后的结果是否相同，以此来判断该二叉树是否为对称二叉

树。

4.3.59. 按之字形顺序打印二叉树（待深入理解）

题目：

请实现一个函数按照之字形顺序打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右到左的顺序打印，即第一行按照

从左到右的顺序打印，第二层按照从右到左顺序打印，第三行再按照从左到右的顺序打印，其他以此类推。

思路：

按之字形顺序打印二叉树需要两个栈。我们在打印某一行结点时，把下一层的子结点保存到相应的栈里。如果当前打印的是奇数层

， 则先保存左子结点再保存右子结点到一个栈里；如果当前打印的是偶数层，则先保存右子结点再保存左子结点到第二个栈里。每

一个栈遍历完成后进入下一层循环。

详细资料可以参考： [《按之字形顺序打印二叉树》](#)

4.3.60. 从上到下按层打印二叉树，同一层结点从左至右输出。每一层输出一行。

题目：

从上到下按层打印二叉树，同一层的结点按从左到右的顺序打印，每一层打印一行。

思路：

用一个队列来保存将要打印的结点。为了把二叉树的每一行单独打印到一行里，我们需要两个变量：一个变量表示在当前的层中还

没有打印的结点数，另一个变量表示下一次结点的数目。

4.3.61. 序列化二叉树（待深入理解）

题目：

请实现两个函数，分别用来序列化和反序列化二叉树。

思路：

数组模拟



4.3.62. 二叉搜索树的第 K 个节点

题目：

给定一颗二叉搜索树，请找出其中的第 k 小的结点。

思路：

对一颗树首先进行中序遍历，在遍历的同时记录已经遍历的节点数，当遍历到第 k 个节点时，这个节点即为第 k 大的节点。

4.3.63. 数据流中的中位数（待深入理解）

题目：

如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有值排序之后位于中间的数值。如果数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。

4.3.64. 滑动窗口中的最大值（待深入理解）

题目：

给定一个数组和滑动窗口的大小，找出所有滑动窗口里数值的最大值。例如，如果输入数组 {2,3,4,2,6,2,5,1} 及滑动窗口的

大小 3，那么一共存在 6 个滑动窗口，他们的最大值分别为 {4,4,6,6,6,5}；针对数组 {2,3,4,2,6,2,5,1} 的滑动窗口有以下

6 个： {[2,3,4],2,6,2,5,1}, {2,[3,4,2],6,2,5,1}, {2,3,[4,2,6],2,5,1}, {2,3,4,[2,6,2],5,1}, {2 ,3,4,2,[6,2,5],1}, {2,3,4,2,6,[2,5,1]}。

思路：

使用队列的方式模拟

4.3.65. 矩阵中的路径（待深入理解）

题目：

请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一个格子开始，每

一步可以在矩阵中向左，向右，向上，向下移动一个格子。如果一条路径经过了矩阵中的某一个格子，则该路径不能再进入该格子

。例如 `a b c e s f c s a d e e` 矩阵中包含一条字符串"bcced"的路径，但是矩阵中不包含"abcb"路径，因为字符串的

第一个字符 `b` 占据了矩阵中的第一行第二个格子之后，路径不能再次进入该格子。

4.3.66. 机器人的运动范围（待深入理解）

题目：

地上有一个 m 行和 n 列的方格。一个机器人从坐标 $0,0$ 的格子开始移动，每一次只能向左，右，上，下四个方向移动一格，但是不能

进入行坐标和列坐标的数位之和大于 k 的格子。例如，当 k 为 18 时，机器人能够进入方格 $(35,37)$ ，因为 $3+5+3+7 = 18$ 。但是

，它不能进入方格 $(35,38)$ ，因为 $3+5+3+8 = 19$ 。请问该机器人能够达到多少个格子？

剑指 offer 相关资料可以参考：《剑指 offer 题目练习及思路分析》《JS 版剑指 offer》《剑指 Offer 学习心得》

4.3.67 相关算法题

(1). 明星问题

题目：

有 n 个人，其中一个明星和 $n-1$ 个群众，群众都认识明星，明星不认识任何群众，群众和群众之间的认识关系不知道，现有一个

函数 `foo(A, B)`，若 A 认识 B 返回 `true`，若 A 不认识 B 返回 `false`，试设计一种算法找出明星，并给出时间复杂度。

思路：

(1) 第一种方法我们可以直接使用双层循环遍历的方式，每一个人都和其他人进行判断，如果一个人谁都不认识，那么他就是明星。

这种方法的时间复杂度为 $O(n^2)$ 。

(2) 上一种方法没有充分利用题目所给的条件，其实我们每一次比较，都可以排除一个人的可能。比如如果 A 认识 B，那么说明

A 就不会是明星，因此 A 就可以从数组中移除。如果 A 不认识 B，那么说明 B 不可能是明星，因此 B 就可以从数组中移

除。因此每一次判断都能够减少一个可能性，我们只需要从数组从前往后进行遍历，每次移除一个不可能的人，直到数组中只剩

一人为止，那么这个人就是明星。这种方法的时间复杂度为 $O(n)$ 。

详细资料可以参考： [《一个明星和 n-1 个群众》](#)

(2). 正负数组求和

题目：

有两个数组，一个数组里存放的是正整数，另一个数组里存放的是负整数，都是无序的，现在从两个数组里各拿一个，使得它们的和最接近零。

思路：

(1) 首先我们可以对两个数组分别进行排序，正数数组按从小到大排序，负数数组按从大到小排序。排序完成后我们使用两个指针分

别指向两个数组的首部，判断两个指针的和。如果和大于 0，则负数指针往后移动一个位置，如果和小于 0，则正数指针往后移动一个位置，每一次记录和的值，和当前保存下来的最小值进行比较。

5.计算机网络知识总结

5.1 应用层

应用层协议定义了应用进程间的交互和通信规则，不同主机的应用进程间如何相互传递报文，比如传递的报文的类型、格式、有哪些字段等等。

5.1.1 HTTP 协议

(1) 概况

HTTP 是超文本传输协议，它定义了客户端和服务器之间交换报文的格式和方式，默认使用 80 端口。它使用 TCP 作为传输层协议，保证了数据传输的可靠性。

HTTP 是一个无状态的协议，HTTP 服务器不会保存关于客户的任何信息。

HTTP 有两种连接模式，一种是持续连接，一种非持续连接。非持续连接指的是服务器必须为每一个请求的对象建立和维护一个全新的连接。持续连接下，TCP 连接默认不关闭，可以被多个请求复用。采用持续连接的好处是可以避免每次建立 TCP 连接三次握手时所花费的时间。在 HTTP1.0 以前使用的非持续的连接，但是可以在请求时，加上 Connection: keep-alive 来要求服务器不要关闭 TCP 连接。HTTP1.1 以后默认采用的是持续的连接。目前对于同一个域，大多数浏览器支持同时建立 6 个持久连接。

(2) HTTP 请求报文

HTTP 报文有两种，一种是请求报文，一种是响应报文。

HTTP 请求报文的格式如下：

```
GET / HTTP/1.1User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5)Accept:  
/*
```

HTTP 请求报文的第一行叫做请求行，后面的行叫做头部行，头部行后还可以跟一个实体主体。请求头部之后有一个空行，这个空行不能省略，它用来划分头部与实体。

请求行包含三个字段：方法字段、URL 字段和 HTTP 版本字段。

方法字段可以取几种不同的值，一般有 GET、POST、HEAD、PUT 和 DELETE。

一般 GET 方法只被用于向服务器获取数据。POST 方法用于将实体提交到指定的资源，通常会造成服务器资源的修改。HEAD 方法与 GET 方法类似，但是在返回的响应中，不包含请求对象。PUT 方法用于上传文件到服务器，DELETE 方法用于删除服务器上的对象。虽然请求的方法很多，但更多表达的是一种语义上的区别，并不是说 POST 能做的事情，GET 就不能做了，主要看我们如何选择。更多的方法可以参看[文档](#)。

(3)HTTP 响应报文

HTTP 报文有两种，一种是请求报文，一种是响应报文。

HTTP 响应报文的格式如下：

```
HTTP/1.0 200 OK  
  
Content-Type: text/plain  
  
Content-Length: 137582  
  
Expires: Thu, 05 Dec 1997 16:00:00 GMT  
  
Last-Modified: Wed, 5 August 1996 15:55:28 GMT  
  
Server: Apache 0.84
```

```
<html>  
  <body>Hello World</body>  
</html>
```

HTTP 响应报文的第一行叫做状态行，后面的行是首部行，最后是实体主体。

状态行包含了三个字段：协议版本字段、状态码和相应的状态信息。

实体部分是报文的主要部分，它包含了所请求的对象。

常见的状态有

200-请求成功、202-服务器端已经收到请求消息，但是尚未进行处理 301-永久移动、302-临时移动、304-所请求的资源未修改、400-客户端请求的语法错误、404-请求的资源不存在 500-服务器内部错误。

一般 1XX 代表服务器接收到请求、2XX 代表成功、3XX 代表重定向、4XX 代表客户端错误、5XX 代表服务器端错误。

更多关于状态码的可以查看：

[《HTTP 状态码》](#)

(4) 首部行

首部可以分为四种首部，请求首部、响应首部、通用首部和实体首部。通用首部和实体首部在请求报文和响应报文中都可以设置，区别在于请求首部和响应首部。

常见的请求首部有 `Accept` 可接收媒体资源的类型、`Accept-Charset` 可接收的字符集、`Host` 请求的主机名。

常见的响应首部有 `ETag` 资源的匹配信息，`Location` 客户端重定向的 URI。

常见的通用首部有 Cache-Control 控制缓存策略、Connection 管理持久连接。

常见的实体首部有 Content-Length 实体主体的大小、Expires 实体主体的过期时间、Last-Modified 资源的最后修改时间。

更多关于首部的资料可以查看：

[《HTTP 首部字段详细介绍》](#)

[《图解 HTTP》](#)

(5) HTTP/1.1 协议缺点

HTTP/1.1 默认使用了持久连接，多个请求可以复用同一个 TCP 连接，但是在同一个 TCP 连接里面，数据请求的通信次序 是固定的。服务器只有处理完一个请求的响应后，才会进行下一个请求的处理，如果前面请求的响应特别慢的话，就会造成许多请求排队等待的情况，这种情况被称为“队头堵塞”。队头阻塞会导致持久连接在达到最大数量时，剩余的资源需要等待其他 资源请求完成后才能发起请求。

为了避免这个问题，一个是减少请求数，一个是同时打开多个持久连接。这就是我们对网站优化时，使用雪碧图、合并脚本的原因。

5.1.2 HTTP/2 协议

2009 年，谷歌公开了自行研发的 SPDY 协议，主要解决 HTTP/1.1 效率不高的问题。这个协议在 Chrome 浏览器上证明可行以后，就被当作 HTTP/2 的基础，主要特性都在 HTTP/2 之中得到继承。2015 年，HTTP/2 发布。

HTTP/2 主要有以下新的特性：

(1) 二进制协议

HTTP/2 是一个二进制协议。在 HTTP/1.1 版中，报文的头信息必须是文本(ASCII 编码)，数据体可以是文本，也可以是 二进制。HTTP/2 则是一个彻底的二进制协议，头信息和数据体都是二进制，并且统称为"帧"，可以分为头信息帧和数据帧。 帧的概念是它实现多路复用的基础。

(2) 多路复用

HTTP/2 实现了多路复用，HTTP/2 仍然复用 TCP 连接，但是在在一个连接里，客户端和服务器都可以同时发送多个请求或回 应，而且不用按照顺序一一发送，这样就避免了"队头堵塞"的问题。

(3) 数据流

HTTP/2 使用了数据流的概念，因为 HTTP/2 的数据包是不按顺序发送的，同一个连接里面连续的数据包，可能属于不同的 请求。因此，必须要对数据包做标记，指出它属于哪个请求。HTTP/2 将每个请求或回应的所有数据包，称为一个数据流。每 个数据流都有一个独一无二的编号。数据包发送的时候，都必须标记数据流 ID ， 用来区分它属于哪个数据流。

(4) 头信息压缩

HTTP/2 实现了头信息压缩，由于 HTTP 1.1 协议不带有状态，每次请求都必须附上所有信息。所以，请求的很多字段都是 重复的，比如 Cookie 和 User Agent ，一模一样的内容，每次请求都必须附带，这会浪费很多带宽，也影响速度。

HTTP/2 对这一点做了优化，引入了头信息压缩机制。一方面，头信息使用 `gzip` 或 `compress` 压缩后再发送；另一方面，客户端和服务器同时维护一张头信息表，所有字段都会存入这个表，生成一个索引号，以后就不发送同样字段了，只发送索引号，这样就能提高速度了。

(5) 服务器推送

HTTP/2 允许服务器未经请求，主动向客户端发送资源，这叫做服务器推送。使用服务器推送，提前给客户端推送必要的资源，这样就可以相对减少一些延迟时间。这里需要注意的是 `http2` 下服务器主动推送的是静态资源，和 `WebSocket` 以及使用 `SSE` 等方式向客户端发送即时数据的推送是不同的。

详细的资料可以参考：[《HTTP 协议入门》](#) [《HTTP/2 服务器推送 \(Server Push\) 教程》](#)



(6) HTTP/2 协议缺点



因为 `HTTP/2` 使用了多路复用，一般来说同一域名下只需要使用一个 `TCP` 连接。由于多个数据流使用同一个 `TCP` 连接，遵守同一个流量状态控制和拥塞控制。只要一个数据流遭遇到拥塞，剩下的数据流就没法发出去，这样就导致了后面的所有数据都会被阻塞。`HTTP/2` 出现的这个问题是由于其使用 `TCP` 协议的问题，与它本身的实现其实并没有多大关系。



(7) HTTP/3 协议

由于 `TCP` 本身存在的一些限制，`Google` 就开发了一个基于 `UDP` 协议的 `QUIC` 协议，并且使用在了 `HTTP/3` 上。`QUIC` 协议在 `UDP` 协议上实现了多路复用、有序交付、重传等等功能

详细资料可以参考：[《如何看待 HTTP/3 ？》](#)

5.1.3 HTTPS 协议

(1) HTTP 存在的问题

1.

HTTP 报文使用明文方式发送，可能被第三方窃听。

2.

3.

HTTP 报文可能被第三方截取后修改通信内容，接收方没有办法发现报文内容的修改。

4.

5.

HTTP 还存在认证的问题，第三方可以冒充他人参与通信。

6.

(2) HTTPS 简介

HTTPS 指的是超文本传输安全协议，HTTPS 是基于 HTTP 协议的，不过它会使用 TLS/SSL 来对数据加密。使用 TLS/SSL 协议，所有的信息都是加密的，第三方没有办法窃听。并且它提供了一种校验机制，信息一旦被篡改，通信的双方会立刻发现。它还配备了身份证书，防止身份被冒充的情况出现。

(3) TLS 握手过程

1.

第一步，客户端向服务器发起请求，请求中包含使用的协议版本号、生成的一个随机数、以及客户端支持的加密方法。

2.

3.

第二步，服务器端接收到请求后，确认双方使用的加密方法、并给出服务器的证书、以及一个服务器生成的随机数。

4.

5.

第三步，客户端确认服务器证书有效后，生成一个新的随机数，并使用数字证书中的公钥，加密这个随机数，然后发给服务器。并且还会提供一个前面所有内容的 hash 的值，用来供服务器检验。

6.

7.

第四步，服务器使用自己的私钥，来解密客户端发送过来的随机数。并提供前面所有内容的 hash 值来供客户端检验。

8.

9.

第五步，客户端和服务器端根据约定的加密方法使用前面的三个随机数，生成对话秘钥，以后的对话过程都使用这个秘钥 来加密信息。

10.

(4) 实现原理

TLS 的握手过程主要用到了三个方法来保证传输的安全。

首先是对称加密的方法，对称加密的方法是，双方使用同一个秘钥对数据进行加密和解密。但是对称加密的存在一个问题，就是如何保证秘钥传输的安全性，因为秘钥还是会通过网络传输的，一旦秘钥被其他人获取到，那么整个加密过程就毫无作用了。这就要用到非对称加密的方法。

非对称加密的方法是，我们拥有两个秘钥，一个是公钥，一个是私钥。公钥是公开的，私钥是保密的。用私钥加密的数据，只有对应的公钥才能解密，用公钥加密的数据，只有对应的私钥才能解密。我们可以将公钥公布出去，任何想和我们通信的客户，都可以使用我们提供的公钥对数据进行加密，这样我们就可以使用私钥进行解密，这样就能保证数据的安全了。但是非对称加密有一个缺点就是加密的过程很慢，因此如果每次通信都使用非对称加密的方式的话，反而会造成等待时间过长的问题。

因此我们可以使用对称加密和非对称加密结合的方式，因为对称加密的方式的缺点是无法保证秘钥的安全传输，因此我们可以 非对称加密的方式来对对称加密的秘钥进行传输，然后以后的通信使用对称加密的方式来加密，这样就解决了两个方法各自存在的问题。

但是现在的办法也不一定是安全的，因为我们没有办法确定我们得到的公钥就一定是安全的公钥。可能存在一个中间人，截取了对方发给我们的公钥，然后将他自己的公钥发送给我们，当我们使用他的公钥加密后发送的信息，就可以被他用自己的私钥解密。然后他伪装成我们以同样的方法向对方发送信息，这样我们的信息就被窃取了，然而我们自己还不知道。

为了解决这样的问题，我们可以使用数字证书的方式，首先我们使用一种 Hash 算法来对我们的公钥和其他信息进行加密生成一个信息摘要，然后让有公信力的认证中心（简称 CA）用它的私钥对消息摘要加密，形成签名。最后将原始的信息和签名合在一起，称为数字证书。当接收方收到数字证书的时候，先根据原始信息使用同样的 Hash 算法生成一个摘要，然后使用公证处的公钥来对数字证书中的摘要进行解密，最后将解密的摘要和我们生成的摘要进行对比，就能发现我们得到的信息是否被更改了。这个方法最要的是认证中心的可靠性，一般浏览器里会内置一些顶层的认证中心的证书，相当于我们自动信任了他们，只有这样我们才能保证数据的安全。

详细资料可以参考：[《一个故事讲完 https》](#)[《SSL/TLS 协议运行机制的概述》](#)[《图解 SSL/TLS 协议》](#) [《RSA 算法原理（一）》](#) [《RSA 算法原理（二）》](#) [《分钟让你理解 HTTPS》](#)

5.1.4 DNS 协议

（1）概况

DNS 协议提供的是一种主机名到 IP 地址的转换服务，就是我们常说的域名系统。它是一个由分层的 DNS 服务器组成的分布式数据库，是定义了主机如何查询这个分布式数据库的方式的应用层协议。DNS 协议运行在 UDP 协议之上，使用 53 号端口。

(2) 域名的层级结构

域名的层级结构可以如下

主机名.次级域名.顶级域名.根域名

即

host.sld.tld.root

根据域名的层级结构，管理不同层级域名的服务器，可以分为根域名服务器、顶级域名服务器和权威域名服务器。

(3) 查询过程

DNS 的查询过程一般为，我们首先将 DNS 请求发送到本地 DNS 服务器，由本地 DNS 服务器来代为请求。

1. 从“根域名服务器”查到“顶级域名服务器”的 NS 记录和 A 记录（IP 地址）。
2. 从“顶级域名服务器”查到“次级域名服务器”的 NS 记录和 A 记录（IP 地址）。
3. 从“次级域名服务器”查出“主机名”的 IP 地址。

比如我们如果想要查询 www.baidu.com 的 IP 地址，我们首先会将请求发送到本地的 DNS 服务器中，本地 DNS 服务 器会判断是否存在该域名的缓存，如果不存在，则向根域名服务器发送一个请求，根域名服务器返回负责 .com 的顶级域名 服务器的 IP 地址的列表。然后本地 DNS 服务器再向其中一个负责 .com 的顶级域名服务器发送一个请求，负责 .com 的顶级域名服务器返回

负责 `.baidu` 的权威域名服务器的 IP 地址列表。然后本地 DNS 服务器再向其中一个权威域名服务器发送一个请求，最后权威域名服务器返回一个对应的主机名的 IP 地址列表。

(4) DNS 记录和报文

DNS 服务器中以资源记录的形式存储信息，每一个 DNS 响应报文一般包含多条资源记录。一条资源记录的具体的格式为

$(Name, Value, Type, TTL)$

其中 `TTL` 是资源记录的生存时间，它定义了资源记录能够被其他的 DNS 服务器缓存多长时间。

常用的一共有四种 `Type` 的值，分别是 `A`、`NS`、`CNAME` 和 `MX`，不同 `Type` 的值，对应资源记录代表的意义不同。

1.

如果 `Type = A`，则 `Name` 是主机名，`Value` 是主机名对应的 IP 地址。

因此一条记录为 `A` 的资源记录，提供了标准的主机名到 IP 地址的映射。

2.

3.

如果 `Type = NS`，则 `Name` 是个域名，`Value` 是负责该域名的 DNS 服务器的主机名。这个记录主要用于 DNS 链式查询时，返回下一级需要查询的 DNS 服务器的信息。

4.

5.

如果 Type = CNAME，则 Name 为别名，Value 为该主机的规范主机名。该条记录用于向查询的主机返回一个主机名 对应的规范主机名，从而告诉查询主机去查询这个主机名的 IP 地址。主机别名主要是为了通过给一些复杂的主机名提供 一个便于记忆的简单的别名。

- 6.
- 7.

如果 Type = MX，则 Name 为一个邮件服务器的别名，Value 为邮件服务器的规范主机名。它的作用和 CNAME 是一样的，都是为了解决规范主机名不利于记忆的缺点。

- 8.

(5) 递归查询和迭代查询

递归查询指的是查询请求发出后，域名服务器代为向下一级域名服务器发出请求，最后向用户返回查询的最终结果。使用递归 查询，用户只需要发出一次查询请求。

迭代查询指的是查询请求后，域名服务器返回单次查询的结果。下一级的查询由用户自己请求。使用迭代查询，用户需要发出 多次的查询请求。

一般我们向本地 DNS 服务器发送请求的方式就是递归查询，因为我们只需要发出一次请求，然后本地 DNS 服务器返回给我们最终的请求结果。而本地 DNS 服务器向其他域名服务器请求的过程是迭代查询的过程，因为每一次域名服务器只返回单次 查询的结果，下一级的查询由本地 DNS 服务器自己进行。

(6) DNS 缓存

DNS 缓存的原理非常简单，在一个请求链中，当某个 DNS 服务器接收到一个 DNS 回答后，它能够将回答中的信息缓存在本地存储器中。返回的资源记录中的 TTL 代表了该条记录的缓存的时间。

(7) DNS 实现负载平衡

DNS 可以用于在冗余的服务器上实现负载平衡。因为现在一般的大型网站使用多台服务器提供服务，因此一个域名可能会对应 多个服务器地址。当用户发起网站域名的 DNS 请求的时候，DNS 服务器返回这个域名所对应的服务器 IP 地址的集合，但在 每个回答中，会循环这些 IP 地址的顺序，用户一般会选择排在前面的地址发送请求。以此将用户的请求均衡的分配到各个不 同的服务器上，这样来实现负载均衡。

详细资料可以参考： [《DNS 原理入门》](#) [《根域名的知识》](#)

5.2 传输层

传输层协议主要是为不同主机上的不同进程间提供了逻辑通信的功能。传输层只工作在端系统中。

5.2.1 多路复用与多路分解

将传输层报文段中的数据交付到正确的套接字的工作被称为多路分解。

在源主机上从不同的套接字中收集数据，封装头信息生成报文段后，将报文段传递到网络层，这个过程被称为多路复用。

无连接的多路复用和多路分解指的是 UDP 套接字的分配过程，一个 UDP 套接字由一个二元组来标识，这个二元组包含了一 个目的地址和一个目的端口号。因此不同源地址和端口号的 UDP 报文段到达主机后，如果它们拥有相同的目的地址和目的端 口号，那么不同的报文段将会转交到同一个 UDP 套接字中。

面向连接的多路复用和多路分解指的是 TCP 套接字的分配过程，一个 TCP 套接字由一个四元组来标识，这个四元组包含了 源 IP 地址、源端口号、目的地地址和目的端口号。因此，一个 TCP 报文段从网络中到达一台主机上时，该主机使用全部 4 个 值来将报文段定向到相应的套接字。

5.2.2 UDP 协议

UDP 是一种无连接的，不可靠的传输层协议。它只提供了传输层需要实现的最低限度的功能，除了复用/分解功能和少量的差错检测外，它几乎没有对 IP 增加其他的东西。UDP 协议适用于对实时性要求高的应用场景。

特点：

1.

使用 UDP 时，在发送报文段之前，通信双方没有握手的过程，因此 UDP 被称为是无连接的传输层协议。因为没有握手过程，相对于 TCP 来说，没有建立连接的时延。因为没有连接，所以不需要在端系统中保存连接的状态。

2.

3.

UDP 提供尽力而为的交付服务，也就是说 UDP 协议不保证数据的可靠交付。

4.

5.

UDP 没有拥塞控制和流量控制的机制，所以 UDP 报文段的发送速率没有限制。

6.

7.

因为一个 UDP 套接字只使用目的地址和目的端口来标识，所以 UDP 可以支持一对一、一对多、多对一和多对多的交互通信。

8.

9.

UDP 首部小，只有 8 个字节。

10.

(1) UDP 报文段结构

UDP 报文段由首部和应用数据组成。报文段首部包含四个字段，分别是源端口号、目的端口号、长度和检验和，每个字段的长 度为两个字节。长度字段指的是整个报文段的长度，包含了首部和应用数据的大小。校验和是 UDP 提供的一种差错校验机制。 虽然提供了差错校验的机制，但是 UDP 对于差错的恢复无能为力。



5.2.3 TCP 协议



TCP 协议是面向连接的，提供可靠数据传输服务的传输层协议。

特点：

1.

TCP 协议是面向连接的，在通信双方进行通信前，需要通过三次握手建立连接。它需要在端系统中维护双方连接的状态信息。

2.

3.

TCP 协议通过序号、确认号、定时重传、检验和等机制，来提供可靠的 数据传输服务。

4.

5.

TCP 协议提供的是点对点的服务，即它是在单个发送方和单个接收方之间的连接。

6.

7.

TCP 协议提供的是全双工的服务，也就是说连接的双方能够向对方发送和接收数据。

8.

9.

TCP 提供了拥塞控制机制，在网络拥塞的时候会控制发送数据的速率，有助于减少数据包的丢失和减轻网络中的拥塞程度。

10.

11.

TCP 提供了流量控制机制，保证了通信双方的发送和接收速率相同。如果接收方可接收的缓存很小时，发送方会降低发送速率，避免因为缓存填满而造成的数据包的丢失。

(1) TCP 报文段结构

TCP 报文段由首部和数据组成，它的首部一般为 20 个字节。

源端口和目的端口号用于报文段的多路复用和分解。

32 比特的序号和 32 比特的确认号，用与实现可靠数据运输服务。

16 比特的接收窗口字段用于实现流量控制，该字段表示接收方愿意接收的字节的数量。

4 比特的首部长度字段，该字段指示了以 32 比特的字为单位的 TCP 首部的长度。

6 比特的标志字段，ACK 字段用于指示确认序号的值是有效的，RST、SYN 和 FIN 比特用于连接建立和拆除。设置 PSH 字段指示接收方应该立即将数据交给上层，URG 字段用来指示报文段里存在紧急的数据。

校验和提供了对数据的差错检测。

(2) TCP 三次握手的过程

第一次握手，客户端向服务器发送一个 SYN 连接请求报文段，报文段的首部中 SYN 标志位置为 1，序号字段是一个任选的随机数。它代表的是客户端数据的初始序号。

第二次握手，服务器端接收到客户端发送的 SYN 连接请求报文段后，服务器首先会为该连接分配 TCP 缓存和变量，然后向客户端发送 SYN ACK 报文段，报文段的首部中 SYN 和 ACK 标志位都被置为 1，代表这是一个对 SYN 连接请求的确认，同时序号字段是服务器端产生的一个任选的随机数，它代表的是服务器端数据的初始序号。确认号字段为客户端发送的序号加一。

第三次握手，客户端接收到服务器的肯定应答后，它也会为这次 TCP 连接分配缓存和变量，同时向服务器端发送一个对服务器端的报文段的确认。第三次握手可以在报文段中携带数据。

在我看来，TCP 三次握手的建立连接的过程就是相互确认初始序号的过程，告诉对方，什么样序号的报文段能够被正确接收。第三次握手的作用是客户端对服务器端的初始序号的确认。如果只使用两次握手，那么服务器就没有办法知道自己的序号是否已被确认。同时这样也是为了防止失效的请求报文段被服务器接收，而出现错误的情况。

详细资料可以参考：《TCP 为什么是三次握手，而不是两次或四次？》《TCP 的三次握手与四次挥手》

(3) TCP 四次挥手的过程

因为 TCP 连接是全双工的，也就是说通信的双方都可以向对方发送和接收消息，所以断开连接需要双方的确认。

第一次挥手，客户端认为没有数据要再发送给服务器端，它就向服务器发送一个 FIN 报文段，申请断开客户端到服务器端的连接。发送后客户端进入 FIN_WAIT_1 状态。

第二次挥手，服务器端接收到客户端释放连接的请求后，向客户端发送一个确认报文段，表示已经接收到了客户端释放连接的请求，以后不再接收客户端发送过来的数据。但是因为连接是全双工的，所以此时，服务器端还可以向客户端发送数据。服务器端进入 CLOSE_WAIT 状态。客户端收到确认后，进入 FIN_WAIT_2 状态。

第三次挥手，服务器端发送完所有数据后，向客户端发送 FIN 报文段，申请断开服务器端到客户端的连接。发送后进入 LAST_ACK 状态。

第四次挥手，客户端接收到 FIN 请求后，向服务器端发送一个确认应答，并进入 TIME_WAIT 阶段。该阶段会持续一段时间，这个时间为报文段在网络中的最大生存时间，如果该时间内服务端没有重发请求的话，客户端进入 CLOSED 的状态。如果收到服务器的重发请求就重新发送确认报文段。服务器端收到客户端的确认报文段后就进入 CLOSED 状态，这样全双工的连接就被释放了。

TCP 使用四次挥手的原因是因为 TCP 的连接是全双工的，所以需要双方分别释放到对方的连接，单独一方的连接释放，只代表不能再向对方发送数据，连接处于的是半释放的状态。

最后一次挥手后，客户端会等待一段时间再关闭的原因，是为了防止发送给服务器的确认报文段丢失或者出错，从而导致服务器端不能正常关闭。

详细资料可以参考：

[《前端面试之道》](#)

(4) ARQ 协议

ARQ 协议指的是自动重传请求，它通过超时和重传来保证数据的可靠交付，它是 TCP 协议实现可靠数据传输的一个很重要的机制。

它分为停止等待 ARQ 协议和连续 ARQ 协议。

一、停止等待 ARQ 协议

停止等待 ARQ 协议的基本原理是，对于发送方来说发送方每发送一个分组，就为这个分组设置一个定时器。当发送分组的确认回答返回了，则清除定时器，发送下一个分组。如果在规定的时间内没有收到已发送分组的肯定回答，则重新发送上一个分组。

对于接受方来说，每次接受到一个分组，就返回对这个分组的肯定应答，当收到冗余的分组时，就直接丢弃，并返回一个对冗余分组的确认。当收到分组损坏的情况的时候，直接丢弃。

使用停止等待 ARQ 协议的缺点是每次发送分组必须等到分组确认后才能发送下一个分组，这样会造成信道的利用率过低。

二、连续 ARQ 协议

连续 ARQ 协议是为了解决停止等待 ARQ 协议对于信道的利用率过低的问题。它通过连续发送一组分组，然后再等待对分组的确认回答，对于如何处理分组中可能出现的差错恢复情况，一般可以使用滑动窗口协议和选择重传协议来实现。

1. 滑动窗口协议

使用滑动窗口协议，在发送方维持了一个发送窗口，发送窗口以前的分组是已经发送并确认了的分组，发送窗口中包含了已经发送但未确认的分组和允许发送

但还未发送的分组，发送窗口以后的分组是缓存中还不允许发送的分组。当发送方向接收方发送分组时，会依次发送窗口内的所有分组，并且设置一个定时器，这个定时器可以理解为是最早发送但未收到确认的分组。如果在定时器的时间内收到某一个分组的确认回答，则滑动窗口，将窗口的首部移动到确认分组的后一个位置，此时如果还有已发送但没有确认的分组，则重新设置定时器，如果没有了则关闭定时器。如果定时器超时，则重新发送所有已经发送但还未收到确认的分组。

接收方使用的是累计确认的机制，对于所有按序到达的分组，接收方返回一个分组的肯定回答。如果收到了一个乱序的分组，那么接方会直接丢弃，并返回一个最近的按序到达的分组的肯定回答。使用累计确认保证了确认号以前的分组都已经按序到达了，所以发送窗口可以移动到已确认分组的后面。

滑动窗口协议的缺点是因为使用了累计确认的机制，如果出现了只是窗口中的第一个分组丢失，而后面的分组都按序到达的情况的话，那么滑动窗口协议会重新发送所有的分组，这样就造成了大量不必要的分组的丢弃和重传。

1. 选择重传协议

因为滑动窗口使用累计确认的方式，所以会造成很多不必要的分组的重传。使用选择重传协议可以解决这个问题。

选择重传协议在发送方维护了一个发送窗口。发送窗口的以前是已经发送并确认的分组，窗口内包含了已发送但未被确认的分组，已确认的乱序分组，和允许发送但还未发送的分组，发送窗口以后的是缓存中还不允许发送的分组。选择重传协议与滑动窗口协议最大的不同是，发送方发送分组时，为一个分组都创建了一个定时器。当发送方接受到一个分组的确认应答后，取消该分组的定时器，并判断接受该分组后，是否存在由窗口首部为首的连续的确认分组，如果有则向后移动窗口的位置，如果没有则将该分组标识为已接收的乱序分组。当某一个分组定时器到时后，则重新传递这个分组。

在接收方，它会确认每一个正确接收的分组，不管这个分组是按序的还是乱序的，乱序的分组将被缓存下来，直到所有的乱序分组都到达形成一个有序序列后，再将这一段分组交付给上层。对于不能被正确接收的分组，接收方直接忽略该分组。

详细资料可以参考：[《TCP 连续 ARQ 协议和滑动窗口协议》](#)

(5) TCP 的可靠运输机制

TCP 的可靠运输机制是基于连续 ARQ 协议和滑动窗口协议的。

TCP 协议在发送方维持了一个发送窗口，发送窗口以前的报文段是已经发送并确认了的报文段，发送窗口中包含了已经发送但未确认的报文段和允许发送但还未发送的报文段，发送窗口以后的报文段是缓存中还不允许发送的报文段。当发送方向接收方发送报文时，会依次发送窗口内的所有报文段，并且设置一个定时器，这个定时器可以理解为是最早发送但未收到确认的报文段。如果在定时器的时间内收到某一个报文段的确认回答，则滑动窗口，将窗口的首部向后滑动到确认报文段的后一个位置，此时如果还有已发送但没有确认的报文段，则重新设置定时器，如果没有了则关闭定时器。如果定时器超时，则重新发送所有已经发送但还未收到确认的报文段，并将超时的间隔设置为以前的两倍。当发送方收到接收方的三个冗余的确认应答后，这是一种指示，说明该报文段以后的报文段很有可能发生丢失了，那么发送方会启用快速重传的机制，就是当前定时器结束前，发送所有的已发送但未确认的报文段。

接收方使用的是累计确认的机制，对于所有按序到达的报文段，接收方返回一个报文段的肯定回答。如果收到了一个乱序的报文段，那么接方会直接丢弃，并返回一个最近的按序到达的报文段的肯定回答。使用累计确认保证了返回的确认号之前的报文段都已经按序到达了，所以发送窗口可以移动到已确认报文段的后面。

发送窗口的大小是变化的，它是由接收窗口剩余大小和网络中拥塞程度来决定的，TCP 就是通过控制发送窗口的长度来控制报文段的发送速率。

但是 TCP 协议并不完全和滑动窗口协议相同，因为许多的 TCP 实现会将失序的报文段给缓存起来，并且发生重传时，只会重传一个报文段，因此 TCP 协议的可靠传输机制更像是窗口滑动协议和选择重传协议的一个混合体。

(6) TCP 的流量控制机制

TCP 提供了流量控制的服务，这个服务的主要目的是控制发送方的发送速率，保证接收方来得及接收。因为一旦发送的速率大于接收方所能接收的速率，就会造成报文段的丢失。接收方主要是通过接收窗口来告诉发送方自己所能接收的大小，发送方根据接收方的接收窗口的大小来调整发送窗口的大小，以此来达到控制发送速率的目的。

(7) TCP 的拥塞控制机制

TCP 的拥塞控制主要是根据网络中的拥塞情况来控制发送方数据的发送速率，如果网络处于拥塞的状态，发送方就减小发送的速率，这样一方面是为了避免继续增加网络中的拥塞程度，另一方面也是为了避免网络拥塞可能造成的报文段丢失。

TCP 的拥塞控制主要使用了四个机制，分别是慢启动、拥塞避免、快速重传和快速恢复。

慢启动的基本思想是，因为在发送方刚开始发送数据的时候，并不知道网络中的拥塞程度，所以先以较低的速率发送，进行试探，每次收到一个确认报文，就将发送窗口的长度加一，这样每个 RTT 时间后，发送窗口的长度就会加倍。当发送窗口的大小达到一个阈值的时候就进入拥塞避免算法。

拥塞避免算法是为了避免可能发生的拥塞，将发送窗口的大小由每过一个 RTT 增长一倍，变为每过一个 RTT，长度只加一。这样将窗口的增长速率由指数增长，变为加法线性增长。

快速重传指的是，当发送方收到三个冗余的确认应答时，因为 TCP 使用的是累计确认的机制，所以很有可能是发生了报文段的丢失，因此采用立即重传的机制，在定时器结束前发送所有已发送但还未接收到确认应答的报文段。

快速恢复是对快速重传的后续处理，因为网络中可能已经出现了拥塞情况，所以会将慢启动的阀值减小为原来的一半，然后将拥塞窗口的值置为减半后的阀值，然后开始执行拥塞避免算法，使得拥塞窗口缓慢地加性增大。简单来理解就是，乘性减，加性增。

TCP 认为网络拥塞的主要依据是报文段的重传次数，它会根据网络中的拥塞程度，通过调整慢启动的阀值，然后交替使用上面四种机制来达到拥塞控制的目的。

详细资料可以参考：[《TCP 的拥塞控制机制》](#) [《网络基本功：TCP 拥塞控制机制》](#)

5.2.4 网络层

网络层协议主要实现了不同主机间的逻辑通信功能。网络层协议一共包含两个主要的组件，一个 IP 网际协议，一个是路由选择协议。

IP 网际协议规定了网络层的编址和转发方式，比如说我们接入网络的主机都会被分配一个 IP 地址，常用的比如 IPv4 使用 32 位来分配地址，还有 IPv6 使用 128 位来分配地址。

路由选择协议决定了数据报从源到目的地所流经的路径，常见的比如距离向量路由选择算法等。

5.2.5 数据链路层

数据链路层提供的服务是如何将数据报通过单一通信链路从一个结点移动到相邻节点。每一台主机都有一个唯一的 MAC 地址，这是由网络适配器决定的，在全世界都是独一无二的。

5.2.6 物理层

物理层提供的服务是尽可能的屏蔽掉组成网络的物理设备和传输介质间的差异，使数据链路层不需要考虑网络的具体传输介质是什么。

详细资料可以参考：《搞定计算机网络面试，看这篇就够了（补充版）》《互联网协议入门（一）》《互联网协议入门（二）》

5.3 常考面试题

5.3.1. Post 和 Get 的区别？

Post 和 Get 是 HTTP 请求的两种方法。

(1) 从应用场景上来说，GET 请求是一个幂等的请求，一般 Get 请求用于对服务器资源不会产生影响的场景，比如说请求一个网

页。而 Post 不是一个幂等的请求，一般用于对服务器资源会产生影响的情景。比如注册用户这一类的操作。

(2) 因为不同的应用场景，所以浏览器一般会对 `Get` 请求缓存，但很少对 `Post` 请求缓存。

(3) 从发送的报文格式来说，`Get` 请求的报文中实体部分为空，`Post` 请求的报文中实体部分一般为向服务器发送的数据。

(4) 但是 `Get` 请求也可以将请求的参数放入 `url` 中向服务器发送，这样的做法相对于 `Post` 请求来说，一个方面是不太安全，

因为请求的 `url` 会被保留在历史记录中。并且浏览器由于对 `url` 有一个长度上的限制，所以会影响 `get` 请求发送数据时

的长度。这个限制是浏览器规定的，并不是 `RFC` 规定的。还有就是 `post` 的参数传递支持更多的数据类型。

5.3.2. TLS/SSL 中什么一定要用三个随机数，来生成“会话密钥”？

客户端和服务器都需要生成随机数，以此来保证每次生成的秘钥都不相同。使用三个随机数，是因为 `SSL` 的协议默认不信任每个主

机都能产生完全随机的数，如果只使用一个伪随机的数来生成秘钥，就很容易被破解。通过使用三个随机数的方式，增加了自由度，

一个伪随机可能被破解，但是三个伪随机就很接近于随机了，因此可以使用这种方法来保持生成秘钥的随机性和安全性。

5.3.3. SSL 连接断开后如何恢复？

一共有两种方法来恢复断开的 `SSL` 连接，一种是使用 `session ID`，一种是 `session ticket`。

使用 `session ID` 的方式，每一次的会话都有一个编号，当对话中断后，下一次重新连接时，只要客户端给出这个编号，服务器

如果有这个编号的记录，那么双方就可以继续使用以前的秘钥，而不用重新生成一把。目前所有的浏览器都支持这一种方法。但是

这种方法有一个缺点是，`session ID` 只能够存在一台服务器上，如果我们的请求通过负载平衡被转移到了其他的服务器上，那

么就无法恢复对话。

另一种方式是 **session ticket** 的方式，**session ticket** 是服务器在上一次对话中发送给客户的，这个 **ticket** 是加密的

，只有服务器能够解密，里面包含了本次会话的信息，比如对话秘钥和加密方法等。这样不管我们的请求是否转移到其他的服务器

上，当服务器将 **ticket** 解密以后，就能够获取上次对话的信息，就不用重新生成对话秘钥了。

5.3.4. RSA 算法的安全性保障？

对极大整数做因数分解的难度决定了 **RSA** 算法的可靠性。换言之，对一极大整数做因数分解愈困难，**RSA** 算法愈可靠。现在 **102**

4 位的 **RSA** 密钥基本安全，**2048** 位的密钥极其安全。

5.3.5. DNS 为什么使用 **UDP** 协议作为传输层协议？

DNS 使用 **UDP** 协议作为传输层协议的主要原因是为了避免使用 **TCP** 协议时造成的连接时延。因为为了得到一个域名的 **IP** 地

址，往往向多个域名服务器查询，如果使用 **TCP** 协议，那么每次请求都会存在连接时延，这样使 **DNS** 服务变得很慢，因为大

多数的地址查询请求，都是浏览器请求页面时发出的，这样会造成网页的等待时间过长。

使用 **UDP** 协议作为 **DNS** 协议会有一个问题，由于历史原因，物理链路的最小 **MTU = 576**，所以为了限制报文长度不超过 **576**，

UDP 的报文段的长度被限制在 **512** 个字节以内，这样一旦 **DNS** 的查询或者应答报文，超过了 **512** 字节，那么基于 **UDP** 的

DNS 协议就会被截断为 **512** 字节，那么有可能用户得到的 **DNS** 应答就是不完整的。这里 **DNS** 报文的长度一旦超过限制，并不

会像 **TCP** 协议那样被拆分成多个报文段传输，因为 **UDP** 协议不会维护连接状态，所以我们没有办法确定那几个报文段属于同一

个数据，**UDP** 只会将多余的数据给截取掉。为了解决这个问题，我们可以使用 **TCP** 协议去请求报文。

DNS 还存在的一个问题就是安全问题，就是我们没有办法确定我们得到的应答，一定是一个安全的应答，因为应答可以被他人伪造，

所以现在有了 DNS over HTTPS 来解决这个问题。

详细资料可以参考： [《为什么 DNS 使用 UDP 而不是 TCP?》](#)

5.3.6. 当你在浏览器中输入 Google.com 并且按下回车之后发生了什么？

(1) 首先会对 URL 进行解析，分析所需要使用的传输协议和请求的资源的路径。如果输入的 URL 中的协议或者主机名不合法，

将会把地址栏中输入的内容传递给搜索引擎。如果没有问题，浏览器会检查 URL 中是否出现了非法字符，如果存在非法字

符，则对非法字符进行转义后再进行下一过程。

(2) 浏览器会判断所请求的资源是否在缓存里，如果请求的资源在缓存里并且没有失效，那么就直接使用，否则向服务器发起新

的请求。

(3) 下一步我们首先需要获取的是输入的 URL 中的域名的 IP 地址，首先会判断本地是否有该域名的 IP 地址的缓存，如果

有则使用，如果没有则向本地 DNS 服务器发起请求。本地 DNS 服务器也会先检查是否存在缓存，如果没有就会先向根域

名服务器发起请求，获得负责的顶级域名服务器的地址后，再向顶级域名服务器请求，然后获得负责的权威域名服务器的地

址后，再向权威域名服务器发起请求，最终获得域名的 IP 地址后，本地 DNS 服务器再将这个 IP 地址返回给请求的用

户。用户向本地 DNS 服务器发起请求属于递归请求，本地 DNS 服务器向各级域名服务器发起请求属于迭代请求。

(4) 当浏览器得到 IP 地址后，数据传输还需要知道目的主机 MAC 地址，因为应用层下发数据给传输层，TCP 协议会指定源

端口号和目的端口号，然后下发给网络层。网络层会将本机地址作为源地址，获取的 IP 地址作为目的地址。然后将下发给

数据链路层，数据链路层的发送需要加入通信双方的 MAC 地址，我们本机的 MAC 地址作为源 MAC 地址，目的 MAC 地

址需要分情况处理，通过将 IP 地址与我们本机的子网掩码相与，我们可以判断我们是否与请求主机在同一个子网里，如果

在同一个子网里，我们可以使用 ARP 协议获取到目的主机的 MAC 地址，如果我们不在一个子网里，那么我们的请求应该

转发给我们的网关，由它代为转发，此时同样可以通过 ARP 协议来获取网关的 MAC 地址，此时目的主机的 MAC 地址应

该为网关的地址。

(5) 下面是 TCP 建立连接的三次握手的过程，首先客户端向服务器发送一个 SYN 连接请求报文段和一个随机序号，服务端接

收到请求后向服务器端发送一个 SYN ACK 报文段，确认连接请求，并且也向客户端发送一个随机序号。客户端接收服务器的

确认应答后，进入连接建立的状态，同时向服务器也发送一个 ACK 确认报文段，服务器端接到确认后，也进入连接建立

状态，此时双方的连接就建立起来了。

(6) 如果使用的是 HTTPS 协议，在通信前还存在 TLS 的一个四次握手的过程。首先由客户端向服务器端发送使用的协议的版

本号、一个随机数和可以使用的加密方法。服务器端收到后，确认加密的方法，也向客户端发送一个随机数和自己的数字证

书。客户端收到后，首先检查数字证书是否有效，如果有效，则再生成一个随机数，并使用证书中的公钥对随机数加密，然后

发送给服务器端，并且还会提供一个前面所有内容的 hash 值供服务器端检验。服务器端接收后，使用自己的私钥对数据解

密，同时向客户端发送一个前面所有内容的 hash 值供客户端检验。这个时候双方都有了三个随机数，按照之前所约定的加

密方法，使用这三个随机数生成一把秘钥，以后双方通信前，就使用这个秘钥对数据进行加密后再传输。

(7) 当页面请求发送到服务器端后，服务器端会返回一个 `html` 文件作为响应，浏览器接收到响应后，开始对 `html` 文件进行

解析，开始页面的渲染过程。

(8) 浏览器首先会根据 `html` 文件构建 `DOM` 树，根据解析到的 `css` 文件构建 `CSSOM` 树，如果遇到 `script` 标签，则判断

是否含有 `defer` 或者 `async` 属性，要不然 `script` 的加载和执行会造成页面的渲染的阻塞。当 `DOM` 树和 `CSSOM` 树建

立好后，根据它们来构建渲染树。渲染树构建好后，会根据渲染树来进行布局。布局完成后，最后使用浏览器的 `UI` 接口对页

面进行绘制。这个时候整个页面就显示出来了。

(9) 最后一步是 `TCP` 断开连接的四次挥手过程。

详细资料可以参考：《当你在浏览器中输入 `Google.com` 并且按下回车之后发生了什么？》

5.3.7. 谈谈 `CDN` 服务？

`CDN` 是一个内容分发网络，通过对源网站资源的缓存，利用本身多台位于不同地域、不同运营商的服务器，向用户提供就近访问的

功能。也就是说，用户的请求并不是直接发送给源网站，而是发送给 `CDN` 服务器，由 `CDN` 服务器将请求定位到最近的含有该资源

的服务器上去请求。这样有利于提高网站的访问速度，同时通过这种方式也减轻了源服务器的访问压力。

详细资料可以参考：《`CDN` 是什么？使用 `CDN` 有什么优势？》

5.3.8. 什么是正向代理和反向代理？

我们常说的代理也就是指正向代理，正向代理的过程，它隐藏了真实的请求客户端，服务端不知道真实的客户端是谁，客户端请求的

服务都被代理服务器代替来请求。

反向代理隐藏了真实的服务端，当我们请求一个网站的时候，背后可能有成千上万台服务器为我们服务，但具体是哪一台，我们不知

道，也不需要知道，我们只需要知道反向代理服务器是谁就好了，反向代理服务器会帮我们把请求转发到真实的服务器那里去。反向

代理器一般用来实现负载平衡。

详细资料可以参考：[《正向代理与反向代理有什么区别》](#) [《webpack 配置 proxy 反向代理的原理是什么？》](#)

5.3.9. 负载平衡的两种实现方式？

一种是使用反向代理的方式，用户的请求都发送到反向代理服务上，然后由反向代理服务器来转发请求到真实的服务器上，以此来实

现集群的负载平衡。

另一种是 DNS 的方式，DNS 可以用于在冗余的服务器上实现负载平衡。因为现在一般的大型网站使用多台服务器提供服务，因此一

个域名可能会对应多个服务器地址。当用户向网站域名请求的时候，DNS 服务器返回这个域名所对应的服务器 IP 地址的集合，但在

每个回答中，会循环这些 IP 地址的顺序，用户一般会选择排在前面的地址发送请求。以此将用户的请求均衡的分配到各个不同的服

务器上，这样来实现负载均衡。这种方式有一个缺点就是，由于 DNS 服务器中存在缓存，所以有可能一个服务器出现故障后，域名解

析仍然返回的是那个 IP 地址，就会造成访问的问题。

详细资料可以参考：[《负载均衡的原理》](#)

5.3.10. http 请求方法 options 方法有什么用？

`OPTIONS` 请求与 `HEAD` 类似，一般也是用于客户端查看服务器的性能。这个方法会请求服务器返回该资源所支持的所有 `HTTP` 请求方法，该方法会用 '*' 来代替资源名称，向服务器发送 `OPTIONS` 请求，可以测试服务器功能是否正常。`JS` 的 `XMLHttpRequest`

对象进行 `CORS` 跨域资源共享时，对于复杂请求，就是使用 `OPTIONS` 方法发送嗅探请求，以判断是否有对指定资源的访问权限。

相关资料可以参考： [《HTTP 请求方法》](#)

5.3.11. http1.1 和 http1.0 之间有哪些区别？

`http1.1` 相对于 `http1.0` 有这样几个区别：

(1) 连接方面的区别，`http1.1` 默认使用持久连接，而 `http1.0` 默认使用非持久连接。`http1.1` 通过使用持久连接来使多个 `http` 请求复用同一个 `TCP` 连接，以此来避免使用非持久连接时每次需要建立连接的时延。

(2) 资源请求方面的区别，在 `http1.0` 中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，`http1.1` 则在请求头引入了 `range` 头域，它允许只请求资源的某个部分，即返回码是 `206 (Partial Content)`，这样就方便了开发者自由的选择以便于充分利用带宽和连接。

(3) 缓存方面的区别，在 `http1.0` 中主要使用 `header` 里的 `If-Modified-Since, Expires` 来做为缓存判断的标准，`http1.1` 则引入了更多的缓存控制策略例如 `Etag, If-Unmodified-Since, If-Match, If-None-Match` 等更多可供选择的缓存头来控制缓存策略。

(4) `http1.1` 中还新增了 `host` 字段，用来指定服务器的域名。`http1.0` 中认为每台服务器都绑定一个唯一的 `IP` 地址，因此，请求消息中的 `URL` 并没有传递主机名 (`hostname`)。但随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机，并且它们共享一个 `IP` 地址。因此有了 `host` 字段，就可以将请求发往同一台服务器上的不同网站。

(5) `http1.1` 相对于 `http1.0` 还新增了很多方法，如 `PUT, HEAD, OPTIONS` 等。

详细资料可以参考： [《HTTP1.0、HTTP1.1 和 HTTP2.0 的区别》](#) [《HTTP 协议入门》](#) [《网络---一篇文章详解请求头 Host 的概念》](#)

5.3.12. 网站域名加 `www` 与不加 `www` 的区别？

详细资料可以参考：[《为什么域名前要加 `www` 前缀 `www` 是什么意思？》](#) [《为什么越来越多的网站域名不加「`www`」前缀？》](#) [《域名有 `www` 与没有 `www` 有什么区别？》](#)

5.3.13. 即时通讯的实现，短轮询、长轮询、**SSE** 和 **WebSocket** 间的区别？

短轮询和长轮询的目的都是用于实现客户端和服务器端的一个即时通讯。

短轮询的基本思路就是浏览器每隔一段时间向浏览器发送 `http` 请求，服务器端在收到请求后，不论是否有数据更新，都直接进行

响应。这种方式实现的即时通信，本质上还是浏览器发送请求，服务器接受请求的一个过程，通过让客户端不断的进行请求，使得客

户端能够模拟实时地收到服务器端的数据的变化。这种方式的优点是比较简单，易于理解。缺点是这种方式由于需要不断的建立 `ht`

`tp` 连接，严重浪费了服务器端和客户端的资源。当用户增加时，服务器端的压力就会变大，这是很不合理的。

长轮询的基本思路是，首先由客户端向服务器发起请求，当服务器收到客户端发来的请求后，服务器端不会直接进行响应，而是先将

这个请求挂起，然后判断服务器端数据是否有更新。如果有更新，则进行响应，如果一直没有数据，则到达一定的时间限制才返回。

客户端 `JavaScript` 响应处理函数会在处理完服务器返回的信息后，再次发出请求，重新建立连接。长轮询和短轮询比起来，它的

优点是明显减少了很多不必要的 `http` 请求次数，相比之下节约了资源。长轮询的缺点在于，连接挂起也会导致资源的浪费。

SSE 的基本思想是，服务器使用流信息向服务器推送信息。严格地说，`http` 协议无法做到服务

方法，就是服务器向客户端声明，接下来要发送的是流信息。也就是说，发送的不是一次性的数据包，而是一个数据流，会连续不断

地发送过来。这时，客户端不会关闭连接，会一直等着服务器发过来的数据流，视频播放就是这样的例子。**SSE** 就是利用这种机

制，使用流信息向浏览器推送信息。它基于 **http** 协议，目前除了 **IE/Edge**，其他浏览器都支持。它相对于前面两种方式来说，不

需要建立过多的 **http** 请求，相比之下节约了资源。

上面三种方式本质上都是基于 **http** 协议的，我们还可以使用 **WebSocket** 协议来实现。

WebSocket 是 **HTML5** 定义的一个新协

议，与传统的 **http** 协议不同，该协议允许由服务器主动的向客户端推送信息。使用 **WebSocket** 协议的缺点是在服务器端的配置

比较复杂。**WebSocket** 是一个全双工的协议，也就是通信双方是平等的，可以相互发送消息，而 **SSE** 的方式是单向通信的，只能

由服务器端向客户端推送信息，如果客户端需要发送信息就是属于下一个 **http** 请求了。

详细资料可以参考：[《轮询、长轮询、长连接、websocket》](#) [《Server-Sent Events 教程》](#) [《WebSocket 教程》](#)

5.3.14. 怎么实现多个网站之间共享登录状态

在多个网站之间共享登录状态指的就是单点登录。多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统。

我认为单点登录可以这样来实现，首先将用户信息的验证中心独立出来，作为一个单独的认证中心，该认证中心的作用是判断客户端发

送的账号密码的正确性，然后向客户端返回对应的用户信息，并且返回一个由服务器端秘钥加密的登录信息的 **token** 给客户端，该

token 具有一定的有效时限。当一个应用系统跳转到另一个应用系统时，通过 **url** 参数的方式来传递 **token**，然后转移到的应用站

点发送给认证中心，认证中心对 **token** 进行解密后验证，如果用户信息没有失效，则向客户端返回对应的用户信息，如果失效了则将

页面重定向会单点登录页面。

6. 常用工具知识总结

6.1. git 与 svn 的区别在哪里？

git 和 svn 最大的区别在于 git 是分布式的，而 svn 是集中式的。因此我们不能再离线的情况下使用 svn。如果服务器

出现问题，我们就没有办法使用 svn 来提交我们的代码。

svn 中的分支是整个版本库的复制的一份完整目录，而 git 的分支是指针指向某次提交，因此 git 的分支创建更加开销更小

并且分支上的变化不会影响到其他人。svn 的分支变化会影响到所有的人。

svn 的指令相对于 git 来说要简单一些，比 git 更容易上手。

详细资料可以参考： [《常见工作流比较》](#) [《对比 Git 与 SVN，这篇讲的很易懂》](#) [《GIT 与 SVN 世纪大战》](#) [《Git 学习小记之分支原理》](#)

6.2. 经常使用的 git 命令？

```
git init          // 新建 git 代码库  
git add          // 添加指定文件到暂存区  
git rm           // 删除工作区文件，并且将这次删除放入暂存区  
git commit -m [message] // 提交暂存区到仓库区  
git branch       // 列出所有分支  
git checkout -b [branch] // 新建一个分支，并切换到该分支  
git status        // 显示有变更的文件
```

详细资料可以参考： [《常用 Git 命令清单》](#)

6.3. git pull 和 git fetch 的区别

git fetch 只是将远程仓库的变化下载下来，并没有和本地分支合并。

git pull 会将远程仓库的变化下载下来，并和当前分支合并。

[《详解 git pull 和 git fetch 的区别》](#)

6.4. git rebase 和 git merge 的区别

git merge 和 git rebase 都是用于分支合并，关键在 commit 记录的处理上不同。

git merge 会新建一个新的 commit 对象，然后两个分支以前的 commit 记录都指向这个新 commit 记录。这种方法会

保留之前每个分支的 commit 历史。

git rebase 会先找到两个分支的第一个共同的 commit 祖先记录，然后将提取当前分支这之后的所有 commit 记录，然后

将这个 commit 记录添加到目标分支的最新提交后面。经过这个合并后，两个分支合并后的 commit 记录就变为了线性的记

录了。

[《git rebase 和 git merge 的区别》](#) [《git merge 与 git rebase 的区别》](#)

7.面试记录总结

7.1.阿里巴巴（获得 OFFER）

7.1.1. 2019-3-25 阿里巴巴（淘宝）一面

1. 笔试题随机排序
2. 笔试题实现商品分配
3. 浏览器存储机制，`cacheStorage`
4. `cookie` 原理
5. 项目 `mvp` 原理
6. `Vue` 组件间通信
7. 双向绑定的原理
8. 网站性能优化
9. 页面的可用性时间的计算 `performance api`
10. `Webpack` 配置
11. `Webassembly`
12. 网络安全
13. `This` 的指向
14. 前沿知识
15. `Hybrid`
16. `Node.js`
17. 原型链
18. 跨域
19. 移动端的点击事件
20. 移动端布局
21. 前端路由的实现方式

面试时间：3.25 20:10-22:25 135 分钟

7.1.2. 2019-3-28 阿里巴巴（淘宝）二面

1. 项目 `mvp` 模式
2. 图片优化

3. 移动端开发基础
4. WebAssembly

7.1.3. 2019-4-1 阿里巴巴（淘宝）三面

1. 项目介绍
2. Vuex
3. 项目 mvp 模式介绍
4. Ajax 请求创建
5. Promise 调用
6. Flex 布局
7. 盒模型
8. Git rebase
9. 随机排序
10. Promise
11. Fetch 没回答
12. Grid 没回答

面试时间: 4.1 15:20-15:57 37 分钟

7.1.4. 2019-4-3 阿里巴巴（淘宝）四面 (hr)

1. 项目介绍
2. 自我评价、同学评价
3. 家乡
4. 自己的优点
5. 学校课程
6. 未来 5 年的规划

7. 投了哪些公司
8. 学习生涯
9. 有没有女朋友
10. 性格
11. 从项目中学到的东西

面试时间: 4.3 16:20-16:47 27 分钟

7.1.5. 2019-4-29 阿里巴巴（阿里云）一面

1. 快速排序
2. 反转链表
3. 继承
4. 深度优先遍历
5. Es6
6. 情景题 ui 组件设计
7. 列表数据加载问题
8. 懒加载扩展

7.1.6. 2019-5-27 阿里巴巴（阿里云）二面

1. Es6 新特性
2. Object 方法
3. Html5 方法
4. Js 性能优化
5. 快速排序不用递归实现

面试时间: 5.27 20:32-20:58 26 分钟

7.1.7. 2019-5-29 阿里巴巴（淘宝二轮）一面

1. 简历第一个项目
2. 简历第二个项目
3. 简历第三个项目
4. 服务端了解知识
5. Node.js 了解
6. 移动端相关
7. React 基础

面试时间：5.29 16:58-18:35 97 分钟

7.1.8. 2019-5-31 阿里巴巴（淘宝二轮）二面

1. 项目经理
2. 技术选型
3. 为何这样选择
4. 项目提效率提升亮点
5. 项目目的

面试时间：5.31 12:58-13:20 22 分钟

7.1.9. 2019-5-31 阿里巴巴（淘宝二轮）三面（hr）

1. 项目介绍
2. 收获
3. 个人优势
4. 和腾讯 offer 的选择
5. 职业规划
6. 如何获取前端前沿知识

面试时间： 5.31 13:31-13:48 17 分钟

7.2 腾讯（获得 OFFER）

7.2.1. 2019-4-26 腾讯 (TEG) 一面

1. 实习时间
2. 前端项目
3. 节流与防抖
4. margin 重叠
5. BFC
6. This 对象
7. Loader 和 plugin 的差别
8. 原型的获取
9. 单页应用的 seo
10. EventBus
11. Vuex

面试时间： 4.26 18:28-19:05 37 分钟

7.2.2. 2019-4-29 腾讯 (TEG) 二面

1. 项目
2. Vue router 实现
3. Vuex
4. 项目管理规范
5. 5 个价值不同的问题分给 5 个不同的人的方式
6. For of 和 for in 的区别
7. 图片加载
8. 正则表达式
9. 页面遍历

10. 如何判断参数是否传入
11. 路由如何保存滚动位置

面试时间：4.29 19:15-20:30 75 分钟

7.2.3. 2019-5-9 腾讯（TEG）三面

1. 项目经历
2. 最困难的事
3. 兴趣爱好
4. 项目分工、人员讨论
5. 性格
6. 面试是否会准备

面试时间：5.9 19:03-19:43 40 分钟

7.2.4. 2019-5-17 腾讯（TEG）四面（hr）

1. 项目问题
2. 新的解决方案
3. 实验室
4. 最难的问题
5. 家乡
6. 父母工作
7. 对职位的看法
8. 是否支持外地工作
9. 是否有直系亲属在腾讯
10. 实习时间
11. 中间是否回校
12. 兴趣爱好
13. 薪资有要求吗

面试时间：5.17 14:23-14:45 22 分钟

7.3 网易互娱（获得 OFFER）

7.3.1. 2019-4-15 网易互娱一面

-
- 1. 项目介绍
 - 2. Vue 理解
 - 3. 路由的理解
 - 4. ES6 理解

面试时间: 4.15 11:11-11:44 33 分钟

7.3.2. 2019-4-18 网易互娱二面

-
- 1. 项目介绍
 - 2. 权限系统的理解
 - 3. 对于炉石传说的理解
 - 4. 代码规范
 - 5. 商品利润下降的原因

面试时间: 4.18 10:00-10:42 42 分钟

7.4 字节跳动

7.4.1. 2019-3-23 字节跳动一面

-
- 1. Webpack 了解
 - 2. http 缓存
 - 3. http1.0 和 http1.1 的区别
 - 4. css 上下固定为 100px, 中间为自适应高度
 - 5. 一道代码分析题
 - 6. BFC
 - 7. 类数组有哪些, 如何转换

8. 跨域
9. cors 简单请求和复杂请求的区别
10. 项目中图片的性能优化
11. 前端的性能优化
12. Base64 在 html 中的缺点
13. 500 张图片，如何实现预加载优化
14. 二维码扫描登录的原理，服务器推送，客户端轮询

面试时间：3.23 9:00-10:17 67 分钟

7.5 微众银行

7.5.1. 2019-4-1 微众银行一面

1. Vue 双向绑定
2. 虚拟 Dom
3. Diff 算法
4. 闭包
5. 闭包造成内存泄漏举例
6. 继承
7. http 和 https 的区别
8. es6 的了解
9. 是否会愿意留在公司
10. 你的优点
11. 为什么选择微众

面试时间: 4.1 19:00-19:40 40 分钟

7.6 酷家乐（获得 OFFER）

7.6.1. 2019-4-18 酷家乐一面

1. 基本数据类型
2. null 和 undefined 的区别
3. class 相对于 es5 的继承有什么区别
4. 作用域和闭包
5. Webpack loader 和 plugins 的区别
6. 原型链和 this
7. 输入 url 的过程
8. 层叠上下文
9. Git rebase 和 git merge 的区别
10. 前端学习的方式
11. 为什么选择前端

面试时间：4.18 14:35-15:10 35 分钟

7.6.2. 2019-4-22 酷家乐二面

1. 项目介绍
2. Vue 的双向绑定机制
3. 权限管理
4. Vue 中组件通信方式
5. Vue Data 中为什么要使用函数的方式
6. 面向对象设计问题，自动超市购买商品设计
7. 智力题，五只鸡五天能下五个蛋，多少只鸡一百天下一百个蛋
8. 智力题，药罐污染问题

面试时间：4.22 15:06-16:06 60 分钟

7.6.3. 2019-4-25 酷家乐三面

1. 实习时间
2. 前端方向
3. 项目
4. 懒加载
5. 状态持久化
6. 图片优化的方式
7. 浏览器如何判断是否支持 webp 格式图片
8. Display 的常见属性
9. 改变 url 的几种方式

面试时间： 4.25 20:35-21:10 35 分钟

7.7 京东

7.7.1. 2019-4-22 京东一面

1. 前端工程师的理解
2. 项目介绍
3. Vuex 的底层实现
4. Vue router 的实现
5. 缓存的了解
6. Computed 和 watch 的区别
7. Proto 和 prototype
8. Object.defineProperty() 方法
9. 发布订阅者模式和观察者模式的区别
10. no-cache

11. cache-control 可以有几个值，没有限制吧....

12. webpack-loader

13. 其他构建工具

面试时间: 4.22 11:01-10:32 31 分钟

7.8 亿联网络

7.8.1. 2019-4-24 亿联网络一面

1. 项目介绍
2. 懒加载具体实现
3. 数组打平
4. Kmp
5. 磁盘读取
6. 数据库索引

面试时间: 4.24 17:00-17:30 30 分钟

7.9 OPPO (获得 OFFER)

7.9.1. 2019-4-26 OPPO 一面

1. Js 数据类型
2. 判断 Array
3. DOMContentLoaded 事件和 Load 事件的区别
4. 闭包
5. 模块化
6. 模块循环引用
7. Js 文件异步加载

8. Vue 双向绑定

9. Watch 和 computed

10. 虚拟 dom

11. 浏览器缓存

面试时间： 4.26 14:00-14:30 30 分钟

7.9.2. 2019-5-8 OPPO 二面 (hr)

1. 自我介绍
2. 压力最大的时候
3. 学习方法
4. 为什么选择 oppo
5. 实习时间
6. 其他的爱好
7. 其他的公司

面试时间： 5.8 16:30-16:50 20 分钟

7.10 华为 (获得 OFFER)

7.10.1. 2019-4-28 华为一面

1. 项目经历
2. 实习时间

面试时间： 4.28 14:00-14:25 25 分钟

7.10.2. 2019-4-28 华为二面

1. 项目经历
2. 什么是 es5

- 3. Webpack
- 4. 后端框架
- 5. 有女朋友吗
- 6. 工作地点
- 7. 实习时间

面试时间：4.28 14:30-14:55 25 分钟

