

1. Github
2. 关于更新
3. 在线版
4. 300本计算机电子书
5. 如何赞赏
6. 交流
7. 如何写好一份简历?
 - 7.1. 简历模板
 - 7.2. 基本信息
 - 7.3. 专业技能
 - 7.4. 工作经历
 - 7.5. 项目经历
 - 7.6. 奖项荣誉
8. 如何高效刷题?
 - 8.1. 按题目分类刷题
 - 8.2. 难度要循序渐进
 - 8.3. 做好总结
9. 没有项目经验, 怎么办?
10. 面试时间问你的职业规划, 该怎么回答?
11. HR问目前拿到哪几个offer了, 怎么回答好?
12. 面试官: 你有什么要问我的吗?
13. 关于内推
14. Java的特点
15. Java是如何实现跨平台的?
16. Java 与 C++ 的区别
17. JDK/JRE/JVM三者的关系
18. Java程序是编译执行还是解释执行?
19. 面向对象和面向过程的区别?
20. 面向对象有哪些特性?
21. 面向对象编程的六大原则?
22. 数组到底是不是对象?
23. Java的基本数据类型有哪些?
24. 为什么不能用浮点型表示金额?
25. 什么是值传递和引用传递?
26. 了解Java的包装类型吗? 为什么需要包装类?
27. 自动装箱和拆箱
28. String 为什么不可变?
29. 为何JDK9要将String的底层实现由char[]改成byte[]?
30. String, StringBuffer 和 StringBuilder区别
31. 什么是StringJoiner?
32. String 类的常用方法有哪些?
33. new String("dabin")会创建几个对象?
34. 什么是字符串常量池?
35. String最大长度是多少?
36. Object常用方法有哪些?
37. 讲讲深拷贝和浅拷贝?
38. 两个对象的hashCode()相同, 则 equals()是否也一定为 true?
39. Java创建对象有几种方式?
40. 说说类实例化的顺序
41. equals和==有什么区别?
42. 常见的关键字有哪些?
43. final, finally, finalize 的区别
44. final关键字的作用?
45. 方法重载和重写的区别?
46. 接口与抽象类区别?

47. 常见的Exception有哪些？
48. Error和Exception的区别？
49. 运行时异常和非运行时异常（checked）的区别？
50. throw和throws的区别？
51. 通过故事讲清楚NIO
52. BIO/NIO/AIO区别的区别？
53. 守护线程是什么？
54. Java支持多继承吗？
55. 如何实现对象克隆？
56. 同步和异步的区别？
57. 阻塞和非阻塞的区别？
58. Java8的新特性有哪些？
59. 序列化和反序列化
60. 什么时候需要用到序列化和反序列化呢？
61. 实现序列化和反序列化为什么要实现 Serializable 接口？
62. 实现 Serializable 接口之后，为什么还要显示指定 serialVersionUID 的值？
63. static 属性为什么不会被序列化？
64. transient关键字的作用？
65. 什么是反射？
66. 反射有哪些应用场景呢？
67. 讲讲什么是泛型？
68. 如何停止一个正在运行的线程？
69. 什么是跨域？
70. 跨域问题怎么解决呢？
71. 设计接口要注意什么？
72. 过滤器和拦截器有什么区别？
73. 对接第三方接口要考虑什么？
74. 后端接口性能优化有哪些方法？
75. 常见的集合有哪些？
76. List、Set和Map 的区别
77. ArrayList 了解吗？
78. ArrayList 的扩容机制？
79. 怎么在遍历 ArrayList 时移除一个元素？
80. Arraylist 和 Vector 的区别
81. Arraylist 与 LinkedList的区别
82. HashMap
 - 82.1. 解决hash冲突的办法有哪些？HashMap用的哪种？
 - 82.2. 使用的hash算法？
 - 82.3. 为什么建议设置HashMap的容量？
 - 82.4. 扩容过程？
 - 82.5. put方法流程？
 - 82.6. 红黑树的特点？
 - 82.7. 在解决 hash 冲突的时候，为什么选择先用链表，再转红黑树？
 - 82.8. HashMap 的长度为什么是 2 的幂次方？
 - 82.9. HashMap默认加载因子是多少？为什么是 0.75？
 - 82.10. 一般用什么作为HashMap的key？
 - 82.11. HashMap为什么线程不安全？
 - 82.12. HashMap和HashTable的区别？
83. LinkedHashMap底层原理？
84. 讲一下TreeMap？
85. HashSet底层原理？
86. HashSet、LinkedHashSet 和 TreeSet 的区别？
87. 什么是fail fast？
88. 什么是fail safe？
89. 讲一下ArrayDeque？
90. 哪些集合类是线程安全的？哪些不安全？
91. 迭代器 Iterator 是什么？
92. Iterator 和 ListIterator 有什么区别？

- 93. 如何让一个集合不能被修改?
- 94. 并发容器
 - 94.1. ConcurrentHashMap
 - 94.1.1. put执行流程?
 - 94.1.2. 怎么扩容?
 - 94.1.3. ConcurrentHashMap 和 Hashtable 的区别?
 - 94.2. CopyOnWrite
 - 94.3. CopyOnWriteArrayList
 - 94.4. ConcurrentLinkedQueue
 - 94.5. 阻塞队列
 - 94.5.1. JDK提供的阻塞队列
 - 94.5.2. 原理
- 95. 参考链接
- 96. 线程池
 - 96.1. 为什么平时都是使用线程池创建线程, 直接new一个线程不好吗?
 - 96.2. 为什么使用线程池?
 - 96.3. 线程池执行原理?
 - 96.4. 线程池参数有哪些?
 - 96.5. 线程池大小怎么设置?
 - 96.6. 线程池的类型有哪些? 适用场景?
- 97. 进程线程
 - 97.1. 线程的生命周期
 - 97.2. 讲讲线程中断?
 - 97.3. 创建线程有哪几种方式?
 - 97.4. 什么是线程死锁?
 - 97.5. 线程死锁怎么产生? 怎么避免?
 - 97.6. 线程run和start的区别?
 - 97.7. 线程都有哪些方法?
- 98. volatile底层原理
- 99. synchronized的用法有哪些?
- 100. synchronized的作用有哪些?
- 101. synchronized 底层实现原理?
- 102. volatile和synchronized的区别是什么?
- 103. ReentrantLock和synchronized区别
- 104. wait()和sleep()的异同点?
- 105. Runnable和Callable有什么区别?
- 106. 线程执行顺序怎么控制?
- 107. 守护线程是什么?
- 108. 线程间通信方式
- 109. ThreadLocal
 - 109.1. ThreadLocal原理
 - 109.2. ThreadLocal内存泄漏的原因?
 - 109.3. ThreadLocal使用场景有哪些?
- 110. AQS原理
- 111. ReentrantLock 是如何实现可重入性的?
- 112. 锁的分类
 - 112.1. 公平锁与非公平锁
 - 112.2. 共享式与独占式锁
 - 112.3. 悲观锁与乐观锁
- 113. 乐观锁有什么问题?
- 114. 什么是CAS?
- 115. CAS存在的问题?
- 116. 并发工具
 - 116.1. CountDownLatch
 - 116.2. CyclicBarrier
 - 116.3. CyclicBarrier和CountDownLatch区别
 - 116.4. Semaphore
- 117. 原子类

- 117.1. 基本类型原子类
- 117.2. 数组类型原子类
- 117.3. 引用类型原子类
- 118. 为什么要使用Executor线程池框架呢？
- 119. 如何停止一个正在运行的线程？
- 120. 什么是Daemon线程？
- 121. SynchronizedMap和ConcurrentHashMap有什么区别？
- 122. 怎么判断线程池的任务是不是执行完了？
- 123. 什么是Future？
- 124. 事务的四大特性？
- 125. 数据库的三大范式
- 126. 事务隔离级别有哪些？
- 127. 生产环境数据库一般用的什么隔离级别呢？
- 128. 编码和字符集的关系
- 129. utf8和utf8mb4的区别
- 130. 索引
 - 130.1. 什么是索引？
 - 130.2. 索引的优缺点？
 - 130.3. 索引的作用？
 - 130.4. 什么情况下需要建索引？
 - 130.5. 什么情况下不建索引？
 - 130.6. 索引的数据结构
 - 130.7. Hash索引和B+树索引的区别？
 - 130.8. 为什么B+树比B树更适合实现数据库索引？
 - 130.9. 索引有什么分类？
 - 130.10. 什么是最左匹配原则？
 - 130.11. 什么是聚集索引？
 - 130.12. 什么是覆盖索引？
 - 130.13. 索引的设计原则？
 - 130.14. 索引什么时候会失效？
 - 130.15. 什么是前缀索引？
 - 130.16. 索引下推
- 131. 常见的存储引擎有哪些？
- 132. MyISAM和InnoDB的区别？
- 133. MySQL有哪些锁？
- 134. MVCC 实现原理？
- 135. 快照读和当前读
- 136. 共享锁和排他锁
- 137. bin log/redo log/undo log
- 138. bin log和redo log有什么区别？
- 139. 讲一下MySQL架构？
- 140. 分库分表
- 141. 什么是分区表？
- 142. 分区表类型
- 143. 分区的问题？
- 144. 查询语句执行流程？
- 145. 更新语句执行过程？
- 146. exist和in的区别？
- 147. MySQL中int(10)和char(10)的区别？
- 148. truncate、delete与drop区别？
- 149. having和where区别？
- 150. 什么是MySQL主从同步？
- 151. 为什么要做主从同步？
- 152. 乐观锁和悲观锁是什么？
- 153. 用过processlist吗？
- 154. MySQL查询 limit 1000,10 和limit 10 速度一样快吗？
- 155. 深分页怎么优化？
- 156. 高度为3的B+树，可以存放多少数据？

157. MySQL单表多大进行分库分表?
158. 大表查询慢怎么优化?
159. 说说count(1)、count(*)和count(字段名)的区别
160. MySQL中DATETIME 和 TIMESTAMP有什么区别?
161. 说说为什么不建议用外键?
162. 使用自增主键有什么好处?
163. 自增主键保存在什么地方?
164. 自增主键一定是连续的吗?
165. InnoDB的自增值为什么不能回收利用?
166. MySQL数据如何同步到Redis缓存?
167. 为什么阿里Java手册禁止使用存储过程?
168. Redis是什么?
169. Redis优缺点?
170. Redis为什么这么快?
171. 讲讲Redis的线程模型?
172. Redis应用场景有哪些?
173. Memcached和Redis的区别?
174. 为什么要用 Redis 而不用 map/guava 做缓存?
175. Redis 数据类型有哪些?
176. SortedSet和List异同点?
177. Redis的内存用完了会怎样?
178. Redis如何做内存优化?
179. keys命令存在的问题?
180. Redis事务
181. Redis事务支持隔离性吗?
182. Redis事务保证原子性吗, 支持回滚吗?
183. 持久化机制
184. RDB和AOF如何选择?
185. Redis有哪些部署方案?
186. 主从架构
187. 哨兵Sentinel
188. Redis cluster
189. 过期键的删除策略?
190. 内存淘汰策略有哪些?
191. 如何保证缓存与数据库双写时的数据一致性?
192. 缓存常见问题
 - 192.1. 缓存穿透
 - 192.2. 缓存雪崩
 - 192.3. 缓存击穿
 - 192.4. 缓存预热
 - 192.5. 缓存降级
193. Redis 怎么实现消息队列?
194. Redis 怎么实现延时队列
195. pipeline的作用?
196. LUA脚本
197. 什么是RedLock?
198. Redis大key怎么处理?
199. Redis常见性能问题和解决方案?
200. 说说为什么Redis过期了为什么内存没释放?
201. Redis突然变慢, 有哪些原因?
202. 为什么 Redis 集群的最大槽数是 16384 个?
203. 简介
204. 基本概念
205. 为什么要使用消息队列?
206. 使用了消息队列会有什么缺点
207. Exchange 类型
 - 207.1. direct
 - 207.2. fanout

- 207.3. topic
- 207.4. headers
- 208. 消息丢失
 - 208.1. 生产者确认机制
 - 208.2. 路由不可达消息
 - 208.3. 消费者手动消息确认
 - 208.4. 持久化
 - 208.5. 镜像队列
- 209. 重复消费
- 210. 消费端限流
- 211. 死信队列
- 212. RabbitMQ消息怎么路由?
- 213. RabbitMQ如何保证消息的顺序性?
- 214. Kafka 都有哪些特点?
- 215. 请简述下你在哪些场景下会选择 Kafka?
- 216. Kafka 的设计架构你知道吗?
- 217. Kafka 分区的目的?
- 218. 你知道 Kafka 是如何做到消息的有序性?
- 219. Kafka Producer 的执行过程?
- 220. 讲一下你使用 Kafka Consumer 消费消息时的线程模型, 为何如此设计?
- 221. 请谈一谈 Kafka 数据一致性原理
- 222. ISR、OSR、AR 是什么?
- 223. LEO、HW、LSO、LW等分别代表什么
- 224. 数据传输的事务有几种?
- 225. Kafka 消费者是否可以消费指定分区消息?
- 226. Kafka消息是采用Pull模式, 还是Push模式?
- 227. Kafka 高效文件存储设计特点
- 228. Kafka创建Topic时如何将分区放置到不同的Broker中
- 229. 谈一谈 Kafka 的再均衡
- 230. Kafka 是如何实现高吞吐率的?
- 231. Kafka 缺点?
- 232. Kafka 新旧消费者的区别
- 233. Kafka 分区数可以增加或减少吗? 为什么?
- 234. Spring的优点
- 235. Spring 用到了哪些设计模式?
- 236. 什么是AOP?
- 237. AOP有哪些实现方式?
- 238. Spring AOP的实现原理
- 239. JDK动态代理和CGLIB动态代理的区别?
- 240. Spring AOP相关术语
- 241. Spring通知有哪些类型?
- 242. 什么是IOC?
- 243. IOC的好处?
- 244. 什么是依赖注入?
- 245. IOC容器初始化过程?
- 246. Bean的生命周期
- 247. BeanFactory和FactoryBean的区别?
- 248. BeanFactory和ApplicationContext有什么区别?
- 249. Bean注入容器有哪些方式?
- 250. Bean的作用域
- 251. Spring自动装配的方式有哪些?
- 252. @Autowired和@Resource的区别?
- 253. @Qualifier 注解有什么作用
- 254. @Bean和@Component有什么区别?
- 255. @Component、@Controller、@Repository和@Service 的区别?
- 256. Spring 事务实现方式有哪些?
- 257. 有哪些事务传播行为?
- 258. Spring事务在什么情况下会失效?

259. Spring怎么解决循环依赖的问题?

260. Spring启动过程

261. Spring 的单例 Bean 是否有并发安全问题?

262. Spring Bean如何保证并发安全?

263. @Async注解的原理

264. 说说你对 SpringMVC 的理解

265. 什么是MVC模式?

266. SpringMVC 有哪些优点?

267. Spring MVC和Struts的区别

268. Spring MVC的工作原理

269. Spring MVC的主要组件?

270. Spring MVC的常用注解由有哪些?

271. @Controller 注解有什么用?

272. @RequestMapping 注解有什么用?

273. @RestController 和 @Controller 有什么区别?

274. @RequestMapping 和 @GetMapping 注解有什么不同?

275. @RequestParam 和 @PathVariable 两个注解的区别

276. @RequestBody和@RequestParam的区别

277. Spring MVC的异常处理

278. SpringMVC 用什么对象从后台向前台传递数据的?

279. SpringMvc的Controller是不是单例模式?

280. 介绍下 Spring MVC 拦截器?

281. SpringMvc怎么配置拦截器?

282. Spring MVC 的拦截器和 Filter 过滤器有什么差别?

283. 什么是REST?

284. 使用REST有什么优势呢?

285. Mybatis是什么?

286. **ORM是什么**

287. Mybatis和Hibernate的区别?

288. MyBatis框架的优缺点及其适用的场合

289. Mybatis的工作原理

290. Mybatis都有哪些Executor执行器? 它们之间的区别是什么?

291. MyBatis中接口绑定有几种实现方式?

292. Mybatis 是如何进行分页的?

293. 分页插件的基本原理是什么?

294. 简述Mybatis的插件运行原理

295. 如何编写一个插件?

296. .Mybatis 是否支持延迟加载?

297. 延迟加载的基本原理是什么?

298. #{}和\${}的区别是什么?

299. Mybatis的预编译

300. 一级缓存和二级缓存

301. 什么是微服务?

302. 分布式和微服务的区别

303. 服务怎么划分?

304. 微服务设计原则

305. 微服务之间是如何通讯的?

306. 熔断器

307. 服务网关

307.1. 何为网关?

307.2. 何为API网关

308. 服务配置统一管理

309. 服务链路追踪

310. 微服务框架

311. 简介

311.1. 事务

311.2. 分布式事务

311.3. 强一致性、弱一致性、最终一致性

- 312. 分布式事务的基础
 - 312.1. CAP理论
 - 312.2. BASE理论
- 313. 分布式事务解决方案
 - 313.1. 2PC方案
 - 313.2. TCC
 - 313.3. 本地消息表
 - 313.4. MQ事务
 - 313.5. Saga事务
 - 313.6. 最大努力通知方案
- 314. 参考文章
- 315. 统计不同号码的个数
- 316. 出现频率最高的100个词
- 317. 查找两个大文件共同的URL
- 318. 如何在100亿数据中找到中位数?
- 319. 如何查询最热门的查询串?
- 320. 如何找出排名前 500 的数?
- 321. 如何按照 query 的频度排序?
- 322. 大数据中 TopK 问题的常用套路
 - 322.1. 1. 堆排序法
 - 322.2. 2. 类似快排法
 - 322.3. 3. 使用 bitmap
 - 322.4. 4. 使用 hash
 - 322.5. 5. 字典树
 - 322.6. 6. 混合查询
 - 322.6.1. 方法一
 - 322.6.2. 方法二
- 323. 操作系统的四个特性?
- 324. 进程线程
- 325. 并发和并行
- 326. 多线程相较单线程的好处
- 327. 什么是协程?
- 328. 线程和协程有什么区别呢?
- 329. 进程通信
- 330. 什么是死锁?
- 331. 死锁怎么产生? 怎么避免?
- 332. 进程调度策略有哪几种?
- 333. 进程有哪些状态?
- 334. 操作系统里的内存碎片怎么理解?
- 335. 虚拟内存
- 336. 什么是分页?
- 337. 什么是分段?
- 338. 分页和分段有什区别?
- 339. 页面置换算法
- 340. 用户态和内核态
 - 340.1. 为什么要有这两种状态?
 - 340.2. 什么时候转换
- 341. 什么是缓冲区溢出? 有什么危害?
- 342. IO多路复用
- 343. 各种数据结构应用场景
- 344. 数组
- 345. 链表
- 346. 哈希表
- 347. 栈
- 348. 队列
- 349. 树
 - 349.1. 二叉树
 - 349.2. 二叉查找树

- 349.3. AVL树
- 349.4. 红黑树
- 349.5. B树
- 350. 图
 - 350.1. 相关术语
 - 350.2. 类型
 - 350.3. 图的存储结构
 - 350.4. 图的遍历
- 351. 二叉树的遍历
 - 351.1. 前序遍历
 - 351.2. 中序遍历
 - 351.3. 后序遍历
 - 351.4. 层序遍历
- 352. 排序算法
 - 352.1. 冒泡排序
 - 352.2. 插入排序
 - 352.3. 选择排序
 - 352.4. 希尔排序
 - 352.5. 基数排序
 - 352.6. 计数排序
 - 352.7. 快速排序
 - 352.8. 归并排序
 - 352.9. 堆排序
- 353. 动态规划
 - 353.1. 不同路径
 - 353.2. 最长回文子串
 - 353.3. 最大子数组和
 - 353.4. 最长公共子序列
 - 353.5. 接雨水
 - 353.6. 单词拆分
- 354. 回溯算法
 - 354.1. 组合总和
 - 354.2. 全排列
 - 354.3. 全排列II
- 355. 贪心算法
 - 355.1. 买卖股票的最佳时机 II
 - 355.2. 跳跃游戏
 - 355.3. 加油站
- 356. 双指针
 - 356.1. 反转链表
 - 356.2. 反转链表II
 - 356.3. 删除链表倒数第n个节点
 - 356.4. 三数之和
 - 356.5. 环形链表
 - 356.6. 环形链表II
- 357. 字节跳动抖音电商
- 358. 美团二面
- 359. Java
- 360. Java Web
- 361. 数据库
- 362. 框架
 - 362.1. Spring
 - 362.2. SpringMVC
 - 362.3. Mybatis
 - 362.4. SpringBoot
 - 362.5. SpringCloud
- 363. 中间件
- 364. JVM

- 365. 并发
- 366. 计算机基础知识
 - 366.1. 数据结构和算法
 - 366.2. 计算机网络
 - 366.3. 操作系统
 - 366.4. linux

前言

大家好，我是大彬。

最近在面试，看了很多面经，饱受Java八股文的折磨。于是大彬花了几个月时间将常见的面试题总结了一下，汇集成**面试手册**。这些题目都是我自己整理的，不是网上复制粘贴来的，希望对小伙伴们有所帮助！

因个人水平有限，难免会有出错、理解的不对的地方，希望大家可以及时指正~

有更好的见解也欢迎与我交流，我的微信号是i_am_dabin，QQ邮箱 1713476357@qq.com。

1. Github

手册也保存在Github/Gitee上，有问题也可以在上面提issue，当然大家有想补充的面试题，也可以提PR，贡献一份力量。

github地址: <https://github.com/Tyson0314/Java-learning>

gitee地址: <https://gitee.com/tysondai/Java-learning>

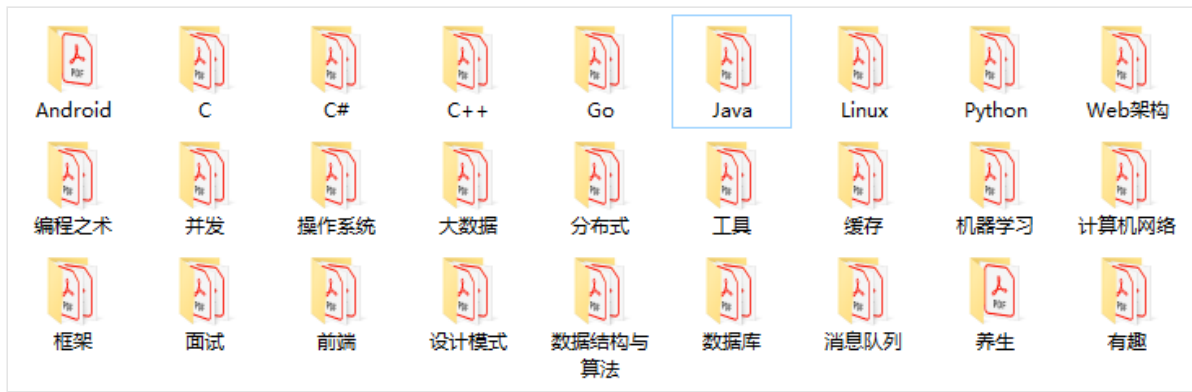
2. 关于更新

《面试手册》目前计划一个月左右更新一次，大家可以在公众号【程序员大彬】后台回复【手册】即可获取最新版！如果觉得内容不错的话，欢迎转发分享！

3. 在线版

大彬的[小破站](#)已上线，本手册上的所有内容都会同步到小破站上，欢迎大家访问~





Github地址: <https://github.com/Tyson0314/java-books>

5. 如何赞赏

如果觉得本文档对你有帮助的话，欢迎加入我的[知识星球](#)。创建[知识星球](#)主要是为了帮助那些**自学编程、想要拿到更好offer、想拥有一个良好的学习环境、抱团取暖**的小伙伴，在你们遇到问题的时候，可以在学习圈进行讨论交流，或者**向大彬提问**，帮助你更快得到解答。另外，知识星球可以将知识沉淀下来，管理更高效。

目前大彬的学习圈已经积累了很多优质内容了，像**Java面试手册完整版（星球专属）、高频场景设计题目、LeetCode刷题笔记**等。



大彬 送你一张星球优惠券

「大彬的学习圈」

立减

¥ 50

新人立减券

2023/12/12 13:00 后失效

 知识星球

长按扫码领取优惠 ▶



6. 交流

此外，我建了一个**互联网求职交流群**，群里会分享各大互联网公司校招资讯、面试和笔试题目，也可以获得阿里、字节等大厂的**内推机会**。交流群**学习氛围**很浓厚，截个图给大家感受下~



有需要的小伙伴可以**扫码加我微信**，我拉你进群（群成员**超过200**，不能直接扫码进群，所以可以加我微信，我拉你进群哈）



面试前准备

1. 如何写好一份简历？

很多同学刚开始找工作时，投出去很多简历，但是都石沉大海了，没有后文。之所以简历不通过，往往都是简历不够“好看”。

很多大公司HR经常一天要看几百份，甚至上千份简历，基本都是10秒内看一份简历，就是这关键的10秒，HR就决定了要邀约面试还是PASS。所以，写好简历非常重要！一份好的简历可以帮你更容易地通过筛选，进入面试阶段。

接下来聊一下怎么写好简历。

分成以下几个方面：

- 简历模板
- 基本信息
- 专业技能
- 工作经历
- 项目经历
- 奖项荣誉

这些都是一个简历必要的信息。

1.1. 简历模板

像我们搞技术的，选个简约的模板就可以了，不要太多花哨的东西，简历太过花里胡哨，往往会引起HR和面试官反感，很有可能就失去了这次面试机会。

像下面这份简历就是反例，不够简洁，不仅不能吸引到面试官，反而更容易被面试官PASS掉。

 OfficePLUS 高级平面设计师	工作经历 微软在线 OfficePLUS 2013.06-2015.09 平面设计师 - 负责公司线上宣传内容的设计，包括网站 Banner，UI，电子海报等 - 负责公司线下物料的制作，设计了 dm 单，文化衫，纪念品，名片等物料。 - 负责公司新版 logo 设计工作 Office PLUS 4A 广告公司 2012.06-2013.06 平面设计师 - 负责 XX 公司全套 VI 设计工作 - 为 XX 公司制作地铁宣传海报 - 参与超过 10 家公司的线下物料设计公司 必应中国 Bing 2011.06-2012.04 平面设计师实习 - 负责公司各种线下物料的制作 - 制作网站 Banner，微博微信宣传图片 北京 PLUS 街拍社 2014.06-2015.09 社长/摄影师 - 社团的主要发起人之一，担任摄影师，为多个社团组织进行拍摄 - 一年时间，社团由 6 个人发展为拥有超过 20 名成员，50 名会员的校园知名社团 - 若要添加内容直接复制网站相应内容进行修改	
专业技能 PS AI CDR PPT FLASH HTML 5 Freehand 点击技能条后，再点击一次灰色条，按住 shift 拖动灰色条右侧。	教育背景 OfficePLUS 美术学院 本科 2008-2012 平面设计专业	所获荣誉 2014 全国平面设计金奖 2013 北京市 XXX 设计金奖 2013 Adobe 设计奖 2012 Microsoft 认证设计师
		手机 / 18010101010 邮箱 / Officeplus@Microsoft.com QQ / 3093242906 地址 / 北京市海淀区中关村东路 1 号清华科技园 D 座 15 层 (100084) ;

而下面这份简历，相比之下就整洁很多，看起来清爽很多，也是我自己个人在用的简历模板。目前投出去的简历，基本都能有面试机会。

求职意向: 市场运营专员

生日: 1991.05 性别: 女 邮箱: service@500d.me

民族: 汉族 电话: 135xxxxxxx 地址: 广东省广州市海珠区

教育背景

★ 2005.07-2009.06 市场营销 (本科)

主修课程: 管理学、微观经济学、宏观经济学、管理信息系统、统计学、财务管理、市场营销、经济法

校园经历

★ 2009.03-2011.06 校园大使主席

目标带领自己的团队, 辅助完成在各高校的“伏龙计划”, 向全球顶尖的 AXA 金融公司推送实习生资源。

整体运营前期开展了相关的线上线下宣传活动, 中期为进行咨询的人员提供讲解。后期进行了项目的维护阶段, 保证了整个项目的完整性。

实习经历

★ 2018-04 至今 阿里巴巴 后端开发

★ 2015.03-2018.03 字节跳动 软件工程师

负责公司业务系统的设计及改进, 参与公司网上商城系统产品功能设计及实施工作。

负责客户调研、客户需求分析、方案写作等工作, 参与公司多个大型电子商务项目的策划工作。

奖项荣誉

★ 2014 年暑假社会实践活动“优秀团队二等奖”

★ 2014 年暑假社会实践活动“优秀调研报告”

★ 2015 年暑假社会实践活动“先进个人”

技能证书

★ 资格证书: 正在参加 CFA 的考试, 已过 Level 1; CPA 已经通过考试资格许可, 计划在年底考完四个科目; CFP 已经在研究生期间学习完了所有要求的科目, 计划在明年通过考试。

★ 语言能力: 在美国留学 6 年, 有地道的口语, 在美国公司工作了一年半, 能够在全英文的环境下熟练与人沟通和表达。同时也具备流利的普通话能力, 具备良好的沟通和人际关系技巧。

自我评价

★ 营销专业的学科背景, 系统学习了定性与定量的研究方法, 掌握了问卷、访谈等数据搜集技术, 可通过前后测进行需求评估及用户研究, 并结合 SPSS、Excel 等进行数据处理分析工作;

错别字绝对是零容忍, 如果应聘者连自己的简历都不重视, 那么HR更不会重视, 反手就是挂掉。所以写完简历以后一定要仔细检查, 可以让朋友帮忙看一遍, 让他给下建议。

还有比较重要的一点是, 简历的格式保存为PDF之后, 再往招聘网站或者邮箱投递, 不要使用word格式的, 因为word很多版本, 容易出现版本不兼容的情况, 导致排版混乱, 影响应聘。

1.2. 基本信息

个人信息主要有:

姓名: 这个不用说了, 总不该写错吧。

电话: 记住填写自己常用的手机号码, 有些同学有两个号码的, 更要注意! 还有就是, 千万别写错号码, 写错了用人方想电话联系就联系不上了, 在候选人比较多的情况下, 你可能就被PASS了。

电子邮箱：这个跟电话一样，也是很重要的联系方式。很多笔试面试邀请都是通过邮箱发出，offer发放也是直接发到邮箱，也不可以出错。至于用什么邮箱，随个人喜好，我是用的QQ邮箱。

求职意向：写明岗位，后端、前端、测试、运维等等。

工作年限：应届生，实习生等不需要写，工作的小伙伴就写上对应的年限。很重要的一点就是，不要造假！现在是互联网时代，学历信息在学信网一查就知道。

博客/开源项目：没有的话，可不写。如果你有好的开源项目或者博客，面试会很加分。但是要有一定的含金量，不是烂大街的项目或者博客。

1.3. 专业技能

见过很多同学简历上出现精通两个字，精通Java、精通MySQL、精通Redis等等，切记不要用这个词，面试官自己可能都不敢说自己精通，写上精通会吓到面试官的。一般写熟悉或者熟练掌握等，当然如果你真的牛，能够扛得住面试官的灵魂拷问，那写上精通也没问题。

可以到招聘网站上看看应聘岗位的岗位要求，下面是阿里巴巴国际事业部java开发岗位的要求：

1. 本科及以上学历，计算机软件或相关专业；
2. 2年及以上使用JAVA开发的经验，JAVA基础扎实，理解io、多线程、集合等基础框架，对JVM原理有一定的了解，对Spring,ibatis,struts等开源框架熟悉；
3. 熟悉分布式系统的设计和应用，熟悉分布式、缓存、消息等机制；能对分布式常用技术进行合理应用，解决问题；
4. 掌握多线程及高性能的设计与编码及性能调优；有高并发应用开发经验；
5. 掌握Linux 操作系统和大型数据库（Oracle、MySql）；对sql优化有丰富的经验；
6. 学习能力强，适应能力好，有强烈的责任心，具备耐心/细心的品质；
7. 有良好英语沟通者优先考虑。

基本就是Java基础、并发、数据库、中间件、框架等，可以参考以下模板：

1. 熟练使用Java面向对象编程，具有良好的编程习惯以及CSDN技术文档编写习惯；
2. 熟练使用Eclipse/IDEA等开发工具；
3. 熟练SVN，Maven，Git等项目管理工具；
4. 熟练Spring，SpringMVC，Hibernate，MyBatis等开源框架技术；
5. 掌握SpringBoot+SpringCloud微服务架构；
6. 掌握MySQL数据库及其优化，了解Oracle
7. 掌握Redis中央缓存和ES，Lucene全文检索，分布式文件系统FastDFS；
8. 掌握HTML，CSS，JavaScript，Ajax，Bootstrap，jQuery，EasyUI，layui，Vue，Node.js等前端开发技术；
9. 掌握FreeMarker，Velocity，Thymeleaf等模板引擎技术
10. 了解Linux常用命令及阿里云部署项目；
11. 了解Docker，Nginx；
12. 了解微信公众号、小程序的开发；
13. 具体所会的技术点可查看个人技术博客 [\[链接\]](#)
14. GitHub开源项目：[\[链接\]](#)

1.4. 工作经历

- **在职时间：**如实填写，造假的话，一查社保公积金就被发现了，后面就别想进这家公司了，得不偿失。
- **公司名称：**还是一样，不能造假。无论大厂小厂，都如实填写。
- **岗位：**主要看你以前所做的工作，跟你应聘的岗位相关度如何。建议还是在一个领域内深造，不要频繁换岗，一会搞前端，一会做后台，这样不利于自身发展。

不要频繁跳槽，频繁跳槽会给人留下不好的印象，像某些大厂招聘要求五年内只在两家公司工作过（五二原则），所以还是沉住心，尽量不要短时间多次跳槽。

1.5. 项目经历

项目名称：主要负责的项目。

项目描述：简要描述项目背景、项目内容。

技术栈：列举项目中的技术栈，如果是当下比较流行的技术，那面试的时候会加分。不是比较大众的技术也没关系，只要对所用的技术有深入的了解，面试的时候也能打动面试官。

项目职责：写清楚你在项目中的角色，以及个人职责，比如你在项目中负责后端开发，具体负责哪一块业务等等。

难点攻克：很大的加分项。解决过什么疑难问题，**性能优化**（最好有数据支撑，比如SQL执行时间从10秒到50毫秒这样）。可以说是简历上最大的亮点项了。

```
项目名称：[模糊]
项目描述：该系统主要为用户提供[模糊]，比如提供[模糊]、[模糊]、[模糊]等服务，主要有用户模块，服务模块，评论模块，订单模块，资讯模块，基础数据模块等
项目架构：Spring+SpringMVC+MyBatis+Layui
负责模块：用户模块，登录模块，后台页面搭建
技术要点：
1) 基础数据的CRUD
2) MyBatis-generator代码生成器技术
3) QQ第三方登录
4) 使用百度地图的API[模糊]路线
5) 基于Layui的后台页面搭建
6) Lucene查询，关键字高亮显示
7) EasyPoi完成报表的导入导出
```

写到简历上的项目，一定是自己比较熟悉的，面试官可能会深挖里面的一些细节，为什么这么设计，有没有其他替代方案等等，只有自己亲手做过的项目，才能应对面试官的灵魂拷问，不然很容易就gg了。例如大彬部署过nginx服务器，没有很深入了解nginx，然后我在简历上写熟悉nginx，面试官看到我的简历之后，围绕着nginx一波询问，然后我说我只部署过nginx，其他关于nginx底层原理我都不了解，这不就很尴尬了。

关于项目经历这块，面试常见的问题：

- 负责项目的哪块内容？
- 选用这些技术的原因？
- xx功能怎么设计？
- 项目中遇到过什么疑难Bug？
- 使用一些开源框架的时候有没有遇到什么问题，怎么解决的？
- ...

项目经历中要突出自己的贡献，不要只会描述项目，要突出自己的贡献，做了哪些功能，还是优化了什么性能参数，最后说说带来什么样的影响，例如这个功能被多少人使用，性能提升了多少倍。

1.6. 奖项荣誉

校招的话，如果拿到奖学金、ACM比赛拿过奖，可以写上去，会很加分。有些公司还很看重四六级证书，如果四六级比较高分，也可以放上去。

社招的话，一般比较看重项目经验，这一块可以弱化，去掉也没关系。

最后，大彬精心挑选了一些**简历模板**，有需要的小伙伴可以自行下载，[下载地址](#)

2. 如何高效刷题？

很多小伙伴刚开始刷LeetCode时，可能会吃力，经常刷了忘，忘了再刷，进入死循环，刷题效率很差。

其实，刷题也要讲究技巧，掌握了正确的刷题方法，便能事半功倍~

另外送大家一本谷歌大佬撰写的**算法手册**，整整 **300** 道 LeetCode 题目，并且都是**最优解**，非常强！这本手册帮助不少朋友加入大厂，大家加油！

1 题目分类	1	7.4 分割类型题	47
2 最易懂的贪心算法	3	7.5 子序列问题	49
2.1 算法解释	3	7.6 背包问题	51
2.2 分配问题	3	7.7 字符串编辑	57
2.3 区间问题	5	7.8 股票交易	59
2.4 练习	6	7.9 练习	62
3 玩转双指针	8	8 化繁为简的分治法	64
3.1 算法解释	8	8.1 算法解释	64
3.2 Two Sum	9	8.2 表达式问题	64
3.3 归并两个有序数组	10	8.3 练习	66
3.4 快慢指针	10	9 巧解数学问题	67
3.5 滑动窗口	11	9.1 引言	67
3.6 练习	13	9.2 公倍数与公因数	67
4 居合斩！二分查找	14	9.3 质数	67
4.1 算法解释	14	9.4 数字处理	69
4.2 求开方	14	9.5 随机与取样	71
4.3 查找区间	15	9.6 练习	74
4.4 旋转数组查找数字	17	10 神奇的位运算	76
4.5 练习	18	10.1 常用技巧	76
5 千奇百怪的排序算法	19	10.2 位运算基础问题	76
5.1 常用排序算法	19	10.3 二进制特性	78
5.2 快速选择	21	10.4 练习	80
5.3 桶排序	22		
5.4 练习	23		

需要的可以自行下载：

http://mp.weixin.qq.com/s?__biz=Mzg2OTY1NzY0MQ==&mid=100000413&idx=1&sn=adaa8dc2633ac97e25657f43420ce7cd&chksm=4e98ffdb79ef76cd6b98cdbe6cabb46459e283ca16de879f310b730d670e7dfe2514908e5eb0#rd

下面分享几点我自己的刷题经验。

主要有以下3点技巧：

1. 按题目分类来刷。
2. 难度要循序渐进。
3. 做好总结。

2.1. 按题目分类刷题

LeetCode上面的题目都有进行分类，建议在一个时间段只刷同一类型的题目，可以更全面的认识这一类型的数据结构or算法，以加深对此类题型的理解。就好比练功夫，前期把一些基本招式都熟悉掌握，后面再串通这些招式，融会贯通。

我个人也是比较习惯按照分类来刷题，自我感觉效果还可以。

我将LeetCode题目进行了整理分类，大家可以参考下：

数组操作

- LeetCode54 螺旋矩阵
- LeetCode76 最小覆盖子串
- LeetCode75 颜色分类
- LeetCode73 矩阵置零
- LeetCode384 打乱数组
- LeetCode581 最短无序连续子数组
- LeetCode945 使数组唯一的最小增量

链表操作

- LeetCode206 反转链表
- LeetCode19 删除链表的倒数第N个节点
- LeetCode25 k个一组翻转链表
- LeetCode141 环形链表
- LeetCode142 环形链表II
- LeetCode61 旋转链表
- LeetCode138 复制带随机指针的链表
- LeetCode160 相交链表
- LeetCode707 设计链表

栈

- LeetCode20 有效的括号
- LeetCode32 最长有效括号
- LeetCode155 最小栈
- LeetCode224 基本计算器
- LeetCode232 用栈实现队列
- LeetCode316 去除重复字母

树的遍历

- LeetCode94 二叉树的中序遍历
- LeetCode102 二叉树的层次遍历
- LeetCode110 平衡二叉树
- LeetCode144 二叉树的前序遍历
- LeetCode145 二叉树的后序遍历

二叉搜索树

- LeetCode98 验证二叉搜索树
- LeetCode450 删除二叉搜索树中的节点
- LeetCode701 二叉搜索树中的插入操作

递归

- LeetCode21 合并两个有序链表
- LeetCode101 对称二叉树
- LeetCode104 二叉树的最大深度
- LeetCode226 翻转二叉树
- LeetCode236 二叉树的最近公共祖先

双指针/滑动窗口

- LeetCode3 无重复字符的最长子串
- LeetCode11 盛最多水的容器
- LeetCode15 三数之和
- LeetCode16 最接近的三数之和
- LeetCode26 删除排序数组中的重复项
- LeetCode42 接雨水
- LeetCode121 买卖股票的最佳时机
- LeetCode209 长度最小的子数组

快慢指针遍历

- LeetCode141 环形链表
- LeetCode202 快乐数
- LeetCode876 链表的中间结点

动态规划

- LeetCode5 最长回文子串
- LeetCode53 最大子序和
- LeetCode62 不同路径
- LeetCode64 最小路径和
- LeetCode70 爬楼梯
- LeetCode118 杨辉三角
- LeetCode300 最长上升子序列
- LeetCode1143 最长公共子序列

回溯算法

- LeetCode10 正则表达式匹配
- LeetCode22 括号生成
- LeetCode40 组合总和2
- LeetCode46 全排列

贪心算法

- LeetCode 11. 盛最多水的容器
- LeetCode 406. 根据身高重建队列
- LeetCode 55. 跳跃游戏
- LeetCode 122. 买卖股票的最佳时机 II
- LeetCode 309. 最佳买卖股票时机含冷冻期
- LeetCode 714. 买卖股票的最佳时机含手续费

并查集

- LeetCode200 岛屿的个数
- LeetCode547 省份数量

位运算

- LeetCode52 N皇后II
- LeetCode338 比特位计数
- LeetCode191 位1的个数
- LeetCode231 2的幂

2.2. 难度要循序渐进

这一点是针对初学者来说的，切记一上来就干hard级别的题目，会让你怀疑人生的。。。

正确的做法是**循序渐进**，从容易到中等，再过渡到困难级别。不过国内大厂考察算法，一般都是中等难度，困难级别的应该很少考察。

2.3. 做好总结

多做总结！多做总结！多做总结！

做好总结很重要，特别是对于没思路的题目，看了其他大佬的解法之后，多思考有哪些题目也是类似解法，这种题目的关键解题步骤，把自己的理解写下来，方便自己日后查看。

虽然总结可能会花费你半个钟甚至更多的时间，但是不总结的话，下次你遇到这个题目，可能会花更多的时间去思考、解答。

大彬也花了很长时间写了题解，目前更新在小破站上，需要的可以参考下：<https://topjavaer.cn/leetcode/>

3. 没有项目经验，怎么办？

[没有项目经验，怎么办？](#)

4. 面试时问你的职业规划，该怎么回答？

建议紧扣工作和学习两个维度回答

第一点: 介绍自己认真思考过这个问题，自己的规划是基于目前的实际情况来设计的，不是凭空想的。加分!加分! 加分!

第二点: 工作方面，突出自己打算通过积极完成工作任务，积累各方面的经验，让自己成为这个领域的专业人士，也希望有机会能够带领团队，成为优秀的管理者，为单位做出更大贡献，获得双赢。加分!加分! 加分!

第三点: 在学习方面，打算在专业领域做进一步学习和研究，将实践经验与专业知识相结合，为自己的职业成长做好铺垫，打好基础。加分!加分!加分!

根据以上的关键点，参考回复:

参考回复1:

首先，自己认真思考过这个问题，自己的规划是基于目前的实际情况来设计的，不是凭空想的，自己毕竟刚毕业或者工作时间不长，经验还欠缺

其次，在工作方面，我打算通过积极完成工作任务，积累各方面的经验，让自己成为这个领域的专业人士，也希望有机会能够带领团队，成为优秀的管理者，为单位做出更大贡献

第三，在学习方面，打算在专业领域做进一步学习和研究，将实践经验与专业知识相结合，为自己的职业成长做好铺垫，打好基础。谢谢!

参考回复2:

未来一年：我希望通过自己的学习和实践，加深对行业和岗位的理解，能够独立完成一些小项目，推动业务的发展。对此，我会针对性的看一些相关的书籍，与公司和行业的前辈多沟通交流，多调研用户加深业务的熟悉程度，并有机会成为业务本身的用户。

未来三年：可以同时独立负责几个项目，在完成本身任务的基础上主动做优化和迭代，总结形成自己的方法论和工作方式。对此，我会多整理复盘学习成果，延伸学习产品、技术等专业知识。

未来五年：可以带自己的team或者实习生，将所学所得与他们进行沟通交流，从团队协作和业务两方面共同提升团队效益；同时有机会跟业内的大牛老师交流学习共事。对此，我会学习一些心理学、管理学等相关知识，拓宽对团队管理方面的学习，并拓展在业务本身和上下游各联动产业的学习。

Tips: 回答这个问题强调你稳定性，踏实工作的态度，重点在工作技能方面的提升与内在积累，不要描述外在的东西，比如职位，薪资。

5. HR问目前拿到哪几个offer了，怎么回答好？

这是比较常见的面试问题。

HR提问的目的是要把握候选人的求职进展情况。

如果你手里已经有Offer了，公司综合评估你是比较适合的人选，就会争取尽快和你进行薪酬谈判，尽快发Offer，推进的速度可以很快。

HR也会根据你拿到的Offer的情况，评估你在市场上对标的位置——有助于HR评估你接公司Offer，并且顺利入职的可能性。假如你拿到的offer是30w左右，而HR根据你前边面试情况定级，最高只能给到20w，那么这种情况大概率你会选择30w年薪的offer，HR可能就不会给你发offer了，而是考虑其他有可能入职的应聘者。

那么该如何回答这个问题呢。

没有Offer

如果你目前还没有offer，可以说“目前没有，有两家公司在谈，还没发书面Offer。”

或者说：“还没有，我刚刚开始看机会。”

如果你真的是面试很多家，但是都没有什么成果。这个千万不要让对方知道，不然对方通过该信息得出的结论就是：这个人可能水平不是很高，没啥竞争力。可能HR面试你就会被挂了。

不管面了多少家，只要没有Offer，都可以说刚刚开始面试。

手头有Offer，还在择优选择中

如果你手里有Offer，可以坦诚说大体信息。我们的目的是拿到所有Offer，所以，一定要表达，我更倾向你公司Offer。

比如我目前手里有两个Offer，但是我更倾向加入咱们公司，因为咱们公司发展前景更好之类的话，让HR相信你是想加入他们公司的。

如果对方没有问你什么公司，Offer多少。你就不要继续讲具体的内容。

可能HR会追问“是哪家公司？给你Offer多少？”

这个时候，我建议不要透露公司具体的名字。可以说某行业，比如“我有两个Offer，一个是金融行业独角兽，一个是互联网公司的”。这样含糊回答即可。

另外，不要说有太多Offer，即便你真的有很多，就说2-3个。站在HR的角度，候选人手里那么多Offer，还没确定去哪家，在市场上不断面试，大概率是来练手的。这种情况在HR面试估计就被pass了。

总之，回答要往对自己有利的方向进行，还有就是自信。

参考链接：<https://www.zhihu.com/question/23751641>

6. 面试官：你有什么要问我的吗？

很多时候，在面试接近尾声的时候，面试官会问应聘者一个问题：“你有什么要问我的吗？”。

有些人没面试经验比较少，一下子没想到合适的问题，回答“没有”，然后就结束了。其实这样就白白浪费了一个了解公司、团队的好机会，而且可能给面试官一个不好的印象，他可能觉得你对他们公司并不感兴趣。

今天就来跟大家聊聊怎样去回答这个问题。

大部分公司都是有2~5轮面试，每一面的面试官不一样，一般是项目负责人、部门负责人或者HR，要根据面试官的身份来进行提问。

1. 面试官是项目负责人

可以提问跟岗位内容相关的具体问题，比如岗位所需能力、团队分工、工作内容等，重点在于体现专业性和利他性。比如：

- 您认为我目前的能力跟应聘岗位之间的匹配度如何？
- 请问您所在的团队整体工作流程是怎样的？如何进行分工的？
- 请问我如果有幸加入您的团队，我应该是负责哪一块业务？可以给我详细介绍下这一块的业务吗？

2. 面试官是部门负责人

如果是面试官是部门负责人的话，可以了解关于工作和岗位、考核相关的问题，比如：

- 如果我能有幸加入您负责的部门，您期望我能给团队带来哪些价值？
- 您认为考核这个岗位员工的最重要指标有哪些？

也可以提一些关于行业和企业战略的问题，比如

- 您对这个行业未来的发展怎么看？您对企业未来的战略有什么想法？

3. 面试官是HR

针对HR面试，可以了解以下内容：

1. 应聘岗位情况(很多时候招聘网站上放的是通用JD，通用JD的问题就在于岗位的具体需求可能会被遗漏)；
2. 职业发展(员工在企业内的晋升空间，轮岗机会以及培训机会)

具体可以提问的问题如下：

1. 以您来看，这个岗位未来在公司内部的发展如何？
2. 这个职位未来几年的职业发展是怎样的？
3. 我想了解一下公司的培训机制和学习机制？
4. 请问下我应聘的岗位的晋升机制是怎样的？

总的提问原则只有一个：**把合适的问题给到合适的人，不给对方制造麻烦。**

另外附上网上看到的**面试50问**，供大家参考~

面试50问

职责篇

1. 工作时间/计划是怎样的？

2. 平时会处理哪些任务？
3. 团队中初级人员和高级人员是如何平衡的？
4. 针对新员工有哪些培训？
5. 如果按照工作计划执行，有多少工作是需要自己独立完成的？
6. 完成核心工作大概大概需要多久？
7. 对这个岗位的定义是什么？

技术篇

1. 你常用的堆栈
2. 你如何用源代码
3. 你如何测试代码？
4. 你平时如何追踪bug？
5. 如何集成和部署更改，CI/CD吗？
6. 基础架构
7. 从规划到完成任务的工作流程是什么？
8. 怎么为灾难恢复
9. 是否有标准化的开发环境
10. 可以以多快的速度为产品设置新的本地测试环境
11. 可以以多快的速度响应代码或依赖项中的安全问题？
12. 是否允许所有开发人员拥有其计算机的本地管理员权限？

团队篇

1. 这项工作是如何组织的？
2. 团队内/团队间的沟通情况是怎样的？
3. 遇到了意见分歧该如何解决？
4. 设定优先事项/时间表的人是谁？
5. 不能在预期时间内完成会怎样？
6. 每周都开啥会？
7. 产品/服务时间表是怎样的？（可以从多长时间发布一次/持续部署时长/多个发布流
8. 出现生产事故后怎么处理？
9. 团队正在经历的尚未解决的挑战是什么？

公司篇

1. 是否有会议/旅行预算，使用规则是什么？
2. 晋升过程是怎样的？
3. 是否设置了单独的技术向或管理向的职业发展
4. 年假、事假、病假、产假等每年都有多少天？
5. 对多元化招聘有什么看法？
6. 公司内部是否有自己的学习资源，比如电子订阅文档或在线课程等？
7. 有获得认证的预算吗？
8. 公司什么时候会达到成熟阶段？
9. 我能为FOSS项目做贡献吗？是否需要先获得批准？
10. 是否会被要求签署非竞业协议

公司营收情况

1. 公司目前赚钱吗？
2. 如果没有，那距离赚钱还有多久？
3. 公司目前的发展资金来自哪里？谁在决定高层次的计划和方向？
4. 公司靠什么赚钱？
5. 是什么阻止公司赚更多的钱？

远程工作

1. 公司远程工作的员工占比多少？
2. 公司是否提供一些硬件设备，多长时间更新一次？
3. 是否可以通过公司购买额外的物品或家具？预算是什么样的？
4. 预计多长时间来一次办公室？
5. 办公室和会议室是否支持视频会议？

工作环境篇

1. 办公室布局是什么样的，是开放式/小隔间还是办公室？
2. 我的新团队是否有支持/市场等团队支持？

7. 关于内推

为了更好的帮助大家**掌握最新的招聘动态**，大彬整理了**2023校招和社招**的最新招聘信息（含内推码），目前已经更新了**400**家公司，内容持续更新中。

最近更新的部分公司名单：

公司	网申链接	内推码（内推链接）
微众银行	https://app-tc.mokahr.com/campus-recruitment/webankhr	NTAALFJ
海能达	http://zpxz.hytera.com	DSD8WSA5
小马智行	https://campus.pony.ai	NTANGNz
银星智能	http://szyxjt.zhiye.com/campus/jobs	EVKMAB
安谋科技	https://app.mokahr.com/campus-recruitment/armchina/68023#/	NTAHJRQ
智奥科技	https://app.mokahr.com/campus_apply/dolphindb/37786	DSmQ1BAH
剑心互娱	https://app.mokahr.com/campus-recruitment/jjianxin/6390#/	NTADqpd
锐明技术	http://streamax.zhiye.com/campus/jobs	EVBMHB
厦门银行	http://xmccb.zhaopin.com/	
宁德时代	https://app.mokahr.com/m/campus_apply/catlhr/73943?recommendCode=DStvZ8gt#/jobs	DStvZ8gt
禾赛科技	https://kwh0jtf778.jobs.feishu.cn/s/FSuK21r	MDB339G
中科创达	https://thundersoft.jobs.feishu.cn	N7V31XA
用友网络	https://career.yonyou.com/SU61108d5b0dcad4106ff62e80/pb/index.html#/	6ramGyvt

详情见文档：<https://docs.qq.com/sheet/DYW9ObnpobXNRTXpg>

Java核心知识

1. Java的特点

Java是一门面向对象的编程语言。面向对象和面向过程的区别参考下一个问题。

Java具有平台独立性和移植性。

- Java有一句口号：`write once, run anywhere`，一次编写、到处运行。这也是Java的魅力所在。而实现这种特性的正是Java虚拟机JVM。已编译的Java程序可以在任何带有JVM的平台上运行。你可以在windows平台编写代码，然后拿到linux上运行。只要你在编写完代码后，将代码编译成.class文件，再把class文件打成Java包，这个jar包就可以在不同的平台上运行了。

Java具有稳健性。

- Java是一个强类型语言，它允许扩展编译时检查潜在类型不匹配问题的功能。Java要求显式的方法声明，它不支持C风格的隐式声明。这些严格的要求保证编译程序能捕捉调用错误，这就导致更可靠的程序。
- 异常处理是Java中使得程序更稳健的另一个特征。异常是某种类似于错误的异常条件出现的信号。使用 `try/catch/finally` 语句，程序员可以找到出错的处理代码，这就简化了出错处理和恢复的任务。

2. Java是如何实现跨平台的？

Java是通过JVM（Java虚拟机）实现跨平台的。

JVM可以理解成一个软件，不同的平台有不同的版本。我们编写的Java代码，编译后会生成.class文件（字节码文件）。Java虚拟机就是负责将字节码文件翻译成特定平台下的机器码，通过JVM翻译成机器码之后才能运行。不同平台下编译生成的字节码是一样的，但是由JVM翻译成的机器码却不一样。

只要在不同平台上安装对应的JVM，就可以运行字节码文件，运行我们编写的Java程序。

因此，运行Java程序必须有JVM的支持，因为编译的结果不是机器码，必须要经过JVM的翻译才能执行。

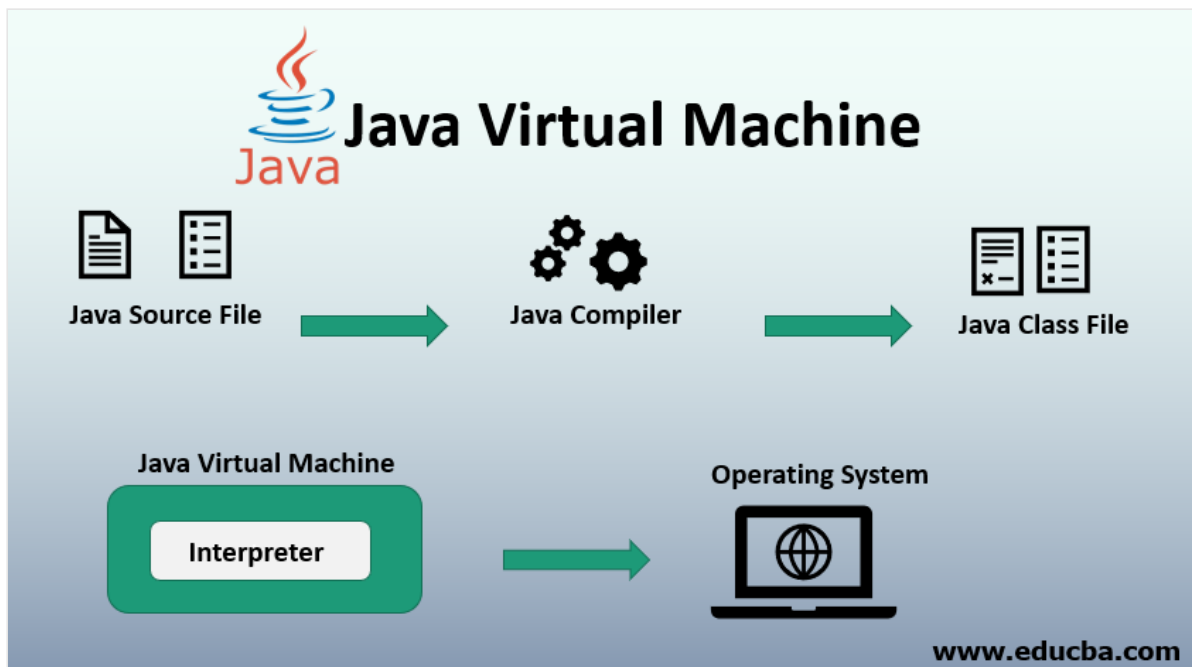
3. Java 与 C++ 的区别

- Java 是纯粹的面向对象语言，所有的对象都继承自 `java.lang.Object`，C++ 兼容 C，不但支持面向对象也支持面向过程。
- Java 通过虚拟机从而实现跨平台特性，C++ 依赖于特定的平台。
- Java 没有指针，它的引用可以理解安全指针，而 C++ 具有和 C 一样的指针。
- Java 支持自动垃圾回收，而 C++ 需要手动回收。
- Java 不支持多重继承，只能通过实现多个接口来达到相同目的，而 C++ 支持多重继承。

4. JDK/JRE/JVM三者的关系

JVM

英文名称（Java Virtual Machine），就是我们耳熟能详的Java虚拟机。Java能够跨平台运行的核心在于JVM。

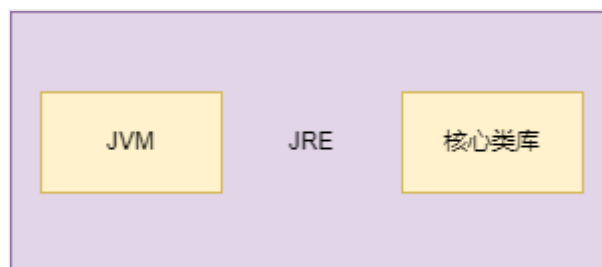


所有的java程序会首先被编译为.class的类文件，这种类文件可以在虚拟机上执行。也就是说class文件并不直接与机器的操作系统交互，而是经过虚拟机间接与操作系统交互，由虚拟机将程序解释给本地系统执行。

针对不同的系统有不同的 jvm 实现，有 Linux 版本的 jvm 实现，也有Windows 版本的 jvm 实现，但是同一段代码在编译后的字节码是一样的。这就是Java能够跨平台，实现一次编写，多处运行的原因所在。

JRE

英文名称（Java Runtime Environment），就是Java 运行时环境。我们编写的Java程序必须要在JRE才能运行。它主要包含两个部分，JVM 和 Java 核心类库。



JRE是Java的运行环境，并不是一个开发环境，所以没有包含任何开发工具，如编译器和调试器等。

如果你只是想运行Java程序，而不是开发Java程序的话，那么你只需要安装JRE即可。

JDK

英文名称（Java Development Kit），就是 Java 开发工具包

学过Java的同学，都应该安装过JDK。当我们安装完JDK之后，目录结构是这样的

software (F:) > java > jdk1.8 >				搜索"jdk1.8"
名称	修改日期	类型	大小	
bin	2020/5/14/星期四 8:44	文件夹		
db	2020/5/14/星期四 8:44	文件夹		
include	2020/5/14/星期四 8:44	文件夹		
jre	2020/5/14/星期四 8:44	文件夹		
lib	2020/5/14/星期四 8:44	文件夹		
COPYRIGHT	2016/4/1/星期五 1:16	文件	4 KB	
javafx-src.zip	2020/5/14/星期四 8:44	WinRAR ZIP 压缩...	4,973 KB	
LICENSE	2020/5/14/星期四 8:44	文件	1 KB	
README.html	2020/5/14/星期四 8:44	Chrome HTML D...	1 KB	
release	2020/5/14/星期四 8:44	文件	1 KB	
src.zip	2016/4/1/星期五 1:16	WinRAR ZIP 压缩...	20,750 KB	

可以看到，JDK目录下有个JRE，也就是JDK中已经集成了JRE，不用单独安装JRE。

另外，JDK中还有一些好用的工具，如jinfo, jps, jstack等。

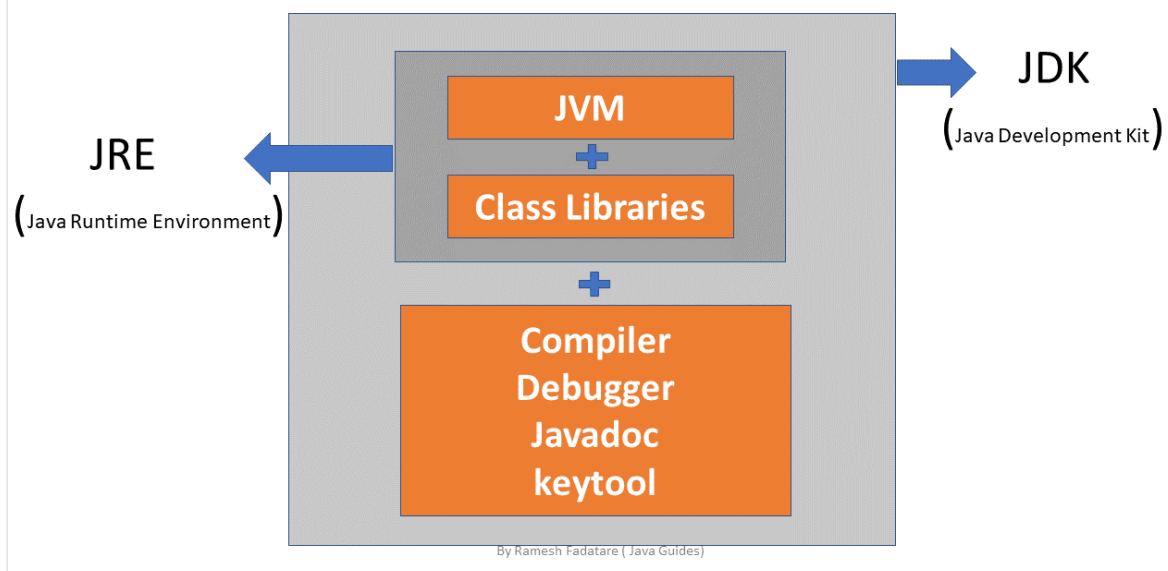
javadoc.exe	2020/5/14/星期四 8:44	应用程序	17 KB
javafxpackager.exe	2020/5/14/星期四 8:44	应用程序	131 KB
javah.exe	2020/5/14/星期四 8:44	应用程序	16 KB
javap.exe	2020/5/14/星期四 8:44	应用程序	16 KB
javapackager.exe	2020/5/14/星期四 8:44	应用程序	131 KB
java-rmi.exe	2020/5/14/星期四 8:44	应用程序	16 KB
javaw.exe	2020/5/14/星期四 8:44	应用程序	203 KB
javaws.exe	2020/5/14/星期四 8:44	应用程序	309 KB
jcmd.exe	2020/5/14/星期四 8:44	应用程序	16 KB
jconsole.exe	2020/5/14/星期四 8:44	应用程序	17 KB
jdb.exe	2020/5/14/星期四 8:44	应用程序	17 KB
jdeps.exe	2020/5/14/星期四 8:44	应用程序	16 KB
jhat.exe	2020/5/14/星期四 8:44	应用程序	16 KB
jinfo.exe	2020/5/14/星期四 8:44	应用程序	17 KB
jjs.exe	2020/5/14/星期四 8:44	应用程序	16 KB
jli.dll	2020/5/14/星期四 8:44	应用程序扩展	171 KB
jmap.exe	2020/5/14/星期四 8:44	应用程序	17 KB
jmc.exe	2020/5/14/星期四 8:44	应用程序	315 KB
jmc.ini	2020/5/14/星期四 8:44	配置设置	1 KB
jps.exe	2020/5/14/星期四 8:44	应用程序	16 KB
jrunscript.exe	2020/5/14/星期四 8:44	应用程序	17 KB
jsadebugd.exe	2020/5/14/星期四 8:44	应用程序	17 KB
jstack.exe	2020/5/14/星期四 8:44	应用程序	17 KB
jstat.exe	2020/5/14/星期四 8:44	应用程序	16 KB
jstatd.exe	2020/5/14/星期四 8:44	应用程序	16 KB
jvisualvm.exe	2020/5/14/星期四 8:44	应用程序	193 KB

最后，总结一下JDK/JRE/JVM，他们三者的关系

JRE = JVM + Java 核心类库

JDK = JRE + Java工具 + 编译器 + 调试器

Understand JDK, JRE and JVM



5. Java程序是编译执行还是解释执行？

先看看什么是编译型语言和解释型语言。

编译型语言

在程序运行之前，通过编译器将源程序编译成机器码可运行的二进制，以后执行这个程序时，就不用再进行编译了。

优点：编译器一般会有预编译的过程对代码进行优化。因为编译只做一次，运行时不需要编译，所以编译型语言的程序执行效率高，可以脱离语言环境独立运行。

缺点：编译之后如果需要修改就需要整个模块重新编译。编译的时候根据对应的运行环境生成机器码，不同的操作系统之间移植就会有问题，需要根据运行的操作系统环境编译不同的可执行文件。

总结：执行速度快、效率高；依靠编译器、跨平台性差些。

代表语言：C、C++、Pascal、Object-C以及Swift。

解释型语言

定义：解释型语言的源代码不是直接翻译成机器码，而是先翻译成中间代码，再由解释器对中间代码进行解释运行。在运行的时候才将源程序翻译成机器码，翻译一句，然后执行一句，直至结束。

优点：

1. 有良好的平台兼容性，在任何环境中都可以运行，前提是安装了解释器（如虚拟机）。
2. 灵活，修改代码的时候直接修改就可以，可以快速部署，不用停机维护。

缺点：每次运行的时候都要解释一遍，性能上不如编译型语言。

总结：解释型语言执行速度慢、效率低；依靠解释器、跨平台性好。

代表语言：JavaScript、Python、Erlang、PHP、Perl、Ruby。

对于Java这种语言，它的**源代码**会先通过javac编译成**字节码**，再通过jvm将字节码转换成**机器码**执行，即解释运行 和编译运行配合使用，所以可以称为混合型或者半编译型。

6. 面向对象和面向过程的区别？

面向对象和面向过程是一种软件开发思想。

- 面向过程就是分析出解决问题所需要的步骤，然后用函数按这些步骤实现，使用的时候依次调用就可以了。
- 面向对象是把构成问题事务分解成各个对象，分别设计这些对象，然后将他们组装成有完整功能的系统。面向过程只用函数实现，面向对象是用类实现各个功能模块。

以五子棋为例，面向过程的设计思路就是首先分析问题的步骤：

1、开始游戏，2、黑子先走，3、绘制画面，4、判断输赢，5、轮到白子，6、绘制画面，7、判断输赢，8、返回步骤2，9、输出最后结果。把上面每个步骤用分别的函数来实现，问题就解决了。

而面向对象的设计则是从另外的思路来解决问题。整个五子棋可以分为：

1. 黑白双方
2. 棋盘系统，负责绘制画面
3. 规则系统，负责判定诸如犯规、输赢等。

黑白双方负责接受用户的输入，并告知棋盘系统棋子布局发生变化，棋盘系统接收到了棋子的变化的信息就负责在屏幕上显示出这种变化，同时利用规则系统来对棋局进行判定。

7. 面向对象有哪些特性？

面向对象四大特性：封装，继承，多态，抽象

1、封装就是将类的信息隐藏在类内部，不允许外部程序直接访问，而是通过该类的方法实现对隐藏信息的操作和访问。良好的封装能够减少耦合。

2、继承是从已有的类中派生出新的类，新的类继承父类的属性和行为，并能扩展新的能力，大大增加程序的重用性和易维护性。在Java中是单继承的，也就是说一个子类只有一个父类。

3、多态是同一个行为具有多个不同表现形式的能力。在不修改程序代码的情况下改变程序运行时绑定的代码。实现多态的三要素：继承、重写、父类引用指向子类对象。

- 静态多态性：通过重载实现，相同的方法有不同的参数列表，可以根据参数的不同，做出不同的处理。
- 动态多态性：在子类中重写父类的方法。运行期间判断所引用对象的实际类型，根据其实际类型调用相应的方法。

4、抽象。把客观事物用代码抽象出来。

8. 面向对象编程的六大原则？

- **对象单一职责**：我们设计创建的对象，必须职责明确，比如商品类，里面相关的属性和方法都必须跟商品相关，不能出现订单等不相关的内容。这里的类可以是模块、类库、程序集，而不单单指类。
- **里式替换原则**：子类能够完全替代父类，反之则不行。通常用于实现接口时运用。因为子类能够完全替代基（父）类，那么这样父类就拥有很多子类，在后续的程序扩展中就很容易进行扩展，程序完全不需要进行修改即可进行扩展。比如IA的实现为A，因为项目需求变更，现在需要新的实现，直接在容器注入处更换接口即可。
- **迪米特法则**，也叫最小原则，或者说最小耦合。通常在设计程序或开发程序的时候，尽量要高内聚，低耦合。当两个类进行交互的时候，会产生依赖。而迪米特法则就是建议这种依赖越少越好。就像构造函数注入父类对象时一样，当需要依赖某个对象时，并不在意其内部是怎么实现的，而是在容器中注入相应的实现，既符合里式替换原则，又起到了解耦的作用。
- **开闭原则**：开放扩展，封闭修改。当项目需求发生变更时，要尽可能的不去对原有的代码进行修改，而在原有的基础上进行扩展。
- **依赖倒置原则**：高层模块不应该直接依赖于底层模块的具体实现，而应该依赖于底层的抽象。接口和抽象类不应该依赖于实现类，而实现类依赖接口或抽象类。

- **接口隔离原则**：一个对象和另外一个对象交互的过程中，依赖的内容最小。也就是说在接口设计的时候，在遵循对象单一职责的情况下，尽量减少接口的内容。

简洁版：

- 单一职责：对象设计要求独立，不能设计万能对象。
- 开闭原则：对象修改最小化。
- 里式替换：程序扩展中抽象被具体可以替换（接口、父类、可以被实现类对象、子类替换对象）
- 迪米特：高内聚，低耦合。尽量不要依赖细节。
- 依赖倒置：面向抽象编程。也就是参数传递，或者返回值，可以使用父类类型或者接口类型。从广义上讲：基于接口编程，提前设计好接口框架。
- 接口隔离：接口设计大小要适中。过大导致污染，过小，导致调用麻烦。

9. 数组到底是不是对象？

先说说对象的概念。对象是根据某个类创建出来的一个实例，表示某类事物中一个具体的个体。

对象具有各种属性，并且具有一些特定的行为。站在计算机的角度，对象就是内存中的一个内存块，在这个内存块封装了一些数据，也就是类中定义各个属性。

所以，对象是用来封装数据的。

java中的数组具有java中其他对象的一些基本特点。比如封装了一些数据，可以访问属性，也可以调用方法。

因此，可以说，数组是对象。

也可以通过代码验证数组是对象的事实。比如以下的代码，输出结果为java.lang.Object。

```
Class clz = int[].class;
System.out.println(clz.getSuperclass().getName());
```

由此，可以看出，数组类的父类就是Object类，那么可以推断出数组就是对象。

10. Java的基本数据类型有哪些？

- byte, 8bit
- char, 16bit
- short, 16bit
- int, 32bit
- float, 32bit
- long, 64bit
- double, 64bit
- boolean, 只有两个值：true、false，可以使用用 1 bit 来存储

简单类型	boolean	byte	char	short	Int	long	float	double
二进制位数	1	8	16	16	32	64	32	64
包装类	Boolean	Byte	Character	Short	Integer	Long	Float	Double

在Java规范中，没有明确指出boolean的大小。在《Java虚拟机规范》给出了单个boolean占4个字节，和boolean数组1个字节的定义，具体 **还要看虚拟机实现是否按照规范来**，因此boolean占用1个字节或者4个字节都是有可能的。

11. 为什么不能用浮点型表示金额？

由于计算机中保存的小数其实是十进制的小数的近似值，并不是准确值，所以，千万不要在代码中使用浮点数来表示金额等重要的指标。

建议使用BigDecimal或者Long来表示金额。

12. 什么是值传递和引用传递？

- 值传递是对基本型变量而言的，传递的是该变量的一个副本，改变副本不影响原变量。
- 引用传递一般是对于对象型变量而言的，传递的是该对象地址的一个副本，并不是原对象本身，两者指向同一片内存空间。所以对引用对象进行操作会同时改变原对象。

java中不存在引用传递，只有值传递。即不存在变量a指向变量b，变量b指向对象的这种情况。

13. 了解Java的包装类型吗？为什么需要包装类？

Java 是一种面向对象语言，很多地方都需要使用对象而不是基本数据类型。比如，在集合类中，我们是无法将 int、double 等类型放进去的。因为集合的容器要求元素是 Object 类型。

为了让基本类型也具有对象的特征，就出现了包装类型。相当于将基本类型包装起来，使得它具有了对象的性质，并且为其添加了属性和方法，丰富了基本类型的操作。

14. 自动装箱和拆箱

Java中基础数据类型与它们对应的包装类见下表：

原始类型	包装类型
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

装箱：将基础类型转化为包装类型。

拆箱：将包装类型转化为基础类型。

当基础类型与它们的包装类有如下几种情况时，编译器会**自动**帮我们进行装箱或拆箱：

- 赋值操作（装箱或拆箱）
- 进行加减乘除混合运算（拆箱）
- 进行>,<,<==比较运算（拆箱）

- 调用equals进行比较（装箱）
- ArrayList、HashMap等集合类添加基础类型数据时（装箱）

示例代码：

```
Integer x = 1; // 装箱 调用 Integer.valueOf(1)
int y = x; // 拆箱 调用了 x.intValue()
```

下面看一道常见的面试题：

```
Integer a = 100;
Integer b = 100;
System.out.println(a == b);

Integer c = 200;
Integer d = 200;
System.out.println(c == d);
```

输出：

```
true
false
```

为什么第三个输出是false？看看 Integer 类的源码就知道啦。

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

`Integer c = 200;` 会调用 `Integer.valueOf(200)`。而从Integer的valueOf()源码可以看到，这里的实现并不是简单的new Integer，而是用IntegerCache做一个cache。

```
private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static final Integer cache[];

    static {
        // high value may be configured by property
        int h = 127;
        String integerCacheHighPropValue =
            sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
        if (integerCacheHighPropValue != null) {
            try {
                int i = parseInt(integerCacheHighPropValue);
                i = Math.max(i, 127);
                // Maximum array size is Integer.MAX_VALUE
                h = Math.min(i, Integer.MAX_VALUE - (-low) - 1);
            } catch (NumberFormatException nfe) {
                // If the property cannot be parsed into an int, ignore it.
            }
        }
        high = h;
    }
}
```

```
}  
...  
}
```

这是IntegerCache静态代码块中的一段，默认Integer cache 的下限是-128，上限默认127。当赋值100给Integer时，刚好在这个范围内，所以从cache中取对应的Integer并返回，所以a和b返回的是同一个对象，所以 `==` 比较是相等的，当赋值200给Integer时，不在cache 的范围内，所以会new Integer并返回，当然 `==` 比较的结果是不相等的。

15. String 为什么不可变?

先看看什么是不可变的对象。

如果一个对象，在它创建完成之后，不能再改变它的状态，那么这个对象就是不可变的。不能改变状态的意思是，不能改变对象内的成员变量，包括基本数据类型的值不能改变，引用类型的变量不能指向其他的对象，引用类型指向的对象的状态也不能改变。

接着来看Java8 String类的源码：

```
public final class String  
    implements java.io.Serializable, Comparable<String>, CharSequence {  
    /** The value is used for character storage. */  
    private final char value[];  
  
    /** Cache the hash code for the string */  
    private int hash; // Default to 0  
}
```

从源码可以看出，String对象其实在内部就是一个个字符，存储在这个value数组里面的。

value数组用final修饰，final 修饰的变量，值不能被修改。因此value不可以指向其他对象。

String类内部所有的字段都是私有的，也就是被private修饰。而且String没有对外提供修改内部状态的方法，因此value数组不能改变。

所以，String是不可变的。

那为什么String要设计成不可变的?

主要有以下几点原因：

1. **线程安全**。同一个字符串实例可以被多个线程共享，因为字符串不可变，本身就是线程安全的。
2. **支持hash映射和缓存**。因为String的hash值经常会使用到，比如作为 Map 的键，不可变的特性使得 hash 值也不会变，不需要重新计算。
3. **出于安全考虑**。网络地址URL、文件路径path、密码通常情况下都是以String类型保存，假若String不是固定不变的，将会引起各种安全隐患。比如将密码用String的类型保存，那么它将一直留在内存中，直到垃圾收集器把它清除。假如String类不是固定不变的，那么这个密码可能会被改变，导致出现安全隐患。
4. **字符串常量池优化**。String对象创建之后，会缓存到字符串常量池中，下次需要创建同样的对象时，可以直接返回缓存的引用。

既然我们的String是不可变的，它内部还有很多substring，replace，replaceAll这些操作的方法。这些方法好像会改变String对象？怎么解释呢？

其实不是的，我们每次调用replace等方法，其实会在堆内存中创建了一个新的对象。然后其value数组引用指向不同的对象。

16. 为何JDK9要将String的底层实现由char[]改成byte[]?

主要是为了节约String占用的内存。

在大部分Java程序的堆内存中，String占用的空间最大，并且绝大多数String只有Latin-1字符，这些Latin-1字符只需要1个字节就够了。

而在JDK9之前，JVM因为String使用char数组存储，每个char占2个字节，所以即使字符串只需要1字节，它也要按照2字节进行分配，浪费了一半的内存空间。

到了JDK9之后，对于每个字符串，会先判断它是不是只有Latin-1字符，如果是，就按照1字节的规格进行分配内存，如果不是，就按照2字节的规格进行分配，这样便提高了内存使用率，同时GC次数也会减少，提升效率。

不过Latin-1编码集支持的字符有限，比如不支持中文字符，因此对于中文字符串，用的是UTF16编码（两个字节），所以用byte[]和char[]实现没什么区别。

17. String, StringBuffer 和 StringBuilder区别

1. 可变性

- String 不可变
- StringBuffer 和 StringBuilder 可变

2. 线程安全

- String 不可变，因此是线程安全的
- StringBuilder 不是线程安全的
- StringBuffer 是线程安全的，内部使用 synchronized 进行同步

18. 什么是StringJoiner?

StringJoiner是Java 8新增的一个API，它基于StringBuilder实现，用于实现对字符串之间通过分隔符拼接的场景。

StringJoiner有两个构造方法，第一个构造要求依次传入分隔符、前缀和后缀。第二个构造则只要求传入分隔符即可（前缀和后缀默认为空字符串）。

```
StringJoiner(CharSequence delimiter, CharSequence prefix, CharSequence suffix)
StringJoiner(CharSequence delimiter)
```

有些字符串拼接场景，使用StringBuffer或StringBuilder则显得比较繁琐。

比如下面的例子：

```
List<Integer> values = Arrays.asList(1, 3, 5);
StringBuilder sb = new StringBuilder("(");

for (int i = 0; i < values.size(); i++) {
    sb.append(values.get(i));
    if (i != values.size() - 1) {
        sb.append(",");
    }
}

sb.append(")");
```

而通过StringJoiner来实现拼接List的各个元素，代码看起来更加简洁。

```
List<Integer> values = Arrays.asList(1, 3, 5);
StringJoiner sj = new StringJoiner(", ", "(", ")");

for (Integer value : values) {
    sj.add(value.toString());
}
```

另外，像平时经常使用的Collectors.joining(",")，底层就是通过StringJoiner实现的。

源码如下：

```
public static Collector<CharSequence, ?, String> joining(
    CharSequence delimiter, CharSequence prefix, CharSequence suffix) {
    return new CollectorImpl<>() {
        () -> new StringJoiner(delimiter, prefix, suffix),
        StringJoiner::add, StringJoiner::merge,
        StringJoiner::toString, CH_NOID);
    }
```

19. String 类的常用方法有哪些？

- indexOf(): 返回指定字符的索引。
- charAt(): 返回指定索引处的字符。
- replace(): 字符串替换。
- trim(): 去除字符串两端空白。
- split(): 分割字符串，返回一个分割后的字符串数组。
- getBytes(): 返回字符串的 byte 类型数组。
- length(): 返回字符串长度。
- toLowerCase(): 将字符串转成小写字母。
- toUpperCase(): 将字符串转成大写字母。
- substring(): 截取字符串。
- equals(): 字符串比较。

20. new String("dabin")会创建几个对象？

使用这种方式会创建两个字符串对象（前提是字符串常量池中没有 "dabin" 这个字符串对象）。

- "dabin" 属于字符串字面量，因此编译时期会在字符串常量池中创建一个字符串对象，指向这个 "dabin" 字符串字面量；
- 使用 new 的方式会在堆中创建一个字符串对象。

21. 什么是字符串常量池？

字符串常量池（String Pool）保存着所有字符串字面量，这些字面量在编译时期就确定。字符串常量池位于堆内存中，专门用来存储字符串常量。在创建字符串时，JVM首先会检查字符串常量池，如果该字符串已经存在池中，则返回其引用，如果不存在，则创建此字符串并放入池中，并返回其引用。

22. String最大长度是多少？

String类提供了一个length方法，返回值为int类型，而int的取值上限为 $2^{31} - 1$ 。

所以理论上String的最大长度为 $2^{31} - 1$ 。

达到这个长度的话需要多大的内存吗？

String内部是使用一个char数组来维护字符序列的，一个char占用两个字节。如果说String最大长度是 $2^{31} - 1$ 的话，那么最大的字符串占用内存空间约等于4GB。

也就是说，我们需要有大于4GB的JVM运行内存才行。

那String一般都存储在JVM的哪块区域呢？

字符串在JVM中的存储分两种情况，一种是String对象，存储在JVM的堆栈中。一种是字符串常量，存储在常量池里面。

什么情况下字符串会存储在常量池呢？

当通过字面量进行字符串声明时，比如String s = "程序新大彬";，这个字符串在编译之后会以常量的形式进入到常量池。

那常量池中的字符串最大长度是 $2^{31} - 1$ 吗？

不是的，常量池对String的长度是有另外限制的。。Java中的UTF-8编码的Unicode字符串在常量池中以CONSTANT_Utf8类型表示。

```
CONSTANT_Utf8_info {  
    u1 tag;  
    u2 length;  
    u1 bytes[length];  
}
```

length在这里就是代表字符串的长度，length的类型是u2，u2是无符号的16位整数，也就是说最大长度可以做到 $2^{16} - 1$ 即 65535。

不过javac编译器做了限制，需要length < 65535。所以字符串常量在常量池中的最大长度是65535 - 1 = 65534。

最后总结一下：

String在不同的状态下，具有不同的长度限制。

- 字符串常量长度不能超过65534
- 堆内字符串的长度不超过 $2^{31} - 1$

23. Object常用方法有哪些？

Java面试经常会出现的一道题目，Object的常用方法。下面给大家整理一下。

Object常用方法有：`toString()`、`equals()`、`hashCode()`、`clone()`等。

toString

默认输出对象地址。

```
public class Person {  
    private int age;  
    private String name;  
  
    public Person(int age, String name) {  
        this.age = age;  
        this.name = name;  
    }  
}
```

```

public static void main(String[] args) {
    System.out.println(new Person(18, "程序员大彬").toString());
}
//output
//me.tyson.java.core.Person@4554617c
}

```

可以重写toString方法，按照重写逻辑输出对象值。

```

public class Person {
    private int age;
    private String name;

    public Person(int age, String name) {
        this.age = age;
        this.name = name;
    }

    @Override
    public String toString() {
        return name + ":" + age;
    }

    public static void main(String[] args) {
        System.out.println(new Person(18, "程序员大彬").toString());
    }
    //output
    //程序员大彬:18
}

```

equals

默认比较两个引用变量是否指向同一个对象（内存地址）。

```

public class Person {
    private int age;
    private String name;

    public Person(int age, String name) {
        this.age = age;
        this.name = name;
    }

    public static void main(String[] args) {
        String name = "程序员大彬";
        Person p1 = new Person(18, name);
        Person p2 = new Person(18, name);

        System.out.println(p1.equals(p2));
    }
    //output
    //false
}

```

可以重写equals方法，按照age和name是否相等来判断：

```

public class Person {
    private int age;
    private String name;

    public Person(int age, String name) {
        this.age = age;
        this.name = name;
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof Person) {
            Person p = (Person) o;
            return age == p.age && name.equals(p.name);
        }
        return false;
    }

    public static void main(String[] args) {
        String name = "程序员大彬";
        Person p1 = new Person(18, name);
        Person p2 = new Person(18, name);

        System.out.println(p1.equals(p2));
    }
    //output
    //true
}

```

hashCode

将与对象相关的信息映射成一个哈希值，默认的实现hashCode值是根据内存地址换算出来。

```

public class Cat {
    public static void main(String[] args) {
        System.out.println(new Cat().hashCode());
    }
    //out
    //1349277854
}

```

clone

Java赋值是复制对象引用，如果我们想要得到一个对象的副本，使用赋值操作是无法达到目的的。

Object对象有个clone()方法，实现了对

象中各个属性的复制，但它的可见范围是protected的。

```
protected native Object clone() throws CloneNotSupportedException;
```

所以实体类使用克隆的前提是：

- 实现Cloneable接口，这是一个标记接口，自身没有方法，这应该是一种约定。调用clone方法时，会判断有没有实现Cloneable接口，没有实现Cloneable的话会抛异常CloneNotSupportedException。
- 覆盖clone()方法，可见性提升为public。

```

public class Cat implements Cloneable {
    private String name;

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public static void main(String[] args) throws CloneNotSupportedException {
        Cat c = new Cat();
        c.name = "程序员大彬";
        Cat cloneCat = (Cat) c.clone();
        c.name = "大彬";
        System.out.println(cloneCat.name);
    }
    //output
    //程序员大彬
}

```

getClass

返回此 Object 的运行时常量，常用于java反射机制。

```

public class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    public static void main(String[] args) {
        Person p = new Person("程序员大彬");
        Class clz = p.getClass();
        System.out.println(clz);
        //获取类名
        System.out.println(clz.getName());
    }
    /**
     * class com.tyson.basic.Person
     * com.tyson.basic.Person
     */
}

```

wait

当前线程调用对象的wait()方法之后，当前线程会释放对象锁，进入等待状态。等待其他线程调用此对象的notify()/notifyAll()唤醒或者等待超时时间wait(long timeout)自动唤醒。线程需要获取obj对象锁之后才能调用 obj.wait()。

notify

obj.notify()唤醒在此对象上等待的单个线程，选择是任意性的。notifyAll()唤醒在此对象上等待的所有线程。

24. 讲讲深拷贝和浅拷贝？

浅拷贝：拷贝对象和原始对象的引用类型引用同一个对象。

以下例子，Cat对象里面有个Person对象，调用clone之后，克隆对象和原对象的Person引用的是同一个对象，这就是浅拷贝。

```
public class Cat implements Cloneable {
    private String name;
    private Person owner;

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public static void main(String[] args) throws CloneNotSupportedException {
        Cat c = new Cat();
        Person p = new Person(18, "程序员大彬");
        c.owner = p;

        Cat cloneCat = (Cat) c.clone();
        p.setName("大彬");
        System.out.println(cloneCat.owner.getName());
    }
    //output
    //大彬
}
```

深拷贝：拷贝对象和原始对象的引用类型引用不同的对象。

以下例子，在clone函数中不仅调用了super.clone，而且调用Person对象的clone方法（Person也要实现Cloneable接口并重写clone方法），从而实现了深拷贝。可以看到，拷贝对象的值不会受到原对象的影响。

```
public class Cat implements Cloneable {
    private String name;
    private Person owner;

    @Override
    protected Object clone() throws CloneNotSupportedException {
        Cat c = null;
        c = (Cat) super.clone();
        c.owner = (Person) owner.clone(); //拷贝Person对象
        return c;
    }

    public static void main(String[] args) throws CloneNotSupportedException {
        Cat c = new Cat();
        Person p = new Person(18, "程序员大彬");
        c.owner = p;

        Cat cloneCat = (Cat) c.clone();
        p.setName("大彬");
        System.out.println(cloneCat.owner.getName());
    }
    //output
    //程序员大彬
}
```

25. 两个对象的hashCode()相同，则 equals()是否也一定为 true?

equals与hashCode的关系：

1. 如果两个对象调用equals比较返回true，那么它们的hashCode值一定要相同；
2. 如果两个对象的hashCode相同，它们并不一定相同。

hashCode方法主要是用来**提升对象比较的效率**，先进行hashCode()的比较，如果不相同，那就不必在进行equals的比较，这样就大大减少了equals比较的次数，当比较对象的数量很大的时候能提升效率。

之所以重写 `equals()` 要重写 `hashCode()`，是为了保证 `equals()` 方法返回true的情况下hashCode值也要一致，如果重写了 `equals()` 没有重写 `hashCode()`，就会出现两个对象相等但 `hashCode()` 不相等的情况。这样，当用其中的一个对象作为键保存到HashMap、HashTable或HashSet中，再以另一个对象作为键值去查找他们的时候，则会查找不到。

26. Java创建对象有几种方式?

Java创建对象有以下几种方式：

- 用new语句创建对象。
- 使用反射，使用Class.newInstance()创建对象。
- 调用对象的clone()方法。
- 运用反序列化手段，调用java.io.ObjectInputStream对象的readObject()方法。

27. 说说类实例化的顺序

Java中类实例化顺序：

1. 静态属性，静态代码块。
2. 普通属性，普通代码块。
3. 构造方法。

```
public class Lifecycle {  
    // 静态属性  
    private static String staticField = getStaticField();  
  
    // 静态代码块  
    static {  
        System.out.println(staticField);  
        System.out.println("静态代码块初始化");  
    }  
  
    // 普通属性  
    private String field = getField();  
  
    // 普通代码块  
    {  
        System.out.println(field);  
        System.out.println("普通代码块初始化");  
    }  
  
    // 构造方法  
    public Lifecycle() {  
        System.out.println("构造方法初始化");  
    }  
}
```

```

// 静态方法
public static String getStaticField() {
    String staticFiled = "静态属性初始化";
    return staticFiled;
}

// 普通方法
public String getField() {
    String filed = "普通属性初始化";
    return filed;
}

public static void main(String[] argc) {
    new Lifecycle();
}

/**
 *      静态属性初始化
 *      静态代码块初始化
 *      普通属性初始化
 *      普通代码块初始化
 *      构造方法初始化
 */
}

```

28. equals和==有什么区别？

- 对于基本数据类型，==比较的是他们的值。基本数据类型没有equal方法；
- 对于复合数据类型，==比较的是它们的存放地址(是否是同一个对象)。equals() 默认比较地址值，重写的话按照重写逻辑去比较。

29. 常见的关键字有哪些？

static

static可以用来修饰类的成员方法、类的成员变量。

static变量也称作**静态变量**，静态变量和非静态变量的区别是：静态变量被所有的对象所共享，在内存中只有一个副本，它当且仅当在类初次加载时会被初始化。而非静态变量是对象所拥有的，在创建对象的时候被初始化，存在多个副本，各个对象拥有的副本互不影响。

以下例子，age为非静态变量，则p1打印结果是：Name:zhangsan, Age:10；若age使用static修饰，则p1打印结果是：Name:zhangsan, Age:12，因为static变量在内存只有一个副本。

```

public class Person {
    String name;
    int age;

    public String toString() {
        return "Name:" + name + ", Age:" + age;
    }

    public static void main(String[] args) {
        Person p1 = new Person();
        p1.name = "zhangsan";
        p1.age = 10;
        Person p2 = new Person();
    }
}

```

```

        p2.name = "lisi";
        p2.age = 12;
        System.out.println(p1);
        System.out.println(p2);
    }
    /**Output
     * Name:zhangsan, Age:10
     * Name:lisi, Age:12
     * ///~
     */
}

```

static方法一般称作**静态方法**。静态方法不依赖于任何对象就可以进行访问，通过类名即可调用静态方法。

```

public class Utils {
    public static void print(String s) {
        System.out.println("hello world: " + s);
    }

    public static void main(String[] args) {
        Utils.print("程序员大彬");
    }
}

```

静态代码块只会在类加载的时候执行一次。以下例子，startDate和endDate在类加载的时候进行赋值。

```

class Person {
    private Date birthDate;
    private static Date startDate, endDate;
    static{
        startDate = Date.valueOf("2008");
        endDate = Date.valueOf("2021");
    }

    public Person(Date birthDate) {
        this.birthDate = birthDate;
    }
}

```

静态内部类

在**静态方法**里，使用非静态内部类依赖于外部类的实例，也就是说需要先创建外部类实例，才能用这个实例去创建非静态内部类。而静态内部类不需要。

```

public class OuterClass {
    class InnerClass {
    }
    static class StaticInnerClass {
    }
    public static void main(String[] args) {
        // 在静态方法里，不能直接使用OuterClass.this去创建InnerClass的实例
        // 需要先创建OuterClass的实例o，然后通过o创建InnerClass的实例
        // InnerClass innerClass = new InnerClass();
        OuterClass outerClass = new OuterClass();
        InnerClass innerClass = outerClass.new InnerClass();
        StaticInnerClass staticInnerClass = new StaticInnerClass();
    }
}

```

```

        outerClass.test();
    }

    public void nonStaticMethod() {
        InnerClass innerClass = new InnerClass();
        System.out.println("nonStaticMethod...");
    }
}

```

final

1. **基本数据类型**用final修饰，则不能修改，是常量；**对象引用**用final修饰，则引用只能指向该对象，不能指向别的对象，但是对象本身可以修改。
2. final修饰的方法不能被子类重写
3. final修饰的类不能被继承。

this

this.属性名称 指访问类中的成员变量，可以用来区分成员变量和局部变量。如下代码所示，**this.name** 访问类Person当前实例的变量。

```

/**
 * @description:
 * @author: 程序员大彬
 * @time: 2021-08-17 00:29
 */
public class Person {
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

```

this.方法名称 用来访问本类的方法。以下代码中，**this.born()** 调用类 Person 的当前实例的方法。

```

/**
 * @description:
 * @author: 程序员大彬
 * @time: 2021-08-17 00:29
 */
public class Person {
    String name;
    int age;

    public Person(String name, int age) {
        this.born();
        this.name = name;
        this.age = age;
    }

    void born() {
    }
}

```

super

super 关键字用于在子类中访问父类的变量和方法。

```
class A {
    protected String name = "大彬";

    public void getName() {
        System.out.println("父类:" + name);
    }
}

public class B extends A {
    @Override
    public void getName() {
        System.out.println(super.name);
        super.getName();
    }

    public static void main(String[] args) {
        B b = new B();
        b.getName();
    }
    /**
     * 大彬
     * 父类:大彬
     */
}
```

在子类B中，我们重写了父类的 `getName()` 方法，如果在重写的 `getName()` 方法中我们要调用父类的相同方法，必须要通过super关键字显式指出。

30. final, finally, finalize 的区别

- final 用于修饰属性、方法和类, 分别表示属性不能被重新赋值, 方法不可被覆盖, 类不可被继承。
- finally 是异常处理语句结构的一部分, 一般以 `try-catch-finally` 出现, `finally` 代码块表示总是被执行。
- finalize 是Object类的一个方法, 该方法一般由垃圾回收器来调用, 当我们调用 `System.gc()` 方法的时候, 由垃圾回收器调用 `finalize()` 方法, 回收垃圾, JVM并不保证此方法总被调用。

31. final关键字的作用?

- final 修饰的类不能被继承。
- final 修饰的方法不能被重写。
- final 修饰的变量叫常量, 常量必须初始化, 初始化之后值就不能被修改。

32. 方法重载和重写的区别?

同个类中的多个方法可以有相同的方法名称, 但是有不同的参数列表, 这就称为方法重载。参数列表又叫参数签名, 包括参数的类型、参数的个数、参数的顺序, 只要有一个不同就叫做参数列表不同。

重载是面向对象的一个基本特性。

```
public class OverrideTest {
    void setPerson() { }

    void setPerson(String name) {
        //set name
    }

    void setPerson(String name, int age) {
        //set name and age
    }
}
```

方法的重写描述的是父类和子类之间的。当父类的功能无法满足子类的需求，可以在子类对方法进行重写。方法重写时，方法名与形参列表必须一致。

如下代码，Person为父类，Student为子类，在Student中重写了dailyTask方法。

```
public class Person {
    private String name;

    public void dailyTask() {
        System.out.println("work eat sleep");
    }
}

public class Student extends Person {
    @Override
    public void dailyTask() {
        System.out.println("study eat sleep");
    }
}
```

33. 接口与抽象类区别？

1、语法层面上的区别

- 抽象类可以有方法实现，而接口的方法中只能是抽象方法（Java 8 之后接口方法可以有默认实现）；
- 抽象类中的成员变量可以是各种类型的，接口中的成员变量只能是public static final类型；
- 接口中不能含有静态代码块以及静态方法，而抽象类可以有静态代码块和静态方法（Java 8之后接口可以有静态方法）；
- 一个类只能继承一个抽象类，而一个类却可以实现多个接口。

2、设计层面上的区别

- 抽象层次不同。抽象类是对整个类整体进行抽象，包括属性、行为，但是接口只是对类行为进行抽象。继承抽象类是一种"是不是"的关系，而接口实现则是"有没有"的关系。如果一个类继承了某个抽象类，则子类必定是抽象类的种类，而接口实现则是具备不具备的关系，比如鸟是否能飞。
- 继承抽象类的是具有相似特点的类，而实现接口的却可以不同的类。

门和警报的例子：

```
class AlarmDoor extends Door implements Alarm {
    //code
}

class BMWCar extends Car implements Alarm {
    //code
}
```

34. 常见的Exception有哪些？

常见的RuntimeException：

1. `ClassCastException` //类型转换异常
2. `IndexOutOfBoundsException` //数组越界异常
3. `NullPointerException` //空指针
4. `ArrayStoreException` //数组存储异常
5. `NumberFormatException` //数字格式化异常
6. `ArithmeticException` //数学运算异常

checked Exception：

1. `NoSuchFieldException` //反射异常，没有对应的字段
2. `ClassNotFoundException` //类没有找到异常
3. `IllegalAccessException` //安全权限异常，可能是反射时调用了private方法

35. Error和Exception的区别？

Error：JVM 无法解决的严重问题，如栈溢出 `StackOverflowError`、内存溢出 `OOM` 等。程序无法处理的错误。

Exception：其它因编程错误或偶然的外在因素导致的一般性问题。可以在代码中进行处理。如：空指针异常、数组下标越界等。

36. 运行时异常和非运行时异常（checked）的区别？

`unchecked exception` 包括 `RuntimeException` 和 `Error` 类，其他所有异常称为检查（checked）异常。

1. `RuntimeException` 由程序错误导致，应该修正程序避免这类异常发生。
2. `checked Exception` 由具体的环境（读取的文件不存在或文件为空或sql异常）导致的异常。必须进行处理，不然编译不通过，可以catch或者throws。

37. throw和throws的区别？

- **throw**：用于抛出一个具体的异常对象。
- **throws**：用在方法签名中，用于声明该方法可能抛出的异常。子类方法抛出的异常范围更加小，或者根本不抛异常。

38. 通过故事讲清楚NIO

下面通过一个例子来讲解下。

假设某银行只有10个职员。该银行的业务流程分为以下4个步骤：

- 1) 顾客填申请表（5分钟）；

- 2) 职员审核 (1分钟) ;
- 3) 职员叫保安去金库取钱 (3分钟) ;
- 4) 职员打印票据, 并将钱和票据返回给顾客 (1分钟) 。

下面我们看看银行不同的工作方式对其工作效率到底有何影响。

首先是BIO方式。

每来一个顾客, 马上由一位职员来接待处理, 并且这个职员需要负责以上4个完整流程。当超过10个顾客时, 剩余的顾客需要排队等候。

一个职员处理一个顾客需要10分钟 (5+1+3+1) 时间。一个小时 (60分钟) 能处理6个顾客, 一共10个职员, 那就是只能处理60个顾客。

可以看到银行职员的工作状态并不饱和, 比如在第1步, 其实是处于等待中。

这种工作其实就是BIO, 每次来一个请求 (顾客), 就分配到线程池中由一个线程 (职员) 处理, 如果超出了线程池的最大上限 (10个), 就扔到队列等待。

那么如何提高银行的吞吐量呢?

思路就是: **分而治之**, 将任务拆分开来, 由专门的人负责专门的任务。

具体来讲, 银行专门指派一名职员A, A的工作就是每当有顾客到银行, 他就递上表格让顾客填写。每当有顾客填好表后, A就将其随机指派给剩余的9名职员完成后续步骤。

这种方式下, 假设顾客非常多, 职员A的工作处于饱和中, 他不断的将填好表的顾客带到柜台处理。

柜台一个职员5分钟能处理完一个顾客, 一个小时9名职员能处理: $9 * (60/5) = 108$ 。

可见工作方式的转变能带来效率的极大提升。

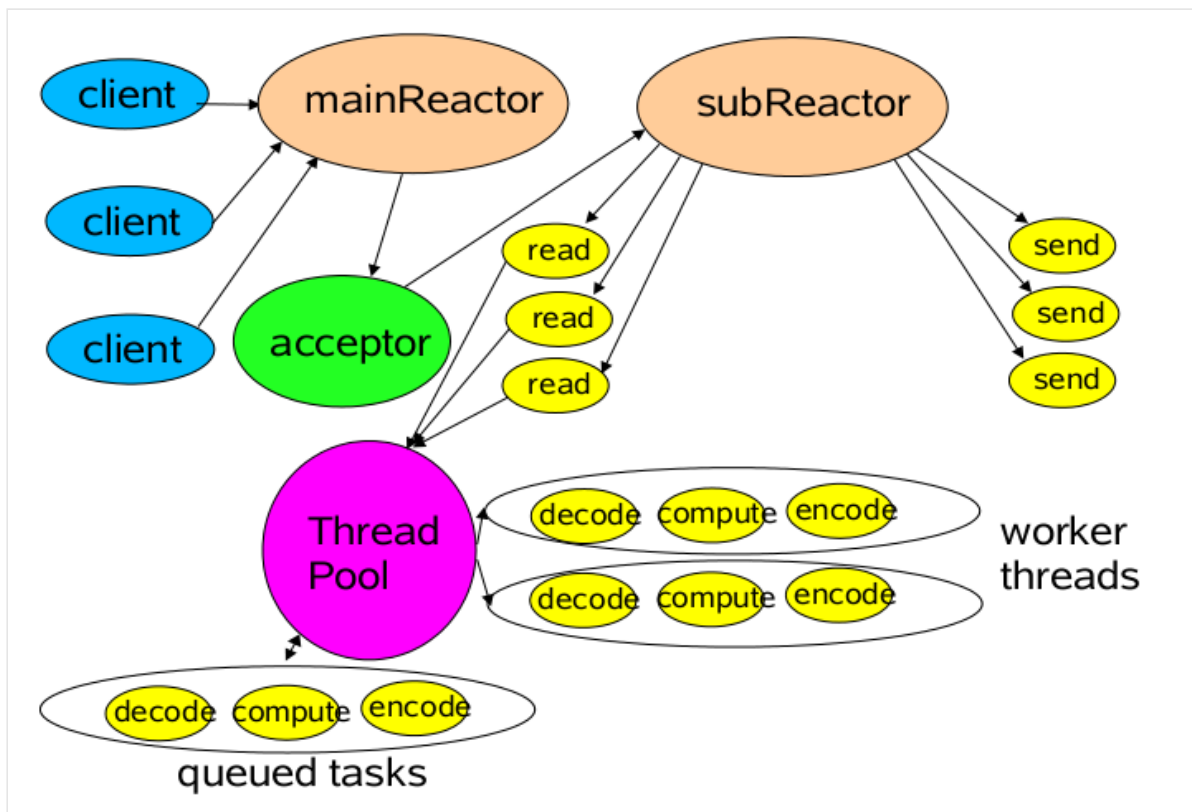
这种工作方式其实就NIO的思路。

下图是非常经典的NIO说明图, **mainReactor** 线程负责监听server socket, 接收新连接, 并将建立的socket分派给 **subReactor**

subReactor 可以是一个线程, 也可以是线程池, 负责多路分离已连接的socket, 读写网络数据。这里的读写网络数据可类比顾客填表这一耗时动作, 对具体的业务处理功能, 其扔给worker线程池完成

可以看到典型NIO有三类线程, 分别是 **mainReactor** 线程、**subReactor** 线程、**work** 线程。

不同的线程干专业的事情, 最终每个线程都没空着, 系统的吞吐量自然就上去了。



那这个流程还有没有什么可以提高的地方呢？

可以看到，在这个业务流程里边第3个步骤，职员叫保安去金库取钱（3分钟）。这3分钟柜台职员是在等待中度过的，可以把这3分钟利用起来。

还是分而治之的思路，指派1个职员B来专门负责第3步骤。

每当柜台员工完成第2步时，就通知职员B来负责与保安沟通取钱。这时候柜台员工可以继续处理下一个顾客。

当职员B拿到钱之后，通知顾客钱已经到柜台了，让顾客重新排队处理，当柜台职员再次服务该顾客时，发现该顾客前3步已经完成，直接执行第4步即可。

在当今web服务中，经常需要通过RPC或者Http等方式调用第三方服务，这里对应的就是第3步，如果这一步耗时较长，通过异步方式将能极大降低资源使用率。

NIO+异步的方式能让少量的线程做大量的事情。这适用于很多应用场景，比如代理服务、api服务、长连接服务等等。这些应用如果用同步方式将耗费大量机器资源。

不过虽然NIO+异步能提高系统吞吐量，但其并不能让一个请求的等待时间下降，相反可能会增加等待时间。

最后，NIO基本思想总结起来就是：**分而治之，将任务拆分开来，由专门的人负责专门的任务**

39. BIO/NIO/AIO区别的区别？

同步阻塞IO：用户进程发起一个IO操作以后，必须等待IO操作的真正完成后，才能继续运行。

同步非阻塞IO：客户端与服务器通过Channel连接，采用多路复用器轮询注册的 **Channel**。提高吞吐量和可靠性。用户进程发起一个IO操作以后，可做其它事情，但用户进程需要轮询IO操作是否完成，这样造成不必要的CPU资源浪费。

异步非阻塞IO：非阻塞异步通信模式，NIO的升级版，采用异步通道实现异步通信，其read和write方法均是异步方法。用户进程发起一个IO操作，然后立即返回，等IO操作真正的完成以后，应用程序会得到IO操作完成的通知。类似Future模式。

40. 守护线程是什么？

- 守护线程是运行在后台的一种特殊进程。
- 它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。
- 在 Java 中垃圾回收线程就是特殊的守护线程。

41. Java支持多继承吗？

java中，**类不支持多继承。接口才支持多继承**。接口的作用是拓展对象功能。当一个子接口继承了多个父接口时，说明子接口拓展了多个功能。当一个类实现该接口时，就拓展了多个的功能。

Java不支持多继承的原因：

- 出于安全性的考虑，如果子类继承的多个父类里面有相同的方法或者属性，子类将不知道具体要继承哪个。
- Java提供了接口和内部类以达到实现多继承功能，弥补单继承的缺陷。

42. 如何实现对象克隆？

- 实现 `Cloneable` 接口，重写 `clone()` 方法。这种方式是浅拷贝，即如果类中属性有自定义引用类型，只拷贝引用，不拷贝引用指向的对象。如果对象的属性的Class也实现 `Cloneable` 接口，那么在克隆对象时也会克隆属性，即深拷贝。
- 结合序列化，深拷贝。
- 通过 `org.apache.commons` 中的工具类 `BeanUtils` 和 `PropertyUtils` 进行对象复制。

43. 同步和异步的区别？

同步：发出一个调用时，在没有得到结果之前，该调用就不返回。

异步：在调用发出后，被调用者返回结果之后会通知调用者，或通过回调函数处理这个调用。

44. 阻塞和非阻塞的区别？

阻塞和非阻塞关注的是线程的状态。

阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会恢复运行。

非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

举个例子，理解下同步、阻塞、异步、非阻塞的区别：

同步就是烧开水，要自己来看开没开；异步就是水开了，然后水壶响了通知你水开了（回调通知）。阻塞是烧开水的过程中，你不能干其他事情，必须在旁边等着；非阻塞是烧开水的过程里可以干其他事情。

45. Java8的新特性有哪些？

- Lambda 表达式：Lambda允许把函数作为一个方法的参数
- Stream API：新添加的Stream API (`java.util.stream`) 把真正的函数式编程风格引入到Java中
- 默认方法：默认方法就是一个在接口里面有了一个实现的方法。
- Optional 类：Optional 类已经成为 Java 8 类库的一部分，用来解决空指针异常。
- Date Time API：加强对日期与时间的处理。

[Java8 新特性总结](#)

46. 序列化和反序列化

- 序列化：把对象转换为字节序列的过程称为对象的序列化。
- 反序列化：把字节序列恢复为对象的过程称为对象的反序列化。

47. 什么时候需要用到序列化和反序列化呢？

当我们只在本地 JVM 里运行下 Java 实例，这个时候是不需要什么序列化和反序列化的，但当我们需要将内存中的对象持久化到磁盘，数据库中时，当我们需要与浏览器进行交互时，当我们需要实现 RPC 时，这个时候就需要序列化和反序列化了。

前两个需要用到序列化和反序列化的场景，是不是让我们有一个很大的疑问？我们在与浏览器交互时，还有将内存中的对象持久化到数据库中时，好像都没有去进行序列化和反序列化，因为我们都没有实现 Serializable 接口，但一直正常运行。

下面先给出结论：

只要我们对内存中的对象进行持久化或网络传输，这个时候都需要序列化和反序列化。

理由：

服务器与浏览器交互时真的没有用到 Serializable 接口吗？JSON 格式实际上就是将一个对象转化为字符串，所以服务器与浏览器交互时的数据格式其实是字符串，我们来看看 String 类型的源码：

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /\* \* The value is used for character storage. \*/
    private final char value\[\];

    /\* \* Cache the hash code for the string \*/
    private int hash; // Default to 0

    /\* \* use serialVersionUID from JDK 1.0.2 for interoperability \*/
    private static final long serialVersionUID = -6849794470754667710L;

    .....
}
```

String 类型实现了 Serializable 接口，并显示指定 serialVersionUID 的值。

然后我们再来看对象持久化到数据库中时的情况，Mybatis 数据库映射文件里的 insert 代码：

```
<insert id="insertUser" parameterType="org.tyshawn.bean.User">
    INSERT INTO t\_user(name, age) VALUES ({name}, {age})
</insert>
```

实际上我们并不是将整个对象持久化到数据库中，而是将对象中的属性持久化到数据库中，而这些属性（如 Date/String）都实现了 Serializable 接口。

48. 实现序列化和反序列化为什么要实现 Serializable 接口？

在 Java 中实现了 Serializable 接口后，JVM 在类加载的时候就会发现我们实现了这个接口，然后在初始化实例对象的时候就会在底层帮我们实现序列化和反序列化。

如果被写对象类型不是 String、数组、Enum，并且没有实现 Serializable 接口，那么在进行序列化的时候，将抛出 NotSerializableException。源码如下：

```
// remaining cases
if (obj instanceof String) {
    writeString((String) obj, unshared);
} else if (cl.isArray()) {
    writeArray(obj, desc, unshared);
} else if (obj instanceof Enum) {
    writeEnum((Enum<?>) obj, desc, unshared);
} else if (obj instanceof Serializable) {
    writeOrdinaryObject(obj, desc, unshared);
} else {
    if (extendedDebugInfo) {
        throw new NotSerializableException(
            cl.getName() + "\n" + debugInfoStack.toString());
    } else {
        throw new NotSerializableException(cl.getName());
    }
}
}
```

49. 实现 Serializable 接口之后，为什么还要显示指定 serialVersionUID 的值？

如果不显示指定 serialVersionUID，JVM 在序列化时会根据属性自动生成一个 serialVersionUID，然后与属性一起序列化，再进行持久化或网络传输。在反序列化时，JVM 会再根据属性自动生成一个新版 serialVersionUID，然后将这个新版 serialVersionUID 与序列化时生成的旧版 serialVersionUID 进行比较，如果相同则反序列化成功，否则报错。

如果显示指定了 serialVersionUID，JVM 在序列化和反序列化时仍然都会生成一个 serialVersionUID，但值为我们显示指定的值，这样在反序列化时新旧版本的 serialVersionUID 就一致了。

如果我们的类写完后不再修改，那么不指定 serialVersionUID，不会有问題，但这在实际开发中是不可能的，我们的类会不断迭代，一旦类被修改了，那旧对象反序列化就会报错。所以在实际开发中，我们都会显示指定一个 serialVersionUID。

50. static 属性为什么不会被序列化？

因为序列化是针对对象而言的，而 static 属性优先于对象存在，随着类的加载而加载，所以不会被序列化。

看到这个结论，是不是有人会问，serialVersionUID 也被 static 修饰，为什么 serialVersionUID 会被序列化？其实 serialVersionUID 属性并没有被序列化，JVM 在序列化对象时会自动生成一个 serialVersionUID，然后将我们显示指定的 serialVersionUID 属性值赋给自动生成的 serialVersionUID。

51. transient 关键字的作用？

Java 语言的关键字，变量修饰符，如果用 transient 声明一个实例变量，当对象存储时，它的值不需要维持。

也就是说被 transient 修饰的成员变量，在序列化的时候其值会被忽略，在被反序列化后，transient 变量的值被设为初始值，如 int 型的是 0，对象型的是 null。

52. 什么是反射？

动态获取的信息以及动态调用对象的方法的功能称为 Java 语言的反射机制。

在运行状态中，对于任意一个类，能够知道这个类的所有属性和方法。对于任意一个对象，能够调用它的任意一个方法和属性。

53. 反射有哪些应用场景呢？

1. JDBC连接数据库时使用 `Class.forName()` 通过反射加载数据库的驱动程序
2. Eclipse、IDEA等开发工具利用反射动态解析对象的类型与结构，动态提示对象的属性和方法
3. Web服务器中利用反射调用了Servlet的 `service` 方法
4. JDK动态代理底层依赖反射实现

54. 讲讲什么是泛型？

Java泛型是JDK 5中引入的一个新特性，允许在定义类和接口的时候使用类型参数。声明的类型参数在使用时用具体的类型来替换。

泛型最大的好处是可以提高代码的复用性。以List接口为例，我们可以将String、Integer等类型放入List中，如不用泛型，存放String类型要写一个List接口，存放Integer要写另外一个List接口，泛型可以很好的解决这个问题。

55. 如何停止一个正在运行的线程？

有几种方式。

1、使用线程的stop方法。

使用stop()方法可以强制终止线程。不过stop是一个被废弃掉的方法，不推荐使用。

使用Stop方法，会一直向上传播ThreadDeath异常，从而使得目标线程解锁所有锁住的监视器，即释放掉所有的对象锁。使得之前被锁住的对象得不到同步的处理，因此可能会造成数据不一致的问题。

2、使用interrupt方法中断线程，该方法只是告诉线程要终止，但最终何时终止取决于计算机。调用interrupt方法仅仅是在当前线程中打了一个停止的标记，并不是真的停止线程。

接着调用 `Thread.currentThread().isInterrupted()` 方法，可以用来判断当前线程是否被终止，通过这个判断我们可以做一些业务逻辑处理，通常如果isInterrupted返回true的话，会抛一个中断异常，然后通过try-catch捕获。

3、设置标志位

设置标志位，当标识位为某个值时，使线程正常退出。设置标志位是用到了共享变量的方式，为了保证共享变量在内存中的可见性，可以使用volatile修饰它，这样的话，变量取值始终会从主存中获取最新值。

但是这种volatile标记共享变量的方式，在线程发生阻塞时是无法完成响应的。比如调用Thread.sleep()方法之后，线程处于不可运行状态，即便是主线程修改了共享变量的值，该线程此时根本无法检查循环标志，所以也就无法实现线程中断。

因此，interrupt() 加上手动抛异常的方式是目前中断一个正在运行的线程**最为正确**的方式了。

56. 什么是跨域？

简单来讲，跨域是指从一个域名的网页去请求另一个域名的资源。由于有**同源策略**的关系，一般是不允许这么直接访问的。但是，很多场景经常会有跨域访问的需求，比如，在前后端分离的模式下，前后端的域名是不一致的，此时就会发生跨域问题。

那什么是同源策略呢？

所谓同源是指"协议+域名+端口"三者相同，即便两个不同的域名指向同一个ip地址，也非同源。

同源策略限制以下几种行为：

1. Cookie、LocalStorage 和 IndexDB 无法读取
2. DOM 和 JS对象无法获得
3. AJAX 请求不能发送

为什么要有同源策略？

举个例子，假如你刚刚在网银输入账号密码，查看了自己的余额，然后再去访问其他带颜色的网站，这个网站可以访问刚刚的网银站点，并且获取账号密码，那后果可想而知。因此，从安全的角度来讲，同源策略是有利于保护网站信息的。

57. 跨域问题怎么解决呢？

嗯，有以下几种方法：

CORS，跨域资源共享

CORS (Cross-origin resource sharing)，跨域资源共享。CORS 其实是浏览器制定的一个规范，浏览器会自动进行 CORS 通信，它的实现主要在服务端，通过一些 HTTP Header 来限制可以访问的域，例如页面 A 需要访问 B 服务器上的数据，如果 B 服务器上声明了允许 A 的域名访问，那么从 A 到 B 的跨域请求就可以完成。

@CrossOrigin注解

如果项目使用的是Springboot，可以在Controller类上添加一个 @CrossOrigin(origins = "*") 注解就可以实现对当前controller 的跨域访问了，当然这个标签也可以加到方法上，或者直接加到入口类上对所有接口进行跨域处理。注意SpringMVC的版本要在4.2或以上版本才支持@CrossOrigin。

nginx反向代理接口跨域

nginx反向代理跨域原理如下：首先同源策略是浏览器的安全策略，不是HTTP协议的一部分。服务器端调用HTTP接口只是使用HTTP协议，不会执行JS脚本，不需要同源策略，也就不存在跨越问题。

nginx反向代理接口跨域实现思路如下：通过nginx配置一个代理服务器（域名与domain1相同，端口不同）做跳板机，反向代理访问domain2接口，并且可以顺便修改cookie中domain信息，方便当前域cookie写入，实现跨域登录。

```
// proxy服务器
server {
    listen      81;
    server_name www.domain1.com;
    location / {
        proxy_pass http://www.domain2.com:8080; #反向代理
        proxy_cookie_domain www.domain2.com www.domain1.com; #修改cookie里域名
        index index.html index.htm;

        add_header Access-Control-Allow-Origin http://www.domain1.com;
    }
}
```

这样我们的前端代理只要访问 <http://www.domain1.com:81/>*就可以了。

通过jsonp跨域

通常为了减轻web服务器的负载，我们把js、css、img等静态资源分离到另一台独立域名的服务器上，在html页面中再通过相应的标签从不同域名下加载静态资源，这是浏览器允许的操作，基于此原理，我们可以通过动态创建script，再请求一个带参网址实现跨域通信。

58. 设计接口要注意什么？

1. **接口参数校验**。接口必须校验参数，比如入参是否允许为空，入参长度是否符合预期。
2. 设计接口时，充分考虑接口的**可扩展性**。思考接口是否可以复用，怎样保持接口的可扩展性。
3. **串行调用考虑改并行调用**。比如设计一个商城首页接口，需要查商品信息、营销信息、用户信息等。如果是串行一个一个查，那耗时就比较大了。这种场景是可以改为并行调用的，降低接口耗时。
4. 接口是否需要**防重处理**。涉及到数据库修改的，要考虑防重处理，可以使用数据库防重表，以唯一流水号作为唯一索引。
5. **日志打印全面**，入参出参，接口耗时，记录好日志，方便甩锅。
6. 修改旧接口时，注意**兼容性设计**。
7. **异常处理得当**。使用finally关闭流资源、使用log打印而不是e.printStackTrace()、不要吞异常等等
8. 是否需要考虑**限流**。限流为了保护系统，防止流量洪峰超过系统的承载能力。

59. 过滤器和拦截器有什么区别？

1、实现原理不同。

过滤器和拦截器底层实现不同。过滤器是基于函数回调的，拦截器是基于Java的反射机制（动态代理）实现的。一般自定义的过滤器中都会实现一个doFilter()方法，这个方法有一个FilterChain参数，而实际上它是一个回调接口。

2、使用范围不同。

过滤器实现的是 javax.servlet.Filter 接口，而这个接口是在Servlet规范中定义的，也就是说过滤器Filter的使用要依赖于Tomcat等容器，导致它只能在web程序中使用。而拦截器是一个Spring组件，并由Spring容器管理，并不依赖Tomcat等容器，是可以单独使用的。拦截器不仅能应用在web程序中，也可以用于Application、Swing等程序中。

3、使用的场景不同。

因为拦截器更接近业务系统，所以拦截器主要用来实现项目中的业务判断的，比如：日志记录、权限判断等业务。而过滤器通常是用来实现通用功能过滤的，比如：敏感词过滤、响应数据压缩等功能。

4、触发时机不同。

过滤器Filter是在请求进入容器后，但在进入servlet之前进行预处理，请求结束是在servlet处理完以后。

拦截器 Interceptor 是在请求进入servlet后，在进入Controller之前进行预处理的，Controller 中渲染了对应的视图之后请求结束。

5、拦截的请求范围不同。

请求的执行顺序是：请求进入容器 -> 进入过滤器 -> 进入 Servlet -> 进入拦截器 -> 执行控制器。可以看到过滤器和拦截器的执行时机也是不同的，过滤器会先执行，然后才会执行拦截器，最后才会进入真正的要调用的方法。

参考链接：<https://segmentfault.com/a/1190000022833940>

60. 对接第三方接口要考虑什么？

嗯，需要考虑以下几点：

1. 确认接口对接的**网络协议**，是https/http或者自定义的私有协议等。
2. 约定好**数据传参、响应格式**（如application/json），弱类型对接强类型语言时要特别注意
3. **接口安全**方面，要确定身份校验方式，使用token、证书校验等
4. 确认是否需要接口调用失败后的**重试**机制，保证数据传输的最终一致性。
5. **日志记录要全面**。接口出入参数，以及解析之后的参数值，都要用日志记录下来，方便定位问题（甩锅）。

参考：<https://blog.csdn.net/gzt19881123/article/details/108791034>

61. 后端接口性能优化有哪些方法？

有以下这些方法：

- 1、**优化索引**。给where条件的关键字段，或者 `order by` 后面的排序字段，加索引。
- 2、**优化sql语句**。比如避免使用select *、批量操作、避免深分页、提升group by的效率等
- 3、**避免大事务**。使用@Transactional注解这种声明式事务的方式提供事务功能，容易造成大事务，引发其他的问题。应该避免在事务中一次性处理太多数据，将一些跟事务无关的逻辑放到事务外面执行。
- 4、**异步处理**。剥离主逻辑和副逻辑，副逻辑可以异步执行，异步写库。比如用户购买的商品发货了，需要发短信通知，短信通知是副流程，可以异步执行，以免影响主流程的执行。
- 5、**降低锁粒度**。在并发场景下，多个线程同时修改数据，造成数据不一致的情况。这种情况下，一般会加锁解决。但如果锁加得不好，导致锁的粒度太粗，也会非常影响接口性能。
- 6、**加缓存**。如果表数据量非常大的话，直接从数据库查询数据，性能会非常差。可以使用Redis 和 memcached提升查询性能，从而提高接口性能。
- 7、**分库分表**。当系统发展到一定的阶段，用户并发量大，会有大量的数据库请求，需要占用大量的数据库连接，同时会带来磁盘IO的性能瓶颈问题。或者数据库表数据非常大，SQL查询即使走了索引，也很耗时。这时，可以通过分库分表解决。分库用于解决数据库连接资源不足问题，和磁盘IO的性能瓶颈问题。分表用于解决单表数据量太大，sql语句查询数据时，即使走了索引也非常耗时问题。
- 8、**避免在循环中查询数据库**。循环查询数据库，非常耗时，最好能在一次查询中获取所有需要的数据。



扫码关注我



微信搜索



程序员大彬

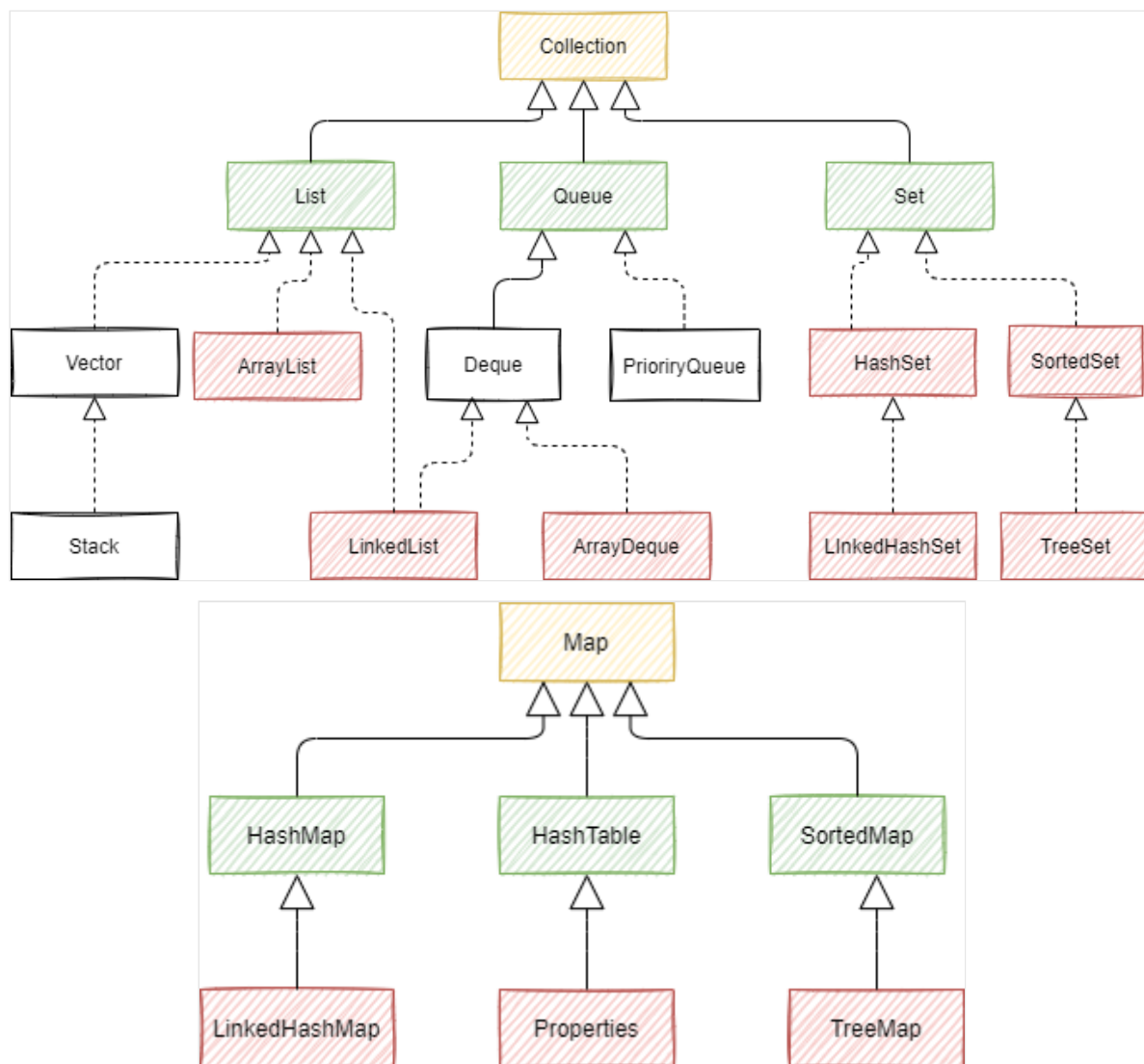
公众号后台回复【面试】获取面试手册
PDF最新版

Java集合

1. 常见的集合有哪些？

Java集合类主要由两个接口**Collection**和**Map**派生出来的，Collection有三个子接口：List、Set、Queue。

Java集合框架图如下：



List代表了有序可重复集合，可直接根据元素的索引来访问；Set代表无序不可重复集合，只能根据元素本身来访问；Queue是队列集合。Map代表的是存储key-value对的集合，可根据元素的key来访问value。

集合体系中常用的实现类有 `ArrayList`、`LinkedList`、`HashSet`、`TreeSet`、`HashMap`、`TreeMap` 等实现类。

2. List、Set和Map 的区别

- List 以索引来存取元素，有序的，元素是允许重复的，可以插入多个null；
- Set 不能存放重复元素，无序的，只允许一个null；
- Map 保存键值对映射；
- List 底层实现有数组、链表两种方式；Set、Map 容器有基于哈希存储和红黑树两种方式实现；
- Set 基于 Map 实现，Set 里的元素值就是 Map的键值。

3. ArrayList 了解吗?

`ArrayList` 的底层是动态数组，它的容量能动态增长。在添加大量元素前，应用可以使用 `ensureCapacity` 操作增加 `ArrayList` 实例的容量。`ArrayList` 继承了 `AbstractList`，并实现了 `List` 接口。

4. ArrayList 的扩容机制?

`ArrayList`扩容的本质就是计算出新的扩容数组的size后实例化，并将原有数组内容复制到新数组中去。默认情况下，新的容量会是原容量的1.5倍。以JDK1.8为例说明:

```
public boolean add(E e) {
    //判断是否可以容纳e，若能，则直接添加在末尾；若不能，则进行扩容，然后再把e添加在末尾
    ensureCapacityInternal(size + 1); // Increments modCount!!
    //将e添加到数组末尾
    elementData[size++] = e;
    return true;
}

// 每次在add()一个元素时，arraylist都需要对这个list的容量进行一个判断。通过
// ensureCapacityInternal()方法确保当前ArrayList维护的数组具有存储新元素的能力，经过处理之后
// 将元素存储在数组elementData的尾部

private void ensureCapacityInternal(int minCapacity) {
    ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));
}

private static int calculateCapacity(Object[] elementData, int minCapacity) {
    //如果传入的是个空数组则最小容量取默认容量与minCapacity之间的最大值
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        return Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    return minCapacity;
}

private void ensureExplicitCapacity(int minCapacity) {
    modCount++;
    // 若ArrayList已有的存储能力满足最低存储要求，则返回add直接添加元素；如果最低要求的
    // 存储能力>ArrayList已有的存储能力，这就表示ArrayList的存储能力不足，因此需要调用 grow();方法进行扩容
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}

private void grow(int minCapacity) {
    // 获取elementData数组的内存空间长度
    int oldCapacity = elementData.length;
    // 扩容至原来的1.5倍
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    //校验容量是否够
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    //若预设值大于默认的最大值，检查是否溢出
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
}
```

```
// 调用Arrays.copyOf方法将elementData数组指向新的内存空间
// 并将elementData的数据复制到新的内存空间
elementData = Arrays.copyOf(elementData, newCapacity);
}
```

5. 怎么在遍历 ArrayList 时移除一个元素?

foreach删除会导致快速失败问题，可以使用迭代器的 remove() 方法。

```
Iterator itr = list.iterator();
while(itr.hasNext()) {
    if(itr.next().equals("jay")) {
        itr.remove();
    }
}
```

6. Arraylist 和 Vector 的区别

1. ArrayList在内存不够时扩容为原来的1.5倍，Vector是扩容为原来的2倍。
2. Vector属于线程安全级别的，但是大多数情况下不使用Vector，因为操作Vector效率比较低。

7. Arraylist 与 LinkedList的区别

1. ArrayList基于动态数组实现；LinkedList基于链表实现。
2. 对于随机index访问的get和set方法，ArrayList的速度要优于LinkedList。因为ArrayList直接通过数组下标直接找到元素；LinkedList要移动指针遍历每个元素直到找到为止。
3. 新增和删除元素，LinkedList的速度要优于ArrayList。因为ArrayList在新增和删除元素时，可能扩容和复制数组；LinkedList实例化对象需要时间外，只需要修改指针即可。

8. HashMap

HashMap 使用数组+链表+红黑树（JDK1.8增加了红黑树部分）实现的，链表长度大于8（`TREEIFY_THRESHOLD`）时，会把链表转换为红黑树，红黑树节点个数小于6（`UNTREEIFY_THRESHOLD`）时才转化为链表，防止频繁的转化。

8.1. 解决hash冲突的办法有哪些？HashMap用的哪种？

解决Hash冲突方法有：开放定址法、再哈希法、链地址法。HashMap中采用的是 链地址法。

- 开放定址法基本思想就是，如果 $p=H(key)$ 出现冲突时，则以 p 为基础，再次hash， $p_1=H(p)$ ，如果 p_1 再次出现冲突，则以 p_1 为基础，以此类推，直到找到一个不冲突的哈希地址 p_i 。因此开放定址法所需要的hash表的长度要大于等于所需要存放的元素，而且因为存在再次hash，所以 **只能在删除的节点上做标记，而不能真正删除节点。**
- 再哈希法提供多个不同的hash函数，当 $R_1=H_1(key_1)$ 发生冲突时，再计算 $R_2=H_2(key_1)$ ，直到没有冲突为止。这样做虽然不易产生堆集，但增加了计算的时间。
- 链地址法将哈希值相同的元素构成一个同义词的单链表，并将单链表的头指针存放在哈希表的第 i 个单元中，查找、插入和删除主要在同义词链表中进行。链表法适用于经常进行插入和删除的情况。

8.2. 使用的hash算法？

Hash算法：取key的hashCode值、高位运算、取模运算。

```
h=key.hashCode() //第一步 取hashCode值
h^(h>>16) //第二步 高位参与运算，减少冲突
return h&(length-1); //第三步 取模运算
```

在JDK1.8的实现中，优化了高位运算的算法，通过 `hashCode()` 的高16位异或低16位实现的：这么做可以在数组比较小的时候，也能保证考虑到高低位都参与到Hash的计算中，可以减少冲突，同时不会有太大的开销。

8.3. 为什么建议设置HashMap的容量？

HashMap有扩容机制，就是当达到扩容条件时会进行扩容。扩容条件就是当HashMap中的元素个数超过临界值时就会自动扩容（ $\text{threshold} = \text{loadFactor} * \text{capacity}$ ）。

如果我们没有设置初始容量大小，随着元素的不断增加，HashMap会发生多次扩容。而HashMap每次扩容都需要重建hash表，非常影响性能。所以建议开发者在创建HashMap的时候指定初始化容量。

8.4. 扩容过程？

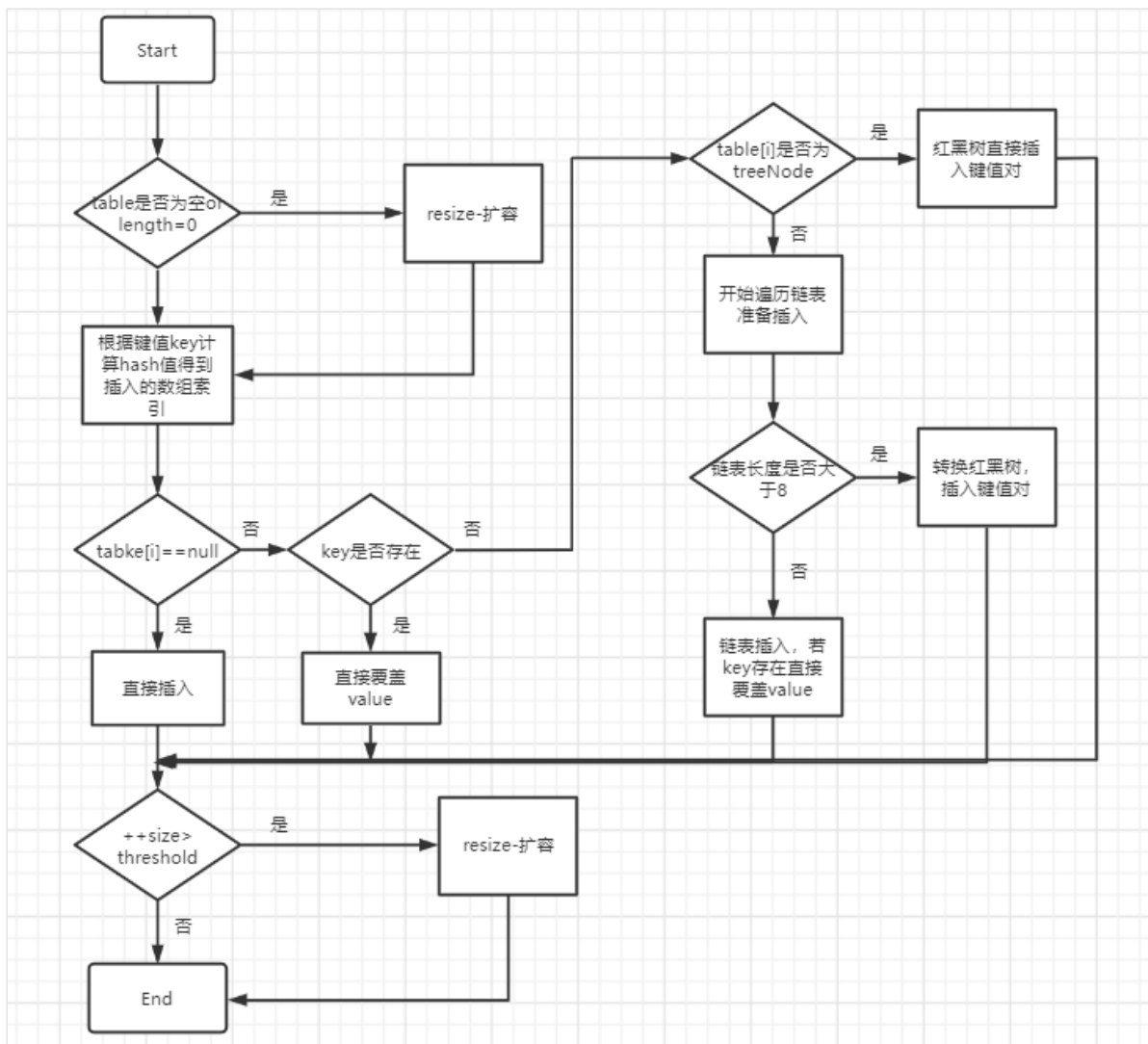
1.8扩容机制：当元素个数大于 `threshold` 时，会进行扩容，使用2倍容量的数组代替原有数组。采用尾插入的方式将原数组元素拷贝到新数组。1.8扩容之后链表元素相对位置没有变化，而1.7扩容之后链表元素会倒置。

1.7链表新节点采用的是头插法，这样在线程一扩容迁移元素时，会将元素顺序改变，导致两个线程中出现元素的相互指向而形成循环链表，1.8采用了尾插法，避免了这种情况的发生。

原数组的元素在重新计算hash之后，因为数组容量n变为2倍，那么n-1的mask范围在高位多1bit。在元素拷贝过程不需要重新计算元素在数组中的位置，只需要看看原来的hash值新增的那个bit是1还是0，是0的话索引没变，是1的话索引变成“原索引+oldCap”（根据 `e.hash & oldCap == 0` 判断）。这样可以省去重新计算hash值的时间，而且由于新增的1bit是0还是1可以认为是随机的，因此resize的过程会均匀的把之前的冲突的节点分散到新的bucket。

8.5. put方法流程？

1. 如果table没有初始化就先进行初始化过程
2. 使用hash算法计算key的索引
3. 判断索引处有没有存在元素，没有就直接插入
4. 如果索引处存在元素，则遍历插入，有两种情况，一种是链表形式就直接遍历到尾端插入，一种是红黑树就按照红黑树结构插入
5. 链表的数量大于阈值8，就要转换成红黑树的结构
6. 添加成功后会检查是否需要扩容



8.6. 红黑树的特点?

- 每个节点或者是黑色，或者是红色。
- 根节点和叶子节点（NIL）是黑色的。
- 如果一个节点是红色的，则它的子节点必须是黑色的。
- 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

8.7. 在解决 hash 冲突的时候，为什么选择先用链表，再转红黑树？

因为红黑树需要进行左旋，右旋，变色这些操作来保持平衡，而单链表不需要。所以，当元素个数小于8个的时候，采用链表结构可以保证查询性能。而当元素个数大于8个的时候并且数组容量大于等于64，会采用红黑树结构。因为红黑树搜索时间复杂度是 $O(\log n)$ ，而链表是 $O(n)$ ，在 n 比较大的时候，使用红黑树可以加快查询速度。

8.8. HashMap 的长度为什么是 2 的幂次方？

Hash 值的范围值比较大，使用之前需要先对数组的长度取模运算，得到的余数才是元素存放的位置也就是对应的数组下标。这个数组下标的计算方法是 $(n - 1) \& hash$ 。将 HashMap 的长度定为 2 的幂次方，这样就可以使用 $(n - 1) \& hash$ 位运算代替 % 取余的操作，提高性能。

8.9. HashMap默认加载因子是多少？为什么是 0.75？

先看下HashMap的默认构造函数：

```
int threshold;           // 容纳键值对的最大值
final float loadFactor;   // 负载因子
int modCount;
int size;
```

Node[] table的初始化长度length为16，默认的loadFactor是0.75，0.75是对空间和时间效率的一个平衡选择，根据泊松分布，loadFactor 取0.75碰撞最小。一般不会修改，除非在时间和空间比较特殊的情况下：

- 如果内存空间很多而又对时间效率要求很高，可以降低负载因子Load factor的值。
- 如果内存空间紧张而对时间效率要求不高，可以增加负载因子loadFactor的值，这个值可以大于1。

8.10. 一般用什么作为HashMap的key？

一般用 `Integer`、`String` 这种不可变类当 HashMap 当 key。`String`类比较常用。

- 因为 `String` 是不可变的，所以在它创建的时候`hashCode`就被缓存了，不需要重新计算。这就是 HashMap 中的key经常使用字符串的原因。
- 获取对象的时候要用到 `equals()` 和 `hashCode()` 方法，而`Integer`、`String`这些类都已经重写了 `hashCode()` 以及 `equals()` 方法，不需要自己去重写这两个方法。

8.11. HashMap为什么线程不安全？

- 多线程下扩容死循环。JDK1.7中的 HashMap 使用头插法插入元素，在多线程的环境下，扩容的时候有可能导致**环形链表**的出现，形成死循环。
- 在JDK1.8中，在多线程环境下，会发生**数据覆盖**的情况。

8.12. HashMap和HashTable的区别？

HashMap和Hashtable都实现了Map接口。

1. HashMap可以接受为null的key和value，key为null的键值对放在下标为0的头结点的链表中，而Hashtable则不行。
2. HashMap是非线程安全的，HashTable是线程安全的。Jdk1.5提供了ConcurrentHashMap，它是HashTable的替代。
3. Hashtable很多方法是同步方法，在单线程环境下它比HashMap要慢。
4. 哈希值的使用不同，HashTable直接使用对象的hashCode。而HashMap重新计算hash值。

9. LinkedHashMap底层原理？

HashMap是无序的，迭代HashMap所得到元素的顺序并不是它们最初放到HashMap的顺序，即不能保持它们的插入顺序。

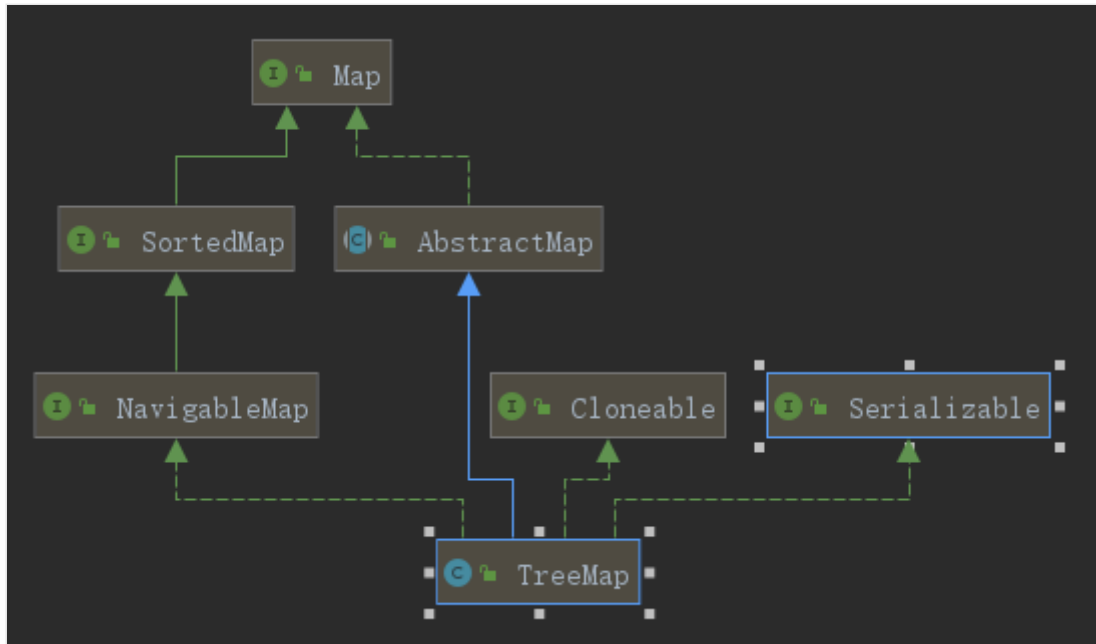
LinkedHashMap继承于HashMap，是HashMap和LinkedList的融合体，具备两者的特性。每次put操作都会将entry插入到双向链表的尾部。

10. 讲一下TreeMap？

TreeMap是一个能比较元素大小的Map集合，会对传入的key进行了大小排序。可以使用元素的自然顺序，也可以使用集合中自定义的比较器来进行排序。

```
public class TreeMap<K,V>
    extends AbstractMap<K,V>
    implements NavigableMap<K,V>, Cloneable, java.io.Serializable {
}
```

TreeMap 的继承结构：



TreeMap的特点：

1. TreeMap是有序的key-value集合，通过红黑树实现。根据键的自然顺序进行排序或根据提供的Comparator进行排序。
2. TreeMap继承了AbstractMap，实现了NavigableMap接口，支持一系列的导航方法，给定具体搜索目标，可以返回最接近的匹配项。如floorEntry()、ceilingEntry()分别返回小于等于、大于等于给定键关联的Map.Entry()对象，不存在则返回null。lowerKey()、floorKey、ceilingKey、higherKey()只返回关联的key。

11. HashSet底层原理？

HashSet 基于 HashMap 实现。放入HashSet中的元素实际上由HashMap的key来保存，而HashMap的value则存储了一个静态的Object对象。

```
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable {
    static final long serialVersionUID = -5024744406713321676L;

    private transient HashMap<E,Object> map; //基于HashMap实现
    //...
}
```

12. HashSet、LinkedHashSet 和 TreeSet 的区别？

HashSet 是 **Set** 接口的主要实现类，**HashSet** 的底层是 **HashMap**，线程不安全的，可以存储 null 值；

LinkedHashSet 是 **HashSet** 的子类，能够按照添加的顺序遍历；

TreeSet 底层使用红黑树，能够按照添加元素的顺序进行遍历，排序的方式可以自定义。

13. 什么是fail fast?

fast-fail是Java集合的一种错误机制。当多个线程对同一个集合进行操作时，就有可能产生fast-fail事件。例如：当线程a正通过iterator遍历集合时，另一个线程b修改了集合的内容，此时modCount（记录集合操作过程的修改次数）会加1，不等于expectedModCount，那么线程a访问集合的时候，就会抛出ConcurrentModificationException，产生fast-fail事件。边遍历边修改集合也会产生fast-fail事件。

解决方法：

- 使用Collections.synchronizedList方法或在修改集合内容的地方加上synchronized。这样的话，增删集合内容的同步锁会阻塞遍历操作，影响性能。
- 使用CopyOnWriteArrayList来替换ArrayList。在对CopyOnWriteArrayList进行修改操作的时候，会拷贝一个新的数组，对新的数组进行操作，操作完成后再把引用移到新的数组。

14. 什么是fail safe?

采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。java.util.concurrent包下的容器都是安全失败，可以在多线程下并发使用，并发修改。

原理：由于迭代时是对原集合的拷贝进行遍历，所以在遍历过程中对原集合所作的修改并不能被迭代器检测到，所以不会触发Concurrent Modification Exception。

缺点：基于拷贝内容的优点是避免了Concurrent Modification Exception，但同样地，迭代器并不能访问到修改后的内容，即：迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器是不知道的。

15. 讲一下ArrayDeque?

ArrayDeque实现了双端队列，内部使用循环数组实现，默认大小为16。它的特点有：

1. 在两端添加、删除元素的效率较高
2. 根据元素内容查找和删除的效率比较低。
3. 没有索引位置的概念，不能根据索引位置进行操作。

ArrayDeque和LinkedList都实现了Deque接口，如果只需要从两端进行操作，ArrayDeque效率更高一些。如果同时需要根据索引位置进行操作，或者经常需要在中间进行插入和删除（LinkedList有相应的api，如add(int index, E e)），则应该选LinkedList。

ArrayDeque和LinkedList都是线程不安全的，可以使用Collections工具类中synchronizedXxx()转换成线程同步。

16. 哪些集合类是线程安全的？哪些不安全？

线程安全的集合类：

- Vector：比ArrayList多了同步机制。
- Hashtable。
- ConcurrentHashMap：是一种高效并且线程安全的集合。
- Stack：栈，也是线程安全的，继承于Vector。

线程不安全的集合类：

- HashMap
- ArrayList
- LinkedList
- HashSet

- TreeSet
- TreeMap

17. 迭代器 Iterator 是什么？

Iterator模式用同一种逻辑来遍历集合。它可以把访问逻辑从不同类型的集合类中抽象出来，不需要了解集合内部实现便可以遍历集合元素，统一使用 Iterator 提供的接口去遍历。它的特点是更加安全，因为它可以保证，在当前遍历的集合元素被更改的时候，就会抛出 `ConcurrentModificationException` 异常。

```
public interface Collection<E> extends Iterable<E> {  
    Iterator<E> iterator();  
}
```

主要有三个方法：`hasNext()`、`next()`和`remove()`。

18. Iterator 和 ListIterator 有什么区别？

ListIterator 是 Iterator的增强版。

- ListIterator遍历可以是逆向的，因为有`previous()`和`hasPrevious()`方法，而Iterator不可以。
- ListIterator有`add()`方法，可以向List添加对象，而Iterator却不能。
- ListIterator可以定位当前的索引位置，因为有`nextIndex()`和`previousIndex()`方法，而Iterator不可以。
- ListIterator可以实现对象的修改，`set()`方法可以实现。Iterator仅能遍历，不能修改。
- ListIterator只能用于遍历List及其子类，Iterator可用来遍历所有集合。

19. 如何让一个集合不能被修改？

可以采用Collections包下的`unmodifiableMap`/`unmodifiableList`/`unmodifiableSet`方法，通过这个方法返回的集合，是不可以修改的。如果修改的话，会抛出 `java.lang.UnsupportedOperationException` 异常。

```
List<String> list = new ArrayList<>();  
list.add("x");  
Collection<String> clist = Collections.unmodifiableCollection(list);  
clist.add("y"); // 运行时此行报错  
System.out.println(list.size());
```

对于List/Set/Map集合，Collections包都有相应的支持。

那使用final关键字进行修饰可以实现吗？

答案是不可以。

final关键字修饰的成员变量如果是引用类型的话，则表示这个引用的地址值是不能改变的，但是这个引用所指向的对象里面的内容还是可以改变的。

而集合类都是引用类型，用final修饰的话，集合里面的内容还是可以修改的。

20. 并发容器

JDK 提供的这些容器大部分在 `java.util.concurrent` 包中。

- **ConcurrentHashMap**: 线程安全的 HashMap
- **CopyOnWriteArrayList**: 线程安全的 List，在读多写少的场合性能非常好，远远好于 Vector。

- **ConcurrentLinkedQueue**: 高效的并发队列，使用链表实现。可以看做一个线程安全的LinkedList，这是一个非阻塞队列。
- **BlockingQueue**: 阻塞队列接口，JDK 内部通过链表、数组等方式实现了这个接口。非常适合用于作为数据共享的通道。
- **ConcurrentSkipListMap**: 跳表的实现。使用跳表的数据结构进行快速查找。

20.1. ConcurrentHashMap

多线程环境下，使用HashMap进行put操作会引起死循环，应该使用支持多线程的ConcurrentHashMap。

JDK1.8 ConcurrentHashMap取消了segment分段锁，而采用CAS和synchronized来保证并发安全。数据结构采用数组+链表/红黑二叉树。synchronized只锁定当前链表或红黑二叉树的首节点，相比1.7锁定HashEntry数组，锁粒度更小，支持更高的并发量。当链表长度过长时，Node会转换成TreeNode，提高查找速度。

20.1.1. put执行流程？

在put的时候需要锁住Segment，保证并发安全。调用get的时候不加锁，因为node数组成员val和指针next是用volatile修饰的，更改后的值会立刻刷新到主存中，保证了可见性，node数组table也用volatile修饰，保证在运行过程对其他线程具有可见性。

```
transient volatile Node<K,V>[] table;

static class Node<K,V> implements Map.Entry<K,V> {
    volatile V val;
    volatile Node<K,V> next;
}
```

put 操作流程：

1. 如果table没有初始化就先进行初始化过程
2. 使用hash算法计算key的位置
3. 如果这个位置为空则直接CAS插入，如果不为空的话，则取出这个节点来
4. 如果取出来的节点的hash值是MOVED(-1)的话，则表示当前正在对这个数组进行扩容，复制到新的数组，则当前线程也去帮助复制
5. 如果这个节点，不为空，也不在扩容，则通过synchronized来加锁，进行添加操作，这里有两种情况，一种是链表形式就直接遍历到尾端插入或者覆盖掉相同的key，一种是红黑树就按照红黑树结构插入
6. 链表的数量大于阈值8，就会转换成红黑树的结构或者进行扩容（table长度小于64）
7. 添加成功后会检查是否需要扩容

20.1.2. 怎么扩容？

数组扩容transfer方法中会设置一个步长，表示一个线程处理的数组长度，最小值是16。在一个步长范围内只有一个线程会对其进行复制移动操作。

20.1.3. ConcurrentHashMap 和 Hashtable 的区别？

1. Hashtable通过使用synchronized修饰方法的方式来实现多线程同步，因此，Hashtable的同步会锁住整个数组。在高并发的情况下，性能会非常差。ConcurrentHashMap采用了更细粒度的锁来提高在并发情况下的效率。注：synchronized容器（同步容器）也是通过synchronized关键字来实现线程安全，在使用的时候会对所有的数据加锁。
2. Hashtable默认的大小为11，当达到阈值后，每次按照下面的公式对容量进行扩充： $newCapacity = oldCapacity * 2 + 1$ 。ConcurrentHashMap默认大小是16，扩容时容量扩大为原来的2倍。

20.2. CopyOnWrite

Copy-On-Write，写时复制。当我们往容器添加元素时，不直接往容器添加，而是先将当前容器进行复制，复制出一个新的容器，然后往新的容器添加元素，添加完元素之后，再将原容器的引用指向新容器。这样做的好处就是可以对 `CopyOnWrite` 容器进行并发的读而不需要加锁，因为当前容器不会被修改。

```
public boolean add(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock(); //add方法需要加锁
    try {
        Object[] elements = getArray();
        int len = elements.length;
        Object[] newElements = Arrays.copyOf(elements, len + 1); //复制新数组
        newElements[len] = e;
        setArray(newElements); //原容器的引用指向新容器
        return true;
    } finally {
        lock.unlock();
    }
}
```

从JDK1.5开始Java并发包里提供了两个使用CopyOnWrite机制实现的并发容器，它们是 `CopyOnWriteArrayList` 和 `CopyOnWriteArraySet`。

缺点：

- 内存占用问题。由于CopyOnWrite的写时复制机制，在进行写操作的时候，内存里会同时驻扎两个对象的内存。
- CopyOnWrite容器不能保证数据的实时一致性，可能读取到旧数据。

20.3. CopyOnWriteArrayList

`CopyOnWriteArrayList`是Java并发包中提供的一个并发容器。`CopyOnWriteArrayList`相当于线程安全的`ArrayList`，`CopyOnWriteArrayList`使用了一种叫写时复制的方法，当有新元素add到`CopyOnWriteArrayList`时，先从原有的数组中拷贝一份出来，然后在新的数组做写操作，写完之后，再将原来的数组引用指向到新数组。

`CopyOnWriteArrayList`中add方法添加的时候是需要加锁的，保证同步，避免了多线程写的时候复制出多个副本。读的时候不需要加锁，如果读的时候有其他线程正在向 `CopyOnWriteArrayList` 添加数据，还是可以读到旧的数据。

`CopyOnWrite`并发容器用于读多写少的并发场景。

优点：

读操作性能很高，因为无需任何同步措施，比较适用于**读多写少**的并发场景。Java的list在遍历时，若中途有别的线程对list容器进行修改，则会抛出`ConcurrentModificationException`异常。而`CopyOnWriteArrayList`由于其“读写分离”的思想，遍历和修改操作分别作用在不同的list容器，所以在使用迭代器进行遍历时候，也就不会抛出`ConcurrentModificationException`异常了。

缺点：

一是**内存占用问题**，毕竟每次执行写操作都要将原容器拷贝一份，数据量大时，对内存压力较大，可能会引起频繁GC；

二是无法保证实时性，Vector对于读写操作均加锁同步，可以保证读和写的强一致性。而CopyOnWriteArrayList由于其实现策略的原因，写和读分别作用在新老不同容器上，在写操作执行过程中，读不会阻塞但读取到的却是老容器的数据。

20.4. ConcurrentLinkedQueue

非阻塞队列。高效的并发队列，使用链表实现。可以看做一个线程安全的 `LinkedList`，通过 CAS 操作实现。

如果对队列加锁的成本较高则适合使用无锁的 `ConcurrentLinkedQueue` 来替代。适合在对性能要求相对较高，同时有多个线程对队列进行读写的场景。

非阻塞队列中的几种主要方法： `add(E e)`：将元素e插入到队列末尾，如果插入成功，则返回true；如果插入失败（即队列已满），则会抛出异常； `remove()`：移除队首元素，若移除成功，则返回true；如果移除失败（队列为空），则会抛出异常； `offer(E e)`：将元素e插入到队列末尾，如果插入成功，则返回true；如果插入失败（即队列已满），则返回false； `poll()`：移除并获取队首元素，若成功，则返回队首元素；否则返回null； `peek()`：获取队首元素，若成功，则返回队首元素；否则返回null

对于非阻塞队列，一般情况下建议使用offer、poll和peek三个方法，不建议使用add和remove方法。因为使用offer、poll和peek三个方法可以通过返回值判断操作成功与否，而使用add和remove方法却不能达到这样的效果。

20.5. 阻塞队列

阻塞队列是 `java.util.concurrent` 包下重要的数据结构，`BlockingQueue` 提供了线程安全的队列访问方式：当阻塞队列进行插入数据时，如果队列已满，线程将会阻塞等待直到队列非满；从阻塞队列取数据时，如果队列已空，线程将会阻塞等待直到队列非空。并发包下很多高级同步类的实现都是基于 `BlockingQueue` 实现的。`BlockingQueue` 适合用于作为数据共享的通道。

使用阻塞算法的队列可以用一个锁（入队和出队用同一把锁）或两个锁（入队和出队用不同的锁）等方式来实现。

阻塞队列和一般的队列的区别就在于：

1. 多线程支持，多个线程可以安全的访问队列
2. 阻塞操作，当队列为空的时候，消费线程会阻塞等待队列不为空；当队列满了的时候，生产线程就会阻塞直到队列不满

方法

方法\处理方式	抛出异常	返回特殊值	一直阻塞	超时退出
插入方法	<code>add(e)</code>	<code>offer(e)</code>	<code>put(e)</code>	<code>offer(e,time,unit)</code>
移除方法	<code>remove()</code>	<code>poll()</code>	<code>take()</code>	<code>poll(time,unit)</code>
检查方法	<code>element()</code>	<code>peek()</code>	不可用	不可用

20.5.1. JDK提供的阻塞队列

JDK 7 提供了7个阻塞队列，如下

1、ArrayBlockingQueue

有界阻塞队列，底层采用数组实现。`ArrayBlockingQueue` 一旦创建，容量不能改变。其并发控制采用可重入锁来控制，不管是插入操作还是读取操作，都需要获取到锁才能进行操作。此队列按照先进先出（FIFO）的原则对元素进行排序。默认情况下不能保证线程访问队列的公平性，参数 `fair` 可用于设置线程是否公平访问队列。为了保证公平性，通常会降低吞吐量。

```
private static ArrayBlockingQueue<Integer> blockingQueue = new
ArrayBlockingQueue<Integer>(10,true);//fair
```

2、LinkedBlockingQueue

`LinkedBlockingQueue` 是一个用单向链表实现的有界阻塞队列，可以当做无界队列也可以当做有界队列来使用。通常在创建 `LinkedBlockingQueue` 对象时，会指定队列最大的容量。此队列的默认和最大长度为 `Integer.MAX_VALUE`。此队列按照先进先出的原则对元素进行排序。与 `ArrayBlockingQueue` 相比起来具有更高的吞吐量。

3、PriorityBlockingQueue

支持优先级的**无界**阻塞队列。默认情况下元素采取自然顺序升序排列。也可以自定义类实现

`compareTo()` 方法来指定元素排序规则，或者初始化 `PriorityBlockingQueue` 时，指定构造参数 `Comparator` 来进行排序。

`PriorityBlockingQueue` 只能指定初始的队列大小，后面插入元素的时候，如果空间不够的话会**自动扩容**。

`PriorityQueue` 的线程安全版本。不可以插入 null 值，同时，插入队列的对象必须是可比较大小的（comparable），否则报 `ClassCastException` 异常。它的插入操作 `put` 方法不会 block，因为它是无界队列（`take` 方法在队列为空的时候会阻塞）。

4、DelayQueue

支持延时获取元素的无界阻塞队列。队列使用 `PriorityBlockingQueue` 来实现。队列中的元素必须实现 `Delayed` 接口，在创建元素时可以指定多久才能从队列中获取当前元素。只有在延迟期满时才能从队列中提取元素。

5、SynchronousQueue

不存储元素的阻塞队列，每一个 `put` 必须等待一个 `take` 操作，否则不能继续添加元素。支持公平访问队列。

`SynchronousQueue` 可以看成是一个传球手，负责把生产者线程处理的数据直接传递给消费者线程。队列本身不存储任何元素，非常适合传递性场景。`SynchronousQueue` 的吞吐量高于

`LinkedBlockingQueue` 和 `ArrayBlockingQueue`。

6、LinkedTransferQueue

由链表结构组成的无界阻塞 `TransferQueue` 队列。相对于其他阻塞队列，多了 `tryTransfer` 和 `transfer` 方法。

`transfer` 方法：如果当前有消费者正在等待接收元素（`take` 或者带时间限制的 `poll` 方法），`transfer` 可以把生产者传入的元素立刻传给消费者。如果没有消费者等待接收元素，则将元素放在队列的 `tail` 节点，并等到该元素被消费者消费了才返回。

`tryTransfer` 方法：用来试探生产者传入的元素能否直接传给消费者。如果没有消费者在等待，则返回 `false`。和上述方法的区别是该方法无论消费者是否接收，方法立即返回。而 `transfer` 方法是必须等到消费者消费了才返回。

20.5.2. 原理

JDK使用通知模式实现阻塞队列。所谓通知模式，就是当生产者往满的队列里添加元素时会阻塞生产者，当消费者消费了一个队列中的元素后，会通知生产者当前队列可用。

ArrayBlockingQueue使用Condition来实现：

```
private final Condition notEmpty;
private final Condition notFull;

public ArrayBlockingQueue(int capacity, boolean fair) {
    if (capacity <= 0)
        throw new IllegalArgumentException();
    this.items = new Object[capacity];
    lock = new ReentrantLock(fair);
    notEmpty = lock.newCondition();
    notFull = lock.newCondition();
}

public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == 0) // 队列为空时，阻塞当前消费者
            notEmpty.await();
        return dequeue();
    } finally {
        lock.unlock();
    }
}

public void put(E e) throws InterruptedException {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == items.length)
            notFull.await();
        enqueue(e);
    } finally {
        lock.unlock();
    }
}

private void enqueue(E x) {
    final Object[] items = this.items;
    items[putIndex] = x;
    if (++putIndex == items.length)
        putIndex = 0;
    count++;
    notEmpty.signal(); // 队列不为空时，通知消费者获取元素
}
```

21. 参考链接

<http://www.importnew.com/20386.html>

<https://www.cnblogs.com/yangming1996/p/7997468.html>

<https://coolshell.cn/articles/9606.htm> (HashMap 死循环)



扫码关注我



微信搜索



程序员大彬

公众号后台回复【面试】获取面试手册
PDF最新版

Java并发

1. 线程池

线程池：一个管理线程的池子。

1.1. 为什么平时都是使用线程池创建线程，直接new一个线程不好吗？

嗯，手动创建线程有两个缺点

1. 不受控风险
2. 频繁创建开销大

为什么不受控？

系统资源有限，每个人针对不同业务都可以手动创建线程，并且创建线程没有统一标准，比如创建的线程有没有名字等。当系统运行起来，所有线程都在抢占资源，毫无规则，混乱场面可想而知，不好管控。

频繁手动创建线程为什么开销会大？跟new Object() 有什么差别？

虽然Java中万物皆对象，但是new Thread() 创建一个线程和 new Object()还是有区别的。

new Object()过程如下：

1. JVM分配一块内存 M
2. 在内存 M 上初始化该对象
3. 将内存 M 的地址赋值给引用变量 obj

创建线程的过程如下：

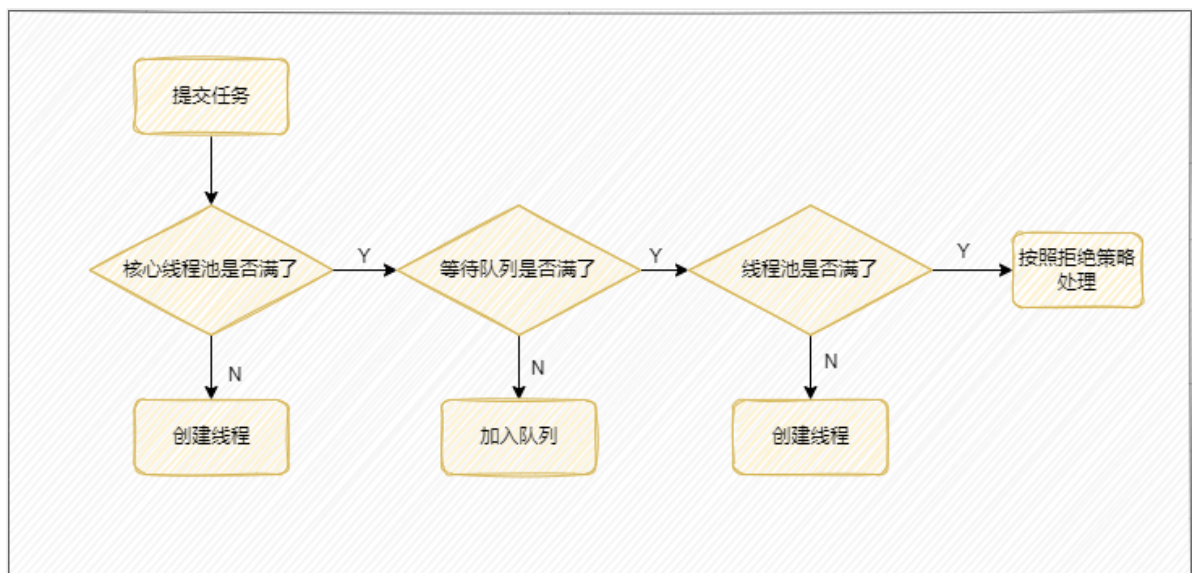
1. JVM为一个线程栈分配内存，该栈为每个线程方法调用保存一个栈帧
2. 每一栈帧由一个局部变量数组、返回值、操作数堆栈和常量池组成
3. 每个线程获得一个程序计数器，用于记录当前虚拟机正在执行的线程指令地址
4. 系统创建一个与Java线程对应的本机线程
5. 将与线程相关的描述符添加到JVM内部数据结构中
6. 线程共享堆和方法区域

创建一个线程大概需要1M左右的空间（Java8，机器规格2c8G）。可见，频繁手动创建/销毁线程的代价是非常大的。

1.2. 为什么使用线程池？

- **降低资源消耗。**通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- **提高响应速度。**当任务到达时，可以不需要等到线程创建就能立即执行。
- **提高线程的可管理性。**统一管理线程，避免系统创建大量同类线程而导致消耗完内存。

1.3. 线程池执行原理？



1. 当线程池里存活的线程数小于核心线程数 `corePoolSize` 时，这时对于一个新提交的任务，线程池会创建一个线程去处理任务。当线程池里面存活的线程数小于等于核心线程数 `corePoolSize` 时，线程池里面的线程会一直存活，就算空闲时间超过了 `keepAliveTime`，线程也不会被销毁，而是一直阻塞在那里一直等待任务队列的任务来执行。
2. 当线程池里面存活的线程数已经等于 `corePoolSize` 了，这是对于一个新提交的任务，会被放进任务队列 `workQueue` 排队等待执行。
3. 当线程池里面存活的线程数已经等于 `corePoolSize` 了，并且任务队列也满了，假设 `maximumPoolSize > corePoolSize`，这时如果再来新的任务，线程池就会继续创建新的线程来处理新的任务，知道线程数达到 `maximumPoolSize`，就不会再创建了。
4. 如果当前的线程数达到了 `maximumPoolSize`，并且任务队列也满了，如果还有新的任务过来，那就直接采用拒绝策略进行处理。默认的拒绝策略是抛出一个 `RejectedExecutionException` 异常。

1.4. 线程池参数有哪些？

ThreadPoolExecutor 的通用构造函数：

```
public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler);
```

- 1、`corePoolSize`：当有新任务时，如果线程池中线程数没有达到线程池的基本大小，则会创建新的线程执行任务，否则将任务放入阻塞队列。当线程池中存活的线程数总是大于 `corePoolSize` 时，应该考虑调大 `corePoolSize`。
- 2、`maximumPoolSize`：当阻塞队列填满时，如果线程池中线程数没有超过最大线程数，则会创建新的线程运行任务。否则根据拒绝策略处理新任务。非核心线程类似于临时借来的资源，这些线程在空闲时间超过 `keepAliveTime` 之后，就应该退出，避免资源浪费。
- 3、`BlockingQueue`：存储等待运行的任务。
- 4、`keepAliveTime`：非核心线程空闲后，保持存活的时间，此参数只对非核心线程有效。设置为0，表示多余的空闲线程会被立即终止。
- 5、`TimeUnit`：时间单位

```
TimeUnit.DAYS
TimeUnit.HOURS
TimeUnit.MINUTES
TimeUnit.SECONDS
TimeUnit.MILLISECONDS
TimeUnit.MICROSECONDS
TimeUnit.NANOSECONDS
```

6、**ThreadFactory**：每当线程池创建一个新的线程时，都是通过线程工厂方法来完成的。在 ThreadFactory 中只定义了一个方法 newThread，每当线程池需要创建新线程就会调用它。

```
public class MyThreadFactory implements ThreadFactory {
    private final String poolName;

    public MyThreadFactory(String poolName) {
        this.poolName = poolName;
    }

    public Thread newThread(Runnable runnable) {
        return new MyAppThread(runnable, poolName); //将线程池名字传递给构造函数，用于
        区分不同线程池的线程
    }
}
```

7、**RejectedExecutionHandler**：当队列和线程池都满了的时候，根据拒绝策略处理新任务。

AbortPolicy：默认的策略，直接抛出RejectedExecutionException
DiscardPolicy：不处理，直接丢弃
DiscardOldestPolicy：将等待队列队首的任务丢弃，并执行当前任务
CallerRunsPolicy：由调用线程处理该任务

1.5. 线程池大小怎么设置？

如果线程池线程数量太小，当有大量请求需要处理，系统响应比较慢，会影响用户体验，甚至会出现任务队列大量堆积任务导致OOM。

如果线程池线程数量过大，大量线程可能会同时抢占 CPU 资源，这样会导致大量的上下文切换，从而增加线程的执行时间，影响了执行效率。

CPU 密集型任务(N+1)：这种任务消耗的主要是 CPU 资源，可以将线程数设置为 **N (CPU 核心数) +1**，多出来的一个线程是为了防止某些原因导致的线程阻塞（如IO操作，线程sleep，等待锁）而带来的影响。一旦某个线程被阻塞，释放了CPU资源，而在这种情况下多出来的一个线程就可以充分利用 CPU 的空闲时间。

I/O 密集型任务(2N)：系统的大部分时间都在处理 IO 操作，此时线程可能会被阻塞，释放CPU资源，这时就可以将 CPU 交出给其它线程使用。因此在 IO 密集型任务的应用中，可以多配置一些线程，具体的计算方法：**最佳线程数 = CPU核心数 * (1/CPU利用率) = CPU核心数 * (1 + (IO耗时/CPU耗时))**，一般可设置为2N。

1.6. 线程池的类型有哪些？适用场景？

常见的线程池有 `FixedThreadPool`、`SingleThreadExecutor`、`CachedThreadPool` 和 `ScheduledThreadPool`。这几个都是 `ExecutorService` 线程池实例。

FixedThreadPool

固定线程数的线程池。任何时间点，最多只有 `nThreads` 个线程处于活动状态执行任务。

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>());
}
```

使用无界队列 `LinkedBlockingQueue`（队列容量为 `Integer.MAX_VALUE`），运行中的线程池不会拒绝任务，即不会调用 `RejectedExecutionHandler.rejectedExecution()` 方法。

`maxThreadPoolSize` 是无效参数，故将它的值设置为与 `coreThreadPoolSize` 一致。

`keepAliveTime` 也是无效参数，设置为 `0L`，因为此线程池里所有线程都是核心线程，核心线程不会被回收（除非设置了 `executor.allowCoreThreadTimeOut(true)`）。

适用场景：适用于处理CPU密集型的任务，确保CPU在长期被工作线程使用的情况下，尽可能的少的分配线程，即适用执行长期的任务。需要注意的是，`FixedThreadPool` 不会拒绝任务，**在任务比较多的时候会导致 OOM。**

SingleThreadExecutor

只有一个线程的线程池。

```
public static ExecutionService newSingleThreadExecutor() {
    return new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS, new
        LinkedBlockingQueue<Runnable>());
}
```

使用无界队列 `LinkedBlockingQueue`。线程池只有一个运行的线程，新来的任务放入工作队列，线程处理完任务就循环从队列里获取任务执行。保证顺序的执行各个任务。

适用场景：适用于串行执行任务的场景，一个任务一个任务地执行。**在任务比较多的时候也是会导致 OOM。**

CachedThreadPool

根据需要创建新线程的线程池。

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L, TimeUnit.SECONDS,
        new SynchronousQueue<Runnable>());
}
```

如果主线程提交任务的速度高于线程处理任务的速度时，`cachedThreadPool` 会不断创建新的线程。极端情况下，这样会导致耗尽 `cpu` 和内存资源。

使用没有容量的 `SynchronousQueue` 作为线程池工作队列，当线程池有空闲线程时，`SynchronousQueue.offer(Runnable task)` 提交的任务会被空闲线程处理，否则会创建新的线程处理任务。

适用场景：用于并发执行大量短期的小任务。`CachedThreadPool` 允许创建的线程数量为 `Integer.MAX_VALUE`，可能会创建大量线程，从而导致 OOM。

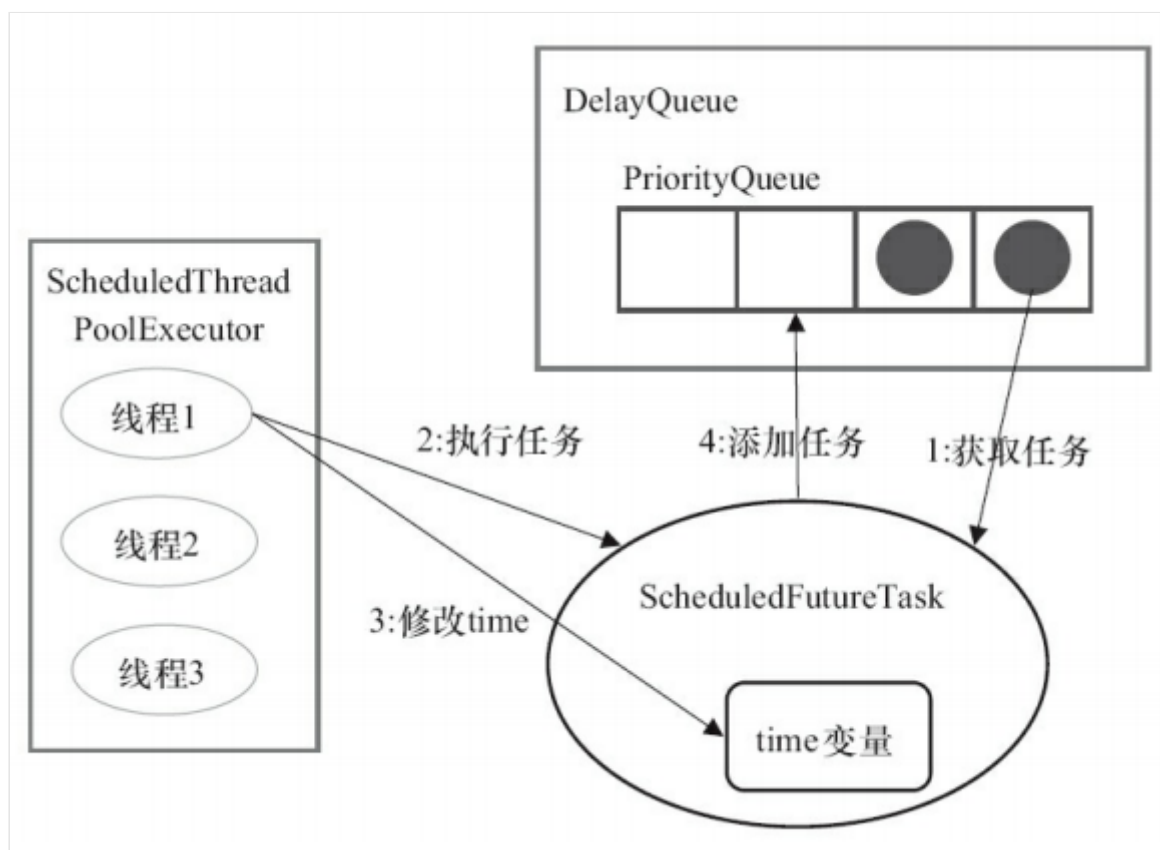
ScheduledThreadPoolExecutor

在给定的延迟后运行任务，或者定期执行任务。在实际项目中基本不会被用到，因为有其他方案选择比如 `quartz`。

使用的任务队列 `DelayQueue` 封装了一个 `PriorityQueue`，`PriorityQueue` 会对队列中的任务进行排序，时间早的任务先被执行(即 `ScheduledFutureTask` 的 `time` 变量小的先执行)，如果 `time` 相同则先提交的任务会被先执行(`ScheduledFutureTask` 的 `sequenceNumber` 变量小的先执行)。

执行周期任务步骤：

1. 线程从 `DelayQueue` 中获取已到期的 `ScheduledFutureTask` (`DelayQueue.take()`)。到期任务是指 `ScheduledFutureTask` 的 `time` 大于等于当前系统的时间；
2. 执行这个 `ScheduledFutureTask`；
3. 修改 `ScheduledFutureTask` 的 `time` 变量为下次将要被执行的时间；
4. 把这个修改 `time` 之后的 `ScheduledFutureTask` 放回 `DelayQueue` 中 (`DelayQueue.add()`)。



适用场景：周期性执行任务的场景，需要限制线程数量的场景。

2. 进程线程

进程是指一个内存中运行的应用程序，每个进程都有自己独立的一块内存空间。

线程是比进程更小的执行单位，它是在一个进程中独立的控制流，一个进程可以启动多个线程，每条线程并行执行不同的任务。

2.1. 线程的生命周期

初始(NEW): 线程被构建，还没有调用 start()。

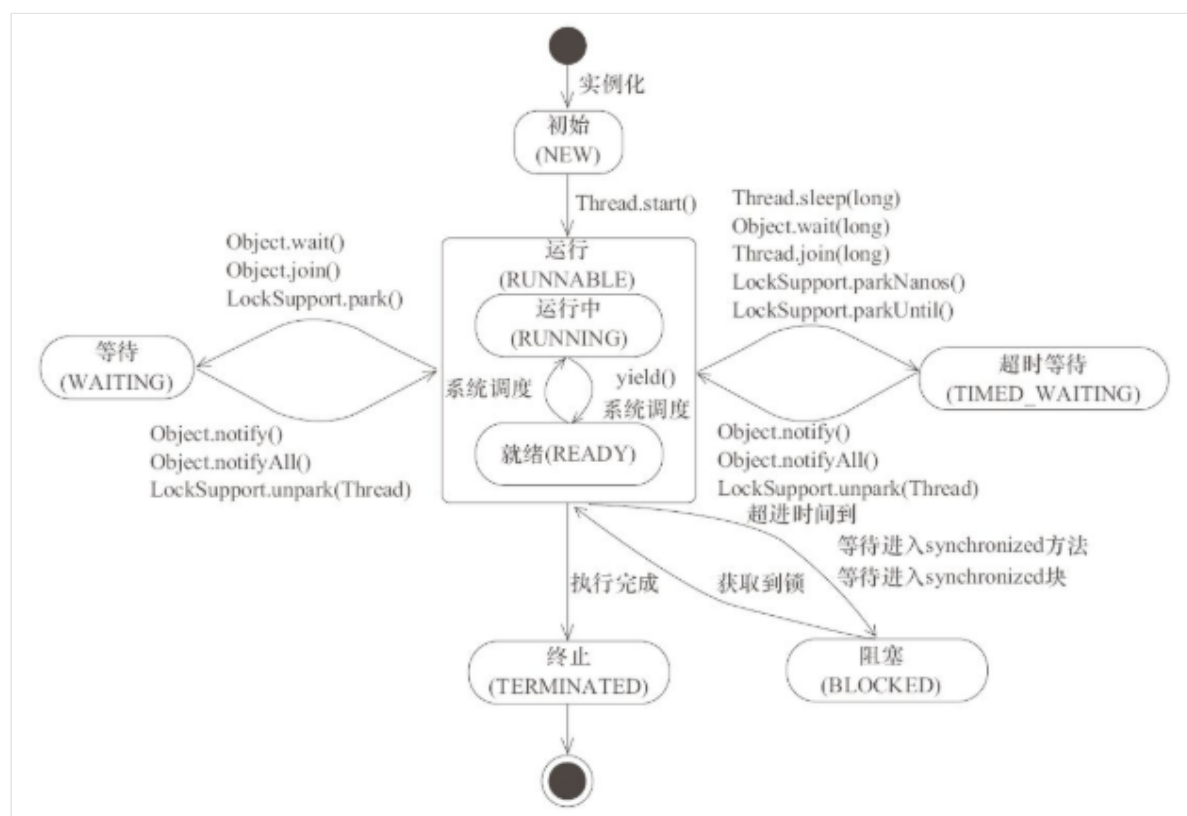
运行(RUNNABLE): 包括操作系统的就绪和运行两种状态。

阻塞(BLOCKED): 一般是被动的，在抢占资源中得不到资源，被动的挂起在内存，等待资源释放将其唤醒。线程被阻塞会释放CPU，不释放内存。

等待(WAITING): 进入该状态的线程需要等待其他线程做出一些特定动作（通知或中断）。

超时等待(TIMED_WAITING): 该状态不同于WAITING，它可以在指定的时间后自行返回。

终止(TERMINATED): 表示该线程已经执行完毕。



图片来源：Java并发编程的艺术

2.2. 讲讲线程中断?

线程中断即线程运行过程中被其他线程给打断了，它与 stop 最大的区别是：stop 是由系统强制终止线程，而线程中断则是给目标线程发送一个中断信号，如果目标线程没有接收线程中断的信号并结束线程，线程则不会终止，具体是否退出或者执行其他逻辑取决于目标线程。

线程中断三个重要的方法：

1、java.lang.Thread#interrupt

调用目标线程的 `interrupt()` 方法，给目标线程发一个中断信号，线程被打上中断标记。

2、java.lang.Thread#isInterrupted()

判断目标线程是否被中断，不会清除中断标记。

3、java.lang.Thread#interrupted

判断目标线程是否被中断，会清除中断标记。

```

private static void test2() {
    Thread thread = new Thread(() -> {
        while (true) {
            Thread.yield();

            // 响应中断
            if (Thread.currentThread().isInterrupted()) {
                System.out.println("Java技术栈线程被中断，程序退出。");
                return;
            }
        }
    });
    thread.start();
    thread.interrupt();
}

```

2.3. 创建线程有哪几种方式？

- 通过扩展 `Thread` 类来创建多线程
- 通过实现 `Runnable` 接口来创建多线程
- 实现 `Callable` 接口，通过 `FutureTask` 接口创建线程。
- 使用 `Executor` 框架来创建线程池。

继承 Thread 创建线程代码如下。run()方法是由jvm创建完操作系统级线程后回调的方法，不可以手动调用，手动调用相当于调用普通方法。

```

/**
 * @author: 程序员大彬
 * @time: 2021-09-11 10:15
 */
public class MyThread extends Thread {
    public MyThread() {
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread() + ":" + i);
        }
    }

    public static void main(String[] args) {
        MyThread mThread1 = new MyThread();
        MyThread mThread2 = new MyThread();
        MyThread myThread3 = new MyThread();
        mThread1.start();
        mThread2.start();
        myThread3.start();
    }
}

```

Runnable 创建线程代码：

```

/**
 * @author: 程序员大彬
 * @time: 2021-09-11 10:04

```

```

*/
public class RunnableTest {
    public static void main(String[] args){
        Runnable1 r = new Runnable1();
        Thread thread = new Thread(r);
        thread.start();
        System.out.println("主线程: ["+Thread.currentThread().getName()+"]");
    }
}

class Runnable1 implements Runnable{
    @Override
    public void run() {
        System.out.println("当前线程: "+Thread.currentThread().getName());
    }
}

```

实现Runnable接口比继承Thread类所具有的优势:

1. 可以避免java中的单继承的限制
2. 线程池只能放入实现Runnable或Callable类线程，不能直接放入继承Thread的类

Callable 创建线程代码:

```

/**
 * @author: 程序员大彬
 * @time: 2021-09-11 10:21
 */
public class CallableTest {
    public static void main(String[] args) {
        Callable1 c = new Callable1();

        //异步计算的结果
        FutureTask<Integer> result = new FutureTask<>(c);

        new Thread(result).start();

        try {
            //等待任务完成，返回结果
            int sum = result.get();
            System.out.println(sum);
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
}

class Callable1 implements Callable<Integer> {

    @Override
    public Integer call() throws Exception {
        int sum = 0;

        for (int i = 0; i <= 100; i++) {
            sum += i;
        }
    }
}

```

```

        return sum;
    }
}

```

使用 Executor 创建线程代码：

```

/**
 * @author: 程序员大彬
 * @time: 2021-09-11 10:44
 */
public class ExecutorsTest {
    public static void main(String[] args) {
        //获取ExecutorService实例，生产禁用，需要手动创建线程池
        ExecutorService executorService = Executors.newCachedThreadPool();
        //提交任务
        executorService.submit(new RunnableDemo());
    }
}

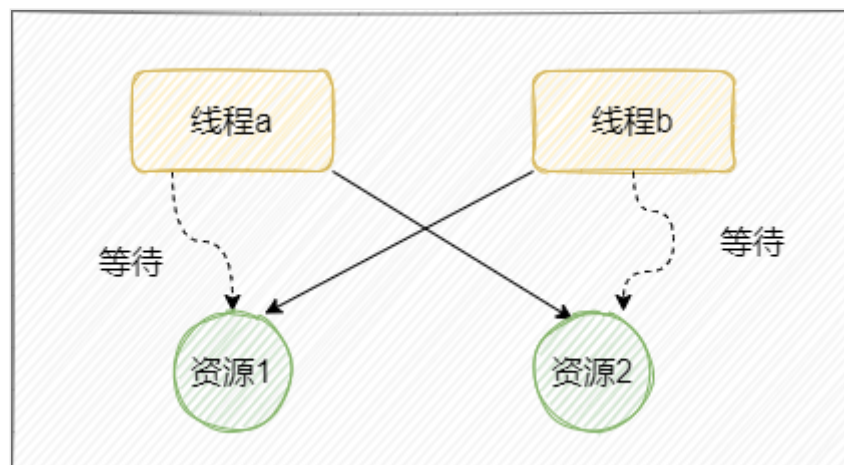
class RunnableDemo implements Runnable {
    @Override
    public void run() {
        System.out.println("大彬");
    }
}

```

2.4. 什么是线程死锁？

线程死锁是指两个或两个以上的线程在执行过程中，因争夺资源而造成的一种互相等待的现象。若无外力作用，它们都将无法推进下去。

如下图所示，线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方持有的资源，所以这两个线程就会互相等待而进入死锁状态。



下面通过例子说明线程死锁，代码来自并发编程之美。

```

public class DeadLockDemo {
    private static Object resource1 = new Object();//资源 1
    private static Object resource2 = new Object();//资源 2

    public static void main(String[] args) {
        new Thread(() -> {
            synchronized (resource1) {

```

```

        System.out.println(Thread.currentThread() + "get resource1");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread() + "waiting get
resource2");
        synchronized (resource2) {
            System.out.println(Thread.currentThread() + "get
resource2");
        }
    }
}, "线程 1").start();

new Thread() -> {
    synchronized (resource2) {
        System.out.println(Thread.currentThread() + "get resource2");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread() + "waiting get
resource1");
        synchronized (resource1) {
            System.out.println(Thread.currentThread() + "get
resource1");
        }
    }
}, "线程 2").start();
}
}

```

代码输出如下：

```

Thread[线程 1,5,main]get resource1
Thread[线程 2,5,main]get resource2
Thread[线程 1,5,main]waiting get resource2
Thread[线程 2,5,main]waiting get resource1

```

线程 A 通过 `synchronized (resource1)` 获得 resource1 的监视器锁，然后通过 `Thread.sleep(1000)`。让线程 A 休眠 1s 为的是让线程 B 得到执行然后获取到 resource2 的监视器锁。线程 A 和线程 B 休眠结束了都开始企图请求获取对方的资源，然后这两个线程就会陷入互相等待的状态，这也就产生了死锁。

2.5. 线程死锁怎么产生？怎么避免？

死锁产生的四个必要条件：

- 互斥：一个资源每次只能被一个进程使用
- 请求与保持：一个进程因请求资源而阻塞时，不释放获得的资源
- 不剥夺：进程已获得的资源，在未使用之前，不能强行剥夺
- 循环等待：进程之间循环等待着资源

避免死锁的方法：

- 互斥条件不能破坏，因为加锁就是为了保证互斥
- 一次性申请所有的资源，避免线程占有资源而且在等待其他资源
- 占有部分资源的线程进一步申请其他资源时，如果申请不到，主动释放它占有的资源
- 按序申请资源

2.6. 线程run和start的区别？

- 当程序调用 `start()` 方法，将会创建一个新线程去执行 `run()` 方法中的代码。`run()` 就像一个普通方法一样，直接调用 `run()` 的话，不会创建新线程。
- 一个线程的 `start()` 方法只能调用一次，多次调用会抛出 `java.lang.IllegalThreadStateException` 异常。`run()` 方法则没有限制。

2.7. 线程都有哪些方法？

start

用于启动线程。

getPriority

获取线程优先级，默认是5，线程默认优先级为5，如果不手动指定，那么线程优先级具有继承性，比如线程A启动线程B，那么线程B的优先级和线程A的优先级相同

setPriority

设置线程优先级。CPU会尽量将执行资源让给优先级比较高的线程。

interrupt

告诉线程，你应该中断了，具体到底中断还是继续运行，由被通知的线程自己处理。

当对一个线程调用 `interrupt()` 时，有两种情况：

1. 如果线程处于被阻塞状态（例如处于sleep, wait, join 等状态），那么线程将立即退出被阻塞状态，并抛出一个 `InterruptedException` 异常。
2. 如果线程处于正常活动状态，那么会将该线程的中断标志设置为 `true`。不过，被设置中断标志的线程可以继续正常运行，不受影响。

`interrupt()` 并不能真正的中断线程，需要被调用的线程自己进行配合才行。

join

等待其他线程终止。在当前线程中调用另一个线程的 `join()` 方法，则当前线程转入阻塞状态，直到另一个进程运行结束，当前线程再由阻塞转为就绪状态。

yield

暂停当前正在执行的线程对象，把执行机会让给相同或者更高优先级的线程。

sleep

使线程转到阻塞状态。`millis` 参数设定睡眠的时间，以毫秒为单位。当睡眠结束后，线程自动转为 `Runnable` 状态。

3. volatile底层原理

`volatile` 是轻量级的同步机制，`volatile` 保证变量对所有线程的可见性，不保证原子性。

1. 当对 `volatile` 变量进行写操作的时候，JVM会向处理器发送一条 `LOCK` 前缀的指令，将该变量所在缓存行的数据写回系统内存。
2. 由于缓存一致性协议，每个处理器通过嗅探在总线上传播的数据来检查自己的缓存是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行置为无效状态，当处理器对这个数据进行修改操作的时候，会重新从系统内存中把数据读到处理器缓存中。

来看看缓存一致性协议是什么。

缓存一致性协议：当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本，会发出信号通知其他CPU将该变量的缓存行置为无效状态，因此当其他CPU需要读取这个变量时，就会从内存重新读取。

`volatile` 关键字的两个作用：

1. 保证了不同线程对共享变量进行操作时的**可见性**，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。
2. 禁止进行**指令重排序**。

指令重排序是JVM为了优化指令，提高程序运行效率，在不影响单线程程序执行结果的前提下，尽可能地提高并行度。Java编译器会在生成指令系列时在适当的位置会插入 `内存屏障` 指令来禁止处理器重排序。插入一个内存屏障，相当于告诉CPU和编译器先于这个命令的必须先执行，后于这个命令的必须后执行。对一个volatile字段进行写操作，Java内存模型将在写操作后插入一个写屏障指令，这个指令会把之前的写入值都刷新到内存。

4. synchronized的用法有哪些？

1. **修饰普通方法：**作用于当前对象实例，进入同步代码前要获得当前对象实例的锁
2. **修饰静态方法：**作用于当前类，进入同步代码前要获得当前类对象的锁，synchronized关键字加到static 静态方法和 synchronized(class)代码块上都是给 Class 类上锁
3. **修饰代码块：**指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁

5. synchronized的作用有哪些？

原子性：确保线程互斥的访问同步代码；

可见性：保证共享变量的修改能够及时可见；

有序性：有效解决重排序问题。

6. synchronized 底层实现原理？

synchronized 同步代码块的实现是通过 `monitorenter` 和 `monitorexit` 指令，其中 `monitorenter` 指令指向同步代码块的开始位置，`monitorexit` 指令则指明同步代码块的结束位置。当执行 `monitorenter` 指令时，线程试图获取锁也就是获取 `monitor` 的持有权（monitor对象存在于每个Java对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因）。

其内部包含一个计数器，当计数器为0则可以成功获取，获取后将锁计数器设为1也就是加1。相应的在执行 `monitorexit` 指令后，将锁计数器设为0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止

synchronized 修饰的方法并没有 `monitorenter` 指令和 `monitorexit` 指令，取得代之的确实是 `ACC_SYNCHRONIZED` 标识，该标识指明了该方法是一个同步方法，JVM 通过该 `ACC_SYNCHRONIZED` 访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。

7. volatile和synchronized的区别是什么？

1. `volatile` 只能使用在变量上；而 `synchronized` 可以在类，变量，方法和代码块上。
2. `volatile` 至保证可见性；`synchronized` 保证原子性与可见性。
3. `volatile` 禁用指令重排序；`synchronized` 不会。
4. `volatile` 不会造成阻塞；`synchronized` 会。

8. ReentrantLock和synchronized区别

1. 使用synchronized关键字实现同步，线程执行完同步代码块会**自动释放锁**，而ReentrantLock需要手动释放锁。
2. synchronized是**非公平锁**，ReentrantLock可以设置为公平锁。
3. ReentrantLock上等待获取锁的线程是**可中断的**，线程可以放弃等待锁。而synchronized会无限期等待下去。
4. ReentrantLock **可以设置超时获取锁**。在指定的截止时间之前获取锁，如果截止时间到了还没有获取到锁，则返回。
5. ReentrantLock 的 tryLock() 方法**可以尝试非阻塞的获取锁**，调用该方法后立刻返回，如果能够获取则返回true，否则返回false。

9. wait()和sleep()的异同点？

相同点：

1. 它们都可以使当前线程暂停运行，把机会交给其他线程
2. 任何线程在调用wait()和sleep()之后，在等待期间被中断都会抛出 `InterruptedException`

不同点：

1. `wait()` 是Object超类中的方法；而 `sleep()` 是线程Thread类中的方法
2. 对锁的持有不同，`wait()` 会释放锁，而 `sleep()` 并不释放锁
3. 唤醒方法不完全相同，`wait()` 依靠 `notify` 或者 `notifyAll`、中断、达到指定时间来唤醒；而 `sleep()` 到达指定时间被唤醒
4. 调用 `wait()` 需要先获取对象的锁，而 `Thread.sleep()` 不用

10. Runnable和Callable有什么区别？

- Callable接口方法是 `call()`，Runnable的方法是 `run()`；
- Callable接口call方法有返回值，支持泛型，Runnable接口run方法无返回值。
- Callable接口 `call()` 方法允许抛出异常；而Runnable接口 `run()` 方法不能继续上抛异常。

11. 线程执行顺序怎么控制？

假设有T1、T2、T3三个线程，你怎样保证T2在T1执行完后执行，T3在T2执行完后执行？

可以使用**join方法**解决这个问题。比如在线程A中，调用线程B的join方法表示的意思就是：**A等待B线程执行完毕后（释放CPU执行权），在继续执行。**

代码如下：

```
public class ThreadTest {

    public static void main(String[] args) {

        Thread spring = new Thread(new SeasonThreadTask("春天"));
        Thread summer = new Thread(new SeasonThreadTask("夏天"));
        Thread autumn = new Thread(new SeasonThreadTask("秋天"));

        try
```

```

    {
        //春天线程先启动
        spring.start();
        //主线程等待线程spring执行完，再往下执行
        spring.join();
        //夏天线程再启动
        summer.start();
        //主线程等待线程summer执行完，再往下执行
        summer.join();
        //秋天线程最后启动
        autumn.start();
        //主线程等待线程autumn执行完，再往下执行
        autumn.join();
    } catch (InterruptedException e)
    {
        e.printStackTrace();
    }
}

class SeasonThreadTask implements Runnable{

    private String name;

    public SeasonThreadTask(String name){
        this.name = name;
    }

    @Override
    public void run() {
        for (int i = 1; i <4; i++) {
            System.out.println(this.name + "来了： " + i + "次");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

运行结果：

```

春天来了： 1次
春天来了： 2次
春天来了： 3次
夏天来了： 1次
夏天来了： 2次
夏天来了： 3次
秋天来了： 1次
秋天来了： 2次
秋天来了： 3次

```

12. 守护线程是什么？

守护线程是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。在 Java 中垃圾回收线程就是特殊的守护线程。

13. 线程间通信方式

1、使用 Object 类的 `wait()/notify()`。Object 类提供了线程间通信的方法：`wait()`、`notify()`、`notifyAll()`，它们是多线程通信的基础。其中，`wait/notify` 必须配合 `synchronized` 使用，`wait` 方法释放锁，`notify` 方法不释放锁。`wait` 是指在一个已经进入了同步锁的线程内，让自己暂时让出同步锁，以便其他正在等待此锁的线程可以得到同步锁并运行，只有其他线程调用了 `notify()`，`notify` 并不释放锁，只是告诉调用过 `wait()` 的线程可以去参与获得锁的竞争了，但不是马上得到锁，因为锁还在别人手里，别人还没释放，调用 `wait()` 的一个或多个线程就会解除 `wait` 状态，重新参与竞争对象锁，程序如果可以再次得到锁，就可以继续向下运行。

2、使用 `volatile` 关键字。基于 `volatile` 关键字实现线程间相互通信，其底层使用了共享内存。简单来说，就是多个线程同时监听一个变量，当这个变量发生变化的时候，线程能够感知并执行相应的业务。

3、使用UC工具类 `CountDownLatch`。jdk1.5 之后在 `java.util.concurrent` 包下提供了很多并发编程相关的工具类，简化了并发编程开发，`CountDownLatch` 基于 AQS 框架，相当于也是维护了一个线程间共享变量 `state`。

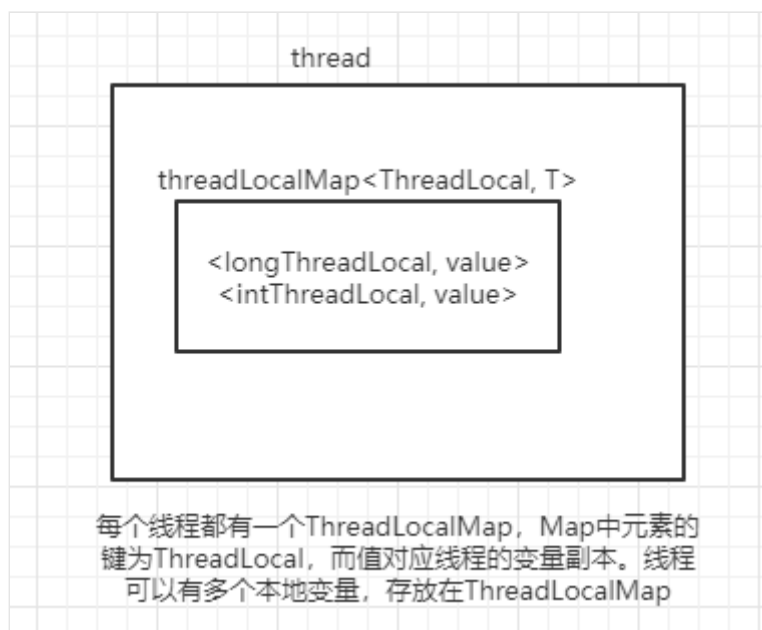
4、基于 `LockSupport` 实现线程间的阻塞和唤醒。`LockSupport` 是一种非常灵活的实现线程间阻塞和唤醒的工具，使用它不用关注是等待线程先进行还是唤醒线程先运行，但是得知道线程的名字。

14. ThreadLocal

线程本地变量。当使用 `ThreadLocal` 维护变量时，`ThreadLocal` 为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程。

14.1. ThreadLocal原理

每个线程都有一个 `ThreadLocalMap`（`ThreadLocal` 内部类），Map 中元素的键为 `ThreadLocal`，而值对应线程的变量副本。



调用 `threadLocal.set()` -->调用 `getMap(Thread)` -->返回当前线程的

`ThreadLocalMap<ThreadLocal, value>` --> `map.set(this, value)`，this是 `threadLocal` 本身。源码如下：

```
public void set(T value) {
```

```

Thread t = Thread.currentThread();
ThreadLocalMap map = getMap(t);
if (map != null)
    map.set(this, value);
else
    createMap(t, value);
}

ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}

void createMap(Thread t, T firstValue) {
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}

```

调用 `get()` --> 调用 `getMap(Thread)` --> 返回当前线程的 `ThreadLocalMap<ThreadLocal, value>` --> `map.getEntry(this)`, 返回 `value`。源码如下:

```

public T get() {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    return setInitialValue();
}

```

`threadLocals` 的类型 `ThreadLocalMap` 的键为 `ThreadLocal` 对象, 因为每个线程中可有多个 `threadLocal` 变量, 如 `longLocal` 和 `stringLocal`。

```

public class ThreadLocalDemo {
    ThreadLocal<Long> longLocal = new ThreadLocal<>();

    public void set() {
        longLocal.set(Thread.currentThread().getId());
    }
    public Long get() {
        return longLocal.get();
    }

    public static void main(String[] args) throws InterruptedException {
        ThreadLocalDemo threadLocalDemo = new ThreadLocalDemo();
        threadLocalDemo.set();
        System.out.println(threadLocalDemo.get());

        Thread thread = new Thread(() -> {
            threadLocalDemo.set();
            System.out.println(threadLocalDemo.get());
        });
    }
}

```

```
        thread.start();
        thread.join();

        System.out.println(threadLocalDemo.get());
    }
}
```

`ThreadLocal` 并不是用来解决共享资源的多线程访问问题，因为每个线程中的资源只是副本，不会共享。因此 `ThreadLocal` 适合作为线程上下文变量，简化线程内传参。

14.2. ThreadLocal内存泄漏的原因？

每个线程都有一个 `ThreadLocalMap` 的内部属性，map的key是 `ThreadLocal`，定义为弱引用，value是强引用类型。垃圾回收的时候会自动回收key，而value的回收取决于Thread对象的生命周期。一般会通过线程池的方式复用线程节省资源，这也就导致了线程对象的生命周期比较长，这样便一直存在一条强引用链的关系：`Thread` --> `ThreadLocalMap` --> `Entry` --> `Value`，随着任务的执行，value就有可能越来越多且无法释放，最终导致内存泄漏。

解决方法：每次使用完 `ThreadLocal` 就调用它的 `remove()` 方法，手动将对应的键值对删除，从而避免内存泄漏。

14.3. ThreadLocal使用场景有哪些？

场景1

`ThreadLocal` 用作保存每个线程独享的对象，为每个线程都创建一个副本，这样每个线程都可以修改自己所拥有的副本，而不会影响其他线程的副本，确保了线程安全。

这种场景通常用于保存线程不安全的工具类，典型的使用的类就是 `SimpleDateFormat`。

假如需求为500个线程都要用到 `SimpleDateFormat`，使用线程池来实现线程的复用，否则会消耗过多的内存等资源，如果我们每个任务都创建了一个 `simpleDateFormat` 对象，也就是说，500个任务对应500个 `simpleDateFormat` 对象。但是这么多对象的创建是有开销的，而且这么多对象同时存在在内存中也是一种内存的浪费。可以将`simpleDateFormat` 对象给提取了出来，变成静态变量，但是这样一来就会有线程不安全的问题。我们想要的效果是，既不浪费过多的内存，同时又想保证线程安全。此时，可以使用 `ThreadLocal`来达到这个目的，每个线程都拥有一个自己的 `simpleDateFormat` 对象。

场景2

`ThreadLocal` 用作每个线程内需要独立保存信息，以便供其他方法更方便地获取该信息的场景。每个线程获取到的信息可能都是不一样的，前面执行的方法保存了信息后，后续方法可以通过 `ThreadLocal` 直接获取到，避免了传参，类似于全局变量的概念。

比如Java web应用中，每个线程有自己单独的 `Session` 实例，就可以使用 `ThreadLocal` 来实现。

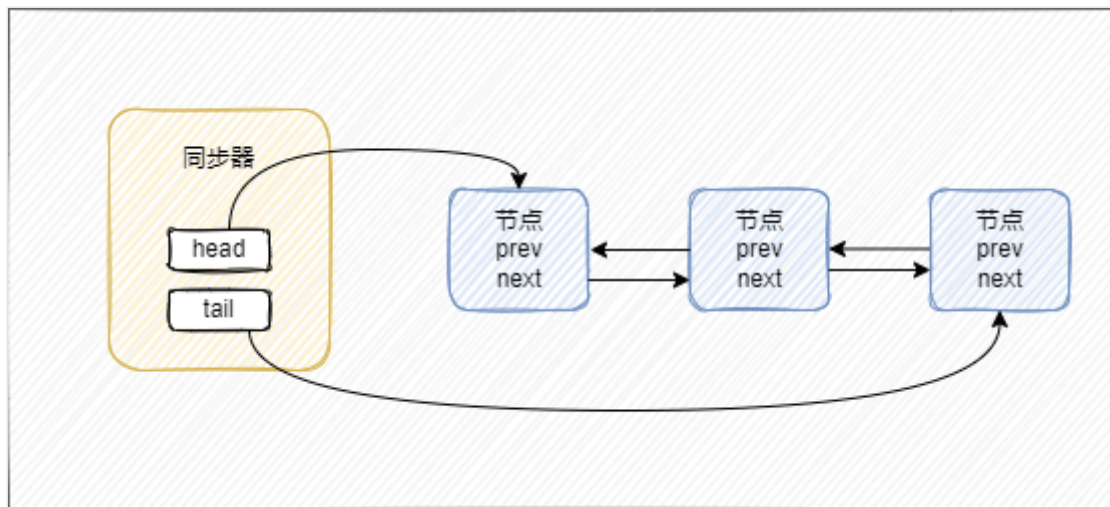
15. AQS原理

AQS，`AbstractQueuedSynchronizer`，抽象队列同步器，定义了一套多线程访问共享资源的同步器框架，许多并发工具的实现都依赖于它，如常用的 `ReentrantLock/Semaphore/CountDownLatch`。

AQS使用一个 `volatile` 的int类型的成员变量 `state` 来表示同步状态，通过CAS修改同步状态的值。当线程调用 `lock` 方法时，如果 `state` =0，说明没有任何线程占有共享资源的锁，可以获得锁并将 `state` 加1。如果 `state` 不为0，则说明有线程目前正在使用共享变量，其他线程必须加入同步队列进行等待。

```
private volatile int state;//共享变量，使用volatile修饰保证线程可见性
```

同步器依赖内部的同步队列（一个FIFO双向队列）来完成同步状态的管理，当前线程获取同步状态失败时，同步器会将当前线程以及等待状态（独占或共享）构造成为一个节点（Node）并将其加入同步队列并进行自旋，当同步状态释放时，会把首节点中的后继节点对应的线程唤醒，使其再次尝试获取同步状态。



16. ReentrantLock 是如何实现可重入性的?

ReentrantLock 内部自定义了同步器sync，在加锁的时候通过CAS算法，将线程对象放到一个双向链表中，每次获取锁的时候，检查当前维护的那个线程ID和当前请求的线程ID是否一致，如果一致，同步状态加1，表示锁被当前线程获取了多次。

源码如下：

```
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

17. 锁的分类

17.1. 公平锁与非公平锁

按照**线程访问顺序**获取对象锁。`synchronized`是非公平锁，`Lock`默认是非公平锁，可以设置为公平锁，公平锁会影响性能。

```
public ReentrantLock() {
    sync = new NonfairSync();
}

public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

17.2. 共享式与独占式锁

共享式与独占式的最主要**区别**在于：同一时刻独占式只能有一个**线程**获取同步状态，而共享式在同一时刻可以有多个线程获取同步状态。例如读操作可以有多个线程同时进行，而写操作同一时刻只能有一个线程进行写操作，其他操作都会被阻塞。

17.3. 悲观锁与乐观锁

悲观锁，**每次访问资源都会加锁**，执行完同步代码释放锁，`synchronized`和`ReentrantLock`属于悲观锁。

乐观锁，不会锁定资源，所有的线程都能访问并修改同一个资源，如果没有冲突就修改成功并退出，否则就会继续循环尝试。乐观锁最常见的实现就是**CAS**。

适用场景：

- 悲观锁适合**写操作多**的场景。
- 乐观锁适合**读操作多**的场景，不加锁可以提升读操作的性能。

18. 乐观锁有什么问题？

乐观锁避免了悲观锁独占对象的问题，提高了并发性能，但它也有缺点：

- 乐观锁只能保证一个**共享变量**的原子操作。
- 长时间自旋可能导致**开销大**。假如CAS长时间不成功而一直自旋，会给CPU带来很大的开销。
- **ABA问题**。CAS的原理是通过比对内存值与预期值是否一样而判断内存值是否被改过，但是会有以下问题：假如内存值原来是A，后来被一条线程改为B，最后又被改成了A，则CAS认为此内存值并没有发生改变。可以引入版本号解决这个问题，每次变量更新都把版本号加一。

19. 什么是CAS？

CAS全称 **Compare And Swap**，比较与交换，是乐观锁的主要实现方式。CAS在不使用锁的情况下实现多线程之间的变量同步。`ReentrantLock`内部的AQS和原子类内部都使用了CAS。

CAS算法涉及到三个操作数：

- 需要读写的内存值V。
- 进行比较的值A。
- 要写入的新值B。

只有当V的值等于A时，才会使用原子方式用新值B来更新V的值，否则会继续重试直到成功更新值。

以 `AtomicInteger` 为例，`AtomicInteger` 的 `getAndIncrement()` 方法底层就是CAS实现，关键代码是 `compareAndSwapInt(obj, offset, expect, update)`，其含义就是，如果 `obj` 内的 `value` 和 `expect` 相等，就证明没有其他线程改变过这个变量，那么就更新它为 `update`，如果不相等，那就会继续重试直到成功更新值。

20. CAS存在的问题？

CAS 三大问题：

1. **ABA问题**。CAS需要在操作值的时候检查内存值是否发生变化，没有发生变化才会更新内存值。但是如果内存值原来是A，后来变成了B，然后又变成了A，那么CAS进行检查时会发现值没有发生变化，但是实际上是有变化的。ABA问题的解决思路就是在变量前面添加版本号，每次变量更新的时候都把版本号加一，这样变化过程就从 `A-B-A` 变成了 `1A-2B-3A`。

JDK从1.5开始提供了`AtomicStampedReference`类来解决ABA问题，原子更新带有版本号的引用类型。

2. **循环时间长开销大**。CAS操作如果长时间不成功，会导致其一直自旋，给CPU带来非常大的开销。
3. **只能保证一个共享变量的原子操作**。对一个共享变量执行操作时，CAS能够保证原子操作，但是对多个共享变量操作时，CAS是无法保证操作的原子性的。

Java从1.5开始JDK提供了`AtomicReference`类来保证引用对象之间的原子性，可以把多个变量放在一个对象里来进行CAS操作。

21. 并发工具

在JDK的并发包里提供了几个非常有用的并发工具类。`CountDownLatch`、`CyclicBarrier`和`Semaphore`工具类提供了一种并发流程控制的手段。

21.1. CountDownLatch

`CountDownLatch`用于某个线程等待其他线程**执行完任务**再执行，与`thread.join()`功能类似。常见的应用场景是开启多个线程同时执行某个任务，等到所有任务执行完再执行特定操作，如汇总统计结果。

```
public class CountDownLatchDemo {
    static final int N = 4;
    static CountDownLatch latch = new CountDownLatch(N);

    public static void main(String[] args) throws InterruptedException {

        for(int i = 0; i < N; i++) {
            new Thread(new Thread1()).start();
        }

        latch.await(1000, TimeUnit.MILLISECONDS); //调用await()方法的线程会被挂起，它会等待直到count值为0才继续执行；等待timeout时间后count值还没变为0的话就会继续执行
        System.out.println("task finished");
    }

    static class Thread1 implements Runnable {

        @Override
        public void run() {
            try {
                System.out.println(Thread.currentThread().getName() + "starts working");
                Thread.sleep(1000);
            } catch (InterruptedException e) {}
        }
    }
}
```

```

        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            latch.countDown();
        }
    }
}
}
}

```

运行结果:

```

Thread-0 starts working
Thread-1 starts working
Thread-2 starts working
Thread-3 starts working
task finished

```

21.2. CyclicBarrier

CyclicBarrier(同步屏障), 用于一组线程互相等待到某个状态, 然后这组线程再**同时**执行。

```

public CyclicBarrier(int parties, Runnable barrierAction) {
}
public CyclicBarrier(int parties) {
}

```

参数parties指让多少个线程或者任务等待至某个状态; 参数barrierAction为当这些线程都达到某个状态时会执行的内容。

```

public class CyclicBarrierTest {
    // 请求的数量
    private static final int threadCount = 10;
    // 需要同步的线程数量
    private static final CyclicBarrier cyclicBarrier = new CyclicBarrier(5);

    public static void main(String[] args) throws InterruptedException {
        // 创建线程池
        ExecutorService threadPool = Executors.newFixedThreadPool(10);

        for (int i = 0; i < threadCount; i++) {
            final int threadNum = i;
            Thread.sleep(1000);
            threadPool.execute(() -> {
                try {
                    test(threadNum);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } catch (BrokenBarrierException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            });
        }
        threadPool.shutdown();
    }
}

```

```

    }

    public static void test(int threadnum) throws InterruptedException,
        BrokenBarrierException {
        System.out.println("threadnum:" + threadnum + "is ready");
        try {
            /**等待60秒，保证子线程完全执行结束*/
            cyclicBarrier.await(60, TimeUnit.SECONDS);
        } catch (Exception e) {
            System.out.println("-----CyclicBarrierException-----");
        }
        System.out.println("threadnum:" + threadnum + "is finish");
    }
}

```

运行结果如下，可以看出CyclicBarrier是可以重用的：

```

threadnum:0is ready
threadnum:1is ready
threadnum:2is ready
threadnum:3is ready
threadnum:4is ready
threadnum:4is finish
threadnum:3is finish
threadnum:2is finish
threadnum:1is finish
threadnum:0is finish
threadnum:5is ready
threadnum:6is ready
...

```

当四个线程都到达barrier状态后，会从四个线程中选择一个线程去执行Runnable。

21.3. CyclicBarrier和CountDownLatch区别

CyclicBarrier 和 CountDownLatch 都能够实现线程之间的等待。

CountDownLatch用于某个线程等待其他线程**执行完任务**再执行。CyclicBarrier用于一组线程互相等待到某个状态，然后这组线程再**同时**执行。CountDownLatch的计数器只能使用一次，而CyclicBarrier的计数器可以使用reset()方法重置，可用于处理更为复杂的业务场景。

21.4. Semaphore

Semaphore类似于锁，它用于控制同时访问特定资源的线程数量，控制并发线程数。

```

public class SemaphoreDemo {
    public static void main(String[] args) {
        final int N = 7;
        Semaphore s = new Semaphore(3);
        for(int i = 0; i < N; i++) {
            new Worker(s, i).start();
        }
    }

    static class Worker extends Thread {

```

```

        private Semaphore s;
        private int num;
        public Worker(Semaphore s, int num) {
            this.s = s;
            this.num = num;
        }

        @Override
        public void run() {
            try {
                s.acquire();
                System.out.println("worker" + num + " using the machine");
                Thread.sleep(1000);
                System.out.println("worker" + num + " finished the task");
                s.release();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

运行结果如下，可以看出并非按照线程访问顺序获取资源的锁，即

```

worker0 using the machine
worker1 using the machine
worker2 using the machine
worker2 finished the task
worker0 finished the task
worker3 using the machine
worker4 using the machine
worker1 finished the task
worker6 using the machine
worker4 finished the task
worker3 finished the task
worker6 finished the task
worker5 using the machine
worker5 finished the task

```

22. 原子类

22.1. 基本类型原子类

使用原子的方式更新基本类型

- AtomicInteger：整型原子类
- AtomicLong：长整型原子类
- AtomicBoolean：布尔型原子类

AtomicInteger 类常用的方法：

```
public final int get() //获取当前的值
public final int getAndSet(int newValue)//获取当前的值，并设置新的值
public final int getAndIncrement()//获取当前的值，并自增
public final int getAndDecrement() //获取当前的值，并自减
public final int getAndAdd(int delta) //获取当前的值，并加上预期的值
boolean compareAndSet(int expect, int update) //如果输入的数值等于预期值，则以原子方式
将该值设置为输入值（update）
public final void lazySet(int newValue)//最终设置为newValue,使用 lazySet 设置之后可能
导致其他线程在之后的一小段时间内还是可以读到旧的值。
```

AtomicInteger 类主要利用 CAS (compare and swap) 保证原子操作，从而避免加锁的高开销。

22.2. 数组类型原子类

使用原子的方式更新数组里的某个元素

- AtomicIntegerArray: 整形数组原子类
- AtomicLongArray: 长整形数组原子类
- AtomicReferenceArray: 引用类型数组原子类

AtomicIntegerArray 类常用方法:

```
public final int get(int i) //获取 index=i 位置元素的值
public final int getAndSet(int i, int newValue)//返回 index=i 位置的当前的值，并将其
设置为新值: newValue
public final int getAndIncrement(int i)//获取 index=i 位置元素的值，并让该位置的元素自
增
public final int getAndDecrement(int i) //获取 index=i 位置元素的值，并让该位置的元素自
减
public final int getAndAdd(int i, int delta) //获取 index=i 位置元素的值，并加上预期的
值
boolean compareAndSet(int i, int expect, int update) //如果输入的数值等于预期值，则以
原子方式将 index=i 位置的元素值设置为输入值（update）
public final void lazySet(int i, int newValue)//最终 将index=i 位置的元素设置为
newValue,使用 lazySet 设置之后可能导致其他线程在之后的一小段时间内还是可以读到旧的值。
```

22.3. 引用类型原子类

- AtomicReference: 引用类型原子类
- AtomicStampedReference: 带有版本号的引用类型原子类。该类将整数值与引用关联起来，可用于解决原子的更新数据和数据的版本号，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。
- AtomicMarkableReference: 原子更新带有标记的引用类型。该类将 boolean 标记与引用关联起来

23. 为什么要使用Executor线程池框架呢？

- 每次执行任务都通过new Thread()去创建线程，比较消耗性能，创建一个线程是比较耗时、耗资源的
- 调用new Thread()创建的线程缺乏管理，可以无限制的创建，线程之间的相互竞争会导致过多占用系统资源而导致系统瘫痪
- 直接使用new Thread()启动的线程不利于扩展，比如定时执行、定期执行、定时定期执行、线程中断等都不好实现

24. 如何停止一个正在运行的线程？

1. 使用共享变量的方式。共享变量可以被多个执行相同任务的线程用来作为是否停止的信号，通知停止线程的执行。
2. 使用interrupt方法终止线程。当一个线程被阻塞，处于不可运行状态时，即使主程序中将该线程的共享变量设置为true，但该线程此时根本无法检查循环标志，当然也就无法立即中断。这时候可以使用Thread提供的interrupt()方法，因为该方法虽然不会中断一个正在运行的线程，但是它可以使一个被阻塞的线程抛出一个中断异常，从而使线程提前结束阻塞状态。

25. 什么是Daemon线程？

后台(daemon)线程，是指在程序运行的时候在后台提供一种通用服务的线程，并且这个线程并不属于程序中不可或缺的部分。因此，当所有的非后台线程结束时，程序也就终止了，同时会杀死进程中的所有后台线程。反过来说，只要有任何非后台线程还在运行，程序就不会终止。必须在线程启动之前调用setDaemon()方法，才能把它设置为后台线程。

注意：后台进程在不执行finally子句的情况下就会终止其run()方法。

比如：JVM的垃圾回收线程就是Daemon线程，Finalizer也是守护线程。

26. SynchronizedMap和ConcurrentHashMap有什么区别？

SynchronizedMap一次锁住整张表来保证线程安全，所以每次只能有一个线程来访问map。

JDK1.8 ConcurrentHashMap采用CAS和synchronized来保证并发安全。数据结构采用数组+链表/红黑二叉树。synchronized只锁定当前链表或红黑二叉树的首节点，支持并发访问、修改。另外ConcurrentHashMap使用了一种不同的迭代方式。当iterator被创建后集合再发生改变就不再是抛出ConcurrentModificationException，取而代之的是在改变时new新的数据而不影响原有的数据，iterator完成后再将头指针替换为新的数据，这样iterator线程可以使用原来老的数据，而写线程也可以并发的完成改变。

27. 怎么判断线程池的任务是不是执行完了？

有几种方法：

- 1、使用线程池的原生函数isTerminated();

executor提供一个原生函数isTerminated()来判断线程池中的任务是否全部完成。如果全部完成返回true，否则返回false。

- 2、使用重入锁，维持一个公共计数。

所有的普通任务维持一个计数器，当任务完成时计数器加一（这里要加锁），当计数器的值等于任务数时，这时所有的任务已经执行完毕了。

- 3、使用CountDownLatch。

它的原理跟第二种方法类似，给CountDownLatch一个计数值，任务执行完毕后，调用countDown()执行计数值减一。最后执行的任务在调用方法的开始调用await()方法，这样整个任务会阻塞，直到这个计数值为零，才会继续执行。

这种方式的缺点就是需要提前知道任务的数量。

- 4、submit向线程池提交任务，使用Future判断任务执行状态。

使用submit向线程池提交任务与execute提交不同，submit会有Future类型的返回值。通过future.isDone()方法可以知道任务是否执行完成。

28. 什么是Future？

在并发编程中，不管是继承thread类还是实现runnable接口，都无法保证获取到之前的执行结果。通过实现Callback接口，并用Future可以来接收多线程的执行结果。

Future表示一个可能还没有完成的异步任务的结果，针对这个结果可以添加Callback以便在任务执行成功或失败后作出相应的操作。

举个例子：比如去吃早点时，点了包子和凉菜，包子需要等3分钟，凉菜只需1分钟，如果是串行的一个执行，在吃上早点的时候需要等待4分钟，但是因为你在等包子的时候，可以同时准备凉菜，所以在准备凉菜的过程中，可以同时准备包子，这样只需要等待3分钟。Future就是后面这种执行模式。

Future接口主要包括5个方法：

1. get()方法可以当任务结束后返回一个结果，如果调用时，工作还没有结束，则会阻塞线程，直到任务执行完毕
2. get(long timeout, TimeUnit unit)做多等待timeout的时间就会返回结果
3. cancel(boolean mayInterruptIfRunning)方法可以用来停止一个任务，如果任务可以停止（通过mayInterruptIfRunning来进行判断），则可以返回true，如果任务已经完成或者已经停止，或者这个任务无法停止，则会返回false。
4. isDone()方法判断当前方法是否完成
5. isCancel()方法判断当前方法是否取消



扫码关注我



微信搜索



程序员大彬

公众号后台回复【面试】获取面试手册
PDF最新版

JVM重要知识点&高频面试题是我的[知识星球](#)内部专属资料，已经整理到java面试手册完整版。

Java面试手册

+

保存并发布于：2023-01-01 23:29:49

☆ 分享 ...

标题2 24px B I U

大纲

11 什么是双亲委派模型？
为什么需要双亲委派模型？
什么是类加载器，类加载器有哪些？
类的实例化顺序？
如何判断一个对象是否存在？
可作为GC Roots的对象有哪些？
什么情况下类会被卸载？
强引用、软引用、弱引用、虚引用...
GC是什么？为什么要GC？
Minor GC 和 Full GC 的区别？
内存的分配策略？
Full GC 的触发条件？
垃圾回收算法有哪些？
有哪些垃圾回收器？
常用的 JVM 调优的命令都有哪些？
对象头了解吗？
Object o = new Object() 占用多少...
main方法执行过程
对象创建过程
如何排查 OOM 的问题？
什么是内存溢出和内存泄露？

类文件

类加载器
class loader

运行时数据区

方法区

虚拟机栈

本地方法栈

堆

程序计数器

执行引擎

本地库接口

本地方法库

程序计数器

线程私有的，作为当前线程的行号指示器，用于记录当前虚拟机正在执行的线程

面试准备

面试题&重要知识点

Java

数据库

常见框架

消息队列

计算机基础

分布式

开发工具

系统设计

海量数据场景高频面试题

面经篇

校招分享

如何设计一个抢红包系统	01-11 22:31
权限系统设计方案	01-10 23:51
实时订阅推送设计与实现	01-04 16:21
如何用 Redis 统计用户访问量？	01-05 00:09
扫码登录原理	01-11 23:20
订单30分钟未支付自动取消怎么实现？	01-02 14:29
如何设计一个短链系统？	01-03 00:17
秒杀系统	01-02 14:27
秒杀系统设计的5个要点	01-04 16:22
微信红包后台系统设计	01-02 14:27
单点登录（SSO）的设计与实现	01-02 14:26
如何设计一个优惠券系统？	01-02 14:26
海量数据场景高频面试题	
面经篇	
校招分享	

除了Java面试手册完整版之外，星球还有很多其他**优质资料**，比如包括Java项目、进阶知识、实战经验总结、优质书籍、笔试面试资源等等。

👉 优质学习资源汇总：

- 1、阿里巴巴技术团队的实战经验总结，链接：🔗 <https://t.zsxq.com/08TUWuPRg>
- 2、Tomcat架构解析、性能优化、源码解读：🔗 <https://t.zsxq.com/09XZgyC7z>
- 3、尚学堂2022版全套的java学习视频（300集），链接：🔗 <https://t.zsxq.com/08NfkBCup>
- 4、并发编程视频教程，包含：线程基础、线程协作、并发工具类、锁、AQS、线程池、实战项目等：🔗 <https://t.zsxq.com/09vF2onhe>
- 5、LeetCode刷题笔记（包含C/C++、Java、Go三个版本），链接：🔗 <https://t.zsxq.com/080mxtafG>
- 6、尚硅谷Java学科全套教程（总207.77GB），链接：🔗 <https://t.zsxq.com/0807PC4s4>
- 7、自动动手写Java虚拟机PDF，链接：🔗 <https://t.zsxq.com/08jBRPI5a>
- 8、字节大佬总结的Java面试资料，涵盖Java开发常见的知识点，链接：🔗 <https://t.zsxq.com/08UxPuP34>
- 9、阿里的算法笔试题精选合集，涵盖70+算法题目、大厂笔试考点，链接：🔗 <https://t.zsxq.com/082R1dqFF>
- 10、Java工程师进阶知识大全，包含高并发架构、分布式架构，链接：🔗 <https://t.zsxq.com/08j7WWUZn>
- 11、MySQL优质教程，包含：6大范式讲解、7大日志剖析、7大SQL性能分析工具等内容，链接：🔗 <https://t.zsxq.com/08d6j44JK>

- 12、阿里巴巴微服务治理白皮书，高效构建完整的微服务治理体系。链接：<https://t.zsxq.com/09fkXNKk>
- 13、RocketMQ 从入门到实战。链接：<https://t.zsxq.com/09MecUORI>
- 14、阿里巴巴性能调优实战手册，包含Java编程性能调优、多线程性能调优、JVM性能监测及调优、设计模式调优、数据库性能调优、实战演练六大模块。链接：<https://t.zsxq.com/09gdKMpY8>
- 15、Java面试真题，跟面试手册内容互补，建议准备面试的手两份一起过一遍！链接：<https://t.zsxq.com/09rVOpWNh>
- 16、畅购商城视频教程（含源码）。链接：<https://t.zsxq.com/09LOcfEDB>
- 17、面试题合集，包含技术和非技术问题，如Dubbo面试题、设计模式面试题、Netty面试题、Tomcat面试题、Nginx面试题、HR面试常见问题、英语面试题等。链接：<https://t.zsxq.com/09G34Sf16>
- 18、精选1000套简历模板，超实用！链接：<https://t.zsxq.com/096963tQt>
- 19、马士兵老师的Java基础视频教程：<https://t.zsxq.com/09twuX7b5>
- 20、MySQL45讲专栏：<https://t.zsxq.com/09V2U7K2A>
- 21、分布式优惠券系统：<https://t.zsxq.com/09ktzHUOZ>
- 22、mpvue课程小程序全栈开发：<https://t.zsxq.com/09aexZzl8>
- 22、微信小程序入门与实战 常用组件API开发技巧项目实战：<https://t.zsxq.com/09aexZzl8>
- 23、尚硅谷最新版尚筹网Java项目实战
<https://t.zsxq.com/094UkGCUy>

- 24、Redis优质教程：<https://t.zsxq.com/090gAzWEx>
- 25、JVM调优&多线程的教程：<https://t.zsxq.com/09u1h71vm>
- 26、Java架构师-技术专家的资料：<https://t.zsxq.com/090oV3kz2>
- 27、SSO单点登录原理：<https://t.zsxq.com/096U2VUJB>
- 28、3天全面深入学习zookeeper（含源码资料）<https://t.zsxq.com/09tijiPEu>
- 29、SpringBoot+SpringCloud全彩指南（终极版），链接：<https://t.zsxq.com/08DEQYliY>
- 30、Mybatis视频教程：<https://t.zsxq.com/09r81SkuK>
- 31、《漫画Java》PDF版本，链接：<https://t.zsxq.com/09L69kPDO>
- 32、Nginx优质视频教程：<https://t.zsxq.com/09gQBCuuF>

专属一对一的提问答疑，帮你解答各种疑难问题，包括自学Java路线、职业规划、面试问题等等。大彬会优先解答球友的问题。



大彬

2023-01-02 20:33

收进专栏

...

Alvin_Xu 提问： 本人是19年毕业至今工作了3年半左右，一直在同一个岗位上，近段时间因为公司项目组资金不足被调岗了，新岗位和自己的技术栈不太匹配，觉得如果一直变动会影响自己的职业发展。

这么些年来一直有点混吃等死，**想在现在开始寻求改变**，希望通过学习，让自己有所变化。不过对于学习方式以及学习路线**很是迷茫**，希望大彬老师可以帮忙解惑，非常感谢。

工作三年半，有点迷茫，想要寻求改变！

技术栈：

有一定的Java基础，对JVM原理，体系结构，垃圾回收机制等有部分了解

熟悉Scala&Spark开发，了解部署spark server项目及脚本

掌握spring, springCloud微服务架构开发，

了解Elastic search,Mule gateway, Feign等的使用

熟悉Jenkins, Kubernetes, PCF, G3等devops部署及调试

熟悉AppDynamic, patrol, LWM等程序监控平台的使用及配置

了解NexusIQ,Checkmarx,sonar等代码质量监控平台的使用及调优

熟悉sqlServer及Mongodb数据库

我一直都是认为，在现在这个就业环境下，“专才”的竞争力是要大于“全才”的，专注一个方向，对你的职业发展更为有利。

从你的技术栈来看，相对还是偏“杂”一些，Java、大数据、devops等都有涉及。而且现在还有调岗的可能，我建议



大彬

2023-01-10 21:41

收进专栏

...

🍌 提问： 想请教一下大彬，对于java开发和大数据开发的明年秋招情况的预测。java岗位多卷度也高，每家公司基本都有相关的职位，大数据开发的话培训班比岗位多，只有大公司和数据公司会有这类岗位。24年秋招的话**选择哪个方向会比较合适呢？**

岗位方向选择：Java还是大数据？

建议选大数据吧。就这几年校招来看，大数据岗位拿offer的难度相比Java还是比较小一些的。而且距离24年秋招还有一年多时间，转大数据完全来得及，而且有了Java基础，再来学大数据，应该会比较快入门。

再说下大数据和后端的差异。大数据门槛比Java高，除了熟悉数据库的操作之外，还要学习大数据整个生态，需要会分布式、数仓、数据分析统计等知识。因为大数据的学习门槛比Java高，所以市场上培训大数据的相比Java会少一些，竞争也相对小，没有Java那么卷。

另外，大数据薪资总体比Java开发高一些（同一家公司同一级别，普通开发岗比大数据开发薪资会少一点），这也是大数据方向的一个优势。

不过呢，小点的公司，可能没有大数据的需求，毕竟业务量不大（小公司通常也不太建议去，坑多）。



2022-12-27 19:29

收进专栏

...

麻烦问一下，遇到一个面试题是：描述最能代表自己编程能力的代码，解释为什么这段代码提现了你的编程能力。这种题怎么答





2人赞过

举了两个例子，帮助球友解答疑惑



大彬：这种可以结合自己平时做的一些项目来讲。
比如说有一些比较复杂的功能，你是如何去实现的，像涉及很多状态流转的，使用了状态机去处理，通过"动作"来进行状态的变更。
或者项目中有很多相似的实现逻辑，把这些相似的逻辑抽象出来，做成公共组件，这也是一种编程能力的体现。
当然还有很多，上面只是两个简答的例子，具体可以结合你自身的项目去思考、挖掘，适当包装一下也是没问题的。

2022-12-27 21:43



大彬：可以从背景（为什么这么做）、实现原理（怎么实现的）、收获（学到了什么）这三个方面来回答

2022-12-27 21:52



回复 大彬：请问工作中如何应用的OOP原则怎么答啊

2022-12-28 13:07

大彬 回复 ：面向对象编程有六大原则：对象单一职责、里式替换原则、迪米特法则、开闭原则、依赖倒置、接口隔离。

以对象单一职责为例，在设计类的时候，必须职责明确，比如商品类，里面相关的属性和方法都必须跟商品相关，不能出现订单等不相关的内容。

其他原则也是类似，可以看看优质专栏-Java基础部分，有关于面向对象编六大原则的介绍

2022-12-28 14:21

另外星球还提供**简历指导、修改服务**，大彬已经帮**90+**个小伙伴修改了简历，相对还是比较有经验的。



求职意向: Java



生日: 1999.11 籍贯: 贵州三穗 电话: 1 学历: 本科学士 政治面貌: 团员 邮箱: 20 .com

教育背景 Education

2019.09-2023.06

天津工业大学

软件工程

主修课程: 高级程序语言设计、数据库原理及应用、数据结构、算法设计与分析、计算机组成原理、操作系统、计算机网络、编译原理、软件系统分析、软件质量测试等。

校园经历: 在校期间任大创信息部副部长, 负责创业就业信息收集整理, 微信公众号的维护和运营。在校期间注重实践, 参加技术学习型社团, 参与过许多小项目, 曾多次在软件实训中获取优异成绩。

项目经历 Project

2021.12-2022.03

企业网盘系统

项目组长、后端开发

项目简介: 基于 Springboot+Dubbo 开发的, 提供文件管理上传下载共享在线编辑的基础网盘。

● 基于 Redis 做分布式锁, 实现文件切块上传功能;

● WebSocket+Netty 做消息推送, 实现文件共享消息提示;

● Elasticsearch 做全文检索引擎, 实现大量文件的检索;

● HDFS 做分布式文件系统, 实现分布式文件存储。

2022.05-2023.07

微服务电商系统

后端开发

项目简介: SpringCloud 开发的电商系统, 集成了监控中心、网关等系统功能。

- 使用 RabbitMq 做消息队列, 实现延迟消息;
- 基于 SpringSecurity 做登录认证和权限控制;
- 使用 Spring 的 AOP, 添加切面实现接口日志记录;
- Docker 做应用容器引擎, Kubernetes 做应用容器管理平台;

专业技能 Expertise

1. 熟练掌握 java 语法, 和 IO、多线程、反射、异常及一些常用类, 掌握 JVM 原理;
2. 熟悉 Spring、SpringMVC、MyBatis、Springboot, 了解 SpringCloud 等开发框架;
3. 熟悉 MySQL 数据库, 具备一定的 SQL 优化能力; 熟悉 Redis 非关系型数据库;
4. 熟悉 HTML、CSS、JavaScript 等基础前端技术, 了解 Vue 等前端开发框架;
5. 熟练使用 Linux 操作系统; 熟悉常用的设计模式; 熟悉一些开发中常用中间件;
6. 具备一定的需求分析和系统设计能力, 较强的逻辑分析和独立解决问题能力。

自我评价 Evaluation

本人热爱 JAVA 开发工作, 注重实干, 熟悉 SpringBoot 开发, 具备良好的编码规范; 熟悉关系型数据库, 熟练使用 SQL 语言, 掌握基本的 SQL 优化; 服从安排、工作积极主动, 责任心与上进心较强, 具备良好的学习能力与合作的团队精神。

主修课程不用写这么多, 甚至可以不用写, 留点版面给其他内容

感觉似乎没有因果关系。可以描述清晰点

消息推送具体描述下怎么实现的

HDFS面试官应该会问, 思考下会问什么问题

延时消息怎么实现的

项目经历这块如果有数据支撑, 会更好, 比如一些慢查询优化, 从xx秒到xx毫秒等



18235... | 78386...qq.com
CSDN : https://blog.csdn.net/w...n_819295
求职意向：后端开发工程师 (北京)



实习经历

建议实习经历写实习项目，项目经历写自己的项目，一般都是这么写的

京东零售 - 市场营销与用户增长部 - 产品研发部

2021.12 - 2022.06

专业技能

1. 熟练运用 **JAVASE**、**JAVAAE** 主流技术；
2. 掌握 **Java** 基础，熟悉面向对象、集合、多线程、锁、I/O模型；
3. 对 **JVM** 原理有初步的了解，包括内存模型、垃圾回收机制以及垃圾回收算法；
4. 深入理解 **Redis** 数据类型和内部实现、熟悉其使用场景、持久化、淘汰策略；
5. 熟练使用 **MySQL**，熟悉MySQL索引、事务、锁机制、SQL调优并在项目中实际运用；
6. 熟悉 **MyBatis**、**Spring**、**SpringMVC**、**SpringBoot** 等主流开发框架；
7. 熟练使用**Kafka**，熟悉Kafka基本结构、消费者组；
8. 熟悉**Linux**操作系统，**Nginx** 等技术；
9. 了解常用的数据结构和算法；

专业技能可以补充分布式、微服务相关的（如果不熟悉建议了，面试大概率会问到）

项目经历

官旗OMS

工作描述，可以按照三个步骤展开：1.使用什么技术；2.做了什么事情；3.达成什么效果。

2021.12 - 2022.06

项目背景：京东在抖音开设官店为了方便管理订单并履约发货，需要在京东商城与多平台之间建立一个中枢。

工作描述：参与了V1.0、V1.1、V1.2迭代需求的开发。在 V1.0 需求中主要负责拉取抖音平台订单消息，对用户的创建订单、支付订单、风控订单等进行对接，解决了mq的重复消费。在 V1.1 需求中对用户的敏感信息进行了加密处理，使用了延迟消费减少了调用抖音接口的次数，并对SQL进行了优化提高了查询效率。在 V1.2 需求中使用多线程并发的拉取抖音订单详情中的信息，对其中的信息进行过滤筛选并归档到 TIDB 中。在日常工作中还负责将线上数据同步到 hive 数据仓库，实现日报的自动出数。

人脸签到

性能优化，如果有数据指标支撑，会更好。比如从xxx秒优化到xxx毫秒，查询时间减少95%等

2021.08 - 2021.09

- 1、该项目是一个前后端分离的项目，主要功能有授权模块、人脸比对、签到、签退、统计时长
- 2、功能的实现主要是基于SpringBoot + WebSocket + Swagger + Redis 技术栈实现，前后端采用JSON作为交互数据，采用了心跳机制保持了长连接，签到或签退会对已在线的用户进行实时广播，使用 Redis 缓存中间件对热数据进行缓存，并解决了缓存穿透的问题。使用Swagger对代码和注释进行开发文档的生成。

教育经历

项目经历这块，建议写上自己的收获

山西农业信息学院 - 数据科学与大数据技术 本科

2019.09 - 2023.06

荣誉奖项

2020年校级一等奖学金、2021年中国大学生计算机设计大赛三等奖

第十一届蓝桥杯大赛二等奖

社团和组织经历

双体系卓越人才教育基地软件部部长

2020.06 - 2021.06

积极组织学员学习软件编程类相关知识，对下一届学员进行学习进度的规划、以及定期检查学习情况，积极完成老师安排的任务、组织同学们参加各种计算机相关的比赛。

知识星球加入方式：



大彬 送你一张星球优惠券

「大彬的学习圈」

立减

¥ **50**

新人立减券

2023/12/12 13:00 后失效

 知识星球

长按扫码领取优惠 ▶



MySQL

1. 事务的四大特性?

事务特性ACID：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。

- **原子性**是指事务包含的所有操作要么全部成功，要么全部失败回滚。
- **一致性**是指一个事务执行之前和执行之后都必须处于一致性状态。比如a与b账户共有1000块，两人之间转账之后无论成功还是失败，它们的账户总和还是1000。
- **隔离性**。跟隔离级别相关，如 `read committed`，一个事务只能读到已经提交的修改。
- **持久性**是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

2. 数据库的三大范式

第一范式1NF

确保数据库表字段的原子性。

比如字段 `userInfo: 广东省 10086'`，依照第一范式必须拆分成 `userInfo: 广东省` `userTel: 10086` 两个字段。

第二范式2NF

首先要满足第一范式，另外包含两部分内容，一是表必须有一个主键；二是非主键列必须完全依赖于主键，而不能只依赖于主键的一部分。

举个例子。假定选课关系表为 `student_course` (student_no, student_name, age, course_name, grade, credit)，主键为(student_no, course_name)。其中学分完全依赖于课程名称，姓名年龄完全依赖学号，不符合第二范式，会导致数据冗余（学生选n门课，姓名年龄有n条记录）、插入异常（插入一门新课，因为没有学号，无法保存新课记录）等问题。

应该拆分成三个表：学生： `student` (student_no, student_name, 年龄)；课程： `course` (course_name, credit)；选课关系： `student_course_relation` (student_no, course_name, grade)。

第三范式3NF

首先要满足第二范式，另外非主键列必须直接依赖于主键，不能存在传递依赖。即不能存在：非主键列 A 依赖于非主键列 B，非主键列 B 依赖于主键的情况。

假定学生关系表为 `Student` (student_no, student_name, age, academy_id, academy_telephone)，主键为"学号"，其中学院id依赖于学号，而学院地点和学院电话依赖于学院id，存在传递依赖，不符合第三范式。

可以把学生关系表分为如下两个表：学生： (student_no, student_name, age, academy_id)；学院： (academy_id, academy_telephone)。

2NF和3NF的区别?

- 2NF依据是非主键列是否完全依赖于主键，还是依赖于主键的一部分。
- 3NF依据是非主键列是直接依赖于主键，还是直接依赖于非主键。

3. 事务隔离级别有哪些?

先了解下几个概念：脏读、不可重复读、幻读。

- **脏读**是指在一个事务处理过程里读取了另一个未提交的事务中的数据。
- **不可重复读**是指在对于数据库中的某行记录，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，另一个事务修改了数据并提交了。
- **幻读**是当某个事务在读取某个范围内的记录时，另外一个事务又在该范围内插入了新的记录。对幻读的正确理解是一个事务内的读取操作的结论不能支撑之后业务的执行。假设事务要新增一条记录，主键为id，在新增之前执行了select，没有发现id为xxx的记录，但插入时出现主键冲突，这就属于幻读，读取不到记录却发现主键冲突是因为记录实际上已经被其他的事务插入了，但当前事务不可见。

不可重复读和脏读的区别是，脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

事务隔离就是为了解决上面提到的脏读、不可重复读、幻读这几个问题。

MySQL数据库为我们提供的四种隔离级别：

- **Serializable** (串行化)：通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。
- **Repeatable read** (可重复读)：MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行，解决了不可重复读的问题。
- **Read committed** (读已提交)：一个事务只能看见已经提交事务所做的改变。可避免脏读的发生。
- **Read uncommitted** (读未提交)：所有事务都可以看到其他未提交事务的执行结果。

查看隔离级别：

```
select @@transaction_isolation;
```

设置隔离级别：

```
set session transaction isolation level read uncommitted;
```

4. 生产环境数据库一般用的什么隔离级别呢？

生产环境大多使用RC。为什么不是RR呢？

可重复读(Repeatable Read)，简称为RR 读已提交(Read Committed)，简称为RC

缘由一：在RR隔离级别下，存在间隙锁，导致出现死锁的几率比RC大的多！ 缘由二：在RR隔离级别下，条件列未命中索引会锁表！而在RC隔离级别下，只锁行！

也就是说，RC的并发性高于RR。

并且大部分场景下，不可重复读问题是可以接受的。毕竟数据都已经提交了，读出来本身就没有太大问题！

[互联网项目中mysql应该选什么事务隔离级别](#)

5. 编码和字符集的关系

我们平时可以在编辑器上输入各种中文英文字母，但这些都是给人读的，不是给计算机读的，其实计算机真正保存和传输数据都是以**二进制**0101的格式进行的。

那么就需要有一个规则，把中文和英文字母转化为二进制。其中d对应十六进制下的64，它可以转换为01二进制的格式。于是字母和数字就这样——对应起来了，这就是**ASCII编码格式**。

它用**一个字节**，也就是**8位**来标识字符，基础符号有128个，扩展符号也是128个。也就只能表示下**英文字母和数字**。

这明显不够用。于是，为了标识**中文**，出现了**GB2312**的编码格式。为了标识**希腊语**，出现了**greek**编码格式，为了标识**俄语**，整了**cp866**编码格式。

为了统一它们，于是出现了**Unicode编码格式**，它用了2~4个字节来表示字符，这样理论上所有符号都能被收录进去，并且它还完全兼容ASCII的编码，也就是说，同样是字母d，在ASCII用64表示，在Unicode里还是用64来表示。

但不同的地方是ASCII编码用1个字节来表示，而Unicode用则两个字节来表示。

同样都是字母d，unicode比ascii多使用了一个字节，如下：

```
D   ASCII:      01100100
D Unicode: 00000000 01100100
```

可以看到，上面的unicode编码，前面的都是0，其实用不上，但还占了个字节，有点浪费。如果我们能做到该隐藏时隐藏，这样就能省下不少空间，按这个思路，就是就有了**UTF-8编码**。

总结一下，按照一定规则把符号和二进制码对应起来，这就是**编码**。而把n多这种已经编码的字符聚在一起，就是我们常说的**字符集**。

比如utf-8字符集就是所有utf-8编码格式的字符的合集。

想看下mysql支持哪些字符集。可以执行 `show charset;`

6. utf8和utf8mb4的区别

上面提到utf-8是在unicode的基础上做的优化，既然unicode有办法表示所有字符，那utf-8也一样可以表示所有字符，为了避免混淆，我在后面叫它**大utf8**。

mysql支持的字符集中有utf8和utf8mb4。

先说**utf8mb4**编码，mb4就是**most bytes 4**的意思，从上图最右边的 `Maxlen` 可以看到，它最大支持用**4个字节**来表示字符，它几乎可以用来表示目前已知的所有的字符。

再说mysql字符集里的**utf8**，它是数据库的**默认字符集**。但注意，**此utf8非彼utf8**，我们叫它**小utf8**字符集。为什么这么说，因为从Maxlen可以看出，它最多支持用3个字节去表示字符，按utf8mb4的命名方式，准确点应该叫它**utf8mb3**。

utf8 就像是阉割版的utf8mb4，只支持部分字符。比如 `emoji` 表情，它就不支持。

而mysql支持的字符集里，第三列，**collation**，它是指**字符集的比较规则**。

比如，"debug"和"Debug"是同一个单词，但它们大小写不同，该不该判为同一个单词呢。

这时候就需要用到collation了。

通过 `SHOW COLLATION WHERE Charset = 'utf8mb4';` 可以查看到 **utf8mb4** 下支持什么比较规则。

```
mysql> SHOW COLLATION WHERE Charset = 'utf8mb4';
```

Collation	Charset	Id	Default	Compiled	Sortlen	Pad_attribute
utf8mb4_0900_ai_ci	utf8mb4	255	Yes	Yes	0	NO PAD
utf8mb4_0900_as_ci	utf8mb4	305		Yes	0	NO PAD
utf8mb4_0900_as_cs	utf8mb4	278		Yes	0	NO PAD
utf8mb4_0900_bin	utf8mb4	309		Yes	1	NO PAD
utf8mb4_bin	utf8mb4	46		Yes	1	PAD SPACE
utf8mb4_croatian_ci	utf8mb4	245		Yes	8	PAD SPACE
utf8mb4_cs_0900_ai_ci	utf8mb4	266		Yes	0	NO PAD
utf8mb4_cs_0900_as_cs	utf8mb4	289		Yes	0	NO PAD
utf8mb4_czech_ci	utf8mb4	234		Yes	8	PAD SPACE
utf8mb4_danish_ci	utf8mb4	235		Yes	8	PAD SPACE
utf8mb4_da_0900_ai_ci	utf8mb4	267		Yes	0	NO PAD
utf8mb4_da_0900_as_cs	utf8mb4	290		Yes	0	NO PAD
utf8mb4_de_pb_0900_ai_ci	utf8mb4	256		Yes	0	NO PAD
utf8mb4_de_pb_0900_as_cs	utf8mb4	279		Yes	0	NO PAD
utf8mb4_eo_0900_ai_ci	utf8mb4	273		Yes	0	NO PAD
utf8mb4_eo_0900_as_cs	utf8mb4	296		Yes	0	NO PAD
utf8mb4_esperanto_ci	utf8mb4	241		Yes	8	PAD SPACE
utf8mb4_estonian_ci	utf8mb4	230		Yes	8	PAD SPACE
utf8mb4_es_0900_ai_ci	utf8mb4	263		Yes	0	NO PAD
utf8mb4_es_0900_as_cs	utf8mb4	286		Yes	0	NO PAD
utf8mb4_es_trad_0900_ai_ci	utf8mb4	270		Yes	0	NO PAD

如果 `collation = utf8mb4_general_ci`，是指使用utf8mb4字符集的前提下，**挨个字符进行比较**（`general`），并且不区分大小写（`_ci, case insensitice`）。

这种情况下，"debug"和"Debug"是同一个单词。

如果改成 `collation=utf8mb4_bin`，就是指**挨个比较二进制位大小**。

于是"debug"和"Debug"就不是同一个单词。

那utf8mb4对比utf8有什么劣势吗？

我们知道数据库表里，字段类型如果是 `char(2)` 的话，里面的 **2** 是指**字符个数**，也就是说**不管这张表用的是什么编码的字符集**，都能放上2个字符。

而char又是**固定长度**，为了能放下2个utf8mb4的字符，char会默认保留 `2*4 (maxlen=4) = 8` 个字节的空间。

如果是utf8mb3，则会默认保留 `2 * 3 (maxlen=3) = 6` 个字节的空间。也就是说，在这种情况下，**utf8mb4会比utf8mb3多使用一些空间**。

7. 索引

7.1. 什么是索引？

索引是存储引擎用于提高数据库表的访问速度的一种**数据结构**。

7.2. 索引的优缺点？

优点：

- **加快数据查找的速度**
- 为用来排序或者是分组的字段添加索引，可以加快分组和排序的速度
- 加快表与表之间的连接

缺点：

- 建立索引需要**占用物理空间**
- 会降低表的增删改的效率，因为每次对表记录进行增删改，需要进行**动态维护索引**，导致增删改时间变长

7.3. 索引的作用？

数据是存储在磁盘上的，查询数据时，如果没有索引，会加载所有的数据到内存，依次进行检索，读取磁盘次数较多。有了索引，就不需要加载所有数据，因为B+树的高度一般在2-4层，最多只需要读取2-4次磁盘，查询速度大大提升。

7.4. 什么情况下需要建索引？

1. 经常用于查询的字段
2. 经常用于连接的字段建立索引，可以加快连接的速度
3. 经常需要排序的字段建立索引，因为索引已经排好序，可以加快排序查询速度

7.5. 什么情况下不建索引？

1. **where** 条件中用不到的字段不适合建立索引
2. 表记录较少。比如只有几百条数据，没必要加索引。
3. 需要经常增删改。需要评估是否适合加索引
4. **参与列计算**的列不适合建索引
5. **区分度不高**的字段不适合建立索引，如性别，只有男/女/未知三个值。加了索引，查询效率也不会提高。

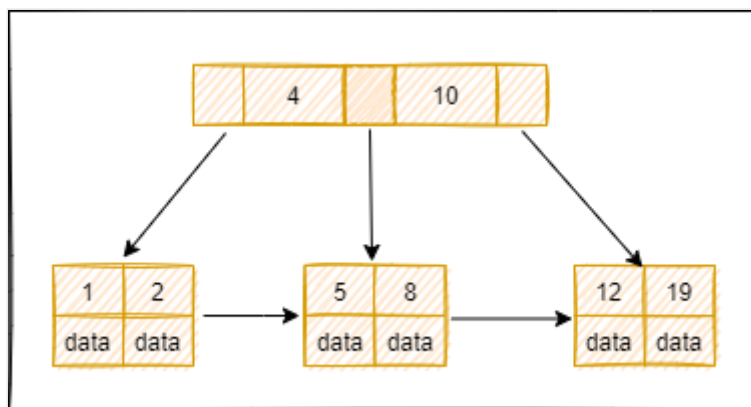
7.6. 索引的数据结构

索引的数据结构主要有B+树和哈希表，对应的索引分别为B+树索引和哈希索引。InnoDB引擎的索引类型有B+树索引和哈希索引，默认的索引类型为B+树索引。

B+树索引

B+ 树是基于B 树和叶子节点顺序访问指针进行实现，它具有B树的平衡性，并且通过顺序访问指针来提高区间查询的性能。

在 B+ 树中，节点中的 **key** 从左到右递增排列，如果某个指针的左右相邻 **key** 分别是 key_i 和 key_{i+1} ，则该指针指向节点的所有 **key** 大于等于 key_i 且小于等于 key_{i+1} 。



进行查找操作时，首先在根节点进行二分查找，找到 **key** 所在的指针，然后递归地在指针所指向的节点进行查找。直到查找到叶子节点，然后在叶子节点上进行二分查找，找出 **key** 所对应的数据项。

MySQL 数据库使用最多的索引类型是 **BTREE** 索引，底层基于B+树数据结构来实现。

```
mysql> show index from blog\G;
***** 1. row *****
      Table: blog
    Non_unique: 0
       Key_name: PRIMARY
   Seq_in_index: 1
```

```
Column_name: blog_id
Collation: A
Cardinality: 4
Sub_part: NULL
Packed: NULL
Null:
Index_type: BTREE
Comment:
Index_comment:
Visible: YES
Expression: NULL
```

哈希索引

哈希索引是基于哈希表实现的，对于每一行数据，存储引擎会对索引列进行哈希计算得到哈希码，并且哈希算法要尽量保证不同的列值计算出的哈希码值是不同的，将哈希码的值作为哈希表的key值，将指向数据行的指针作为哈希表的value值。这样查找一个数据的时间复杂度就是 $O(1)$ ，一般多用于精确查找。

7.7. Hash索引和B+树索引的区别？

- 哈希索引**不支持排序**，因为哈希表是无序的。
- 哈希索引**不支持范围查找**。
- 哈希索引**不支持模糊查询**及多列索引的最左前缀匹配。
- 因为哈希表中会**存在哈希冲突**，所以哈希索引的性能是不稳定的，而B+树索引的性能是相对稳定的，每次查询都是从根节点到叶子节点。

7.8. 为什么B+树比B树更适合实现数据库索引？

- 由于B+树的数据都存储在叶子结点中，叶子结点均为索引，方便扫库，只需要扫一遍叶子结点即可，但是B树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以B+树更加适合在区间查询的情况，而在数据库中基于范围的查询是非常频繁的，所以通常B+树用于数据库索引。
- B+树的节点只存储索引key值，具体信息的地址存在于叶子节点的地址中。这就使以页为单位的索引中可以存放更多的节点。减少更多的I/O支出。
- B+树的查询效率更加稳定，任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

7.9. 索引有什么分类？

1、**主键索引**：名为primary的唯一非空索引，不允许有空值。

2、**唯一索引**：索引列中的值必须是唯一的，但是允许为空值。唯一索引和主键索引的区别是：唯一索引字段可以为null且可以存在多个null值，而主键索引字段不可以为null。唯一索引的用途：唯一标识数据库表中的每条记录，主要是用来防止数据重复插入。创建唯一索引的SQL语句如下：

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name UNIQUE KEY(column_1,column_2,...);
```

3、**组合索引**：在表中的多个字段组合上创建的索引，只有在查询条件中使用了这些字段的左边字段时，索引才会被使用，使用组合索引时需遵循最左前缀原则。

4、**全文索引**：只能在 **CHAR**、**VARCHAR** 和 **TEXT** 类型字段上使用全文索引。

5、**普通索引**：普通索引是最基本的索引，它没有任何限制，值可以为空。

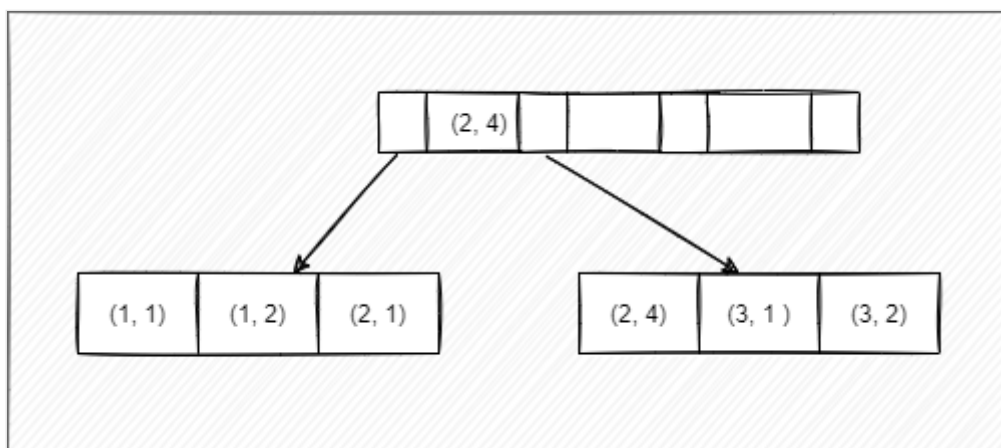
7.10. 什么是最左匹配原则？

如果 SQL 语句中用到了组合索引中的最左边的索引，那么这条 SQL 语句就可以利用这个组合索引去进行匹配。当遇到范围查询(>、<、between、like)就会停止匹配，后面的字段不会用到索引。

对 (a,b,c) 建立索引，查询条件使用 a/ab/abc 会走索引，使用 bc 不会走索引。

对 (a,b,c,d) 建立索引，查询条件为 a = 1 and b = 2 and c > 3 and d = 4，那么a、b和c三个字段能用到索引，而d无法使用索引。因为遇到了范围查询。

如下图，对(a, b) 建立索引，a 在索引树中是全局有序的，而 b 是全局无序，局部有序（当a相等时，会根据b进行排序）。直接执行 b = 2 这种查询条件无法使用索引。



当a的值确定的时候，b是有序的。例如 a = 1 时，b值为1，2是有序的状态。当 a = 2 时候，b的值为1，4也是有序状态。当执行 a = 1 and b = 2 时a和b字段能用到索引。而执行 a > 1 and b = 2 时，a字段能用到索引，b字段用不到索引。因为a的值此时是一个范围，不是固定的，在这个范围内b值不是有序的，因此b字段无法使用索引。

7.11. 什么是聚集索引？

InnoDB使用表的主键构造主键索引树，同时叶子节点中存放的即为整张表的记录数据。聚集索引叶子节点的存储是逻辑上连续的，使用双向链表连接，叶子节点按照主键的顺序排序，因此对于主键的排序查找和范围查找速度比较快。

聚集索引的叶子节点就是整张表的行记录。InnoDB 主键使用的是聚簇索引。聚集索引要比非聚集索引查询效率高很多。

对于 InnoDB 来说，聚集索引一般是表中的主键索引，如果表中没有显示指定主键，则会选择表中的第一个不允许为 NULL 的唯一索引。如果没有主键也没有合适的唯一索引，那么 InnoDB 内部会生成一个隐藏的主键作为聚集索引，这个隐藏的主键长度为6个字节，它的值会随着数据的插入自增。

7.12. 什么是覆盖索引？

select 的数据列只用从索引中就能够取得，不需要回表进行二次查询，也就是说查询列要被所使用的索引覆盖。对于 innodb 表的二级索引，如果索引能覆盖到查询的列，那么就可以避免对主键索引的二次查询。

不是所有类型的索引都可以成为覆盖索引。覆盖索引要存储索引列的值，而哈希索引、全文索引不存储索引列的值，所以MySQL使用b+树索引做覆盖索引。

对于使用了覆盖索引的查询，在查询前面使用 explain，输出的extra列会显示为 using index。

比如 user_like 用户点赞表，组合索引为 (user_id, blog_id)，user_id 和 blog_id 都不为 null。

```
explain select blog_id from user_like where user_id = 13;
```

explain 结果的 **Extra** 列为 **Using index**，查询的列被索引覆盖，并且where筛选条件符合最左前缀原则，通过**索引查找**就能直接找到符合条件的数据，不需要回表查询数据。

```
explain select user_id from user_like where blog_id = 1;
```

explain 结果的 **Extra** 列为 **Using where; Using index**，查询的列被索引覆盖，where筛选条件不符合最左前缀原则，无法通过索引查找找到符合条件的数据，但可以通过**索引扫描**找到符合条件的数据，也不需要回表查询数据。

```
mysql> explain select blog_id from user_like where user_id = 13;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | user_like | NULL | ref | ull | ull | 4 | const | 2 | 100.00 | Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

mysql> explain select user_id from user_like where blog_id = 1;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | user_like | NULL | index | ull | ull | 8 | NULL | 4 | 25.00 | Using where; Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

mysql> show index from user_like;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| user_like | 0 | PRIMARY | 1 | id | A | 4 | NULL | NULL | NULL | BTREE | | | YES |
| user_like | 0 | ull | 1 | user_id | A | 3 | NULL | NULL | NULL | BTREE | | | YES |
| user_like | 0 | ull | 2 | blog_id | A | 4 | NULL | NULL | NULL | BTREE | | | YES |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
组合索引(user_id, blog_id)
```

7.13. 索引的设计原则？

- 对于经常作为查询条件的字段，应该建立索引，以提高查询速度
- 为经常需要排序、分组和联合操作的字段建立索引
- 索引列的**区分度越高**，索引的效果越好。比如使用性别这种区分度很低的列作为索引，效果就会很差。
- 避免给"大字段"建立索引。尽量使用数据量小的字段作为索引。因为 **MySQL** 在维护索引的时候会将字段值一起维护的，那这样必然会导致索引占用更多的空间，另外在排序的时候需要花费更多的时间去对比。
- 尽量使用**短索引**，对于较长的字符串进行索引时应该指定一个较短的前缀长度，因为较小的索引涉及到的磁盘I/O较少，查询速度更快。
- 索引不是越多越好，每个索引都需要额外的物理空间，维护也需要花费时间。
- 频繁增删改的字段不要建立索引。假设某个字段频繁修改，那就意味着需要频繁的重建索引，这必然影响MySQL的性能
- 利用**最左前缀原则**。

7.14. 索引什么时候会失效？

导致索引失效的情况：

- 对于组合索引，不是使用组合索引最左边的字段，则不会使用索引
- 以%开头的like查询如 **%abc**，无法使用索引；非%开头的like查询如 **abc%**，相当于范围查询，会使用索引
- 查询条件中列类型是字符串，没有使用引号，可能会因为类型不同发生隐式转换，使索引失效
- 判断索引列是否不等于某个值时
- 对索引列进行运算
- 查询条件使用 **or** 连接，也会导致索引失效

7.15. 什么是前缀索引?

有时需要在很长的字符列上创建索引，这会造成索引特别大且慢。使用前缀索引可以避免这个问题。

前缀索引是指对文本或者字符串的前几个字符建立索引，这样索引的长度更短，查询速度更快。

创建前缀索引的关键在于选择足够长的前缀以**保证较高的索引选择性**。索引选择性越高查询效率就越高，因为选择性高的索引可以让MySQL在查找时过滤掉更多的数据行。

建立前缀索引的方式：

```
// email列创建前缀索引
ALTER TABLE table_name ADD KEY(column_name(prefix_length));
```

7.16. 索引下推

参考我的另一篇文章：[图解索引下推！](#)

8. 常见的存储引擎有哪些?

MySQL中常用的四种存储引擎分别是：**MyISAM**、**InnoDB**、**MEMORY**、**ARCHIVE**。MySQL 5.5版本后默认的存储引擎为 **InnoDB**。

InnoDB存储引擎

InnoDB是MySQL**默认的事务型存储引擎**，使用最广泛，基于聚簇索引建立的。InnoDB内部做了很多优化，如能够自动在内存中创建自适应hash索引，以加速读操作。

优点：支持事务和崩溃修复能力；引入了行级锁和外键约束。

缺点：占用的数据空间相对较大。

适用场景：需要事务支持，并且有较高的并发读写频率。

MyISAM存储引擎

数据以紧密格式存储。对于只读数据，或者表比较小、可以容忍修复操作，可以使用MyISAM引擎。MyISAM会将表存储在两个文件中，数据文件 **.MYD** 和索引文件 **.MYI**。

优点：访问速度快。

缺点：MyISAM不支持事务和行级锁，不支持崩溃后的安全恢复，也不支持外键。

适用场景：对事务完整性没有要求；表的数据都会只读的。

MEMORY存储引擎

MEMORY引擎将数据全部放在内存中，访问速度较快，但是一旦系统奔溃的话，数据都会丢失。

MEMORY引擎默认使用哈希索引，将键的哈希值和指向数据行的指针保存在哈希索引中。

优点：访问速度较快。

缺点：

1. 哈希索引数据不是按照索引值顺序存储，无法用于排序。
2. 不支持部分索引匹配查找，因为哈希索引是使用索引列的全部内容来计算哈希值的。
3. 只支持等值比较，不支持范围查询。
4. 当出现哈希冲突时，存储引擎需要遍历链表中所有的行指针，逐行进行比较，直到找到符合条件的行。

ARCHIVE存储引擎

ARCHIVE存储引擎非常适合存储大量独立的、作为历史记录的数据。ARCHIVE提供了压缩功能，拥有高效的插入速度，但是这种引擎不支持索引，所以查询性能较差。

9. MyISAM和InnoDB的区别？

1. **存储结构的区别。**每个MyISAM在磁盘上存储成三个文件。文件的名字以表的名字开始，扩展名指出文件类型。 .frm文件存储表定义。数据文件的扩展名为.MYD (MYData)。索引文件的扩展名是.MYI (MYIndex)。InnoDB所有的表都保存在同一个数据文件中（也可能是多个文件，或者是独立的表空间文件），InnoDB表的大小只受限于操作系统文件的大小，一般为2GB。
2. **存储空间的区别。**MyISAM支持支持三种不同的存储格式：静态表(默认，但是注意数据末尾不能有空格，会被去掉)、动态表、压缩表。当表在创建之后并导入数据之后，不会再进行修改操作，可以使用压缩表，极大的减少磁盘的空间占用。InnoDB需要更多的内存和存储，它会在主内存中建立其专用的缓冲池用于高速缓冲数据和索引。
3. **可移植性、备份及恢复。**MyISAM数据是以文件的形式存储，所以在跨平台的数据转移中会很方便。在备份和恢复时可单独针对某个表进行操作。对于InnoDB，可行的方案是拷贝数据文件、备份 binlog，或者用mysqldump，在数据量达到几十G的时候就相对麻烦了。
4. **是否支持行级锁。**MyISAM 只支持表级锁，用户在操作myisam表时，select, update, delete, insert语句都会给表自动加锁，如果加锁以后的表满足insert并发的情况下，可以在表的尾部插入新的数据。而InnoDB 支持行级锁和表级锁，默认为行级锁。行锁大幅度提高了多用户并发操作的性能。
5. **是否支持事务和崩溃后的安全恢复。** MyISAM 不提供事务支持。而InnoDB 提供事务支持，具有事务、回滚和崩溃修复能力。
6. **是否支持外键。**MyISAM不支持，而InnoDB支持。
7. **是否支持MVCC。**MyISAM不支持，InnoDB支持。应对高并发事务，MVCC比单纯的加锁更高效。
8. **是否支持聚集索引。**MyISAM不支持聚集索引，InnoDB支持聚集索引。
9. **全文索引。**MyISAM支持 FULLTEXT类型的全文索引。InnoDB不支持FULLTEXT类型的全文索引，但是innodb可以使用sphinx插件支持全文索引，并且效果更好。
10. **表主键。**MyISAM允许没有任何索引和主键的表存在，索引都是保存行的地址。对于InnoDB，如果没有设定主键或者非空唯一索引，就会自动生成一个6字节的主键(用户不可见)。
11. **表的行数。**MyISAM保存有表的总行数，如果 `select count(*) from table;` 会直接取出该值。InnoDB没有保存表的总行数，如果使用 `select count(*) from table;` 就会遍历整个表，消耗相当大，但是在加了where条件后，MyISAM和InnoDB处理的方式都一样。

10. MySQL有哪些锁？

按锁粒度分类，有行级锁、表级锁和页级锁。

1. 行级锁是mysql中锁定粒度最细的一种锁。表示只针对当前操作的行进行加锁。行级锁能大大减少数据库操作的冲突，其加锁粒度最小，但加锁的开销也最大。
2. 表级锁是mysql中锁定粒度最大的一种锁，表示对当前操作的整张表加锁，它实现简单，资源消耗较少，被大部分mysql引擎支持。最常使用的MyISAM与InnoDB都支持表级锁定。
3. 页级锁是 MySQL 中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级冲突少，但速度慢。因此，采取了折衷的页级锁，一次锁定相邻的一组记录。

按锁级别分类，有共享锁、排他锁和意向锁。

1. 共享锁又称读锁，是读取操作创建的锁。其他用户可以并发读取数据，但任何事务都不能对数据进行修改（获取数据上的排他锁），直到已释放所有共享锁。
2. 排他锁又称写锁、独占锁，如果事务T对数据A加上排他锁后，则其他事务不能再对A加任何类型的封锁。获准排他锁的事务既能读数据，又能修改数据。
3. 意向锁是表级锁，其设计目的主要是为了在一个事务中揭示下一行将要被请求锁的类型。InnoDB中的两个表锁：

意向共享锁 (IS)：表示事务准备给数据行加入共享锁，也就是说一个数据行加共享锁前必须先取得该表的IS锁；

意向排他锁 (IX)：类似上面，表示事务准备给数据行加入排他锁，说明事务在一个数据行加排他锁前必须先取得该表的IX锁。

意向锁是 InnoDB 自动加的，不需要用户干预。

对于INSERT、UPDATE和DELETE，InnoDB 会自动给涉及的数据加排他锁；对于一般的SELECT语句，InnoDB 不会加任何锁，事务可以通过以下语句显式加共享锁或排他锁。

共享锁： `SELECT ... LOCK IN SHARE MODE;`

排他锁： `SELECT ... FOR UPDATE;`

11. MVCC 实现原理？

MVCC(`Multiversion concurrency control`) 就是同一份数据保留多版本的一种方式，进而实现并发控制。在查询的时候，通过 `read view` 和版本链找到对应版本的数据。

作用：提升并发性能。对于高并发场景，MVCC比行级锁开销更小。

MVCC 实现原理如下：

MVCC 的实现依赖于版本链，版本链是通过表的三个隐藏字段实现。

- `DB_TRX_ID`：当前事务id，通过事务id的大小判断事务的时间顺序。
- `DB_ROLL_PTR`：回滚指针，指向当前行记录的上一个版本，通过这个指针将数据的多个版本连接在一起构成 `undo log` 版本链。
- `DB_ROW_ID`：主键，如果数据表没有主键，InnoDB会自动生成主键。

每条表记录大概是这样的：

name	age	DB_ROW_ID	DB_TRX_ID	DB_ROLL_PTR
大彬	18	1	2	0x233333

使用事务更新行记录的时候，就会生成版本链，执行过程如下：

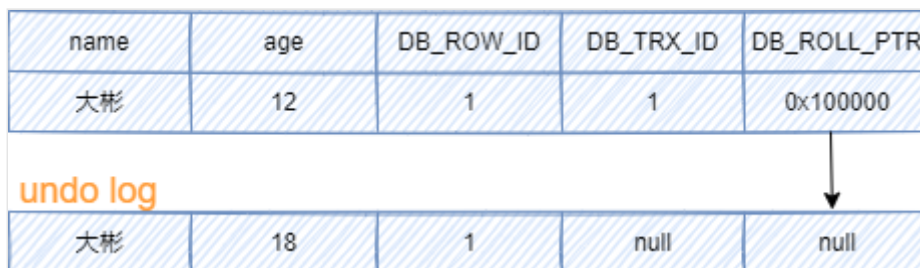
1. 用排他锁锁住该行；
2. 将该行原本的值拷贝到 `undo log`，作为旧版本用于回滚；
3. 修改当前行的值，生成一个新版本，更新事务id，使回滚指针指向旧版本的记录，这样就形成一条版本链。

下面举个例子方便大家理解。

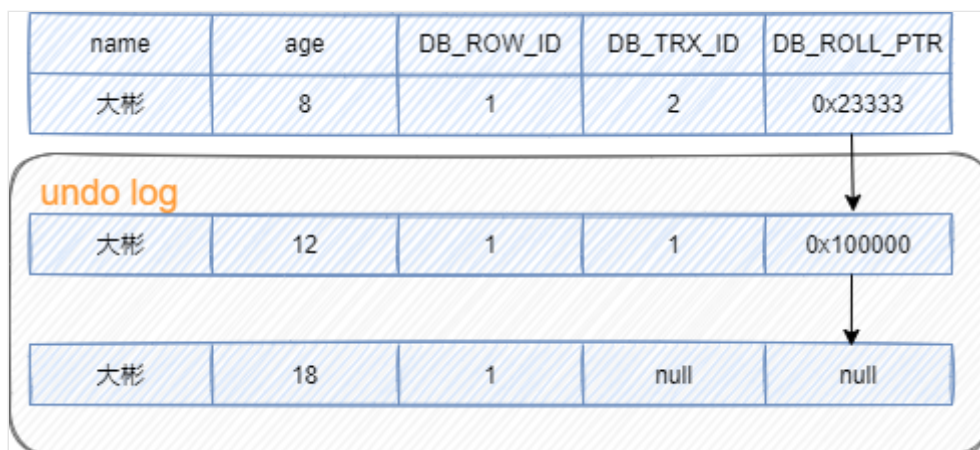
1、初始数据如下，其中 `DB_ROW_ID` 和 `DB_ROLL_PTR` 为空。

name	age	DB_ROW_ID	DB_TRX_ID	DB_ROLL_PTR
大彬	18	1	null	null

2、事务A对该行数据做了修改，将 `age` 修改为12，效果如下：



3、之后事务B也对该行记录做了修改，将 age 修改为8，效果如下：



4、此时undo log有两行记录，并且通过回滚指针连在一起。

接下来了解下read view的概念。

read view可以理解成将数据在每个时刻的状态拍成“照片”记录下来。在获取某时刻t的数据时，到t时间点拍的“照片”上取数据。

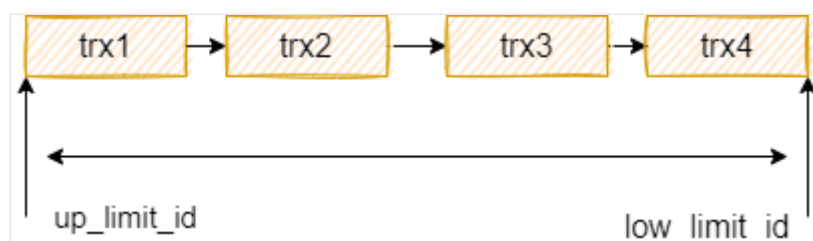
在 **read view** 内部维护一个活跃事务链表，表示生成 **read view** 的时候还在活跃的事务。这个链表包含在创建 **read view** 之前还未提交的事务，不包含创建 **read view** 之后提交的事务。

不同隔离级别创建read view的时机不同。

- read committed：每次执行select都会创建新的read_view，保证能读取到其他事务已经提交的修改。
- repeatable read：在一个事务范围内，第一次select时更新这个read_view，以后不会再更新，后续所有的select都是复用之前的read_view。这样可以保证事务范围内每次读取的内容都一样，即可重复读。

read view的记录筛选方式

前提： **DATA_TRX_ID** 表示每个数据行的最新的事务ID； **up_limit_id** 表示当前快照中的最先开始的事务； **low_limit_id** 表示当前快照中的最慢开始的事务，即最后一个事务。



- 如果 **DATA_TRX_ID** < **up_limit_id**：说明在创建 **read view** 时，修改该数据行的事务已提交，该版本的记录可被当前事务读取到。
- 如果 **DATA_TRX_ID** >= **low_limit_id**：说明当前版本的记录的事务是在创建 **read view** 之后生成的，该版本的数据行不可以被当前事务访问。此时需要通过版本链找到上一个版本，然后重新判断该版本的记录对当前事务的可见性。

- 如果 `up_limit_id <= DATA_TRX_ID < low_limit_i` :
 1. 需要在活跃事务链表中查找是否存在ID为 `DATA_TRX_ID` 的值的事务。
 2. 如果存在，因为在活跃事务链表中的事务是未提交的，所以该记录是不可见的。此时需要通过版本链找到上一个版本，然后重新判断该版本的可见性。
 3. 如果不存在，说明事务 `trx_id` 已经提交了，这行记录是可见的。

总结：InnoDB 的 **MVCC** 是通过 **read view** 和版本链实现的，版本链保存有历史版本记录，通过 **read view** 判断当前版本的数据是否可见，如果不可见，再从版本链中找到上一个版本，继续进行判断，直到找到一个可见的版本。

12. 快照读和当前读

表记录有两种读取方式。

- 快照读：读取的是快照版本。普通的 **SELECT** 就是快照读。通过mvcc来进行并发控制的，不用加锁。
- 当前读：读取的是最新版本。 **UPDATE、DELETE、INSERT、SELECT ... LOCK IN SHARE MODE、SELECT ... FOR UPDATE** 是当前读。

快照读情况下，InnoDB通过 **mvcc** 机制避免了幻读现象。而 **mvcc** 机制无法避免当前读情况下出现的幻读现象。因为当前读每次读取的都是最新数据，这时如果两次查询中间有其它事务插入数据，就会产生幻读。

下面举个例子说明下：

- 1、首先，user表只有两条记录，具体如下：

```
mysql> select * from user;
```

user_id	user_name	user_password	user_mail	user_state	user_reward
1	admin	CVpm/0qEa	xxxx@xxxx.com	1	null
4	adminA	qduyivvumvboysj.1s9QaarInUe.VghyFslfkp0BiZo3/34JiGZ618aBmy		1	NULL

- 2、事务a和事务b同时开启事务 **start transaction** ;
- 3、事务a插入数据然后提交；

```
insert into user(user_name, user_password, user_mail, user_state)
values('tyson', 'a', 'a', 0);
```

- 4、事务b执行全表的update；

```
update user set user_name = 'a';
```

- 5、事务b然后执行查询，查到了事务a中插入的数据。（下图左边是事务b，右边是事务a。事务开始之前只有两条记录，事务a插入一条数据之后，事务b查询出来是三条数据）

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into user(user_name, user_password, user_mail, user_state)
values('tyson', 'a', 'a', 0);
Query OK, 1 row affected (0.05 sec) 2

mysql> commit; 3
Query OK, 0 rows affected (0.43 sec)

mysql>

mysql> select user_name from user;
+-----+
| user_name |
+-----+
| admin     |
| adminA    |
+-----+
2 rows in set (0.00 sec) 1

mysql> update user set user_name='a';
Query OK, 3 rows affected (0.12 sec)
Rows matched: 3 Changed: 3 Warnings: 0 4

mysql> select user_name from user;
+-----+
| user_name |
+-----+
| a         |
| a         |
| a         |
+-----+
3 rows in set (0.00 sec) 5

读到其他事务插入的数据
```

以上就是当前读出现的幻读现象。

那么MySQL是如何避免幻读？

- 在快照读情况下，MySQL通过 **mvcc** 来避免幻读。
- 在当前读情况下，MySQL通过 **next-key** 来避免幻读（加行锁和间隙锁来实现的）。

next-key包括两部分：行锁和间隙锁。行锁是加在索引上的锁，间隙锁是加在索引之间的。

Serializable 隔离级别也可以避免幻读，会锁住整张表，并发性极低，一般不会使用。

13. 共享锁和排他锁

SELECT 的读取锁定主要分为两种方式：共享锁和排他锁。

```
select * from table where id<6 lock in share mode;--共享锁
select * from table where id<6 for update;--排他锁
```

这两种方式主要的不同在于 **LOCK IN SHARE MODE** 多个事务同时更新同一个表时很容易造成死锁。

申请排他锁的前提是，没有线程对该结果集的任何行数据使用排它锁或者共享锁，否则申请会受到阻塞。在进行事务操作时，MySQL会对查询结果集的每行数据添加排它锁，其他线程对这些数据的更改或删除操作会被阻塞（只能读操作），直到该语句的事务被 **commit** 语句或 **rollback** 语句结束为止。

SELECT... FOR UPDATE 使用注意事项：

1. **for update** 仅适用于innodb，且必须在事务范围内才能生效。
2. 根据主键进行查询，查询条件为 **like** 或者不等于，主键字段产生**表锁**。
3. 根据非索引字段进行查询，会产生**表锁**。

14. bin log/redo log/undo log

MySQL日志主要包括查询日志、慢查询日志、事务日志、错误日志、二进制日志等。其中比较重要的是 **bin log**（二进制日志）和 **redo log**（重做日志）和 **undo log**（回滚日志）。

bin log

bin log 是MySQL数据库级别的文件，记录对MySQL数据库执行修改的所有操作，不会记录select和show语句，主要用于恢复数据库和同步数据库。

redo log

`redo log` 是innodb引擎级别，用来记录innodb存储引擎的事务日志，不管事务是否提交都会记录下来，用于数据恢复。当数据库发生故障，innodb存储引擎会使用 `redo log` 恢复到发生故障前的时刻，以此来保证数据的完整性。将参数 `innodb_flush_log_at_tx_commit` 设置为1，那么在执行commit时会将 `redo log` 同步写到磁盘。

undo log

除了记录 `redo log` 外，当进行数据修改时还会记录 `undo log`，`undo log` 用于数据的撤回操作，它保留了记录修改前的内容。通过 `undo log` 可以实现事务回滚，并且可以根据 `undo log` 回溯到某个特定的版本的数据，实现MVCC。

15. bin log和redo log有什么区别？

1. `bin log` 会记录所有日志记录，包括InnoDB、MyISAM等存储引擎的日志；`redo log` 只记录innodb自身的事务日志。
2. `bin log` 只在事务提交前写入到磁盘，一个事务只写一次；而在事务进行过程，会有 `redo log` 不断写入磁盘。
3. `bin log` 是逻辑日志，记录的是SQL语句的原始逻辑；`redo log` 是物理日志，记录的是在某个数据页上做了什么修改。

16. 讲一下MySQL架构？

MySQL主要分为 Server 层和存储引擎层：

- **Server 层**：主要包括连接器、查询缓存、分析器、优化器、执行器等，所有跨存储引擎的功能都在这一层实现，比如存储过程、触发器、视图，函数等，还有一个通用的日志模块 `binlog` 日志模块。
- **存储引擎**：主要负责数据的存储和读取。server 层通过api与存储引擎进行通信。

Server 层基本组件

- **连接器**：当客户端连接 MySQL 时，server层会对其进行身份认证和权限校验。
- **查询缓存**：执行查询语句的时候，会先查询缓存，先校验这个 sql 是否执行过，如果有缓存这个 sql，就会直接返回给客户端，如果没有命中，就会执行后续的操作。
- **分析器**：没有命中缓存的话，SQL 语句就会经过分析器，主要分为两步，词法分析和语法分析，先看 SQL 语句要做什么，再检查 SQL 语句语法是否正确。
- **优化器**：优化器对查询进行优化，包括重写查询、决定表的读写顺序以及选择合适的索引等，生成执行计划。
- **执行器**：首先执行前会校验该用户有没有权限，如果没有权限，就会返回错误信息，如果有权限，就会根据执行计划去调用引擎的接口，返回结果。

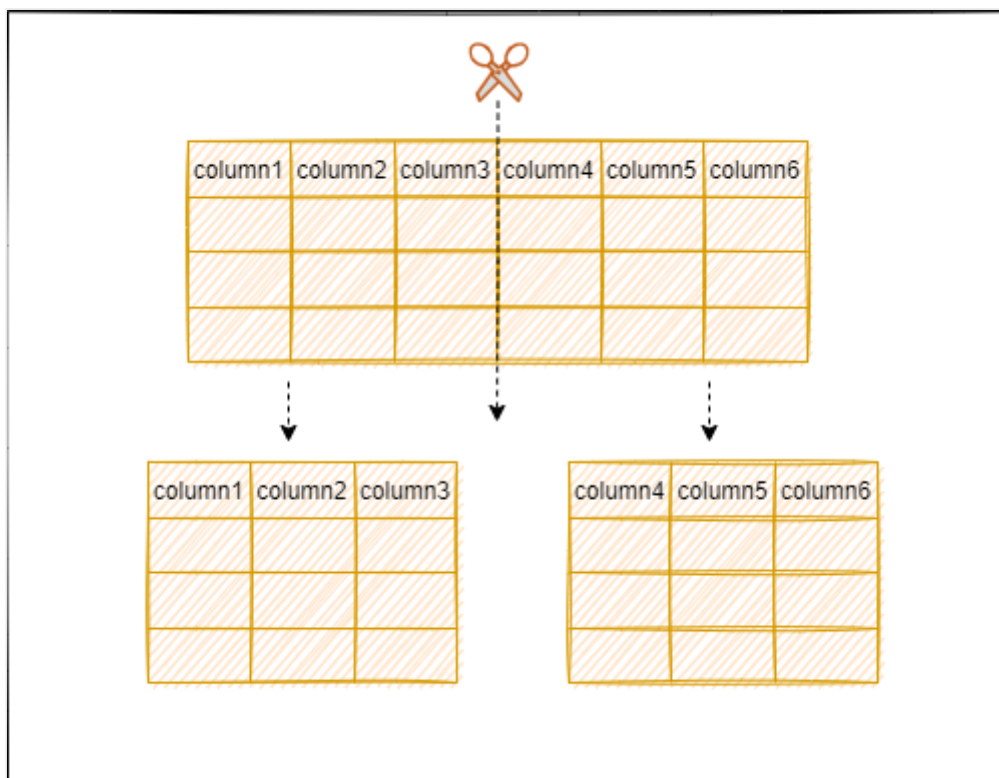
17. 分库分表

当单表的数据量达到1000W或100G以后，优化索引、添加从库等可能对数据库性能提升效果不明显，此时就要考虑对其进行切分了。切分的目的就在于减少数据库的负担，缩短查询的时间。

数据切分可以分为两种方式：垂直划分和水平划分。

垂直划分

垂直划分数据库是根据业务进行划分，例如购物场景，可以将库中涉及商品、订单、用户的表分别划分成一个库，通过降低单库的大小来提高性能。同样的，分表的情况就是将一个大表根据业务功能拆分成一个个子表，例如商品基本信息和商品描述，商品基本信息一般会展示在商品列表，商品描述在商品详情页，可以将商品基本信息和商品描述拆分成两张表。



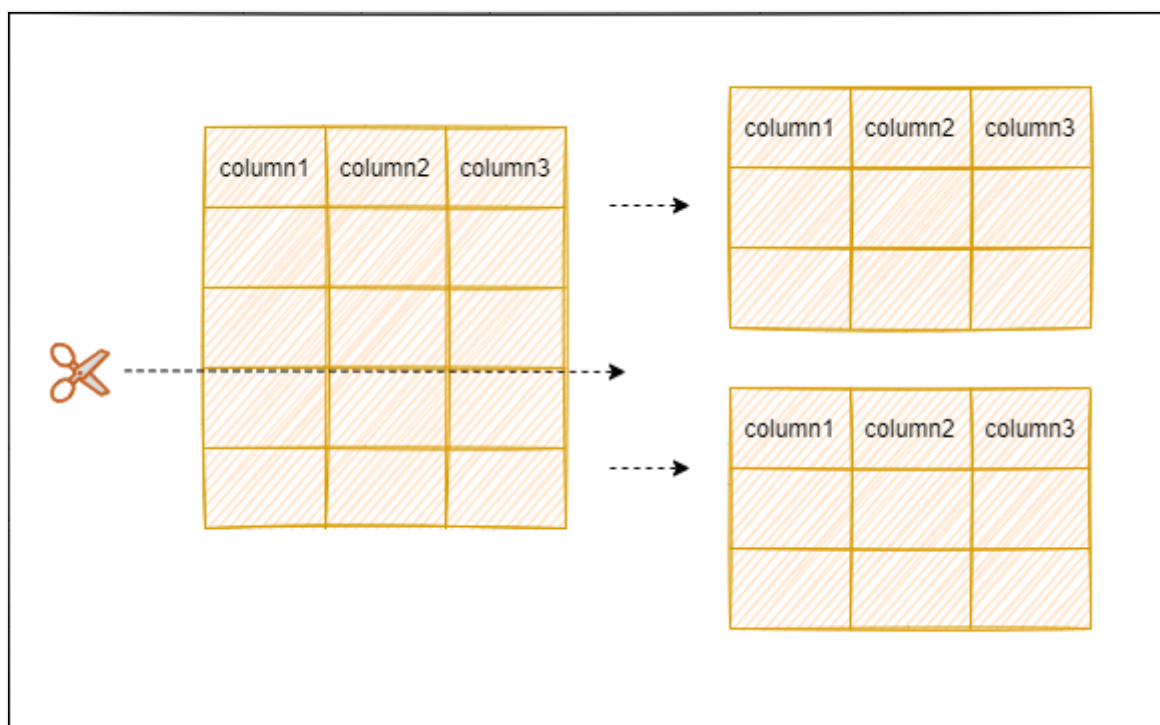
优点：行记录变小，数据页可以存放更多记录，在查询时减少I/O次数。

缺点：

- 主键出现冗余，需要管理冗余列；
- 会引起表连接JOIN操作，可以通过在业务服务器上进行join来减少数据库压力；
- 依然存在单表数据量过大的问题。

水平划分

水平划分是根据一定规则，例如时间或id序列值等进行数据的拆分。比如根据年份来拆分不同的数据库。每个数据库结构一致，但是数据得以拆分，从而提升性能。



优点：单库（表）的数据量得以减少，提高性能；切分出的表结构相同，程序改动较少。

缺点：

- 分片事务一致性难以解决
- 跨节点 join 性能差，逻辑复杂
- 数据分片在扩容时需要迁移

18. 什么是分区表？

分区是把一张表的数据分成N多个区块。分区表是一个独立的逻辑表，但是底层由多个物理子表组成。

当查询条件的数据分布在某一个分区的时候，查询引擎只会去某一个分区查询，而不是遍历整个表。在管理层面，如果需要删除某一个分区的数据，只需要删除对应的分区即可。

分区一般都是放在单机里的，用的比较多的是时间范围分区，方便归档。只不过分库分表需要代码实现，分区则是mysql内部实现。分库分表和分区并不冲突，可以结合使用。

19. 分区表类型

range分区，按照范围分区。比如按照时间范围分区

```
CREATE TABLE test_range_partition(  
    id INT auto_increment,  
    createdate DATETIME,  
    primary key (id,createdate)  
)  
PARTITION BY RANGE (TO_DAYS(createdate)) (  
    PARTITION p201801 VALUES LESS THAN ( TO_DAYS('20180201') ),  
    PARTITION p201802 VALUES LESS THAN ( TO_DAYS('20180301') ),  
    PARTITION p201803 VALUES LESS THAN ( TO_DAYS('20180401') ),  
    PARTITION p201804 VALUES LESS THAN ( TO_DAYS('20180501') ),  
    PARTITION p201805 VALUES LESS THAN ( TO_DAYS('20180601') ),  
    PARTITION p201806 VALUES LESS THAN ( TO_DAYS('20180701') ),  
    PARTITION p201807 VALUES LESS THAN ( TO_DAYS('20180801') ),  
    PARTITION p201808 VALUES LESS THAN ( TO_DAYS('20180901') ),  
    PARTITION p201809 VALUES LESS THAN ( TO_DAYS('20181001') ),  
    PARTITION p201810 VALUES LESS THAN ( TO_DAYS('20181101') ),  
    PARTITION p201811 VALUES LESS THAN ( TO_DAYS('20181201') ),  
    PARTITION p201812 VALUES LESS THAN ( TO_DAYS('20190101') )  
);
```

在 `/var/lib/mysql/data/` 可以找到对应的数据文件，每个分区表都有一个使用#分隔命名的表文件：

```
-rw-r----- 1 MySQL MySQL 65 Mar 14 21:47 db.opt  
-rw-r----- 1 MySQL MySQL 8598 Mar 14 21:50 test_range_partition.frm  
-rw-r----- 1 MySQL MySQL 98304 Mar 14 21:50  
test_range_partition#P#p201801.ibd  
-rw-r----- 1 MySQL MySQL 98304 Mar 14 21:50  
test_range_partition#P#p201802.ibd  
-rw-r----- 1 MySQL MySQL 98304 Mar 14 21:50  
test_range_partition#P#p201803.ibd  
...
```

list分区

list分区和range分区相似，主要区别在于list是枚举值列表的集合，range是连续的区间值的集合。对于list分区，分区字段必须是已知的，如果插入的字段不在分区时的枚举值中，将无法插入。

```
create table test_list_partition
(
    id int auto_increment,
    data_type tinyint,
    primary key(id,data_type)
)partition by list(data_type)
(
    partition p0 values in (0,1,2,3,4,5,6),
    partition p1 values in (7,8,9,10,11,12),
    partition p2 values in (13,14,15,16,17)
);
```

hash分区

可以将数据均匀地分布到预先定义的分区中。

```
create table test_hash_partition
(
    id int auto_increment,
    create_date datetime,
    primary key(id,create_date)
)partition by hash(year(create_date)) partitions 10;
```

20. 分区的问题？

1. 打开和锁住所有底层表的成本可能很高。当查询访问分区表时，MySQL 需要打开并锁住所有的底层表，这个操作在分区过滤之前发生，所以无法通过分区过滤来降低此开销，会影响到查询速度。可以通过批量操作来降低此类开销，比如批量插入、`LOAD DATA INFILE` 和一次删除多行数据。
2. 维护分区的成本可能很高。例如重组分区，会先创建一个临时分区，然后将数据复制到其中，最后再删除原分区。
3. 所有分区必须使用相同的存储引擎。

21. 查询语句执行流程？

查询语句的执行流程如下：权限校验、查询缓存、分析器、优化器、权限校验、执行器、引擎。

举个例子，查询语句如下：

```
select * from user where id > 1 and name = '大彬';
```

1. 首先检查权限，没有权限则返回错误；
2. MySQL 8.0 以前会查询缓存，缓存命中则直接返回，没有则执行下一步；
3. 词法分析和语法分析。提取表名、查询条件，检查语法是否有错误；
4. 两种执行方案，先查 `id > 1` 还是 `name = '大彬'`，优化器根据自己的优化算法选择执行效率最好的方案；
5. 校验权限，有权限就调用数据库引擎接口，返回引擎的执行结果。

22. 更新语句执行过程？

更新语句执行流程如下：分析器、权限校验、执行器、引擎、`redo log` (`prepare` 状态)、`binlog`、`redo log` (`commit` 状态)

举个例子，更新语句如下：

```
update user set name = '大彬' where id = 1;
```

1. 先查询到 id 为1的记录，有缓存会使用缓存。
2. 拿到查询结果，将 name 更新为大彬，然后调用引擎接口，写入更新数据，innodb 引擎将数据保存在内存中，同时记录 redo log，此时 redo log 进入 prepare 状态。
3. 执行器收到通知后记录 binlog，然后调用引擎接口，提交 redo log 为 commit 状态。
4. 更新完成。

为什么记录完 redo log，不直接提交，而是先进入 prepare 状态？

假设先写 redo log 直接提交，然后写 binlog，写完 redo log 后，机器挂了，binlog 日志没有被写入，那么机器重启后，这台机器会通过 redo log 恢复数据，但是这个时候 binlog 并没有记录该数据，后续进行机器备份的时候，就会丢失这一条数据，同时主从同步也会丢失这一条数据。

23. exist和in的区别？

exists 用于对外表记录做筛选。exists 会遍历外表，将外查询表的每一行，代入内查询进行判断。当 exists 里的条件语句能够返回记录行时，条件就为真，返回外表当前记录。反之如果 exists 里的条件语句不能返回记录行，条件为假，则外表当前记录被丢弃。

```
select a.* from A a where exists(select 1 from B b where a.id=b.id)
```

in 是先把后边的语句查出来放到临时表中，然后遍历临时表，将临时表的每一行，代入外查询去查找。

```
select * from A where id in(select id from B)
```

子查询的表比较大的时候，使用 exists 可以有效减少总的循环次数来提升速度；当外查询的表比较大的时候，使用 in 可以有效减少对外查询表循环遍历来提升速度。

24. MySQL中int(10)和char(10)的区别？

int(10)中的10表示的是显示数据的长度，而char(10)表示的是存储数据的长度。

25. truncate、delete与drop区别？

相同点：

1. truncate 和不带 where 子句的 delete、以及 drop 都会删除表内的数据。
2. drop、truncate 都是 DDL 语句（数据定义语言），执行后会自动提交。

不同点：

1. truncate 和 delete 只删除数据不删除表的结构；drop 语句将删除表的结构被依赖的约束、触发器、索引；
2. 一般来说，执行速度: drop > truncate > delete。

26. having和where区别？

- 二者作用的对象不同，where 子句作用于表和视图，having 作用于组。
- where 在数据分组前进行过滤，having 在数据分组后进行过滤。

27. 什么是MySQL主从同步？

主从同步使得数据可以从一个数据库服务器复制到其他服务器上，在复制数据时，一个服务器充当主服务器（`master`），其余的服务器充当从服务器（`slave`）。

因为复制是异步进行的，所以从服务器不需要一直连接着主服务器，从服务器甚至可以通过拨号断断续续地连接主服务器。通过配置文件，可以指定复制所有的数据库，某个数据库，甚至是某个数据库上的某个表。

28. 为什么要做主从同步？

1. 读写分离，使数据库能支撑更大的并发。
2. 在主服务器上生成实时数据，而在从服务器上分析这些数据，从而提高主服务器的性能。
3. 数据备份，保证数据的安全。

29. 乐观锁和悲观锁是什么？

数据库中的并发控制是确保在多个事务同时存取数据库中同一数据时不破坏事务的隔离性和统一性以及数据库的统一性。乐观锁和悲观锁是并发控制主要采用的技术手段。

- 悲观锁：假定会发生并发冲突，会对操作的数据进行加锁，直到提交事务，才会释放锁，其他事务才能进行修改。实现方式：使用数据库中的锁机制。
- 乐观锁：假设不会发生并发冲突，只在提交操作时检查是否数据是否被修改过。给表增加 `version` 字段，在修改提交之前检查 `version` 与原来取到的 `version` 值是否相等，若相等，表示数据没有被修改，可以更新，否则，数据为脏数据，不能更新。实现方式：乐观锁一般使用版本号机制或 `CAS` 算法实现。

30. 用过processlist吗？

`show processlist` 或 `show full processlist` 可以查看当前 MySQL 是否有压力，正在运行的 `SQL`，有没有慢 `SQL` 正在执行。返回参数如下：

1. **id**：线程ID，可以用 `kill id` 杀死某个线程
2. **db**：数据库名称
3. **user**：数据库用户
4. **host**：数据库实例的IP
5. **command**：当前执行的命令，比如 `sleep`，`Query`，`Connect` 等
6. **time**：消耗时间，单位秒
7. **state**：执行状态，主要有以下状态：
 - Sleep，线程正在等待客户端发送新的请求
 - Locked，线程正在等待锁
 - Sending data，正在处理 `SELECT` 查询的记录，同时把结果发送给客户端
 - Kill，正在执行 `kill` 语句，杀死指定线程
 - Connect，一个从节点连上了主节点
 - Quit，线程正在退出
 - Sorting for group，正在为 `GROUP BY` 做排序
 - Sorting for order，正在为 `ORDER BY` 做排序
8. **info**：正在执行的 `SQL` 语句

31. MySQL查询 limit 1000,10 和limit 10 速度一样快吗？

两种查询方式。对应 `limit offset, size` 和 `limit size` 两种方式。

而其实 `limit size`，相当于 `limit 0, size`。也就是从0开始取size条数据。

也就是说，两种方式的**区别在于offset是否为0**。

先来看下limit sql的内部执行逻辑。

MySQL内部分为**server层**和**存储引擎层**。一般情况下存储引擎都用innodb。

server层有很多模块，其中需要关注的是**执行器**是用于跟存储引擎打交道的组件。

执行器可以通过调用存储引擎提供的接口，将一行行数据取出，当这些数据完全符合要求（比如满足其他where条件），则会放到**结果集中**，最后返回给调用mysql的**客户端**。

以主键索引的limit执行过程为例：

执行 `select * from xxx order by id limit 0, 10;`，select后面带的是**星号**，也就是要求获得行数据的**所有字段信息**。

server层会调用innodb的接口，在innodb里的**主键索引**中获取到第0到10条**完整行数据**，依次返回给server层，并放到server层的结果集中，返回给客户端。

把offset搞大点，比如执行的是：`select * from xxx order by id limit 500000, 10;`

server层会调用innodb的接口，由于这次的offset=500000，会在innodb里的**主键索引**中获取到第0到(500000 + 10) 条**完整行数据**，**返回给server层之后根据offset的值挨个抛弃，最后只留下最后面的size条**，也就是10条数据，放到server层的结果集中，返回给客户端。

可以看出，当offset非0时，server层会从引擎层获取到**很多无用的数据**，而获取的这些无用数据都是要耗时的。

因此，mysql查询中 limit 1000,10 会比 limit 10 更慢。原因是 limit 1000,10 会取出1000+10条数据，并抛弃前1000条，这部分耗时更大。

32. 深分页怎么优化？

还是以上面的SQL为空：`select * from xxx order by id limit 500000, 10;`

方法一：

从上面的分析可以看出，当offset非常大时，server层会从引擎层获取到很多无用的数据，而当select后面是*号时，就需要拷贝完整的行信息，**拷贝完整数据相比只拷贝行数据里的其中一两个列字段**更耗费时间。

因为前面的offset条数据最后都是不要的，没有必要拷贝完整字段，所以可以将sql语句修改成：

```
select * from xxx where id >=(select id from xxx order by id limit 500000, 1)
order by id limit 10;
```

先执行子查询 `select id from xxx by id limit 500000, 1`，这个操作，其实也是将在innodb中的**主键索引**中获取到 **500000+1** 条数据，然后server层会抛弃前500000条，只保留最后一条数据的id。

但不同的地方在于，在返回server层的过程中，只会拷贝数据行内的id这一列，而不会拷贝数据行的所有列，当数据量较大时，这部分的耗时还是比较明显的。

在拿到了上面的id之后，假设这个id正好等于500000，那sql就变成了

```
select * from xxx where id >=500000 order by id limit 10;
```

这样innodb再走一次**主键索引**，通过B+树快速定位到id=500000的行数据，时间复杂度是lg(n)，然后向后取10条数据。

方法二：

将所有数据**根据id主键进行排序**，然后分批次取，将当前批次的最大id作为下次筛选的条件进行查询。

```
select * from xxx where id > start_id order by id limit 10;
```

mysql

通过主键索引，每次定位到start_id的位置，然后往后遍历10个数据，这样不管数据多大，查询性能都较为稳定。

33. 高度为3的B+树，可以存放多少数据？

InnoDB存储引擎有自己的最小储存单元——页（Page）。

查询InnoDB页大小的命令如下：

```
mysql> show global status like 'innodb_page_size';
+-----+-----+
| variable_name | value |
+-----+-----+
| Innodb_page_size | 16384 |
+-----+-----+
```

可以看出 innodb 默认的一页大小为 $16384B = 16384/1024 = 16kb$ 。

在MySQL中，B+树一个节点的大小设为一页或页的倍数最为合适。因为如果一个节点的大小 < 1 页，那么读取这个节点的时候其实读取的还是一页，这样就造成了资源的浪费。

B+树中**非叶子节点存的是key + 指针；叶子节点存的是数据行**。

对于叶子节点，如果一行数据大小为1k，那么一页就能存16条数据。

对于非叶子节点，如果key使用的是bigint，则为8字节，指针在MySQL中为6字节，一共是14字节，则16k能存放 $16 * 1024 / 14 = 1170$ 个索引指针。

于是可以算出，对于一颗高度为2的B+树，根节点存储索引指针节点，那么它有1170个叶子节点存储数据，每个叶子节点可以存储16条数据，一共 $1170 * 16 = 18720$ 条数据。而对于高度为3的B+树，就可以存放 $1170 * 1170 * 16 = 21902400$ 条数据（**两千多万条数据**），也就是对于两千多万条的数据，我们只需要**高度为3**的B+树就可以完成，通过主键查询只需要3次IO操作就能查到对应数据。

所以在 InnoDB 中B+树高度一般为3层时，就能满足千万级的数据存储。

参考：<https://www.cnblogs.com/leefreeman/p/8315844.html>

34. MySQL单表多大进行分库分表？

目前主流的有两种说法：

1. MySQL 单表数据量大于 2000 万行，性能会明显下降，考虑进行分库分表。
2. 阿里巴巴《Java 开发手册》提出单表行数超过 500 万行或者单表容量超过 2GB，才推荐进行分库分表。

事实上，这个数值和实际记录的条数无关，而与 MySQL 的配置以及机器的硬件有关。因为MySQL为了提高性能，会将表的索引装载到内存中。在InnoDB buffer size 足够的情况下，其能完成全加载进内存，查询不会有问题。但是，当单表数据库到达某个量级的上限时，导致内存无法存储其索引，使得之后的 SQL 查询会产生磁盘 IO，从而导致性能下降。当然，这个还有具体的表结构的设计有关，最终导致的问题都是内存限制。

因此，对于分库分表，需要结合实际需求，不宜过度设计，在项目一开始不采用分库与分表设计，而是随着业务的增长，在无法继续优化的情况下，再考虑分库与分表提高系统的性能。对此，阿里巴巴《Java 开发手册》补充到：如果预计三年后的数据量根本达不到这个级别，请不要在创建表时就分库分表。

至于MySQL单表多大进行分库分表，应当根据机器资源进行评估。

35. 大表查询慢怎么优化？

某个表有近千万数据，查询比较慢，如何优化？

当MySQL单表记录数过大时，数据库的性能会明显下降，一些常见的优化措施如下：

- 合理建立索引。在合适的字段上建立索引，例如在WHERE和ORDER BY命令上涉及的列建立索引，可根据EXPLAIN来查看是否用了索引还是全表扫描
- 建立分区。对关键字段建立水平分区，比如时间字段，若查询条件往往通过时间范围来进行查询，能提升不少性能
- 利用缓存。利用Redis等缓存热点数据，提高查询效率
- 限定数据的范围。比如：用户在查询历史信息的时候，可以控制在一个月的时间范围内
- 读写分离。经典的数据库拆分方案，主库负责写，从库负责读
- 通过分库分表的方式进行优化，主要有垂直拆分和水平拆分

36. 说说count(1)、count(*)和count(字段名)的区别

嗯，先说说count(1) and count(字段名)的区别。

两者的主要区别是

1. count(1) 会统计表中的所有的记录数，包含字段为null 的记录。
2. count(字段名) 会统计该字段在表中出现的次数，忽略字段为null 的情况。即不统计字段为null 的记录。

接下来看看三者之间的区别。

执行效果上：

- count(*)包括了所有的列，相当于行数，在统计结果的时候，**不会忽略列值为NULL**
- count(1)包括了忽略所有列，用1代表代码行，在统计结果的时候，**不会忽略列值为NULL**
- count(字段名)只包括列名那一列，在统计结果的时候，会忽略列值为空（这里的空不是只空字符串或者0，而是表示null）的计数，**即某个字段值为NULL时，不统计。**

执行效率上：

- 列名为主键，count(字段名)会比count(1)快
- 列名不为主键，count(1)会比count(列名)快
- 如果表多个列并且没有主键，则 count(1) 的执行效率优于 count(*)
- 如果有主键，则 select count(主键)的执行效率是最优的
- 如果表只有一个字段，则 select count(*)最优。

37. MySQL中DATETIME 和 TIMESTAMP有什么区别？

嗯，TIMESTAMP 和 DATETIME 都可以用来存储时间，它们主要有以下区别：

1.表示范围

- DATETIME：1000-01-01 00:00:00.000000 到 9999-12-31 23:59:59.999999
- TIMESTAMP：'1970-01-01 00:00:01.000000' UTC 到 '2038-01-09 03:14:07.999999' UTC

TIMESTAMP 支持的时间范围比 DATETIME 要小，容易出现超出的情况。

2.空间占用

- `TIMESTAMP`：占 4 个字节
- `DATETIME`：在 MySQL 5.6.4 之前，占 8 个字节，之后版本，占 5 个字节

3.存入时间是否会自动转换

`TIMESTAMP` 类型在默认情况下，insert、update 数据时，`TIMESTAMP` 列会自动以当前时间（`CURRENT_TIMESTAMP`）填充/更新。`DATETIME` 则不会做任何转换，也不会检测时区，你给什么数据，它存什么数据。

4. `TIMESTAMP` 比较受时区timezone的影响以及MySQL版本和服务器的SQL MODE的影响。因为 `TIMESTAMP` 存的是时间戳，在不同的时区得出的时间不一致。

5.如果存进NULL，两者实际存储的值不同。

- `TIMESTAMP`：会自动存储当前时间 now()。
- `DATETIME`：不会自动存储当前时间，会直接存入 NULL 值。

38. 说说为什么不建议用外键？

外键是一种约束，这个约束的存在，会保证表间数据的关系始终完整。外键的存在，并非全然没有优点。

外键可以保证数据的完整性和一致性，级联操作方便。而且使用外键可以将数据完整性判断托付给了数据库完成，减少了程序的代码量。

虽然外键能够保证数据的完整性，但是会给系统带来很多缺陷。

- 1、并发问题。在使用外键的情况下，每次修改数据都需要去另外一个表检查数据，需要获取额外的锁。若是在高并发大流量事务场景，使用外键更容易造成死锁。
- 2、扩展性问题。比如从 `MySQL` 迁移到 `Oracle`，外键依赖于数据库本身的特性，做迁移可能不方便。
- 3、不利于分库分表。在水平拆分和分库的情况下，外键是无法生效的。将数据间关系的维护，放入应用程序中，为将来的分库分表省去很多的麻烦。

39. 使用自增主键有什么好处？

自增主键可以让主键索引尽量地保持递增顺序插入，避免了页分裂，因此索引更紧凑，在查询的时候，效率也就更高。

40. 自增主键保存在什么地方？

不同的引擎对于自增值的保存策略不同：

- MyISAM引擎的自增值保存在数据文件中。
- 在MySQL8.0以前，InnoDB引擎的自增值是存在内存中。MySQL重启之后内存中的这个值就丢失了，每次重启后第一次打开表的时候，会找自增值的最大值max(id)，然后将最大值加1作为这个表的自增值；MySQL8.0版本会将自增值的变更记录在redo log中，重启时依靠redo log恢复。

41. 自增主键一定是连续的吗？

不一定，有几种情况会导致自增主键不连续。

1、唯一键冲突导致自增主键不连续。当我们向一个自增主键的InnoDB表中插入数据的时候，如果违反表中定义的唯一索引的唯一约束，会导致插入数据失败。此时表的自增主键的键值是会后加1滚动的。下次再次插入数据的时候，就不能再使用上次因插入数据失败而滚动生成的键值了，必须使用新滚动生成的键值。

2、事务回滚导致自增主键不连续。当我们向一个自增主键的InnoDB表中插入数据的时候，如果显式开启了事务，然后因为某种原因最后回滚了事务，此时表的自增值也会发生滚动，而接下里新插入的数据，也将不能使用滚动过的自增值，而是需要重新申请一个新的自增值。

3、批量插入导致自增值不连续。MySQL有一个批量申请自增id的策略：

- 语句执行过程中，第一次申请自增id，分配1个自增id
- 1个用完以后，第二次申请，会分配2个自增id
- 2个用完以后，第三次申请，会分配4个自增id
- 依次类推，每次申请都是上一次的两倍（最后一次申请不一定全部使用）

如果下一个事务再次插入数据的时候，则会基于上一个事务申请后的自增值基础上再申请。此时就出现自增值不连续的情况出现。

4、自增步长不是1，也会导致自增主键不连续。

42. InnoDB的自增值为什么不能回收利用？

主要为了提升插入数据的效率和并行度。

假设有两个并行执行的事务，在申请自增值的时候，为了避免两个事务申请到相同的自增 id，肯定要加锁，然后顺序申请。

假设事务 A 申请到了 id=2，事务 B 申请到 id=3，那么这时候表 t 的自增值是 4，之后继续执行。

事务 B 正确提交了，但事务 A 出现了唯一键冲突。

如果允许事务 A 把自增 id 回退，也就是把表 t 的当前自增值改回 2，那么就会出现这样的情况：表里面已经有 id=3 的行，而当前的自增 id 值是 2。

接下来，继续执行的其他事务就会申请到 id=2，然后再申请到 id=3。这时，就会出现插入语句报错“主键冲突”。

而为了解决这个主键冲突，有两种方法：

- 每次申请 id 之前，先判断表里面是否已经存在这个 id。如果存在，就跳过这个 id。但是，这个方法成本很高。因为，本来申请 id 是一个很快的操作，现在还要再去主键索引树上判断 id 是否存在。
- 把自增 id 的锁范围扩大，必须等到一个事务执行完成并提交，下一个事务才能再申请自增 id。这个方法的问题，就是锁的粒度太大，系统并发能力大大下降。

可见，这两个方法都会导致性能问题。

因此，InnoDB 放弃了“允许自增 id 回退”这个设计，语句执行失败也不回退自增 id。

43. MySQL数据如何同步到Redis缓存？

参考：<https://cloud.tencent.com/developer/article/1805755>

有两种方案：

1、通过MySQL自动同步刷新Redis，**MySQL触发器+UDF函数**实现。

过程大致如下：

1. 在MySQL中对要操作的数据设置触发器Trigger，监听操作
2. 客户端向MySQL中写入数据时，触发器会被触发，触发之后调用MySQL的UDF函数
3. UDF函数可以把数据写入到Redis中，从而达到同步的效果

2、**解析MySQL的binlog**，实现将数据库中的数据同步到Redis。可以通过canal实现。canal是阿里巴巴旗下的一款开源项目，基于数据库增量日志解析，提供增量数据订阅&消费。

canal的原理如下：

1. canal模拟mysql slave的交互协议，伪装自己为mysql slave，向mysql master发送dump协议
2. mysql master收到dump请求，开始推送binary log给canal
3. canal解析binary log对象（原始为byte流），将数据同步写入Redis。

44. 为什么阿里Java手册禁止使用存储过程？

先看看什么是存储过程。

存储过程是在大型数据库系统中，一组为了完成特定功能的SQL 语句集，它存储在数据库中，一次编译后永久有效，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。

存储过程主要有以下几个缺点。

1. **存储过程难以调试。**存储过程的开发一直缺少有效的 IDE 环境。SQL 本身经常很长，调试式要把句子拆开分别独立执行，非常麻烦。
2. **移植性差。**存储过程的移植困难，一般业务系统总会不可避免地用到数据库独有的特性和语法，更换数据库时这部分代码就需要重写，成本较高。
3. **管理困难。**存储过程的目录是扁平的，而不是文件系统那样的树形结构，脚本少的时候还好办，一旦多起来，目录就会陷入混乱。
4. 存储过程是**只优化一次**，有的时候随着数据量的增加或者数据结构的变化，原来存储过程选择的执行计划也许并不是最优的了，所以这个时候需要手动干预或者重新编译了。



扫码关注我



微信搜索



程序员大彬

公众号后台回复【面试】获取面试手册
PDF最新版

Redis

1. Redis是什么？

Redis (**Remote Dictionary Server**) 是一个使用 C 语言编写的，高性能非关系型的键值对数据库。与传统数据库不同的是，Redis 的数据是存在内存中的，所以读写速度非常快，被广泛应用于缓存方向。Redis可以将数据写入磁盘中，保证了数据的安全不丢失，而且Redis的操作是原子性的。

2. Redis优缺点？

优点：

1. **基于内存操作**，内存读写速度快。
2. **支持多种数据类型**，包括String、Hash、List、Set、ZSet等。
3. **支持持久化**。Redis支持RDB和AOF两种持久化机制，持久化功能可以有效地避免数据丢失问题。
4. **支持事务**。Redis的所有操作都是原子性的，同时Redis还支持对几个操作合并后的原子性执行。
5. **支持主从复制**。主节点会自动将数据同步到从节点，可以进行读写分离。
6. Redis命令的处理是**单线程**的。Redis6.0引入了多线程，需要注意的是，**多线程用于处理网络数据的读写和协议解析**，Redis命令执行还是单线程的。

缺点：

1. 对结构化查询的支持比较差。
2. 数据库容量受到物理内存的限制，不适合用作海量数据的高性能读写，因此Redis适合的场景主要局限在较小数据量的操作。
3. Redis 较难支持在线扩容，在集群容量达到上限时在线扩容会变得更复杂。

3. Redis为什么这么快？

- **基于内存**：Redis是使用内存存储，没有磁盘IO上的开销。数据存在内存中，读写速度快。
- **IO多路复用模型**：Redis 采用 IO 多路复用技术。Redis 使用单线程来轮询描述符，将数据库的操作都转换成了事件，不在网络I/O上浪费过多的时间。
- **高效的数据结构**：Redis 每种数据类型底层都做了优化，目的就是追求更快的速度。

4. 讲讲Redis的线程模型？

Redis基于Reactor模式开发了网络事件处理器，这个处理器被称为文件事件处理器。它的组成结构为4部分：多个套接字、IO多路复用程序、文件事件分派器、事件处理器。因为文件事件分派器队列的消费是单线程的，所以Redis才叫单线程模型。

- 文件事件处理器使用I/O多路复用（multiplexing）程序来同时监听多个套接字，并根据套接字目前执行的任务来为套接字关联不同的事件处理器。
- 当被监听的套接字准备好执行连接accept、read、write、close等操作时，与操作相对应的文件事件就会产生，这时文件事件处理器就会调用套接字之前关联好的事件处理器来处理这些事件。

虽然文件事件处理器以单线程方式运行，但通过使用 I/O 多路复用程序来监听多个套接字，文件事件处理器既实现了高性能的网络通信模型，又可以很好地与 redis 服务器中其他同样以单线程方式运行的模块进行对接，这保持了 Redis 内部单线程设计的简单性。

5. Redis应用场景有哪些？

1. **缓存热点数据**，缓解数据库的压力。
2. 利用 Redis 原子性的自增操作，可以实现**计数器**的功能，比如统计用户点赞数、用户访问数等。

3. **简单的消息队列**，可以使用Redis自身的发布/订阅模式或者List来实现简单的消息队列，实现异步操作。
4. **限速器**，可用于限制某个用户访问某个接口的频率，比如秒杀场景用于防止用户快速点击带来不必要的压力。
5. **好友关系**，利用集合的一些命令，比如交集、并集、差集等，实现共同好友、共同爱好之类的功能。

6. Memcached和Redis的区别？

1. MemCached 数据结构单一，仅用来缓存数据，而 **Redis 支持多种数据类型**。
2. MemCached 不支持数据持久化，重启后数据会消失。**Redis 支持数据持久化**。
3. **Redis 提供主从同步机制和 cluster 集群部署能力**，能够提供高可用服务。Memcached 没有提供原生的集群模式，需要依靠客户端实现往集群中分片写入数据。
4. Redis 的速度比 Memcached 快很多。
5. Redis 使用**单线程的多路 IO 复用模型**，Memcached使用多线程的非阻塞 IO 模型。（Redis6.0引入了多线程IO，**用来处理网络数据的读写和协议解析**，但是命令的执行仍然是单线程）
6. value 值大小不同：Redis 最大可以达到 512M；memcache 只有 1mb。

7. 为什么要用 Redis 而不用 map/guava 做缓存？

使用自带的 map 或者 guava 实现的是**本地缓存**，最主要的特点是轻量以及快速，生命周期随着 jvm 的销毁而结束，并且在多实例的情况下，每个实例都需要各自保存一份缓存，缓存不具有**一致性**。

使用 redis 或 memcached 之类的称为**分布式缓存**，在多实例的情况下，各实例共用一份缓存数据，缓存具有**一致性**。

8. Redis 数据类型有哪些？

基本数据类型：

- 1、**String**：最常用的一种数据类型，String类型的值可以是字符串、数字或者二进制，但值最大不能超过512MB。
- 2、**Hash**：Hash 是一个键值对集合。
- 3、**Set**：无序去重的集合。Set 提供了交集、并集等方法，对于实现共同好友、共同关注等功能特别方便。
- 4、**List**：有序可重复的集合，底层是依赖双向链表实现的。
- 5、**SortedSet**：有序Set。内部维护了一个 **score** 的参数来实现。适用于排行榜和带权重的消息队列等场景。

特殊的数据类型：

- 1、**Bitmap**：位图，可以认为是一个以位为单位数组，数组中的每个单元只能存0或者1，数组的下标在Bitmap 中叫做偏移量。Bitmap的长度与集合中元素个数无关，而是与基数的上限有关。
- 2、**Hyperloglog**。HyperLogLog 是用来做基数统计的算法，其优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定的、并且是很小的。典型的使用场景是统计独立访客。
- 3、**Geospatial**：主要用于存储地理位置信息，并对存储的信息进行操作，适用场景如定位、附近的人等。

9. SortedSet和List异同点？

相同点：

1. 都是有序的；
2. 都可以获得某个范围内的元素。

不同点：

1. 列表基于链表实现，获取两端元素速度快，访问中间元素速度慢；
2. 有序集合基于散列表和跳跃表实现，访问中间元素时间复杂度是 $O(\log N)$ ；
3. 列表不能简单的调整某个元素的位置，有序列表可以（更改元素的分数）；
4. 有序集合更耗内存。

10. Redis的内存用完了会怎样？

如果达到设置的上限，Redis的写命令会返回错误信息（但是读命令还可以正常返回）。

也可以配置内存淘汰机制，当Redis达到内存上限时会冲刷掉旧的内容。

11. Redis如何做内存优化？

可以好好利用Hash,list,sorted set,set等集合类型数据，因为通常情况下很多小的Key-Value可以用更紧凑的方式存放到一起。尽可能使用散列表（hashes），散列表（是说散列表里面存储的数少）使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。比如你的web系统中有一个用户对象，不要为这个用户的名称，姓氏，邮箱，密码设置单独的key，而是应该把这个用户的所有信息存储到一张散列表里面。

12. keys命令存在的问题？

redis的单线程的。keys指令会导致线程阻塞一段时间，直到执行完毕，服务才能恢复。scan采用渐进式遍历的方式来解决keys命令可能带来的阻塞问题，每次scan命令的时间复杂度是 $O(1)$ ，但是要真正实现keys的功能，需要执行多次scan。

scan的缺点：在scan的过程中如果有键的变化（增加、删除、修改），遍历过程可能会有以下问题：新增的键可能没有遍历到，遍历出了重复的键等情况，也就是说scan并不能保证完整的遍历出来所有的键。

13. Redis事务

事务的原理是将一个事务范围内的若干命令发送给Redis，然后再让Redis依次执行这些命令。

事务的生命周期：

1. 使用MULTI开启一个事务
2. 在开启事务的时候，每次操作的命令将会被插入到一个队列中，同时这个命令并不会被真的执行
3. EXEC命令进行提交事务

```
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> SET name tyson
QUEUED
127.0.0.1:6379> SET name sophia
QUEUED
127.0.0.1:6379> EXEC
1) OK
2) OK
```

一个事务范围内某个命令出错不会影响其他命令的执行，不保证原子性：

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set a 1
QUEUED
127.0.0.1:6379> set b 1 2
QUEUED
127.0.0.1:6379> set c 3
QUEUED
127.0.0.1:6379> exec
1) OK
2) (error) ERR syntax error
3) OK
```

WATCH命令

WATCH 命令可以监控一个或多个键，一旦其中有一个键被修改，之后的事务就不会执行（类似于乐观锁）。执行 **EXEC** 命令之后，就会自动取消监控。

```
127.0.0.1:6379> watch name
OK
127.0.0.1:6379> set name 1
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set name 2
QUEUED
127.0.0.1:6379> set gender 1
QUEUED
127.0.0.1:6379> exec
(nil)
127.0.0.1:6379> get gender
(nil)
```

比如上面的代码中：

1. **watch name** 开启了对 **name** 这个 **key** 的监控
2. 修改 **name** 的值
3. 开启事务a
4. 在事务a中设置了 **name** 和 **gender** 的值
5. 使用 **EXEC** 命令提交事务
6. 使用命令 **get gender** 发现不存在，即事务a没有执行

使用 **UNWATCH** 可以取消 **WATCH** 命令对 **key** 的监控，所有监控锁将会被取消。

14. Redis事务支持隔离性吗？

Redis 是单进程程序，并且它保证在执行事务时，不会对事务进行中断，事务可以运行直到执行完所有事务队列中的命令为止。因此，Redis 的事务是总是带有隔离性的。

15. Redis事务保证原子性吗，支持回滚吗？

Redis单条命令是原子性执行的，但事务不保证原子性，且没有回滚。事务中任意命令执行失败，其余的命令仍会被执行。

16. 持久化机制

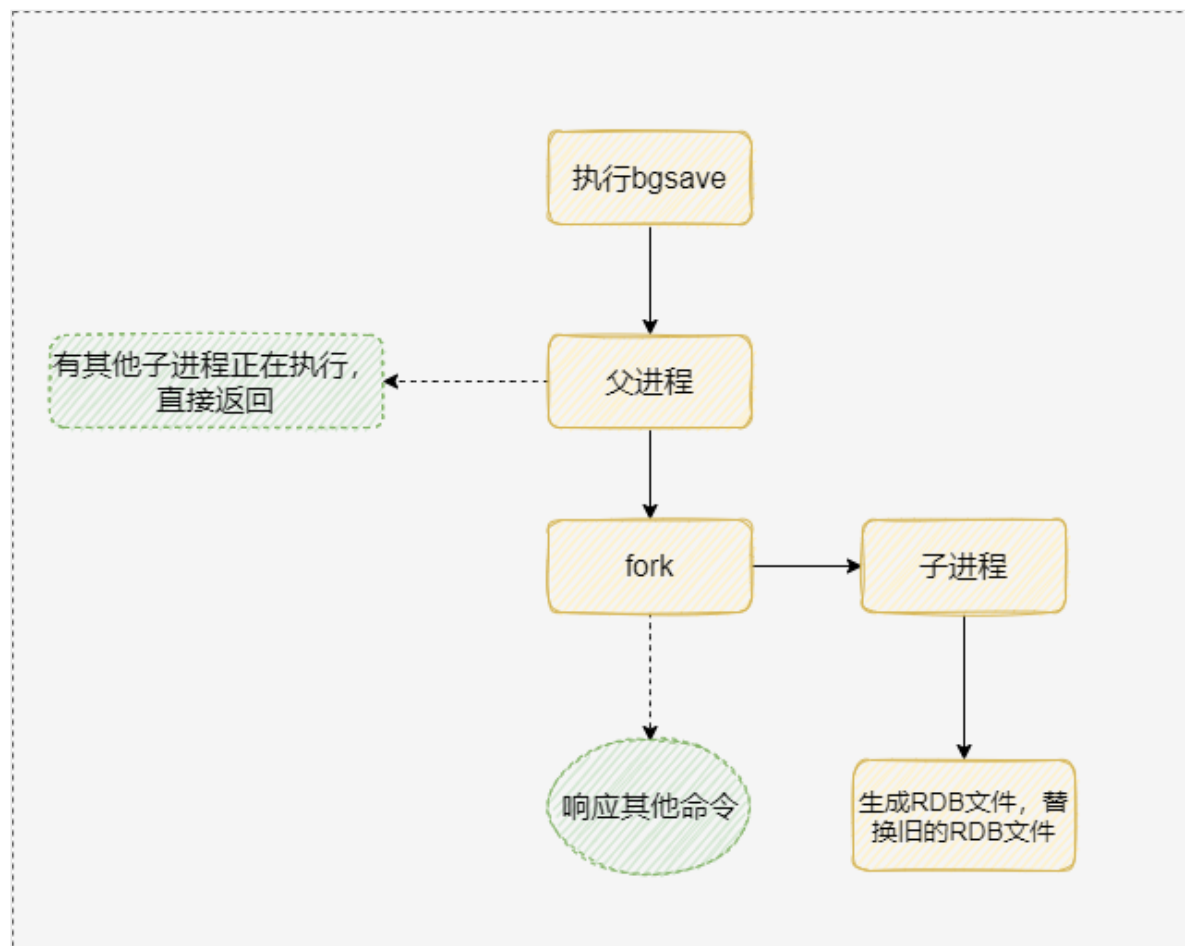
持久化就是把**内存的数据写到磁盘中**，防止服务宕机导致内存数据丢失。

Redis支持两种方式的持久化，一种是 **RDB** 的方式，一种是 **AOF** 的方式。**前者会根据指定的规则定时将内存中的数据存储到硬盘上**，而**后者在每次执行完命令后将命令记录下来**。一般将两者结合使用。

RDB方式

RDB 是 Redis 默认的持久化方案。RDB持久化时会将内存中的数据写入到磁盘中，在指定目录下生成一个 **dump.rdb** 文件。Redis 重启会加载 **dump.rdb** 文件恢复数据。

bgsave 是主流的触发 RDB 持久化的方式，执行过程如下：



- 执行 **BGSAVE** 命令
- Redis 父进程判断当前**是否存在正在执行的子进程**，如果存在，**BGSAVE** 命令直接返回。
- 父进程执行 **fork** 操作**创建子进程**，fork操作过程中父进程会阻塞。
- 父进程 **fork** 完成后，**父进程继续接收并处理客户端的请求**，而**子进程开始将内存中的数据写进硬盘的临时文件**；
- 当子进程写完所有数据后会**用该临时文件替换旧的 RDB 文件**。

Redis启动时会读取RDB快照文件，将数据从硬盘载入内存。通过 RDB 方式的持久化，一旦Redis异常退出，就会丢失最近一次持久化以后更改的数据。

触发 RDB 持久化的方式：

1. **手动触发**：用户执行 **SAVE** 或 **BGSAVE** 命令。**SAVE** 命令执行快照的过程会阻塞所有客户端的请求，应避免在生产环境使用此命令。**BGSAVE** 命令可以在后台异步进行快照操作，快照的同时服务器还可以继续响应客户端的请求，因此需要手动执行快照时推荐使用 **BGSAVE** 命令。
2. **被动触发**：
 - 根据配置规则进行自动快照，如 **SAVE 100 10**，100秒内至少有10个键被修改则进行快照。
 - 如果从节点执行全量复制操作，主节点会自动执行 **BGSAVE** 生成 RDB 文件并发送给从节点。

- 默认情况下执行 `shutdown` 命令时，如果没有开启 AOF 持久化功能则自动执行 `BGSAVE`。

优点：

1. **Redis 加载 RDB 恢复数据远远快于 AOF 的方式。**
2. 使用单独子进程来进行持久化，主进程不会进行任何 IO 操作，**保证了 Redis 的高性能。**

缺点：

1. **RDB方式数据无法做到实时持久化。** 因为 `BGSAVE` 每次运行都要执行 `fork` 操作创建子进程，属于重量级操作，频繁执行成本比较高。
2. RDB 文件使用特定二进制格式保存，Redis 版本升级过程中有多个格式的 RDB 版本，**存在老版本 Redis 无法兼容新版 RDB 格式的问题。**

AOF方式

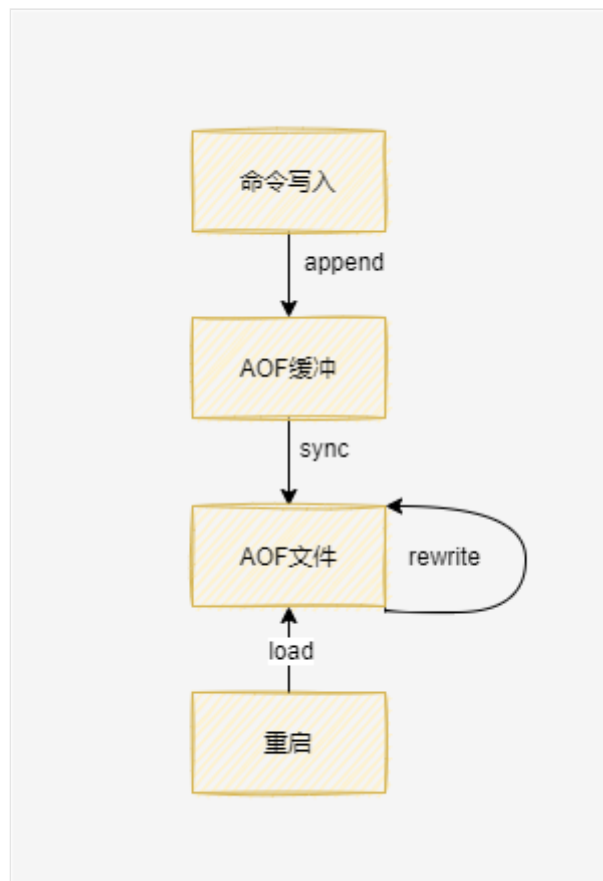
AOF (append only file) 持久化：以独立日志的方式记录每次写命令，Redis重启时会重新执行AOF文件中的命令达到恢复数据的目的。AOF的主要作用是**解决了数据持久化的实时性**，AOF 是Redis持久化的主流方式。

默认情况下Redis没有开启AOF方式的持久化，可以通过 `appendonly` 参数启用：`appendonly yes`。开启AOF方式持久化后每执行一条写命令，Redis就会将该命令写进 `aof_buf` 缓冲区，AOF缓冲区根据对应的策略向硬盘做同步操作。

默认情况下系统**每30秒**会执行一次同步操作。为了防止缓冲区数据丢失，可以在Redis写入AOF文件后主动要求系统将缓冲区数据同步到硬盘上。可以通过 `appendfsync` 参数设置同步的时机。

```
appendfsync always //每次写入aof文件都会执行同步，最安全最慢，不建议配置
appendfsync everysec //既保证性能也保证安全，建议配置
appendfsync no //由操作系统决定何时进行同步操作
```

接下来看一下 AOF 持久化执行流程：



1. 所有的写入命令会追加到 AOF 缓冲区中。
2. AOF 缓冲区根据对应的策略向硬盘同步。
3. 随着 AOF 文件越来越大，需要定期对 AOF 文件进行重写，达到压缩文件体积的目的。AOF文件重写是把Redis进程内的数据转化为写命令同步到新AOF文件的过程。
4. 当 Redis 服务器重启时，可以加载 AOF 文件进行数据恢复。

优点：

1. AOF可以更好的保护数据不丢失，可以配置 AOF 每秒执行一次 **fsync** 操作，如果Redis进程挂掉，最多丢失1秒的数据。
2. AOF以 **append-only** 的模式写入，所以没有磁盘寻址的开销，写入性能非常高。

缺点：

1. 对于同一份文件AOF文件比RDB数据快照要大。
2. 数据恢复比较慢。

17. RDB和AOF如何选择？

通常来说，应该同时使用两种持久化方案，以保证数据安全。

- 如果数据不敏感，且可以从其他地方重新生成，可以关闭持久化。
- 如果数据比较重要，且能够承受几分钟的数据丢失，比如缓存等，只需要使用RDB即可。
- 如果是用做内存数据，要使用Redis的持久化，建议是RDB和AOF都开启。
- 如果只用AOF，优先使用everysec的配置选择，因为它在可靠性和性能之间取了一个平衡。

当RDB与AOF两种方式都开启时，Redis会优先使用AOF恢复数据，因为AOF保存的文件比RDB文件更完整。

18. Redis有哪些部署方案？

单机版：单机部署，单机redis能够承载的 QPS 大概就在上万到几万不等。这种部署方式很少使用。存在的问题：1、内存容量有限 2、处理能力有限 3、无法高可用。

主从模式：一主多从，主负责写，并且将数据复制到其它的 slave 节点，从节点负责读。所有的读请求全部走从节点。这样也可以很轻松实现水平扩容，支撑读高并发。master 节点挂掉后，需要手动指定新的 master，可用性不高，基本不用。

哨兵模式：主从复制存在不能自动故障转移、达不到高可用的问题。哨兵模式解决了这些问题。通过哨兵机制可以自动切换主从节点。master 节点挂掉后，哨兵进程会主动选举新的 master，可用性高，但是每个节点存储的数据是一样的，浪费内存空间。数据量不是很多，集群规模不是很大，需要自动容错容灾的时候使用。

Redis cluster：服务端分片技术，3.0版本开始正式提供。Redis Cluster并没有使用一致性hash，而是采用slot(槽)的概念，一共分成16384个槽。将请求发送到任意节点，接收到请求的节点会将查询请求发送到正确的节点上执行。主要是针对海量数据+高并发+高可用的场景，如果是海量数据，如果你的数据量很大，那么建议就用Redis cluster，所有主节点的容量总和就是Redis cluster可缓存的数据容量。

19. 主从架构

单机的 redis，能够承载的 QPS 大概就在上万到几万不等。对于缓存来说，一般都是用来支撑读高并发的。因此架构做成主从(master-slave)架构，一主多从，主负责写，并且将数据复制到其它的 slave 节点，从节点负责读。所有的读请求全部走从节点。这样也可以很轻松实现水平扩容，支撑读高并发。

Redis的复制功能是支持多个数据库之间的数据同步。主数据库可以进行读写操作，当主数据库的数据发生变化时会自动将数据同步到从数据库。从数据库一般是只读的，它会接收主数据库同步过来的数据。一个主数据库可以有多个从数据库，而一个从数据库只能有一个主数据库。

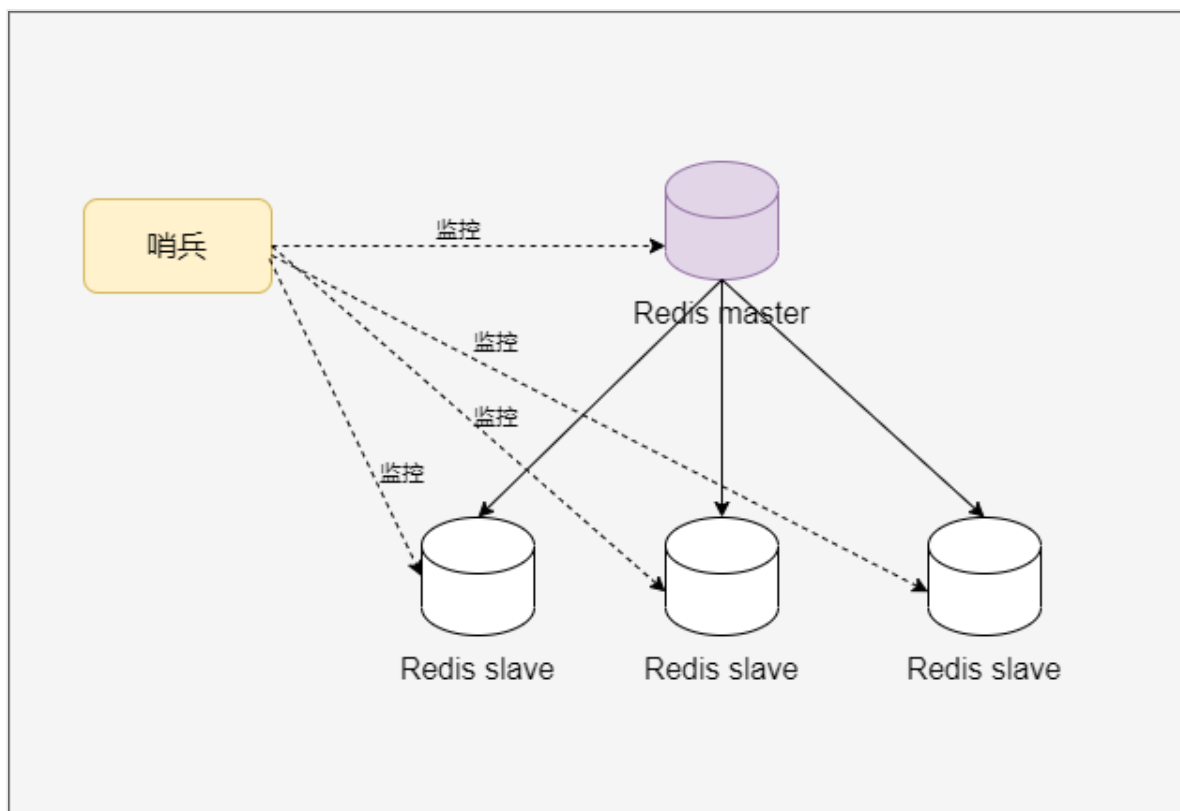
主从复制的原理？

1. 当启动一个从节点时，它会发送一个 **PSYNC** 命令给主节点；
2. 如果是从节点初次连接到主节点，那么会触发一次全量复制。此时主节点会启动一个后台线程，开始生成一份 **RDB** 快照文件；
3. 同时还会将从客户端 client 新收到的所有写命令缓存在内存中。**RDB** 文件生成完毕后，主节点会将 **RDB** 文件发送给从节点，从节点会先将 **RDB** 文件写入本地磁盘，然后再从本地磁盘加载到内存中；
4. 接着主节点会将内存中缓存的写命令发送到从节点，从节点同步这些数据；
5. 如果从节点跟主节点之间网络出现故障，连接断开了，会自动重连，连接之后主节点仅会将部分缺失的数据同步给从节点。

20. 哨兵Sentinel

主从复制存在不能自动故障转移、达不到高可用的问题。哨兵模式解决了这些问题。通过哨兵机制可以自动切换主从节点。

客户端连接Redis的时候，先连接哨兵，哨兵会告诉客户端Redis主节点的地址，然后客户端连接上Redis并进行后续的操作。当主节点宕机的时候，哨兵监测到主节点宕机，会重新推选出某个表现良好的从节点成为新的主节点，然后通过发布订阅模式通知其他的从服务器，让它们切换主机。



工作原理

- 每个 **Sentinel** 以每秒钟一次的频率向它所知道的 **Master**，**Slave** 以及其他 **Sentinel** 实例发送一个 **PING** 命令。
- 如果一个实例距离最后一次有效回复 **PING** 命令的时间超过指定值，则这个实例会被 **Sentinel** 标记为主观下线。
- 如果一个 **Master** 被标记为主观下线，则正在监视这个 **Master** 的所有 **Sentinel** 要以每秒一次的频率确认 **Master** 是否真正进入主观下线状态。
- 当有足够数量的 **Sentinel**（大于等于配置文件指定值）在指定的时间范围内确认 **Master** 的确进入了主观下线状态，则 **Master** 会被标记为客观下线。若没有足够数量的 **Sentinel** 同意 **Master**

已经下线，**Master** 的客观下线状态就会被解除。若 **Master** 重新向 **Sentinel** 的 **PING** 命令返回有效回复，**Master** 的主观下线状态就会被移除。

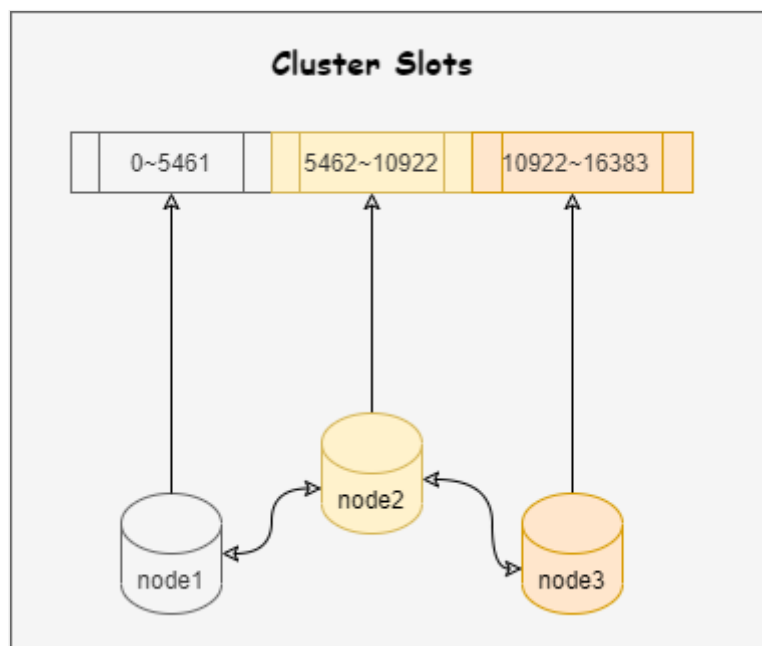
- 哨兵节点会选举出哨兵 leader，负责故障转移的工作。
- 哨兵 leader 会推选出某个表现良好的从节点成为新的主节点，然后通知其他从节点更新主节点信息。

21. Redis cluster

哨兵模式解决了主从复制不能自动故障转移、达不到高可用的问题，但还是存在主节点的写能力、容量受限于单机配置的问题。而cluster模式实现了Redis的分布式存储，每个节点存储不同的内容，解决主节点的写能力、容量受限于单机配置的问题。

Redis cluster集群节点最小配置6个节点以上（3主3从），其中主节点提供读写操作，从节点作为备用节点，不提供请求，只作为故障转移使用。

Redis cluster采用**虚拟槽分区**，所有的键根据哈希函数映射到0~16383个整数槽内，每个节点负责维护一部分槽以及槽所映射的键值数据。



工作原理:

1. 通过哈希的方式，将数据分片，每个节点均分存储一定哈希槽(哈希值)区间的数据，默认分配了 16384 个槽位
2. 每份数据分片会存储在多个互为主从的多节点上
3. 数据写入先写主节点，再同步到从节点(支持配置为阻塞同步)
4. 同一分片多个节点间的数据不保持一致性
5. 读取数据时，当客户端操作的key没有分配在该节点上时，redis会返回转向指令，指向正确的节点
6. 扩容时时需要需要把旧节点的数据迁移一部分到新节点

在 redis cluster 架构下，每个 redis 要放开两个端口号，比如一个是 6379，另外一个就是 加1w 的端口号，比如 16379。

16379 端口号是用来进行节点间通信的，也就是 cluster bus 的东西，cluster bus 的通信，用来进行故障检测、配置更新、故障转移授权。cluster bus 用了另外一种二进制的协议，**gossip** 协议，用于节点间进行高效的数据交换，占用更少的网络带宽和处理时间。

优点:

- 无中心架构，**支持动态扩容**；
- 数据按照 **slot** 存储分布在多个节点，节点间数据共享，**可动态调整数据分布**；

- **高可用性**。部分节点不可用时，集群仍可用。集群模式能够实现自动故障转移（failover），节点之间通过 **gossip** 协议交换状态信息，用投票机制完成 **Slave** 到 **Master** 的角色转换。

缺点：

- **不支持批量操作**（pipeline）。
- 数据通过异步复制，**不保证数据的强一致性**。
- **事务操作支持有限**，只支持多 **key** 在同一节点上的事务操作，当多个 **key** 分布于不同的节点上时无法使用事务功能。
- **key** 作为数据分区的最小粒度，不能将一个很大的键值对象如 **hash**、**list** 等映射到不同的节点。
- **不支持多数据库空间**，单机下的Redis可以支持到16个数据库，集群模式下只能使用1个数据库空间。
- 只能使用0号数据库。

哈希分区算法有哪些？

节点取余分区。使用特定的数据，如Redis的键或用户ID，对节点数量N取余： $\text{hash}(\text{key}) \% N$ 计算出哈希值，用来决定数据映射到哪一个节点上。优点是简单性。扩容时通常采用翻倍扩容，避免数据映射全部被打乱导致全量迁移的情况。

一致性哈希分区。为系统中每个节点分配一个token，范围一般在0~232，这些token构成一个哈希环。数据读写执行节点查找操作时，先根据key计算hash值，然后顺时针找到第一个大于等于该哈希值的token节点。这种方式相比节点取余最大的好处在于加入和删除节点只影响哈希环中相邻的节点，对其他节点无影响。

虚拟槽分区，所有的键根据哈希函数映射到0~16383整数槽内，计算公式： $\text{slot} = \text{CRC16}(\text{key}) \& 16383$ 。每一个节点负责维护一部分槽以及槽所映射的键值数据。**Redis Cluster**采用虚拟槽分区算法。

22. 过期键的删除策略？

- 1、**被动删除**。在访问key时，如果发现key已经过期，那么会将key删除。
- 2、**主动删除**。定时清理key，每次清理会依次遍历所有DB，从db随机取出20个key，如果过期就删除，如果其中有5个key过期，那么就继续对这个db进行清理，否则开始清理下一个db。
- 3、**内存不够时清理**。Redis有最大内存的限制，通过maxmemory参数可以设置最大内存，当使用的内存超过了设置的最大内存，就要进行内存释放，在进行内存释放的时候，会按照配置的淘汰策略清理内存。

23. 内存淘汰策略有哪些？

当Redis的内存超过最大允许的内存之后，Redis 会触发内存淘汰策略，删除一些不常用的数据，以保证Redis服务器正常运行。

Redisv4.0前提供 6 种数据淘汰策略：

- **volatile-lru**：LRU（**Least Recently Used**），最近使用。利用LRU算法移除设置了过期时间的key
- **allkeys-lru**：当内存不足以容纳新写入数据时，从数据集中移除最近最少使用的key
- **volatile-ttl**：从已设置过期时间的数据集中挑选将要过期的数据淘汰
- **volatile-random**：从已设置过期时间的数据集中任意选择数据淘汰
- **allkeys-random**：从数据集中任意选择数据淘汰
- **no-eviction**：禁止删除数据，当内存不足以容纳新写入数据时，新写入操作会报错

Redisv4.0后增加以下两种：

- **volatile-lfu**: LFU, Least Frequently Used, 最少使用, 从已设置过期时间的数据集中挑选最不经使用的数据淘汰。
- **allkeys-lfu**: 当内存不足以容纳新写入数据时, 从数据集中移除最不经使用的key。

内存淘汰策略可以通过配置文件来修改, 相应的配置项是 `maxmemory-policy`, 默认配置是 `noeviction`。

24. 如何保证缓存与数据库双写时的数据一致性?

1、先删除缓存再更新数据库

进行更新操作时, 先删除缓存, 然后更新数据库, 后续的请求再次读取时, 会从数据库读取后再将新数据更新到缓存。

存在的问题: 删除缓存数据之后, 更新数据库完成之前, 这个时间段内如果有新的读请求过来, 就会从数据库读取旧数据重新写到缓存中, 再次造成不一致, 并且后续读的都是旧数据。

2、先更新数据库再删除缓存

进行更新操作时, 先更新MySQL, 成功之后, 删除缓存, 后续读取请求时再将新数据回写缓存。

存在的问题: 更新MySQL和删除缓存这段时间内, 请求读取的还是缓存的旧数据, 不过等数据库更新完成, 就会恢复一致, 影响相对比较小。

3、异步更新缓存

数据库的更新操作完成后不直接操作缓存, 而是把这个操作命令封装成消息扔到消息队列中, 然后由Redis自己去消费更新数据, 消息队列可以保证数据操作顺序一致性, 确保缓存系统的数据正常。

以上几个方案都不完美, 需要根据业务需求, 评估哪种方案影响较小, 然后选择相应的方案。

25. 缓存常见问题

25.1. 缓存穿透

缓存穿透是指查询一个**不存在的数据**, 由于缓存是不命中时被动写的, 如果从DB查不到数据则不写入缓存, 这将导致这个不存在的数据每次请求都要到DB去查询, 失去了缓存的意义。在流量大时, 可能DB就挂掉了。

怎么解决?

1. **缓存空值**, 不会查数据库。
2. 采用**布隆过滤器**, 将所有可能存在的数据哈希到一个足够大的 `bitmap` 中, 查询不存在的数据会被这个 `bitmap` 拦截掉, 从而避免了对 `DB` 的查询压力。

布隆过滤器的原理: 当一个元素被加入集合时, 通过K个哈希函数将这个元素映射成一个位数组中的K个点, 把它们置为1。查询时, 将元素通过哈希函数映射之后会得到k个点, 如果这些点有任何一个0, 则被检元素一定不在, 直接返回; 如果都是1, 则查询元素很可能存在, 就会去查询Redis和数据库。

布隆过滤器一般用于在大数据量的集合中判定某元素是否存在。

25.2. 缓存雪崩

缓存雪崩是指在我们设置缓存时采用了相同的过期时间, **导致缓存存在某一时刻同时失效**, 请求全部转发到DB, DB瞬时压力过重挂掉。

解决方法:

1. 在原有的失效时间基础上**增加一个随机值**, 使得过期时间分散一些。这样每一个缓存的过期时间的重复率就会降低, 就很难引发集体失效的事件。

2. **加锁排队可以起到缓冲的作用**，防止大量的请求同时操作数据库，但它的缺点是**增加了系统的响应时间，降低了系统的吞吐量**，牺牲了一部分用户体验。当缓存未查询到时，对要请求的 key 进行加锁，只允许一个线程去数据库中查，其他线程等候排队。
3. 设置二级缓存。二级缓存指的是除了 Redis 本身的缓存，**再设置一层缓存**，当 Redis 失效之后，先去查询二级缓存。例如可以设置一个本地缓存，在 Redis 缓存失效的时候先去查询本地缓存而非查询数据库。

25.3. 缓存击穿

缓存击穿：大量的请求同时查询一个 key 时，此时这个 key 正好失效了，就会导致大量的请求都落到数据库。**缓存击穿是查询缓存中失效的 key，而缓存穿透是查询不存在的 key。**

解决方法：

- 1、**加互斥锁**。在并发的多个请求中，只有第一个请求线程能拿到锁并执行数据库查询操作，其他的线程拿不到锁就阻塞等着，等到第一个线程将数据写入缓存后，直接走缓存。可以使用 Redis 分布式锁实现，代码如下：

```
public String get(String key) {
    String value = redis.get(key);
    if (value == null) { //缓存值过期
        String unique_key = systemId + ":" + key;
        //设置30s的超时
        if (redis.set(unique_key, 1, 'NX', 'PX', 30000) == 1) { //设置成功
            value = db.get(key);
            redis.set(key, value, expire_secs);
            redis.del(unique_key);
        } else { //其他线程已经到数据库取值并回写到缓存了，可以重试获取缓存值
            sleep(50);
            get(key); //重试
        }
    } else {
        return value;
    }
}
```

- 2、**热点数据不过期**。直接将缓存设置为不过期，然后由定时任务去异步加载数据，更新缓存。这种方式适用于比较极端的场景，例如流量特别特别大的场景，使用时需要考虑业务能接受数据不一致的时间，还有就是异常情况的处理，保证缓存可以定时刷新。

25.4. 缓存预热

缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

解决方案：

1. 直接写个缓存刷新页面，上线时手工操作一下；
2. 数据量不大，可以在项目启动的时候自动进行加载；
3. 定时刷新缓存；

25.5. 缓存降级

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

缓存降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级；比如可以参考日志级别设置预案：

1. 一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；
2. 警告：有些服务在一段时间内成功率有波动（如在95~100%之间），可以自动降级或人工降级，并发送告警；
3. 错误：比如可用率低于90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阈值，此时可以根据情况自动降级或者人工降级；
4. 严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

服务降级的目的，是为了防止Redis服务故障，导致数据库跟着一起发生雪崩问题。因此，对于不重要的缓存数据，可以采取服务降级策略，例如一个比较常见的做法就是，Redis出现问题，不去数据库查询，而是直接返回默认值给用户。

26. Redis 怎么实现消息队列？

使用list类型保存数据信息，rpush生产消息，lpop消费消息，当lpop没有消息时，可以sleep一段时间，然后再检查有没有信息，如果不想sleep的话，可以使用blpop, 在没有信息的时候，会一直阻塞，直到信息的到来。

```
BLPOP queue 0 //0表示不限制等待时间
```

BLPOP和LPOP命令相似，唯一的区别就是当列表没有元素时BLPOP命令会一直阻塞连接，直到有新元素加入。

redis可以通过pub/sub**主题订阅模式**实现一个生产者，多个消费者，当然也存在一定的缺点，当消费者下线时，生产的消息会丢失。

```
PUBLISH channel1 hi
SUBSCRIBE channel1
UNSUBSCRIBE channel1 //退订通过SUBSCRIBE命令订阅的频道。
```

`PSUBSCRIBE channel1?*` 按照规则订阅。 `PUNSUBSCRIBE channel1?*` 退订通过PSUBSCRIBE命令按照某种规则订阅的频道。其中订阅规则要进行严格的字符串匹配，`PUNSUBSCRIBE *` 无法退订 `channel1?*` 规则。

27. Redis 怎么实现延时队列

使用sortedset，拿时间戳作为score，消息内容作为key，调用zadd来生产消息，消费者用`zrangebyscore`指令获取N秒之前的数据轮询进行处理。

28. pipeline的作用？

redis客户端执行一条命令分4个过程：发送命令、命令排队、命令执行、返回结果。使用 `pipeline` 可以批量请求，批量返回结果，执行速度比逐条执行要快。

使用 `pipeline` 组装的命令个数不能太多，不然数据量过大，增加客户端的等待时间，还可能造成网络阻塞，可以将大量命令的拆分多个小的 `pipeline` 命令完成。

原生批命令（`mset`和`mget`）与 `pipeline` 对比：

1. 原生批命令是原子性，`pipeline` 是**非原子性**。`pipeline`命令中途异常退出，之前执行成功的命令**不会回滚**。
2. 原生批命令只有一个命令，但 `pipeline` **支持多命令**。

29. LUA脚本

Redis 通过 LUA 脚本创建具有原子性的命令：当lua脚本命令正在运行的时候，不会有其他脚本或 Redis 命令被执行，实现组合命令的原子操作。

在Redis中执行Lua脚本有两种方法：`eval` 和 `evalsha`。`eval` 命令使用内置的 Lua 解释器，对 Lua 脚本进行求值。

```
//第一个参数是lua脚本，第二个参数是键名参数个数，剩下的是键名参数和附加参数
> eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second
1) "key1"
2) "key2"
3) "first"
4) "second"
```

lua脚本作用

- 1、Lua脚本在Redis中是原子执行的，执行过程中间不会插入其他命令。
- 2、Lua脚本可以将多条命令一次性打包，有效地减少网络开销。

应用场景

举例：限制接口访问频率。

在Redis维护一个接口访问次数的键值对，`key` 是接口名称，`value` 是访问次数。每次访问接口时，会执行以下操作：

- 通过 `aop` 拦截接口的请求，对接口请求进行计数，每次进来一个请求，相应的接口访问次数 `count` 加1，存入redis。
- 如果是第一次请求，则会设置 `count=1`，并设置过期时间。因为这里 `set()` 和 `expire()` 组合操作不是原子操作，所以引入 `lua` 脚本，实现原子操作，避免并发访问问题。
- 如果给定时间范围内超过最大访问次数，则会抛出异常。

```
private String buildLuaScript() {
    return "local c" +
        "\nc = redis.call('get',KEYS[1])" +
        "\nif c and tonumber(c) > tonumber(ARGV[1]) then" +
        "\nreturn c;" +
        "\nend" +
        "\nc = redis.call('incr',KEYS[1])" +
        "\nif tonumber(c) == 1 then" +
        "\nredis.call('expire',KEYS[1],ARGV[2])" +
        "\nend" +
        "\nreturn c;";
}

String luaScript = buildLuaScript();
```

```
RedisScript<Number> redisScript = new DefaultRedisScript<>(luaScript,
Number.class);
Number count = redisTemplate.execute(redisScript, keys, limit.count(),
limit.period());
```

PS：这种接口限流的实现方式比较简单，问题也比较多，一般不会使用，接口限流用的比较多的是令牌桶算法和漏桶算法。

30. 什么是RedLock?

Redis 官方站提出了一种权威的基于 Redis 实现分布式锁的方式名叫 *Redlock*，此种方式比原先的单节点的方法更安全。它可以保证以下特性：

1. 安全特性：互斥访问，即永远只有一个 client 能拿到锁
2. 避免死锁：最终 client 都可能拿到锁，不会出现死锁的情况，即使原本锁住某资源的 client 挂掉了
3. 容错性：只要大部分 Redis 节点存活就可以正常提供服务

31. Redis大key怎么处理?

通常我们会将含有较大数据或含有大量成员、列表数的Key称之为大Key。

以下是对各个数据类型大key的描述：

- value是STRING类型，它的值超过5MB
- value是ZSET、Hash、List、Set等集合类型时，它的成员数量超过1w个

上述的定义并不绝对，主要是根据value的成员数量和大小来确定，根据业务场景确定标准。

怎么处理：

1. 当value是string时，可以使用序列化、压缩算法将key的大小控制在合理范围内，但是序列化和反序列化都会带来更多时间上的消耗。或者将key进行拆分，一个大key分为不同的部分，记录每个部分的key，使用multiget等操作实现事务读取。
2. 当value是list/set等集合类型时，根据预估的数据规模来进行分片，不同的元素计算后分到不同的片。

32. Redis常见性能问题和解决方案?

1. Master最好不要做任何持久化工作，包括内存快照和AOF日志文件，特别是不要启用内存快照做持久化。
2. 如果数据比较关键，某个Slave开启AOF备份数据，策略为每秒同步一次。
3. 为了主从复制的速度和连接的稳定性，Slave和Master最好在同一个局域网内。
4. 尽量避免在压力较大的主库上增加从库
5. Master调用BGREWRITEAOF重写AOF文件，AOF在重写的时候会占大量的CPU和内存资源，导致服务load过高，出现短暂服务暂停现象。
6. 为了Master的稳定性，主从复制不要用图状结构，用单向链表结构更稳定，即主从关系为：
Master<-Slave1<-Slave2<-Slave3...，这样的结构也方便解决单点故障问题，实现Slave对Master的替换，也即，如果Master挂了，可以立马启用Slave1做Master，其他不变。

33. 说说为什么Redis过期了为什么内存没释放?

第一种情况，可能是覆盖之前的key，导致key过期时间发生了改变。

当一个key在Redis中已经存在了，但是由于一些误操作使得key过期时间发生了改变，从而导致这个key在应该过期的时间内并没有过期，从而造成内存的占用。

第二种情况是，Redis过期key的处理策略导致内存没释放。

一般Redis对过期key的处理策略有两种：惰性删除和定时删除。

先说惰性删除的情况

当一个key已经确定设置了xx秒过期同时中间也没有修改它，xx秒之后它确实已经过期了，但是惰性删除的策略它并不会马上删除这个key，而是当再次读写这个key时它才会去检查是否过期，如果过期了就会删除这个key。也就是说，惰性删除策略下，就算key过期了，也不会立刻释放内容，要等到下一次读写这个key才会删除key。

而定时删除会在一定时间内主动淘汰一部分已经过期的数据，默认的时间是每100ms过期一次。因为定时删除策略每次只会淘汰一部分过期key，而不是所有的过期key，如果redis中数据比较多的话要是一次性全量删除对服务器的压力比较大，每一次只挑一批进行删除，所以很可能出现部分已经过期的key并没有及时的被清理掉，从而导致内存没有即时被释放。

34. Redis突然变慢，有哪些原因？

1. **存在bigkey**。如果Redis实例中存储了 bigkey，那么在淘汰删除 bigkey 释放内存时，也会耗时比较长。应该避免存储 bigkey，降低释放内存的耗时。
2. 如果Redis 实例**设置了内存上限 maxmemory**，有可能导致 Redis 变慢。当 Redis 内存达到 maxmemory 后，每次写入新的数据之前，Redis 必须先从实例中踢出一部分数据，让整个实例的内存维持在 maxmemory 之下，然后才能把新数据写进来。
3. **开启了内存大页**。当 Redis 在执行后台 RDB 和 AOF rewrite 时，采用 fork 子进程的方式来处理。但主进程 fork 子进程后，此时的主进程依旧是可以接收写请求的，而进来的写请求，会采用 Copy On Write（写时复制）的方式操作内存数据。

什么是写时复制？

这样做的好处是，父进程有任何写操作，并不会影响子进程的数据持久化。

不过，主进程在拷贝内存数据时，会涉及到新内存的申请，如果此时操作系统开启了内存大页，那么在此期间，客户端即便只修改 10B 的数据，Redis 在申请内存时也会以 2MB 为单位向操作系统申请，申请内存的耗时变长，进而导致每个写请求的延迟增加，影响到 Redis 性能。

解决方案就是关闭内存大页机制。

4. **使用了Swap**。操作系统为了缓解内存不足对应用程序的影响，允许把一部分内存中的数据换到磁盘上，以达到应用程序对内存使用的缓冲，这些内存数据被换到磁盘上的区域，就是 Swap。当内存中的数据被换到磁盘上后，Redis 再访问这些数据时，就需要从磁盘上读取，访问磁盘的速度要比访问内存慢几百倍。尤其是针对 Redis 这种对性能要求极高、性能极其敏感的数据库来说，这个操作延时是无法接受的。解决方案就是增加机器的内存，让 Redis 有足够的内存可以使用。或者整理内存空间，释放出足够的内存供 Redis 使用
5. **网络带宽过载**。网络带宽过载的情况下，服务器在 TCP 层和网络层就会出现数据包发送延迟、丢包等情况。Redis 的高性能，除了操作内存之外，就在于网络 IO 了，如果网络 IO 存在瓶颈，那么也会严重影响 Redis 的性能。解决方案：1、及时确认占满网络带宽 Redis 实例，如果属于正常的业务访问，那就需要及时扩容或迁移实例了，避免因为这个实例流量过大，影响这个机器的其他实例。2、运维层面，需要对 Redis 机器的各项指标增加监控，包括网络流量，在网络流量达到一定阈值时提前报警，及时确认和扩容。
6. **频繁短连接**。频繁的短连接会导致 Redis 大量时间耗费在连接的建立和释放上，TCP 的三次握手和四次挥手同样也会增加访问延迟。应用应该使用长连接操作 Redis，避免频繁的短连接。

35. 为什么 Redis 集群的最大槽数是 16384 个？

Redis Cluster 采用数据分片机制，定义了 16384个 Slot槽位，集群中的每个Redis 实例负责维护一部分槽以及槽所映射的键值数据。

Redis每个节点之间会定期发送ping/pong消息（心跳包包含了其他节点的数据），用于交换数据信息。

Redis集群的节点会按照以下规则发ping消息：

- (1)每秒会随机选取5个节点，找出最久没有通信的节点发送ping消息
- (2)每100毫秒都会扫描本地节点列表，如果发现节点最近一次接受pong消息的时间大于cluster-node-timeout/2 则立刻发送ping消息

心跳包的消息头里面有个myslots的char数组，是一个bitmap，每一个位代表一个槽，如果该位为1，表示这个槽是属于这个节点的。

接下来，解答为什么 Redis 集群的最大槽数是 16384 个，而不是65536 个。

1、如果采用 16384 个插槽，那么心跳包的消息头占用空间 2KB (16384/8)；如果采用 65536 个插槽，那么心跳包的消息头占用空间 8KB (65536/8)。可见采用 65536 个插槽，**发送心跳信息的信息头达 8k，比较浪费带宽。**

2、一般情况下一个Redis集群**不会有超过1000个master节点**，太多可能导致网络拥堵。

3、哈希槽是通过一张bitmap的形式来保存的，在传输过程中，会对bitmap进行压缩。bitmap的填充率越低，**压缩率**越高。其中bitmap 填充率 = slots / N (N表示节点数)。所以，插槽数越低，填充率会降低，压缩率会提高。



扫码关注我



微信搜索



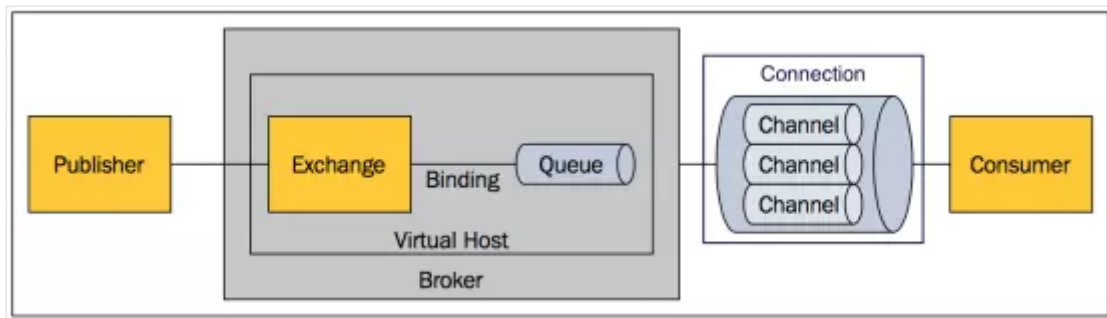
程序员大彬

公众号后台回复【面试】获取面试手册
PDF最新版

RabbitMQ

1. 简介

RabbitMQ是一个由erlang开发的消息队列。消息队列用于应用间的异步协作。



2. 基本概念

Message: 由消息头和消息体组成。消息体是不透明的，而消息头则由一系列的可选属性组成，这些属性包括routing-key、priority、delivery-mode（是否持久性存储）等。

Publisher: 消息的生产者。

Exchange: 接收消息并将消息路由到一个或多个Queue。default exchange 是默认的直连交换机，名字为空字符串，每个新建队列都会自动绑定到默认交换机上，绑定的路由键名称与队列名称相同。

Binding: 通过Binding将Exchange和Queue关联，这样Exchange就知道将消息路由到哪个Queue中。

Queue: 存储消息，队列的特性是先进先出。一个消息可分发到一个或多个队列。

Virtual host: 每个 vhost 本质上就是一个 mini 版的 RabbitMQ 服务器，拥有自己的队列、交换器、绑定和权限机制。vhost 是 AMQP 概念的基础，必须在连接时指定，RabbitMQ 默认的 vhost 是 /。当多个不同的用户使用同一个RabbitMQ server提供的服务时，可以划分出多个vhost，每个用户在自己的vhost创建exchange和queue。

Broker: 消息队列服务器实体。

3. 为什么要使用消息队列？

总结一下，主要三点原因：**解耦、异步、削峰。**

1、解耦。比如，用户下单后，订单系统需要通知库存系统，假如库存系统无法访问，则订单减库存将失败，从而导致订单操作失败。订单系统与库存系统耦合，这个时候如果使用消息队列，可以返回给用户成功，先把消息持久化，等库存系统恢复后，就可以正常消费减去库存了。

2、异步。将消息写入消息队列，非必要的业务逻辑以异步的方式运行，不影响主流程业务。

3、削峰。消费端慢慢的按照数据库能处理的并发量，从消息队列中慢慢拉取消息。在生产中，这个短暂的高峰期积压是允许的。比如秒杀活动，一般会因为流量过大，从而导致流量暴增，应用挂掉。这个时候加上消息队列，服务器接收到用户的请求后，首先写入消息队列，如果消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面。

4. 使用了消息队列会有什么缺点

- 系统可用性降低。引入消息队列之后，如果消息队列挂了，可能会影响到业务系统的可用性。
- 系统复杂性增加。加入了消息队列，要多考虑很多方面的问题，比如：一致性问题、如何保证消息不被重复消费、如何保证消息可靠性传输等。

5. Exchange 类型

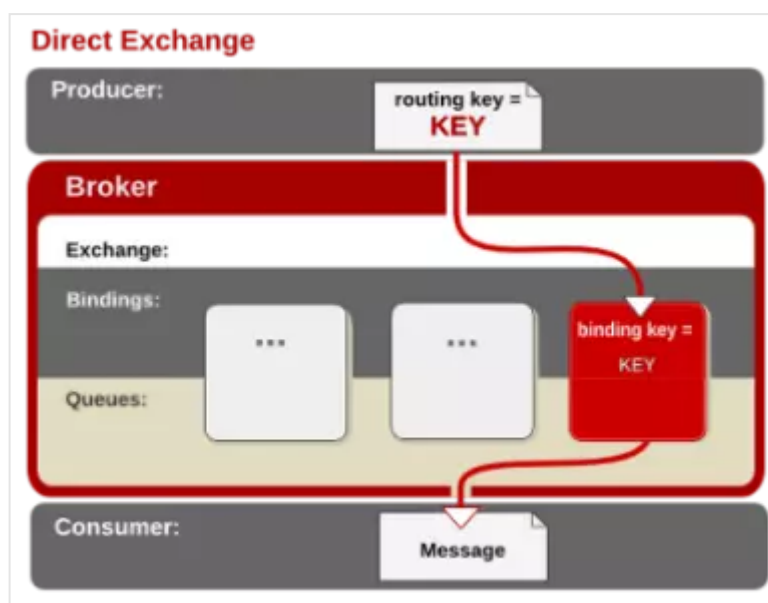
Exchange分发消息时根据类型的不同分发策略不同，目前共四种类型：direct、fanout、topic、headers。headers 模式根据消息的headers进行路由，此外 headers 交换器和 direct 交换器完全一致，但性能差很多。

Exchange规则。

类型名称	类型描述
fanout	把所有发送到该Exchange的消息路由到所有与它绑定的Queue中
direct	Routing Key==Binding Key
topic	模糊匹配
headers	Exchange不依赖于routing key与binding key的匹配规则来路由消息，而是根据发送的消息内容中的header属性进行匹配。

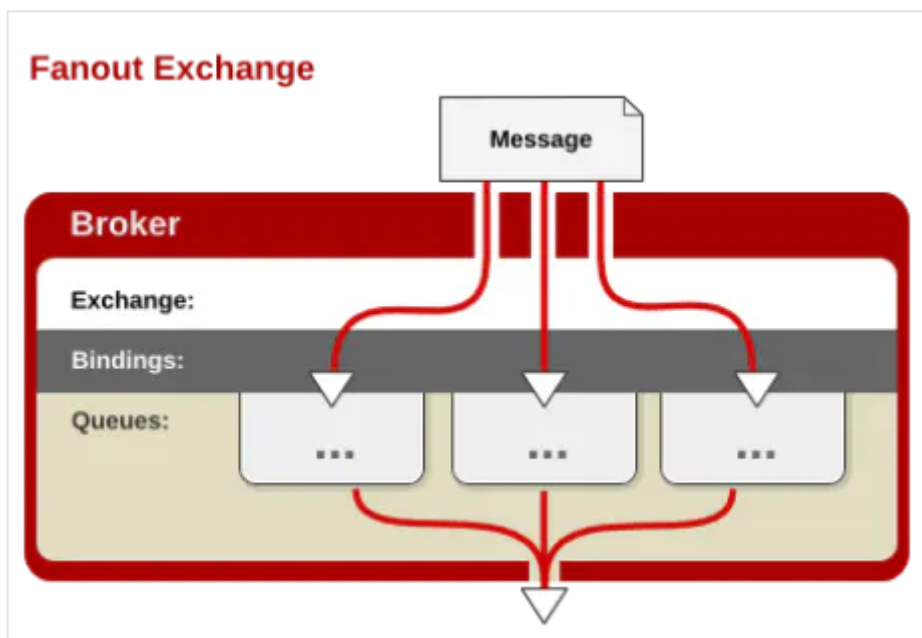
5.1. direct

direct交换机会将消息路由到binding key 和 routing key完全匹配的队列中。它是完全匹配、单播的模式。



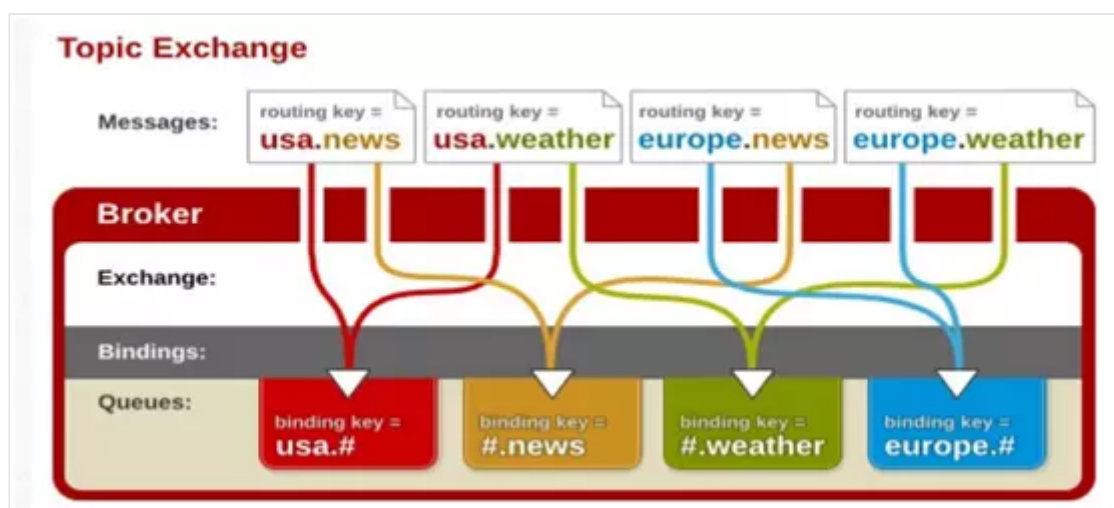
5.2. fanout

所有发到 fanout 类型交换机的消息都会路由到所有与该交换机绑定的队列上去。fanout 类型转发消息是最快的。



5.3. topic

topic交换机使用routing key和binding key进行模糊匹配，匹配成功则将消息发送到相应的队列。routing key和binding key都是句点号“.”分隔的字符串，binding key中可以存在两种特殊字符“*”与“#”，用于做模糊匹配，其中“*”用于匹配一个单词，“#”用于匹配多个单词。



5.4. headers

headers交换机是根据发送的消息内容中的headers属性进行路由的。在绑定Queue与Exchange时指定一组键值对；当消息发送到Exchange时，RabbitMQ会取到该消息的headers（也是一个键值对的形式），对比其中的键值对是否完全匹配Queue与Exchange绑定时指定的键值对；如果完全匹配则消息会路由到该Queue，否则不会路由到该Queue。

6. 消息丢失

消息丢失场景：生产者生产消息到RabbitMQ Server消息丢失、RabbitMQ Server存储的消息丢失和RabbitMQ Server到消费者消息丢失。

消息丢失从三个方面来解决：生产者确认机制、消费者手动确认消息和持久化。

6.1. 生产者确认机制

生产者发送消息到队列，无法确保发送的消息成功的到达server。

解决方法：

1. 事务机制。在一条消息发送之后会使发送端阻塞，等待RabbitMQ的回应，之后才能继续发送下一条消息。性能差。
2. 开启生产者确认机制，只要消息成功发送到交换机之后，RabbitMQ就会发送一个ack给生产者（即使消息没有Queue接收，也会发送ack）。如果消息没有成功发送到交换机，就会发送一条nack消息，提示发送失败。

在 Springboot 是通过 publisher-confirms 参数来设置 confirm 模式：

```
spring:
  rabbitmq:
    #开启 confirm 确认机制
    publisher-confirms: true
```

在生产端提供一个回调方法，当服务端确认了一条或者多条消息后，生产者会回调这个方法，根据具体的结果对消息进行后续处理，比如重新发送、记录日志等。

```
// 消息是否成功发送到Exchange
final RabbitTemplate.ConfirmCallback confirmCallback = (CorrelationData
correlationData, boolean ack, String cause) -> {
    log.info("correlationData: " + correlationData);
    log.info("ack: " + ack);
    if(!ack) {
        log.info("异常处理...");
    }
};

rabbitTemplate.setConfirmCallback(confirmCallback);
```

6.2. 路由不可达消息

生产者确认机制只确保消息正确到达交换机，对于从交换机路由到Queue失败的消息，会被丢弃掉，导致消息丢失。

对于不可路由的消息，有两种处理方式：Return消息机制和备份交换机。

Return消息机制

Return消息机制提供了回调函数 ReturnCallback，当消息从交换机路由到Queue失败才会回调这个方法。需要将 `mandatory` 设置为 `true`，才能监听到路由不可达的消息。

```
spring:
  rabbitmq:
    #触发ReturnCallback必须设置mandatory=true，否则Exchange没有找到Queue就会丢弃掉
    #消息，而不会触发ReturnCallback
    template.mandatory: true
```

通过 ReturnCallback 监听路由不可达消息。

```

final RabbitTemplate.ReturnCallback returnCallback = (Message message, int
replyCode, String replyText, String exchange, String routingKey) ->
    log.info("return exchange: " + exchange + ", routingKey: "
        + routingKey + ", replyCode: " + replyCode + ", replyText: "
+ replyText);
rabbitTemplate.setReturnCallback(returnCallback);

```

当消息从交换机路由到Queue失败时，会返回 `return exchange: , routingKey: MAIL, replyCode: 312, replyText: NO_ROUTE`。

备份交换机

备份交换机alternate-exchange 是一个普通的exchange，当你发送消息到对应的exchange时，没有匹配到queue，就会自动转移到备份交换机对应的queue，这样消息就不会丢失。

6.3. 消费者手动消息确认

有可能消费者收到消息还没来得及处理MQ服务就宕机了，导致消息丢失。因为消息者默认采用自动ack，一旦消费者收到消息后会通知MQ Server这条消息已经处理好了，MQ 就会移除这条消息。

解决方法：消费者设置为手动确认消息。消费者处理完逻辑之后再给broker回复ack，表示消息已经成功消费，可以从broker中删除。当消息者消费失败的时候，给broker回复nack，根据配置决定重新入队还是从broker移除，或者进入死信队列。只要没收到消费者的 acknowledgment，broker 就会一直保存着这条消息，但不会 requeue，也不会分配给其他 消费者。

消费者设置手动ack：

```

#设置消费端手动 ack
spring.rabbitmq.listener.simple.acknowledge-mode=manual

```

消息处理完，手动确认：

```

@RabbitListener(queues = RabbitMqConfig.MAIL_QUEUE)
public void onMessage(Message message, Channel channel) throws IOException {

    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    long deliveryTag = message.getMessageProperties().getDeliveryTag();
    //手工ack: 第二个参数是multiple, 设置为true, 表示deliveryTag序列号之前（包括自身）
    的消息都已经收到，设为false则表示收到一条消息
    channel.basicAck(deliveryTag, true);
    System.out.println("mail listener receive: " + new
String(message.getBody()));
}

```

当消息消费失败时，消费端给broker回复nack，如果consumer设置了requeue为false，则nack后broker会删除消息或者进入死信队列，否则消息会重新入队。

6.4. 持久化

如果RabbitMQ服务异常导致重启，将会导致消息丢失。RabbitMQ提供了持久化的机制，将内存中的消息持久化到硬盘上，即使重启RabbitMQ，消息也不会丢失。

消息持久化需要满足以下条件：

1. 消息设置持久化。发布消息前，设置投递模式delivery mode为2，表示消息需要持久化。
2. Queue设置持久化。
3. 交换机设置持久化。

当发布一条消息到交换机上时，Rabbit会先把消息写入持久化日志，然后才向生产者发送响应。一旦从队列中消费了一条消息的话并且做了确认，RabbitMQ会在持久化日志中移除这条消息。在消费消息前，如果RabbitMQ重启的话，服务器会自动重建交换机和队列，加载持久化日志中的消息到相应的队列或者交换机上，保证消息不会丢失。

6.5. 镜像队列

当MQ发生故障时，会导致服务不可用。引入RabbitMQ的镜像队列机制，将queue镜像到集群中其他的节点之上。如果集群中的一个节点失效了，能自动地切换到镜像中的另一个节点以保证服务的可用性。

通常每一个镜像队列都包含一个master和多个slave，分别对应于不同的节点。发送到镜像队列的所有消息总是被直接发送到master和所有的slave之上。除了publish外所有动作都只会向master发送，然后由master将命令执行的结果广播给slave，从镜像队列中的消费操作实际上是在master上执行的。

7. 重复消费

消息重复的原因有两个：1.生产时消息重复，2.消费时消息重复。

生产者发送消息给MQ，在MQ确认的时候出现了网络波动，生产者没有收到确认，这时候生产者就会重新发送这条消息，导致MQ会接收到重复消息。

消费者消费成功后，给MQ确认的时候出现了网络波动，MQ没有接收到确认，为了保证消息不丢失，MQ就会继续给消费者投递之前的消息。这时候消费者就接收到了两条一样的消息。由于重复消息是由于网络原因造成的，无法避免。

解决方法：发送消息时让每个消息携带一个全局的唯一ID，在消费消息时先判断消息是否已经被消费过，保证消息消费逻辑的幂等性。具体消费过程为：

1. 消费者获取到消息后先根据id去查询redis/db是否存在该消息
2. 如果不存在，则正常消费，消费完毕后写入redis/db
3. 如果存在，则证明消息被消费过，直接丢弃

8. 消费端限流

当RabbitMQ服务器积压大量消息时，队列里的消息会大量涌入消费端，可能导致消费端服务器崩溃。这种情况下需要对消费端限流。

Spring RabbitMQ 提供参数 prefetch 可以设置单个请求处理的消息个数。如果消费者同时处理的消息到达最大值的时候，则该消费者会阻塞，不会消费新的消息，直到有消息 ack 才会消费新的消息。

开启消费端限流：

```
#在单个请求中处理的消息个数，unack的最大数量
spring.rabbitmq.listener.simple.prefetch=2
```

原生 RabbitMQ 还提供 prefetchSize 和 global 两个参数。Spring RabbitMQ没有这两个参数。

```
//单条消息大小限制，0代表不限制
//global：限制限流功能是channel级别的还是consumer级别。当设置为false，consumer级别，限流功能生效，设置为true没有了限流功能，因为channel级别尚未实现。
void basicQos(int prefetchSize, int prefetchCount, boolean global) throws
IOException;
```

9. 死信队列

消费失败的消息存放的队列。

消息消费失败的原因：

- 消息被拒绝并且消息没有重新入队 (requeue=false)
- 消息超时未消费
- 达到最大队列长度

设置死信队列的 exchange 和 queue，然后进行绑定：

```
@Bean
public DirectExchange dlxExchange() {
    return new DirectExchange(RabbitMqConfig.DLX_EXCHANGE);
}

@Bean
public Queue dlxQueue() {
    return new Queue(RabbitMqConfig.DLX_QUEUE, true);
}

@Bean
public Binding bindingDeadExchange(Queue dlxQueue, DirectExchange
deadExchange) {
    return
BindingBuilder.bind(dlxQueue).to(deadExchange).with(RabbitMqConfig.DLX_QUEUE);
}
```

在普通队列加上两个参数，绑定普通队列到死信队列。当消息消费失败时，消息会被路由到死信队列。

```
@Bean
public Queue sendSmsQueue() {
    Map<String, Object> arguments = new HashMap<>(2);
    // 绑定该队列到私信交换机
    arguments.put("x-dead-letter-exchange", RabbitMqConfig.DLX_EXCHANGE);
    arguments.put("x-dead-letter-routing-key", RabbitMqConfig.DLX_QUEUE);
    return new Queue(RabbitMqConfig.MAIL_QUEUE, true, false, false,
arguments);
}
```

生产者完整代码：

```
@Component
```

```

@Slf4j
public class MQProducer {

    @Autowired
    RabbitTemplate rabbitTemplate;

    @Autowired
    RandomUtil randomUtil;

    @Autowired
    UserService userService;

    final RabbitTemplate.ConfirmCallback confirmCallback = (CorrelationData
correlationData, boolean ack, String cause) -> {
        log.info("correlationData: " + correlationData);
        log.info("ack: " + ack);
        if(!ack) {
            log.info("异常处理....");
        }
    };

    final RabbitTemplate.ReturnCallback returnCallback = (Message message, int
replyCode, String replyText, String exchange, String routingKey) ->
        log.info("return exchange: " + exchange + ", routingKey: "
            + routingKey + ", replyCode: " + replyCode + ", replyText: "
+ replyText);

    public void sendMail(String mail) {
        //貌似线程不安全 范围100000 - 999999
        Integer random = randomUtil.nextInt(100000, 999999);
        Map<String, String> map = new HashMap<>(2);
        String code = random.toString();
        map.put("mail", mail);
        map.put("code", code);

        MessageProperties mp = new MessageProperties();
        //在生产环境中这里不用Message，而是使用 fastJson 等工具将对象转换为 json 格式发送
        Message msg = new Message("tyson".getBytes(), mp);
        msg.getMessageProperties().setExpiration("3000");
        //如果消费端要设置为手工 ACK，那么生产端发送消息的时候一定发送 correlationData，
        并且全局唯一，用以唯一标识消息。
        CorrelationData correlationData = new CorrelationData("1234567890"+new
Date());

        rabbitTemplate.setMandatory(true);
        rabbitTemplate.setConfirmCallback(confirmCallback);
        rabbitTemplate.setReturnCallback(returnCallback);
        rabbitTemplate.convertAndSend(RabbitMqConfig.MAIL_QUEUE, msg,
correlationData);

        //存入redis
        userService.updateMailSendState(mail, code, MailConfig.MAIL_STATE_WAIT);
    }
}

```

消费者完整代码：

```

@Slf4j
@Component
public class DeadListener {

    @RabbitListener(queues = RabbitMqConfig.DLX_QUEUE)
    public void onMessage(Message message, Channel channel) throws IOException {

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        long deliveryTag = message.getMessageProperties().getDeliveryTag();
        //手工ack
        channel.basicAck(deliveryTag, false);
        System.out.println("receive--1: " + new String(message.getBody()));
    }
}

```

当普通队列中有死信时，RabbitMQ 就会自动的将这个消息重新发布到设置的死信交换机去，然后被路由到死信队列。可以监听死信队列中的消息做相应的处理。

10. RabbitMQ消息怎么路由？

消息路由必须有三部分：**交换器、路由、绑定**。生产者把消息发布到交换器上；绑定决定了消息如何从路由器路由到特定的队列；消息最终到达队列，并被消费者接收。

- 1、消息发布到交换器时，消息将拥有一个路由键（routing key），在消息创建时设定。
- 2、通过队列路由键，可以把队列绑定到交换器上。
- 3、消息到达交换器后，RabbitMQ会将消息的路由键与队列的路由键进行匹配（针对不同的交换器有不同的路由规则）。如果能够匹配到队列，则消息会投递到相应队列中。

11. RabbitMQ如何保证消息的顺序性？

单线程消费保证消息的顺序性；对消息进行编号，消费者根据编号处理消息。



扫码关注我



微信搜索



程序员大彬

公众号后台回复【面试】获取面试手册
PDF最新版

Kafka

1. Kafka 都有哪些特点？

- 高吞吐量、低延迟：kafka每秒可以处理几十万条消息，它的延迟最低只有几毫秒，每个topic可以分多个partition, consumer group 对partition进行consume操作。
- 可扩展性：kafka集群支持热扩展
- 持久性、可靠性：消息被持久化到本地磁盘，并且支持数据备份防止数据丢失
- 容错性：允许集群中节点失败（若副本数量为n,则允许n-1个节点失败）
- 高并发：支持数千个客户端同时读写

2. 请简述下你在哪些场景下会选择 Kafka？

- 日志收集：一个公司可以用Kafka可以收集各种服务的log，通过kafka以统一接口服务的方式开放给各种consumer，例如hadoop、HBase、Solr等。
- 消息系统：解耦和生产者和消费者、缓存消息等。
- 用户活动跟踪：Kafka经常被用来记录web用户或者app用户的各种活动，如浏览网页、搜索、点击等活动，这些活动信息被各个服务器发布到kafka的topic中，然后订阅者通过订阅这些topic来做实时的监控分析，或者装载到hadoop、数据仓库中做离线分析和挖掘。
- 运营指标：Kafka也经常用来记录运营监控数据。包括收集各种分布式应用的数据，生产各种操作的集中反馈，比如报警和报告。
- 流式处理：比如spark streaming和 Flink

3. Kafka 的设计架构你知道吗？

Kafka 架构分为以下几个部分：

- Producer：消息生产者，就是向 kafka broker 发消息的客户端。
- Consumer：消息消费者，向 kafka broker 取消息的客户端。
- Topic：可以理解为一个队列，一个 Topic 又分为一个或多个分区，
- Consumer Group：这是 kafka 用来实现一个 topic 消息的广播（发给所有的 consumer）和单播（发给任意一个 consumer）的手段。一个 topic 可以有多个 Consumer Group。
- Broker：一台 kafka 服务器就是一个 broker。一个集群由多个 broker 组成。一个 broker 可以容纳多个 topic。
- Partition：为了实现扩展性，一个非常大的 topic 可以分布到多个 broker 上，每个 partition 是一个有序的队列。partition 中的每条消息都会被分配一个有序 id (offset)。将消息发给 consumer，kafka 只保证按一个 partition 中的消息的顺序，不保证一个 topic 的整体（多个 partition 间）的顺序。
- Offset：kafka 的存储文件都是按照 offset.kafka 来命名，用 offset 做名字的好处是方便查找。例如你想找位于 2049 的位置，只要找到 2048.kafka 的文件即可。当然 the first offset 就是 00000000000.kafka。

4. Kafka 分区的目的？

分区对于 Kafka 集群的好处是：实现负载均衡。分区对于消费者来说，可以提高并发度，提高效率。

5. 你知道 Kafka 是如何做到消息的有序性？

kafka 中的每个 partition 中的消息在写入时都是有序的，而且单独一个 partition 只能由一个消费者去消费，可以在里面保证消息的顺序性。但是分区之间的消息是不保证有序的。

6. Kafka Producer 的执行过程？

1, Producer生产消息 --> 2, 从Zookeeper找到Partition的Leader --> 3, 推送消息 --> 4, 通过ISR列表通知给Follower --> 5, Follower从Leader拉取消息, 并发送ack --> 6, Leader收到所有副本的ack, 更新Offset, 并向Producer发送ack, 表示消息写入成功。

7. 讲一下你使用 Kafka Consumer 消费消息时的线程模型，为何如此设计？

Thread-Per-Consumer Model, 这种多线程模型是利用Kafka的topic分多个partition的机制来实现并行：每个线程都有自己的consumer实例，负责消费若干个partition。各个线程之间是完全独立的，不涉及任何线程同步和通信，所以实现起来非常简单。

8. 请谈一谈 Kafka 数据一致性原理

一致性就是说不论是老的 Leader 还是新选举的 Leader, Consumer 都能读到一样的数据。

假设分区的副本为3, 其中副本0是 Leader, 副本1和副本2是 follower, 并且在 ISR 列表里面。虽然副本0已经写入了 Message4, 但是 Consumer 只能读取到 Message2。因为所有的 ISR 都同步了 Message2, 只有 High Water Mark 以上的消息才支持 Consumer 读取, 而 High Water Mark 取决于 ISR 列表里面偏移量最小的分区, 对应于上图的副本2, 这个很类似于木桶原理。

这样做的原因是还没有被足够多副本复制的消息被认为是“不安全”的, 如果 Leader 发生崩溃, 另一个副本成为新 Leader, 那么这些消息很可能丢失了。如果我们允许消费者读取这些消息, 可能会破坏一致性。试想, 一个消费者从当前 Leader (副本0) 读取并处理了 Message4, 这个时候 Leader 挂掉了, 选举了副本1为新的 Leader, 这时候另一个消费者再去从新的 Leader 读取消息, 发现这个消息其实并不存在, 这就导致了数据不一致性问题。

当然, 引入了 High Water Mark 机制, 会导致 Broker 间的消息复制因为某些原因变慢, 那么消息到达消费者的时间也会随之变长 (因为我们会先等待消息复制完毕)。延迟时间可以通过参数 `replica.lag.time.max.ms` 参数配置, 它指定了副本在复制消息时可被允许的最大延迟时间。

9. ISR、OSR、AR 是什么？

ISR: In-Sync Replicas 副本同步队列

OSR: Out-of-Sync Replicas

AR: Assigned Replicas 所有副本

ISR是由leader维护, follower从leader同步数据有一些延迟 (具体可以参见 图文了解 Kafka 的副本复制机制), 超过相应的阈值会把 follower 剔除出 ISR, 存入OSR (Out-of-Sync Replicas) 列表, 新加入的 follower也会先存放在OSR中。AR=ISR+OSR。

10. LEO、HW、LSO、LW等分别代表什么

LEO: 是 LogEndOffset 的简称, 代表当前日志文件中下一条

HW: 水位或水印 (watermark) 一词, 也可称为高水位(high watermark), 通常被用在流式处理领域 (比如Apache Flink、Apache Spark等), 以表征元素或事件在基于时间层面上的进度。在Kafka中, 水位的概念反而与时间无关, 而是与位置信息相关。严格来说, 它表示的就是位置信息, 即位移 (offset)。取 partition 对应的 ISR中 最小的 LEO 作为 HW, consumer 最多只能消费到 HW 所在的位置上一条信息。

LSO: 是 LastStableOffset 的简称, 对未完成的事务而言, LSO 的值等于事务中第一条消息的位置 (firstUnstableOffset), 对已完成的事务而言, 它的值同 HW 相同

LW: Low Watermark 低水位, 代表 AR 集合中最小的 logStartOffset 值。

11. 数据传输的事务有几种？

数据传输的事务定义通常有以下三种级别：

(1) 最多一次: 消息不会被重复发送，最多被传输一次，但也有可能一次不传输 (2) 最少一次: 消息不会被漏发送，最少被传输一次，但也有可能被重复传输. (3) 精确的一次 (Exactly once) : 不会漏传输也不会重复传输,每个消息都传输被

12. Kafka 消费者是否可以消费指定分区消息？

Kafka consumer消费消息时，向broker发出fetch请求去消费特定分区的消息，consumer指定消息在日志中的偏移量 (offset)，就可以消费从这个位置开始的消息，consumer拥有了offset的控制权，可以向后回滚去重新消费之前的消息，这是很有意义的。

13. Kafka消息是采用Pull模式，还是Push模式？

Kafka最初考虑的问题是，consumer应该从brokers拉取消息还是brokers将消息推送到consumer，也就是pull还是push。在这方面，Kafka遵循了一种大部分消息系统共同的传统的设计：producer将消息推送到broker，consumer从broker拉取消息。

一些消息系统比如Scribe和Apache Flume采用了push模式，将消息推送到下游的consumer。这样做有好处也有坏处：由broker决定消息推送的速率，对于不同消费速率的consumer就不太好处理了。消息系统都致力于让consumer以最大的速率最快速的消费消息，但不幸的是，push模式下，当broker推送的速率远大于consumer消费的速率时，consumer恐怕就要崩溃了。最终Kafka还是选取了传统的pull模式。

Pull模式的另外一个好处是consumer可以自主决定是否批量的从broker拉取数据。Push模式必须在不知道下游consumer消费能力和消费策略的情况下决定是立即推送每条消息还是缓存之后批量推送。如果为了避免consumer崩溃而采用较低的推送速率，将可能导致一次只推送较少的消息而造成浪费。Pull模式下，consumer就可以根据自己的消费能力去决定这些策略。

Pull有个缺点是，如果broker没有可供消费的消息，将导致consumer不断在循环中轮询，直到新消息到达。为了避免这点，Kafka有个参数可以让consumer阻塞知道新消息到达(当然也可以阻塞知道消息的数量达到某个特定的量这样就可以批量发

14. Kafka 高效文件存储设计特点

Kafka把topic中一个partition大文件分成多个小文件段，通过多个小文件段，就容易定期清除或删除已经消费完文件，减少磁盘占用。

通过索引信息可以快速定位message和确定response的最大大小。

通过index元数据全部映射到memory，可以避免segment file的IO磁盘操作。

通过索引文件稀疏存储，可以大幅降低index文件元数据占用空间大小

15. Kafka创建Topic时如何将分区放置到不同的Broker中

副本因子不能大于 Broker 的个数；

第一个分区（编号为0）的第一个副本放置位置是随机从 brokerList 选择的；

其他分区的第一个副本放置位置相对于第0个分区依次往后移。也就是如果有5个 Broker，5个分区，假设第一个分区放在第四个 Broker 上，那么第二个分区将会放在第五个 Broker 上；第三个分区将会放在第一个 Broker 上；第四个分区将会放在第二个 Broker 上，依次类推；

剩余的副本相对于第一个副本放置位置其实是由 nextReplicaShift 决定的，而这个数也是随机产生的

16. 谈一谈 Kafka 的再均衡

在Kafka中，当有新消费者加入或者订阅的topic数发生变化时，会触发Rebalance(再均衡：在同一个消费者组当中，分区的所有权从一个消费者转移到另外一个消费者)机制，Rebalance顾名思义就是重新均衡消费者消费。Rebalance的过程如下：

第一步：所有成员都向coordinator发送请求，请求入组。一旦所有成员都发送了请求，coordinator会从中选择一个consumer担任leader的角色，并把组成员信息以及订阅信息发给leader。

第二步：leader开始分配消费方案，指明具体哪个consumer负责消费哪些topic的哪些partition。一旦完成分配，leader会将这个方案发给coordinator。coordinator接收到分配方案之后会把方案发给各个consumer，这样组内的所有成员就都知道自己应该消费哪些分区了。

所以对于Rebalance来说，Coordinator起着至关重要的作用

17. Kafka 是如何实现高吞吐率的？

Kafka是分布式消息系统，需要处理海量的消息，Kafka的设计是把所有的消息都写入速度低容量大的硬盘，以此来换取更强的存储能力，但实际上，使用硬盘并没有带来过多的性能损失。kafka主要使用了以下几个方式实现了超高的吞吐率：

- 顺序读写；
- 零拷贝
- 文件分段
- 批量发送
- 数据压缩。

18. Kafka 缺点？

- 由于是批量发送，数据并非真正的实时；
- 对于mqtt协议不支持；
- 不支持物联网传感数据直接接入；
- 仅支持统一分区内消息有序，无法实现全局消息有序；
- 监控不完善，需要安装插件；
- 依赖zookeeper进行元数据管理；

19. Kafka 新旧消费者的区别

旧的 Kafka 消费者 API 主要包括：SimpleConsumer（简单消费者）和 ZookeeperConsumerConnector（高级消费者）。SimpleConsumer 名字看起来是简单消费者，但是其实用起来很不简单，可以使用它从特定的分区和偏移量开始读取消息。高级消费者和现在新的消费者有点像，有消费者群组，有分区再均衡，不过它使用 ZK 来管理消费者群组，并不具备偏移量和再均衡的可操控性。

现在的消费者同时支持以上两种行为，所以为啥还用旧消费者 API 呢？

20. Kafka 分区数可以增加或减少吗？为什么？

我们可以使用 bin/kafka-topics.sh 命令对 Kafka 增加 Kafka 的分区数据，但是 Kafka 不支持减少分区数。

Kafka 分区数据不支持减少是由很多原因的，比如减少的分区其数据放到哪里去？是删除，还是保留？删除的话，那么这些没消费的消息不就丢了。如果保留这些消息如何放到其他分区里面？追加到其他分区后面的话那么就破坏了 Kafka 单个分区的有序性。如果要保证删除分区数据插入到其他分区保证有序性，那么实现起来逻辑就会非常复杂。

参考: https://blog.51cto.com/u_15127589/2679155

Spring

1. Spring的优点

- 通过控制反转和依赖注入实现**松耦合**。
- 支持**面向切面**的编程，并且把应用业务逻辑和系统服务分开。
- 通过切面和模板减少样板式代码。
- 声明式事务的支持。可以从单调繁冗的事务管理代码中解脱出来，通过声明式方式灵活地进行事务的管理，提高开发效率和质量。
- 方便集成各种优秀框架。内部提供了对各种优秀框架的直接支持（如：Hessian、Quartz、MyBatis等）。
- 方便程序的测试。Spring支持Junit4，添加注解便可以测试Spring程序。

2. Spring 用到了哪些设计模式？

1、**简单工厂模式**：**BeanFactory** 就是简单工厂模式的体现，根据传入一个唯一标识来获得 Bean 对象。

```
@Override
public Object getBean(String name) throws BeansException {
    assertBeanFactoryActive();
    return getBeanFactory().getBean(name);
}
```

2、**工厂方法模式**：**FactoryBean** 就是典型的工厂方法模式。spring在使用 **getBean()** 调用获得该bean时，会自动调用该bean的 **getObject()** 方法。每个 Bean 都会对应一个 **FactoryBean**，如 **SessionFactory** 对应 **SessionFactoryBean**。

3、**单例模式**：一个类仅有一个实例，提供一个访问它的全局访问点。Spring 创建 Bean 实例默认是单例的。

4、**适配器模式**：SpringMVC中的适配器 **HandlerAdatper**。由于应用会有多个Controller实现，如果需要直接调用Controller方法，那么需要先判断是由哪一个Controller处理请求，然后调用相应的方法。当增加新的 Controller，需要修改原来的逻辑，违反了开闭原则（对修改关闭，对扩展开放）。

为此，Spring提供了一个适配器接口，每一种 Controller 对应一种 **HandlerAdapter** 实现类，当请求过来，SpringMVC会调用 **getHandler()** 获取相应的Controller，然后获取该Controller对应的 **HandlerAdapter**，最后调用 **HandlerAdapter** 的 **handle()** 方法处理请求，实际上调用的是Controller的 **handleRequest()**。每次添加新的 Controller 时，只需要增加一个适配器类就可以，无需修改原有的逻辑。

常用的处理器适配器：**SimpleControllerHandlerAdapter**，**HttpRequestHandlerAdapter**，**AnnotationMethodHandlerAdapter**。

```
// Determine handler for the current request.
mappedHandler = getHandler(processedRequest);

HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

// Actually invoke the handler.
mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

public class HttpRequestHandlerAdapter implements HandlerAdapter {
```

```

@Override
public boolean supports(Object handler) {//handler是被适配的对象，这里使用的是对象的适配器模式
    return (handler instanceof HttpRequestHandler);
}

@Override
@Nullable
public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler)
    throws Exception {

    ((HttpRequestHandler) handler).handleRequest(request, response);
    return null;
}
}

```

5、**代理模式**：spring 的 aop 使用了动态代理，有两种方式 `JdkDynamicAopProxy` 和 `Cglib2AopProxy`。

6、**观察者模式**：spring 中 observer 模式常用的地方是 listener 的实现，如 `ApplicationListener`。

7、**模板模式**：Spring 中 `JdbcTemplate`、`hibernateTemplate` 等，就使用到了模板模式。

3. 什么是AOP?

面向切面编程，作为面向对象的一种补充，将公共逻辑（事务管理、日志、缓存等）封装成切面，跟业务代码进行分离，可以减少系统的重复代码和降低模块之间的耦合度。切面就是那些与业务无关，但所有业务模块都会调用的公共逻辑。

4. AOP有哪些实现方式?

AOP有两种实现方式：静态代理和动态代理。

静态代理

静态代理：代理类在编译阶段生成，在编译阶段将通知织入Java字节码中，也称编译时增强。AspectJ使用的是静态代理。

缺点：代理对象需要与目标对象实现一样的接口，并且实现接口的方法，会有冗余代码。同时，一旦接口增加方法，目标对象与代理对象都要维护。

动态代理

动态代理：代理类在程序运行时创建，AOP框架不会去修改字节码，而是在内存中临时生成一个代理对象，在运行期间对业务方法进行增强，不会生成新类。

5. Spring AOP的实现原理

Spring 的 AOP 实现原理其实很简单，就是通过**动态代理**实现的。如果我们为 Spring 的某个 bean 配置了切面，那么 Spring 在创建这个 bean 的时候，实际上创建的是这个 bean 的一个代理对象，我们后续对 bean 中方法的调用，实际上调用的是代理类重写的代理方法。而 Spring 的 AOP 使用了两种动态代理，分别是JDK的动态代理，以及CGLib的动态代理。

6. JDK动态代理和CGLIB动态代理的区别?

Spring AOP中的动态代理主要有两种方式：JDK动态代理和CGLIB动态代理。

JDK动态代理

如果目标类实现了接口，Spring AOP会选择使用JDK动态代理目标类。代理类根据目标类实现的接口动态生成，不需要自己编写，生成的动态代理类和目标类都实现相同的接口。JDK动态代理的核心是 `InvocationHandler` 接口和 `Proxy` 类。

缺点：**目标类必须有实现的接口**。如果某个类没有实现接口，那么这个类就不能用JDK动态代理。

CGLIB动态代理

通过继承实现。如果目标类没有实现接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library）可以在运行时动态生成类的字节码，动态创建目标类的子类对象，在子类对象中增强目标类。

CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为 `final`，那么它是无法使用CGLIB做动态代理的。

优点：目标类不需要实现特定的接口，更加灵活。

什么时候采用哪种动态代理？

1. 如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP
2. 如果目标对象实现了接口，可以强制使用CGLIB实现AOP
3. 如果目标对象没有实现了接口，必须采用CGLIB库

两者的区别：

1. jdk动态代理使用jdk中的类Proxy来创建代理对象，它使用反射技术来实现，不需要导入其他依赖。cglib需要引入相关依赖：`asm.jar`，它使用字节码增强技术来实现。
2. 当目标类实现了接口的时候Spring Aop默认使用jdk动态代理方式来增强方法，没有实现接口的时候使用cglib动态代理方式增强方法。

7. Spring AOP相关术语

(1) **切面** (Aspect)：切面是通知和切点的结合。通知和切点共同定义了切面的全部内容。

(2) **连接点** (Join point)：指方法，在Spring AOP中，一个连接点总是代表一个方法的执行。连接点是在应用执行过程中能够插入切面的一个点。这个点可以是调用方法时、抛出异常时、甚至修改一个字段时。切面代码可以利用这些点插入到应用的正常流程之中，并添加新的行为。

(3) **通知** (Advice)：在AOP术语中，切面的工作被称为通知。

(4) **切入点** (Pointcut)：切点的定义会匹配通知所要织入的一个或多个连接点。我们通常使用明确的类和方法名称，或是利用正则表达式定义所匹配的类和方法名称来指定这些切点。

(5) **引入** (Introduction)：引入允许我们向现有类添加新方法或属性。

(6) **目标对象** (Target Object)：被一个或者多个切面 (aspect) 所通知 (advise) 的对象。它通常是一个代理对象。

(7) **织入** (Weaving)：织入是把切面应用到目标对象并创建新的代理对象的过程。在目标对象的生命周期里有以下时间点可以进行织入：

- 编译期：切面在目标类编译时被织入。AspectJ的织入编译器是以这种方式织入切面的。
- 类加载期：切面在目标类加载到JVM时被织入。需要特殊的类加载器，它可以在目标类被引入应用之前增强该目标类的字节码。AspectJ5的加载时织入就支持以这种方式织入切面。
- 运行期：切面在应用运行的某个时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象动态地创建一个代理对象。SpringAOP就是以这种方式织入切面。

8. Spring通知有哪些类型？

在AOP术语中，切面的工作被称为通知。通知实际上是程序运行时要通过Spring AOP框架来触发的代码段。

Spring切面可以应用5种类型的通知：

1. **前置通知** (Before)：在目标方法被调用之前调用通知功能；
2. **后置通知** (After)：在目标方法完成之后调用通知，此时不会关心方法的输出是什么；
3. **返回通知** (After-returning)：在目标方法成功执行之后调用通知；
4. **异常通知** (After-throwing)：在目标方法抛出异常后调用通知；
5. **环绕通知** (Around)：通知包裹了被通知的方法，在被通知的方法调用之前和调用之后执行自定义的逻辑。

9. 什么是IOC?

IOC：**控制反转**，由Spring容器管理bean的整个生命周期。通过反射实现对其他对象的控制，包括初始化、创建、销毁等，解放手动创建对象的过程，同时降低类之间的耦合度。

10. IOC的好处?

ioc的思想最核心的地方在于，资源不由使用资源者管理，而由不使用资源的第三方管理，这可以带来很多好处。第一，资源集中管理，实现资源的可配置和易管理。第二，降低了使用资源双方的依赖程度，也就是我们说的耦合度。

也就是说，甲方要达成某种目的不需要直接依赖乙方，它只需要达到的目的告诉第三方机构就可以了，比如甲方需要一双袜子，而乙方它卖一双袜子，它要把袜子卖出去，并不需要自己去直接找到一个卖家来完成袜子的卖出。它也只需要找第三方，告诉别人我要卖一双袜子。这下好了，甲乙双方进行交易活动，都不需要自己直接去找卖家，相当于程序内部开放接口，卖家由第三方作为参数传入。甲乙互相不依赖，而且只有在进行交易活动的时候，甲才和乙产生联系。反之亦然。这样做什么好处呢，甲乙可以在对方不真实存在的情况下独立存在，而且保证不交易时候无联系，想交易的时候可以很容易的产生联系。甲乙交易活动不需要双方见面，避免了双方的互不信任造成交易失败的问题。因为交易由第三方来负责联系，而且甲乙都认为第三方可靠。那么交易就能很可靠很灵活的产生和进行了。

这就是ioc的核心思想。生活中这种例子比比皆是，支付宝在整个淘宝体系里就是庞大的ioc容器，交易双方之外的第三方，提供可靠性可依赖可灵活变更交易方的资源管理中心。另外人事代理也是，雇佣机构和个人之外的第三方。

参考链接：<https://www.zhihu.com/question/23277575/answer/24259844>

11. 什么是依赖注入?

在Spring创建对象的过程中，把对象依赖的属性注入到对象中。依赖注入主要有两种方式：构造器注入和属性注入。

12. IOC容器初始化过程?

1. 从XML中读取配置文件。
2. 将bean标签解析成 BeanDefinition，如解析 property 元素，并注入到 BeanDefinition 实例中。
3. 将 BeanDefinition 注册到容器 BeanDefinitionMap 中。
4. BeanFactory 根据 BeanDefinition 的定义信息创建实例化和初始化 bean。

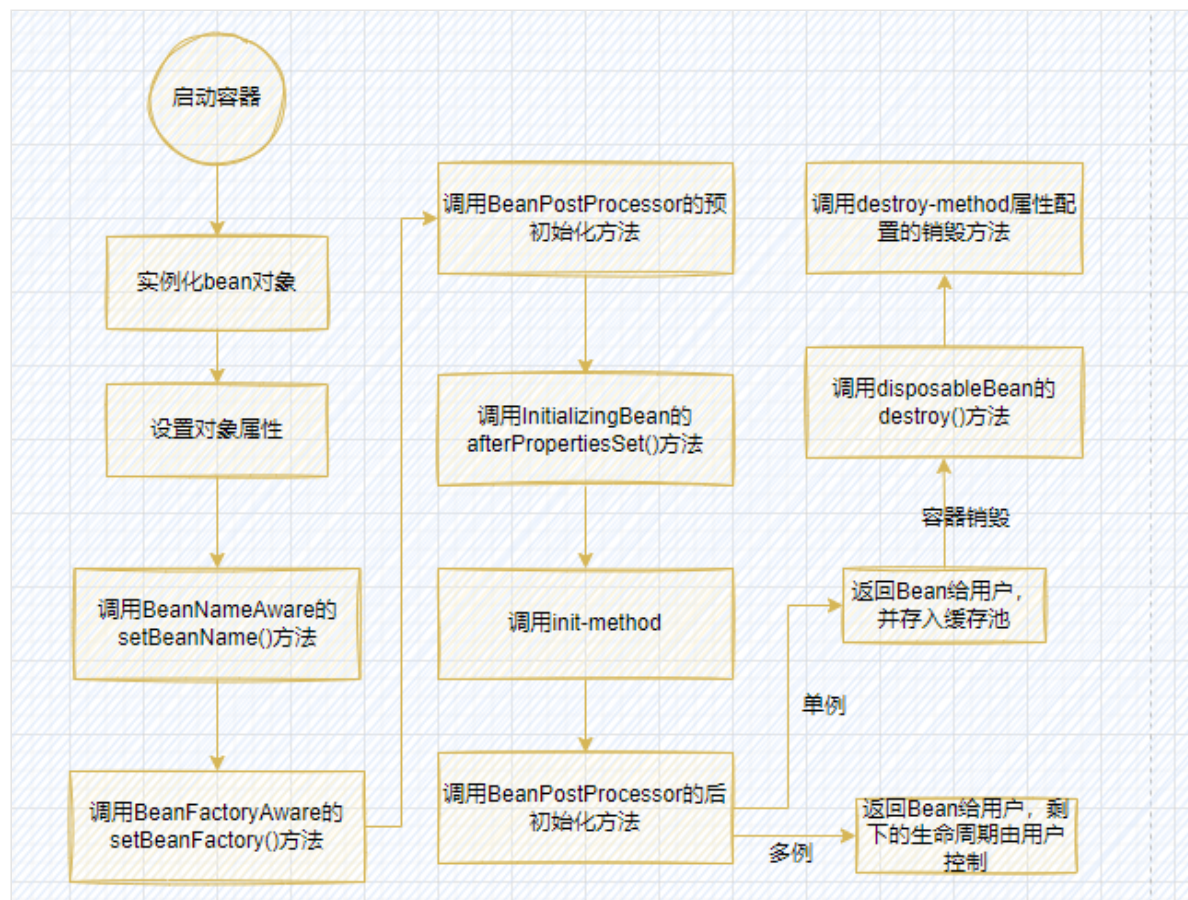
单例bean的初始化以及依赖注入一般都在容器初始化阶段进行，只有懒加载 (lazy-init为true) 的单例bean是在应用第一次调用getBean()时进行初始化和依赖注入。

```
// AbstractApplicationContext
// Instantiate all remaining (non-lazy-init) singletons.
finishBeanFactoryInitialization(beanFactory);
```

多例bean 在容器启动时不实例化，即使设置 lazy-init 为 false 也没用，只有调用了getBean()才进行实例化。

`LoadBeanDefinitions` 采用了模板模式，具体加载 `BeanDefinition` 的逻辑由各个子类完成。

13. Bean的生命周期



1.调用bean的构造方法创建Bean

2.通过反射调用setter方法进行属性的依赖注入

3.如果Bean实现了 `BeanNameAware` 接口，Spring将调用 `setBeanName()`，设置 `Bean` 的name (xml文件中bean标签的id)

4.如果Bean实现了 `BeanFactoryAware` 接口，Spring将调用 `setBeanFactory()` 把bean factory设置给Bean

5.如果存在 `BeanPostProcessor`，Spring将调用它们的 `postProcessBeforeInitialization` (预初始化) 方法，在Bean初始化前对其进行处理

6.如果Bean实现了 `InitializingBean` 接口，Spring将调用它的 `afterPropertiesSet` 方法，然后调用xml定义的 `init-method` 方法，两个方法作用类似，都是在初始化 bean 的时候执行

7.如果存在 `BeanPostProcessor`，Spring将调用它们的 `postProcessAfterInitialization` (后初始化) 方法，在Bean初始化后对其进行处理

8.Bean初始化完成，供应用使用，这里分两种情况：

8.1 如果Bean为单例的话，那么容器会返回Bean给用户，并存入缓存池。如果Bean实现了 `DisposableBean` 接口，Spring将调用它的 `destroy` 方法，然后调用在xml中定义的 `destroy-method` 方法，这两个方法作用类似，都是在Bean实例销毁前执行。

8.2 如果Bean是多例的话，容器将Bean返回给用户，剩下的生命周期由用户控制。

```

public interface BeanPostProcessor {
    @Nullable
    default Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {
        return bean;
    }
    @Nullable
    default Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {
        return bean;
    }
}

public interface InitializingBean {
    void afterPropertiesSet() throws Exception;
}

```

14. BeanFactory和FactoryBean的区别?

BeanFactory: 管理Bean的容器，Spring中生成的Bean都是由这个接口的实现来管理的。

FactoryBean: 通常是用来创建比较复杂的bean，一般的bean 直接用xml配置即可，但如果一个bean的创建过程中涉及到很多其他的bean 和复杂的逻辑，直接用xml配置比较麻烦，这时可以考虑用FactoryBean，可以隐藏实例化复杂Bean的细节。

当配置文件中bean标签的class属性配置的实现类是FactoryBean时，通过 `getBean()`方法返回的不是FactoryBean本身，而是调用FactoryBean#`getObject()`方法所返回的对象，相当于FactoryBean#`getObject()`代理了`getBean()`方法。如果想得到FactoryBean必须使用 `'&' + beanName`的方式获取。

Mybatis 提供了 `SqlSessionFactoryBean`，可以简化 `SqlSessionFactory` 的配置：

```

public class SqlSessionFactoryBean implements FactoryBean<SqlSessionFactory>,
    InitializingBean, ApplicationListener<ApplicationEvent> {
    @Override
    public void afterPropertiesSet() throws Exception {
        notNull(dataSource, "Property 'dataSource' is required");
        notNull(sqlSessionFactoryBuilder, "Property 'sqlSessionFactoryBuilder' is
            required");
        state((configuration == null && configLocation == null) || !(configuration
            != null && configLocation != null),
            "Property 'configuration' and 'configLocation' can not specified
            with together");
        this.sqlSessionFactory = buildSqlSessionFactory();
    }

    protected SqlSessionFactory buildSqlSessionFactory() throws IOException {
        //复杂逻辑
    }

    @Override
    public SqlSessionFactory getObject() throws Exception {
        if (this.sqlSessionFactory == null) {
            afterPropertiesSet();
        }
        return this.sqlSessionFactory;
    }
}

```

```
}
```

在 xml 配置 SqlSessionFactoryBean:

```
<bean id="tradeSqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="trade" />
    <property name="mapperLocations" value="classpath*:mapper/trade/*Mapper.xml"
/>
    <property name="configLocation" value="classpath:mybatis-config.xml" />
    <property name="typeAliasesPackage"
value="com.bytebeats.mybatis3.domain.trade" />
</bean>
```

Spring 将会在应用启动时创建 `SqlSessionFactory`，并使用 `sqlSessionFactory` 这个名字存储起来。

15. BeanFactory和ApplicationContext有什么区别？

BeanFactory和ApplicationContext是Spring的两大核心接口，都可以当做Spring的容器。其中ApplicationContext是BeanFactory的子接口。

两者区别如下：

1、功能上的区别。BeanFactory是Spring里面最底层的接口，包含了各种Bean的定义，读取bean配置文档，管理bean的加载、实例化，控制bean的生命周期，维护bean之间的依赖关系。

ApplicationContext接口作为BeanFactory的派生，除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能，如继承MessageSource、支持国际化、统一的资源文件访问方式、同时加载多个配置文件等功能。

2、加载方式的区别。BeanFactory采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFactory加载后，直至第一次使用调用getBean方法才会抛出异常。

而ApplicationContext是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。ApplicationContext启动后预载入所有的单例Bean，那么在需要的时候，不需要等待创建bean，因为它们已经创建好了。

相对于基本的BeanFactory，ApplicationContext唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。

3、创建方式的区别。BeanFactory通常以编程的方式被创建，ApplicationContext还能以声明的方式创建，如使用ContextLoader。

4、注册方式的区别。BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册。

16. Bean注入容器有哪些方式？

1、@Configuration + @Bean

@Configuration用来声明一个配置类，然后使用 @Bean 注解，用于声明一个bean，将其加入到Spring容器中。

```

@Configuration
public class MyConfiguration {
    @Bean
    public Person person() {
        Person person = new Person();
        person.setName("大彬");
        return person;
    }
}

```

2、通过包扫描特定注解的方式

@ComponentScan放置在我们的配置类上，然后可以指定一个路径，进行扫描带有特定注解的bean，然后加至容器中。

特定注解包括@Controller、@Service、@Repository、@Component

```

@Component
public class Person {
    //...
}

@ComponentScan(basePackages = "com.dabin.test.*")
public class Demo1 {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext applicationContext = new
        AnnotationConfigApplicationContext(Demo1.class);
        Person bean = applicationContext.getBean(Person.class);
        System.out.println(bean);
    }
}

```

3、@Import注解导入

@Import注解平时开发用的不多，但是也是非常重要的，在进行Spring扩展时经常会用到，它经常搭配自定义注解进行使用，然后往容器中导入一个配置文件。

```

@ComponentScan
/*把用到的资源导入到当前容器中*/
@Import({Person.class})
public class App {
    public static void main(String[] args) throws Exception {
        ConfigurableApplicationContext context =
        SpringApplication.run(App.class, args);
        System.out.println(context.getBean(Person.class));
        context.close();
    }
}

```

4、实现BeanDefinitionRegistryPostProcessor进行后置处理。

在Spring容器启动的时候会执行 BeanDefinitionRegistryPostProcessor 的 postProcessBeanDefinitionRegistry 方法，就是等beanDefinition加载完毕之后，对beanDefinition进行后置处理，可以在此进行调整IOC容器中的beanDefinition，从而干扰到后面进行初始化bean。

在下面的代码中，我们手动向beanDefinitionRegistry中注册了person的BeanDefinition。最终成功将person加入到applicationContext中。

```
public class Demo1 {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext applicationContext = new
AnnotationConfigApplicationContext();
        MyBeanDefinitionRegistryPostProcessor
beanDefinitionRegistryPostProcessor = new
MyBeanDefinitionRegistryPostProcessor();

        applicationContext.addBeanFactoryPostProcessor(beanDefinitionRegistryPostProces
sor);

        applicationContext.refresh();
        Person bean = applicationContext.getBean(Person.class);
        System.out.println(bean);
    }
}

class MyBeanDefinitionRegistryPostProcessor implements
BeanDefinitionRegistryPostProcessor {
    @Override
    public void postProcessBeanDefinitionRegistry(BeaDefinitionRegistry
registry) throws BeansException {
        AbstractBeanDefinition beanDefinition =
BeanDefinitionBuilder.rootBeanDefinition(Person.class).getBeanDefinition();
        registry.registerBeanDefinition("person", beanDefinition);
    }

    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory
beanFactory) throws BeansException {
    }
}
```

5、使用FactoryBean接口

如下图代码，使用@Configuration + @Bean的方式将 PersonFactoryBean 加入到容器中，这里没有向容器中直接注入 Person，而是注入 PersonFactoryBean，然后从容器中拿Person这个类型的bean。

```
@Configuration
public class Demo1 {
    @Bean
    public PersonFactoryBean personFactoryBean() {
        return new PersonFactoryBean();
    }

    public static void main(String[] args) {
        AnnotationConfigApplicationContext applicationContext = new
AnnotationConfigApplicationContext(Demo1.class);
        Person bean = applicationContext.getBean(Person.class);
        System.out.println(bean);
    }
}

class PersonFactoryBean implements FactoryBean<Person> {
    @Override
```

```

    public Person getObject() throws Exception {
        return new Person();
    }

    @Override
    public Class<?> getObjectType() {
        return Person.class;
    }
}

```

17. Bean的作用域

- 1、**singleton**：单例，Spring中的bean默认都是单例的。
- 2、**prototype**：每次请求都会创建一个新的bean实例。
- 3、**request**：每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效。
- 4、**session**：每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP session内有效。
- 5、**global-session**：全局session作用域。

18. Spring自动装配的方式有哪些？

Spring的自动装配有三种模式：**byType**(根据类型)、**byName**(根据名称)、**constructor**(根据构造函数)。

byType

找到与依赖类型相同的bean注入到另外的bean中，这个过程需要借助setter注入来完成，因此必须存在set方法，否则注入失败。

当xml文件中存在多个相同类型名称不同的实例Bean时，Spring容器依赖注入仍然会失败，因为存在多种适合的选项，Spring容器无法知道该注入那种，此时我们需要为Spring容器提供帮助，指定注入那个Bean实例。可以通过 `<bean>` 标签的autowire-candidate设置为false来过滤那些不需要注入的实例Bean

```

<bean id="userDao" class="com.zejian.spring.springIoc.dao.impl.UserDaoImpl" />

<!-- autowire-candidate="false" 过滤该类型 -->
<bean id="userDao2" autowire-candidate="false"
class="com.zejian.spring.springIoc.dao.impl.UserDaoImpl" />

<!-- byType 根据类型自动装配userDao-->
<bean id="userService" autowire="byType"
class="com.zejian.spring.springIoc.service.impl.UserServiceImpl" />

```

byName

将属性名与bean名称进行匹配，如果找到则注入依赖bean。

```

<bean id="userDao" class="com.zejian.spring.springIoc.dao.impl.UserDaoImpl" />
<bean id="userDao2" class="com.zejian.spring.springIoc.dao.impl.UserDaoImpl" />

<!-- byName 根据名称自动装配，找到UserServiceImpl名为 userDao属性并注入-->
<bean id="userService" autowire="byName"
class="com.zejian.spring.springIoc.service.impl.UserServiceImpl" />

```

constructor

存在单个实例则优先按类型进行参数匹配（无论名称是否匹配），当存在多个类型相同实例时，按名称优先匹配，如果没有找到对应名称，则注入失败。

19. @Autowired和@Resource的区别？

Autowired是spring的注解。默认情况下@Autowired是按类型匹配的(byType)。如果需要按名称(byName)匹配的话，可以使用@Qualifier注解与@Autowired结合。@Autowired 可以传递一个 `required=false` 的属性，false指明当userDao实例存在就注入不存就忽略，如果为true，就必须注入，若userDao实例不存在，就抛出异常。

```

public class UserServiceImpl implements UserService {
    //标注成员变量
    @Autowired
    @Qualifier("userDao1")
    private UserDao userDao;
}

```

Resource是j2ee的注解，默认按 byName模式自动注入。@Resource有两个中重要的属性：name和type。name属性指定bean的名字，type属性则指定bean的类型。因此使用name属性，则按byName模式的自动注入策略，如果使用type属性，则按 byType模式自动注入策略。倘若既不指定name也不指定type属性，Spring容器将通过反射技术默认按byName模式注入。

```

@Resource(name="userDao")
private UserDao userDao; //用于成员变量

//也可以用于set方法标注
@Resource(name="userDao")
public void setUserDao(UserDao userDao) {
    this.userDao= userDao;
}

```

上述两种自动装配的依赖注入并不适合简单值类型，如int、boolean、long、String以及Enum等，对于这些类型，Spring容器也提供了@Value注入的方式。

@Value和@Autowired、@Resource类似，也是用来对属性进行注入的，只不过@Value是用来从Properties文件中来获取值的，并且@Value可以解析SpEL(Spring表达式)。

比如，jdbc.properties文件如下：

```

jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://127.0.0.1:3306/test?characterEncoding=UTF-8&allowMultiQueries=true
jdbc.username=root
jdbc.password=root

```

利用注解@Value获取jdbc.url和jdbc.username的值，实现如下：

```

public class UserServiceImpl implements UserService {
    //占位符方式
    @Value("${jdbc.url}")
    private String url;
    //SpEL表达式方式，其中代表xml配置文件中的id值configProperties
    @Value("#{configProperties['jdbc.username']}")
    private String userName;
}

```

20. @Qualifier 注解有什么作用

当需要创建多个相同类型的 bean 并希望仅使用属性装配其中一个 bean 时，可以使用 `@Qualifier` 注解和 `@Autowired` 通过指定应该装配哪个 bean 来消除歧义。

21. @Bean和@Component有什么区别？

都是使用注解定义 Bean。`@Bean` 是使用 Java 代码装配 Bean，`@Component` 是自动装配 Bean。

`@Component` 注解用在类上，表明一个类会作为组件类，并告知Spring要为这个类创建bean，每个类对应一个 Bean。

`@Bean` 注解用在方法上，表示这个方法会返回一个 Bean。`@Bean` 需要在配置类中使用，即类上需要加上`@Configuration`注解。

```

@Component
public class Student {
    private String name = "lkm";

    public String getName() {
        return name;
    }
}

@Configuration
public class WebSocketConfig {
    @Bean
    public Student student(){
        return new Student();
    }
}

```

`@Bean` 注解更加灵活。当需要将第三方类装配到 Spring 容器中，因为没办法源代码上添加 `@Component` 注解，只能使用 `@Bean` 注解的方式，当然也可以使用 xml 的方式。

22. @Component、@Controller、@Repository和@Service 的区别？

`@Component`：最普通的组件，可以被注入到spring容器进行管理。

`@Controller`：将类标记为 Spring Web MVC 控制器。

`@Service`：将类标记为业务层组件。

`@Repository`：将类标记为数据访问组件，即DAO组件。

23. Spring 事务实现方式有哪些？

事务就是一系列的操作原子执行。Spring事务机制主要包括声明式事务和编程式事务。

- **编程式事务**：通过编程的方式管理事务，这种方式带来了很大的灵活性，但很难维护。
- **声明式事务**：将事务管理代码从业务方法中分离出来，通过aop进行封装。Spring声明式事务使得我们无需去处理获得连接、关闭连接、事务提交和回滚等这些操作。使用 `@Transactional` 注解开启声明式事务。

`@Transactional` 相关属性如下：

属性	类型	描述
value	String	可选的限定描述符，指定使用的事务管理器
propagation	enum: Propagation	可选的事务传播行为设置
isolation	enum: Isolation	可选的事务隔离级别设置
readOnly	boolean	读写或只读事务，默认读写
timeout	int (in seconds granularity)	事务超时时间设置
rollbackFor	Class对象数组，必须继承自Throwable	导致事务回滚的异常类数组
rollbackForClassName	类名数组，必须继承自Throwable	导致事务回滚的异常类名字数组
noRollbackFor	Class对象数组，必须继承自Throwable	不会导致事务回滚的异常类数组
noRollbackForClassName	类名数组，必须继承自Throwable	不会导致事务回滚的异常类名字数组

24. 有哪些事务传播行为？

在TransactionDefinition接口中定义了七个事务传播行为：

1. `PROPAGATION_REQUIRED` 如果存在一个事务，则支持当前事务。如果没有事务则开启一个新的事务。如果嵌套调用的两个方法都加了事务注解，并且运行在相同线程中，则这两个方法使用相同的事务中。如果运行在不同线程中，则会开启新的事务。
2. `PROPAGATION_SUPPORTS` 如果存在一个事务，支持当前事务。如果没有事务，则非事务的执行。
3. `PROPAGATION_MANDATORY` 如果已经存在一个事务，支持当前事务。如果不存在事务，则抛出异常 `IllegalTransactionStateException`。
4. `PROPAGATION_REQUIRES_NEW` 总是开启一个新的事务。需要使用JtaTransactionManager作为事务管理器。
5. `PROPAGATION_NOT_SUPPORTED` 总是非事务地执行，并挂起任何存在的事务。需要使用JtaTransactionManager作为事务管理器。
6. `PROPAGATION_NEVER` 总是非事务地执行，如果存在一个活动事务，则抛出异常。
7. `PROPAGATION_NESTED` 如果一个活动的事务存在，则运行在一个嵌套的事务中。如果没有活动事务，则按PROPAGATION_REQUIRED 属性执行。

PROPAGATION_NESTED 与PROPAGATION_REQUIRES_NEW的区别：

使用 `PROPAGATION_REQUIRES_NEW` 时，内层事务与外层事务是两个独立的事务。一旦内层事务进行了提交后，外层事务不能对其进行回滚。两个事务互不影响。

使用 `PROPAGATION_NESTED` 时，外层事务的回滚可以引起内层事务的回滚。而内层事务的异常并不会导致外层事务的回滚，它是一个真正的嵌套事务。

25. Spring事务在什么情况下会失效？

1. 访问权限问题

java的访问权限主要有四种：private、default、protected、public，它们的权限从左到右，依次变大。

如果事务方法的访问权限不是定义成public，这样会导致事务失效，因为spring要求被代理方法必须是 `public` 的。

翻开源码，可以看到，在 `AbstractFallbackTransactionAttributeSource` 类的 `computeTransactionAttribute` 方法中有个判断，如果目标方法不是public，则返回null，即不支持事务。

```
protected TransactionAttribute computeTransactionAttribute(Method method,
    @Nullable Class<?> targetClass) {
    // Don't allow no-public methods as required.
    if (allowPublicMethodsOnly() && !Modifier.isPublic(method.getModifiers())) {
        return null;
    }
    ...
}
```

2. 方法用final修饰

如果事务方法用final修饰，将会导致事务失效。因为spring事务底层使用了aop，也就是通过jdk动态代理或者cglib，帮我们生成了代理类，在代理类中实现的事务功能。

但如果某个方法用final修饰了，那么在它的代理类中，就无法重写该方法，而添加事务功能。

同理，如果某个方法是static的，同样无法通过动态代理，变成事务方法。

3. 对象没有被spring管理

使用spring事务的前提是：对象要被spring管理，需要创建bean实例。如果类没有加@Controller、@Service、@Component、@Repository等注解，即该类没有交给spring去管理，那么它的方法也不会生成事务。

4. 表不支持事务

如果MySQL使用的存储引擎是myisam，这样的话是不支持事务的。因为myisam存储引擎不支持事务。

5. 方法内部调用

如下代码所示，update方法上面没有加 `@Transactional` 注解，调用有 `@Transactional` 注解的updateOrder方法，updateOrder方法上的事务会失效。

因为发生了自身调用，调用该类自己的方法，而没有经过Spring的代理类，只有在外部调用事务才会生效。

```

@Service
public class OrderServiceImpl implements OrderService {

    public void update(Order order) {
        this.updateOrder(order);
    }

    @Transactional
    public void updateOrder(Order order) {
        // update order
    }
}

```

解决方法：

- 1、再声明一个service，将内部调用改为外部调用
- 2、使用编程式事务
- 3、使用AopContext.currentProxy()获取代理对象

```

@Servcie
public class OrderServiceImpl implements OrderService {

    public void update(Order order) {
        ((OrderService)AopContext.currentProxy()).updateOrder(order);
    }

    @Transactional
    public void updateOrder(Order order) {
        // update order
    }
}

```

6.未开启事务

如果是spring项目，则需要在配置文件中手动配置事务相关参数。如果忘了配置，事务肯定是不生效的。

如果是springboot项目，那么不需要手动配置。因为springboot已经在

`DataSourceTransactionManagerAutoConfiguration` 类中帮我们开启了事务。

7.吞了异常

有时候事务不会回滚，有可能是在代码中手动catch了异常。因为开发者自己捕获了异常，又没有手动抛出，把异常吞掉了，这种情况下spring事务不会回滚。

如果想要spring事务能够正常回滚，必须抛出它能够处理的异常。如果没有抛异常，则spring认为程序是正常的。

26. Spring怎么解决循环依赖的问题？

首先，有两种Bean注入的方式。

构造器注入和属性注入。

对于构造器注入的循环依赖，Spring处理不了，会直接抛出 `BeanCurrentlyInCreationException` 异常。

对于属性注入的循环依赖（单例模式下），是通过三级缓存处理来循环依赖的。

而非单例对象的循环依赖，则无法处理。

下面分析单例模式下属性注入的循环依赖是怎么处理的：

首先，Spring单例对象的初始化大略分为三步：

1. `createBeanInstance`：实例化bean，使用构造方法创建对象，为对象分配内存。
2. `populateBean`：进行依赖注入。
3. `initializeBean`：初始化bean。

Spring为了解决单例的循环依赖问题，使用了三级缓存：

`singletonObjects`：完成了初始化的单例对象map，bean name --> bean instance

`earlySingletonObjects`：完成实例化未初始化的单例对象map，bean name --> bean instance

`singletonFactories`：单例对象工厂map，bean name --> ObjectFactory，单例对象实例化完成之后会加入singletonFactories。

在调用`createBeanInstance`进行实例化之后，会调用`addSingletonFactory`，将单例对象放到`singletonFactories`中。

```
protected void addSingletonFactory(String beanName, ObjectFactory<?>
singletonFactory) {
    Assert.notNull(singletonFactory, "Singleton factory must not be null");
    synchronized (this.singletonObjects) {
        if (!this.singletonObjects.containsKey(beanName)) {
            this.singletonFactories.put(beanName, singletonFactory);
            this.earlySingletonObjects.remove(beanName);
            this.registeredSingletons.add(beanName);
        }
    }
}
```

假如A依赖了B的实例对象，同时B也依赖A的实例对象。

1. A首先完成了实例化，并且将自己添加到`singletonFactories`中
2. 接着进行依赖注入，发现自己依赖对象B，此时就尝试去`get(B)`
3. 发现B还没有被实例化，对B进行实例化
4. 然后B在初始化的时候发现自己依赖了对象A，于是尝试`get(A)`，尝试一级缓存`singletonObjects`和二级缓存`earlySingletonObjects`没找到，尝试三级缓存`singletonFactories`，由于A初始化时将自己添加到了`singletonFactories`，所以B可以拿到A对象，然后将A从三级缓存中移到二级缓存中
5. B拿到A对象后顺利完成了初始化，然后将自己放入到一级缓存`singletonObjects`中
6. 此时返回A中，A此时能拿到B的对象顺利完成自己的初始化

由此看出，属性注入的循环依赖主要是通过将实例化完成的bean添加到`singletonFactories`来实现的。而使用构造器依赖注入的bean在实例化的时候会进行依赖注入，不会被添加到`singletonFactories`中。比如A和B都是通过构造器依赖注入，A在调用构造器进行实例化的时候，发现自己依赖B，B没有被实例化，就会对B进行实例化，此时A未实例化完成，不会被添加到`singletonFactories`。而B依赖于A，B会去三级缓存寻找A对象，发现不存在，于是又会实例化A，A实例化了两次，从而导致抛异常。

总结：1、利用缓存识别已经遍历过的节点；2、利用Java引用，先提前设置对象地址，后完善对象。

27. Spring启动过程

1. 读取web.xml文件。
2. 创建 ServletContext，为 ioc 容器提供宿主环境。
3. 触发容器初始化事件，调用 contextLoaderListener.contextInitialized()方法，在这个方法会初始化一个应用上下文WebApplicationContext，即 Spring 的 ioc 容器。ioc 容器初始化完成之后，会被存储到 ServletContext 中。
4. 初始化web.xml中配置的Servlet。如DispatcherServlet，用于匹配、处理每个servlet请求。

28. Spring 的单例 Bean 是否有并发安全问题？

当多个用户同时请求一个服务时，容器会给每一个请求分配一个线程，这时多个线程会并发执行该请求对应的业务逻辑，如果业务逻辑有对单例状态的修改（体现为此单例的成员属性），则必须考虑线程安全问题。

无状态bean和有状态bean

- 有实例变量的bean，可以保存数据，是非线程安全的。
- 没有实例变量的bean，不能保存数据，是线程安全的。

在Spring中无状态的Bean适合用单例模式，这样可以共享实例提高性能。有状态的Bean在多线程环境下不安全，一般用 **Prototype** 模式或者使用 **ThreadLocal** 解决线程安全问题。

29. Spring Bean如何保证并发安全？

Spring的Bean默认都是单例的，某些情况下，单例是并发不安全的。

以 **Controller** 举例，假如我们在 **Controller** 中定义了成员变量。当多个请求来临，进入的都是同一个单例的 **Controller** 对象，并对此成员变量的值进行修改操作，因此会互相影响，会有并发安全的问题。

应该怎么解决呢？

为了让多个HTTP请求之间不互相影响，可以采取以下措施：

1、单例变原型

对 web 项目，可以 **Controller** 类上加注解 **@Scope("prototype")** 或 **@Scope("request")**，对非 web 项目，在 **Component** 类上添加注解 **@Scope("prototype")**。

这种方式实现起来非常简单，但是很大程度上增大了 Bean 创建实例化销毁的服务器资源开销。

2、尽量避免使用成员变量

在业务允许的条件下，可以将成员变量替换为方法中的局部变量。这种方式个人认为是最恰当的。

3、使用并发安全的类

如果非要在单例Bean中使用成员变量，可以考虑使用并发安全的容器，如 **ConcurrentHashMap**、**ConcurrentHashSet** 等等，将我们的成员变量包装到这些并发安全的容器中进行管理即可。

4、分布式或微服务的并发安全

如果还要进一步考虑到微服务或分布式服务的影响，方式3便不合适了。这种情况下可以借助于可以共享某些信息的分布式缓存中间件，如Redis等。这样即可保证同一种服务的不同服务实例都拥有同一份共享信息了。

30. @Async注解的原理

当我们调用第三方接口或者方法的时候，我们不需要等待方法返回就去执行其它逻辑，这时如果响应时间过长，就会极大的影响程序的执行效率。所以这时就需要使用异步方法来并行执行我们的逻辑。在springboot中可以使用@Async注解实现异步操作。

使用@Async注解实现异步操作的步骤：

1.首先在启动类上添加 @EnableAsync 注解。

```
@Configuration
@EnableAsync
public class App {
    public static void main(String[] args) {
        ApplicationContext ctx = new
            AnnotationConfigApplicationContext(App.class);
        MyAsync service = ctx.getBean(MyAsync.class);
        System.out.println(service.getClass());
        service.async1();
        System.out.println("main thread finish...");
    }
}
```

2.在对应的方法上添加@Async注解。

```
@Component
public class MyAsync {
    @Async
    public void asyncTest() {
        try {
            TimeUnit.SECONDS.sleep(20);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("asyncTest...");
    }
}
```

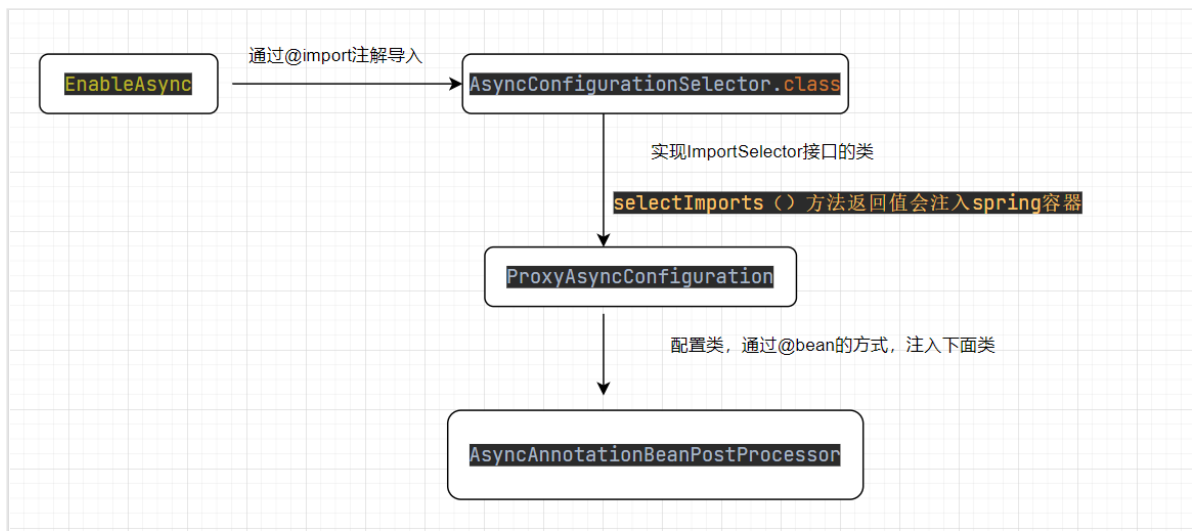
运行代码，控制台输出：

```
main thread finish...
asyncTest...
```

证明asyncTest方法异步执行了。

原理：

我们在主启动类上贴了一个@EnableAsync注解，才能使用@Async生效。@EnableAsync的作用是通过@import导入了AsyncConfigurationSelector。在AsyncConfigurationSelector的selectImports方法将ProxyAsyncConfiguration定义为Bean注入容器。在ProxyAsyncConfiguration中通过@Bean的方式注入AsyncAnnotationBeanPostProcessor类。



代码如下：

```
@Import(AsyncConfigurationSelector.class)
public @interface EnableAsync {
}

public class AsyncConfigurationSelector extends
AdviceModeImportSelector<EnableAsync> {
    public String[] selectImports(AdviceMode adviceMode) {
        switch (adviceMode) {
            case PROXY:
                return new String[] { ProxyAsyncConfiguration.class.getName() };
                //...
        }
    }
}

public class ProxyAsyncConfiguration extends AbstractAsyncConfiguration {
    @Bean(name = TaskManagementConfigUtils.ASYNC_ANNOTATION_PROCESSOR_BEAN_NAME)
    public AsyncAnnotationBeanPostProcessor asyncAdvisor() {
        //创建postProcessor
        AsyncAnnotationBeanPostProcessor bpp = new
AsyncAnnotationBeanPostProcessor();
        //...
    }
}
```

AsyncAnnotationBeanPostProcessor往往期创建了一个增强器AsyncAnnotationAdvisor。在AsyncAnnotationAdvisor的buildAdvice方法中，创建了AnnotationAsyncExecutionInterceptor。

```
public class AsyncAnnotationBeanPostProcessor extends
AbstractBeanFactoryAwareAdvisingPostProcessor {
    @Override
    public void setBeanFactory(BeansFactory beanFactory) {
        super.setBeanFactory(beanFactory);
        //创建一个增强器
        AsyncAnnotationAdvisor advisor = new
AsyncAnnotationAdvisor(this.executor, this.exceptionHandler);
        //...
        advisor.setBeanFactory(beanFactory);
        this.advisor = advisor;
    }
}
```

```

    }
}

public class AsyncAnnotationAdvisor extends AbstractPointcutAdvisor implements
BeanFactoryAware {
    public AsyncAnnotationAdvisor(
        @Nullable Supplier<Executor> executor, @Nullable
Supplier<AsyncUncaughtExceptionHandler> exceptionHandler) {
        //增强方法
        this.advice = buildAdvice(executor, exceptionHandler);
        this.pointcut = buildPointcut(asyncAnnotationTypes);
    }

    // 委托给AnnotationAsyncExecutionInterceptor拦截器
    protected Advice buildAdvice(
        @Nullable Supplier<Executor> executor, @Nullable
Supplier<AsyncUncaughtExceptionHandler> exceptionHandler) {
        //拦截器
        AnnotationAsyncExecutionInterceptor interceptor = new
AnnotationAsyncExecutionInterceptor(null);
        interceptor.configure(executor, exceptionHandler);
        return interceptor;
    }
}

```

AnnotationAsyncExecutionInterceptor继承自AsyncExecutionInterceptor，间接实现了MethodInterceptor。该拦截器的实现的invoke方法把原来方法的调用提交到新的线程池执行，从而实现了方法的异步。

```

public class AsyncExecutionInterceptor extends AsyncExecutionAspectSupport
implements MethodInterceptor, Ordered {
    public Object invoke(final MethodInvocation invocation) throws Throwable {
        //...
        //构建放到AsyncTaskExecutor执行Callable Task
        Callable<Object> task = () -> {
            //...
        };
        //提交到新的线程池执行
        return doSubmit(task, executor, invocation.getMethod().getReturnType());
    }
}

```

由上面分析可以看到，@Async注解其实是通过代理的方式来实现异步调用的。

那使用@Async有什么要注意的呢？

- 1.使用@Aysnc的时候最好配置一个线程池Executor以让线程复用节省资源，或者为SimpleAsyncTaskExecutor设置基于线程池实现的ThreadFactory，在否则会默认使用SimpleAsyncTaskExecutor，该executor会在每次调用时新建一个线程。
- 2.调用本类的异步方法是不会起作用的。这种方式绕过了代理而直接调用了方法，@Async注解会失效。



扫码关注我



微信搜索



程序员大彬

公众号后台回复【面试】获取面试手册
PDF最新版

SpringMVC

1. 说说你对 SpringMVC 的理解

SpringMVC是一种基于 Java 的实现MVC设计模型的请求驱动类型的轻量级Web框架，属于Spring框架的一个模块。

它通过一套注解，让一个简单的Java类成为处理请求的控制器，而无须实现任何接口。同时它还支持 RESTful编程风格的请求。

2. 什么是MVC模式？

MVC的全名是 **Model View Controller**，是模型(model) - 视图(view) - 控制器(controller)的缩写，是一种软件设计典范。它是用一种业务逻辑、数据与界面显示分离的方法来组织代码，将众多的业务逻辑聚集到一个部件里面，在需要改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑，达到减少编码的时间。

View，视图是指用户看到并与之交互的界面。比如由html元素组成的网页界面，或者软件的客户端界面。MVC的好处之一在于它能为应用程序处理很多不同的视图。在视图中其实没有真正的处理发生，它只是作为一种输出数据并允许用户操纵的方式。

model，模型是指模型表示业务规则。在MVC的三个部件中，模型拥有最多的处理任务。被模型返回的数据是中立的，模型与数据格式无关，这样一个模型能为多个视图提供数据，由于应用于模型的代码只需写一次就可以被多个视图重用，所以减少了代码的重复性。

controller，控制器是指控制器接受用户的输入并调用模型和视图去完成用户的需求，控制器本身不输出任何东西和做任何处理。它只是接收请求并决定调用哪个模型构件去处理请求，然后再确定用哪个视图来显示返回的数据。

3. SpringMVC 有哪些优点？

1. 与 Spring 集成使用非常方便，生态好。
2. 配置简单，快速上手。
3. 支持 RESTful 风格。
4. 支持各种视图技术，支持各种请求资源映射策略。

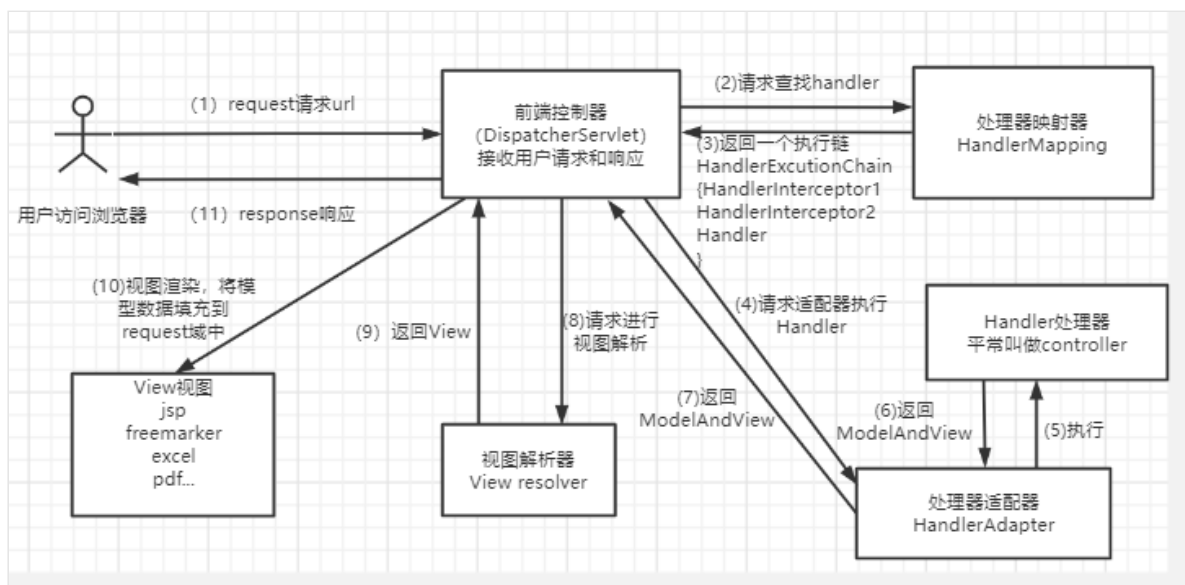
4. Spring MVC和Struts的区别

1. Spring MVC是基于方法开发，Struts2是基于类开发的。
 - Spring MVC会将用户请求的URL路径信息与Controller的某个方法进行映射，所有请求参数会注入到对应方法的形参上，生成Handler对象，对象中只有一个方法；
 - Struts每处理一次请求都会实例一个Action，Action类的所有方法使用的请求参数都是Action类中的成员变量，随着方法增多，整个Action也会变得混乱。
2. Spring MVC支持单例开发模式，Struts只能使用多例
 - Struts由于只能通过类的成员变量接收参数，故只能使用多例。
3. Struts2 的核心是基于一个Filter即StrutsPreparedAndExcuteFilter，Spring MVC的核心是基于一个Servlet即DispatcherServlet(前端控制器)。
4. Struts处理速度稍微比Spring MVC慢，Struts使用了Struts标签，加载数据较慢。

5. Spring MVC的工作原理

Spring MVC的工作原理如下：

1. DispatcherServlet 接收用户的请求
2. 找到用于处理request的 handler 和 Interceptors，构造成 HandlerExecutionChain 执行链
3. 找到 handler 相对应的 HandlerAdapter
4. 执行所有注册拦截器的preHandler方法
5. 调用 HandlerAdapter 的 handle() 方法处理请求，返回 ModelAndView
6. 倒序执行所有注册拦截器的postHandler方法
7. 请求视图解析和视图渲染



6. Spring MVC的主要组件?

- 前端控制器 (DispatcherServlet)：接收用户请求，给用户返回结果。
- 处理器映射器 (HandlerMapping)：根据请求的url路径，通过注解或者xml配置，寻找匹配的Handler。
- 处理器适配器 (HandlerAdapter)：Handler 的适配器，调用 handler 的方法处理请求。
- 处理器 (Handler)：执行相关的请求处理逻辑，并返回相应的数据和视图信息，将其封装到 ModelAndView 对象中。
- 视图解析器 (ViewResolver)：将逻辑视图名解析成真正的视图View。
- 视图 (View)：接口类，实现类可支持不同的View类型 (JSP、FreeMarker、Excel等)。

7. Spring MVC的常用注解由有哪些?

- @Controller：用于标识此类的实例是一个控制器。
- @RequestMapping：映射Web请求（访问路径和参数）。
- @ResponseBody：注解返回数据而不是返回页面
- @RequestBody：注解实现接收 http 请求的 json 数据，将 json 数据转换为 java 对象。
- @PathVariable：获得URL中路径变量中的值
- @RestController：@Controller+@ResponseBody
- @ExceptionHandler标识一个方法为全局异常处理的方法。

8. @Controller 注解有什么用?

@Controller 注解标记一个类为 Spring Web MVC 控制器。Spring MVC 会将扫描到该注解的类，然后扫描这个类下面带有 **@RequestMapping** 注解的方法，根据注解信息，为这个方法生成一个对应的处理器对象，在上面的 HandlerMapping 和 HandlerAdapter 组件中讲到过。

当然，除了添加 **@Controller** 注解这种方式以外，你还可以实现 Spring MVC 提供的 **Controller** 或者 **HttpRequestHandler** 接口，对应的实现类也会被作为一个处理器对象

9. @RequestMapping 注解有什么用？

`@RequestMapping` 注解，用于配置处理器的 HTTP 请求方法，URI 等信息，这样才能将请求和方法进行映射。这个注解可以作用于类上面，也可以作用于方法上面，在类上面一般是配置这个控制器的 URI 前缀。

10. @RestController 和 @Controller 有什么区别？

`@RestController` 注解，在 `@Controller` 基础上，增加了 `@ResponseBody` 注解，更加适合目前前后端分离的架构下，提供 Restful API，返回 JSON 数据格式。

11. @RequestMapping 和 @GetMapping 注解有什么不同？

1. `@RequestMapping`：可注解在类和方法上；`@GetMapping` 仅可注册在方法上
2. `@RequestMapping`：可进行 GET、POST、PUT、DELETE 等请求方法；`@GetMapping` 是 `@RequestMapping` 的 GET 请求方法的特例。

12. @RequestParam 和 @PathVariable 两个注解的区别

两个注解都用于方法参数，获取参数值的方式不同，`@RequestParam` 注解的参数从请求携带的参数中获取，而 `@PathVariable` 注解从请求的 URI 中获取

13. @RequestBody和@RequestParam的区别

`@RequestBody` 一般处理的是在 ajax 请求中声明 `contentType: "application/json; charset=utf-8"` 时候。也就是 json 数据或者 xml 数据。

`@RequestParam` 一般就是在 ajax 里面没有声明 `contentType` 的时候，为默认的 `x-www-form-urlencoded` 格式时。

14. Spring MVC的异常处理

可以将异常抛给 Spring 框架，由 Spring 框架来处理；我们只需要配置简单的异常处理器，在异常处理器中添视图页面即可。

- 使用系统定义好的异常处理器 `SimpleMappingExceptionResolver`
- 使用自定义异常处理器
- 使用异常处理注解

15. SpringMVC 用什么对象从后台向前台传递数据的？

1. 将数据绑定到 request；
2. 返回 `ModelAndView`；
3. 通过 `ModelMap` 对象，可以在这个对象里面调用 `put` 方法，把对象加到里面，前端就可以通过 `el` 表达式拿到；
4. 绑定数据到 Session 中。

16. SpringMvc的Controller是不是单例模式？

单例模式。在多线程访问的时候有线程安全问题，解决方案是在控制器里面不要写可变状态量，如果需要使用这些可变状态，可以使用 `ThreadLocal`，为每个线程单独生成一份变量副本，独立操作，互不影响。

17. 介绍下 Spring MVC 拦截器？

Spring MVC 拦截器对应HandlerInterceor接口，该接口位于org.springframework.web.servlet的包中，定义了三个方法，若要实现该接口，就要实现其三个方法：

1. **前置处理 (preHandle()方法)**：该方法在执行控制器方法之前执行。返回值为Boolean类型，如果返回false，表示拦截请求，不再向下执行，如果返回true，表示放行，程序继续向下执行（如果后面没有其他Interceptor，就会执行controller方法）。所以此方法可对请求进行判断，决定程序是否继续执行，或者进行一些初始化操作及对请求进行预处理。
2. **后置处理 (postHandle()方法)**：该方法在执行控制器方法调用之后，且在返回ModelAndView之前执行。由于该方法会在DispatcherServlet进行返回视图渲染之前被调用，所以此方法多被用于处理返回的视图，可通过此方法对请求域中的模型和视图做进一步的修改。
3. **已完成处理 (afterCompletion()方法)**：该方法在执行完控制器之后执行，由于是在Controller方法执行完毕后执行该方法，所以该方法适合进行一些资源清理，记录日志信息等处理操作。

可以通过拦截器进行权限检验，参数校验，记录日志等操作

18. SpringMvc怎么配置拦截器？

有两种写法，一种是实现HandlerInterceptor接口，另外一种是继承适配器类，接着在接口方法当中，实现处理逻辑；然后在SpringMvc的配置文件中配置拦截器即可：

```
<!-- 配置SpringMvc的拦截器 -->
<mvc:interceptors>
    <bean id="myInterceptor" class="com.dabin.MyHandlerInterceptor"></bean>

    <!-- 只拦截部分请求 -->
    <mvc:interceptor>
        <mvc:mapping path="/xxx.do" />
        <bean class="com.dabin.MyHandlerInterceptorAdapter" />
    </mvc:interceptor>
</mvc:interceptors>
```

19. Spring MVC 的拦截器和 Filter 过滤器有什么差别？

有以下几点：

- **功能相同**：拦截器和 Filter 都能实现相应的功能
- **容器不同**：拦截器构建在 Spring MVC 体系中；Filter 构建在 Servlet 容器之上
- **使用便利性不同**：拦截器提供了三个方法，分别在不同的时机执行；过滤器仅提供一个方法

20. 什么是REST？

REST，英文全称，Resource Representational State Transfer，对资源的访问状态的变化通过url的变化表述出来。

Resource：**资源**。资源是REST架构或者说整个网络处理的核心。

Representational：**某种表现形式**，比如用JSON，XML，JPEG等。

State Transfer：**状态变化**。通过HTTP method实现。

REST描述的是在网络中client和server的一种交互形式。用大白话来说，就是**通过URL就知道要什么资源，通过HTTP method就知道要干什么，通过HTTP status code就知道结果如何。**

举个例子：

```
GET /tasks 获取所有任务
POST /tasks 创建新任务
GET /tasks/{id} 通过任务id获取任务
PUT /tasks/{id} 更新任务
DELETE /tasks/{id} 删除任务
```

GET代表获取一个资源，POST代表添加一个资源，PUT代表修改一个资源，DELETE代表删除一个资源。

server提供的RESTful API中，URL中只使用名词来指定资源，原则上不使用动词。用 **HTTP Status Code** 传递server的状态信息。比如最常用的 200 表示成功，500 表示Server内部错误等。

21. 使用REST有什么优势呢？

第一，**风格统一**了，不会出现 `delUser/deleteUser/removeUser` 各种命名的代码了。

第二，**面向资源**，一目了然，具有自解释性。

第三，**充分利用 HTTP 协议本身语义**。



扫码关注我



微信搜索



程序员大彬

公众号后台回复【面试】获取面试手册
PDF最新版

Springboot

Springboot重要知识点&高频面试题是我的[知识星球](#)内部专属资料，已经整理到Java面试手册完整版。

Java面试手册

面试准备

面试题&重要知识点

- Java
- 数据库
- 常见框架
 - Netty重要知识点总结
 - Spring高频面试题
 - SpringMVC高频面试题
 - Mybatis高频面试题
 - SpringBoot高频面试题**
 - SpringCloud高频面试题
- 消息队列
- 计算机基础
- 分布式
- 开发工具

- 系统设计
- 海量数据场景高频面试题
- 面经篇
- 校招分享

2608 字

保存并发布于：2023-01-01 23:22:41

标题2 24px B I S U 插入 删除 分享

spring-boot-starter-parent 有什么用？

新创建一个 Spring Boot 项目，默认都是有 parent 的，这个 parent 就是 spring-boot-starter-parent，spring-boot-starter-parent 主要有如下作用：

1. 定义了 Java 编译版本。
2. 使用 UTF-8 格式编码。
3. 执行打包操作的配置。
4. 自动化的资源过滤。
5. 自动化的插件配置。
6. 针对 application.properties 和 application.yml 的资源过滤，包括通过 profile 定义的不同环境的配置文件，例如 application-dev.properties 和 application-dev.yml。

Spring Boot 打成的 jar 和普通的 jar 有什么区别？

- Spring Boot 项目最终打包成的 jar 是可执行 jar，这种 jar 可以直接通过 `java -jar xxx.jar` 命令来运行，这种 jar 不可以作为普通的 jar 被其他项目依赖，即使依赖了也无法使用其中的类。

大纲

Springboot的优点

SpringBoot 中的 starter 到底是什么？

运行 SpringBoot 有哪几种方式？

SpringBoot 常用的 Starter 有哪些？

Spring Boot 的核心注解是哪个？

自动配置原理

实现自动配置

@Value注解的原理

Spring Boot 需要独立的容器运行吗？

Spring Boot 支持哪些日志框架？

YAML 配置的优势在哪里？

什么是 Spring Profiles？

SpringBoot多数据源事务如何管理

spring-boot-starter-parent 有什么...

Spring Boot 打成的 jar 和普通的 jar ...

SpringBoot多数据源拆分的思路

面试准备		
面试题&重要知识点		
Java		
数据库		
常见框架		
消息队列		
计算机基础		
分布式		
开发工具		
系统设计		
如何设计一个抢红包系统		01-11 22:31
权限系统设计方案		01-10 23:51
实时订阅推送设计与实现		01-04 16:21
如何用 Redis 统计用户访问量？		01-05 00:09
扫码登录原理		01-11 23:20
订单30分钟未支付自动取消怎么实现？		01-02 14:29
如何设计一个短链系统？		01-03 00:17
秒杀系统		01-02 14:27
秒杀系统设计的5个要点		01-04 16:22
微信红包后台系统设计		01-02 14:27
单点登录（SSO）的设计与实现		01-02 14:26
如何设计一个优惠券系统？		01-02 14:26
海量数据场景高频面试题		
面经篇		
校招分享		

除了Java面试手册完整版之外，星球还有很多其他**优质资料**，比如包括Java项目、进阶知识、实战经验总结、优质书籍、笔试面试资源等等。

👉 优质学习资源汇总：

- 1、阿里巴巴技术团队的实战经验总结，链接：<https://t.zsxq.com/08TUWuPRg>
- 2、Tomcat架构解析、性能优化、源码解读：<https://t.zsxq.com/09XZgyC7z>
- 3、尚学堂2022版全套的java学习视频（300集），链接：<https://t.zsxq.com/08NfkBCup>
- 4、并发编程视频教程，包含：线程基础、线程协作、并发工具类、锁、AQS、线程池、实战项目等：<https://t.zsxq.com/09vF2onhe>
- 5、LeetCode刷题笔记（包含C/C++、Java、Go三个版本），链接：<https://t.zsxq.com/080mxtafG>
- 6、尚硅谷Java学科全套教程（总207.77GB），链接：<https://t.zsxq.com/0807PC4s4>
- 7、自动动手写Java虚拟机PDF，链接：<https://t.zsxq.com/08jBRPI5a>
- 8、字节大佬总结的Java面试资料，涵盖Java开发常见的知识点，链接：<https://t.zsxq.com/08UxPuP34>
- 9、阿里的算法笔试题精选合集，涵盖70+算法题目、大厂笔试考点，链接：<https://t.zsxq.com/082R1dqFF>
- 10、Java工程师进阶知识大全，包含高并发架构、分布式架构，链接：<https://t.zsxq.com/08j7WWUZn>
- 11、MySQL优质教程，包含：6 大范式讲解、7 大日志剖析、7 大 SQL 性能分析工具等内容，链接：<https://t.zsxq.com/08d6j44JK>

- 12、阿里巴巴微服务治理白皮书，高效构建完整的微服务治理体系。链接：<https://t.zsxq.com/09fkXNKk>
- 13、RocketMQ 从入门到实战。链接：<https://t.zsxq.com/09MecUORI>
- 14、阿里巴巴性能调优实战手册，包含Java编程性能调优、多线程性能调优、JVM性能监测及调优、设计模式调优、数据库性能调优、实战演练六大模块。链接：<https://t.zsxq.com/09gdKMpY8>
- 15、Java面试真题，跟面试手册内容互补，建议准备面试的手两份一起过一遍！链接：<https://t.zsxq.com/09rVOpWNh>
- 16、畅购商城视频教程（含源码）。链接：<https://t.zsxq.com/09LOcfEDB>
- 17、面试题合集，包含技术和非技术问题，如Dubbo面试题、设计模式面试题、Netty面试题、Tomcat面试题、Nginx面试题、HR面试常见问题、英语面试题等。链接：<https://t.zsxq.com/09G34Sf16>
- 18、精选1000套简历模板，超实用！链接：<https://t.zsxq.com/096963tQt>
- 19、马士兵老师的Java基础视频教程：<https://t.zsxq.com/09twuX7b5>
- 20、MySQL45讲专栏：<https://t.zsxq.com/09V2U7K2A>
- 21、分布式优惠券系统：<https://t.zsxq.com/09ktzHUOZ>
- 22、mpvue课程小程序全栈开发：<https://t.zsxq.com/09aexZzl8>
- 22、微信小程序入门与实战 常用组件API开发技巧项目实战：<https://t.zsxq.com/09aexZzl8>
- 23、尚硅谷最新版尚筹网Java项目实战
<https://t.zsxq.com/094UkGCUy>

- 24、Redis优质教程：<https://t.zsxq.com/090gAzWEx>
- 25、JVM调优&多线程的教程：<https://t.zsxq.com/09u1h71vm>
- 26、Java架构师-技术专家的资料：<https://t.zsxq.com/090oV3kz2>
- 27、SSO单点登录原理：<https://t.zsxq.com/096U2VUJB>
- 28、3天全面深入学习zookeeper（含源码资料）<https://t.zsxq.com/09tijiPEu>
- 29、SpringBoot+SpringCloud全彩指南（终极版），链接：<https://t.zsxq.com/08DEQYliY>
- 30、Mybatis视频教程：<https://t.zsxq.com/09r81SkuK>
- 31、《漫画Java》PDF版本，链接：<https://t.zsxq.com/09L69kPDO>
- 32、Nginx优质视频教程：<https://t.zsxq.com/09gQBCuuF>

专属一对一的提问答疑，帮你解答各种疑难问题，包括自学Java路线、职业规划、面试问题等等。大彬会优先解答球友的问题。



大彬

2023-01-02 20:33

收进专栏

...

Alvin_Xu 提问： 本人是19年毕业至今工作了3年半左右，一直在同一个岗位上，近段时间因为公司项目组资金不足被调岗了，新岗位和自己的技术栈不太匹配，觉得如果一直变动会影响自己的职业发展。

这么些年来一直有点混吃等死，**想在现在开始寻求改变**，希望通过学习，让自己有所变化。不过对于学习方式以及学习路线**很是迷茫**，希望大彬老师可以帮忙解惑，非常感谢。

工作三年半，有点迷茫，想要寻求改变！

技术栈：

有一定的Java基础，对JVM原理，体系结构，垃圾回收机制等有部分了解

熟悉Scala&Spark开发，了解部署spark server项目及脚本

掌握spring, springCloud微服务架构开发，

了解Elastic search,Mule gateway, Feign等的使用

熟悉Jenkins, Kubernetes, PCF, G3等devops部署及调试

熟悉AppDynamic, patrol, LWM等程序监控平台的使用及配置

了解NexusIQ,Checkmarx,sonar等代码质量监控平台的使用及调优

熟悉sqlServer及Mongodb数据库

我一直都是认为，在现在这个就业环境下，“专才”的竞争力是要大于“全才”的，专注一个方向，对你的职业发展更为有利。

从你的技术栈来看，相对还是偏“杂”一些，Java、大数据、devops等都有涉及。而且现在还有调岗的可能，我建议



大彬

2023-01-10 21:41

收进专栏

...

 提问： 想请教一下大彬，对于java开发和大数据开发的明年秋招情况的预测。java岗位多卷度也高，每家公司基本都有相关的职位，大数据开发的话培训班比岗位多，只有大公司和数据公司会有这类岗位。24年秋招的话**选择哪个方向会比较合适呢？**

岗位方向选择：Java还是大数据？

建议选大数据吧。就这几年校招来看，大数据岗位拿offer的难度相比Java还是比较小一些的。而且距离24年秋招还有一年多时间，转大数据完全来得及，而且有了Java基础，再来学大数据，应该会比较快入门。

再说下大数据和后端的差异。大数据门槛比Java高，除了熟悉数据库的操作之外，还要学习大数据整个生态，需要会分布式、数仓、数据分析统计等知识。因为大数据的学习门槛比Java高，所以市场上培训大数据的相比Java会少一些，竞争也相对小，没有Java那么卷。

另外，大数据薪资总体比Java开发高一些（同一家公司同一级别，普通开发岗比大数据开发薪资会少一点），这也是大数据方向的一个优势。

不过呢，小点的公司，可能没有大数据的需求，毕竟业务量不大（小公司通常也不太建议去，坑多）。



2022-12-27 19:29

收进专栏

...

麻烦问一下，遇到一个面试题是：描述最能代表自己编程能力的代码，解释为什么这段代码提现了你的编程能力。这种题怎么答





2人赞过

举了两个例子，帮助球友解答疑惑



大彬：这种可以结合自己平时做的一些项目来讲。
比如说有一些比较复杂的功能，你是如何去实现的，像涉及很多状态流转的，使用了状态机去处理，通过"动作"来进行状态的变更。
或者项目中有很多相似的实现逻辑，把这些相似的逻辑抽象出来，做成公共组件，这也是一种编程能力的体现。
当然还有很多，上面只是两个简答的例子，具体可以结合你自身的项目去思考、挖掘，适当包装一下也是没问题的。
2022-12-27 21:43



大彬：可以从背景（为什么这么做）、实现原理（怎么实现的）、收获（学到了什么）这三个方面来回答
2022-12-27 21:52



回复 大彬：请问工作中如何应用的OOP原则怎么答啊
2022-12-28 13:07

大彬 回复



面向对象编程有六大原则：对象单一职责、里式替换原则、迪米特法则、开闭原则、依赖倒置、接口隔离。
以对象单一职责为例，在设计类的时候，必须职责明确，比如商品类，里面相关的属性和方法都必须跟商品相关，不能出现订单等不相关的内容。
其他原则也是类似，可以看看优质专栏-Java基础部分，有关于面向对象编六大原则的介绍
2022-12-28 14:21

另外星球还提供**简历指导、修改服务**，大彬已经帮**90+**个小伙伴修改了简历，相对还是比较有经验的。



求职意向: Java



生日: 1999.11

籍贯: 贵州三穗

电话: 1

学历: 本科学士

政治面貌: 团员

邮箱: 20

.com



教育背景 Education

2019.09-2023.06

天津工业大学

软件工程

主修课程: 高级程序语言设计、数据库原理及应用、数据结构、算法设计与分析、计算机组成原理、操作系统、计算机网络、编译原理、软件系统分析、软件质量测试等。

校园经历: 在校期间任大创信息部副部长, 负责创业就业信息收集整理, 微信公众号的维护和运营。在校期间注重实践, 参加技术学习型社团, 参与过许多小项目, 曾多次在软件实训中获取优异成绩。



项目经历 Project

2021.12-2022.03

企业网盘系统

项目组长、后端开发

项目简介: 基于 Springboot+Dubbo 开发的, 提供文件管理上传下载共享在线编辑的基础网盘。

● 基于 Redis 做分布式锁, 实现文件切块上传功能;

● WebSocket+Netty 做消息推送, 实现文件共享消息提示;

● Elasticsearch 做全文检索引擎, 实现大量文件的检索;

● HDFS 做分布式文件系统, 实现分布式文件存储。

2022.05-2023.07

微服务电商系统

后端开发

项目简介: SpringCloud 开发的电商系统, 集成了监控中心、网关等系统功能。

● 使用 RabbitMq 做消息队列, 实现延迟消息;

● 基于 SpringSecurity 做登录认证和权限控制;

● 使用 Spring 的 AOP, 添加切面实现接口日志记录;

● Docker 做应用容器引擎, Kubernetes 做应用容器管理平台;



专业技能 Expertise

1. 熟练掌握 java 语法, 和 IO、多线程、反射、异常及一些常用类, 掌握 JVM 原理;
2. 熟悉 Spring、SpringMVC、MyBatis、Springboot, 了解 SpringCloud 等开发框架;
3. 熟悉 MySQL 数据库, 具备一定的 SQL 优化能力; 熟悉 Redis 非关系型数据库;
4. 熟悉 HTML、CSS、JavaScript 等基础前端技术, 了解 Vue 等前端开发框架;
5. 熟练使用 Linux 操作系统; 熟悉常用的设计模式; 熟悉一些开发中常用中间件;
6. 具备一定的需求分析和系统设计能力, 较强的逻辑分析和独立解决问题能力。



自我评价 Evaluation

本人热爱 JAVA 开发工作, 注重实干, 熟悉 SpringBoot 开发, 具备良好的编码规范; 熟悉关系型数据库, 熟练使用 SQL 语言, 掌握基本的 SQL 优化; 服从安排、工作积极主动, 责任心与上进心较强, 具备良好的学习能力与合作的团队精神。

主修课程不用写这么多, 甚至可以不用写, 留点版面给其他内容

感觉似乎没有因果关系。可以描述清晰点

消息推送具体描述下怎么实现的

HDFS面试官应该会问, 思考下会问什么问题

延时消息怎么实现的

项目经历这块如果有数据支撑, 会更好, 比如一些慢查询优化, 从xx秒到xx毫秒等



18235... | 78386...qq.com
CSDN : https://blog.csdn.net/w...n_819295
求职意向：后端开发工程师 (北京)



实习经历

建议实习经历写实习项目，项目经历写自己的项目，一般都是这么写的

京东零售 - 市场营销与用户增长部 - 产品研发部

2021.12 - 2022.06

专业技能

1. 熟练运用 **JAVASE**、**JAVAAE** 主流技术；
2. 掌握 **Java** 基础，熟悉面向对象、集合、多线程、锁、I/O模型；
3. 对 **JVM** 原理有初步的了解，包括内存模型、垃圾回收机制以及垃圾回收算法；
4. 深入理解 **Redis** 数据类型和内部实现、熟悉其使用场景、持久化、淘汰策略；
5. 熟练使用 **MySQL**，熟悉MySQL索引、事务、锁机制、SQL调优并在项目中实际运用；
6. 熟悉 **MyBatis**、**Spring**、**SpringMVC**、**SpringBoot** 等主流开发框架；
7. 熟练使用**Kafka**，熟悉Kafka基本结构、消费者组；
8. 熟悉**Linux**操作系统，**Nginx** 等技术；
9. 了解常用的数据结构和算法；

专业技能可以补充分布式、微服务相关的（如果不熟悉建议了，面试大概率会问到）

项目经历

官旗OMS

工作描述，可以按照三个步骤展开：1.使用什么技术；2.做了什么事情；3.达成什么效果。

2021.12 - 2022.06

项目背景：京东在抖音开设官店为了方便管理订单并履约发货，需要在京东商城与多平台之间建立一个中枢。

工作描述：参与了V1.0、V1.1、V1.2迭代需求的开发。在 V1.0 需求中主要负责拉取抖音平台订单消息，对用户的创建订单、支付订单、风控订单等进行对接，解决了mq的重复消费。在 V1.1 需求中对用户的敏感信息进行了加密处理，使用了延迟消费减少了调用抖音接口的次数，并对SQL进行了优化提高了查询效率。在 V1.2 需求中使用多线程并发的拉取抖音订单详情中的信息，对其中的信息进行过滤筛选并归档到 TIDB 中。在日常工作中还负责将线上数据同步到 hive 数据仓库，实现日报的自动出数。

人脸签到

性能优化，如果有数据指标支撑，会更好。比如从xxx秒优化到xxx毫秒，查询时间减少95%等

2021.08 - 2021.09

- 1、该项目是一个前后端分离的项目，主要功能有授权模块、人脸比对、签到、签退、统计时长
- 2、功能的实现主要是基于SpringBoot + WebSocket + Swagger + Redis 技术栈实现，前后端采用JSON作为交互数据，采用了心跳机制保持了长连接，签到或签退会对已在线的用户进行实时广播，使用 Redis 缓存中间件对热数据进行缓存，并解决了缓存穿透的问题。使用Swagger对代码和注释进行开发文档的生成。

教育经历

项目经历这块，建议写上自己的收获

山西农业信息学院 - 数据科学与大数据技术 本科

2019.09 - 2023.06

荣誉奖项

2020年校级一等奖学金、2021年中国大学生计算机设计大赛三等奖

第十一届蓝桥杯大赛二等奖

社团和组织经历

双体系卓越人才教育基地软件部部长

2020.06 - 2021.06

积极组织学员学习软件编程类相关知识，对下一届学员进行学习进度的规划、以及定期检查学习情况，积极完成老师安排的任务、组织同学们参加各种计算机相关的比赛。

知识星球加入方式：



大彬 送你一张星球优惠券

「大彬的学习圈」

立减

¥ **50**

新人立减券

2023/12/12 13:00 后失效

 知识星球

长按扫码领取优惠 ▶



Mybatis

1. Mybatis是什么?

- MyBatis框架是一个开源的数据持久层框架。
- 它的内部封装了通过JDBC访问数据库的操作，支持普通的SQL查询、存储过程和高级映射，几乎消除了所有的JDBC代码和参数的手工设置以及结果集的检索。
- MyBatis作为持久层框架，其主要思想是将程序中的大量SQL语句剥离出来，配置在配置文件当中，实现SQL的灵活配置。
- 这样做的好处是将SQL与程序代码分离，可以在不修改代码的情况下，直接在配置文件当中修改SQL。

2. ORM是什么

ORM (Object Relational Mapping)，对象关系映射，是一种为了解决关系型数据库数据与简单Java对象 (POJO) 的映射关系的技术。简单的说，ORM是通过使用描述对象和数据库之间映射的元数据，将程序中的对象自动持久化到关系型数据库中。

3. Mybatis和Hibernate的区别?

主要有以下几点区别：

1. Hibernate的**开发难度**大于MyBatis，主要由于Hibernate比较复杂，庞大，学习周期比较长。
2. Hibernate属于**全自动**ORM映射工具，使用Hibernate查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。而MyBatis在查询关联对象或关联集合对象时，需要手动编写sql来完成，所以，称之为半自动ORM映射工具。
3. **数据库扩展性**的区别。Hibernate与数据库具体的关联在XML中，所以HQL对具体是用什么数据库并不是很关心。MyBatis由于所有sql都是依赖数据库书写的，所以扩展性、迁移性比较差。
4. **缓存机制**的区别。Hibernate的二级缓存配置在SessionFactory生成配置文件中详细配置，然后再在具体的表对象映射中配置那种缓存。MyBatis的二级缓存配置都是在每个具体的表对象映射中进行详细配置，这样针对不同的表可以自定义不同的缓冲机制，并且MyBatis可以在命名空间中共享相同的缓存配置和实例，通过Cache-ref来实现。
5. **日志系统完善性**的区别。Hibernate日志系统非常健全，涉及广泛，而MyBatis则除了基本记录功能外，功能薄弱很多。
6. **sql的优化上，Mybatis要比Hibernate方便很多**。由于Mybatis的sql都是写在xml里，因此优化sql比Hibernate方便很多。而Hibernate的sql很多都是自动生成的，无法直接维护sql；总之写sql的灵活度上Hibernate不及Mybatis。

4. MyBatis框架的优缺点及其适用的场合

优点

1. 与JDBC相比，减少了50%以上的代码量。
2. MyBatis是易学的持久层框架，小巧并且简单易学。
3. MyBatis相当灵活，不会对应用程序或者数据库的现有设计强加任何影响，SQL写在XML文件里，从程序代码中彻底分离，降低耦合度，便于统一的管理和优化，并可重用。
4. 提供XML标签，支持编写动态的SQL，满足不同的业务需求。
5. 提供映射标签，支持对象与数据库的ORM字段关系映射。

缺点

1. SQL语句的编写工作量较大，对开发人员编写SQL的能力有一定的要求。
2. SQL语句依赖于数据库，导致数据库不具有好的移植性，不可以随便更换数据库。

适用场景

MyBatis专注于SQL自身，是一个足够灵活的DAO层解决方案。对性能的要求很高，或者需求变化较多的项目，例如Web项目，那么MyBatis是不二的选择。

5. Mybatis的工作原理

- 读取MyBatis配置文件：mybatis-config.xml为MyBatis的全局配置文件，配置了MyBatis的运行环境等信息，例如数据库连接信息。
- 加载映射文件。映射文件即SQL映射文件，该文件中配置了操作数据库的SQL语句，需要在MyBatis配置文件mybatis-config.xml中加载。mybatis-config.xml文件可以加载多个映射文件，每个文件对应数据库中的一张表。
- 构造会话工厂：通过MyBatis的环境等配置信息构建会话工厂SqlSessionFactory。
- 创建会话对象：由会话工厂创建SqlSession对象，该对象中包含了执行SQL语句的所有方法。
- Executor执行器：MyBatis底层定义了一个Executor 接口来操作数据库，它将根据SqlSession传递的参数动态地生成需要执行的SQL语句，同时负责查询缓存的维护。
- MappedStatement 对象：在Executor接口的执行方法中有一个MappedStatement类型的参数，该参数是对映射信息的封装，用于存储要映射的SQL语句的id、参数等信息。
- 输入参数映射：输入参数类型可以是Map、List等集合类型，也可以是基本数据类型和POJO类型。输入参数映射过程类似于JDBC对PreparedStatement对象设置参数的过程。
- 输出结果映射：输出结果类型可以是Map、List等集合类型，也可以是基本数据类型和POJO类型。输出结果映射过程类似于JDBC对结果集的解析过程。

6. Mybatis都有哪些Executor执行器？它们之间的区别是什么？

Mybatis有三种基本的Executor执行器，`SimpleExecutor`、`ReuseExecutor`、`BatchExecutor`。

SimpleExecutor：每执行一次update或select，就开启一个Statement对象，用完立刻关闭Statement对象。

ReuseExecutor：执行update或select，以sql作为key查找Statement对象，存在就使用，不存在就创建，用完后，不关闭Statement对象，而是放置于Map<String, Statement>内，供下一次使用。简言之，就是重复使用Statement对象。

BatchExecutor：执行update（没有select，JDBC批处理不支持select），将所有sql都添加到批处理中（addBatch()），等待统一执行（executeBatch()），它缓存了多个Statement对象，每个Statement对象都是addBatch()完毕后，等待逐一执行executeBatch()批处理。与JDBC批处理相同。

作用范围：Executor的这些特点，都严格限制在SqlSession生命周期范围内。

7. MyBatis中接口绑定有几种实现方式？

1. 通过注解绑定，在接口的方法上面加上 @Select@Update等注解里面包含Sql语句来绑定（SQL语句比较简单的时候，推荐注解绑定）
2. 通过xml里面写SQL来绑定，指定xml映射文件里面的namespace必须为接口的全路径名（SQL语句比较复杂的时候，推荐xml绑定）

8. Mybatis 是如何进行分页的？

Mybatis 使用 RowBounds 对象进行分页，它是针对 ResultSet 结果集执行的内存分页，而非物理分页，先把数据都查出来，然后再做分页。

可以在 sql 内直接书写带有物理分页的参数来完成物理分页功能，也可以使用分页插件来完成物理分页。

9. 分页插件的基本原理是什么？

分页插件的基本原理是使用 Mybatis 提供的插件接口，实现自定义插件，在插件的拦截方法内拦截待执行的 sql，然后重写 sql（SQL 拼接 limit），根据 dialect 方言，添加对应的物理分页语句和物理分页参数，用到了技术 JDK 动态代理，用到了责任链设计模式。

10. 简述Mybatis的插件运行原理

Mybatis仅可以编写针对 `ParameterHandler`、`ResultSetHandler`、`StatementHandler`、`Executor` 这4种接口的插件，Mybatis使用JDK的动态代理，为需要拦截的接口生成代理对象以实现接口方法拦截功能，每当执行这4种接口对象的方法时，就会进入拦截方法，具体就是 `InvocationHandler` 的 `invoke()` 方法，当然，只会拦截那些你指定需要拦截的方法。

11. 如何编写一个插件？

编写插件：实现 Mybatis 的 `Interceptor` 接口并复写 `intercept()` 方法，然后再给插件编写注解，指定要拦截哪一个接口的哪些方法即可，最后在配置文件中配置你编写的插件。

12. Mybatis 是否支持延迟加载？

Mybatis 仅支持 association 关联对象和 collection 关联集合对象的延迟加载，association 指的就是一对一，collection 指的就是一对多查询。在 Mybatis 配置文件中，可以配置是否启用延迟加载 `lazyLoadingEnabled=true|false`。

13. 延迟加载的基本原理是什么？

延迟加载的基本原理是，使用 CGLIB 创建目标对象的代理对象，当调用目标方法时，进入拦截器方法。

比如调用 `a.getB().getName()`，拦截器 `invoke()` 方法发现 `a.getB()` 是 null 值，那么就会单独发送事先保存好的查询关联 B 对象的 sql，把 B 查询上来，然后调用 `a.setB(b)`，于是 a 的对象 b 属性就有值了，接着完成 `a.getB().getName()` 方法的调用。

当然了，不光是 Mybatis，几乎所有的包括 Hibernate，支持延迟加载的原理都是一样的。

14. #{}和\${}的区别是什么？

`#{}` 被解析成预编译语句，预编译之后可以直接执行，不需要重新编译sql。

```
//sqlMap 中如下的 sql 语句
select * from user where name = #{name};
//解析成为预编译语句；编译好SQL语句再取值
select * from user where name = ?;
```

`${}` 仅仅为一个字符串替换，每次执行sql之前需要进行编译，存在 sql 注入问题。

```
select * from user where name = '${name}'
//传递的参数为 "ruhua" 时，解析为如下，然后发送数据库服务器进行编译。取值以后再去编译SQL语句。
select * from user where name = "ruhua";
```

15. Mybatis的预编译

数据库接受到sql语句之后，需要词法和语义解析，优化sql语句，制定执行计划。这需要花费一些时间。如果一条sql语句需要反复执行，每次都进行语法检查和优化，会浪费很多时间。预编译语句就是将sql语句中的 `值用占位符替代`，即将 `sql语句模板化`。一次编译、多次运行，省去了解析优化等过程。

mybatis是通过 `PreparedStatement` 和占位符来实现预编译的。

mybatis底层使用 `PreparedStatement`，默认情况下，将对所有的 sql 进行预编译，将#{ }替换为?，然后将带有占位符?的sql模板发送至mysql服务器，由服务器对此无参数的sql进行编译后，将编译结果缓存，然后直接执行带有真实参数的sql。

预编译的作用：

1. 预编译阶段可以优化 sql 的执行。预编译之后的 sql 多数情况下可以直接执行，数据库服务器不需要再次编译，可以提升性能。
2. 预编译语句对象可以重复利用。把一个 sql 预编译后产生的 `PreparedStatement` 对象缓存下来，下次对于同一个sql，可以直接使用这个缓存的 `PreparedStatement` 对象。
3. 防止SQL注入。使用预编译，而其后注入的参数将 **不会再进行SQL编译**。也就是说其后注入进来的参数系统不会认为它会是一条SQL语句，而默认其是一个参数。

16. 一级缓存和二级缓存

缓存：合理使用缓存是优化中最常见的方法之一，将从数据库中查询出来的数据放入缓存中，下次使用时不必从数据库查询，而是直接从缓存中读取，避免频繁操作数据库，减轻数据库的压力，同时提高系统性能。

一级缓存是SqlSession级别的缓存：Mybatis对缓存提供支持，默认情况下只开启一级缓存，一级缓存作用范围为同一个SqlSession。在SQL和参数相同的情况下，我们使用同一个SqlSession对象调用同一个Mapper方法，往往只会执行一次SQL。因为在使用SqlSession第一次查询后，Mybatis会将结果放到缓存中，以后再次查询时，如果没有声明需要刷新，并且缓存没超时的情况下，SqlSession只会取出当前缓存的数据，不会再次发送SQL到数据库。若使用不同的SqlSession，因为不同的SqlSession是相互隔离的，不会使用一级缓存。

二级缓存是mapper级别的缓存：可以使缓存在各个SqlSession之间共享。二级缓存默认不开启，需要在mybatis-config.xml开启二级缓存：

```
<!-- 通知 MyBatis 框架开启二级缓存 -->
<settings>
  <setting name="cacheEnabled" value="true"/>
</settings>
```

并在相应的Mapper.xml文件添加cache标签，表示对哪个mapper 开启缓存：

```
<cache/>
```

二级缓存要求返回的POJO必须是可序列化的，即要求实现Serializable接口。

当开启二级缓存后，数据的查询执行的流程就是 二级缓存 -> 一级缓存 -> 数据库。



扫码关注我



微信搜索



程序员大彬

公众号后台回复【面试】获取面试手册
PDF最新版

微服务

1. 什么是微服务？

微服务是将一个原本独立的系统拆分成多个小型服务，这些小型服务都在各自独立的进程中运行，服务和 service 之间采用轻量级的通信机制进行协作。每个服务可以被独立的部署到生产环境。

[从单体应用到微服务](#)

单体系统的缺点：

1. 修改一个小功能，就需要将整个系统重新部署上线，影响其他功能的运行；
2. 功能模块互相依赖，强耦合，扩展困难。如果出现性能瓶颈，需要对整体应用进行升级，虽然影响性能的可能只是其中一个小模块；

单体系统的优点：

1. 容易部署，程序单一，不存在分布式集群的复杂部署环境；
2. 容易测试，没有复杂的服务调用关系。

微服务的**优点**：

1. 不同的服务可以使用不同的技术；
2. 隔离性。一个服务不可用不会导致其他服务不可用；
3. 可扩展性。某个服务出现性能瓶颈，只需对此服务进行升级即可；
4. 简化部署。服务的部署是独立的，哪个服务出现问题，只需对此服务进行修改重新部署；

微服务的**缺点**：

1. 网络调用频繁。性能相对函数调用较差。
2. 运维成本增加。系统由多个独立运行的微服务构成，需要设计一个良好的监控系统对各个微服务的运行状态进行监控。

2. 分布式和微服务的区别

从概念理解，分布式服务架构强调的是服务化以及服务的**分散化**，微服务则更强调服务的**专业化和精细分工**；

从实践的角度来看，**微服务架构通常是分布式服务架构**，反之则未必成立。

一句话概括：分布式：分散部署；微服务：分散能力。

3. 服务怎么划分？

横向拆分：按照不同的业务域进行拆分，例如订单、营销、风控、积分资源等。形成独立的业务领域微服务集群。

纵向拆分：把一个业务功能里的不同模块或者组件进行拆分。例如把公共组件拆分成独立的原子服务，下沉到底层，形成相对独立的原子服务层。这样一纵一横，就可以实现业务的服务化拆分。

要做好微服务的分层：梳理和抽取核心应用、公共应用，作为独立的服务下沉到核心和公共能力层，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求

总之，微服务的设计一定要 **渐进式** 的，总的原则是 **服务内部高内聚，服务之间低耦合**。

4. 微服务设计原则

单一职责原则

意思是每个微服务只需要实现自己的业务逻辑就可以了，比如订单管理模块，它只需要处理订单的业务逻辑就可以了，其它的不必考虑。

服务自治原则

意思是每个微服务从开发、测试、运维等都是独立的，包括存储的数据库也都是独立的，自己就有一套完整的流程，我们完全可以把它当成一个项目来对待。不必依赖于其它模块。

轻量级通信原则

首先是通信的语言非常的轻量，第二，该通信方式需要是跨语言、跨平台的，之所以要跨平台、跨语言就是为了让每个微服务都有足够的独立性，可以不受技术的钳制。

接口明确原则

由于微服务之间可能存在着调用关系，为了尽量避免以后由于某个微服务的接口变化而导致其它微服务都做调整，在设计之初就要考虑到所有情况，让接口尽量做的更通用，更灵活，从而尽量避免其它模块也做调整。

5. 微服务之间是如何通讯的？

1、RPC

优点：简单，常见。因为没有中间件代理，系统更简单

缺点：

1. 只支持请求/响应的模式，不支持别的，比如通知、发布/订阅
2. 降低了可用性，因为客户端和服务端在请求过程中必须都是可用的

2、消息队列

除了标准的基于RPC通信的微服务架构，还有基于消息队列通信的微服务架构，这种架构下的微服务采用发送消息（Publish Message）与监听消息（Subscribe Message）的方式来实现彼此之间的交互。

网易的蜂巢平台就采用了基于消息队列的微服务架构设计思路，如下图所示，微服务之间通过RabbitMQ传递消息，实现通信。

与上面几种微服务架构相比，基于消息队列的微服务架构并不多，案例也相对较少，更多地体现为一种与业务相关的设计经验，各家有各家的实现方式，缺乏公认的设计思路与参考架构，也没有形成一个知名的开源平台。因此，如果需要实施这种微服务架构，则基本上需要项目组自己从零开始去设计实现一个微服务架构基础平台，其代价是成本高、风险大。

优点：

- 把客户端和服务端解耦，更松耦合提高可用性，因为消息中间件缓存了消息，直到消费者可以消费
- 支持很多通信机制比如通知、发布/订阅等

缺点：

- 缺乏公认的设计思路与参考架构，也没有形成一个知名的开源平台
- 成本高、风险大

6. 熔断器

雪崩效应：假如存在这样的调用链路，a服务->b服务->c服务，当c服务挂了的时候，b服务调用c服务会等待超时，a服务调用b服务也会等待超时，调用方长时间等不到相应而占用线程，如果有大量的请求过来，就会造成线程池打满，导致整个链路的服务奔溃。

为了解决分布式系统的雪崩效应，分布式系统引进了**熔断器机制**。

当一个服务的处理用户请求的失败次数在一定时间内小于设定的阈值时，熔断器处于关闭状态，服务正常。

当服务处理用户请求失败次数在一定时间内大于设定的阈值时，说明服务出现故障，打开熔断器，这时所有的请求会快速返回失败的错误信息，不执行业务逻辑，从而防止故障的蔓延。

当处于打开状态的熔断器时，一段时间后处于半打开状态，并执行一定数量的请求，剩余的请求会执行快速失败，若执行请求失败了，则继续打开熔断器，若成功了，则将熔断器关闭。

7. 服务网关

7.1. 何为网关？

通俗一点的讲：网关就是要去别的网络的时候，把报文首先发送到的那台设备。稍微专业一点的术语，网关就是当前主机的默认路由。

网关一般就是一台路由器，有点像“一个小区中的一个邮局”，小区里面的住户互相是知道怎么走，但是要向外地投递东西就不知道了，怎么办？把地址写好送到本小区的邮局就好了。

那么，怎么区分是“本小区”和“外地小区”的呢？根据IP地址 + 掩码。如果是在一个范围内的，就是本小区（局域网内部），如果掩不住的，就是外地的。

例如，你的机器的IP地址是：192.168.0.2/24，网关是192.168.0.1

如果机器访问的IP地址范围是：192.168.0.1~192.168.0.254的，说明是一个小区的邻居，你的机器就直接发送了（和网关没关系）。如果你访问的IP地址不是这个范围的，则就投递到192.168.0.1上，让这台设备来转发。

参考：<https://www.zhihu.com/question/362842680/answer/951412213>

7.2. 何为API网关

假设你正在开发一个电商网站，那么这里会涉及到很多后端的微服务，比如会员、商品、推荐服务等。

那么这里就会遇到一个问题，APP/Browser怎么去访问这些后端的服务？如果业务比较简单的话，可以给每个业务都分配一个独立的域名(<https://service.api.company.com>)，但这种方式会有几个问题：

- 每个业务都会需要鉴权、限流、权限校验等逻辑，如果每个业务都各自为战，自己造轮子实现一遍，会很蛋疼，完全可以抽出来，放到一个统一的地方去做。
- 如果业务量比较简单的话，这种方式前期不会有什么问题，但随着业务越来越复杂，比如淘宝、亚马逊打开一个页面可能会涉及到数百个微服务协同工作，如果每一个微服务都分配一个域名的话，一方面客户端代码会很难维护，涉及到数百个域名
- 每上线一个新的服务，都需要运维参与，申请域名、配置Nginx等，当上线、下线服务器时，同样也需要运维参与，另外采用域名这种方式，对于环境的隔离也不太友好，调用者需要自己根据域名自己进行判断。
- 另外还有一个问题，后端每个微服务可能采用了不同的协议，比如HTTP、AMQP、自定义TCP协议等，但是你不可能要求客户端去适配这么多种协议，这是一项非常有挑战的工作，项目会变的非常复杂且很难维护。

更好的方式是采用API网关（也叫做服务网关），实现一个API网关**接管所有的入口流量**，类似Nginx的作用，将所有用户的请求转发给后端的服务器，但网关做的不仅仅只是简单的转发，也会针对流量做一些扩展，比如鉴权、限流、权限、熔断、协议转换、错误码统一、缓存、日志、监控、告警等，这样将通用的逻辑抽出来，由网关统一去做，业务方也能够更专注于业务逻辑，提升迭代的效率。

通过引入API网关，客户端只需要与API网关交互，而不用与各个业务方的接口分别通讯，但多引入一个组件就多引入了一个潜在的故障点，因此要实现一个高性能、稳定的网关，也会涉及到很多点。

网关层通常以集群的形式存在。并在服务网关层前通常会加上Nginx 用来负载均衡。

服务网关基本功能：

- 智能路由：接收**外部**一切请求，并转发到后端的对外服务。注意：我们只转发外部请求，服务之间的请求不走网关，这就表示全链路追踪、内部服务API监控、内部服务之间调用的容错、智能路由不能在网关完成；当然，也可以将所有的服务调用都走网关，那么几乎所有的功能都可以集成到网关中，但是这样的话，网关的压力会很大，不堪重负。
- 权限校验：网关可以做一些用户身份认证，权限认证，防止非法请求操作API 接口，对内部服务起到保护作用
- API监控：监控经过网关的请求，以及网关本身的一些性能指标（gc等）
- 限流：与监控配合，进行限流操作
- API日志统一收集：类似于一个aspect切面，记录接口的进入和出去时的相关日志

当然，网关实现这些功能，需要做高可用，否则网关很可能成为架构的瓶颈，最常用的网关组件Zuul、Nginx

8. 服务配置统一管理

在微服务架构中，需要有统一管理配置文件的组件，例如：SpringCloud Config组件、阿里的Diamond、百度的Disconf、携程的Apollo等

9. 服务链路追踪

在微服务架构中，必须实现分布式链路追踪，去跟进一个请求到底有哪些服务参与、参与顺序，是每个请求链路清晰可见，便于问题快速定位。

常用链路追踪组件有Google的Dapper、Twitter 的Zipkin，以及阿里Eagleeye。

10. 微服务框架

市面常用微服务框架有：Spring Cloud 、Dubbo 、kubernetes

- 从功能模块上考虑，Dubbo缺少很多功能模块，例如网关、链路追踪等
- 从学习成本上考虑，Dubbo 版本趋于稳定，稳定完善、可以即学即用，难度简单，Spring cloud 基于Spring Boot，需要先掌握Spring Boot，例外Spring cloud 大多为英文文档，要求学习者有一定的英文阅读能力
- 从开发风格考虑，Dubbo倾向于xml的配置方式，Spring cloud 基于Spring Boot，采用基于注解和JavaBean配置方式的敏捷开发
- 从开发速度上考虑，Spring cloud 具有更高的开发和部署速度
- 从通信方式上考虑，Spring cloud 基于HTTP Restful 风格，服务于服务之间完全无关、无耦合。Dubbo 基于远程调用，对接口、平台和语言有强依赖性，如果需要实现跨平台，需要有额外的中间件。

所以Dubbo专注于服务治理；Spring Cloud关注于微服务架构生态。

分布式事务

1. 简介

1.1. 事务

事务是应用程序中一系列严密的操作，所有操作必须成功完成，否则在每个操作中所作的所有更改都会被撤消。也就是事务具有原子性，一个事务中的一系列的操作要么全部成功，要么一个都不做。事务应该具有 4 个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为 ACID 特性。

1.2. 分布式事务

分布式事务是指事务的参与者，支持事务的服务器，资源服务器以及事务管理器分别位于分布式系统的不同节点之上。通常一个分布式事务中会涉及对多个数据源或业务系统的操作。分布式事务也可以被定义为一种嵌套型的事务，同时也就具有了ACID事务的特性。

1.3. 强一致性、弱一致性、最终一致性

强一致性

任何一次读都能读到某个数据的最近一次写的的数据。系统中的所有进程，看到的操作顺序，都和全局时钟下的顺序一致。简言之，在任意时刻，所有节点中的数据是一样的。

弱一致性

数据更新后，如果能容忍后续的访问只能访问到部分或者全部访问不到，则是弱一致性。

最终一致性

不保证在任意时刻任意节点上的同一份数据都是相同的，但是随着时间的迁移，不同节点上的同一份数据总是在向趋同的方向变化。简单说，就是在一段时间后，节点间的数据会最终达到一致状态。

由于分布式事务方案，无法做到完全的ACID的保证，没有一种完美的方案，能够解决掉所有业务问题。因此在实际应用中，会根据业务的不同特性，选择最适合的分布式事务方案。

2. 分布式事务的基础

2.1. CAP理论

Consistency（一致性）：数据一致更新，所有数据变动都是同步的（强一致性）。

Availability（可用性）：好的响应性能。

Partition tolerance（分区容错性）：可靠性。

定理：任何分布式系统**只可同时满足二点**，没法三者兼顾。

CA系统（放弃P）：指将所有数据（或者仅仅是那些与事务相关的数据）都放在一个分布式节点上，就不会存在网络分区。所以强一致性以及可用性得到满足。

CP系统（放弃A）：如果要求数据在各个服务器上**是强一致的**，然而网络分区会导致同步时间无限延长，那么如此一来可用性就得不到保障了。坚持事务ACID（原子性、一致性、隔离性和持久性）的传统数据库以及对结果一致性非常敏感的应用通常会做出这样的选择。

AP系统（放弃C）：这里所说的放弃一致性，并不是完全放弃数据一致性，而是放弃数据的强一致性，而保留数据的最终一致性。如果即要求系统高可用又要求分区容错，那么就要放弃一致性了。因为一旦发生网络分区，节点之间将无法通信，为了满足高可用，每个节点只能用本地数据提供服务，这样就会导致数据不一致。一些遵守BASE原则数据库，（如：Cassandra、CouchDB等）往往会放宽对一致性的要求（满足最终一致性即可），一次来获取基本的可用性。

2.2. BASE理论

BASE 是 Basically Available(基本可用)、Soft state(软状态)和 Eventually consistent (最终一致性)三个短语的缩写。是对CAP中AP的一个扩展。

1. 基本可用:分布式系统在出现故障时，允许损失部分可用功能，保证核心功能可用。
2. 软状态:允许系统中存在中间状态，这个状态不影响系统可用性，这里指的是CAP中的不一致。
3. 最终一致:最终一致是指经过一段时间后，所有节点数据都将会达到一致。

BASE解决了CAP中理论没有网络延迟，在BASE中用软状态和最终一致，保证了延迟后的一致性。BASE和 ACID 是相反的，它完全不同于ACID的强一致性模型，而是通过牺牲强一致性来获得可用性，并允许数据在一段时间内是不一致的，但最终达到一致状态。

3. 分布式事务解决方案

分布式事务的实现主要有以下 6 种方案：

- 2PC 方案
- TCC 方案
- 本地消息表
- MQ事务
- Saga事务
- 最大努力通知方案

3.1. 2PC方案

2PC方案分为两阶段：

第一阶段：事务管理器要求每个涉及到事务的数据库预提交(precommit)此操作，并反映是否可以提交。

第二阶段：事务协调器要求每个数据库提交数据，或者回滚数据。

优点： 尽量保证了数据的强一致，实现成本较低，在各大主流数据库都有自己实现，对于MySQL是从5.5开始支持。

缺点：

- 单点问题:事务管理器在整个流程中扮演的角色很关键，如果其宕机，比如在第一阶段已经完成，在第二阶段正准备提交的时候事务管理器宕机，资源管理器就会一直阻塞，导致数据库无法使用。
- 同步阻塞:在准备就绪之后，资源管理器中的资源一直处于阻塞，直到提交完成，释放资源。
- 数据不一致:两阶段提交协议虽然为分布式数据强一致性所设计，但仍然存在数据不一致性的可能，比如在第二阶段中，假设协调者发出了事务commit的通知，但是因为网络问题该通知仅被一部分参与者所收到并执行了commit操作，其余的参与者则因为没有收到通知一直处于阻塞状态，这时候就产生了数据的不一致性。

总的来说，2PC方案比较简单，成本较低，但是其单点问题，以及不能支持高并发（由于同步阻塞）依然是其最大的弱点。

3.2. TCC

TCC 的全称是：Try、Confirm、Cancel。

- **Try 阶段**：这个阶段说的是对各个服务的资源做检测以及对资源进行 **锁定或者预留**。
- **Confirm 阶段**：这个阶段说的是在各个服务中执行实际的操作。
- **Cancel 阶段**：如果任何一个服务的业务方法执行出错，那么这里就需要 **进行补偿**，就是执行已经执行成功的业务逻辑的回滚操作。（把那些执行成功的回滚）
-

举个简单的例子如果你用100元买了一瓶水，Try阶段:你需要向你的钱包检查是否够100元并锁住这100元，水也是一样的。

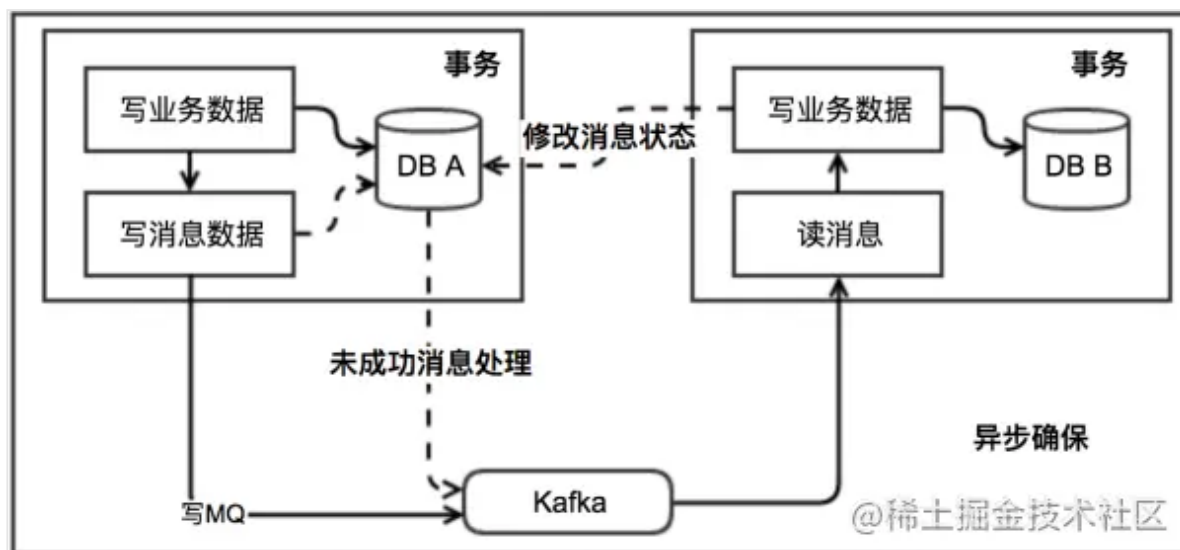
如果有一个失败，则进行cancel(释放这100元和这一瓶水)，如果cancel失败不论什么失败都进行重试cancel，所以需要保持幂等。

如果都成功，则进行confirm,确认这100元扣，和这一瓶水被卖，如果confirm失败无论什么失败则重试(会依靠活动日志进行重试)。

这种方案说实话几乎很少人使用，但是也有使用的场景。因为这个**事务回滚实际上是严重依赖于你自己写代码来回滚和补偿了**，会造成补偿代码巨大。

3.3. 本地消息表

本地消息表的核心是将需要分布式处理的任务通过消息日志的方式来异步执行。消息日志可以存储到本地文本、数据库或消息队列，再通过业务规则自动或人工发起重试。人工重试更多的是应用于支付场景，通过对账系统对事后问题的处理。



对于本地消息队列来说核心是把大事务转变为小事务。还是举上面用100元去买一瓶水的例子。

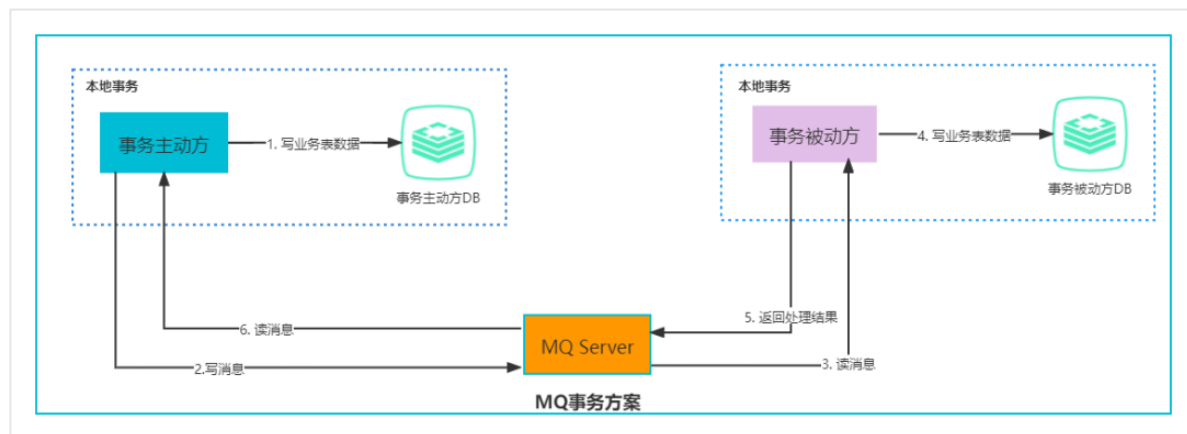
- 1.当你扣钱的时候，你需要在你扣钱的服务器上新增加一个本地消息表，你需要把你扣钱和写入减去水的库存到本地消息表放入同一个事务(依靠数据库本地事务保证一致性)。
- 2.这个时候有个定时任务去轮询这个本地事务表，把没有发送的消息，扔给商品库存服务器，叫他减去水的库存，到达商品服务器之后这个时候得先写入这个服务器的事务表，然后进行扣减，扣减成功后，更新事务表中的状态。
- 3.商品服务器通过定时任务扫描消息表或者直接通知扣钱服务器，扣钱服务器本地消息表进行状态更新。
- 4.针对一些异常情况，定时扫描未成功处理的消息，进行重新发送，在商品服务器接到消息之后，首先判断是否是重复的，如果已经接收，在判断是否执行，如果执行在马上又进行通知事务，如果未执行，需要重新执行需要由业务保证幂等，也就是不会多扣一瓶水。

本地消息队列是BASE理论，是最终一致模型，适用于对一致性要求不高的。实现这个模型时需要注意重试的幂等。

3.4. MQ事务

基于 MQ 的分布式事务方案其实是对本地消息表的封装，将本地消息表基于 MQ 内部，其他方面的协议基本与本地消息表一致。

MQ事务方案整体流程和本地消息表的流程很相似，如下图：

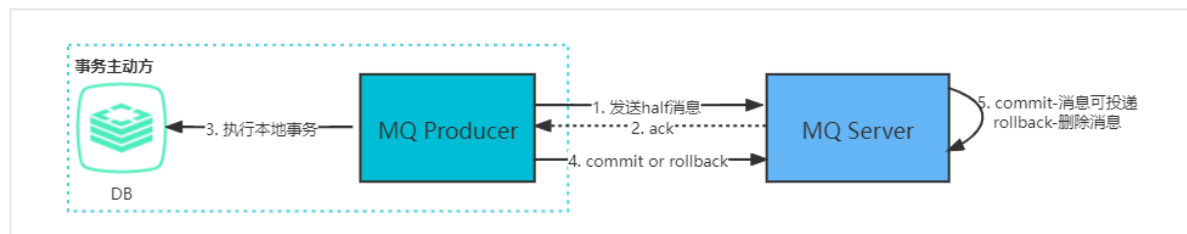


从上图可以看出和本地消息表方案唯一不同就是将本地消息表存在了MQ内部，而不是业务数据库中。

那么MQ内部的处理尤为重要，下面主要基于 RocketMQ 4.3 之后的版本介绍 MQ 的分布式事务方案。

在本地消息表方案中，保证事务主动方发写业务表数据和写消息表数据的一致性是基于数据库事务，RocketMQ 的事务消息相对于普通 MQ提供了 2PC 的提交接口，方案如下：

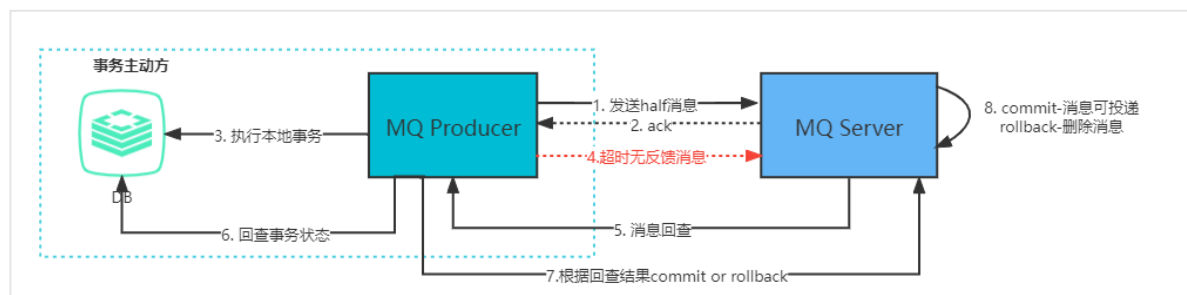
正常情况：事务主动方发消息



这种情况下，事务主动方服务正常，没有发生故障，发消息流程如下：

- 发送方向 MQ 服务端(MQ Server)发送 half 消息。
- MQ Server 将消息持久化成功之后，向发送方 ack 确认消息已经发送成功。
- 发送方开始执行本地事务逻辑。
- 发送方根据本地事务执行结果向 MQ Server 提交二次确认（commit 或是 rollback）。
- MQ Server 收到 commit 状态则将半消息标记为可投递，订阅方最终将收到该消息；MQ Server 收到 rollback 状态则删除半消息，订阅方将不会接受该消息。

异常情况：事务主动方消息恢复



在断网或者应用重启等异常情况下，图中 4 提交的二次确认超时未到达 MQ Server，此时处理逻辑如下：

- MQ Server 对该消息发起消息回查。
- 发送方收到消息回查后，需要检查对应消息的本地事务执行的最终结果。
- 发送方根据检查得到的本地事务的最终状态再次提交二次确认。
- MQ Server 基于 commit/rollback 对消息进行投递或者删除。

优点

相比本地消息表方案，MQ 事务方案优点是：

- 消息数据独立存储，降低业务系统与消息系统之间的耦合。
- 吞吐量大于使用本地消息表方案。

缺点

- 一次消息发送需要两次网络请求(half 消息 + commit/rollback 消息)。
- 业务处理服务需要实现消息状态回查接口。

3.5. Saga事务

Saga是由一系列的本地事务构成。每一个本地事务在更新完数据库之后，会发布一条消息或者一个事件来触发Saga中的下一个本地事务的执行。如果一个本地事务因为某些业务规则无法满足而失败，Saga会执行在这个失败的事务之前成功提交的所有事务的补偿操作。

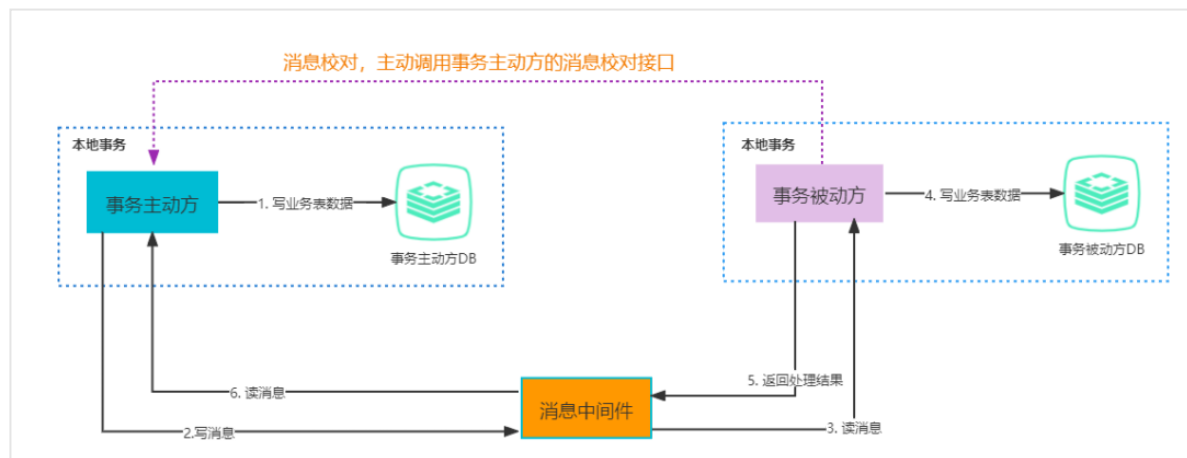
Saga的实现有很多种方式，其中最流行的两种方式是：

- **基于事件的方式。**这种方式没有协调中心，整个模式的工作方式就像舞蹈一样，各个舞蹈演员按照预先编排的动作和走位各自表演，最终形成一只舞蹈。处于当前Saga下的各个服务，会产生某类事件，或者监听其它服务产生的事件并决定是否需要对监听到的事件做出响应。
- **基于命令的方式。**这种方式的工作形式就像一只乐队，由一个指挥家（协调中心）来协调大家的工作。协调中心来告诉Saga的参与方应该执行哪一个本地事务。

3.6. 最大努力通知方案

最大努力通知也称为定期校对，是对MQ事务方案的进一步优化。它在事务主动方增加了消息校对的接口，如果事务被动方没有接收到消息，此时可以调用事务主动方提供的消息校对的接口主动获取。

最大努力通知的整体流程如下图：



在可靠消息事务中，事务主动方需要将消息发送出去，并且消息接收方成功接收，这种可靠性发送是由事务主动方保证的；

但是最大努力通知，事务主动方尽最大努力（重试，轮询....）将事务发送给事务接收方，但是仍然存在消息接收不到，此时需要事务被动方主动调用事务主动方的消息校对接口查询业务消息并消费，这种通知的可靠性是由事务被动方保证的。

最大努力通知适用于业务通知类型，例如微信交易的结果，就是通过最大努力通知方式通知各个商户，既有回调通知，也有交易查询接口。

4. 参考文章

<https://www.pdai.tech/md/arch/arch-z-transection.html>

<https://juejin.cn/post/6844903647197806605#heading-15>

场景设计

系统设计高频面试题是我的[知识星球](#)内部专属资料，已经整理到Java面试手册完整版。

The image shows a web browser window with two main panels. The left panel displays a document titled "Java面试手册" (Java Interview Manual) with a table of contents. The right panel shows a document editor interface with a toolbar and a text area.

Table of Contents (Left Panel):

- Java面试手册
- 面试准备
- 面试题&重要知识点
- 系统设计
 - 扫码登录原理
 - 订单30分钟未支付自动取消怎么实现?
 - 如何设计一个短链系统?**
 - 秒杀系统
 - 微信红包后台系统设计
 - 单点登录 (SSO) 的设计与实现
 - 如何设计一个优惠券系统?
- 海量数据场景高频面试题
- 面经篇
- 校招分享

Document Editor (Right Panel):

The editor has a toolbar with various icons for text formatting and editing. The text area contains the following content:

访问 B

网址 B 服务器

短链生成方式

据统计, 目前全球有 58 亿的网页, Java 中 int 取值最多是 $2^{32} = 4294967296 < 43 \text{ 亿} < 58 \text{ 亿}$, long 取值是 $2^{64} > 58 \text{ 亿}$. 所以如果是用数字的话, int 勉强能够支撑 (毕竟不是所有网站都会调用短链服务创建短链), 使用 long 就比较保险, 但会造成空间浪费。

新浪微博使用 8 位字符串表示原始链接, 这种字符串可以理解成数字的 62 进制表示, $62^8 = 3521614606208 > 3521 \text{ 亿} > 58 \text{ 亿}$, 也就是可以解决目前全球已知的网址。62 进制就是由 10 个数字 + (a-z)26 个小写字母 + (A-Z)26 个大写字母组成的数。

哈希算法

对原始链接取 Hash 值, 是一种比较简单的思路。有很多现成的算法可以实现, 但是有个避不开的问题就是: Hash 碰撞, 所以选一个碰撞率低的算法比较重要。

推荐MurmurHash 算法, 这种算法是一种非加密型哈希函数, 适用于一般的哈希检索操作, 目前 Redis, Memcached, Cassandra, HBase, Lucene 都在用这种算法。

对于碰撞问题, 最简单的一种思路是, 如果发生碰撞, 就给原始 URL 附上特

👉 优质学习资源汇总:

- 1、阿里巴巴技术团队的实战经验总结, 链接: <https://t.zsxq.com/08TUWuPR9>
- 2、Tomcat架构解析、性能优化、源码解读: <https://t.zsxq.com/09XZgyC7z>
- 3、尚学堂2022版全套的java学习视频 (300集), 链接: <https://t.zsxq.com/08NfkBCup>
- 4、并发编程视频教程, 包含: 线程基础、线程协作、并发工具类、锁、AQS、线程池、实战项目等: <https://t.zsxq.com/09vF2onhe>
- 5、LeetCode刷题笔记 (包含C/C++、Java、Go三个版本), 链接: <https://t.zsxq.com/080mxtafG>
- 6、尚硅谷Java学科全套教程 (总207.77GB), 链接: <https://t.zsxq.com/0807PC4s4>
- 7、自动动手写Java虚拟机PDF, 链接: <https://t.zsxq.com/08jBRPI5a>
- 8、字节大佬总结的Java面试资料, 涵盖Java开发常见的知识点, 链接: <https://t.zsxq.com/08UxPuP34>
- 9、阿里的算法笔试题精选合集, 涵盖70+算法题目、大厂笔试考点, 链接: <https://t.zsxq.com/082R1dqFF>
- 10、Java工程师进阶知识大全, 包含高并发架构、分布式架构, 链接: <https://t.zsxq.com/08j7WWUZn>
- 11、MySQL优质教程, 包含: 6 大范式讲解、7 大日志剖析、7 大 SQL 性能分析工具等内容, 链接: <https://t.zsxq.com/08d6j44JK>

- 12、阿里巴巴微服务治理白皮书，高效构建完整的微服务治理体系。链接：<https://t.zsxq.com/09fkXNKKJ>
- 13、RocketMQ 从入门到实战。链接：<https://t.zsxq.com/09MecUORI>
- 14、阿里巴巴性能调优实战手册，包含Java编程性能调优、多线程性能调优、JVM性能监测及调优、设计模式调优、数据库性能调优、实战演练六大模块。链接：<https://t.zsxq.com/09gdKMpY8>
- 15、Java面试真题，跟面试手册内容互补，建议准备面试的手两份一起过一遍！链接：<https://t.zsxq.com/09rVOpWNh>
- 16、畅购商城视频教程（含源码）。链接：<https://t.zsxq.com/09LOcfEDB>
- 17、面试题合集，包含技术和非技术问题，如Dubbo面试题、设计模式面试题、Netty面试题、Tomcat面试题、Nginx面试题、HR面试常见问题、英语面试题等。链接：<https://t.zsxq.com/09G34Sf16>
- 18、精选1000套简历模板，超实用！链接：<https://t.zsxq.com/096963tQt>
- 19、马士兵老师的Java基础视频教程：<https://t.zsxq.com/09twuX7b5>
- 20、MySQL45讲专栏：<https://t.zsxq.com/09V2U7K2A>
- 21、分布式优惠券系统：<https://t.zsxq.com/09ktzHUOZ>
- 22、mpvue课程小程序全栈开发：<https://t.zsxq.com/09aexZzl8>
- 22、微信小程序入门与实战 常用组件API开发技巧项目实战：<https://t.zsxq.com/09aexZzl8>
- 23、尚硅谷最新版尚筹网Java项目实战
<https://t.zsxq.com/094UkGCUy>

- 24、Redis优质教程：<https://t.zsxq.com/090gAzwEx>
- 25、JVM调优&多线程的教程：<https://t.zsxq.com/09u1h71vm>
- 26、Java架构师-技术专家的资料：<https://t.zsxq.com/090oV3kz2>
- 27、SSO单点登录原理：<https://t.zsxq.com/096U2VUJB>
- 28、3天全面深入学习zookeeper（含源码资料）<https://t.zsxq.com/09tijiPEu>
- 29、SpringBoot+SpringCloud全彩指南（终极版），链接：<https://t.zsxq.com/08DEQYliY>
- 30、Mybatis视频教程：<https://t.zsxq.com/09r81SkuK>
- 31、《漫画Java》PDF版本，链接：<https://t.zsxq.com/09L69kPDO>
- 32、Nginx优质视频教程：<https://t.zsxq.com/09gQBCuuF>

专属一对一的提问答疑，帮你解答各种疑难问题，包括自学Java路线、职业规划、面试问题等等。大彬会优先解答球友的问题。



大彬

2023-01-02 20:33

收进专栏

...

Alvin_Xu 提问： 本人是19年毕业至今工作了3年半左右，一直在同一个岗位上，近段时间因为公司项目组资金不足被调岗了，新岗位和自己的技术栈不太匹配，觉得如果一直变动会影响自己的职业发展。

这么些年来一直有点混吃等死，**想在现在开始寻求改变**，希望通过学习，让自己有所变化。不过对于学习方式以及学习路线**很是迷茫**，希望大彬老师可以帮忙解惑，非常感谢。

工作三年半，有点迷茫，想要寻求改变！

技术栈：

有一定的Java基础，对JVM原理，体系结构，垃圾回收机制等有部分了解

熟悉Scala&Spark开发，了解部署spark server项目及脚本

掌握spring，springCloud微服务架构开发，

了解Elastic search,Mule gateway, Feign等的使用

熟悉Jenkins, Kubernetes, PCF, G3等devops部署及调试

熟悉AppDynamic, patrol, LWM等程序监控平台的使用及配置

了解NexusIQ,Checkmarx,sonar等代码质量监控平台的使用及调优

熟悉sqlServer及Mongodb数据库

我一直都是认为，在现在这个就业环境下，“专才”的竞争力是要大于“全才”的，专注一个方向，对你的职业发展更为有利。

从你的技术栈来看，相对还是偏“杂”一些，Java、大数据、devops等都有涉及。而且现在还有调岗的可能，我建议



大彬

2023-01-10 21:41

收进专栏

...

🍌 提问： 想请教一下大彬，对于java开发和大数据开发的明年秋招情况的预测。java岗位多卷度也高，每家公司基本都有相关的职位，大数据开发的话培训班比岗位多，只有大公司和数据公司会有这类岗位。24年秋招的话**选择哪个方向会比较合适呢？**

岗位方向选择：Java还是大数据？

建议选大数据吧。就这几年校招来看，大数据岗位拿offer的难度相比Java还是比较小一些的。而且距离24年秋招还有一年多时间，转大数据完全来得及，而且有了Java基础，再来学大数据，应该会比较快入门。

再说下大数据和后端的差异。大数据门槛比Java高，除了熟悉数据库的操作之外，还要学习大数据整个生态，需要会分布式、数仓、数据分析统计等知识。因为大数据的学习门槛比Java高，所以市场上培训大数据的相比Java会少一些，竞争也相对小，没有Java那么卷。

另外，大数据薪资总体会比Java开发高一些（同一家公司同一级别，普通开发岗比大数据开发薪资会少一点），这也是大数据方向的一个优势。

不过呢，小点的公司，可能没有大数据的需求，毕竟业务量不大（小公司通常也不太建议去，坑多）。



2022-12-27 19:29

收进专栏

...

麻烦问一下，遇到一个面试题是：描述最能代表自己编程能力的代码，解释为什么这段代码提现了你的编程能力。这种题怎么答





2人赞过

举了两个例子，帮助球友解答疑惑



大彬：这种可以结合自己平时做的一些项目来讲。
比如说有一些比较复杂的功能，你是如何去实现的，像涉及很多状态流转的，使用了状态机去处理，通过"动作"来进行状态的变更。
或者项目中有很多相似的实现逻辑，把这些相似的逻辑抽象出来，做成公共组件，这也是一种编程能力的体现。
当然还有很多，上面只是两个简答的例子，具体可以结合你自身的项目去思考、挖掘，适当包装一下也是没问题的。
2022-12-27 21:43



大彬：可以从背景（为什么这么做）、实现原理（怎么实现的）、收获（学到了什么）这三个方面来回答
2022-12-27 21:52



回复 大彬：请问工作中如何应用的OOP原则怎么答啊
2022-12-28 13:07

大彬 回复



面向对象编程有六大原则：对象单一职责、里式替换原则、迪米特法则、开闭原则、依赖倒置、接口隔离。
以对象单一职责为例，在设计类的时候，必须职责明确，比如商品类，里面相关的属性和方法都必须跟商品相关，不能出现订单等不相关的内容。
其他原则也是类似，可以看看优质专栏-Java基础部分，有关于面向对象编六大原则的介绍
2022-12-28 14:21

另外星球还提供**简历指导、修改服务**，大彬已经帮**90+**个小伙伴修改了简历，相对还是比较有经验的。



求职意向: Java



生日: 1999.11

籍贯: 贵州三穗

电话: 1

学历: 本科学士

政治面貌: 团员

邮箱: 20

.com



教育背景 Education

2019.09-2023.06

天津工业大学

软件工程

主修课程: 高级程序语言设计、数据库原理及应用、数据结构、算法设计与分析、计算机组成原理、操作系统、计算机网络、编译原理、软件系统分析、软件质量测试等。

校园经历: 在校期间任大创信息部副部长, 负责创业就业信息收集整理, 微信公众号的维护和运营。在校期间注重实践, 参加技术学习型社团, 参与过许多小项目, 曾多次在软件实训中获取优异成绩。



项目经历 Project

2021.12-2022.03

企业网盘系统

项目组长、后端开发

项目简介: 基于 Springboot+Dubbo 开发的, 提供文件管理上传下载共享在线编辑的基础网盘。

● 基于 Redis 做分布式锁, 实现文件切块上传功能;

● WebSocket+Netty 做消息推送, 实现文件共享消息提示;

● Elasticsearch 做全文检索引擎, 实现大量文件的检索;

● HDFS 做分布式文件系统, 实现分布式文件存储。

2022.05-2023.07

微服务电商系统

后端开发

项目简介: SpringCloud 开发的电商系统, 集成了监控中心、网关等系统功能。

● 使用 RabbitMq 做消息队列, 实现延迟消息;

● 基于 SpringSecurity 做登录认证和权限控制;

● 使用 Spring 的 AOP, 添加切面实现接口日志记录;

● Docker 做应用容器引擎, Kubernetes 做应用容器管理平台;



专业技能 Expertise

1. 熟练掌握 java 语法, 和 IO、多线程、反射、异常及一些常用类, 掌握 JVM 原理;
2. 熟悉 Spring、SpringMVC、MyBatis、Springboot, 了解 SpringCloud 等开发框架;
3. 熟悉 MySQL 数据库, 具备一定的 SQL 优化能力; 熟悉 Redis 非关系型数据库;
4. 熟悉 HTML、CSS、JavaScript 等基础前端技术, 了解 Vue 等前端开发框架;
5. 熟练使用 Linux 操作系统; 熟悉常用的设计模式; 熟悉一些开发中常用中间件;
6. 具备一定的需求分析和系统设计能力, 较强的逻辑分析和独立解决问题能力。



自我评价 Evaluation

本人热爱 JAVA 开发工作, 注重实干, 熟悉 SpringBoot 开发, 具备良好的编码规范; 熟悉关系型数据库, 熟练使用 SQL 语言, 掌握基本的 SQL 优化; 服从安排、工作积极主动, 责任心与上进心较强, 具备良好的学习能力与合作的团队精神。

主修课程不用写这么多, 甚至可以不用写, 留点版面给其他内容

感觉似乎没有因果关系。可以描述清晰点

消息推送具体描述下怎么实现的

HDFS面试官应该会问, 思考下会问什么问题

延时消息怎么实现的

项目经历这块如果有数据支撑, 会更好, 比如一些慢查询优化, 从xx秒到xx毫秒等



18235 | 78386@qq.com
CSDN : https://blog.csdn.net/w_n_819295
求职意向：后端开发工程师 (北京)



实习经历

建议实习经历写实习项目，项目经历写自己的项目，一般都是这么写的

京东零售 - 市场营销与用户增长部 - 产品研发部

2021.12 - 2022.06

专业技能

1. 熟练运用 **JAVASE**、**JAVAAE** 主流技术；
2. 掌握 **Java** 基础，熟悉面向对象、集合、多线程、锁、I/O模型；
3. 对 **JVM** 原理有初步的了解，包括内存模型、垃圾回收机制以及垃圾回收算法；
4. 深入理解 **Redis** 数据类型和内部实现、熟悉其使用场景、持久化、淘汰策略；
5. 熟练使用 **MySQL**，熟悉MySQL索引、事务、锁机制、SQL调优并在项目中实际运用；
6. 熟悉 **MyBatis**、**Spring**、**SpringMVC**、**SpringBoot** 等主流开发框架；
7. 熟练使用**Kafka**，熟悉Kafka基本结构、消费者组；
8. 熟悉**Linux**操作系统，**Nginx** 等技术；
9. 了解常用的数据结构和算法；

专业技能可以补充分布式、微服务相关的（如果不熟悉建议了，面试大概率会问到）

项目经历

官旗OMS

工作描述，可以按照三个步骤展开：1.使用什么技术；2.做了什么事情；3.达成什么效果。

2021.12 - 2022.06

项目背景：京东在抖音开设官店为了方便管理订单并履约发货，需要在京东商城与多平台之间建立一个中枢。

工作描述：参与了V1.0、V1.1、V1.2迭代需求的开发。在 V1.0 需求中主要负责拉取抖音平台订单消息，对用户的创建订单、支付订单、风控订单等进行对接，解决了mq的重复消费。在 V1.1 需求中对用户的敏感信息进行了加密处理，使用了延迟消费减少了调用抖音接口的次数，并对SQL进行了优化提高了查询效率。在 V1.2 需求中使用多线程并发的拉取抖音订单详情中的信息，对其中的信息进行过滤筛选并归档到 TIDB 中。在日常工作中还负责将线上数据同步到 hive 数据仓库，实现日报的自动出数。

人脸签到

性能优化，如果有数据指标支撑，会更好。比如从xxx秒优化到xxx毫秒，查询时间减少95%等

2021.08 - 2021.09

- 1、该项目是一个前后端分离的项目，主要功能有授权模块、人脸比对、签到、签退、统计时长
- 2、功能的实现主要是基于SpringBoot + WebSocket + Swagger + Redis 技术栈实现，前后端采用JSON作为交互数据，采用了心跳机制保持了长连接，签到或签退会对已在线的用户进行实时广播，使用 Redis 缓存中间件对热数据进行缓存，并解决了缓存穿透的问题。使用Swagger对代码和注释进行开发文档的生成。

教育经历

项目经历这块，建议写上自己的收获

山西农业信息学院 - 数据科学与大数据技术 本科

2019.09 - 2023.06

荣誉奖项

2020年校级一等奖学金、2021年中国大学生计算机设计大赛三等奖

第十一届蓝桥杯大赛二等奖

社团和组织经历

双体系卓越人才教育基地软件部部长

2020.06 - 2021.06

积极组织学员学习软件编程类相关知识，对下一届学员进行学习进度的规划、以及定期检查学习情况，积极完成老师安排的任务、组织同学们参加各种计算机相关的比赛。

知识星球加入方式：



大彬 送你一张星球优惠券

「大彬的学习圈」

立减

¥ **50**

新人立减券

2023/12/12 13:00 后失效

 知识星球

长按扫码领取优惠 ▶



海量数据场景

1. 统计不同号码的个数

题目来自百度二面。

题目描述

已知某个文件内包含大量电话号码，每个号码为8位数字，如何统计不同号码的个数？

思路分析

这类题目其实是求解数据重复的问题。对于这类问题，可以使用**位图法**处理

8位电话号码可以表示的范围为00000000 ~ 99999999。如果用 bit表示一个号码，那么总共需要1亿个 bit，总共需要大约**10MB**的内存。

申请一个位图并初始化为0，然后遍历所有电话号码，把遍历到的电话号码对应的位图中的bit设置为1。当遍历完成后，如果bit值为1，则表示这个电话号码在文件中存在，否则这个bit对应的电话号码在文件中不存在。

最后这个**位图中bit值为1的数量**就是不同电话号码的个数了。

那么如何确定电话号码对应的是位图中的哪一位呢？

可以使用下面的方法来做**电话号码和位图的映射**。

```
00000000 对应位图最后一位：0x0000...000001。  
00000001 对应位图倒数第二位：0x0000...0000010（1 向左移 1 位）。  
00000002 对应位图倒数第三位：0x0000...0000100（1 向左移 2 位）。  
.....  
00000012 对应位图的倒数第十三位：0x0000...0001 0000 0000 0000（1 向左移 12 位）。
```

也就是说，电话号码就是1这个数字左移的次数。

具体实现

首先位图可以使用一个**int数组**来实现（在Java中int占用**4byte**）。

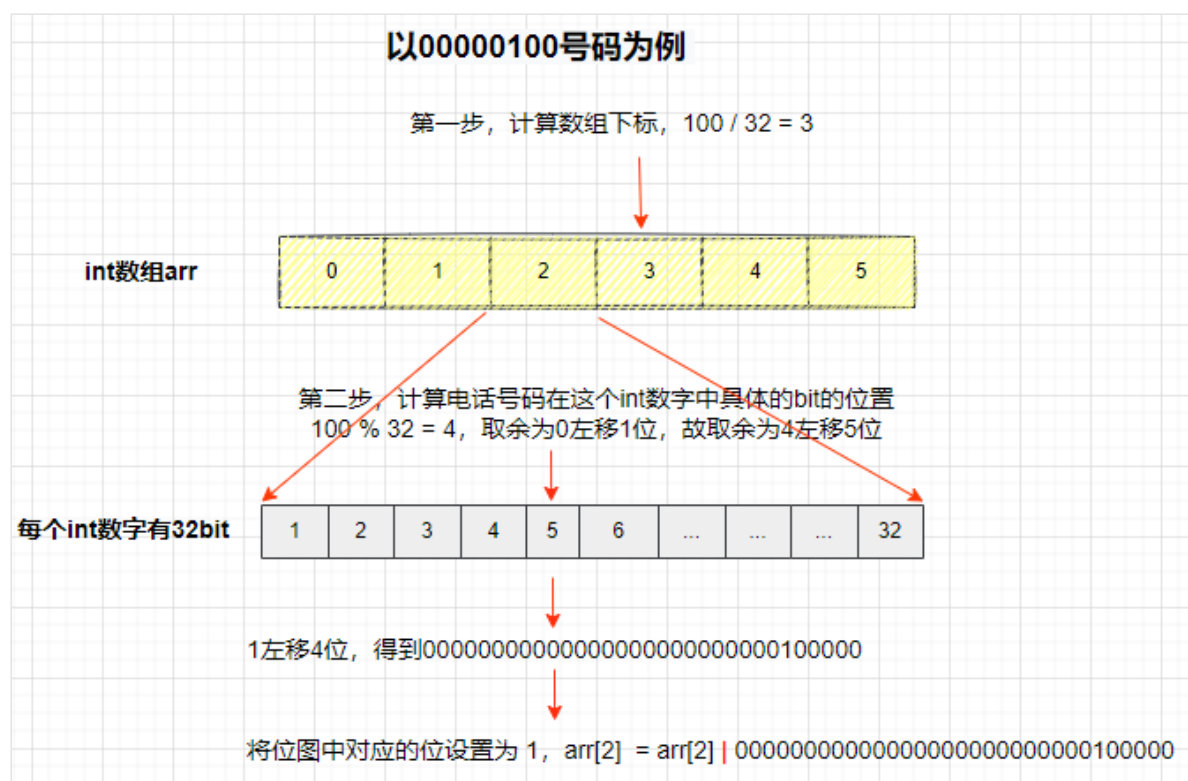
假设电话号码为 P，而通过电话号码获取位图中对应位置的方法为：

第一步，因为int整数占用 $4 \times 8 = 32$ bit，通过 $P/32$ 就可以计算出该电话号码在 bitmap 数组中的下标，从而可以确定它对应的 bit 在数组中的位置。

第二步，通过 $P\%32$ 就可以计算出这个电话号码在这个int数字中具体的bit的位置。只要把1向左移 $P\%32$ 位，然后把得到的值与这个数组中的值做或运算，就可以把这个电话号码在位图中对应的位设置为1。

以00000100号码为例。

1. 首先计算数组下标， $100 / 32 = 3$ ，得到数组下标位3。
2. 然后计算电话号码在这个int数字中具体的bit的位置， $100 \% 32 = 4$ 。取余为0左移1位，故取余为4左移5位，得到000...000010000
3. 将位图中对应的位设置为 1，即 $arr[3] = arr[3] | 000..00010000$ 。
4. 这就将电话号码映射到了位图的某一位了。



最后，统计位图中bit值为1的数量，便能得到不同电话号码的个数了。

2. 出现频率最高的100个词

题目描述

假如有一个**1G**大小的文件，文件里每一行是一个词，每个词的大小不超过**16byte**，要求返回出现频率最高的100个词。内存大小限制是**10M**

解法1

由于内存限制，我们无法直接将大文件的所有词一次性读到内存中。

可以采用**分治策略**，把一个大文件分解成多个小文件，保证每个文件的大小小于10M，进而直接将单个小文件读取到内存中进行处理。

第一步，首先遍历大文件，对遍历到的每个词x，执行 `hash(x) % 500`，将结果为i的词存放到文件f(i)中，遍历结束后，可以得到500个小文件，每个小文件的大小为2M左右；

第二步，接着统计每个小文件中出现频数最高的100个词。可以使用HashMap来实现，其中key为词，value为该词出现的频率。

对于遍历到的词x，如果在map中不存在，则执行 `map.put(x, 1)`。

若存在，则执行 `map.put(x, map.get(x)+1)`，将该词出现的次数加1。

第三步，在第二步中找出了每个文件出现频率最高的100个词之后，通过维护一个**小顶堆**来找出所有小文件中出现频率最高的100个词。

具体方法是，遍历第一个文件，把第一个文件中出现频率最高的100个词构建成为一个小顶堆。

如果第一个文件中词的个数小于100，可以继续遍历第二个文件，直到构建好有100个结点的小顶堆为止。

继续遍历其他小文件，如果遍历到的词的出现次数大于堆顶上词的出现次数，可以用新遍历到的词替换堆顶的词，然后重新调整这个堆为小顶堆。

当遍历完所有小文件后，这个小顶堆中的词就是出现频率最高的100个词。

总结一下，这种解法的主要思路如下：

1. 采用**分治**的思想，进行哈希取余
2. 使用**HashMap**统计每个小文件单词出现的次数
3. 使用**小顶堆**，遍历步骤2中的小文件，找出词频top100的单词

但是很容易可以发现问题，在第二步中，如果这个1G的大文件中有某个词词频过高，可能导致小文件大小超过10m。这种情况下该怎么处理呢？

接下来看另外一种解法。

解法2

第一步：使用多路归并排序对大文件进行排序，这样相同的单词肯定是紧挨着的

多路归并排序对大文件进行排序的步骤如下：

- ① 将文件按照顺序切分成大小不超过2m的小文件，总共500个小文件
- ② 使用10MB内存**分别**对 500 个小文件中的单词进行**排序**
- ③ 使用一个大小为500大小的堆，对500个小文件进行**多路排序**，结果写到一个大文件中

其中第三步，对500个小文件进行多路排序的思路如下：

- 初始化一个最小堆，大小就是有序小文件的个数500。堆中的每个节点存放每个有序小文件对应的输入流。
- 按照每个有序文件中的下一行数据对所有文件输入流进行排序，单词小的输入文件流放在堆顶。
- 拿出堆顶的输入流，并将其下一行数据写入到最终排序的文件中，如果拿出来输入流中还有数据的话，那么将这个输入流再一次添加到栈中。否则说明该文件输入流中没有数据了，那么可以关闭这个流。
- 循环这个过程，直到所有文件输入流都没有数据为止。

第二步：

- ① 初始化一个100个节点的**小顶堆**，用于保存100个出现频率最多的单词
- ② 遍历整个文件，一个单词一个单词的从文件中取出来，并计数
- ③ 等到遍历的单词和上一个单词不同的话，那么上一个单词及其频率如果大于堆顶的词的频率，那么放在堆中，否则不放

最终，小顶堆中就是出现频率前100的单词了。

解法2相对解法1，更加严谨，如果某个词词频过高或者整个文件都是同一个词的话，解法1不适用。

3. 查找两个大文件共同的URL

题目

给定 a、b 两个文件，各存放 50 亿个 URL，每个 URL 各占 64B，找出 a、b 两个文件共同的 URL。内存限制是 4G。

分析

每个 URL 占 64B，那么 50 亿个 URL 占用的空间大小约为 320GB。

$$5,000,000,000 * 64B \approx 320GB$$

由于内存大小只有 4G，因此，不可能一次性把所有 URL 加载到内存中处理。

可以采用分治策略，也就是把一个文件中的 URL 按照某个特征划分为多个小文件，使得每个小文件大小不超过 4G，这样就可以把这个小文件读到内存中进行了。

首先遍历文件a, 对遍历到的 URL 进行哈希取余 $\text{hash}(\text{URL}) \% 1000$, 根据计算结果把遍历到的 URL 存储到 a0, a1, a2, ..., a999, 这样每个大小约为 300MB。使用同样的方法遍历文件 b, 把文件 b 中的 URL 分别存储到文件 b0, b1, b2, ..., b999 中。这样处理过后, 所有可能相同的 URL 都在对应的小文件中, 即 a0 对应 b0, ..., a999 对应 b999, 不对应的小文件不可能有相同的 URL。那么接下来, 我们只需要求出这 1000 对小文件中相同的 URL 就好了。

接着遍历 a_i ($i \in [0, 999]$), 把 URL 存储到一个 HashSet 集合中。然后遍历 b_i 中每个 URL, 看在 HashSet 集合中是否存在, 若存在, 说明这就是共同的 URL, 可以把这个 URL 保存到一个单独的文件中。

总结

最后总结一下:

1. 分而治之, 进行哈希取余;
2. 对每个子文件进行 HashSet 统计。

4. 如何在100亿数据中找到中位数?

题目描述

给定100亿个无符号的乱序的整数序列, 如何求出这100亿个数的中位数 (中位数指的是排序后最中间那个数), 内存只有**512M**。

分析

中位数问题可以看做一个统计问题, 而不是排序问题, 无符号整数大小为4B, 则能表示的数的范围可为 $0 \sim 2^{32} - 1$ (40亿), 如果没有限制内存大小, 则可以用一个 2^{32} (4GB) 大小的数组 (也叫做桶) 来保存100亿个数中每个无符号数出现的次数。遍历这100亿个数, 当元素值等于桶元素索引时, 桶元素的值加1。当统计完100亿个数以后, 则从索引为0的值开始累加桶的元素值, 当累加值等于50亿时, 这个值对应的索引为中位数。时间复杂度为 $O(n)$ 。

因为题目要求内存限制512M, 所以上述解法不合适。

下面分享另一种解法 (分治法的思想)

如果100亿个数字保存在一个大文件中, 可以依次读一部分文件到内存(不超过内存限制), 将每个数字用二进制表示, 比较二进制的最高位(第32位, 符号位, 0是正, 1是负)。

如果数字的最高位为0, 则将这个数字写入 file_0 文件中; 如果最高位为 1, 则将该数字写入 file_1 文件中。从而将100亿个数字分成了两个文件。

假设 file_0 文件中有 60 亿 个数字, file_1 文件中有 40 亿 个数字。那么中位数就在 file_0 文件中, 并且是 file_0 文件中所有数字排序之后的第 10 亿 个数字。因为 file_1 中的数都是负数, file_0 中的数都是正数, 也即这里一共只有 40 亿 个负数, 那么排序之后的第 50 亿 个数一定位于 file_0 中。

现在, 我们只需要处理 file_0 文件了, 不需要再考虑 file_1 文件。对于 file_0 文件, 同样采取上面的措施处理: 将 file_0 文件依次读一部分到内存(不超过内存限制), 将每个数字用二进制表示, 比较二进制的次高位 (第31位), 如果数字的次高位为0, 写入 file_0_0 文件中; 如果次高位为1, 写入 file_0_1 文件中。

现假设 file_0_0 文件中有 30 亿 个数字, file_0_1 中也有 30 亿 个数字, 则中位数就是: file_0_0 文件中的数字从小到大排序之后的第 10 亿 个数字。

抛弃 file_0_1 文件, 继续对 file_0_0 文件 根据 次次高位(第30位) 划分, 假设此次划分的两个文件为: file_0_0_0 中有 5 亿 个数字, file_0_0_1 中有 25 亿 个数字, 那么中位数就是 file_0_0_1 文件中的所有数字排序之后的 第 5 亿 个数。

按照上述思路, 直到划分的文件可直接加载进内存时, 就可以直接对数字进行快速排序, 找出中位数了。

5. 如何查询最热门的查询串？

题目描述

搜索引擎会通过日志文件把用户每次检索使用的所有查询串都记录下来，每个查询串的长度不超过 255 字节。

假设目前有 1000w 个记录（这些查询串的重复度比较高，虽然总数是 1000w，但如果除去重复后，则不超过 300w 个）。请统计最热门的 10 个查询串，要求使用的内存不能超过 1G。（一个查询串的重复度越高，说明查询它的用户越多，也就越热门。）

解答思路

每个查询串最长为 255B，1000w 个串需要占用约 2.55G 内存，因此，我们无法将所有字符串全部读入到内存中处理。

方法一：分治法

分治法依然是一个非常实用的方法。

划分为多个小文件，保证单个小文件中的字符串能被直接加载到内存中处理，然后求出每个文件中出现次数最多的 10 个字符串；最后通过一个小顶堆统计出所有文件中出现最多的 10 个字符串。

方法可行，但不是最好，下面介绍其他方法。

方法二：HashMap 法

虽然字符串总数比较多，但去重后不超过 300w，因此，可以考虑把所有字符串及出现次数保存在一个 HashMap 中，所占用的空间为 $300w \times (255 + 4) \approx 777M$ （其中，4 表示整数占用的 4 个字节）。由此可见，1G 的内存空间完全够用。

思路如下：

首先，遍历字符串，若不在 map 中，直接存入 map，value 记为 1；若在 map 中，则把对应的 value 加 1，这一步时间复杂度 $O(N)$ 。

接着遍历 map，构建一个 10 个元素的小顶堆，若遍历到的字符串的出现次数大于堆顶字符串的出现次数，则进行替换，并将堆调整为小顶堆。

遍历结束后，堆中 10 个字符串就是出现次数最多的字符串。这一步时间复杂度 $O(N \log 10)$ 。

方法三：前缀树法

方法二使用了 HashMap 来统计次数，当这些字符串有大量相同前缀时，可以考虑使用前缀树来统计字符串出现的次数，树的结点保存字符串出现次数，0 表示没有出现。

思路如下：

在遍历字符串时，在前缀树中查找，如果找到，则把结点中保存的字符串次数加 1，否则为这个字符串构建新结点，构建完成后把叶子结点中字符串的出现次数置为 1。

最后依然使用小顶堆来对字符串的出现次数进行排序。

方法总结

前缀树经常被用来统计字符串的出现次数。它的另外一个大的用途是字符串查找，判断是否有重复的字符串等。

作者：yanglbme 链接：<https://juejin.cn/post/6844904003998842887>

6. 如何找出排名前 500 的数？

题目描述

有 1w 个数组，每个数组有 500 个元素，并且有序排列。如何在这 10000*500 个数中找出前 500 的数？

方法1

题目中每个数组是排好序的，可以使用**归并**的方法。

先将第1个和第2个归并，得到500个数据。然后再将结果和第3个数组归并，得到500个数据，以此类推，直到最后找出前500个的数。

方法2

对于这种topk问题，更常用的方法是使用**堆排序**。

对本题而言，假设数组降序排列，可以采用以下方法：

首先建立大顶堆，堆的大小为数组的个数，即为10000，把每个数组最大的值存到堆中。

接着删除堆顶元素，保存到另一个大小为 500 的数组中，然后向大顶堆插入删除的元素所在数组的下一个元素。

重复上面的步骤，直到删除完第 500 个元素，也即找出了最大的前 500 个数。

示例代码如下：

```
import lombok.Data;

import java.util.Arrays;
import java.util.PriorityQueue;

@Data
public class DataWithSource implements Comparable<DataWithSource> {
    /**
     * 数值
     */
    private int value;

    /**
     * 记录数值来源的数组
     */
    private int source;

    /**
     * 记录数值在数组中的索引
     */
    private int index;

    public DataWithSource(int value, int source, int index) {
        this.value = value;
        this.source = source;
        this.index = index;
    }

    /**
     *
     * 由于 PriorityQueue 使用小顶堆来实现，这里通过修改
     * 两个整数的比较逻辑来让 PriorityQueue 变成大顶堆
     */
}
```

```

@Override
public int compareTo(DataWithSource o) {
    return Integer.compare(o.getValue(), this.value);
}
}

class Test {
    public static int[] getTop(int[][] data) {
        int rowSize = data.length;
        int columnSize = data[0].length;

        // 创建一个columnSize大小的数组，存放结果
        int[] result = new int[columnSize];

        PriorityQueue<DataWithSource> maxHeap = new PriorityQueue<>();
        for (int i = 0; i < rowSize; ++i) {
            // 将每个数组的最大一个元素放入堆中
            DataWithSource d = new DataWithSource(data[i][0], i, 0);
            maxHeap.add(d);
        }

        int num = 0;
        while (num < columnSize) {
            // 删除堆顶元素
            DataWithSource d = maxHeap.poll();
            result[num++] = d.getValue();
            if (num >= columnSize) {
                break;
            }

            d.setValue(data[d.getSource()][d.getIndex() + 1]);
            d.setIndex(d.getIndex() + 1);
            maxHeap.add(d);
        }
        return result;
    }

    public static void main(String[] args) {
        int[][] data = {
            {29, 17, 14, 2, 1},
            {19, 17, 16, 15, 6},
            {30, 25, 20, 14, 5},
        };

        int[] top = getTop(data);
        System.out.println(Arrays.toString(top)); // [30, 29, 25, 20, 19]
    }
}

```

7. 如何按照 query 的频度排序?

题目描述

有 10 个文件，每个文件大小为 1G，每个文件的每一行存放的都是用户的 query，每个文件的 query 都可能重复。要求按照 query 的频度排序。

解答思路

如果 query 的重复度比较大，可以考虑一次性把所有 query 读入内存中处理；如果 query 的重复率不高，那么可用内存不足以容纳所有的 query，这时候就需要采用分治法或其他的方法来解决。

方法一：HashMap 法

如果 query 重复率高，说明不同 query 总数比较小，可以考虑把所有的 query 都加载到内存中的 HashMap 中。接着就可以按照 query 出现的次数进行排序。

方法二：分治法

分治法需要根据数据量大小以及可用内存的大小来确定问题划分的规模。对于这道题，可以顺序遍历 10 个文件中的 query，通过 Hash 函数 `hash(query) % 10` 把这些 query 划分到 10 个小文件中。之后对每个小文件使用 HashMap 统计 query 出现次数，根据次数排序并写入到零外一个单独文件中。

接着对所有文件按照 query 的次数进行排序，这里可以使用归并排序（由于无法把所有 query 都读入内存，因此需要使用外排序）。

方法总结

- 内存若够，直接读入进行排序；
- 内存不够，先划分为小文件，小文件排好序后，整理使用外排序进行归并。

8. 大数据中 TopK 问题的常用套路

对于海量数据到处理经常会涉及到 topK 问题。在设计数据结构和算法的时候，主要需要考虑的应该是当前算法（包括数据结构）跟给定情境（比如数据量级、数据类型）的适配程度，和当前问题最核心的瓶颈（如降低时间复杂度，还是降低空间复杂度）是什么。

首先，我们来举几个常见的 topK 问题的例子：

1. 给定 100 个 int 数字，在其中找出最大的 10 个；
2. 给定 10 亿个 int 数字，在其中找出最大的 10 个（这 10 个数字可以无序）；
3. 给定 10 亿个 int 数字，在其中找出最大的 10 个（这 10 个数字依次排序）；
4. 给定 10 亿个不重复的 int 数字，在其中找出最大的 10 个；
5. 给定 10 个数组，每个数组中有 1 亿个 int 数字，在其中找出最大的 10 个；
6. 给定 10 亿个 string 类型的数字，在其中找出最大的 10 个（仅需要查 1 次）；
7. 给定 10 亿个 string 类型的数字，在其中找出最大的 k 个（需要反复多次查询，其中 k 是一个随机数字）。

上面这些问题看起来很相似，但是解决的方式却千差万别。稍有不慎，就可能使得 topK 问题成为系统的瓶颈。不过也不用太担心，接下来我会总结几种常见的解决思路，遇到问题的时候，大家把这些基础思路融会贯通并且杂糅组合，即可做到见招拆招。

8.1. 1. 堆排序法

这里说的是堆排序法，而不是快排或者希尔排序。虽然理论时间复杂度都是 $O(n\log n)$ ，但是堆排在做 topK 的时候有一个优势，就是可以维护一个仅包含 k 个数字的小顶堆（想清楚，为啥是小顶堆哦），当新加入的数字大于堆顶数字的时候，将堆顶元素剔除，并加入新的数字。

用 C++ 来说明，堆在 stl 中是 priority_queue（不是 set）。

```
int main() {
    const int topK = 3;
    vector<int> vec = {4,1,5,8,7,2,3,0,6,9};
    priority_queue<int, vector<int>, greater<>> pq;    // 小顶堆
    for (const auto& x : vec) {
        pq.push(x);
    }
}
```

```

        if (pq.size() > topK) {
            // 如果超出个数，则弹出堆顶（最小的）数据
            pq.pop();
        }
    }

    while (!pq.empty()) {
        cout << pq.top() << endl;    // 输出依次为7,8,9
        pq.pop();
    }

    return 0;
}

```

Java 中同样提供了 PriorityQueue 的数据结构。

8.2.2. 类似快排法

快排大家都知道，针对 topK 问题，可以对快排进行改进。仅对部分数据进行递归计算。比如，在 100 个数字中，找最大的 10 个，第一次循环的时候，povit 被移动到了 80 的位置，则接下来仅需要在后面的 20 个数字中找最大的 10 个即可。

这样做的优势是，理论最优时间复杂度可以达到 $O(n)$ ，不过平均时间复杂度还是 $O(n\log n)$ 。需要说明的是，通过这种方式，找出来的最大的 k 个数字之间，是无序的。

```

int partition(vector<int>& arr, int begin, int end) {
    int left = begin;
    int right = end;
    int povit = arr[begin];

    while (left < right) {
        while (left < right && arr[right] >= povit) {right--;}
        while (left < right && arr[left] <= povit) {left++;}
        if (left < right) {swap(arr[left], arr[right]);}
    }

    swap(arr[begin], arr[left]);
    return left;
}

void partSort(vector<int>& arr, int begin, int end, int target) {
    if (begin >= end) {
        return;
    }

    int povit = partition(arr, begin, end);
    if (target < povit) {
        partSort(arr, begin, povit - 1, target);
    } else if (target > povit) {
        partSort(arr, povit + 1, end, target);
    }
}

vector<int> getMaxNumbers(vector<int>& arr, int k) {
    int size = (int)arr.size();
    // 把求最大的k个数，转换成求最小的size-k个数字
    int target = size - k;
}

```

```

    partSort(arr, 0, size - 1, target);
    vector<int> ret(arr.end() - k, arr.end());
    return ret;
}

int main() {
    vector<int> vec = {4,1,5,8,7,2,3,0,6,9};
    auto ret = getMaxNumbers(vec, 3);

    for (auto x : ret) {
        cout << x << endl;    // 输出7, 8, 9 (理论上无序)
    }

    return 0;
}

```

8.3. 3. 使用 bitmap

有时候 topK 问题会遇到数据量过大，内存无法全部加载。这个时候，可以考虑将数据存放至 bitmap 中，方便查询。

比如，给出 10 个 int 类型的数据，分别是【13, 12, 11, 1, 2, 3, 4, 5, 6, 7】，int 类型的数据每个占据 4 个字节，那这个数组就占据了 40 个字节。现在，把它们放到一个 16 个长度 bool 的 bitmap 中，结果就是【0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0】，在将空间占用降低至 4 字节的同时，也可以很方便的看出，最大的 3 个数字，分别是 11, 12 和 13。

需要说明的是，bitmap 结合跳表一起使用往往有奇效。比如以上数据还可以记录成：从第 1 位开始，有连续 7 个 1；从第 11 位开始，有连续 3 个 1。这样做，空间复杂度又得到了进一步的降低。

这种做法的优势，当然是降低了空间复杂度。不过需要注意一点，bitmap 比较适合不重复且有范围（比如，数据均在 0 ~ 10 亿之间）的数据的查询。至于有重复数据的情况，可以考虑与 hash 等结构的混用。

8.4. 4. 使用 hash

如果遇到了查询 string 类型数据的大小，可以考虑 hash 方法。

举个例子，10 个 string 数字

【"1001", "23", "1002", "3003", "2001", "1111", "65", "834", "5", "987"】找最大的 3 个。我们先通过长度进行 hash，得到长度最大为 4，且有 5 个长度为 4 的 string。接下来再通过最高位值做 hash，发现有 1 个最高位为"3"的，1 个为"2"的，3 个为"1"的。接下来，可以通过再设计 hash 函数，或者是循环的方式，在 3 个最高位为"1"的 string 中找到最大的一个，即可找到 3 个最值大的数据。

这种方法比较适合网址或者电话号码的查询。缺点就是如果需要多次查询的话，需要多次计算 hash，并且需要根据实际情况设计多个 hash 函数。

8.5. 5. 字典树

字典树（trie）的具体结构和查询方式，不在这里赘述了，自行百度一下就有很多。这里主要说一下优缺点。

字典树的思想，还是通过前期建立索引信息，后期可以反复多次查询，并且后期增删数据也很方便。比较适合需要反复多次查询的情况。

比如，反复多次查询字符串（例如：z>y>...>b>a）最大的 k 个 url 这种，使用字典树把数据存储一遍，就非常适合。既减少了空间复杂度，也加速了查询效率。

8.6.6. 混合查询

以上几种方法，都是比较独立的方法。其实，在实际工作中，遇到更多的问题还是混合问题，这就需要我们相关的内容，融会贯通并且做到活学活用。

我举个例子：我们的分布式服务跑在 10 台不同机器上，每台机器上部署的服务均被请求 10000 次，并且记录了个这 10000 次请求的耗时（耗时值为 int 数据），找出这 10*10000 次请求中，从高到低的找出耗时最大的 50 个。看看这个问题，很现实吧。我们试着用上面介绍的方法，组合一起来求解。

8.6.1. 方法一

首先，对每台机器上的 10000 个做类似快排，找出每台机器上 top50 的耗时信息。此时，单机上的这 50 条数据是无序的。

然后，再将 10 台机器上的 50 条数据（共 500 条）放到一起，再做一次类似快排，找到最大的 50 个（此时应该这 50 个应该是无序的）。

最后，对这 50 个数据做快排，从而得到最终结果。

8.6.2. 方法二

首先通过堆排，分别找出 10 台机器上耗时最高的 50 个数据，此时的这 50 个数据，已经是从小到大的有序的了。

然后，我们依次取出 10 台机器中，耗时最高的 5 条放入小顶堆中。

最后，遍历 10 台机器上的数据，每台机器从第 6 个数据开始往下循环，如果这个值比堆顶的数据大，则抛掉堆顶数据并且把它加入，继续用下一个值进行同样比较。如果这个值比堆顶的值小，则结束当前循环，并且在下一台机器上做同样操作。

以上我介绍了两种方法，并不是为了说明哪种方法更好，或者时间复杂度更低。而是想说同样的事情有多种不同的解决方法，而且随着数据量的增加，可能会需要更多组合形式。在这个领域，数据决定了数据结构，数据结构决定了算法。

没有最好的方法，只有不断找寻更好的方法的程序员。适合的，才会是最好的。

计算机网络

计算机网络重要知识点&高频面试题是我的[知识星球](#)内部专属资料，已经整理到[Java面试手册完整版](#)。

Java面试手册

+

保存并发布于: 2023-01-01 22:58:51

☆

🔗

分享

...

⋮

⋮

面试准备

面试题&重要知识点

Java

数据库

常见框架

消息队列

计算机基础

计算机网络高频面试题

操作系统高频面试题

数据结构重要知识点总结

算法重要知识点总结

分布式

开发工具

系统设计

海量数据场景高频面试题

面经篇

校招分享

四次挥手

1. A的应用进程先向其TCP发出连接释放报文段 ($FIN=1, seq=u$), 并停止再发送数据, 主动关闭TCP连接, 进入 $FIN-WAIT-1$ (终止等待1) 状态, 等待B的确认。

2. B收到连接释放报文段后即发出确认报文段 ($ACK=1, ack=u+1, seq=v$), B进入 $CLOSE-WAIT$ (关闭等待) 状态, 此时的TCP处于半关闭状态, A到B的连接释放。

3. A收到B的确认后, 进入 $FIN-WAIT-2$ (终止等待2) 状态, 等待B发出的连

大纲

网络分层结构

三次握手

两次握手可以吗?

四次挥手

第四次挥手为什么要等待2MSL?

为什么是四次挥手?

TCP有哪些特点?

说说TCP报文首部有哪些字段, 其...

TCP和UDP的区别?

TCP和UDP 分别对应的常见应用...

TCP的粘包和拆包

说说TCP是如何确保可靠性的呢?

说下TCP的滑动窗口机制

详细讲一下拥塞控制?

HTTP协议的特点?

HTTP报文格式

HTTP状态码有哪些?

HTTP 协议包括哪些请求?

HTTP状态码301和302的区别?

URI和URL的区别

POST和GET的区别?

404 Not Found 错误码的含义

- 面试准备
- 面试题&重要知识点
- Java
- 数据库
- 常见框架
- 消息队列
- 计算机基础
- 分布式
- 开发工具
- 系统设计

如何设计一个抢红包系统	01-11 22:31
权限系统设计方案	01-10 23:51
实时订阅推送设计与实现	01-04 16:21
如何用 Redis 统计用户访问量?	01-05 00:09
扫码登录原理	01-11 23:20
订单30分钟未支付自动取消怎么实现?	01-02 14:29
如何设计一个短链系统?	01-03 00:17
秒杀系统	01-02 14:27
秒杀系统设计的5个要点	01-04 16:22
微信红包后台系统设计	01-02 14:27
单点登录 (SSO) 的设计与实现	01-02 14:26
如何设计一个优惠券系统?	01-02 14:26

- 海量数据场景高频面试题
- 面经篇
- 校招分享

除了Java面试手册完整版之外，星球还有很多其他**优质资料**，比如包括Java项目、进阶知识、实战经验总结、优质书籍、笔试面试资源等等。

👉 优质学习资源汇总：

- 1、阿里巴巴技术团队的实战经验总结，链接：<https://t.zsxq.com/08TUWuPRg>
- 2、Tomcat架构解析、性能优化、源码解读：<https://t.zsxq.com/09XZgyC7z>
- 3、尚学堂2022版全套的java学习视频（300集），链接：<https://t.zsxq.com/08NfkBCup>
- 4、并发编程视频教程，包含：线程基础、线程协作、并发工具类、锁、AQS、线程池、实战项目等：<https://t.zsxq.com/09vF2onhe>
- 5、LeetCode刷题笔记（包含C/C++、Java、Go三个版本），链接：<https://t.zsxq.com/080mxtafG>
- 6、尚硅谷Java学科全套教程（总207.77GB），链接：<https://t.zsxq.com/0807PC4s4>
- 7、自动动手写Java虚拟机PDF，链接：<https://t.zsxq.com/08jBRPI5a>
- 8、字节大佬总结的Java面试资料，涵盖Java开发常见的知识点，链接：<https://t.zsxq.com/08UxPuP34>
- 9、阿里的算法笔试题精选合集，涵盖70+算法题目、大厂笔试考点，链接：<https://t.zsxq.com/082R1dqFF>
- 10、Java工程师进阶知识大全，包含高并发架构、分布式架构，链接：<https://t.zsxq.com/08j7WWUZn>
- 11、MySQL优质教程，包含：6大范式讲解、7大日志剖析、7大SQL性能分析工具等内容，链接：<https://t.zsxq.com/08d6j44JK>

- 12、阿里巴巴微服务治理白皮书，高效构建完整的微服务治理体系。链接：<https://t.zsxq.com/09fkXNKk>
- 13、RocketMQ 从入门到实战。链接：<https://t.zsxq.com/09MecUORI>
- 14、阿里巴巴性能调优实战手册，包含Java编程性能调优、多线程性能调优、JVM性能监测及调优、设计模式调优、数据库性能调优、实战演练六大模块。链接：<https://t.zsxq.com/09gdKMpY8>
- 15、Java面试真题，跟面试手册内容互补，建议准备面试的手两份一起过一遍！链接：<https://t.zsxq.com/09rVOpWNh>
- 16、畅购商城视频教程（含源码）。链接：<https://t.zsxq.com/09LOcfEDB>
- 17、面试题合集，包含技术和非技术问题，如Dubbo面试题、设计模式面试题、Netty面试题、Tomcat面试题、Nginx面试题、HR面试常见问题、英语面试题等。链接：<https://t.zsxq.com/09G34Sf16>
- 18、精选1000套简历模板，超实用！链接：<https://t.zsxq.com/096963tQt>
- 19、马士兵老师的Java基础视频教程：<https://t.zsxq.com/09twuX7b5>
- 20、MySQL45讲专栏：<https://t.zsxq.com/09V2U7K2A>
- 21、分布式优惠券系统：<https://t.zsxq.com/09ktzHUOZ>
- 22、mpvue课程小程序全栈开发：<https://t.zsxq.com/09aexZzl8>
- 22、微信小程序入门与实战 常用组件API开发技巧项目实战：<https://t.zsxq.com/09aexZzl8>
- 23、尚硅谷最新版尚筹网Java项目实战
<https://t.zsxq.com/094UkGCUy>

- 24、Redis优质教程：<https://t.zsxq.com/090gAzWEx>
- 25、JVM调优&多线程的教程：<https://t.zsxq.com/09u1h71vm>
- 26、Java架构师-技术专家的资料：<https://t.zsxq.com/090oV3kz2>
- 27、SSO单点登录原理：<https://t.zsxq.com/096U2VUJB>
- 28、3天全面深入学习zookeeper（含源码资料）<https://t.zsxq.com/09tijiPEu>
- 29、SpringBoot+SpringCloud全彩指南（终极版），链接：<https://t.zsxq.com/08DEQYliY>
- 30、Mybatis视频教程：<https://t.zsxq.com/09r81SkuK>
- 31、《漫画Java》PDF版本，链接：<https://t.zsxq.com/09L69kPDO>
- 32、Nginx优质视频教程：<https://t.zsxq.com/09gQBCuuF>

专属一对一的提问答疑，帮你解答各种疑难问题，包括自学Java路线、职业规划、面试问题等等。大彬会优先解答球友的问题。



大彬

2023-01-02 20:33

收进专栏

...

Alvin_Xu 提问： 本人是19年毕业至今工作了3年半左右，一直在同一个岗位上，近段时间因为公司项目组资金不足被调岗了，新岗位和自己的技术栈不太匹配，觉得如果一直变动会影响自己的职业发展。

这么些年来一直有点混吃等死，**想在现在开始寻求改变**，希望通过学习，让自己有所变化。不过对于学习方式以及学习路线**很是迷茫**，希望大彬老师可以帮忙解惑，非常感谢。

工作三年半，有点迷茫，想要寻求改变！

技术栈：

有一定的Java基础，对JVM原理，体系结构，垃圾回收机制等有部分了解

熟悉Scala&Spark开发，了解部署spark server项目及脚本

掌握spring, springCloud微服务架构开发，

了解Elastic search,Mule gateway, Feign等的使用

熟悉Jenkins, Kubernetes, PCF, G3等devops部署及调试

熟悉AppDynamic, patrol, LWM等程序监控平台的使用及配置

了解NexusIQ,Checkmarx,sonar等代码质量监控平台的使用及调优

熟悉sqlServer及Mongodb数据库

我一直都是认为，在现在这个就业环境下，“专才”的竞争力是要大于“全才”的，专注一个方向，对你的职业发展更为有利。

从你的技术栈来看，相对还是偏“杂”一些，Java、大数据、devops等都有涉及。而且现在还有调岗的可能，我建议



大彬

2023-01-10 21:41

收进专栏

...

🍌 提问： 想请教一下大彬，对于java开发和大数据开发的明年秋招情况的预测。java岗位多卷度也高，每家公司基本都有相关的职位，大数据开发的话培训班比岗位多，只有大公司和数据公司会有这类岗位。24年秋招的话**选择哪个方向会比较合适呢？**

岗位方向选择：Java还是大数据？

建议选大数据吧。就这几年校招来看，大数据岗位拿offer的难度相比Java还是比较小一些的。而且距离24年秋招还有一年多时间，转大数据完全来得及，而且有了Java基础，再来学大数据，应该会比较快入门。

再说下大数据和后端的差异。大数据门槛比Java高，除了熟悉数据库的操作之外，还要学习大数据整个生态，需要会分布式、数仓、数据分析统计等知识。因为大数据的学习门槛比Java高，所以市场上培训大数据的相比Java会少一些，竞争也相对小，没有Java那么卷。

另外，大数据薪资总体比Java开发高一些（同一家公司同一级别，普通开发岗比大数据开发薪资会少一点），这也是大数据方向的一个优势。

不过呢，小点的公司，可能没有大数据的需求，毕竟业务量不大（小公司通常也不太建议去，坑多）。



2022-12-27 19:29

收进专栏

...

麻烦问一下，遇到一个面试题是：描述最能代表自己编程能力的代码，解释为什么这段代码提现了你的编程能力。这种题怎么答





2人赞过

举了两个例子，帮助球友解答疑惑



大彬：这种可以结合自己平时做的一些项目来讲。
比如说有一些比较复杂的功能，你是如何去实现的，像涉及很多状态流转的，使用了状态机去处理，通过"动作"来进行状态的变更。
或者项目中有很多相似的实现逻辑，把这些相似的逻辑抽象出来，做成公共组件，这也是一种编程能力的体现。
当然还有很多，上面只是两个简答的例子，具体可以结合你自身的项目去思考、挖掘，适当包装一下也是没问题的。
2022-12-27 21:43



大彬：可以从背景（为什么这么做）、实现原理（怎么实现的）、收获（学到了什么）这三个方面来回答
2022-12-27 21:52



回复 大彬：请问工作中如何应用的OOP原则怎么答啊
2022-12-28 13:07

大彬 回复 ：面向对象编程有六大原则：对象单一职责、里式替换原则、迪米特法则、开闭原则、依赖倒置、接口隔离。
以对象单一职责为例，在设计类的时候，必须职责明确，比如商品类，里面相关的属性和方法都必须跟商品相关，不能出现订单等不相关的内容。
其他原则也是类似，可以看看优质专栏-Java基础部分，有关于面向对象编六大原则的介绍
2022-12-28 14:21

另外星球还提供**简历指导、修改服务**，大彬已经帮**90+**个小伙伴修改了简历，相对还是比较有经验的。



求职意向: Java



生日: 1999.11

籍贯: 贵州三穗

电话: 1

学历: 本科学士

政治面貌: 团员

邮箱: 20

.com



教育背景 Education

2019.09-2023.06

天津工业大学

软件工程

主修课程: 高级程序语言设计、数据库原理及应用、数据结构、算法设计与分析、计算机组成原理、操作系统、计算机网络、编译原理、软件系统分析、软件质量测试等。

校园经历: 在校期间任大创信息部副部长, 负责创业就业信息收集整理, 微信公众号的维护和运营。在校期间注重实践, 参加技术学习型社团, 参与过许多小项目, 曾多次在软件实训中获取优异成绩。



项目经历 Project

2021.12-2022.03

企业网盘系统

项目组长、后端开发

项目简介: 基于 Springboot+Dubbo 开发的, 提供文件管理上传下载共享在线编辑的基础网盘。

● 基于 Redis 做分布式锁, 实现文件切块上传功能;

● WebSocket+Netty 做消息推送, 实现文件共享消息提示;

● Elasticsearch 做全文检索引擎, 实现大量文件的检索;

● HDFS 做分布式文件系统, 实现分布式文件存储。

2022.05-2023.07

微服务电商系统

后端开发

项目简介: SpringCloud 开发的电商系统, 集成了监控中心、网关等系统功能。

● 使用 RabbitMq 做消息队列, 实现延迟消息;

● 基于 SpringSecurity 做登录认证和权限控制;

● 使用 Spring 的 AOP, 添加切面实现接口日志记录;

● Docker 做应用容器引擎, Kubernetes 做应用容器管理平台;



专业技能 Expertise

1. 熟练掌握 java 语法, 和 IO、多线程、反射、异常及一些常用类, 掌握 JVM 原理;
2. 熟悉 Spring、SpringMVC、MyBatis、Springboot, 了解 SpringCloud 等开发框架;
3. 熟悉 MySQL 数据库, 具备一定的 SQL 优化能力; 熟悉 Redis 非关系型数据库;
4. 熟悉 HTML、CSS、JavaScript 等基础前端技术, 了解 Vue 等前端开发框架;
5. 熟练使用 Linux 操作系统; 熟悉常用的设计模式; 熟悉一些开发中常用中间件;
6. 具备一定的需求分析和系统设计能力, 较强的逻辑分析和独立解决问题能力。



自我评价 Evaluation

本人热爱 JAVA 开发工作, 注重实干, 熟悉 SpringBoot 开发, 具备良好的编码规范; 熟悉关系型数据库, 熟练使用 SQL 语言, 掌握基本的 SQL 优化; 服从安排、工作积极主动, 责任心与上进心较强, 具备良好的学习能力与合作的团队精神。

主修课程不用写这么多, 甚至可以不用写, 留点版面给其他内容

感觉似乎没有因果关系。可以描述清晰点

消息推送具体描述下怎么实现的

HDFS面试官应该会问, 思考下会问什么问题

延时消息怎么实现的

项目经历这块如果有数据支撑, 会更好, 比如一些慢查询优化, 从xx秒到xx毫秒等



18235... | 78386...qq.com
CSDN : https://blog.csdn.net/w...n_819295
求职意向：后端开发工程师 (北京)



实习经历

建议实习经历写实习项目，项目经历写自己的项目，一般都是这么写的

京东零售 - 市场营销与用户增长部 - 产品研发部

2021.12 - 2022.06

专业技能

1. 熟练运用 **JAVASE**、**JAVAAE** 主流技术；
2. 掌握 **Java** 基础，熟悉面向对象、集合、多线程、锁、I/O模型；
3. 对 **JVM** 原理有初步的了解，包括内存模型、垃圾回收机制以及垃圾回收算法；
4. 深入理解 **Redis** 数据类型和内部实现、熟悉其使用场景、持久化、淘汰策略；
5. 熟练使用 **MySQL**，熟悉MySQL索引、事务、锁机制、SQL调优并在项目中实际运用；
6. 熟悉 **MyBatis**、**Spring**、**SpringMVC**、**SpringBoot** 等主流开发框架；
7. 熟练使用**Kafka**，熟悉Kafka基本结构、消费者组；
8. 熟悉**Linux**操作系统，**Nginx** 等技术；
9. 了解常用的数据结构和算法；

专业技能可以补充分布式、微服务相关的（如果不熟悉建议了，面试大概率会问到）

项目经历

官旗OMS

工作描述，可以按照三个步骤展开：1.使用什么技术；2.做了什么事情；3.达成什么效果。

2021.12 - 2022.06

项目背景：京东在抖音开设官店为了方便管理订单并履约发货，需要在京东商城与多平台之间建立一个中枢。

工作描述：参与了V1.0、V1.1、V1.2迭代需求的开发。在 V1.0 需求中主要负责拉取抖音平台订单消息，对用户的创建订单、支付订单、风控订单等进行对接，解决了mq的重复消费。在 V1.1 需求中对用户的敏感信息进行了加密处理，使用了延迟消费减少了调用抖音接口的次数，并对SQL进行了优化提高了查询效率。在 V1.2 需求中使用多线程并发的拉取抖音订单详情中的信息，对其中的信息进行过滤筛选并归档到 TIDB 中。在日常工作中还负责将线上数据同步到 hive 数据仓库，实现日报的自动出数。

人脸签到

性能优化，如果有数据指标支撑，会更好。比如从xxx秒优化到xxx毫秒，查询时间减少95%等

2021.08 - 2021.09

- 1、该项目是一个前后端分离的项目，主要功能有授权模块、人脸比对、签到、签退、统计时长
- 2、功能的实现主要是基于SpringBoot + WebSocket + Swagger + Redis 技术栈实现，前后端采用JSON作为交互数据，采用了心跳机制保持了长连接，签到或签退会对已在线的用户进行实时广播，使用 Redis 缓存中间件对热数据进行缓存，并解决了缓存穿透的问题。使用Swagger对代码和注释进行开发文档的生成。

教育经历

项目经历这块，建议写上自己的收获

山西农业信息学院 - 数据科学与大数据技术 本科

2019.09 - 2023.06

荣誉奖项

2020年校级一等奖学金、2021年中国大学生计算机设计大赛三等奖

第十一届蓝桥杯大赛二等奖

社团和组织经历

双体系卓越人才教育基地软件部部长

2020.06 - 2021.06

积极组织学员学习软件编程类相关知识，对下一届学员进行学习进度的规划、以及定期检查学习情况，积极完成老师安排的任务、组织同学们参加各种计算机相关的比赛。

知识星球加入方式：



大彬 送你一张星球优惠券

「大彬的学习圈」

立减

¥ **50**

新人立减券

2023/12/12 13:00 后失效

 知识星球

长按扫码领取优惠 ▶



操作系统

1. 操作系统的四个特性？

并发：同一段时间内多个程序执行（与并行区分，并行指的是同一时刻有多个事件，多处理器系统可以使程序并行执行）

共享：系统中的资源可以被内存中多个并发执行的进程共同使用

虚拟：通过分时复用（如分时系统）以及空分复用（如虚拟内存）技术把一个物理实体虚拟为多个

异步：系统进程用一种走走停停的方式执行，（并不是一下子走完），进程什么时候以怎样的速度向前推进是不可预知的

2. 进程线程

进程是指一个内存中运行的应用程序，每个进程都有自己独立的一块内存空间。

线程是比进程更小的执行单位，它是在一个进程中独立的控制流，一个进程可以启动多个线程，每条线程并行执行不同的任务。

进程和线程的区别如下：

- 调度：进程是资源管理的基本单位，线程是程序执行的基本单位。
- 切换：线程上下文切换比进程上下文切换要快得多。
- 拥有资源：进程是拥有资源的一个独立单位，线程不拥有系统资源，但是可以访问隶属于进程的资源。
- 系统开销：创建或撤销进程时，系统都要为之分配或回收系统资源，如内存空间，I/O设备等，OS所付出的开销显著大于在创建或撤销线程时的开销，进程切换的开销也远大于线程切换的开销。

3. 并发和并行

并发就是在一段时间内，多个任务都会被处理；但在某一时刻，只有一个任务在执行。单核处理器可以做到并发。比如有两个进程 A 和 B，A 运行一个时间片之后，切换到 B，B 运行一个时间片之后又切换到 A。因为切换速度足够快，所以宏观上表现为在一段时间内能同时运行多个程序。

并行就是在同一时刻，有多个任务在执行。这个需要多核处理器才能完成，在微观上就能同时执行多条指令，不同的程序被放到不同的处理器上运行，这个是物理上的多个进程同时进行。

4. 多线程相较单线程的好处

- 1、并发提升程序执行效率
- 2、提升CPU利用率，访存的时候可以切换线程来执行
- 3、更快的响应速度，可以有专门的线程来监听用户请求和专门的线程来处理请求。比如监听线程和工作线程是两个线程，这样监听就负责监听，工作的就负责工作，监听到用户请求马上把请求转到工作线程去处理，监听线程继续监听

5. 什么是协程？

协程是一种用户态的轻量级线程。

协程不是由操作系统内核管理，而是完全由用户程序所控制，这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。

协程可以理解为可以暂停执行的函数。它拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈，直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。

6. 线程和协程有什么区别呢？

1、线程是抢占式，而协程是非抢占式的，所以需要用户自己释放使用权来切换到其他协程，因此同一时间其实只有一个协程拥有运行权，相当于单线程的能力。2、线程是协程的资源。协程通过可以关联任意线程或线程池的执行器（Interceptor）来间接使用线程的资源的。

7. 进程通信

进程间通信方式有以下几种：

1、管道通信

匿名管道(pipe)：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。有名管道是半双工的通信方式，数据只能单向流动。

2、消息队列

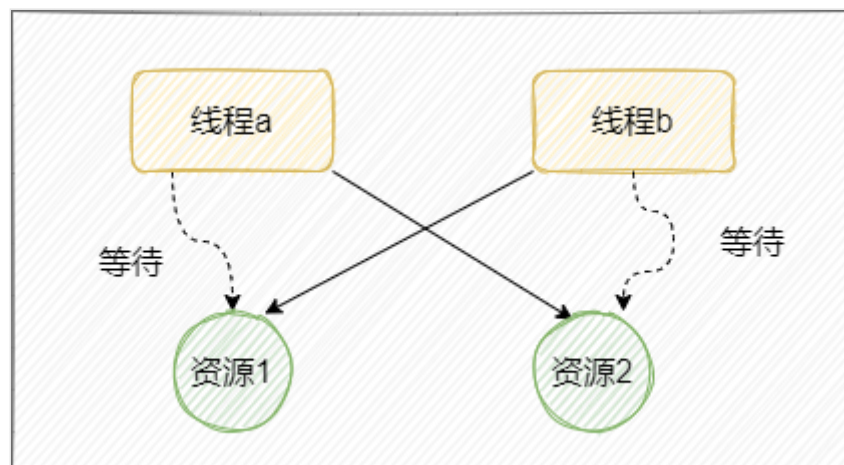
3、**共享内存**。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量，配合使用，来实现进程间的同步和通信。

4、**信号量**。信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

8. 什么是死锁？

死锁是指两个或两个以上的线程在执行过程中，因争夺资源而造成的一种互相等待的现象。若无外力作用，它们都将无法推进下去。

如下图所示，线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方持有的资源，所以这两个线程就会互相等待而进入死锁状态。



下面通过例子说明线程死锁，代码来自并发编程之美。

```
public class DeadLockDemo {
    private static Object resource1 = new Object();//资源 1
    private static Object resource2 = new Object();//资源 2

    public static void main(String[] args) {
        new Thread(() -> {
```

```

        synchronized (resource1) {
            System.out.println(Thread.currentThread() + "get resource1");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread() + "waiting get
resource2");

            synchronized (resource2) {
                System.out.println(Thread.currentThread() + "get
resource2");
            }
        }
    }, "线程 1").start();

    new Thread(() -> {
        synchronized (resource2) {
            System.out.println(Thread.currentThread() + "get resource2");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread() + "waiting get
resource1");

            synchronized (resource1) {
                System.out.println(Thread.currentThread() + "get
resource1");
            }
        }
    }, "线程 2").start();
}
}

```

代码输出如下：

```

Thread[线程 1,5,main]get resource1
Thread[线程 2,5,main]get resource2
Thread[线程 1,5,main]waiting get resource2
Thread[线程 2,5,main]waiting get resource1

```

线程 A 通过 `synchronized (resource1)` 获得 resource1 的监视器锁，然后通过 `Thread.sleep(1000)`。让线程 A 休眠 1s 为的是让线程 B 得到执行然后获取到 resource2 的监视器锁。线程 A 和线程 B 休眠结束了都开始企图请求获取对方的资源，然后这两个线程就会陷入互相等待的状态，这也就产生了死锁。

9. 死锁怎么产生？怎么避免？

死锁产生的四个必要条件：

- 互斥：一个资源每次只能被一个进程使用
- 请求与保持：一个进程因请求资源而阻塞时，不释放获得的资源
- 不剥夺：进程已获得的资源，在未使用之前，不能强行剥夺
- 循环等待：进程之间循环等待着资源

避免死锁的方法：

- 互斥条件不能破坏，因为加锁就是为了保证互斥
- 一次性申请所有的资源，避免线程占有资源而且在等待其他资源
- 占有部分资源的线程进一步申请其他资源时，如果申请不到，主动释放它占有的资源
- 按序申请资源

10. 进程调度策略有哪几种？

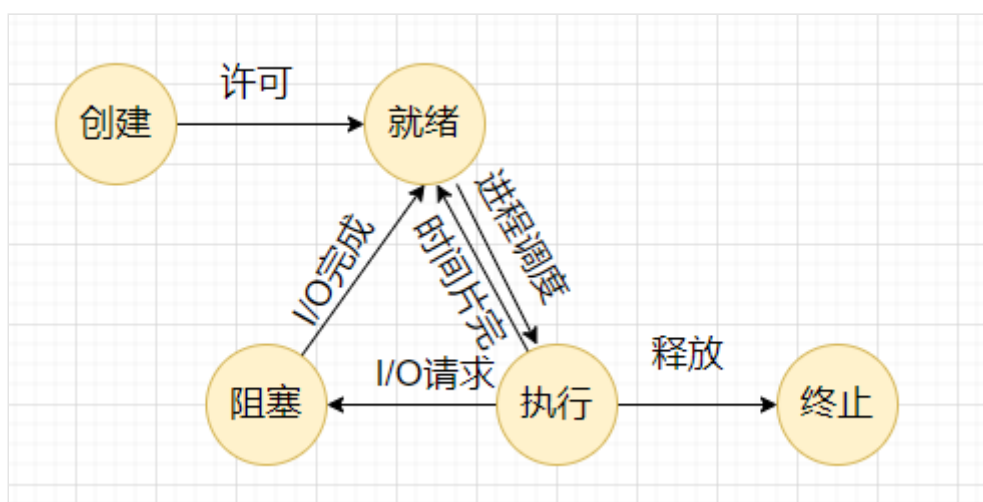
- **先来先服务**：非抢占式的调度算法，按照请求的顺序进行调度。有利于长作业，但不利于短作业，因为短作业必须一直等待前面的长作业执行完毕才能执行，而长作业又需要执行很长时间，造成了短作业等待时间过长。另外，对 **I/O** 密集型进程也不利，因为这种进程每次进行 **I/O** 操作之后又得重新排队。
- **短作业优先**：非抢占式的调度算法，按估计运行时间最短的顺序进行调度。长作业有可能会饿死，处于一直等待短作业执行完毕的状态。因为如果一直有短作业到来，那么长作业永远得不到调度。
- **最短剩余时间优先**：最短作业优先的抢占式版本，按剩余运行时间的顺序进行调度。当一个新的作业到达时，其整个运行时间与当前进程的剩余时间作比较。如果新的进程需要的时间更少，则挂起当前进程，运行新的进程。否则新的进程等待。
- **时间片轮转**：将所有就绪进程按 **FCFS** 的原则排成一个队列，每次调度时，把 **CPU** 时间分配给队首进程，该进程可以执行一个时间片。当时间片用完时，由计时器发出时钟中断，调度程序便停止该进程的执行，并将它送往就绪队列的末尾，同时继续把 **CPU** 时间分配给队首的进程。

时间片轮转算法的效率和时间片的大小有很大关系：因为进程切换都要保存进程的信息并且载入新进程的信息，如果时间片太小，会导致进程切换得太频繁，在进程切换上就会花过多时间。而如果时间片过长，那么实时性就不能得到保证。

- **优先级调度**：为每个进程分配一个优先级，按优先级进行调度。为了防止低优先级的进程永远得不到调度，可以随着时间的推移增加等待进程的优先级。

11. 进程有哪些状态？

进程一共有 **5** 种状态，分别是创建、就绪、运行（执行）、终止、阻塞。



- 运行状态就是进程正在 **CPU** 上运行。在单处理机环境下，每一时刻最多只有一个进程处于运行状态。
- 就绪状态就是说进程已处于准备运行的状态，即进程获得了除 **CPU** 之外的一切所需资源，一旦得到 **CPU** 即可运行。
- 阻塞状态就是进程正在等待某一事件而暂停运行，比如等待某资源为可用或等待 **I/O** 完成。即使 **CPU** 空闲，该进程也不能运行。

运行态→阻塞态：往往是由于等待外设，等待主存等资源分配或等待人工干预而引起的。**阻塞态→就绪态**：则是等待的条件已满足，只需分配处理器后就能运行。**运行态→就绪态**：不是由于自身原因，而是由外界原因使运行状态的进程让出处理器，这时候就变成就绪态。例如时间片用完，或有更高优先级的进程来抢占处理器等。**就绪态→运行态**：系统按某种策略选中就绪队列中的一个进程占用处理器，此时就变成了运行态。

12. 操作系统里的内存碎片怎么理解？

内存碎片通常分为内部碎片和外部碎片：

1. 内部碎片是由于采用固定大小的内存分区，当一个进程不能完全使用分给它的固定内存区域时就会产生内部碎片。通常内部碎片难以完全避免
2. 外部碎片是由于某些未分配的连续内存区域太小，以至于不能满足任意进程的内存分配请求，从而不能被进程利用的内存区域。

有什么解决办法？

现在普遍采取的内存分配方式是段页式内存分配。将内存分为不同的段，再将每一段分成固定大小的页。通过页表机制，使段内的页可以不必连续处于同一内存区域。

13. 虚拟内存

虚拟存储器就是具有请求调入功能，能从逻辑上对内存容量加以扩充的一种存储器系统，虚拟内存有多次性，对换性和虚拟性三个特征，它可以将程序分多次调入内存，使得在较小的用户空间可以执行较大的用户程序，所以同时容纳更多的进程并发执行，从而提高系统的吞吐量。发生缺页时可以调入一个段也可以调入一个页，取决于内存的存储管理方式。虚拟性表示虚拟内存和物理内存的映射。

Linux下，进程不能直接读写内存物理地址，只能访问【虚拟内存地址】。操作系统会把虚拟内存地址-->物理地址。

虚拟内存解决有限的内存空间加载较大应用程序的问题，根据需要在内存和磁盘之间来回传送数据。

通过段页表的形式，虚拟内存中取一段连续的内存空间映射到主内存中，主内存空间的程序段可以不连续。

14. 什么是分页？

把内存空间划分为**大小相等且固定的块**，作为主存的基本单位。因为程序数据存储在不同的页面中，而页面又离散的分布在内存中，**因此需要一个页表来记录映射关系，以实现从页号到物理块号的映射。**

访问分页系统中内存数据需要**两次的内存访问**（一次是从内存中访问页表，从中找到指定的物理块号，加上页内偏移得到实际物理地址；第二次就是根据第一次得到的物理地址访问内存取出数据）。

15. 什么是分段？

分页是为了提高内存利用率，而分段是为了满足程序员在编写代码的时候的一些逻辑需求(比如数据共享，数据保护，动态链接等)。

分段内存管理当中，**地址是二维的，一维是段号，二维是段内地址；其中每个段的长度是不一样的，而且每个段内部都是从0开始编址的。**由于分段管理中，每个段内部是连续内存分配，但是段和段之间是离散分配的，因此也存在一个逻辑地址到物理地址的映射关系，相应的就是段表机制。

16. 分页和分段有什么区别？

- 分页对程序员是透明的，但是分段需要程序员显式划分每个段。
- 分页的地址空间是一维地址空间，分段是二维的。
- 页的大小不可变，段的大小可以动态改变。

- 分页主要用于实现虚拟内存，从而获得更大的地址空间；分段主要是为了使程序和数据可以被划分为逻辑上独立的地址空间并且有助于共享和保护。

17. 页面置换算法

为什么要页面置换：

因为应用程序是分多次装入内存的，所以运行到一定的时间，一定会发生缺页。地址映射的过程中，如果页面中发现要访问的页面不在内存中，会产生缺页中断。此时操作系统必须在内存里选择一个页面把他移出内存，为即将调入的页面让出空间。选择淘汰哪一页的规则就是页面置换算法

几种页面置换算法：

最佳置换算法（理想）：将当前页面中在未来最长时间不会被访问的页置换出去

先进先出：淘汰最早调入的页面

最近最久未使用 LRU：每个页面有一个t来记录上次页面被访问直到现在，每次置换时置换t值最大的页面（用寄存器或栈实现）

时钟算法clock（也被称为最近未使用算法NRU）：页面设置访问为，将页面链接为一个环形列表，每个页有一个访问位0/1, 1表示又一次获救的机会，下次循环指针指向它时可以免除此次置换，但是会把访问位置为0，代表他下次如果碰到循环指针就该被置换了。页面被访问的时候访问位设为1。页面置换的时候，如果当前指针的访问位为0，置换，否则将这个值置为0，循环直到遇到访问位为0的页面。

改进型Clock算法：在clock算法的基础上添加一个修改位，优先替换访问位和修改位都是0的页面，其次替换访问位为0修改位为1的页面。

最少使用算法LFU：设置寄存器记录页面被访问次数，每次置换当前访问次数最少的。

18. 用户态和内核态

内核态：cpu可以访问内存的所有数据，包括外围设备，例如硬盘，网卡，cpu也可以将自己从一个程序切换到另一个程序。

用户态：只能受限的访问内存，且不允许访问外围设备，占用cpu的能力被剥夺，cpu资源可以被其他程序获取。

最大的区别就是权限不同，在运行在用户态下的程序不能直接访问操作系统内核数据结构和程序。

18.1. 为什么要有这两种状态？

内核速度快但是资源有限，能控制的进程数不多，所以需要速度慢一些的用户态协助，但是为了避免用户态被恶意利用，所以限制了用户态程序的权限。

需要限制不同的程序之间的访问能力，防止他们获取别的程序的内存数据，或者获取外围设备的数据，并发送到网络，CPU划分出**两个权限等级** -- 用户态和内核态。

18.2. 什么时候转换

1、系统调用：

用户进程主动发起的。用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作，比如fork()就是执行一个创建新进程的系统调用

用户程序使用系统调用，系统调用会转换为内核态并调用操作系统

2、发生异常：

会从当前运行进程切换到处理此异常的内核相关程序中

3、外围设备的中断：

所有程序都运行在用户态，但在从硬盘读取数据、或从键盘输入时，这些事情只有操作系统能做，程序需要向操作系统请求以程序的名义来执行这些操作。这个时候用户态程序切换到内核态。

19. 什么是缓冲区溢出？有什么危害？

缓冲区溢出是指当计算机向缓冲区填充数据时超出了缓冲区本身的容量，溢出的数据覆盖在合法数据上。

危害有以下两点：

- 程序崩溃，导致拒绝额服务
- 跳转并且执行一段恶意代码

造成缓冲区溢出的主要原因是程序中没有仔细检查用户输入。

20. IO多路复用

IO多路复用是指内核一旦发现进程指定的一个或者多个IO条件准备读取，它就通知该进程。IO多路复用适用如下场合：

- 当客户处理多个描述字时（一般是交互式输入和网络套接口），必须使用I/O复用。
- 当一个客户同时处理多个套接口时，而这种情况是可能的，但很少出现。
- 如果一个TCP服务器既要处理监听套接口，又要处理已连接套接口，一般也要用到I/O复用。
- 如果一个服务器即要处理TCP，又要处理UDP，一般要使用I/O复用。
- 如果一个服务器要处理多个服务或多个协议，一般要使用I/O复用。
- 与多进程和多线程技术相比，I/O多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。



扫码关注我



微信搜索



程序员大彬

公众号后台回复【面试】获取面试手册
PDF最新版

数据结构

1. 各种数据结构应用场景

- 栈：逆序输出；语法检查，符号成对判断；方法调用
- 二叉树：表达式树
- B+/B-树：文件系统；数据库索引
- 哈夫曼树：数据压缩算法
- 哈希表：提高查找性能
- 红黑树：大致平衡的二叉查找树，相对AVL树，插入删除结点较快，查找性能没有提升

2. 数组

数组的优点：

- 存取速度快

数组的缺点：

- 事先必须知道数组的长度
- 插入删除元素很慢
- 空间通常是有限制的
- 需要大块连续的内存块
- 插入删除元素的效率很低

3. 链表

优点：

- 空间没有限制
- 插入删除元素很快

缺点：存取速度很慢

分类

- 单向链表 一个节点指向下一个节点。
- 双向链表 一个节点有两个指针域。
- 循环链表 能通过任何一个节点找到其他所有的节点，将两种(双向/单向)链表的最后一个结点指向第一个结点从而实现循环

4. 哈希表

散列表（Hash table，也叫哈希表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

5. 栈

我们把类似于弹夹那种先进后出的数据结构称为栈，栈是限定**仅在表尾进行插入和删除操作**的线性表，我们把允许插入和删除的一端称为栈顶，另一端称为栈底，不含任何数据元素的栈称为空栈，栈又称后进后出的线性表，简称LIFO结构。

栈的特殊之处在于限制了这个线性表的**插入和删除位置，它始终只在栈顶进行**。这也就使得：栈底是固定的，最先进栈的只能在栈底。

栈的插入操作，叫做进栈；栈的删除操作叫做出栈。

6. 队列

队列是只允许**在一端进行插入操作，而在另一端进行删除操作**的线性表，队列是一种先进先出的线性表，简称FIFO，允许插入的一端称为队尾（Rear），允许删除的一端称为队头(Front)。向队中插入元素称为进队，新元素进队后成为新的队尾元素；向队中删除元素称为出队，元素出队后，其后继元素就成为新的队头元素。

7. 树

树是一种数据结构，它看上去像一棵"圣诞树"，它的根在上，叶朝下。

树有多个节点(node)，用以储存元素。某些节点之间存在一定的关系，用连线表示，连线称为边(edge)。边的上端节点称为父节点，下端称为子节点。树像是一个不断分叉的树根。

7.1. 二叉树

最多有两棵子树的树被称为二叉树

满二叉树: 二叉树中所有非叶子结点的度都是2，且叶子结点都在同一层次上

完全二叉树: 如果一个二叉树与满二叉树前m个节点的结构相同，这样的二叉树被称为完全二叉树

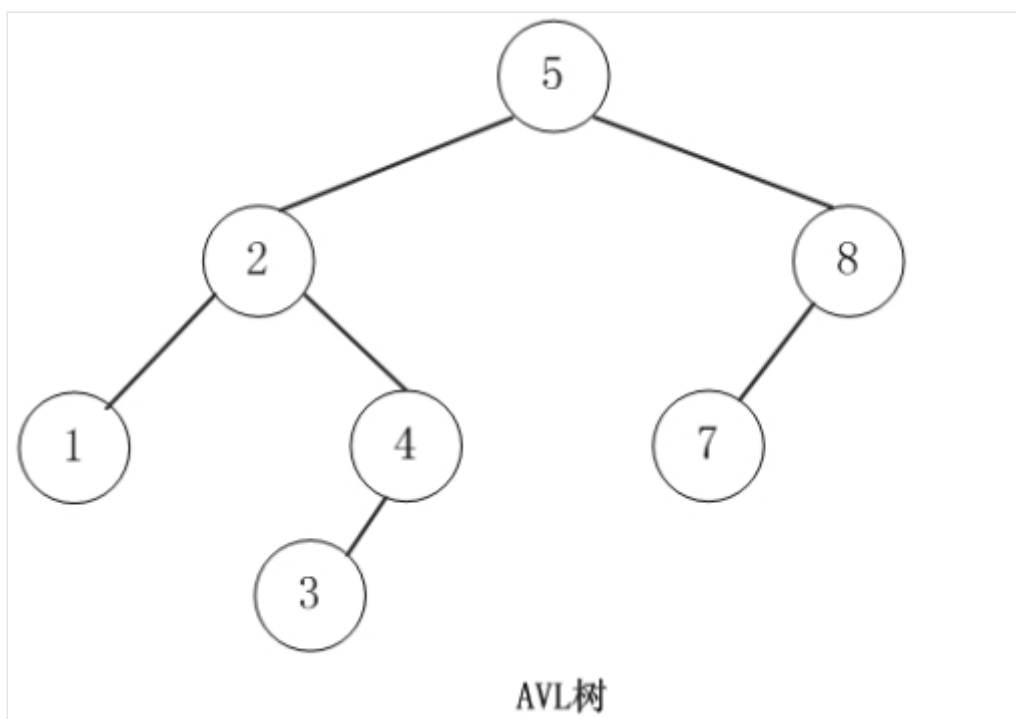
7.2. 二叉查找树

指一棵空树或者具有下列性质的二叉树。

- 若任意节点的左子树不空，则左子树上所有节点的值均小于它的根节点的值；
- 若任意节点的右子树不空，则右子树上所有节点的值均大于它的根节点的值；
- 任意节点的左、右子树也分别为二叉查找树；
- 没有键值相等的节点。

7.3. AVL树

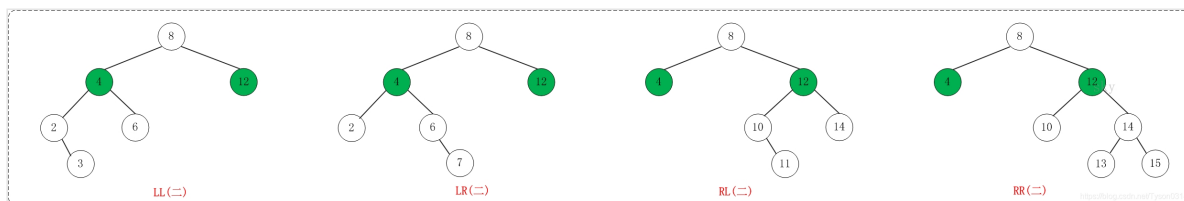
平衡二叉搜索树，它是一棵空树或它的左右两个子树的高度差的绝对值不超过1。



简单了解一下**左旋与右旋**的概念。

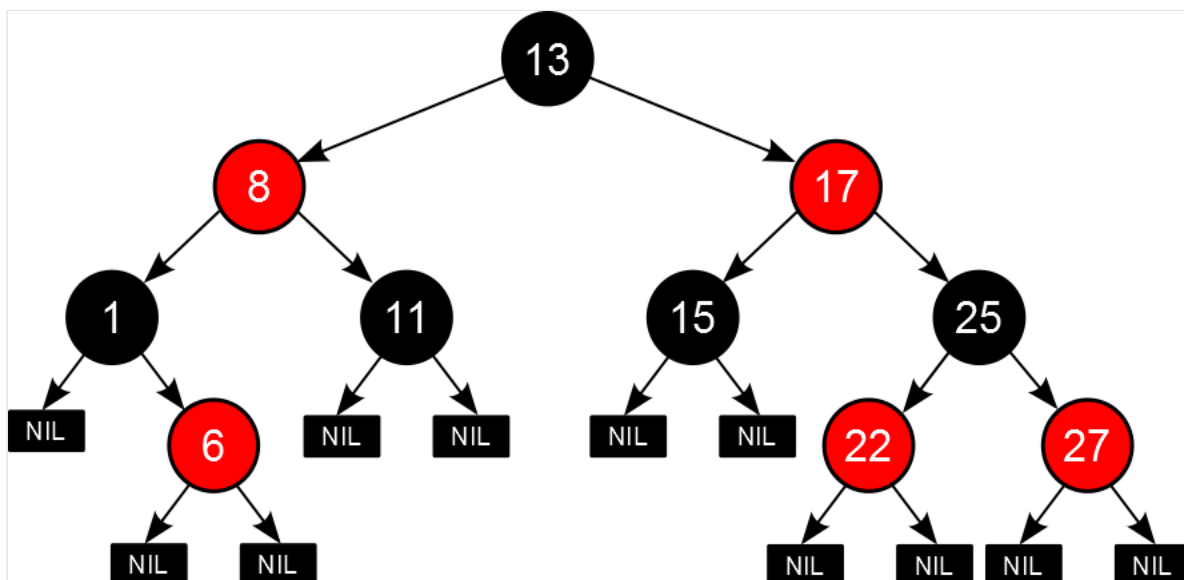
左旋与右旋就是为了解决不平衡问题而产生的，我们构建一颗AVL树的过程会出现结点平衡因子（平衡因子就是二叉排序树中每个结点的左子树和右子树的高度差。）绝对值大于1的情况，这时就可以通过左旋或者右旋操作来达到平衡的目的。

四种旋转情况



7.4. 红黑树

红黑树是对AVL树的优化，只要求部分平衡，用非严格的平衡来换取增删节点时候旋转次数的降低，提高了插入和删除的性能。查找性能并没有提高，查找的时间复杂度是 $O(\log n)$ 。红黑树通过左旋、右旋和变色维持平衡。



对于插入节点，AVL和红黑树都是最多两次旋转来实现平衡。对于删除节点，avl需要维护从被删除节点到根节点root这条路径上所有节点的平衡，旋转的量级为 $O(\log N)$ ，而红黑树最多只需旋转3次。

红黑树的特性：

- 每个节点或者是黑色，或者是红色。
- 根节点和叶子节点是黑色，叶子节点为空。
- 红色节点的子节点必须是黑色的。
- 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点，保证没有一条路径会比其他路径长一倍。

优点：相比avl树，红黑树插入删除的效率更高。红黑树维持红黑性质所做的红黑变换和旋转的开销，相较于avl树维持平衡的开销要小得多。

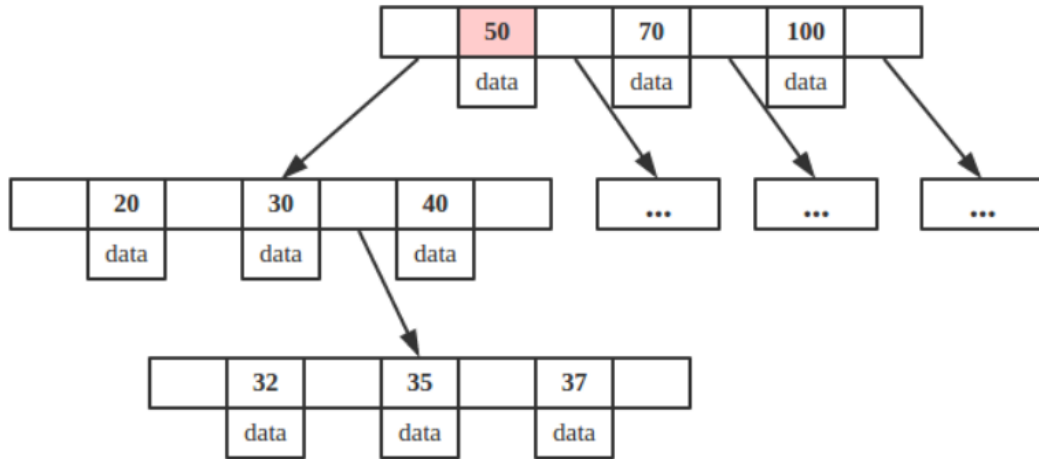
应用场景

- Java ConcurrentHashMap & TreeMap
- C++ STL: map & set
- linux进程调度Completely Fair Scheduler,用红黑树管理进程控制块
- epoll在内核中的实现，用红黑树管理事件块
- nginx中，用红黑树管理timer等

7.5. B树

也称B-树，属于多叉树又名平衡多路查找树。

B-树



B-树

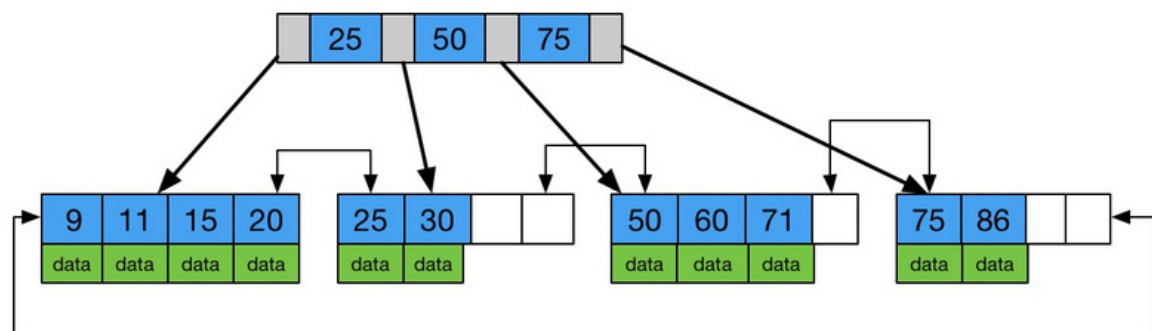
规则：

- $1 < \text{子节点数} \leq m$ ， m 代表一个树节点最多有多少个查找路径
- 每个节点最多有 $m-1$ 个关键字，非根节点至少有 $m/2$ 个关键字，根节点最少可以只有1个关键字
- 每个节点都有指针指向子节点，指针个数=关键字个数+1，叶子节点指针指向null

B-树的特性：

1. 关键字集合分布在整颗树中；
2. 任何一个关键字只出现在一个节点中；
3. 搜索有可能在非叶子结点结束；

B+树是B-树的变体，也是一种多路搜索树。B+的搜索与B-树基本相同，区别是B+树只有达到叶子结点才命中，B-树可以在非叶子结点命中。B+树更适合文件索引系统。



简化 B+ 树

B-和B+树的区别：

- B+树的非叶子节点不包含data，叶子节点使用链表连接，便于区间查找和遍历。B-树需要遍历整棵树，范围查询性能没有B+树好。

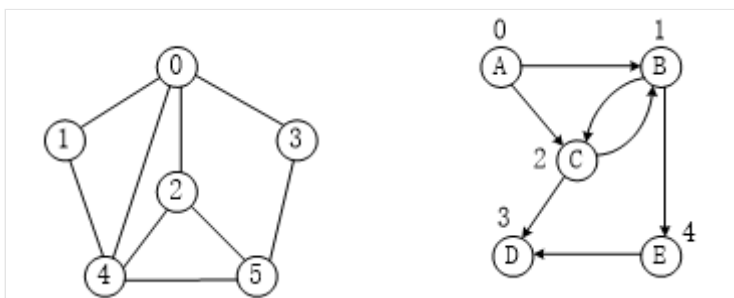
- B-树的非树节点存放数据和索引，搜索可能在非叶子结点结束，访问更快。

8. 图

图(Graph)是由顶点的有穷非空集合和顶点之间边的集合组成，通常表示为: $G(V,E)$ ，其中， G 表示一个图， V 是图 G 中顶点的集合， E 是图 G 中边的集合。

和线性表，树的差异:

- 线性表中我们把数据元素叫元素，树中将数据元素叫结点，在图中数据元素，我们则称之为顶点(Vertex)。
- 线性表可以没有元素，称为空表；树中可以没有节点，称为空树；但是，在图中不允许没有顶点(有穷非空性)。
- 线性表中的各元素是线性关系，树中的各元素是层次关系，而图中各顶点的关系是用边来表示(边集可以为空)。



8.1. 相关术语

- 顶点的度

顶点 V_i 的度(Degree)是指在图中与 V_i 相关联的边的条数。对于有向图来说，有入度(In-degree)和出度(Out-degree)之分，有向图顶点的度等于该顶点的入度和出度之和。

- 邻接

若无向图中的两个顶点 V_1 和 V_2 存在一条边 (V_1, V_2) ，则称顶点 V_1 和 V_2 邻接(Adjacent)；

若有向图中存在一条边 $\langle V_3, V_2 \rangle$ ，则称顶点 V_3 与顶点 V_2 邻接，且是 V_3 邻接到 V_2 或 V_2 邻接直 V_3 ；

- 路径

在无向图中，若从顶点 V_i 出发有一组边可到达顶点 V_j ，则称顶点 V_i 到顶点 V_j 的顶点序列为从顶点 V_i 到顶点 V_j 的路径(Path)。

- 连通

若从 V_i 到 V_j 有路径可通，则称顶点 V_i 和顶点 V_j 是连通(Connected)的。

- 权(Weight)

有些图的边或弧具有与它相关的数字，这种与图的边或弧相关的数叫做权(Weight)。

8.2. 类型

无向图

如果图中任意两个顶点之间的边都是无向边(简而言之就是没有方向的边)，则称该图为无向图(Undirected graphs)。

无向图中的边使用小括号“()”表示; 比如 (V_1, V_2) ;

有向图

如果图中任意两个顶点之间的边都是有向边(简而言之就是有方向的边), 则称该图为有向图(Directed graphs)。

有向图中的边使用尖括号“<>”表示; 比如<V1,V2>

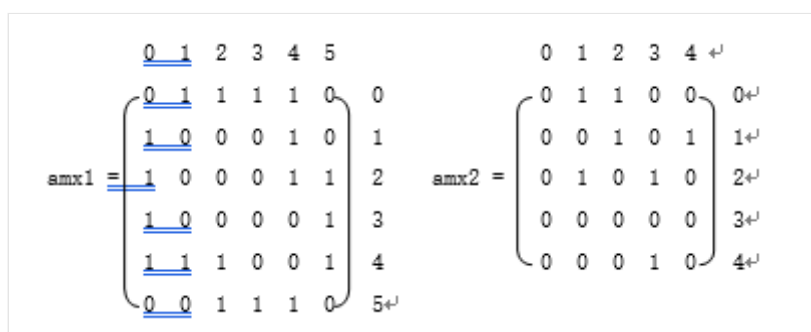
完全图

- **无向完全图**: 在无向图中, 如果任意两个顶点之间都存在边, 则称该图为无向完全图。(含有n个顶点的无向完全图有 $(n \times (n-1))/2$ 条边)
- **有向完全图**: 在有向图中, 如果任意两个顶点之间都存在方向互为相反的两条弧, 则称该图为有向完全图。(含有n个顶点的有向完全图有 $n \times (n-1)$ 条边)

8.3. 图的存储结构

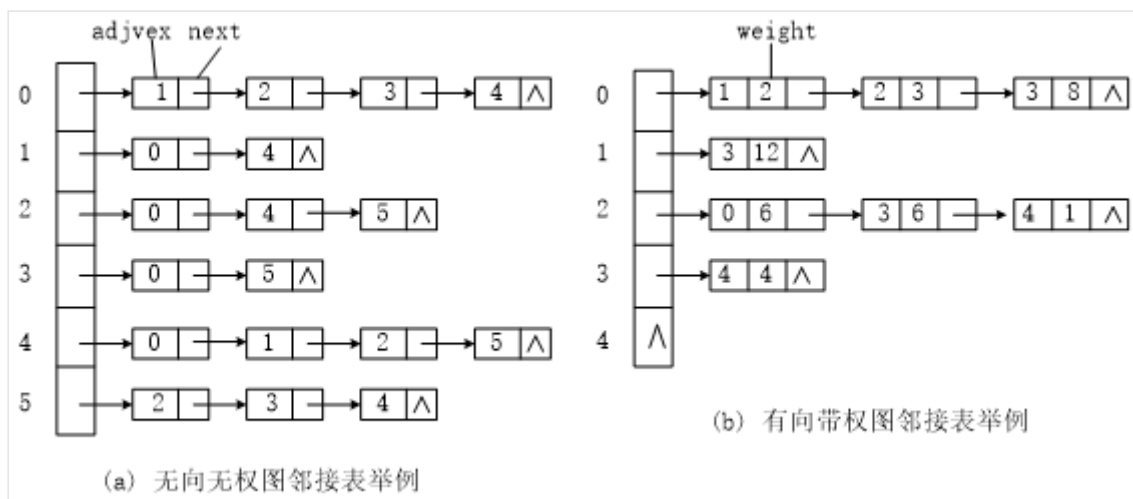
1、邻接矩阵

图的邻接矩阵(Adjacency Matrix)存储方式是用两个数组来表示图。一个一维数组存储图中顶点信息, 一个二维数组(称为邻接矩阵)存储图中的边或弧的信息。



2、邻接表

邻接表由表头节点和表节点两部分组成, 图中每个顶点均对应一个存储在数组中的表头节点。如果这个表头节点所对应的顶点存在邻接节点, 则把邻接节点依次存放于表头节点所指向的单向链表中。



下面给出建立图的邻接表中所使用的边结点类的定义

```

public class EdgeNode {    //定义邻接表中的边结点类型
    int adjvex;            //邻接点域
    int weight;            //边的权值域，假定为整型，对于无权图，边的权值为1
    EdgeNode next;        //指向下一个边结点的链接域
    public EdgeNode(int adj, EdgeNode nt)
    {    //对无权图中的边结点进行初始化
        adjvex=adj; next=nt; weight=1;
    }
    public EdgeNode(int adj, int wgt, EdgeNode nt)
    {    //对有权图中的边结点进行初始化
        adjvex=adj; next=nt; weight=wgt;
    }
}

```

图的接口类定义如下：

```

public interface Graph
{
    boolean createGraph(EdgeElement[]d); //根据边集数组参数d建立一个图
    int graphType();                    //返回图的类型
    int vertices();                     //返回图中的顶点数
    int edges();                        //返回图中的边数
    boolean find(int i, int j);         //从图中查找一条边(i,j)是否存在
    boolean putEdge(EdgeElement theEdge); //向图中插入一条边theEdge
    boolean removeEdge(int i, int j);   //从图中删除一条边(i,j)
    int degree(int i);                  //返回顶点i的度
    int inDegree(int i);                //返回顶点i的入度
    int outDegree(int i);               //返回顶点i的出度
    void output();                      //以图的顶点集和边集的形式输出一个图
    void depthFirstSearch(int v);       //从顶点v开始深度优先搜索遍历图
    void breadthFirstSearch(int v);     //从顶点v开始广度优先搜索遍历图
    void clearGraph();                  //清除图中的所有内容
}

```

8.4. 图的遍历

深度优先遍历

```

private void dfs(int i, boolean[] visited) {
    System.out.println(i + " ");
    visited[i] = true;
    EdgeNode p = a[i];
    while(p != null) {
        int j = p.adjvex;
        if(!visited[j]) {
            dfs(j, visited);
        }
        p = p.next;
    }
}

```

广度优先搜索

```

private void bfs(int i, boolean[] visited) {
    LinkedList<Integer> queue = new LinkedList<>();

```

```
System.out.print(i + " ");
visited[i] = true;
queue.offer(i);
while(!queue.isEmpty()) {
    int k = queue.poll();
    EdgeNode p = a[k];

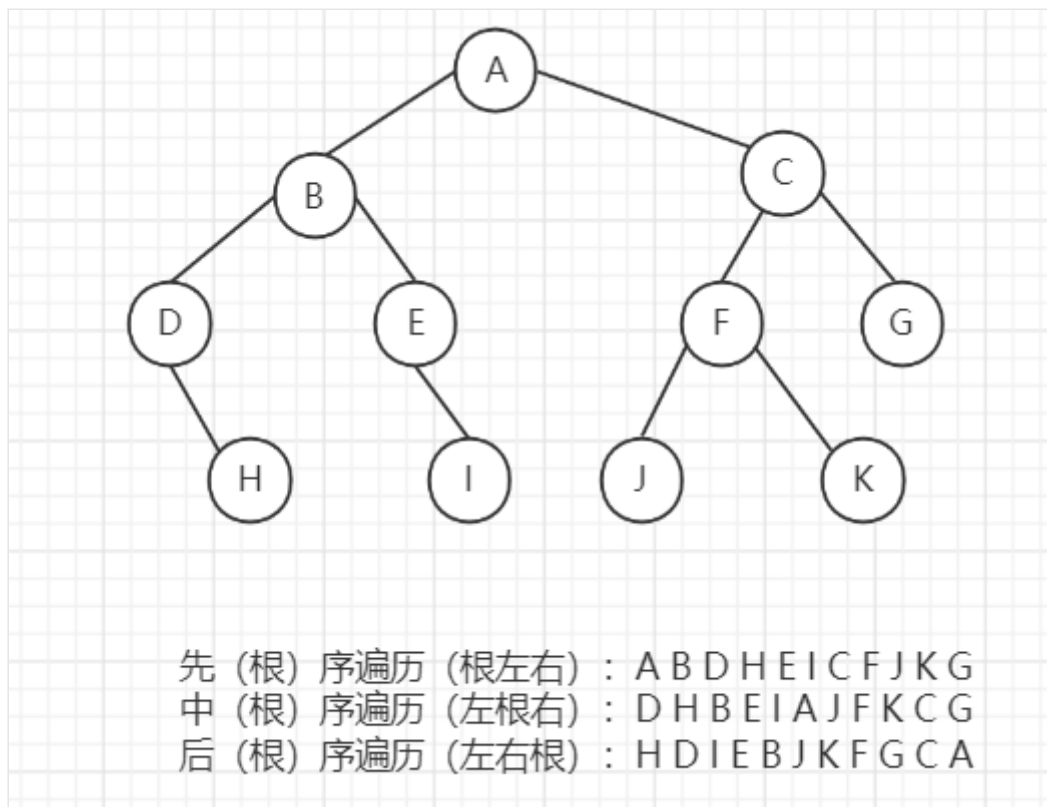
    while(p != null) {
        int j = p.adjvex;
        if(!visited[j]) {
            System.out.print(j + " ");
            visited[j] = true;
            queue.offer(j);
        }
        p = p.next;
    }
}
}
```

算法

1. 二叉树的遍历

二叉树是一种非常重要的数据结构，很多其它数据结构都是基于二叉树的基础演变而来的。

二叉树的先序、中序和后序属于深度优先遍历DFS，层次遍历属于广度优先遍历BFS。



四种主要的遍历思想为：

前序遍历：根结点 ---> 左子树 ---> 右子树

中序遍历：左子树 ---> 根结点 ---> 右子树

后序遍历：左子树 ---> 右子树 ---> 根结点

层次遍历：只需按层次遍历即可

1.1. 前序遍历

遍历思路：根结点 ---> 左子树 ---> 右子树。

根据前序遍历的顺序，优先访问根结点，然后在访问左子树和右子树。所以，对于任意结点node，第一部分即直接访问之，之后在判断左子树是否为空，不为空时即重复上面的步骤，直到其为空。若为空，则需要访问右子树。注意，在访问过左孩子之后，需要反过来访问其右孩子，可以是栈这种数据结构来支持。对于任意一个结点node，具体步骤如下：

1. 访问结点，并把结点node入栈，当前结点置为左孩子；
2. 判断结点node是否为空，若为空，则取出栈顶结点并出栈，将右孩子置为当前结点；否则重复a)步直到当前结点为空或者栈为空（可以发现栈中的结点就是为了访问右孩子才存储的）

```
public void preOrderTraverse2(TreeNode root) {  
    LinkedList<TreeNode> stack = new LinkedList<>();  
    TreeNode pNode = root;
```

```

while (pNode != null || !stack.isEmpty()) {
    if (pNode != null) {
        System.out.print(pNode.val+" ");
        stack.push(pNode);
        pNode = pNode.left;
    } else { //pNode == null && !stack.isEmpty()
        TreeNode node = stack.pop();
        pNode = node.right;
    }
}
}
}

```

1.2. 中序遍历

遍历思路：左子树 ---> 根结点 ---> 右子树

```

public List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> res = new ArrayList<>();
    Deque<TreeNode> deque = new ArrayDeque<>();

    while (!deque.isEmpty() || root != null) {
        while (root != null) {
            deque.push(root);
            root = root.left;
        }
        root = deque.pop();
        res.add(root.val);
        root = root.right;
    }

    return res;
}

```

1.3. 后序遍历

遍历思路：左子树 ---> 右子树 ---> 根结点。

使用 null 作为标志位，访问到 null 说明此次递归调用结束。

```

class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> res = new LinkedList<>();
        if (root == null) {
            return res;
        }

        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);
        while (!stack.isEmpty()) {
            root = stack.pop();
            if (root != null) {
                stack.push(root); //最后访问
                stack.push(null);
                if (root.right != null) {
                    stack.push(root.right);
                }
            }
        }
    }
}

```

```
        if (root.left != null) {
            stack.push(root.left);
        }
    } else { //值为null说明此次递归调用结束，将节点值存进结果
        res.add(stack.pop().val);
    }
}

return res;
}
```

1.4. 层序遍历

只需要一个队列即可，先在队列中加入根结点。之后对于任意一个结点来说，在其出队列的时候，访问之。同时如果左孩子和右孩子有不为空，入队列。

```
public void levelTraverse(TreeNode root) {
    if (root == null) {
        return;
    }
    LinkedList<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    while (!queue.isEmpty()) {
        TreeNode node = queue.poll();
        System.out.print(node.val+" ");
        if (node.left != null) {
            queue.offer(node.left);
        }
        if (node.right != null) {
            queue.offer(node.right);
        }
    }
}
```

2. 排序算法

常见的排序算法主要有：冒泡排序、插入排序、选择排序、快速排序、归并排序、堆排序、基数排序。各种排序算法的时间空间复杂度、稳定性见下图。

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	Shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n\log_2n)$	$O(n\log_2n)$	$O(n\log_2n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n\log_2n)$	$O(n\log_2n)$	$O(n^2)$	$O(\log_2n)$	不稳定
归并排序		$O(n\log_2n)$	$O(n\log_2n)$	$O(n\log_2n)$	$O(n)$	稳定
基数排序		$O(d(r+n))$	$O(d(r+n))$	$O(d(r+n))$	$O(d(r+n))$	稳定

注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数

2.1. 冒泡排序

冒泡排序是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果它们的顺序错误就把它们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

思路：

- 比较相邻的元素。如果第一个比第二个大，就交换它们两个；
- 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；
- 针对所有的元素重复以上的步骤，除了最后一个；
- 重复步骤1~3，直到排序完成。

代码实现：

```
public void bubbleSort(int[] arr) {
    if (arr == null) {
        return;
    }
    boolean flag;
    for (int i = arr.length - 1; i > 0; i--) {
        flag = false;
        for (int j = 0; j < i; j++) {
            if (arr[j] > arr[j + 1]) {
                int tmp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;
                flag = true;
            }
        }
        if (!flag) {
            return;
        }
    }
}
```

2.2. 插入排序

插入排序（Insertion-Sort）的算法描述是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。插入排序在实现上，通常采用in-place排序（即只需用到O(1)的额外空间的排序），因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

算法描述：

一般来说，插入排序都采用in-place在数组上实现。具体算法描述如下：

- 从第一个元素开始，该元素可以认为已经被排序；
- 取出下一个元素，在已经排序的元素序列中从后向前扫描；
- 如果该元素（已排序）大于新元素，将该元素移到下一位置；
- 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；
- 将新元素插入到该位置后；
- 重复步骤2~5。

代码实现：

```

public void insertSort(int[] arr) {
    if (arr == null) {
        return;
    }
    for (int i = 1; i < arr.length; i++) {
        int tmp = arr[i];
        int j = i;
        for (; j > 0 && tmp < arr[j - 1]; j--) {
            arr[j] = arr[j - 1];
        }
        arr[j] = tmp;
    }
}

```

2.3. 选择排序

表现最稳定的排序算法之一，因为无论什么数据进去都是 $O(n^2)$ 的时间复杂度，所以用到它的时候，数据规模越小越好。唯一的好处可能就是不占用额外的内存空间。

选择排序(Selection-sort)是一种简单直观的排序算法。它的工作原理：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

思路：n个记录的直接选择排序可经过n-1趟直接选择排序得到有序结果。具体算法描述如下：

- 初始状态：无序区为 $R[1..n]$ ，有序区为空；
- 第i趟排序($i=1,2,3...n-1$)开始时，当前有序区和无序区分别为 $R[1..i-1]$ 和 $R(i..n)$ 。该趟排序从当前无序区中-选出关键字最小的记录 $R[k]$ ，将它与无序区的第1个记录R交换，使 $R[1..i]$ 和 $R[i+1..n)$ 分别变为记录个数增加1个的新有序区和记录个数减少1个的新无序区；
- n-1趟结束，数组有序化了。

代码实现：

```

public void selectionSort(int[] arr) {
    if (arr == null) {
        return;
    }
    for (int i = 0; i < arr.length - 1; i++) {
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[i] > arr[j]) {
                int tmp = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp;
            }
        }
    }
}

```

2.4. 希尔排序

希尔排序是希尔（Donald Shell）于1959年提出的一种排序算法。希尔排序也是一种插入排序，它是简单插入排序经过改进之后的一个更高效的版本，也称为缩小增量排序，同时该算法是冲破 $O(n^2)$ 的第一批算法之一。它与插入排序的不同之处在于，它会优先比较距离较远的元素。希尔排序又叫缩小增量排序。

希尔排序是把记录按下表的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰被分成一组，算法便终止。

代码实现：

```
public static int[] ShellSort(int[] array) {
    int len = array.length;
    int temp, gap = len / 2;
    while (gap > 0) {
        for (int i = gap; i < len; i++) {
            temp = array[i];
            int preIndex = i - gap;
            while (preIndex >= 0 && array[preIndex] > temp) {
                array[preIndex + gap] = array[preIndex];
                preIndex -= gap;
            }
            array[preIndex + gap] = temp;
        }
        gap /= 2;
    }
    return array;
}
```

2.5. 基数排序

基数排序也是非比较的排序算法，对每一位进行排序，从最低位开始排序，复杂度为 $O(kn)$ ， n 为数组长度， k 为数组中的数的最大的位数；

基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，低优先级相同的低优先级高的在前。基数排序基于分别排序，分别收集，所以是稳定的。

算法描述：

- 取得数组中的最大数，并取得位数；
- arr 为原始数组，从最低位开始取每个位组成radix数组；
- 对radix进行计数排序（利用计数排序适用于小范围数的特点）；

代码实现：

```
public static int[] RadixSort(int[] array) {
    if (array == null || array.length < 2)
        return array;
    // 1.先算出最大数的位数;
    int max = array[0];
    for (int i = 1; i < array.length; i++) {
        max = Math.max(max, array[i]);
    }
    int maxDigit = 0;
    while (max != 0) {
        max /= 10;
        maxDigit++;
    }
    int mod = 10, div = 1;
    ArrayList<ArrayList<Integer>> bucketList = new ArrayList<ArrayList<Integer>>
();
    for (int i = 0; i < 10; i++)
```

```

        bucketList.add(new ArrayList<Integer>());
        for (int i = 0; i < maxDigit; i++, mod *= 10, div *= 10) {
            for (int j = 0; j < array.length; j++) {
                int num = (array[j] % mod) / div;
                bucketList.get(num).add(array[j]);
            }
            int index = 0;
            for (int j = 0; j < bucketList.size(); j++) {
                for (int k = 0; k < bucketList.get(j).size(); k++)
                    array[index++] = bucketList.get(j).get(k);
                bucketList.get(j).clear();
            }
        }
        return array;
    }
}

```

2.6. 计数排序

计数排序的核心在于将输入的数据值转化为键存储在额外开辟的数组空间中。作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数。

计数排序(Counting sort)是一种稳定的排序算法。计数排序使用一个额外的数组C，其中第i个元素是待排序数组A中值等于i的元素的个数。然后根据数组C来将A中的元素排到正确的位置。它只能对整数进行排序。

```

public static int[] CountingSort(int[] array) {
    if (array.length == 0) return array;
    int bias, min = array[0], max = array[0];
    for (int i = 1; i < array.length; i++) {
        if (array[i] > max)
            max = array[i];
        if (array[i] < min)
            min = array[i];
    }
    bias = 0 - min;
    int[] bucket = new int[max - min + 1];
    Arrays.fill(bucket, 0);
    for (int i = 0; i < array.length; i++) {
        bucket[array[i] + bias]++;
    }
    int index = 0, i = 0;
    while (index < array.length) {
        if (bucket[i] != 0) {
            array[index] = i - bias;
            bucket[i]--;
            index++;
        } else
            i++;
    }
    return array;
}
}

```

2.7. 快速排序

快速排序是由**冒泡排序**改进而得到的，是一种排序执行效率很高的排序算法，它利用**分治法**来对待排序序列进行分治排序，它的思想主要是通过一趟排序将待排记录分隔成独立的两部分，其中的一部分比关键字小，后面一部分比关键字大，然后再对这前后的两部分分别采用这种方式进行排序，通过递归的运算最终达到整个序列有序。

快速排序的过程如下：

1. 在待排序的N个记录中任取一个元素（通常取第一个记录）作为基准，称为基准记录；
2. 定义两个索引 left 和 right 分别表示首索引和尾索引，key 表示基准值；
3. 首先，尾索引向前扫描，直到找到比基准值小的记录，并替换首索引对应的值；
4. 然后，首索引向后扫描，直到找到比基准值大于的记录，并替换尾索引对应的值；
5. 若在扫描过程中首索引等于尾索引(left = right)，则一趟排序结束；将基准值(key)替换首索引所对应的值；
6. 再进行下一趟排序时，待排序列被分成两个区：[0,left-1]和[right+1,end]
7. 对每一个分区重复以上步骤，直到所有分区中的记录都有序，排序完成

快排为什么比冒泡效率高？

快速排序之所以比较快，是因为相比冒泡排序，每次的交换都是跳跃式的，每次设置一个基准值，将小于基准值的都交换到左边，大于基准值的都交换到右边，这样不会像冒泡一样每次都只交换相邻的两个数，因此比较和交换的此数都变少了，速度自然更高。

快速排序的平均时间复杂度是 $O(n\lg n)$ ，最坏时间复杂度是 $O(n^2)$ 。

```
public void quickSort(int[] arr) {
    if (arr == null) {
        return;
    }
    quickSortHelper(arr, 0, arr.length - 1);
}

private void quickSortHelper(int[] arr, int left, int right) {
    if (left > right) {
        return;
    }
    int tmp = arr[left];
    int i = left;
    int j = right;
    while (i < j) {
        //j先走，最终循环终止时，j停留的位置就是arr[left]的正确位置
        //改为i<=j，则会进入死循环，[1,5,5,5,5]->[1] 5 [5,5,5,5]->[5,5,5,5]，会死循环
        while (i < j && arr[j] >= tmp) {
            j--;
        }
        while (i < j && arr[i] <= tmp) {
            i++;
        }
        if (i < j) {
            int tmp1 = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp1;
        } else {
            break;
        }
    }
}
```

```

        //当循环终止的时候，i=j，因为是j先走的，j所在位置的值小于arr[left]，交换arr[j]和
arr[left]
        arr[left] = arr[j];
        arr[j] = tmp;

        quickSortHelper(arr, left, j - 1);
        quickSortHelper(arr, j + 1, right);
    }

```

2.8. 归并排序

归并排序 (merge sort) 是一类与插入排序、交换排序、选择排序不同的另一种排序方法。归并的含义是将两个或两个以上的有序表合并成一个新的有序表。归并排序有多路归并排序、两路归并排序，可用于内排序，也可以用于外排序。

两路归并排序算法思路是递归处理。每个递归过程涉及三个步骤

- 分解：把待排序的 n 个元素的序列分解成两个子序列，每个子序列包括 $n/2$ 个元素
- 治理：对每个子序列分别调用归并排序 MergeSort，进行递归操作
- 合并：合并两个排好序的子序列，生成排序结果

例如数组A有7个数据，分别是：49 38 65 97 76 13 27，那么采用归并排序算法

初始值 [49] [38] [65] [97] [76] [13] [27]

看成由长度为1的7个子序列组成

第一次合并之后 [38 49] [65 97] [13 76] [27]

看成由长度为1或2的4个子序列组成

第二次合并之后 [38 49 65 97] [13 27 76]

看成由长度为4或3的2个子序列组成

第三次合并之后 [13 27 38 49 65 76 97]

图6 归并排序算法过程图

时间复杂度：对长度为 n 的序列，需进行 $\log n$ 次二路归并，每次归并的时间为 $O(n)$ ，故时间复杂度是 $O(n \lg n)$ 。

空间复杂度：归并排序需要辅助空间来暂存两个有序子序列归并的结果，故其辅助空间复杂度为 $O(n)$

```

public class MergeSort {
    public void mergeSort(int[] arr) {
        if (arr == null || arr.length == 0) {
            return;
        }
        //辅助数组
        int[] tmpArr = new int[arr.length];
        mergeSort(arr, tmpArr, 0, arr.length - 1);
    }

    private void mergeSort(int[] arr, int[] tmpArr, int left, int right) {

```

```

    if (left < right) {
        int mid = (left + right) >> 1;
        mergeSort(arr, tmpArr, left, mid);
        mergeSort(arr, tmpArr, mid + 1, right);
        merge(arr, tmpArr, left, mid, right);
    }
}

private void merge(int[] arr, int[] tmpArr, int left, int mid, int right) {
    int i = left;
    int j = mid + 1;
    int tmpIndex = left;
    while (i <= mid && j <= right) {
        if (arr[i] < arr[j]) {
            tmpArr[tmpIndex++] = arr[i];
            i++;
        } else {
            tmpArr[tmpIndex++] = arr[j];
            j++;
        }
    }

    while (i <= mid) {
        tmpArr[tmpIndex++] = arr[i++];
    }

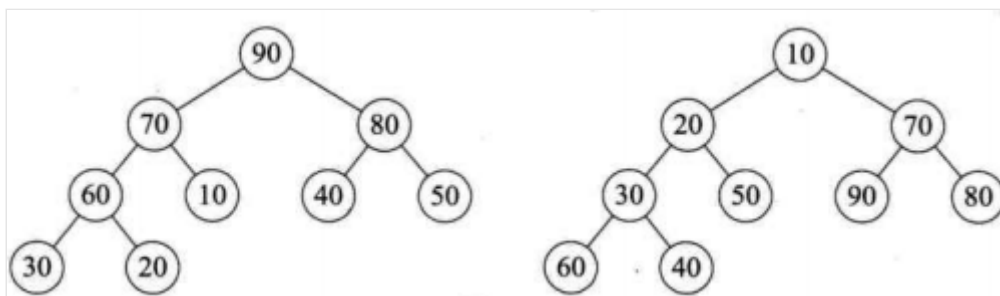
    while (j <= right) {
        tmpArr[tmpIndex++] = arr[j++];
    }

    for (int m = left; m <= right; m++) {
        arr[m] = tmpArr[m];
    }
}
}

```

2.9. 堆排序

堆是具有下列性质的完全二叉树：每个结点的值都大于或等于其左右孩子结点的值，称为大顶堆；或者每个结点的值都小于或等于其左右孩子结点的值，称为小顶堆。



Top大问题解决思路：使用一个固定大小的**最小堆**，当堆满后，每次添加数据的时候与堆顶元素比较，若小于堆顶元素，则舍弃，若大于堆顶元素，则删除堆顶元素，添加新增元素，对堆进行重新排序。

对于n个数，取Top m个数，时间复杂度为 $O(n \log m)$ ，这样在n较大情况下，是优于 $n \log n$ （其他排序算法）的时间复杂度的。

PriorityQueue 是一种基于优先级堆的优先级队列。每次从队列中取出的是具有最高优先权的元素。如果不提供Comparator的话，优先队列中元素默认按自然顺序排列，也就是数字默认是小的在队列头。优先级队列用数组实现，但是数组大小可以动态增加，容量无限。

```
//找出前k个最大数，采用小顶堆实现
public static int[] findKMax(int[] nums, int k) {
    PriorityQueue<Integer> pq = new PriorityQueue<>(k); //队列默认自然顺序排列，小顶堆，不必重写compare

    for (int num : nums) {
        if (pq.size() < k) {
            pq.offer(num);
        } else if (pq.peek() < num) { //如果堆顶元素 < 新数，则删除堆顶，加入新数入堆
            pq.poll();
            pq.offer(num);
        }
    }

    int[] result = new int[k];
    for (int i = 0; i < k && !pq.isEmpty(); i++) {
        result[i] = pq.poll();
    }
    return result;
}
```

3. 动态规划

动态规划常常适用于有重叠子问题的问题。动态规划的基本思想：若要解一个给定问题，我们需要解其不同部分（即子问题），再根据子问题的解以得出原问题的解。

动态规划法试图仅仅解决每个子问题一次，一旦某个给定子问题的解已经算出，则将其记忆化存储，以便下次遇到同一个子问题的时候直接查表得到解。

动态规划的解题思路：1、状态定义；2、状态转移方程；3、初始状态。

3.1. 不同路径

[不同路径](#)

3.2. 最长回文子串

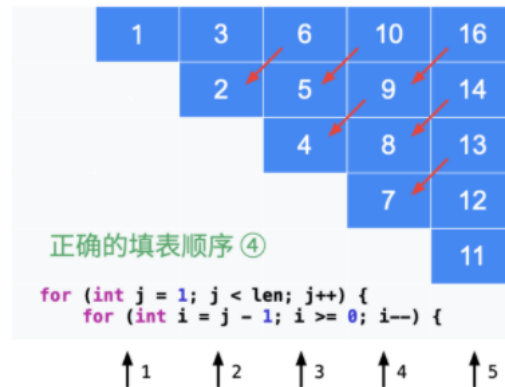
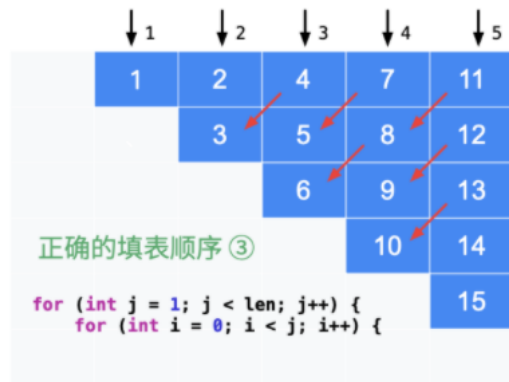
从给定的字符串 `s` 中找到最长的回文子串的长度。

例如 `s = "babbar"` 的最长回文子串是 `"abba"`，长度是 `4`。

解题思路：

1. 定义状态。`dp[i][j]` 表示子串 `s[i..j]` 是否为回文子串
2. 状态转移方程：`dp[i][j] = (s[i] == s[j]) and dp[i + 1][j - 1]`
3. 初始化的时候，单个字符一定是回文串，因此把对角线先初始化为 `true`，即 `dp[i][i] = true`。
4. 只要一得到 `dp[i][j] = true`，就记录子串的长度和起始位置

注意事项：总是先得到小子串的回文判定，然后大子串才能参考小子串的判断结果，即填表顺序很重要。



时间复杂度 $O(N^2)$ ，空间复杂度 $O(N^2)$ ，因为使用了二维数组。

```
public class Solution {

    public String longestPalindrome(String s) {
        // 特判
        int len = s.length();
        if (len < 2) {
            return s;
        }

        int maxLen = 1;
        int begin = 0;

        // dp[i][j] 表示 s[i, j] 是否是回文串
        boolean[][] dp = new boolean[len][len];
        char[] charArray = s.toCharArray();

        for (int i = 0; i < len; i++) {
            dp[i][i] = true;
        }
        for (int j = 1; j < len; j++) {
            for (int i = 0; i < j; i++) {
                if (charArray[i] != charArray[j]) {
                    dp[i][j] = false;
                } else {
                    if (j - i < 3) {
                        dp[i][j] = true;
                    } else {
                        dp[i][j] = dp[i + 1][j - 1];
                    }
                }
            }
        }

        // 只要 dp[i][j] == true 成立，就表示子串 s[i..j] 是回文，此时记录回文长度和起始位置
        if (dp[i][j] && j - i + 1 > maxLen) {
            maxLen = j - i + 1;
            begin = i;
        }
    }

    return s.substring(begin, begin + maxLen); //substring(i, j)截取i到j(不包含j)的字符串
}
```

```
}
```

3.3. 最大子数组和

题目描述：给你一个整数数组 `nums`，请你找出一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

示例：

输入： `nums = [-2,1,-3,4,-1,2,1,-5,4]`
输出： `6`
解释： 连续子数组 `[4,-1,2,1]` 的和最大，为 `6`。

1、首先确定dp数组（dp table）以及下标的含义。

`dp[i]`表示以`nums[i]`结尾的子数组的最大和。

2、确定递推公式。

`dp[i] = dp[i - 1] > 0 ? ( dp[i - 1] + nums[i]) : nums[i]`

`dp[i+1]`取决于`dp[i]`的值，不需要使用数组保存状态，只需要一个变量`sum`来保存上一个状态即可。

3、dp数组如何初始化。

从递推公式可以看出来`dp[i]`是依赖于`dp[i - 1]`的状态，`dp[0]`就是递推公式的基础。

`dp[0]`应该是多少呢？根据`dp[i]`的定义，很明显`dp[0]`应为`nums[0]`即`dp[0] = nums[0]`。

示例代码如下：

```
class Solution {
    public int maxSubArray(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int max = nums[0];
        int sum = 0;

        for (int num : nums) {
            if (sum > 0) {
                sum += num;
            } else {
                sum = num;
            }
            max = Math.max(max, sum);
        }

        return max;
    }
}
```

3.4. 最长公共子序列

一个字符串的 子序列 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。

动态规划。 $dp[i][j]$ 表示text1以i-1结尾的子串和text2以j-1结尾的子串的最长公共子序列的长度。dp横坐标或纵坐标为0表示空字符串， $dp[0][j] = dp[i][0] = 0$ ，无需额外处理base case。

		m					
		0	1	2	3	4	5
n	str2 \ str1	" "	a	b	c	d	e
	0	" "					
	1	a	1	1 (i,j)	1 (i,j+1)	1	1
	2	c	1	1 (i+1,j)	2 (i+1,j+1)	2	2
	3	e	1	1	2	2	3

```
class Solution {
    public int longestCommonSubsequence(String text1, String text2) {
        char[] arr1 = text1.toCharArray();
        char[] arr2 = text2.toCharArray();
        //dp[0][x]和dp[x][0]表示有一个为空字符串
        //dp[1][1]为text1第一个字符和text2第一个字符的最长公共子序列的长度
        //dp[i][j]表示text1以i-1结尾的子串和text2以j-1结尾的子串的最长公共子序列的长度
        int len1 = arr1.length;
        int len2 = arr2.length;
        int[][] dp = new int[len1 + 1][len2 + 1];

        for (int i = 1; i < len1 + 1; i++) {
            for (int j = 1; j < len2 + 1; j++) {
                if (arr1[i - 1] == arr2[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }

        return dp[len1][len2];
    }
}
```

`dp[i][j]` 表示text1以i结尾的子串和text2以j结尾的子串的最长公共子序列的长度。需要处理base case。

```
class Solution {
    public int longestCommonSubsequence(String text1, String text2) {
        char[] arr1 = text1.toCharArray();
        char[] arr2 = text2.toCharArray();

        int len1 = arr1.length;
        int len2 = arr2.length;
        // `dp[i][j]` 表示text1以i结尾的子串和text2以j结尾的子串的最长公共子序列的长度。
        int[][] dp = new int[len1][len2];

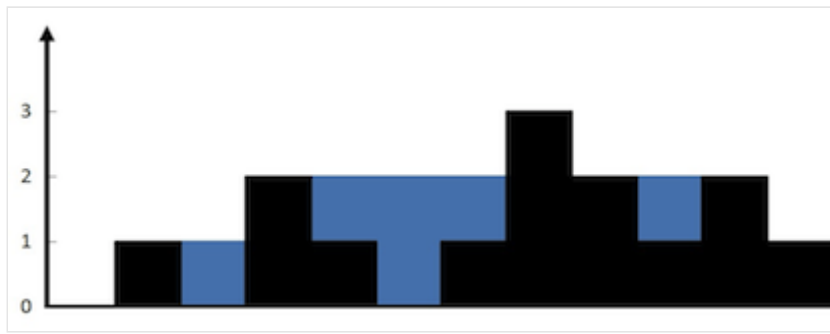
        if (arr1[0] == arr2[0]) {
            dp[0][0] = 1;
        }
        for (int i = 1; i < len1; i++) {
            if (arr1[i] == arr2[0]) {
                dp[i][0] = 1;
            } else {
                dp[i][0] = dp[i - 1][0];
            }
        }
        for (int i = 1; i < len2; i++) {
            if (arr1[0] == arr2[i]) {
                dp[0][i] = 1;
            } else {
                dp[0][i] = dp[0][i - 1];
            }
        }

        for (int i = 1; i < len1; i++) {
            for (int j = 1; j < len2; j++) {
                if (arr1[i] == arr2[j]) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }

        return dp[len1 - 1][len2 - 1];
    }
}
```

3.5. 接雨水

给定 `n` 个非负整数表示每个宽度为 `1` 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。



示例：

输入：height = [0,1,0,2,1,0,1,3,2,1,2,1]

输出：6

解释：上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。

动态规划，使用两个数组空间。leftMax[i] 代表第 i 列左边（不包含自身）最高的墙的高度，rightMax[i] 代表第 i 列右边最高的墙的高度。

```
class Solution {
    public int trap(int[] height) {
        int len = height.length;
        int res = 0;
        int[] leftMax = new int[len];
        int[] rightMax = new int[len];

        for (int i = 1; i < len; i++) {
            leftMax[i] = Math.max(leftMax[i - 1], height[i - 1]);
        }

        for (int j = len - 2; j > 0; j--) {
            rightMax[j] = Math.max(rightMax[j + 1], height[j + 1]);
        }

        for (int i = 1; i < len - 1; i++) {
            int min = Math.min(leftMax[i], rightMax[i]);
            if (min > height[i]) {
                res += min - height[i];
            }
        }

        return res;
    }
}
```

3.6. 单词拆分

题解

方法一：动态规划

""	l	e	e	t	c	o	d	e
TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	TRUE

1. 初始化 $dp = [False, \dots, False]$, 长度为 $n + 1$ 。 n 为字符串长度。 $dp[i]$ 表示 s 的前 i 位是否可以用 $wordDict$ 中的单词表示。
2. 初始化 $dp[0] = True$, 空字符可以被表示。
3. 遍历字符串的所有子串, 遍历开始索引 i , 遍历区间 $[0, n]$:
 - 遍历结束索引 j , 遍历区间 $[i + 1, n + 1]$:
 - 若 $dp[i] = True$ 且 $s[i, \dots, j)$ 在 $wordlist$ 中: $dp[j] = True$ 。解释: $dp[i] = True$ 说明 s 的前 i 位可以用 $wordDict$ 表示, 则 $s[i, \dots, j)$ 出现在 $wordDict$ 中, 说明 s 的前 j 位可以表示。
4. 返回 $dp[n]$

```
class solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        int len = s.length(), maxw = 0;
        //dp[i]表示前i个字母组成的字符串是否可以被拆分
        boolean[] dp = new boolean[len + 1];
        //状态转移方程初始化条件
        dp[0] = true;
        Set<String> set = new HashSet();
        for(String str : wordDict){
            set.add(str);
            maxw = Math.max(maxw, str.length());
        }
        for(int i = 1; i < len + 1; i++){
            for(int j = i; j >= 0 && j >= i - maxw; j--){
                if(dp[j] && set.contains(s.substring(j, i))){
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[len];
    }
}
```

4. 回溯算法

回溯算法的基本思想是：从一条路往前走，能进则进，不能进则退回来，换一条路再试。

4.1. 组合总和

题目描述：给定一个**无重复元素**的数组 `candidates` 和一个目标数 `target`，找出 `candidates` 中所有可以使数字和为 `target` 的组合。

示例：

```
输入: candidates = [2,3,6,7], target = 7,  
输出:  
[  
  [7],  
  [2,2,3]  
]
```

使用回溯算法。

```
class Solution {  
    private List<List<Integer>> ans = new ArrayList<>();  
    public List<List<Integer>> combinationSum2(int[] candidates, int target) {  
        if (candidates == null || candidates.length == 0) {  
            return ans;  
        }  
        Arrays.sort(candidates); // 排序方便回溯剪枝  
        Deque<Integer> path = new ArrayDeque<>(); // 作为栈来使用, 效率高于Stack; 也可以  
        // 作为队列来使用, 效率高于LinkedList; 线程不安全  
        combinationSum2Helper(candidates, target, 0, path);  
        return ans;  
    }  
  
    public void combinationSum2Helper(int[] arr, int target, int start,  
    Deque<Integer> path) {  
        if (target == 0) {  
            ans.add(new ArrayList(path));  
        }  
  
        for (int i = start; i < arr.length; i++) {  
            if (target < arr[i]) { // 剪枝  
                return;  
            }  
            if (i > start && arr[i] == arr[i - 1]) { // 在一个层级, 会产生重复  
                continue;  
            }  
            path.addLast(arr[i]);  
            combinationSum2Helper(arr, target - arr[i], i + 1, path);  
            path.removeLast();  
        }  
    }  
}
```

4.2. 全排列

给定一个 **没有重复** 数字的序列, 返回其所有可能的全排列。

示例:

输入: [1,2,3]

输出:

```
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]
```

使用回溯。注意与组合总和的区别（数字有无顺序）。

```
class Solution {
    private List<List<Integer>> ans = new ArrayList<>();
    public List<List<Integer>> permute(int[] nums) {
        boolean[] flag = new boolean[nums.length];
        ArrayDeque<Integer> path = new ArrayDeque<>();
        permuteHelper(nums, flag, path);

        return ans;
    }

    private void permuteHelper(int[] nums, boolean[] flag, ArrayDeque<Integer>
path) {
        if (path.size() == nums.length) {
            ans.add(new ArrayList<>(path));
            return;
        }
        for (int i = 0; i < nums.length; i++) {
            if (flag[i]) {
                continue; //继续循环
            }
            path.addLast(nums[i]);
            flag[i] = true;
            permuteHelper(nums, flag, path);
            path.removeLast();
            flag[i] = false;
        }
    }
}
```

4.3. 全排列II

给定一个可包含重复数字的序列，返回所有不重复的全排列。注意与组合总和的区别。

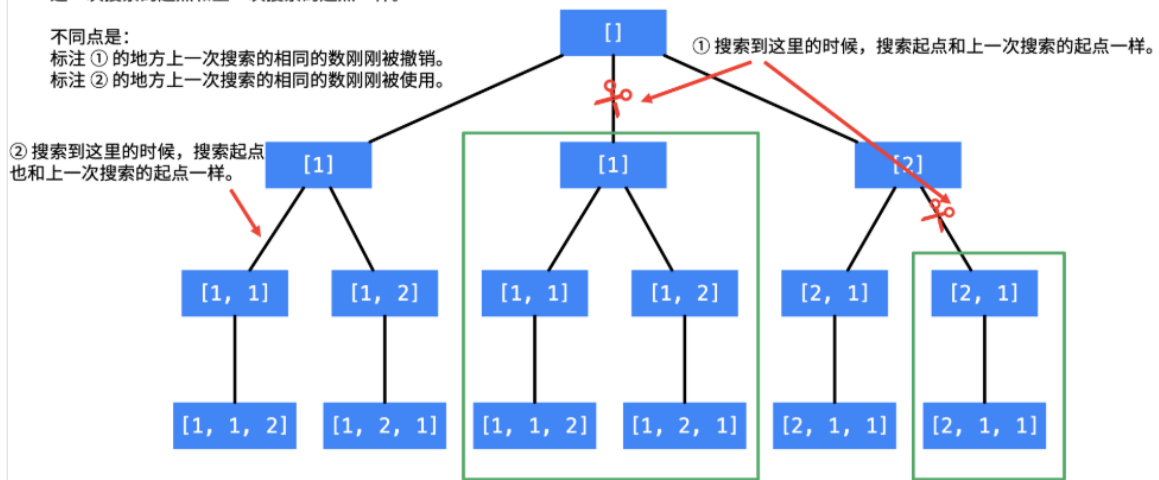
1、排序；2、同一层级相同元素剪枝。

LeetCode 第 47 题：“全排列 II” 题解配图 (1)

重点想象深度优先遍历在这棵树上执行的过程，哪些地方遍历下去一定会产生重复，这些地方的状态的特点是什么？对比图中标注 ① 和 ② 的地方。

相同点是：
这一次搜索的起点和上一次搜索的起点一样。

不同点是：
标注 ① 的地方上一次搜索的相同的数刚刚被撤销。
标注 ② 的地方上一次搜索的相同的数刚刚被使用。



```
class Solution {
    private List<List<Integer>> ans = new ArrayList<>();
    public List<List<Integer>> permuteUnique(int[] nums) {
        if (nums == null || nums.length == 0) {
            return ans;
        }
        ArrayDeque<Integer> path = new ArrayDeque<>();
        boolean[] used = new boolean[nums.length];
        Arrays.sort(nums); // 切记
        dps(nums, used, path);

        return ans;
    }

    private void dps(int[] nums, boolean[] used, ArrayDeque<Integer> path) {
        if (path.size() == nums.length) {
            ans.add(new ArrayList<>(path));
            return;
        }
        for (int i = 0; i < nums.length; i++) {
            if (used[i]) {
                continue;
            }
            if ((i > 0 && nums[i] == nums[i - 1]) && !used[i - 1]) { // 同一层相同的元素，剪枝
                continue; // 继续循环，不是return退出循环
            }
            path.addLast(nums[i]);
            used[i] = true;
            dps(nums, used, path);
            path.removeLast();
            used[i] = false;
        }
    }
}
```

5. 贪心算法

贪心算法，是寻找**最优解**问题的常用方法，这种方法模式一般将求解过程分成**若干个步骤**，但每个步骤都应用贪心原则，选取当前状态下**最好/最优的选择**（局部最有利的选择），并以此希望最后堆叠出的结果也是最好/最优的解。

贪婪法的基本步骤：

1. 从某个初始解出发；
2. 采用迭代的过程，当可以向目标前进一步时，就根据局部最优策略，得到一部分解，缩小问题规模；
3. 将所有解综合起来。

5.1. 买卖股票的最佳时机 II

题目描述：

给你一个整数数组 `prices`，其中 `prices[i]` 表示某支股票第 `i` 天的价格。

在每一天，你可以决定是否购买和/或出售股票。你在任何时候 最多 只能持有一股 股票。你也可以先购买，然后在 同一天 出售。

返回 你能获得的 最大 利润。

示例：

输入: `prices = [1,2,3,4,5]`

输出: 4

解释: 在第 1 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = $5 - 1 = 4$ 。

总利润为 4。

思路：可以尽可能地完成更多的交易，但不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

```
//输入: [7,1,5,3,6,4]
//输出: 7
class Solution {
    public int maxProfit(int[] prices) {
        int profit = 0;
        for (int i = 1; i < prices.length; i++) {
            int tmp = prices[i] - prices[i - 1];
            if (tmp > 0) {
                profit += tmp;
            }
        }

        return profit;
    }
}
```

5.2. 跳跃游戏

题目描述

给定一个非负整数数组 `nums`，你最初位于数组的 **第一个下标**。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个下标。

示例：

输入：nums = [2,3,1,1,4]

输出：true

解释：可以先跳 1 步，从下标 0 到达下标 1，然后再从下标 1 跳 3 步到达最后一个下标。

解题思路：

1. 如果某一个作为起跳点的格子可以跳跃的距离是 3，那么表示后面 3 个格子都可以作为起跳点
2. 可以对每一个能作为起跳点的格子都尝试跳一次，把能跳到最远的距离不断更新
3. 如果可以一直跳到最后，就成功了

```
class Solution {
    public boolean canJump(int[] nums) {
        if (nums == null || nums.length == 0) {
            return true;
        }

        int maxIndex = nums[0];
        for (int i = 1; i < nums.length; i++) {
            if (maxIndex >= i) {
                maxIndex = Math.max(maxIndex, i + nums[i]);
            } else {
                return false;
            }
        }

        return true;
    }
}
```

5.3. 加油站

题目描述

在一条环路上有 n 个加油站，其中第 i 个加油站有汽油 $gas[i]$ 升。

你有一辆油箱容量无限的汽车，从第 i 个加油站开往第 $i+1$ 个加油站需要消耗汽油 $cost[i]$ 升。你从其中的一个加油站出发，开始时油箱为空。

给定两个整数数组 gas 和 $cost$ ，如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。如果存在解，则保证它是唯一的。

示例

输入：gas = [1,2,3,4,5], cost = [3,4,5,1,2]

输出：3

解释：

从 3 号加油站(索引为 3 处)出发，可获得 4 升汽油。此时油箱有 $= 0 + 4 = 4$ 升汽油

开往 4 号加油站，此时油箱有 $4 - 1 + 5 = 8$ 升汽油

开往 0 号加油站，此时油箱有 $8 - 2 + 1 = 7$ 升汽油

开往 1 号加油站，此时油箱有 $7 - 3 + 2 = 6$ 升汽油

开往 2 号加油站，此时油箱有 $6 - 4 + 3 = 5$ 升汽油

开往 3 号加油站，你需要消耗 5 升汽油，正好足够你返回到 3 号加油站。

因此，3 可为起始索引。

思路：

1. 遍历一周，总获得的油量少于要花掉的油量必然没有结果；
2. 先苦后甜，记录遍历时所存的油量最少的站点，由于题目有解只有唯一解，所以从当前站点的下一个站点开始是唯一可能成功开完全程的。

```
class Solution {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        int minIdx=0;
        int sum=Integer.MAX_VALUE;
        int num=0;
        for (int i = 0; i < gas.length; i++) {
            num+=gas[i]-cost[i];
            if(num<sum){
                sum=num;
                minIdx=i;
            }
        }
        return num<0?-1:(minIdx+1)%gas.length;
    }
}
```

6. 双指针

6.1. 反转链表

题目描述

给你单链表的头节点 `head`，请你反转链表，并返回反转后的链表。

示例

输入：head = [1,2,3,4,5]
输出：[5,4,3,2,1]

思路：

1. 定义两个指针，第一个指针叫 pre，最初是指向 null 的。
2. 第二个指针 cur 指向 head，然后不断遍历 cur。
3. 每次迭代到 cur，都将 cur 的 next 指向 pre，然后 pre 和 cur 前进一位。
4. 都迭代完了(cur 变成 null 了)，pre 就是最后一个节点了。

```
class Solution {
    public ListNode reverseList(ListNode head) {
        if (head == null) {
            return null;
        }

        ListNode pre = null;
        ListNode cur = head;
        ListNode tmp = null;
        while (cur != null) {
            tmp = cur.next;
            cur.next = pre;
            pre = cur;
            cur = tmp;
        }
    }
}
```

```

    }

    return pre;
}
}

```

6.2. 反转链表II

题目描述

给你单链表的头指针 head 和两个整数 left 和 right，其中 $left \leq right$ 。请你反转从位置 left 到位置 right 的链表节点，返回 反转后的链表。

示例

输入: head = [1,2,3,4,5], left = 2, right = 4
输出: [1,4,3,2,5]

思路：双指针+头插法。

```

class Solution {
    public ListNode reverseBetween(ListNode head, int m, int n) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode pre = dummy;
        for (int i = 1; i < m; i++) {
            pre = pre.next;
        }
        head = pre.next;
        for (int i = m; i < n; i++) {
            ListNode cur = head.next;
            head.next = cur.next;
            cur.next = pre.next;
            pre.next = cur;
        }

        return dummy.next;
    }
}

```

6.3. 删除链表倒数第n个节点

题目描述

给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。

示例

输入: head = [1,2,3,4,5], n = 2
输出: [1,2,3,5]

思路：使用快慢指针，快指针先走n步。

```

class Solution {

```

```

public ListNode removeNthFromEnd(ListNode head, int n) {
    ListNode tmp = new ListNode(0); //技巧
    tmp.next = head;

    ListNode fast = tmp;
    ListNode slow = tmp;

    while (n-- > 0) {
        fast = fast.next;
    }

    while (fast.next != null) {
        fast = fast.next;
        slow = slow.next;
    }
    slow.next = slow.next.next;

    return tmp.next;
}
}

```

6.4. 三数之和

[题目链接](#)

题目描述

给你一个包含 n 个整数的数组 $nums$ ，判断 $nums$ 中是否存在三个元素 a, b, c ，使得 $a + b + c = 0$ ？请你找出所有和为 0 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例

输入： $nums = [-1, 0, 1, 2, -1, -4]$
 输出： $[[-1, -1, 2], [-1, 0, 1]]$

思路：

- 首先对数组进行排序，排序后固定一个数 $nums[i]$ ，再使用左右指针指向 $nums[i]$ 后面的两端，数字分别为 $nums[L]$ 和 $nums[R]$ ，计算三个数的和 sum 判断是否满足为 0，满足则添加进结果集
- 如果 $nums[i]$ 大于 0，则三数之和必然无法等于 0，结束循环
- 如果 $nums[i] == nums[i-1]$ ，则说明该数字重复，会导致结果重复，所以应该跳过
- 当 $sum == 0$ 时， $nums[L] == nums[L+1]$ 则会导致结果重复，应该跳过， $L++$
- 当 $sum == 0$ 时， $nums[R] == nums[R-1]$ 则会导致结果重复，应该跳过， $R--$

参考代码：

```

class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        Arrays.sort(nums);
    }
}

```

```

    for (int i = 0; i < nums.length; i++) {
        if (nums[i] > 0) { //最左边的数字大于0，则sum不会等于0，退出
            break;
        }
        if (i > 0 && nums[i] == nums[i - 1]) { //去重复
            continue;
        }

        int left = i + 1;
        int right = nums.length - 1;

        while (left < right) {
            int sum = nums[i] + nums[left] + nums[right];
            if (sum == 0) {
                res.add(Arrays.asList(nums[i], nums[left], nums[right]));
            }
            //array to list
            while (left < right && nums[left] == nums[left + 1]) {
                left++;
            }
            while (left < right && nums[right] == nums[right - 1]) {
                right--;
            }
            left++;
            right--;
        } else if (sum > 0) {
            right--;
        } else {
            left++;
        }
    }

    return res;
}

```

6.5. 环形链表

[题目链接](#)

题目描述

给你一个链表的头节点 head，判断链表中是否有环。

如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达，则链表中存在环。为了表示给定链表中的环，评测系统内部使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。注意：pos 不作为参数进行传递。仅仅是为了标识链表的实际情况。

如果链表中存在环，则返回 true。否则，返回 false。

示例

输入：head = [3,2,0,-4]，pos = 1

输出：true

解释：链表中有一个环，其尾部连接到第二个节点。

思路

快慢指针。快指针每次走两步，慢指针走一步，相当于慢指针不动，快指针每次走一步，如果是环形链表，则一定会相遇。

```
public class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null) {
            return false;
        }

        ListNode quick = head;
        ListNode slow = head;

        while (quick != null && quick.next != null) {
            slow = slow.next;
            quick = quick.next.next;

            if (slow == quick) {
                return true;
            }
        }

        return false;
    }
}
```

6.6. 环形链表II

[题目链接](#)

题目描述

给定一个链表的头节点 `head`，返回链表开始入环的第一个节点。如果链表无环，则返回 `null`。

如果链表中有某个节点，可以通过连续跟踪 `next` 指针再次到达，则链表中存在环。为了表示给定链表中的环，评测系统内部使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 `pos` 是 -1，则在该链表中没有环。注意：`pos` 不作为参数进行传递，仅仅是为了标识链表的实际情况。

不允许修改 链表。

示例

输入: `head = [3,2,0,-4]`, `pos = 1`
输出: 返回索引为 1 的链表节点
解释: 链表中有一个环，其尾部连接到第二个节点

解题思路

方法一：头结点到入环结点的距离为 a ，入环结点到相遇结点的距离为 b ，相遇结点到入环结点的距离为 c 。然后，当 `fast` 以 `slow` 的两倍速度前进并和 `slow` 相遇时，`fast` 走过的距离是 `slow` 的两倍，即有等式： $a+b+c+b=2(a+b)$ ，可以得出 $a=c$ ，所以说，让 `fast` 和 `slow` 分别从相遇结点和头结点同时同步长出发，他们的相遇结点就是入环结点。

```
public class Solution {
    public ListNode detectCycle(ListNode head) {
        ListNode fast = head;
        ListNode slow = head;
```

```

        while (true) {
            if (fast == null || fast.next == null) {
                return null;
            }
            fast = fast.next.next;
            slow = slow.next;
            if (fast == slow) {
                break;
            }
        }

        fast = head;

        while (slow != fast) {
            slow = slow.next;
            fast = fast.next;
        }

        return fast;
    }
}

```

方法二：先算出环的大小 n ，快指针先走 n 步，然后快慢指针一起走，相遇的地方即是环的入口。

```

public class Solution {
    public ListNode detectCycle(ListNode head) {
        ListNode slow = head;
        ListNode fast = head;
        //快慢指针找出环的大小
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (fast == slow) {
                break;
            }
        }

        if (fast == null || fast.next == null) {
            return null;
        }

        int cycleSize = 1;
        while (fast.next != slow) {
            cycleSize++;
            fast = fast.next;
        }

        //快慢指针重新从链表首部出发，快指针先走sizeOfCycle步
        //然后两个指针同时一起走，步长为1，相遇节点即是环的入口
        fast = head;
        slow = head;
        while (cycleSize-- > 0) {
            fast = fast.next;
        }
        while (fast != slow) {
            fast = fast.next;
            slow = slow.next;
        }
    }
}

```

```
    }  
    return fast;  
}  
}
```

大厂面经

大厂面试题专栏

1. 字节跳动抖音电商

面试官：看你简历上写了熟悉集合相关内容，你了解HashMap吗？讲一下HashMap的put方法？

独白：果然一上来就是 `HashMap` ...

大彬：`HashMap` 实现了Map接口，用于保存键值对映射。其底层是使用数组+链表+红黑树（JDK1.8增加了红黑树部分）实现的。

大彬：它的put方法过程如下：

1. 如果table没有初始化就先进行初始化过程
2. 使用hash算法计算key的索引
3. 判断索引处有没有存在元素，没有就直接插入
4. 如果索引处存在元素，则遍历插入，有两种情况，一种是链表形式就直接遍历到尾端插入，一种是红黑树就按照红黑树结构插入
5. 链表的数量大于阈值8，就要转换成红黑树的结构
6. 添加成功后会检查是否需要扩容

面试官：嗯，刚刚你提到HashMap的扩容，详细讲一下？

独白：emm，给自己挖坑了...

大彬：以JDK1.8为例，当往 `HashMap` 放入元素时，如果元素个数大于 `threshold` 时，会进行扩容，使用2倍容量的数组代替原有数组。

大彬：由于数组的容量是以2的幂次方扩容的，那么一个Entity在扩容时，新的位置要么在原位置，要么在原长度+原位置的位置。

大彬：原因是数组长度变为原来的2倍，表现在二进制上就是多了一个高位参与数组下标计算。

大彬：也就是说，在元素拷贝过程不需要重新计算元素在数组中的位置，只需要看看原来的hash值新增的那个bit是1还是0，是0的话索引没变，是1的话索引变成“原索引+oldCap”（根据 `e.hash & (oldCap - 1) == 0` 判断）。

大彬：这样可以省去重新计算hash值的时间，而且由于新增的1bit是0还是1可以认为是随机的，因此resize的过程会均匀的把之前的冲突的节点分散到新的bucket。

面试官：小伙子，基础还算不错。看你简历上写了精通MySQL，来讲讲一下MySQL的索引结构？

独白：卧槽，以后再也不敢写精通了....还好昨天背了大厂面试手册，现在一点都不慌，需要的可以后台回复【手册】获取

大彬：MySQL 数据库使用最多的索引类型是 `BTREE` 索引，底层基于B+树数据结构来实现。

大彬：B+ 树是基于B 树和叶子节点顺序访问指针进行实现，它具有B树的平衡性，并且通过顺序访问指针来提高区间查询的性能。

大彬：进行查找操作时，首先在根节点进行二分查找，找到key所在的指针，然后递归地在指针所指向的节点进行查找。直到查找到叶子节点，然后在叶子节点上进行二分查找，找出 key 所对应的数据项。

面试官：为什么索引要用B+树来实现呢，而不是用二叉树？

大彬：B+树有个特点，就是够矮够胖，能有效地减少访问节点次数从而提高性能。

大彬：虽然二叉树也有很好的查找性能 $\log_2 N$ ，但是当N比较大的时候，树的深度比较高。数据查询的时间主要依赖于磁盘IO的次数，二叉树深度越大，查找的次数越多，性能越差。最坏的情况会退化成链表。所以，B+树更适合作为MySQL索引结构。

面试官：那又为什么不用B树呢？

独白：现在面试也太卷了叭，这是要造火箭啊...

大彬：因为B树的分支结点存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫。而由于B+树的数据都存储在叶子结点中，叶子结点均为索引，方便扫库，只需要扫一遍叶子结点即可。所以B+树更加适合在区间查询的情况，而在数据库中基于范围的查询是非常频繁的，所以B+树更适合用于数据库索引。

面试官：知道聚集索引吗？

大彬：聚集索引严格来说并不是索引类型，而是一种数据存储方式，具体细节依赖于其实现方式。如innodb聚集索引的叶子节点存放了整张表的行记录。

大彬：聚集索引类似字典的拼音目录。表中的数据按照聚集索引的规则来存储的。就像新华字典，整本字典是按照A-Z的顺序来排列。这也是一个表只能有一个聚集索引的原因。

面试官：那聚簇索引相比非聚簇索引有什么优点？

大彬：1. 数据访问更快，因为聚簇索引将索引和数据保存在同一个B+树中，因此从聚簇索引中获取数据比非聚簇索引更快。**大彬：**2. 聚集索引叶子节点的存储是逻辑上连续的，所以对于主键的排序查找和范围查找速度会更快。

面试官：嗯，问点其他的，线程池知道吧？

大彬：线程池，顾名思义，就是一个管理线程的池子。

面试官：那为什么使用线程池呢？

大彬：之所以使用线程池，主要有三点原因：

- **降低资源消耗。**通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- **提高响应速度。**当任务到达时，可以不需要等到线程创建就能立即执行。
- **提高线程的可管理性。**统一管理线程，避免系统创建大量同类线程而导致消耗完内存。

面试官：嗯，那你讲一下线程池的几个参数？

独白：老八股文了嘿嘿~

大彬：先来看看 `ThreadPoolExecutor` 的通用构造函数：

```
public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long
    keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory
    threadFactory, RejectedExecutionHandler handler);
```

大彬：其中有7个参数。分别是 `corePoolSize`，`maximumPoolSize`，`keepAliveTime`，`timeUnit`，`workQueue`，`threadFactory`，`rejectedExecutionHandler`

大彬：`corePoolSize`。当有新任务时，如果线程池中线程数没有达到线程池的基本大小，则会创建新的线程执行任务，否则将任务放入阻塞队列。当线程池中存活的线程数总是大于 `corePoolSize` 时，应该考虑调大 `corePoolSize`。

大彬：`maximumPoolSize`。当阻塞队列填满时，如果线程池中线程数没有超过最大线程数，则会创建新的线程运行任务。否则根据拒绝策略处理新任务。非核心线程类似于临时借来的资源，这些线程在空闲时间超过 `keepAliveTime` 之后，就应该退出，避免资源浪费。

大彬：`BlockingQueue`。存储等待运行的任务。

大彬： `keepAliveTime`。非核心线程空闲后，保持存活的时间，此参数只对非核心线程有效。设置为0，表示多余的空闲线程会被立即终止。

大彬： `TimeUnit`。时间单位，具体如下：

```
TimeUnit.DAYS
TimeUnit.HOURS
TimeUnit.MINUTES
TimeUnit.SECONDS
TimeUnit.MILLISECONDS
TimeUnit.MICROSECONDS
TimeUnit.NANOSECONDS
```

大彬： `ThreadFactory`。每当线程池创建一个新的线程时，都是通过线程工厂方法来完成的。在 `ThreadFactory` 中只定义了一个方法 `newThread`，每当线程池需要创建新线程就会调用它。

```
public class MyThreadFactory implements ThreadFactory {
    private final String poolName;

    public MyThreadFactory(String poolName) {
        this.poolName = poolName;
    }

    public Thread newThread(Runnable runnable) {
        return new MyAppThread(runnable, poolName); //将线程池名字传递给构造函数，用于
        区分不同线程池的线程
    }
}
```

大彬： `RejectedExecutionHandler`。当队列和线程池都满了时，根据拒绝策略处理新任务。

`AbortPolicy`：默认的策略，直接抛出 `RejectedExecutionException`
`DiscardPolicy`：不处理，直接丢弃
`DiscardOldestPolicy`：将等待队列队首的任务丢弃，并执行当前任务
`CallerRunsPolicy`：由调用线程处理该任务

面试官：好的。你了解 Spring AOP 吗？

大彬： AOP，其实就是面向切面编程，将一些公共逻辑（事务管理、日志、缓存等）封装成切面，跟业务代码进行分离，可以减少系统的重复代码和降低模块之间的耦合度。切面就是那些与业务无关，但所有业务模块都会调用的公共逻辑。

大彬： Spring AOP是通过动态代理技术实现的。

面试官：哦，那动态代理的实现方式有哪些？

大彬： 动态代理技术的实现方式有两种：

1. 基于接口的 JDK 动态代理。
2. 基于继承的 CGLib 动态代理。在 Spring 中，如果目标类没有实现接口，那么 Spring AOP 会选择使用 CGLIB 来动态代理目标类。

面试官：你刚刚提到 CGLib 动态代理，能详细介绍下吗？

大彬： CGLIB，就是 `Code Generator Library`，它是一个强大的、高性能的代码生成库，被广泛应用于 AOP 框架中，用以提供方法拦截操作。

大彬：CGLIB代理主要通过对字节码的操作，为对象引入间接级别，以控制对象的访问。

大彬：CGLIB 动态代理相对于 JDK 动态代理局限性就小很多，目标对象不需要实现接口，底层是通过**继承目标对象产生代理子对象**。

面试官：来做道题吧。求根节点到叶节点数字之和。

独白：没有困难的题目，只有勇敢的刷题人！

大彬：思路：深度优先搜索。从根节点开始，遍历每个节点，如果遇到叶子节点，则将叶子节点对应的数字加到数字之和。如果当前节点不是叶子节点，则计算其子节点对应的数字，然后对子节点递归遍历。

```
// 输入: [1,2,3]
//      1
//     / \
//    2   3
// 输出: 25
class Solution {
    public int sumNumbers(TreeNode root) {
        if (root == null) {
            return 0;
        }

        return sumNumbersHelper(root, 0);
    }

    private int sumNumbersHelper(TreeNode node, int sum) {
        if (node == null) {
            return 0;
        }
        if (sum > Integer.MAX_VALUE / 10 || (sum == Integer.MAX_VALUE / 10 &&
node.val > Integer.MAX_VALUE % 10)) {
            throw new IllegalArgumentException("exceed max int value");
        }
        sum = sum * 10 + node.val;

        if (node.left == null && node.right == null) {
            return sum;
        }

        return sumNumbersHelper(node.right, sum) + sumNumbersHelper(node.left,
sum);
    }
}
```

面试官：不错，好好准备二面吧~

2. 美团二面

面试官：首先你来讲讲进程和线程有什么区别？

大彬：进程是系统进行资源分配和调度的独立单位，每一个进程都有自己的内存空间和系统资源

大彬：线程是进程的一个实体，是CPU调度和分派的基本单位，它是比进程更小的能独立运行的基本单位

大彬：多线程是实现并发机制的一个有效手段。进程和线程一样都是实现并发的基本单位

面试官：那为什么要用多线程呢？

大彬：使用多线程最主要的原因是提高系统的资源利用率。

大彬：多个线程同时运行，可以减少线程上下文切换的开销，提高并发的能力和CPU的利用效率。

大彬：在平时工作中多线程也是常见的。比如Tomcat每处理一个请求都会从线程连接池里取一个线程去处理。

面试官：嗯，平时在使用多线程的时候，可能会遇到线程安全的问题吧。讲讲什么是线程安全？

大彬：我是这么理解的，当多个线程访问一个对象时，如果不用考虑这些线程在运行时环境下的调度和交替执行，也不需要进行额外的同步，调用这个对象的行为都可以获得正确的结果，那这个对象就是线程安全的。

面试官：那你平时怎么处理线程安全问题的？

大彬：这个还得具体问题具体分析。首先判断有没有线程安全问题，若有则根据具体的情况去处理线程安全的问题。

大彬：比如涉及到操作的原子性，可以考虑使用 `atomic` 包下的原子类。

大彬：如果涉及到对线程的控制，可以考虑线程工具类 `CountDownLatch` / `Semaphore` 等等。

大彬：集合类的话，考虑 `java.util.concurrent` 包下的集合类。

大彬：还有 `synchronized` 和 `lock` 包下的类，redis分布式锁等。

面试官：嗯哼，刚提到Redis分布式锁，你觉得什么场景下需要使用分布式锁呢？

大彬：在单机环境下，线程安全问题可以通过 `ReentrantLock`、`synchronized` 以及 `concurrent` 并发包下一些线程安全的类等来避免。

大彬：而在多机部署环境，需要在多进程下保证线程的安全性，Java提供的这些API仅能保证在单个JVM进程内对多线程访问共享资源的线程安全，已经不满足需求了。这时候就需要使用分布式锁来保证线程安全。

大彬：Redis 2.6.12 之前的版本中采用 `setnx + expire` 方式实现分布式锁。在 Redis 2.6.12 版本后 `setnx` 增加了过期时间参数，只需要使用 `setnx` 就可以实现分布式锁了。

面试官：那再讲讲Redis分布式锁的原理？

大彬：首先介绍下Redis的加锁逻辑。

大彬：`setnx` 争抢key的锁，如果已有key存在，则不作操作，过段时间继续重试，保证只有一个客户端能持有锁。

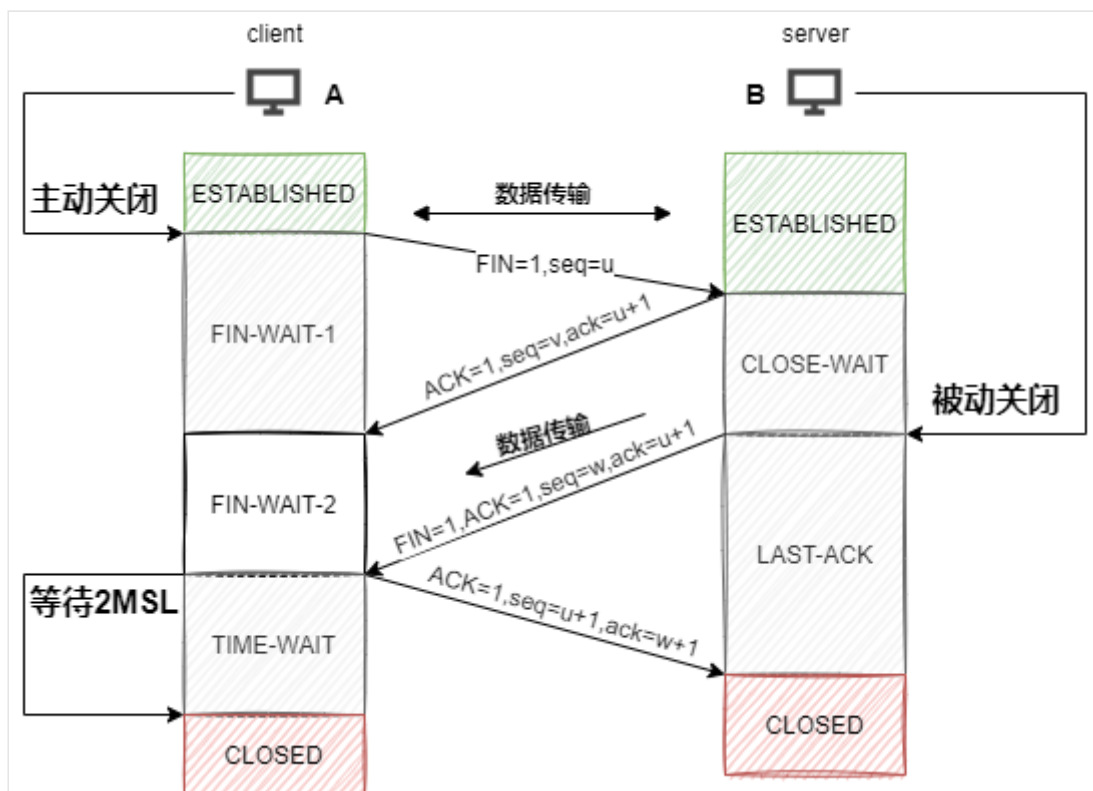
大彬：value设置为 `requestId`（可以使用机器ip拼接当前线程名称），表示这把锁是哪个请求加的，在解锁的时候需要判断当前请求是否持有锁，防止误解锁。比如客户端A加锁，在执行解锁之前，锁过期了，此时客户端B尝试加锁成功，然后客户端A再执行 `del()` 方法，则将客户端B的锁给解除了。

大彬：再用 `expire` 给锁加一个过期时间，防止异常导致锁没有释放。

大彬：然后是解锁逻辑。

大彬：首先获取锁对应的value值，检查是否与 `requestId` 相等，如果相等则删除锁。使用lua脚本实现原子操作，保证线程安全。

面试官：不错，看你简历上写了熟悉TCP，来介绍下TCP四次挥手？



大彬：假设A是client端，B是server端。

1. 首先A的应用进程先向其TCP发出连接释放报文段（**FIN=1, seq=u**），并停止再发送数据，主动关闭TCP连接，进入 **FIN-WAIT-1**（终止等待1）状态，等待B的确认。
2. B收到连接释放报文段后即发出确认报文段（**ACK=1, seq=v, ack=u+1**），B进入 **CLOSE-WAIT**（关闭等待）状态，此时的TCP处于半关闭状态，A到B的连接释放。
3. A收到B的确认后，进入 **FIN-WAIT-2**（终止等待2）状态，等待B发出的连接释放报文段。
4. B发送完数据，就会发出连接释放报文段（**FIN=1, ACK=1, seq=w, ack=u+1**），B进入 **LAST-ACK**（最后确认）状态，等待A的确认。
5. A收到B的连接释放报文段后，对此发出确认报文段（**ACK=1, seq=u+1, ack=w+1**），A进入 **TIME-WAIT**（时间等待）状态。此时TCP未释放掉，需要经过时间等待计时器设置的时间 **2MSL**（最大报文段生存时间）后，A才进入 **CLOSED** 状态。B收到A发出的确认报文段后关闭连接，若没收到A发出的确认报文段，B就会重传连接释放报文段。

面试官：建立连接时三次握手，为什么连接释放要四次挥手，三次不行吗？

大彬：因为建立连接时，当Server端收到Client端的 **SYN** 连接请求报文后，可以直接发送 **SYN+ACK** 报文。

大彬：但是在关闭连接时，当Server端收到Client端发出的连接释放报文时，很可能并不会立即关闭 **SOCKET**，所以Server端先回复一个 **ACK** 报文，告诉Client端我收到你的连接释放报文了。只有等到Server端所有的报文都发送完了，这时Server端才能发送连接释放报文，之后两边才会真正的断开连接。故需要四次挥手。

面试官：嗯，你了解https吗？https是为了解决什么问题？

独白：一点也不慌哈哈

大彬：HTTP是明文传输，容易被黑客窃听或篡改，不安全。

大彬：HTTPS 主要解决了 HTTP 明文协议的缺陷，在 HTTP 的基础上加入 SSL/TLS 协议，依靠 SSL 证书来验证服务器的身份，为客户端和服务端之间建立SSL通道，确保数据传输安全。

面试官：那http跟https具体有什么区别呢？

大彬：http和https的区别如下：

1. HTTP是超文本传输协议，信息是**明文传输**；HTTPS则是具有**安全性**的ssl加密传输协议。
2. HTTP和HTTPS用的端口不一样，HTTP端口是80，HTTPS是443。
3. HTTPS协议**需要到CA机构申请证书**，一般需要一定的费用。
4. HTTP运行在TCP协议之上；HTTPS运行在SSL协议之上，SSL运行在TCP协议之上。

面试官：不错，再来问点MySQL相关的

面试官：什么情况下索引会失效？

大彬：主要有这么几种情况会导致索引失效。

- 对于组合索引，不是使用组合索引最左边的字段，则不会使用索引
- 以%开头的like查询如 **%abc**，无法使用索引；非%开头的like查询如 **abc%**，相当于范围查询，会使用索引
- 查询条件中列类型是字符串，没有使用引号，可能会因为类型不同发生隐式转换，使索引失效
- 判断索引列是否不等于某个值时
- 对索引列进行运算
- 查询条件使用 **or** 连接，也会导致索引失效



扫码关注我



微信搜索



程序员大彬

公众号后台回复【面试】获取面试手册
PDF最新版

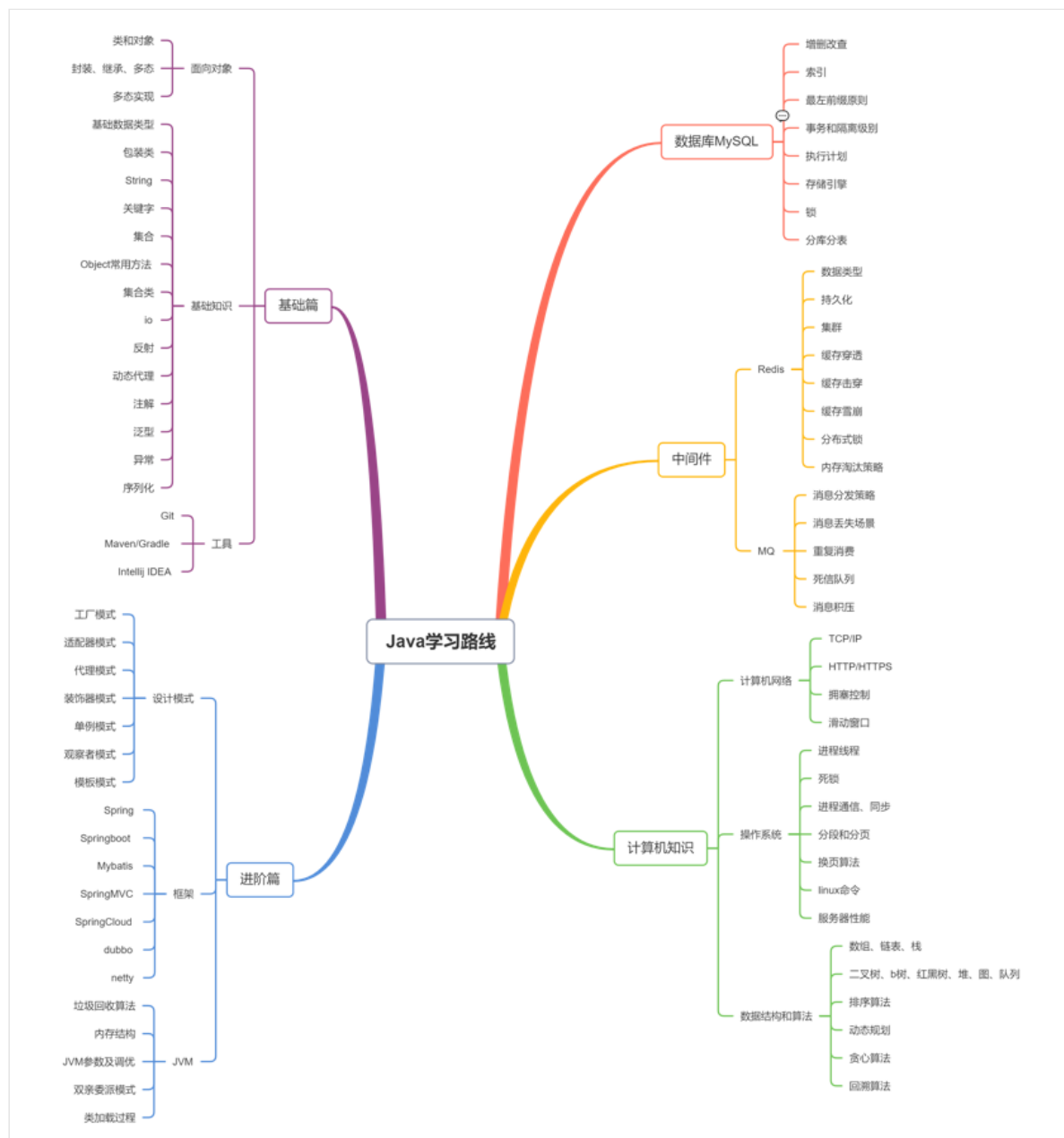
Java自学路线

本人211非科班，大学学的物理，大三开始自学JAVA，并成功拿到了几个中大厂的offer。自学路上也遇到不少坑，在这里整理了一些学习过程中踩坑总结的经验，希望自学的小伙伴可以少走弯路：

- **注重实践**，不要只是埋头看书，一定要多动手写代码。
- 刚开始自学的时候，可以不用太深究细节，不然可能会怀疑自己的学习能力。等到后面有了一定的基础，回过头来重新回顾，可能会恍然大悟，没有当初想的那么难。
- 可以适当加一些交流群，遇到不懂的知识点，多与其他人交流。

下面是Java学习路线图：

公众号【程序员大彬】后台回复【自学路线】下载高清图



内容还是挺多的，不过**掌握好学习方法，学好Java并不难！**

1. Java

刚开始入门看的马士兵老师的JAVA基础视频，后面发现看视频效率较低，于是抛弃看视频学习的方式，转向看书（每个人学习方式不一样，个人觉得看视频学习，不好做笔记，而且经常看了就忘，所以后期自学基本以看书为主）。

推荐书籍：

- 《head first java》
- 《JAVA核心技术卷》

head first系列的书籍讲解比较有趣，比较好理解。《JAVA核心技术卷》难度相对适中，内容也比较全面，部分章节（如Swing）可以跳过。

Java视频学习路线可以参考我的另一个回答：<https://www.zhihu.com/question/433615385/answer/2244848102>

再次强调：**多敲代码！多敲代码！多敲代码！**

学习编程就是看书加实践，要多动手，不然看过的知识点很快就会忘，而且多实践也会遇到很多坑，丰富经验。可以到github上找一些项目练练手，通过做项目巩固知识，而且每实现一个功能之后，会有满满的成就感，也会激励你不断去学习。

Java基础知识主要有：

- 类与对象
- 面向对象特性
- 抽象类和接口
- 数据类型、重写重载、封装继承多态
- 容器类Map/List/Set等
- IO操作：InputStream，OutputStream，Reader/Writer，文件读取，各种流读取等
- 异常处理
- 反射机制
- 泛型

2. Java Web

Java Web是一系列技术的综合，也是大多数Java开发者的技术方向。有必要学习一下。这部分可以看看视频教程。

视频推荐尚硅谷的JavaWeb全套教程，HTML/CSS/JavaScript等跟前端相关的可以倍速观看。

<https://www.bilibili.com/video/BV1Y7411K7zz>

黑马程序员的java web教程总体也不错，IDEA版本的：<https://www.bilibili.com/video/BV1qv4y1o79t>

3. 数据库

数据库也是必学的一块内容，主要是MySQL数据库。书籍推荐：

- 《MySQL必知必会》
- 《高性能mysql》

《MySQL必知必会》主要是Mysql的基础语法，很好理解。后面有了基础再看《高性能mysql》，这本书主要讲解索引、SQL优化、高级特性等，很多Mysql相关面试题出自《高性能MySQL》这本书，值得一看。

MySQL基础知识：

- 增删改查
- 事务特性、隔离级别
- 索引原理、优化
- b+树
- 最左匹配原则
- 存储引擎

- MVCC
- 执行计划
- 分库分表
- 日志, bin log/undo log/redo log
- ...

4. 框架

主流框架主要有：

- spring：面向切面、依赖注入。
- springboot：习惯优于配置、自动配置。目前很多公司内部都是使用Spring Boot。
- springmvc：基于MVC架构模式的轻量级Web框架
- Mybatis：orm框架。
- springcloud

4.1. Spring

大部分公司都会用到 Spring框架，必学！。主要理解 Spring 面向切面、依赖注入的特性，学会使用 Spring 构建应用程序。推荐书籍《Spring实战》，通过demo的方式带你一步步搭建Spring应用

4.2. SpringMVC

SpringMVC是基于**MVC架构模式**的轻量级Web框架，对于初学者，需要掌握Web请求从发出到相应的这个过程，SpringMVC做了什么，还有MVC模式的思想。

4.3. Mybatis

MyBatis 是一款优秀的持久层框架，MyBatis 帮助我们做了很多事情：建立连接、操作 Statment、ResultSet、处理 JDBC 相关异常等，简化了开发流程。推荐书籍《深入浅出Mybatis》。

4.4. SpringBoot

学完 SSM，就要进一步学习 SpringBoot 了，相信很多人在学了 Spring 之后，面对各种各样的配置，想必都会头疼。而 SpringBoot 的出现解决了这个问题，SpringBoot 去除了大量的 XML 配置文件，简化了复杂的依赖管理。书籍推荐《Spring Boot实战》。

4.5. SpringCloud

现在面试基本都会问到微服务相关的内容，最好了解下微服务相关的知识。服务注册与发现、负载均衡、服务降级、API网关等。推荐书籍《spring cloud微服务实战》

5. 中间件

- redis：用来缓存热点数据，加快读写速度，从而提高性能。推荐书籍《redis实战》和《redis设计与实现》
- rabbitMQ：消息队列是基础数据结构中 **FIFO** 的一种数据结构，用来解决应用解耦、异步消息、流量削峰等问题，可以实现高性能、高可用、可伸缩和最终一致性。
- kafaka：一种分布式的，基于发布/订阅的消息系统。

6. JVM

jvm也是面试经常会问的内容。Java开发者不用自己进行内存管理、垃圾回收，jvm帮我们做了，但是还是有必要了解下jvm的工作原理，这样在出现oom等问题的时候，才有思路去排查和解决问题。书籍推荐周老师的《深入理解Java虚拟机》。

jvm的基础知识：

- jvm内存结构（程序计数器、虚拟机栈、本地方法栈、堆、方法区、运行时常量池、直接内存）
- 类加载过程
- 双亲委派
- 垃圾回收算法
- 垃圾回收器
- 调优工具（jstack/jstat/jmap，了解即可）

7. 并发

并发编程的相关内容可以看看《JAVA并发编程实战》这本书，主要内容有：

- 线程池原理
- 线程间通信方式
- 锁（synchronized、ReentrantLock）
- 并发工具类（CountDownLatch/CyclicBarrier/Semaphore）
- 原子类
- AQS

8. 计算机基础知识

平时抽空学习计算机基础知识，巩固基础。

8.1. 数据结构和算法

推荐书籍《数据结构与算法分析 java语言描述》，用java语言描述各种数据结构和算法，对于Java开发者来说，更容易理解，主要学习以下内容：

- 基本数据结构（数组、链表、栈、队列等）
- 树（二叉树、avl树、b树、红黑树等）
- 堆结构
- 排序算法（冒泡排序、选择排序、插入排序、快速排序、归并排序、堆排序等及时间空间复杂度）
- 动态规划、回溯、贪心算法（多刷刷leetcode）

8.2. 计算机网络

推荐《计算机网络自顶向下方法》：

- 网络分层结构
- TCP/IP
- 三次握手四次挥手
- 滑动窗口、拥塞控制
- HTTP/HTTPS
- 网络安全问题（CSRF、XSS、SQL注入等）

8.3. 操作系统

学起来比较头大，可以到b站找找视频看，推荐[清华大学的公开课](#)。书籍推荐《深入理解计算机系统》，大部头，经典书籍，值得多读几遍。

核心知识点：

- 进程和线程
- 内存管理
- 文件管理

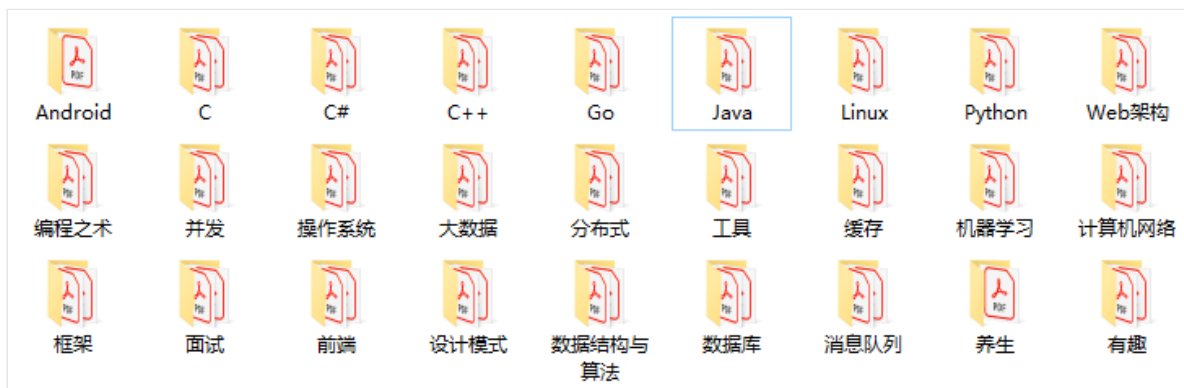
- I/O
- 死锁
- ...

8.4. linux

要熟悉常用的Linux命令。推荐书籍《鸟哥的linux私房菜》。

上面提到的书籍，我已经整理了电子版，放到github和gitee上了，在这里分享有需要的人，希望可以帮助到曾经像我一样的新手，节省找资料的时间（花了一个多月的时间整理的，希望对大家有帮助，欢迎star~）

仓库持续更新中~



小伙伴们有需要的自取：

github仓库：<https://github.com/Tyson0314/java-books>

如果github访问不了，可以访问gitee仓库。

gitee仓库：<https://gitee.com/tysondai/java-books>



扫码关注我



微信搜索



程序员大彬

公众号后台回复【面试】获取面试手册
PDF最新版