

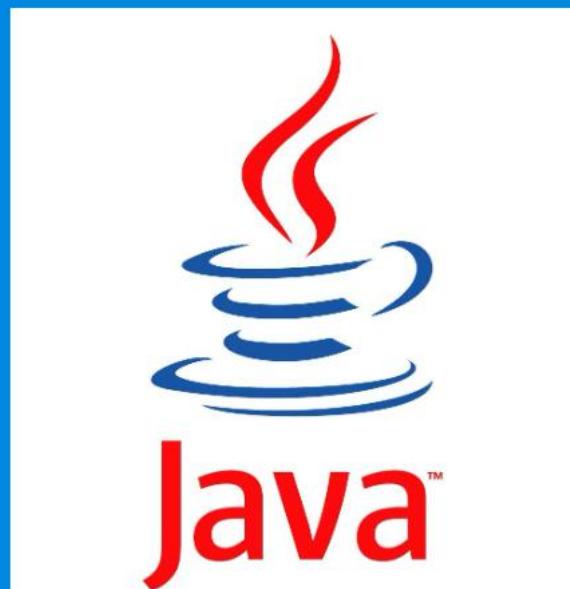
2021

# Java 大厂面试真题

最新汇总

内含800道面试真题

整理人：极客时间训练营教研部



# 目录

.....	1
<b>大厂面试的基本流程</b> .....	<b>16</b>
字节跳动.....	16
阿里.....	16
腾讯.....	16
网易游戏.....	17
<b>面试前需要准备:</b> .....	<b>17</b>
<b>JAVA 进阶训练营</b> .....	<b>18</b>
<b>2021 【美团】面试真题:</b> .....	<b>22</b>
1、SPRING AOP 底层原理.....	22
2、HASHMAP 的底层数据结构是怎样的 ? .....	22
3、HASHMAP 的扩容机制是怎样的? .....	23
4、CONCURRENTHASHMAP 的存储结构是怎样的? .....	23
5、线程池大小如何设置? .....	24
6、IO 密集=NCPU*2 是怎么计算出来? .....	24
7、G1 收集器有哪些特点? .....	24
8、你有哪些手段来排查 OOM 的问题? .....	25
9、请你谈谈 MYSQL 事务隔离级别, MYSQL 的默认隔离级别是什么? .....	25
10、可重复读解决了哪些问题? .....	25
11、对 SQL 慢查询会考虑哪些优化 ? .....	25
12、谈一谈缓存穿透、缓存击穿和缓存雪崩, 以及解决办法? .....	26
13、LRU 是什么? 如何实现? .....	27
14、什么是堆内存? 参数如何设置? .....	27
15、栈和队列, 举个使用场景例子? .....	27
16、MYSQL 为什么 INNODB 是默认引擎? .....	27
17、MYSQL 索引底层结构为什么使用 B+树? .....	28
18、B+ 树的叶子节点链表是单向还是双向? .....	28
19、MVCC 是什么? 它的底层原理是什么? .....	28
20、UNDO LOG 具体怎么回滚事务 ? .....	28
21、如何查询慢 SQL 产生的原因.....	29
22、索引失效的情况有哪些? .....	29
23、一个 REDIS 实例最多能存放多少的 KEYS? LIST、SET、SORTED SET 他们最多能存放多少元素? .....	29
24、REDIS 数据结构 压缩列表和跳跃表的区别.....	29
25、为什么数据量小的时候用压缩列表 ?.....	30
26、REDIS 主从同步是怎么实现的? .....	30
27、REDIS 持久化 RDB 和 AOF 优缺点.....	30
28、谈谈自己对于 SPRING AOP 的了解? .....	31
29、SPRING BEAN 容器的生命周期是什么样的? .....	31
30、RABBITMQ 如何保证消息不丢失 ? .....	32
<b>2021 【阿里】面试真题:</b> .....	<b>32</b>
1、TCP 和 UDP 区别? .....	32
2、TCP/IP 协议涉及哪几层架构? .....	32
3、描述下 TCP 连接 4 次挥手的过程? 为什么要 4 次挥手? .....	33
4、计算机插上电源操作系统做了什么? .....	33
5、LINUX 操作系统设备文件有哪些? .....	33
6、多线程同步有哪些方法? .....	33
7、一个对象的两个方法加 SYNCHRONIZED, 一个线程进去 SLEEP, 另一个线程可以进入到另一个方法吗? .....	33
8、什么是可重入锁 (REENTRANTLOCK) ? .....	33

9、创建线程的三个方法是什么? .....	34
10、JAVA 怎么获取多线程的返回值? .....	34
11、线程池有哪几种创建方式? .....	34
12、线程池参数有哪些? .....	35
13、线程池拒绝策略有哪些? .....	35
14、你认为对线程池的核心参数实现自定义可配置，三个核心参数是? .....	35
15、THREADPOOLEXECUTOR 线程池，COREPOOLSIZE=5，MAXIMUMPOOLSIZE=10，QUEUECAPACITY=10，有 20 个耗时任务交给这个线程池执行，线程池会如何执行这 20 个任务? .....	35
16、给用户发消息任务超出队列，你用哪个拒绝策略？有其他方法吗？ .....	36
17、JAVA8 新特性有哪些了解?.....	36
18、什么时候用多线程、为什么要设计多线程? .....	36
19、多线程越多效率越高吗? .....	36
20、多线程会产生哪些并发问题 ?.....	37
21、MYBATIS 如何将对象转换成 SQL? .....	37
22、虚拟内存是什么，虚拟内存的原理是什么? .....	37
23、栈会溢出吗？什么时候溢出？方法区会溢出吗? .....	37
24、JVM 如何加载类的? .....	38
25、自己写过 STRING 类能加载吗，之前的 STRING 是什么时候加载进去的? .....	39
26、描述 THREADLOCAL (线程本地变量) 的底层实现原理及常用场景? .....	39
27、什么是微服务架构? .....	39
28、微服务有哪些特点? .....	40
29、LAMBDA 表达式是啥？优缺点? .....	40
30、讲一下 LAMBDA 的表达式作用域 (LAMBDA SCOPES) 。 .....	40
31、MYSQL 事务的特性有什么，说一下分别是什么意思? .....	41
<b>2021 【京东】面试真题: .....</b>	<b>41</b>

1、哪些情况下的对象会被垃圾回收机制处理掉? .....	41
2、讲一下常见编码方式? .....	41
3、UTF-8 编码中的中文占几个字节; INT 型几个字节? .....	42
4、静态代理和动态代理的区别，什么场景使用? .....	42
5、简述下 JAVA 的异常体系。 .....	42
6、谈谈你对解析与分派的认识。 .....	43
7、修改对象 A 的 EQUALS 方法的签名，那么使用 HASHMAP 存放这个对象实例的时候，会用哪个 EQUALS 方法? .....	43
8、JAVA 中实现多态的机制是什么? .....	43
9、如何将一个 JAVA 对象序列化到文件里? .....	43
10、说说你对 JAVA 反射的理解。 .....	44
11、说说你对 JAVA 注解的理解。 .....	44
12、说一下泛型原理，并举例说明。 .....	45
13、谈谈你对 JAVA 中 STRING 的了解。 .....	45
14、STRING 为什么要设计成不可变的? .....	46
15、REDIS 常见的几种数据结构说一下？各自的使用场景? .....	46
16、谈一谈缓存穿透、缓存击穿和缓存雪崩，以及各自的解决方案? .....	47
17、讲下 KAFKA、RABBITMQ、ROCKETMQ 之间的区别是什么? .....	48
18、KAFKA 的架构说一下? .....	48
19、KAFKA 怎么保证消息是有序的? .....	49
20、KAFKA 怎么保证消息不丢失? .....	49
21、KAFKA 怎么解决重复消费? .....	49
22、介绍下 MYSQL 聚簇索引与非聚簇索引的区别 (INNODB 与 MYISAM 引擎) ? .....	50
23、然后给一个联合索引(A,B)和一个语句, SELECT * FROM TABLE WHERE B = 'xxx'，判断是否能命中索引？为什么? .....	50
24、JAVA 多线程有哪几种实现方式? .....	50
25、用过 CONCURRENTHASHMAP，讲一下他和 HASHTABLE 的不同之处? .....	51
26、JAVA 怎么实现线程安全? .....	51

27、描述 THREADLOCAL（线程本地变量）的底层实现原理及常用场景。 .....	51
28、介绍下 SPRING BEAN 都有哪些作用域？ .....	52
29、注解 @AUTOWIRED 和 @RESOURCE 有什么区别？ .....	52
30、RPC 的实现基础？ .....	52
31、CMS，G1 垃圾回收器中的三色标记了解吗？ .....	53
<b>2021 【腾讯】面试真题： .....</b>	<b>54</b>
1、KAFKA 是什么？主要应用场景有哪些？ .....	54
2、KAFKA 为什么有 TOPIC 还要用 PARTITION？.....	54
3、客户端和服务器之间最多能建立多少个连接？ .....	55
4、HASHMAP 结构，线程不安全举个例子？ .....	55
5、MYSQL 索引分类？.....	55
6、了解线程 & 进程的区别吗？ .....	55
7、JAVA 进程间的几种通信方式？ .....	56
8、多台服务器同时对一个数据定时任务，怎么处理？ .....	56
9、常见分布式锁的几种实现方式？ .....	57
10、REDIS 分布式锁实现原理？ .....	57
11、REDIS 的数据类型及它们的使用场景？ .....	57
12、信号量与信号的区别？.....	57
13、SELECT 和 EPOLL 的底层结构是什么原理.....	58
14、场景题：1亿个数据取出最大前100个有什么方法？ .....	58
15、KAFKA 如何保证消息可靠？.....	59
16、消息队列的使用场景？.....	60
17、乐观锁和悲观锁的理解及如何实现，有哪些实现方式？ .....	60
18、ARRAYLIST 和 LINKEDLIST 的区别在哪里？ .....	60
19、谈谈你对 SQL 注入式攻击的理解？ .....	61
20、数据库事务的特性？ .....	61
21、REDIS 如何做内存优化？ .....	61
22、缓存穿透，缓存击穿，缓存雪崩都是咋回事？解决办法？ .....	62
23、数组和链表的区别？当数组内存过大时会出现什么问题？链表增删过多会出现的什么问题？ .....	62
24、常见排序算法和分别的复杂度？ .....	63
25、JDK 1.8 的 JVM 内存划分模型，堆和栈的区别.....	63
26、简单描述 MYSQL 中，索引，主键，唯一索引，联合索引的区别，对数据库的性能有什么影响（从读写两方面）？ .....	64
27、I/O 模型有哪几种？ .....	64
28、当你用浏览器打开一个链接的时候，计算机做了哪些工作步骤？ .....	64
29、虚拟 DOM 的优劣如何？.....	64
30、幻读是什么，用什么隔离级别可以防止幻读？ .....	65
<b>2021 【百度】面试真题： .....</b>	<b>65</b>
1、SPRINGBOOT 也有定时任务？是什么注解？ .....	65
2、请描述线程的生命周期，它们之间如何切换？ .....	65
3、什么情况线程会进入 WAITING 状态？ .....	65
4、简述多进程开发中 JOIN 和 DEAMON 的区别？ .....	66
5、异步和同步、阻塞和非阻塞之间的区别？ .....	66
6、为什么要分内核态和用户态？.....	66
7、说下类加载器与类加载？加载的类信息放在哪个区域？ .....	67
8、UDP 协议和 TCP 协议的区别？ .....	67
9、LIMIT 1000000 加载很慢的话，你是怎么解决的呢？ .....	67
10、MYSQL 的索引分类是什么？ .....	68
11、什么是散列表？ SELECT * 和 SELECT 1？ .....	68
12、MYSQL 的主从复制了解吗？ .....	68
13、SPRING 框架事务注解用什么注解？使用该注解的失效场景？ .....	68
14、FINAL、FINALLY、FINALLIZE? FINALLY 是在 RETURN 之前执行还是之后？FINALLY 块里的代码一定会执行吗？ ...	69

15、I/O 多路复用实现方式有哪些? .....	69
16、SELECT、POLL、EPOLL 区别有哪些? .....	69
17、哈希算法解决哈希冲突方式有哪些? .....	70
18、如何保证 REDIS 中的数据不丢失? .....	70
19、如何保证 REDIS 中的数据都是热点数据? .....	70
20、REDIS 持久化机制是如何做的? .....	71
21、REDIS 为什么在使用 RDB 进行快照时会通过子进程的方式进行实现? .....	72
22、介绍下 MYSQL 的主从复制原理? 产生主从延迟的原因? .....	72
23、父进程如果宕掉, 子进程会怎样? .....	72
24、孤儿进程和僵尸进程有什么区别? .....	72
25、MYSQL 中有哪几种锁? .....	73
26、互斥锁 (Mutex) 和自旋锁 (Spinlock) 分别在什么场景使用? .....	73
27、描述 SYNCHRONIZED、REENTRANTLOCK 的区别 ? .....	73
28、HASHMAP 扩容操作是怎么实现的? .....	73
29、CONCURRENTHASHMAP 1.7 与 1.8 区别? .....	73
30、如何使用 JAVA 的反射? .....	74
<b>2021 【华为】面试真题: .....</b>	<b>74</b>
1、JAVA 常用集合及特点? .....	74
2、开启一个线程的方法? .....	75
3、JAVA 面向对象包括哪些特性, 怎么理解的? .....	75
4、JAVA 如何保证线程安全? .....	76
5、介绍 SPRING MVC 的工作流程 ? .....	76
6、SPRING 框架中用到了哪些设计模式? .....	76
7、REDIS 的特点是什么? .....	77
8、为什么使用 REDIS, 有什么好处? .....	77
9、REDIS 雪崩和击穿了解吗? .....	77
10、什么是面向对象, 谈谈你的理解? .....	78
11、访问数据库除了 JDBC 还有什么? .....	78
12、你知道有哪些设计原则? .....	78
13、在生产环境 LINUX 服务器上, 发现某台运行 JAVA 服务的服务器的 CPU100%, 不借助任何可视化工具, 怎么进行问题的定位? .....	78
14、JDK 里面带的工具你知道哪些? .....	79
15、基本数据类型 BIT 长度? .....	79
16、CHAR 能不能存中文? .....	79
17、谈谈你对泛型的理解? .....	79
18、JAVA 程序是怎样运行的? .....	79
19、GC ROOT 有哪些? .....	80
20、栈帧的大小什么时候确定? .....	80
21、静态 FILED 声明和构造器哪个先执行? .....	80
22、线程创建方式是什么? .....	80
23、传统 I/O 跟 NIO 的区别? .....	81
24、消息队列的在各种场景下如何选型? .....	81
25、JAVA 的安全性体现在哪里? .....	83
26、STATIC 方法怎么访问非 STATIC 变量? .....	83
27、讲下你理解的 JAVA 多继承? .....	83
28、JAVA 基本类型有哪些? .....	83
29、线程池如果满了会怎么样? .....	83
30、什么是双亲委派机制, 它有什么作用? .....	84
<b>JAVA I/O.....</b>	<b>84</b>
1、I/O 流的分类 .....	84
2、字节流如何转为字符流? .....	85
3、字节流和字符流, 你更喜欢使用哪一个? .....	85

4、SYSTEM.OUT.PRINTLN 是什么? .....	85
5、什么是 FILTER 流? .....	85
5、有哪些可用的 FILTER 流? .....	85
6、有哪些 FILTER 流的子类? .....	85
7、NIO 和 I/O 的主要区别.....	86
8、BIO、NIO、AIO 有什么区别? .....	86
9、NIO 有哪些核心组件? .....	87
10、SELECT、POLL 和 EPOLL 什么区别.....	87
11、什么是 JAVA 序列化, 如何实现 JAVA 序列化? .....	88
12、如何实现对象克隆? .....	88
13、什么是缓冲区? 有什么作用? .....	88
14、什么是阻塞 IO? 什么是非阻塞 IO? .....	88
15、请说一下 PRINTSTREAM BUFFEREDWRITER PRINTWRITER 有什么不同? .....	89
<b>KAFKA.....</b>	<b>89</b>
1、KAFKA 是什么? 主要应用场景有哪些? .....	89
2、和其他消息队列相比,KAFKA 的优势在哪里? .....	89
3、什么是 PRODUCER、CONSUMER、BROKER、TOPIC、PARTITION? .....	90
4、KAFKA 的多副本机制了解吗? .....	90
5、KAFKA 的多分区 (PARTITION) 以及多副本 (REPLICA) 机制有什么好处呢? .....	90
6、ZOOKEEPER 在 KAFKA 中的作用知道吗? .....	91
7、KAFKA 如何保证消息的消费顺序? .....	91
8、KAFKA 如何保证消息不丢失? .....	92
9、KAFKA 判断一个节点是否还活着有那两个条件? .....	92
10、PRODUCER 是否直接将数据发送到 BROKER 的 LEADER(主节点)? .....	92
11、KAFA CONSUMER 是否可以消费指定分区消息吗? .....	92
12、KAFKA 高效文件存储设计特点是什么? .....	93
13、PARTITION 的数据如何保存到硬盘? .....	93
14、KAFKA 生产数据时数据的分组策略是怎样的? .....	93
15、CONSUMER 是推还是拉? .....	93
16、KAFKA 维护消费状态跟踪的方法有什么? .....	93
<b>MYSQL.....</b>	<b>94</b>
1、据库三大范式是什么.....	94
2、MYSQL 有关权限的表都有哪几个? .....	94
3、MYSQL 的 BINLOG 有几种录入格式? 分别有什么区别? .....	94
4、MYSQL 存储引擎 MYISAM 与 INNODB 区别.....	94
5、MYISAM 索引与 INNODB 索引的区别? .....	95
6、什么是索引? .....	95
7、索引有哪些优缺点? .....	95
索引的优点.....	95
索引的缺点.....	95
8、索引有哪几种类型? .....	95
9、MYSQL 中有哪几种锁? .....	96
10、MYSQL 中 INNODB 支持的四种事务隔离级别名称, 以及逐级之间的区别? .....	96
11、CHAR 和 VARCHAR 的区别? .....	96
12、主键和候选键有什么区别? .....	96
13、如何在 UNIX 和 MYSQL 时间戳之间进行转换? .....	96
14、MYISAM 表类型将去哪里存储, 并且还提供其存储格式? .....	97
15、MYSQL 里记录货币用什么字段类型好.....	97
16、创建索引时需要注意什么? .....	97
17、使用索引查询一定能提高查询的性能吗? 为什么.....	97
18、百万级别或以上的数据如何删除.....	98
19、什么是最左前缀原则? 什么是最左匹配原则.....	98

20、什么是聚簇索引？何时使用聚簇索引与非聚簇索引.....	98
21、MYSQL 连接器.....	98
22、MYSQL 查询缓存.....	99
23、MYSQL 分析器.....	99
24、MYSQL 优化器.....	99
25、MYSQL 执行器.....	99
26、什么是临时表，何时删除临时表？.....	99
28、什么叫外链接？.....	100
29、什么叫内链接？.....	101
30、使用 UNION 和 UNION ALL 时需要注意些什么？.....	101
31、MYISAM 存储引擎的特点.....	101
32、INNODB 存储引擎的特点.....	101
<b>NETTY.....</b>	<b>102</b>
1、为什么要用 NETTY 呢？ .....	102
2、NETTY 有哪些应用场景？ .....	102
3、NETTY 的优势有哪些？ .....	103
3、NETTY 核心组件有哪些？分别有什么作用？ .....	103
5、EVENTLOOPGROUP 了解么?和 EVENTLOOP 啥关系?.....	104
6、请说下对 BOOTSTRAP 和 SERVERBOOTSTRAP 的了解? .....	104
7、请说下 NETTY 线程模型？ .....	104
8、NETTY 服务端和客户端的启动过程是怎样的？ .....	105
9、什么是 TCP 粘包/拆包?.....	106
10、如何在 NETTY 中解决 TCP 粘包问题？ .....	106
11、TCP 长连接和短连接了解么？ .....	106
12、为什么需要心跳机制？NETTY 心跳机制了解么？ .....	107
13、讲讲 NETTY 的零拷贝.....	107
14、NETTY 和 TOMCAT 的区别？ .....	107
15、NETTY 发送消息有几种方式？ .....	108
<b>分布式.....</b>	<b>108</b>
1、分布式服务接口的幂等性如何设计？ .....	108
2、分布式系统中的接口调用如何保证顺序性？ .....	108
3、说说 ZooKEEPER 一般都有哪些使用场景？ .....	108
4、说说你们的分布式 SESSION 方案是啥？怎么做的？ .....	109
5、分布式事务了解吗？ .....	109
6、那常见的分布式锁有哪些解决方案？ .....	110
7、ZK 和 REDIS 的区别，各自有什么优缺点？ .....	110
8、MYSQL 如何做分布式锁？ .....	110
9、你了解业界哪些大公司的分布式锁框架.....	110
10、请讲一下你对 CAP 理论的理解.....	111
11、请讲一下你对 BASE 理论的理解.....	112
12、分布式与集群的区别是什么？ .....	112
13、请讲一下 BASE 理论的三要素.....	112
14、请说一下对两阶段提交协议的理解.....	113
15、请讲一下对 TCC 协议的理解.....	114
<b>微服务.....</b>	<b>114</b>
1、你对微服务是怎么理解的？ .....	114
2、说说微服务架构的优势。 .....	114
3、微服务有哪些特点？ .....	114
4、单体应用、SOA 和微服务架构有什么区别？ .....	115
5、在使用微服务架构时，你面临的挑战是什么？ .....	115
6、什么是 SPRING BOOT？ .....	115

7、SPRING BOOT 有哪些优点? .....	115
8、什么是 JavaConfig? .....	116
9、什么是 SPRING CLOUD? .....	116
10、使用 SPRING BOOT 开发分布式微服务时, 我们需要关注哪些问题? .....	116
11、服务注册和发现是什么意思? SPRING CLOUD 如何实现? .....	117
12、负载平衡的意义什么? .....	117
13、什么是 HYSTRIX? .....	117
14、什么是 NETFLIX FEIGN? 它的优点是什么? .....	117
15、SPRING CLOUD 断路器的作用.....	117
<b>消息队列.....</b>	<b>118</b>
1、为什么使用消息队列? .....	118
2、消息队列有什么优点和缺点? .....	118
3、KAFKA、ACTIVEMQ、RABBITMQ、ROCKETMQ 都有什么区别, 以及适合哪些场景? .....	118
4、如何保证消息不被重复消费? .....	119
5、如何保证消息消费的幂等性? .....	119
6、如何保证消息的可靠性传输? (如何处理消息丢失的问题) .....	120
7、如果让你写一个消息队列, 该如何进行架构设计啊? 说一下你的思路。.....	121
8、如何解决消息队列的延时以及过期失效问题? .....	122
9、消息队列满了以后该怎么处理? .....	123
10、消息如何分发? .....	123
11、消息怎么路由? .....	123
12、消息基于什么传输? .....	123
13、为什么不应该对所有的 MESSAGE 都使用持久化机制? .....	123
14、如何保证高可用的? RABBITMQ 的集群.....	124
15、RABBITMQ 的工作模式.....	124
16、为什么需要消息系统, MYSQL 不能满足需求吗? .....	125
<b>DUBBO.....</b>	<b>125</b>
1、说说核心的配置有哪些? .....	125
2、DUBBO 支持哪些协议, 每种协议的应用场景, 优缺点? .....	126
3、服务提供者能实现失效踢出是什么原理? .....	127
4、RPC 架构有哪些组件? .....	127
5、DUBBO 服务调用是阻塞的吗? .....	127
6、DUBBO 核心功能有哪些? .....	127
7、DUBBO 服务器注册与发现的流程? .....	128
8、DUBBO MONITOR 实现原理? .....	128
9、DUBBO 和 SPRING CLOUD 有什么关系? .....	128
10、DUBBO 和 SPRING CLOUD 有什么哪些区别? .....	129
11、DUBBO 有哪些注册中心? .....	129
12、DUBBO 的注册中心集群挂掉, 发布者和订阅者之间还能通信么? .....	129
13、DUBBO 集群提供了哪些负载均衡策略? .....	129
14、DUBBO 的集群容错方案有哪些? .....	129
15、DUBBO 超时设置有哪些方式? .....	130
<b>JAVA 集合.....</b>	<b>130</b>
1、说说 LIST, SET, QUEUE, MAP 四者的区别? .....	130
2、如何选用集合?.....	130
3、为什么要使用集合? .....	131
4、ARRAYLIST 和 VECTOR 的区别?.....	131
5、ARRAYLIST 与 LINKEDLIST 区别?.....	131
6、COMPARABLE 和 COMPARATOR 的区别? .....	131
7、无序性和不可重复性的含义是什么.....	132
8、比较 HASHSET、LINKEDHASHSET 和 TREESET 三者的异同.....	132

9、试比较 QUEUE 与 DEQUE 的区别.....	132
10、请谈一下对 PRIORITYQUEUE 的认识? .....	133
11、HASHMAP 和 HASHTABLE 的区别? .....	133
12、HASHSET 如何检查重复? .....	134
13、HASHMAP 的长度为什么是 2 的幂次方? .....	134
这个算法应该如何设计呢? .....	134
14、CONCURRENTHASHMAP 和 HASHTABLE 的区别? .....	134
15、CONCURRENTHASHMAP 线程安全的具体实现方式是怎样的? .....	135
16、TREEMAP 和 TREESET 在排序时如何比较元素? COLLECTIONS 工具类中的 SORT()方法如何比较元素? .....	135
17、COLLECTION 和 COLLECTIONS 有什么区别? .....	136
18、ARRAY 和 ARRAYLIST 有何区别? .....	136
19、HASHMAP 和 CONCURRENTHASHMAP 的区别.....	136
20、如果使用 OBJECT 作为 HASHMAP 的 KEY, 应该怎么办呢? .....	136
21、为什么 HASHMAP 中 STRING、INTEGER 这样的包装类适合作为 K? .....	137
22、什么是哈希冲突? .....	137
23、你知道 FAIL-FAST 和 FAIL-SAFE 吗? .....	137
24、ARRAYS.ASLIST 获得的 LIST 应该注意什么? .....	137
25、FINAL、FINALLY 和 FINALIZE()的区别.....	138
26、内部类有哪些分类, 分别解释一下.....	138
27、项目为 UTF-8 环境, CHARC=中, 是否合法.....	139
28、动态代理是基于什么原理.....	139
29、EXCEPTION 和 ERROR 有什么区别.....	139
30、反射的基本原理, 反射创建类实例的三和方式是什么.....	139
<b>ZOOKEEPER.....</b>	<b>139</b>
1、ZOOKEEPER 是什么? .....	139
2、ZOOKEEPER 提供了什么? .....	140
3、ZOOKEEPER 文件系统.....	140
4、ZAB 协议? .....	140
5、四种类型的数据节点 ZNODE.....	141
6、ZOOKEEPER WATCHER 机制 -- 数据变更通知.....	141
7、客户端注册 WATCHER 实现.....	142
8、服务端处理 WATCHER 实现.....	142
9、客户端回调 WATCHER.....	143
10、ACL 权限控制机制.....	143
12、会话管理.....	144
13、服务器角色.....	144
14、ZOOKEEPER 下 SERVER 工作状态.....	144
15、数据同步.....	145
16、ZOOKEEPER 是如何保证事务的顺序一致性的? .....	146
17、分布式集群中为什么会有 MASTER? .....	146
18、zk 节点宕机如何处理? .....	146
19、ZOOKEEPER 负载均衡和 NGINX 负载均衡区别.....	146
20、ZOOKEEPER 有哪几种几种部署模式? .....	147
21、集群最少要几台机器, 集群规则是怎样的? .....	147
22、集群支持动态添加机器吗? .....	147
23、ZOOKEEPER 对节点的 WATCH 监听通知是永久的吗? 为什么不是永久的? .....	147
24、ZOOKEEPER 的 JAVA 客户端都有哪些? .....	147
25、CHUBBY 是什么, 和 ZOOKEEPER 比你怎么看? .....	147
26、说几个 ZOOKEEPER 常用的命令。 .....	148
27、ZAB 和 PAXOS 算法的联系与区别? .....	148
28、ZOOKEEPER 的典型应用场景.....	148
<b>JAVA 并发编程.....</b>	<b>151</b>

1、在 JAVA 中守护线程和本地线程区别? .....	151
2、线程与进程的区别? .....	151
3、什么是多线程中的上下文切换? .....	151
4、死锁与活锁的区别, 死锁与饥饿的区别? .....	152
5、什么是线程组, 为什么在 JAVA 中不推荐使用? .....	152
6、为什么使用 EXECUTOR 框架? .....	153
7、在 JAVA 中 EXECUTOR 和 EXECUTORS 的区别? .....	153
8、什么是原子操作? 在 JAVA CONCURRENCY API 中有哪些原子类(ATOMIC CLASSES)? .....	153
9、JAVA CONCURRENCY API 中的 LOCK 接口(LOCK INTERFACE)是什么? 对比同步它有什么优势? .....	154
10、什么是 EXECUTORS 框架? .....	154
11、什么是阻塞队列? 阻塞队列的实现原理是什么? 如何使用阻塞队列来实现生产者-消费者模型? .....	155
12、什么是 CALLABLE 和 FUTURE?.....	155
13、什么是 FUTURETASK? 使用 EXECUTORSERVICE 启动任务。 .....	156
14、什么是并发容器的实现? .....	156
15、多线程同步和互斥有几种实现方法, 都是什么? .....	156
16、什么是竞争条件? 你怎样发现和解决竞争? .....	157
17、你将如何使用 THREAD DUMP? 你将如何分析 THREAD DUMP? .....	157
18、为什么我们调用 START()方法时会执行 RUN() 方法, 为什么我们不能直接调用 RUN()方法? .....	158
19、JAVA 中你怎样唤醒一个阻塞的线程? .....	158
20、在 JAVA 中 CYCLIBARRIER 和 COUNTDOWNLATCH 有什么区别? .....	158
21、什么是不可变对象, 它对写并发应用有什么帮助? .....	159
22、什么是多线程中的上下文切换? .....	159
23、JAVA 中用到的线程调度算法是什么? .....	160
24、什么是线程组, 为什么在 JAVA 中不推荐使用? .....	160
25、为什么使用 EXECUTOR 框架比使用应用创建和管理线程好? .....	160
26、JAVA 中有几种方法可以实现一个线程? .....	161
27、如何停止一个正在运行的线程? .....	161
28、NOTIFY()和 NOTIFYALL()有什么区别? .....	161
29、什么是 DAEMON 线程? 它有什么意义? .....	161
30、JAVA 如何实现多线程之间的通讯和协作? .....	161
31、什么是可重入锁 (REENTRANTLOCK) ? .....	162
32、当一个线程进入某个对象的一个 SYNCHRONIZED 的实例方法后, 其它线程是否可进入此对象的其它 方法? .....	162
33、乐观锁和悲观锁的理解及如何实现, 有哪些实现方式? .....	162
34、SYNCHRONIZED 和 CONCURRENTHASHMAP 有什么区别? .....	163
35、COPYONWRITEARRAYLIST 可以用于什么应用场景? .....	163
36、什么叫线程安全? SERVLET 是线程安全吗?.....	164
37、VOLATILE 有什么用? 能否用一句话说明下 VOLATILE 的应用场景? .....	164
38、为什么代码会重排序? .....	164
39、在 JAVA 中 WAIT 和 SLEEP 方法的不同? .....	165
40、一个线程运行时发生异常会怎样? .....	165
41、如何在两个线程间共享数据? .....	165
42、JAVA 中 NOTIFY 和 NOTIFYALL 有什么区别? .....	165
43、为什么 WAIT, NOTIFY 和 NOTIFYALL 这些方法不在 THREAD 类里面? .....	165
44、什么是 THREADLOCAL 变量? .....	165
45、JAVA 中 INTERRUPTED 和 ISINTERRUPTED 方法的区别? .....	166
46、为什么 WAIT 和 NOTIFY 方法要在同步块中调用? .....	166
47、为什么你应该在循环中检查等待条件?.....	166
48、JAVA 中的同步集合与并发集合有什么区别? .....	166
49、什么是线程池? 为什么要使用它? .....	167
50、怎么检测一个线程是否拥有锁? .....	167
51、你如何在 JAVA 中获取线程堆栈? .....	167
52、JVM 中哪个参数是用来控制线程的栈堆栈小的?.....	167
53、THREAD 类中的 YIELD 方法有什么作用? .....	168

54、JAVA 中 CONCURRENTHASHMAP 的并发度是什么? .....	168
55、JAVA 中 SEMAPHORE 是什么? .....	168
56、JAVA 线程池中 SUBMIT() 和 EXECUTE()方法有什么区别? .....	168
57、什么是阻塞式方法? .....	168
58、JAVA 中的 READWRITELOCK 是什么? .....	169
59、VOLATILE 变量和 ATOMIC 变量有什么不同? .....	169
60、可以直接调用 THREAD 类的 RUN ()方法么? .....	169
61、如何让正在运行的线程暂停一段时间? .....	169
62、你对线程优先级的理解是什么? .....	169
63、什么是线程调度器(THREAD SCHEDULER)和时间分片(TIME SLICING )? .....	170
64、你如何确保 MAIN()方法所在的线程是 JAVA 程序最后结束的线程? .....	170
65、线程之间是如何通信的? .....	170
66、为什么线程通信的方法 WAIT(), NOTIFY()和 NOTIFYALL()被定义在 OBJECT 类里? .....	170
67、为什么 WAIT(), NOTIFY()和 NOTIFYALL ()必须在同步方法或者同步块中被调用? .....	171
68、为什么 THREAD 类的 SLEEP()和 YIELD ()方法是静态的? .....	171
69、如何确保线程安全? .....	171
70、同步方法和同步块，哪个是更好的选择? .....	171
71、如何创建守护线程? .....	171
72、什么是 JAVA TIMER 类? 如何创建一个有特定时间间隔的任务? .....	172
<b>MYBATIS.....</b>	<b>172</b>
1、什么是 MYBATIS? .....	172
2、MYBAITS 的优点: .....	172
3、MYBATIS 框架的缺点: .....	173
4、MYBATIS 框架适用场合: .....	173
5、MYBATIS 与 HIBERNATE 有哪些不同? .....	173
6、#{}和\${}的区别是什么? .....	173
7、当实体类中的属性名和表中的字段名不一样，怎么办 ? .....	174
8、模糊查询 LIKE 语句该怎么写?.....	174
9、通常一个 XML 映射文件，都会写一个 DAO 接口与之对应，请问，这个 DAO 接口的工作原理是什么? DAO 接口里的方法，参数不同时，方法能重载吗? .....	175
10、MYBATIS 是如何进行分页的? 分页插件的原理是什么? .....	175
11、MYBATIS 是如何将 SQL 执行结果封装为目标对象并返回的? 都有哪些映射形式? .....	175
12、如何执行批量插入?.....	176
13、如何获取自动生成的(主)键值?.....	176
14、在 MAPPER 中如何传递多个参数?.....	177
15、MYBATIS 动态 SQL 有什么用? 执行原理? 有哪些动态 SQL? .....	178
16、XML 映射文件中，除了常见的 SELECT INSERT UPDAE DELETE 标签之外，还有哪些标签? .....	178
17、MYBATIS 的 XML 映射文件中，不同的 XML 映射文件，ID 是否可以重复? .....	178
18、为什么说 MYBATIS 是半自动 ORM 映射工具? 它与全自动的区别在哪里? .....	178
19、一对一、一对多的关联查询 ? .....	179
20、MYBATIS 实现一对一有几种方式?具体怎么操作的? .....	180
21、MYBATIS 实现一对多有几种方式,怎么操作的? .....	180
22、MYBATIS 是否支持延迟加载? 如果支持，它的实现原理是什么? .....	180
23、MYBATIS 的一级、二级缓存:.....	180
24、什么是 MYBATIS 的接口绑定? 有哪些实现方式? .....	181
25、使用 MYBATIS 的 MAPPER 接口调用时有哪些要求? .....	181
26、MAPPER 编写有哪几种方式? .....	181
27、简述 MYBATIS 的插件运行原理，以及如何编写一个插件。 .....	182
<b>REDIS.....</b>	<b>183</b>
1、什么是 REDIS?.....	183
2、REDIS 的数据类型? .....	184
3、使用 REDIS 有哪些好处? .....	184

4、REDIS 相比 MEMCACHED 有哪些优势? .....	184
5、MEMCACHE 与 REDIS 的区别都有哪些? .....	184
6、REDIS 是单进程单线程的? .....	185
8、REDIS 的持久化机制是什么? 各自的优缺点? .....	185
9、REDIS 常见性能问题和解决方案: .....	186
10、REDIS 过期键的删除策略? .....	186
11、REDIS 的回收策略 (淘汰策略) ?.....	186
12、为什么 REDIS 需要把所有数据放到内存中? .....	187
13、REDIS 的同步机制了解么? .....	187
14、PIPELINE 有什么好处, 为什么要用 PIPELINE? .....	187
15、是否使用过 REDIS 集群, 集群的原理是什么? .....	187
16、REDIS 集群方案什么情况下会导致整个集群不可用? .....	187
17、REDIS 支持的 JAVA 客户端都有哪些? 官方推荐用哪个? .....	188
18、JEDIS 与 REDISSON 对比有什么优缺点? .....	188
19、REDIS 如何设置密码及验证密码? .....	188
20、说说 REDIS 哈希槽的概念? .....	188
21、REDIS 集群的主从复制模型是怎样的? .....	188
22、REDIS 集群会有写操作丢失吗? 为什么? .....	188
23、REDIS 集群之间是如何复制的? .....	188
24、REDIS 集群最大节点个数是多少? .....	189
25、REDIS 集群如何选择数据库? .....	189
26、怎么测试 REDIS 的连通性.....	189
27、怎么理解 REDIS 事务? .....	189
28、REDIS 事务相关的命令有哪几个? .....	189
29、REDIS KEY 的过期时间和永久有效分别怎么设置? .....	189
30、REDIS 如何做内存优化? .....	189
31、REDIS 回收进程如何工作的? .....	189
32、都有哪些办法可以降低 REDIS 的内存使用情况呢? .....	190
33、REDIS 的内存用完了会发生什么? .....	190
34、一个 REDIS 实例最多能存放多少的 KEYS? LIST、SET、SORTED SET 他们最多能存放多少元素? .....	190
35、MYSQL 里有 2000W 数据, REDIS 中只存 20W 的数据, 如何保证 REDIS 中的数据都是热点数据? REDIS 内存数据集大小上升到一定大小的时候, 就会施行数据淘汰策略。相关知识: REDIS 提供 6 种数据淘汰策略: .....	190
36、REDIS 最适合的场景? .....	191
37、假如 REDIS 里面有 1 亿个 KEY, 其中有 10W 个 KEY 是以某个固定的已知的前缀开头的, 如果将 它们全部找出来? .....	192
38、如果有大量的 KEY 需要设置同一时间过期, 一般需要注意什么? .....	192
39、使用过 REDIS 做异步队列么, 你是怎么用的? .....	192
40、使用过 REDIS 分布式锁么, 它是什么回事.....	193
<b>SPRING.....</b>	<b>193</b>
1、不同版本的 SPRING FRAMEWORK 有哪些主要功能? .....	193
2、什么是 SPRING FRAMEWORK? .....	193
3、列举 SPRING FRAMEWORK 的优点。 .....	194
4、SPRING FRAMEWORK 有哪些不同的功能? .....	194
5、SPRING FRAMEWORK 中有多少个模块, 它们分别是什么? .....	194
6、什么是 SPRING 配置文件? .....	195
7、SPRING 应用程序有哪些不同组件? .....	195
8、使用 SPRING 有哪些方式? .....	195
9、什么是 SPRING IOC 容器? .....	196
10、什么是依赖注入? .....	196
11、可以通过多少种方式完成依赖注入? .....	196
12、区分构造函数注入和 SETTER 注入.....	196
13、SPRING 中有多少种 IOC 容器? .....	197

---

14、区分 BEANFACTORY 和 APPLICATIONCONTEXT。	197
15、列举 IoC 的一些好处。	197
16、SPRING IOC 的实现机制。	198
17、什么是 SPRING BEAN?	199
18、SPRING 提供了哪些配置方式?	199
19、SPRING 支持集中 BEAN SCOPE?	199
20、SPRING BEAN 容器的生命周期是什么样的	200
21、什么是 SPRING 的内部 BEAN?	200
22、什么是 SPRING 装配	201
23、自动装配有哪些方式?	201
24、自动装配有什么局限?	202
25、什么是基于注解的容器配置	202
26、如何在 SPRING 中启动注解装配?	202
27、@COMPONENT, @CONTROLLER, @REPOSITORY, @SERVICE 有何区别	202
28、@REQUIRED 注解有什么用?	203
29、@AUTOWIRED 注解有什么用?	203
30、@QUALIFIER 注解有什么用?	204
31、@REQUESTMAPPING 注解有什么用?	205
32、SPRING DAO 有什么用?	205
33、列举 SPRING DAO 抛出的异常。	205
34、SPRING JDBC API 中存在哪些类?	205
35、使用 SPRING 访问 HIBERNATE 的方法有哪些?	205
36、列举 SPRING 支持的事务管理类型	206
37、SPRING 支持哪些 ORM 框架	206
38、什么是 AOP?	206
39、什么是 ASPECT?	206
40、什么是切点 (JOINPOINT)	207
41、什么是通知 (ADVICE) ?	207
43、指出在 SPRING AOP 中 CONCERN 和 CROSS-CUTTINGCONCERN 的不同之处。	208
45、SPRING AOP AND ASPECTJ AOP 有什么区别?	208
46、如何理解 SPRING 中的代理?	208
47、什么是编织 (WEAVING) ?	208
48、SPRING MVC 框架有什么用?	209
49、描述一下 DISPATCHERSERVLET 的工作流程	209
50、介绍一下 WEBAPPLICATIONCONTEXT	210
51、什么是 SPRING?	210
52、使用 SPRING 框架的好处是什么?	210
53、SPRING 由哪些模块组成?	211
54、核心容器 (应用上下文) 模块。	211
55、BEANFACTORY - BEANFACTORY 实现举例。	211
66、XMLBEANFACTORY	211
67、解释 AOP 模块	211
68、解释 JDBC 抽象和 DAO 模块。	212
69、解释对象/关系映射集成模块。	212
70、解释 WEB 模块。	212
72、SPRING 配置文件	212
73、什么是 SPRING IOC 容器	212
74、IOC 的优点是什么?	212
75、APPLICATIONCONTEXT 通常的实现是什么?	213
76、BEAN 工厂和 APPLICATION CONTEXTS 有什么区别?	213
77、一个 SPRING 的应用看起来象什么?	213
78、什么是 SPRING 的依赖注入?	213
79、有哪些不同类型的 IOC (依赖注入) 方式?	213
80、哪种依赖注入方式你建议使用，构造器注入，还是 SETTER 方法注入?	214

81、什么是 SPRING BEANS?.....	214
82、一个 SPRING BEAN 定义包含什么? .....	214
83、如何给 SPRING 容器提供配置元数据?.....	214
84、你怎样定义类的作用域?.....	214
85、解释 SPRING 支持的几种 BEAN 的作用域.....	215
86、SPRING 框架中的单例 BEAN 是线程安全的吗?.....	215
87、解释 SPRING 框架中 BEAN 的生命周期。.....	215
88、哪些是重要的 BEAN 生命周期方法? 你能重载它们吗? .....	216
89、什么是 SPRING 的内部 BEAN? .....	216
90、在 SPRING 中如何注入一个 JAVA 集合? .....	216
91、什么是 BEAN 装配?.....	216
92、什么是 BEAN 的自动装配? .....	216
93、解释不同方式的自动装配.....	216
94、自动装配有哪些局限性 ?.....	217
95、你可以在 SPRING 中注入一个 NULL 和一个空字符串吗? .....	217
96、什么是基于 JAVA 的 SPRING 注解配置? 给一些注解的例子.....	217
97、什么是基于注解的容器配置?.....	217
98、怎样开启注解装配? .....	218
99、@REQUIRED 注解.....	218
100、@AUTOWIRED 注解.....	218
101、@QUALIFIER 注解.....	218
102、在 SPRING 框架中如何更有效地使用 JDBC?.....	218
103、JDBCTEMPLAT.....	218
104、SPRING 对 DAO 的支持.....	218
105、使用 SPRING 通过什么方式访问 HIBERNATE?.....	219
106、SPRING 支持的 ORM.....	219
108、SPRING 支持的事务管理类型.....	219
109、SPRING 框架的事务管理有哪些优点? .....	219
110、你更倾向用那种事务管理类型? .....	220
111、解释 AOP.....	220
112、ASPECT 切面.....	220
113、在 SPRING AOP 中, 关注点和横切关注的区别是什么? .....	220
114、连接点.....	220
115、通知.....	220
116、切点.....	221
117、什么是引入?.....	221
118、什么是目标对象?.....	221
119、什么是代理?.....	221
120、有几种不同类型的自动代理.....	221
121、什么是织入。什么是织入应用的不同点? .....	221
122、解释基于 XML SCHEMA 方式的切面实现。.....	221
123、解释基于注解的切面实现.....	222
124、什么是 SPRING 的 MVC 框架? .....	222
125、DISPATCHERSERVLET.....	222
126、WEBAPPLICATIONCONTEXT.....	222
127、什么是 SPRING MVC 框架的控制器? .....	222
128、@CONTROLLER 注解.....	222
129、@REQUESTMAPPING 注解.....	222
<b>JVM.....</b>	<b>223</b>
<b>1. 什么是 JVM? .....</b>	<b>225</b>
1.1    请问 JDK 与 JVM 有什么区别? .....	225
1.2    你认识哪些 JVM 厂商?.....	225
1.3    ORACLEJDK 与 OPENJDK 有什么区别?.....	226

1.4	开发中使用哪个版本的 JDK? 生产环境呢? 为什么这么选? .....	226
<b>2.</b>	<b>什么是 JAVA 字节码? .....</b>	<b>226</b>
2.1	字节码文件中包含哪些内容?.....	226
2.2	什么是常量?.....	227
2.3	你怎么理解常量池?.....	227
<b>3.</b>	<b>JVM 的运行时数据区有哪些?.....</b>	<b>227</b>
3.1	什么是堆内存? .....	227
3.2	堆内存包括哪些部分? .....	228
3.3	什么是非堆内存?.....	228
<b>4.</b>	<b>什么是内存溢出? .....</b>	<b>228</b>
4.1	什么是内存泄漏?.....	228
4.2	两者有什么关系? .....	229
<b>5.</b>	<b>给定一个具体的类, 请分析对象的内存占用.....</b>	<b>229</b>
5.1	怎么计算出来的?.....	229
5.2	对象头中包含哪些部分? .....	229
<b>6.</b>	<b>常用的 JVM 启动参数有哪些?.....</b>	<b>230</b>
6.1	设置堆内存 XMX 应该考虑哪些因素? .....	232
6.2	假设物理内存是 8G, 设置多大堆内存比较合适?.....	232
6.3	设置的值与 JVM 进程所占用的内存有什么关系?.....	232
6.4	怎样开启 GC 日志? .....	232
<b>7.</b>	<b>JAVA8 默认使用的垃圾收集器是什么?.....</b>	<b>233</b>
7.1	JAVA11 的默认垃圾收集器是什么?.....	233
7.2	常见的垃圾收集器有哪些?.....	233
7.3	什么是串行垃圾收集?.....	233
7.4	什么是并行垃圾收集?.....	233
7.5	什么是并发垃圾收集器?.....	234
7.6	什么是增量式垃圾收集?.....	234
7.7	什么是年轻代? .....	234
7.8	什么是 GC 停顿(GC PAUSE)?.....	234
7.9	GC 停顿与 STW 停顿有什么区别? .....	234
<b>8.</b>	<b>如果 CPU 使用率突然飙升, 你会怎么排查? .....</b>	<b>234</b>
8.1	如果系统响应变慢, 你会怎么排查? .....	235
8.2	系统性能一般怎么衡量? .....	235
<b>9.</b>	<b>使用过哪些 JVM 相关的工具? .....</b>	<b>235</b>
9.1	查看 JVM 进程号的命令是什么?.....	235
9.2	怎么查看剩余内存?.....	235
9.3	查看线程栈的工具是什么? .....	235
9.4	用什么工具来获取堆内存转储?.....	235
9.5	内存 DUMP 时有哪些注意事项?.....	236
9.6	使用 JMAP 转储堆内存大致的参数怎么处理? .....	236
9.7	为什么转储文件以 结尾? .....	236
9.8	内存 DUMP 完成之后, 用什么工具来分析? .....	236
9.9	如果忘记了使用什么参数你一般怎么处理? .....	236
<b>10.</b>	<b>开发性问题: 你碰到过哪些 JVM 问题? .....</b>	<b>236</b>

# 大厂面试的基本流程

以下是一些大厂面试的基本流程，大家可以参考下：

## 字节跳动

字节被称为宇宙条，源于面试难度变态。

**招聘分五轮：笔试 + 三轮专业面 + 一轮 HR 面**

- 笔试：纯算法题，3-5 个，一般完成 1-2 个过
- 一面：基础面，计算机专业基础，一般根据简历上信息问（项目）+ 算法
- 二面：项目经历 + 岗位方向基础 + 算法
- 三面：还是项目经历（发现了什么）+ 专业知识
- 四面：HR 面，HR 面都差不多，主要关心为什么来、职业规划、兴趣爱好、个人管理等等

**总结：**

- 字节面试官一般很有耐心（时长 1 小时+），问的也会很细（比如 C++ sort 函数怎么实现的），十分考验专业功底。
- 算法要求不高，剑指 offer 难度。但一定要会，不会的一般都挂了。
- 字节的面试效率很高，一般第二天就会下面，有时候直接下面（不给太多时间准备）。

## 阿里

**笔试 + 三技术面 + HR 面**

笔试：算法 + 其他，难度较大

一面：基础

二面：项目

三面：项目

HR 面

**总结：**

1. 阿里的整理难度偏大，无论是笔试还是面试
2. 面试不一定有算法题
3. 反馈周期，2 - 7 天

## 腾讯

### **流程：笔试 + 三技术面 + HR 面**

笔试：（不一定有）

一面：计算机基础 + 算法

二面：项目 + 技术 + 算法

三面：项目深入

HR 面

#### **总结：**

- 腾讯的池子可以让人面上十来回。一般某一次挂了，可能过段时间又被其他部门捞起来，继续安排面试，甚至面的岗位不是你投的（论简历的特效）；
- 一般 2-5 天一轮，线下的话当天或隔天；
- 每轮面试大概一个小时左右，每轮有 2-3 个算法；

## **网易游戏**

### **流程：笔试 + 两轮面试**

笔试：算法题，互娱的难度不大，雷火稍大

一面：基础 + 算法

二面：综合面，项目、情景题、基础

#### **总结：**

一面是远程，二面线下（也可能远程，概率小）

## **面试前需要准备：**

1. **Java 八股文：**了解常考的题型和回答思路；
2. **算法：**刷 100-200 道题，记住刷题最重要的是要理解其思想，不要死记硬背，碰上原题很难，但大多数的解题思路是相通的。
3. **项目：**主要准备最近一家公司所负责的业务和项目：
  - 项目的背景，为啥要做这个项目；
  - 系统的演进之路，有哪几个阶段，每个阶段主要做了什么；
  - 项目中的技术选型，在项目中使用一些工具和框架时的调研，为啥选这个；

- 
- 项目的亮点：就是你在项目中做过最牛逼的事，复杂的需求方案设计、性能优化、线上问题处理、项目重构等等；
4. **架构设计：**主要是平台化的一些思想、DDD 领域驱动设计思想，随着经验的增加，这块会越来越重要。
  5. **项目管理：**主要是在主导跨团队的项目时，如何高效的协调好各个团队的工作，使用哪些方法来保障项目的按时交付。在项目遇到困难时，作为项目负责人如何应对等等。跟架构设计一样，这块也是随着经验的增加越来越重要。
  6. **通用问题：**几个比较容易被问到的问题是：1) 为什么离职；2) 在上家公司哪些能力得到了成长；3) 平时怎么学习的？
  7. **问面试官：**每次面试最后面试官一般会问有没有什么想问的，如果不知道问什么，可以问下团队当前负责的业务是什么？主要面临的挑战是什么？



## Java 进阶训练营

学员口中的大厂面试宝典



## 讲师介绍

主讲老师: **秦金卫**

Apache Dubbo/ShardingSphere PMC  
前阿里架构师/前火币高级技术总监

**10 多年研发管理和架构经验;**  
**熟悉各类中间件, 擅长分布式系统架构;**

过去十来年的工作中:

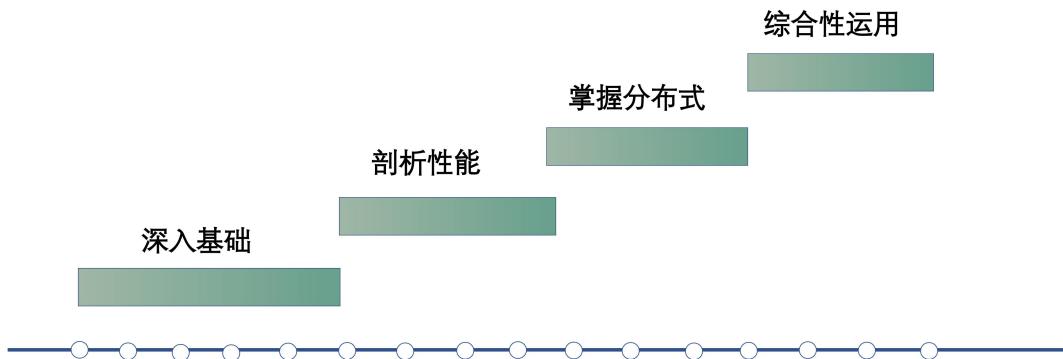
1. 曾作为求职者, 拿到过几乎所有一二线互联网公司的架构师、技术专家/总监职位 Offer。
2. 曾作为面试官, 面试超过 2000 人次, 具备丰富的一线技术人员的面试经验和辅导经验。



秦金卫



# 课程大纲





## 8 个实战项目



JVM—使用 JVM 分析工具剖析 JVM 系统性能



NIO—使用 Netty 实现一个高性能业务网关



并发—使用多线程实现高并发业务处理程序



数据—基于分库分表优化处理千万业务数据



分布式—设计实现一个功能完善的服务框架



分布式—设计实现一个简单高效的消息队列



重构—淘宝某业务系统的优化重构经验拆解



高并发—设计一个简版双十一/618 秒杀系统



郑重承诺

不满意？退全款！！！  
学完 7 天不合适全额退



没时间

项目紧急上线，没时间学了？可延期可退款！



不适应

不喜欢老师的授课风格？退！



听不懂

基础太差，跟不上课？退！



不想学

间断性颓废，就是不想学？退！



.....

但凡你说“课不好”，不用证明，退！

---

2021 【美团】面试真题：

## 1、Spring AOP 底层原理

aop 底层是采用动态代理机制实现的：接口+实现类

- 如果要代理的对象，实现了某个接口，那么 Spring AOP 会使用 JDK Proxy，去创建代理对象。
- 没有实现接口的对象，就无法使用 JDK Proxy 去进行代理了，这时候 Spring AOP 会使用 Cglib 生成一个被代理对象的子类来作为代理。

就是由代理创建出一个和 impl 实现类平级的一个对象，但是这个对象不是一个真正的对象，只是一个代理对象，但它可以实现和 impl 相同的功能，这个就是 aop 的横向机制原理，这样就不需要修改源代码。

## 2、HashMap 的底层数据结构是怎样的？

### JDK1.8 之前

- JDK1.8 之前 HashMap 底层是 数组和链表 结合在一起使用也就是 链表散列。
- HashMap 通过 key 的 hashCode 经过扰动函数处理过后得到 hash 值，然后通过  $(n - 1) \& \text{hash}$  判断当前元素存放的位置（这里的 n 指的是数组的长度），如果当前位置存在元素的话，就判断该元素与要存入的元素的 hash 值以及 key 是否相同，如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。
- 所谓扰动函数指的就是 HashMap 的 hash 方法。使用 hash 方法也就是扰动函数是为了防止一些实现比较差的 hashCode() 方法 换句话说使用扰动函数之后可以减少碰撞。

### JDK1.8 之后

---

当链表长度大于阈值（默认为 8）时，会首先调用 `treeifyBin()` 方法。这个方法会根据 `HashMap` 数组来决定是否转换为红黑树。只有当数组长度大于或者等于 64 的情况下，才会执行转换红黑树操作，以减少搜索时间。否则，就是只是执行 `resize()` 方法对数组扩容。

### 3、`HashMap` 的扩容机制是怎样的？

一般情况下，当元素数量超过阈值时便会触发扩容。每次扩容的容量都是之前容量的 2 倍。`HashMap` 的容量是有上限的，必须小于  $1<<30$ ，即 1073741824。如果容量超出了这个数，则不再增长，且阈值会被设置为 `Integer.MAX_VALUE`。

#### JDK7 中的扩容机制

- 空参数的构造函数：以默认容量、默认负载因子、默认阈值初始化数组。内部数组是空数组。
- 有参构造函数：根据参数确定容量、负载因子、阈值等。
- 第一次 `put` 时会初始化数组，其容量变为不小于指定容量的 2 的幂数，然后根据负载因子确定阈值。
- 如果不是第一次扩容，则 `新容量=旧容量 × 2`，`新阈值=新容量 × 负载因子`。

#### JDK8 的扩容机制

- 空参数的构造函数：实例化的 `HashMap` 默认内部数组是 `null`，即没有实例化。第一次调用 `put` 方法时，则会开始第一次初始化扩容，长度为 16。
- 有参构造函数：用于指定容量。会根据指定的正整数找到不小于指定容量的 2 的幂数，将这个数设置赋值给阈值（threshold）。第一次调用 `put` 方法时，会将阈值赋值给容量，然后让 `阈值 = 容量 × 负载因子`。
- 如果不是第一次扩容，则容量变为原来的 2 倍，阈值也变为原来的 2 倍。（容量和阈值都变为原来的 2 倍时，负载因子还是不变）。

此外还有几个细节需要注意：

- 首次 `put` 时，先会触发扩容（算是初始化），然后存入数据，然后判断是否需要扩容；
- 不是首次 `put`，则不再初始化，直接存入数据，然后判断是否需要扩容；

### 4、`ConcurrentHashMap` 的存储结构是怎样的？

- Java7 中 `ConcurrentHashMap` 使用的分段锁，也就是每一个 `Segment` 上同时只有一个线程可以操作，每一个 `Segment` 都是一个类似 `HashMap` 数组的结构，它可以扩容，它的冲突会转化为链表。但是 `Segment` 的个数一但初始化就不能改变，默认 `Segment` 的个数是 16 个。
- Java8 中的 `ConcurrentHashMap` 使用的 Synchronized 锁加 CAS 的机制。结构也由 Java7 中的 `Segment 数组 + HashEntry 数组 + 链表` 进化成了 `Node 数组 + 链表 / 红黑树`，`Node` 是类似于一个 `HashEntry` 的结构。它的冲突再达到一定大小时会转化成红黑树，在冲突小于一定数量时又退回链表。

## 5、线程池大小如何设置？

- **CPU 密集型任务(N+1)**: 这种任务消耗的主要 CPU 资源，可以将线程数设置为  $N$  ( $CPU$  核心数) +1，比  $CPU$  核心数多出来的一个线程是为了防止线程偶发的缺页中断，或者其它原因导致的任务暂停而带来的影响。一旦任务暂停， $CPU$  就会处于空闲状态，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)**: 这种任务应用起来，系统会用大部分的时间来处理 I/O 交互，而线程在处理 I/O 的时间段内不会占用  $CPU$  来处理，这时就可以将  $CPU$  交给其它线程使用。因此在 I/O 密集型任务的应用中，我们可以多配置一些线程，具体的计算方法是  $2N$ 。

### 如何判断是 CPU 密集任务还是 IO 密集任务？

$CPU$  密集型简单理解就是利用  $CPU$  计算能力的任务比如你在内存中对大量数据进行排序。单凡涉及到网络读取，文件读取这类都是  $IO$  密集型，这类任务的特点是  $CPU$  计算耗费时间相比于等待  $IO$  操作完成的时间来说很少，大部分时间都花在了等待  $IO$  操作完成上。

## 6、 $IO$ 密集= $Ncpu*2$ 是怎么计算出来？

- **I/O 密集型任务**应用起来，系统会用大部分的时间来处理 I/O 交互，而线程在处理 I/O 的时间段内不会占用  $CPU$  来处理，这时就可以将  $CPU$  交给其它线程使用。因此在 I/O 密集型任务的应用中，我们可以多配置一些线程。例如：数据库交互，文件上传下载，网络传输等。 $IO$  密集型，即该任务需要大量的  $IO$ ，即大量的阻塞，故需要多配置线程数。

## 7、G1 收集器有哪些特点？

- G1 的全称是 Garbage-First，意为垃圾优先，哪一块的垃圾最多就优先清理它。
- G1 GC 最主要的设计目标是：将 STW 停顿的时间和分布，变成可预期且可配置的。被视为 JDK1.7 中 HotSpot 虚拟机的一个重要进化特征。它具备一下特点：
- **并发与并发**: G1 能充分利用  $CPU$ 、多核环境下的硬件优势，使用多个  $CPU$  ( $CPU$  或者  $CPU$  核心) 来缩短 Stop-The-World 停顿时间。部分其他收集器原本需要停顿 Java 线程执行的 GC 动作，G1 收集器仍然可以通过并发的方式让 java 程序继续执行。
- **分代收集**: 虽然 G1 可以不需要其他收集器配合就能独立管理整个 GC 堆，但是还是保留了分代的概念。
- **空间整合**: 与 CMS 的“标记-清理”算法不同，G1 从整体来看是基于“标记-整理”算法实现的收集器；从局部上来看是基于“标记-复制”算法实现的。
- **可预测的停顿**: 这是 G1 相对于 CMS 的另一个大优势，降低停顿时间是 G1 和 CMS 共同的关注点，但 G1 除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为 M 毫秒的时间片段内。

G1 收集器在后台维护了一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的 Region (这也就是它的名字 Garbage-First 的由来)

## 8、你有哪些手段来排查 OOM 的问题？

- 增加两个参数 -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/tmp/heapdump.hprof, 当 OOM 发生时自动 dump 堆内存信息到指定目录。
- 同时 jstat 查看监控 JVM 的内存和 GC 情况，先观察问题大概出在什么区域。
- 使用 MAT 工具载入到 dump 文件，分析大对象的占用情况，比如 HashMap 做缓存未清理，时间长了就会内存溢出，可以把改为弱引用。

## 9、请你谈谈 MySQL 事务隔离级别，MySQL 的默认隔离级别是什么？

为了达到事务的四大特性，数据库定义了 4 种不同的事务隔离级别：

- READ-UNCOMMITTED（读取未提交）：最低的隔离级别，允许脏读，也就是可能读取到其他会话中未提交事务修改的数据，可能会导致脏读、幻读或不可重复读。
- READ-COMMITTED（读取已提交）：只能读取到已经提交的数据。Oracle 等多数数据库默认都是该级别（不重复读），可以阻止脏读，但是幻读或不可重复读仍有可能发生。
- REPEATABLE-READ（可重复读）：对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。
- SERIALIZABLE（可串行化）：最高的隔离级别，完全服从 ACID 的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。
- MySQL 默认采用的 REPEATABLE\_READ 隔离级别。

## 10、可重复读解决了哪些问题？

- 可重复读的核心就是一致性读(consistent read);保证多次读取同一个数据时，其值都和事务开始时候的内容是一致，禁止读取到别的事务未提交的数据，会造成幻读。
- 而事务更新数据的时候，只能用当前读。如果当前的记录的行锁被其他事务占用的话，就需要进入锁等待。
- 查询只承认在事务启动前就已经提交完成的数据。
- 可重复读解决的是重复读的问题，可重复读在快照读的情况下不会有幻读，但当前读的时候会有幻读。

## 11、对 SQL 慢查询会考虑哪些优化？

- 分析语句，是否加载了不必要的字段/数据。
- 分析 SQL 执行计划（explain extended），思考可能的优化点，是否命中索引等。
- 查看 SQL 涉及的表结构和索引信息。

- 
- 如果 SQL 很复杂，优化 SQL 结构。
  - 按照可能的优化点执行表结构变更、增加索引、SQL 改写等操作。
  - 查看优化后的执行时间和执行计划。
  - 如果表数据量太大，考虑分表。
  - 利用缓存，减少查询次数。

## 12、谈一谈缓存穿透、缓存击穿和缓存雪崩，以及解决办法？

### 缓存穿透

- 问题：大量并发查询不存在的 KEY，在缓存和数据库中都不存在，同时给缓存和数据库带来压力。
- 原因：一般而言，缓存穿透有 2 种可能性：业务数据被误删，导致缓存和数据库中都没有数据。恶意进行 ddos 攻击。
- 分析：为什么会多次透传呢？不存在一直为空，需要注意让缓存能够区分 KEY 不存在和查询到一个空值。
- 解决办法：缓存空值的 KEY，这样第一次不存在也会被加载会记录，下次拿到有这个 KEY。Bloom 过滤或 RoaringBitmap 判断 KEY 是否存在，如果布隆过滤器中没有查到这个数据，就不去数据库中查。在处理请求前增加恶意请求检查，如果检测到是恶意攻击，则拒绝进行服务。完全以缓存为准，使用延迟异步加载的策略（异步线程负责维护缓存的数据，定期或根据条件触发更新），这样就不会触发更新。

### 缓存击穿

- 问题：某个 KEY 失效的时候，正好有大量并发请求访问这个 KEY。
- 分析：跟穿透其实很像，属于比较偶然的。
- 解决办法：KEY 的更新操作添加全局互斥锁。完全以缓存为准，使用延迟异步加载的策略（异步线程负责维护缓存的数据，定期或根据条件触发更新），这样就不会触发更新。

### 缓存雪崩

- 问题：当某一时刻发生大规模的缓存失效的情况，导致大量的请求无法获取数据，从而将流量压力传导到数据库上，导致数据库压力过大甚至宕机。
- 原因：一般而言，缓存雪崩有 2 种可能性：大量的数据同一个时间失效：比如业务关系强相关的数据要求同时失效 Redis 宕机
- 分析：一般来说，由于更新策略、或者数据热点、缓存服务宕机等原因，可能会导致缓存数据同一个时间点大规模不可用，或者都更新。所以，需要我们的更新策略要在时间上合适，数据要均匀分享，缓存服务器要多台高可用。
- 解决办法：更新策略在时间上做到比较平均。如果数据需要同一时间失效，可以给这批数据加上一些随机值，使得这批数据不要在同一个时间过期，降低数据库的压力。使用的热数据尽量分散到不同的机器上。多台机器做主从复制或者多副本，实现高可用。做好主从的部署，当主节点挂掉后，能快速的使用从结点顶上。实现熔断限流机制，对系统进行负载能力控制。对于非核心功能的业务，拒绝其请求，只允许核心功能业务访问数据库获取数据。服务降价：提供默认返回值，或简单的提示信息。

## 13、LRU 是什么？如何实现？

最近最少使用策略 LRU (Least Recently Used) 是一种缓存淘汰算法，是一种缓存淘汰机制。

- 使用双向链表实现的队列，队列的最大容量为缓存的大小。在使用过程中，把最近使用的页面移动到队列头，最近没有使用的页面将被放在队列尾的位置
- 使用一个哈希表，把页号作为键，把缓存在队列中的节点的地址作为值，只需要把这个页对应的节点移动到队列的前面，如果需要的页面在内存中，此时需要把这个页面加载到内存中，简单的说，就是将一个新节点添加到队列前面，并在哈希表中跟新相应的节点地址，如果队列是满的，那么就从队尾移除一个节点，并将新节点添加到队列的前面。

## 14、什么是堆内存？参数如何设置？

堆内存是指由程序代码自由分配的内存，与栈内存作区分。

在 Java 中，堆内存主要用于分配对象的存储空间，只要拿到对象引用，所有线程都可以访问堆内存。

- `-Xmx`, 指定最大堆内存。如 `-Xmx4g`. 这只是限制了 Heap 部分的最大值为 4g。这个内存不包括栈内存，也不包括堆外使用的内存。
- `-Xms`, 指定堆内存空间的初始大小。如 `-Xms4g`。而且指定的内存大小，并不是操作系统实际分配的初始值，而是 GC 先规划好，用到才分配。专用服务器上需要保持 `-Xms` 和 `-Xmx` 一致，否则应用刚启动可能就有好几个 FullGC。当两者配置不一致时，堆内存扩容可能会导致性能抖动。
- `-Xmn`, 等价于 `-XX:NewSize`, 使用 G1 垃圾收集器 不应该 设置该选项，在其他的某些业务场景下可以设置。官方建议设置为 `-Xmx` 的  $1/2 \sim 1/4$ .
- `-XX: MaxPermSize=size`, 这是 JDK1.7 之前使用的。Java8 默认允许的 Meta 空间无限大，此参数无效。
- `-XX: MaxMetaspaceSize=size`, Java8 默认不限制 Meta 空间，一般不允许设置该选项。
- `-XX: MaxDirectMemorySize=size`, 系统可以使用的最大堆外内存，这个参数跟 `-Dsun.nio.MaxDirectMemorySize` 效果相同。
- `-Xss`, 设置每个线程栈的字节数。例如 `-Xss1m` 指定线程栈为 1MB，与 `-XX:ThreadStackSize=1m` 等价

## 15、栈和队列，举个使用场景例子？

- 栈（后进先出）可以用于字符匹配，数据反转等场景
- 队列（先进先出）可以用于任务队列，共享打印机等场景

## 16、MySQL 为什么 InnoDB 是默认引擎？

---

聚集索引是指数据库表行中数据的物理顺序与键值的逻辑（索引）顺序相同。一个表只能有一个聚簇索引，因为一个表的物理顺序只有一种情况，所以，对应的聚簇索引只能有一个。聚簇索引的叶子节点就是数据节点，既存储索引值，又在叶子节点存储行数据。

Innodb 创建表后生成的文件有：

frm:创建表的语句

idb:表里面的数据+索引文件

## 17、MySQL 索引底层结构为什么使用 B+树？

- 哈希虽然能够提供  $O(1)$  的单数据行操作性能，但是对于范围查询和排序却无法很好地支持，最终导致全表扫描；B 树能够在非叶节点中存储数据，但是这也导致在查询连续数据时可能会带来更多的随机 I/O，而 B+树的所有叶节点可以通过指针相互连接，能够减少顺序遍历时产生的额外随机 I/O；
- 第一，B 树一个节点里存的是数据，而 B+树存储的是索引（地址），所以 B 树里一个节点存不了很多个数据，但是 B+树一个节点能存很多索引，B+树叶子节点存所有的数据。
- 第二，B+树的叶子节点是数据阶段用了一个链表串联起来，便于范围查找。

## 18、B+ 树的叶子节点链表是单向还是双向？

双向链表



## 19、MVCC 是什么？它的底层原理是什么？

MVCC，多版本并发控制，它是通过读取历史版本的数据，来降低并发事务冲突，从而提高并发性能的一种机制。

- 事务版本号
- 表的隐藏列
- undo log
- read view

## 20、undo log 具体怎么回滚事务？

举个例子：

- 对于 insert 类型的 sql，会在 undo log 中记录下方才你 insert 进来的数据的 ID，当你想 roll back 时，根据 ID 完成精准的删除。
- 对于 delete 类型的 sql，会在 undo log 中记录下方你删除的数据，当你回滚时会将删除前的数据 insert 进去。
- 对于 update 类型的 sql，会在 undo log 中记录下修改前的数据，回滚时只需要反向 update 即可。
- 对于 select 类型的 sql，别费心了，select 不需要回滚。

## 21、如何查询慢 SQL 产生的原因

- 分析 SQL 执行计划（explain extended），思考可能的优化点，是否命中索引等。
- 没有索引或者没有用到索引(这是查询慢最常见的问题，是程序设计的缺陷)。
- 内存不足。
- 网络速度慢。
- 是否查询出的数据量过大（可以采用多次查询，其他的方法降低数据量）。
- 是否返回了不必要的行和列。
- 锁或者死锁。
- I/O 吞吐量小，形成了瓶颈效应。
- sp\_lock,sp\_who,活动的用户查看,原因是读写竞争资源。

## 22、索引失效的情况有哪些？

- like 以%开头索引无效，当 like 以&结尾，索引有效。
- or 语句前后没有同事使用索引，当且仅当 or 语句查询条件的前后列均为索引时，索引生效。
- 组合索引，使用的不是第一列索引时候，索引失效，即最左匹配规则。
- 数据类型出现隐式转换，如 varchar 不加单引号的时候可能会自动转换为 int 类型，这个时候索引失效。
- 在索引列上使用 IS NULL 或者 IS NOT NULL 时候，索引失效，因为索引是不索引空值得。
- 在索引字段上使用，NOT、<>、!=、时候是不会使用索引的，对于这样的处理只会进行全表扫描。
- 对索引字段进行计算操作，函数操作时不会使用索引。
- 当全表扫描速度比索引速度快的时候不会使用索引。

## 23、一个 Redis 实例最多能存放多少的 keys？List、Set、Sorted Set 他们最多能存放多少元素？

理论上 Redis 可以处理多达 232 的 keys，并且在实际中进行了测试，每个实例至少存放了 2 亿 5 千万的 keys。我们正在测试一些较大的值。任何 list、set、和 sorted set 都可以放 232 个元素。换句话说，Redis 的存储极限是系统中的可用内存值。

## 24、Redis 数据结构 压缩列表和跳跃表的区别

- 压缩列表（ziplist）本质上就是一个字节数组，是 Redis 为了节约内存而设计的一种线性数据结构，可以包含多个元素，每个元素可以是一个字节数组或一个整数。

- 
- 跳跃表（skiplist）是一种有序数据结构，它通过在每个节点中维持多个指向其他节点的指针，从而达到快速访问节点的目的。跳跃表支持平均  $O(\log N)$ 、最坏  $O(N)$  复杂度的节点查找，还可以通过顺序性操作来批量处理节点。

## 25、为什么数据量小的时候用压缩列表？

为了省内存。

## 26、Redis 主从同步是怎么实现的？

### 全量同步

master 服务器会开启一个后台进程用于将 redis 中的数据生成一个 rdb 文件，与此同时，服务器会缓存所有接收到的来自客户端的写命令（包含增、删、改），当后台保存进程处理完毕后，会将该 rdb 文件传递给 slave 服务器，而 slave 服务器会将 rdb 文件保存在磁盘并通过读取该文件将数据加载到内存，在此之后 master 服务器会将在此期间缓存的命令通过 redis 传输协议发送给 slave 服务器，然后 slave 服务器将这些命令依次作用于自己本地的数据集上最终达到数据的一致性。

### 增量同步

从 redis 2.8 版本以前，并不支持部分同步，当主从服务器之间的连接断掉之后，master 服务器和 slave 服务器之间都是进行全量数据同步。

从 redis 2.8 开始，即使主从连接中途断掉，也不需要进行全量同步，因为从这个版本开始融入了部分同步的概念。部分同步的实现依赖于在 master 服务器内存中给每个 slave 服务器维护了一份同步日志和同步标识，每个 slave 服务器在跟 master 服务器进行同步时都会携带自己的同步标识和上次同步的最后位置。当主从连接断掉之后，slave 服务器隔断时间（默认 1s）主动尝试和 master 服务器进行连接，如果从服务器携带的偏移量标识还在 master 服务器上的同步备份日志中，那么就从 slave 发送的偏移量开始继续上次的同步操作，如果 slave 发送的偏移量已经不再 master 的同步备份日志中（可能由于主从之间断掉的时间比较长或者在断掉的短暂时间内 master 服务器接收到大量的写操作），则必须进行一次全量更新。在部分同步过程中，master 会将本地记录的同步备份日志中记录的指令依次发送给 slave 服务器从而达到数据一致。

### Redis 主从同步策略

主从刚刚连接的时候，进行全量同步；全同步结束后，进行增量同步。当然，如果有需要，slave 在任何时候都可以发起全量同步。redis 策略是，无论如何，首先会尝试进行增量同步，如不成功，要求从机进行全量同步。

## 27、Redis 持久化 RDB 和 AOF 优缺点

### RDB

---

RDB 持久化方式，是将 Redis 某一时刻的数据持久化到磁盘中，是一种快照式的持久化方法。

#### RDB 优点：

- RDB 是一个非常紧凑（有压缩）的文件,它保存了某个时间点的数据,非常适用于数据的备份。
- RDB 作为一个非常紧凑（有压缩）的文件，可以很方便传送到另一个远端数据中心，非常适用于灾难恢复。
- RDB 在保存 RDB 文件时父进程唯一需要做的就是 fork 出一个子进程,接下来的工作全部由子进程来做，父进程不需要再做其他 IO 操作，所以 RDB 持久化方式可以最大化 redis 的性能。
- 与 AOF 相比，在恢复大的数据集的时候，RDB 方式会更快一些。

#### RDB 缺点：

- Redis 意外宕机时，会丢失部分数据。
- 当 Redis 数据量比较大时，fork 的过程是非常耗时的，fork 子进程时是会阻塞的，在这期间 Redis 是不能响应客户端的请求的。

#### AOF

AOF 方式是将执行过的写指令记录下来，在数据恢复时按照从前到后的顺序再将指令都执行一遍。

#### AOF 优点：

- 使用 AOF 会让你的 Redis 更加持久化。
- AOF 文件是一个只进行追加的日志文件，不需要在写入时读取文件。
- Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写。
- AOF 文件可读性高，分析容易。

#### AOF 缺点：

- 对于相同的数据来说，AOF 文件大小通常要大于 RDB 文件。
- 根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB。

## 28、谈谈自己对于 Spring AOP 的了解？

AOP(Aspect-Oriented Programming:面向切面编程)能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可拓展性和可维护性。

## 29、Spring Bean 容器的生命周期是什么样的？

- Bean 容器找到配置文件中 Spring Bean 的定义。
- Bean 容器利用 Java Reflection API 创建一个 Bean 的实例。
- 如果涉及到一些属性值 利用 set()方法设置一些属性值。

- 如果 Bean 实现了 `BeanNameAware` 接口，调用 `setBeanName()`方法，传入 Bean 的名字。
- 如果 Bean 实现了 `BeanClassLoaderAware` 接口，调用 `setBeanClassLoader()`方法，传入 `ClassLoader` 对象的实例。
- 如果 Bean 实现了 `BeanFactoryAware` 接口，调用 `setBeanFactory()`方法，传入 `BeanFactory` 对象的实例。
- 与上面的类似，如果实现了其他 `*.Aware` 接口，就调用相应的方法。
- 如果有和加载这个 Bean 的 Spring 容器相关的 `BeanPostProcessor` 对象，执行 `postProcessBeforeInitialization()` 方法
- 如果 Bean 实现了 `InitializingBean` 接口，执行 `afterPropertiesSet()`方法。
- 如果 Bean 在配置文件中的定义包含 `init-method` 属性，执行指定的方法。
- 如果有和加载这个 Bean 的 Spring 容器相关的 `BeanPostProcessor` 对象，执行 `postProcessAfterInitialization()` 方法
- 当要销毁 Bean 的时候，如果 Bean 实现了 `DisposableBean` 接口，执行 `destroy()` 方法。
- 当要销毁 Bean 的时候，如果 Bean 在配置文件中的定义包含 `destroy-method` 属性，执行指定的方法。

## 30、RabbitMQ 如何保证消息不丢失？

生产者：



方案 1：开启 RabbitMQ 事务(同步，性能差)

方案 2：开启 confirm 模式(异步，性能较好)

MQ：(1)exchange 持久化 (2)queue 持久化 (3)消息持久化

消费者：关闭自动 ACK

2021 【阿里】面试真题：

## 1、TCP 和 UDP 区别？

- TCP 基于连接，UDP 基于无连接。
- TCP 要求系统资源较多，UDP 较少。
- UDP 程序结构较简单。
- TCP 保证数据正确性，UDP 可能丢包。
- TCP 保证数据顺序，UDP 不保证。

## 2、TCP/IP 协议涉及哪几层架构？

---

应用层 传输层 互连网络层 网络接口层。

### 3、描述下 TCP 连接 4 次挥手的过程？为什么要 4 次挥手？

因为 TCP 是全双工，每个方向都必须进行单独关闭。关闭连接时，当 Server 端收到 FIN 报文时，很可能并不会立即关闭 SOCKET，所以只能先回复一个 ACK 报文，告诉 Client 端，“你发的 FIN 报文我收到了”。只有等到 Server 端所有的报文都发送完了，我才能发送 FIN 报文，因此不能一起发送。故需要四步握手。

### 4、计算机插上电源操作系统做了什么？

- 加电——打开电源开关，给主板和内部风扇供电。
- 启动引导程序——CPU 开始执行存储在 ROM BIOS 中的指令。
- 开机自检——计算机对系统的主要部件进行诊断测试。
- 加载操作系统——计算机将操作系统文件从磁盘读到内存中。
- 检查配置文件，定制操作系统的运行环境——读取配置文件，根据用户的设置对操作系统进行定制。
- 准备读取命令和数据——计算机等待用户输入命令和数据。

### 5、Linux 操作系统设备文件有哪些？

字符设备、块设备。

### 6、多线程同步有哪些方法？

- 使用 synchronized 关键字
- wait 和 notify
- 使用特殊域变量 volatile 实现线程同步
- 使用重入锁实现线程同步
- 使用局部变量来实现线程同步
- 使用阻塞队列实现线程同步
- 使用原子变量实现线程同步

### 7、一个对象的两个方法加 synchronized，一个线程进去 sleep，另一个线程可以进入到另一个方法吗？

不能。

### 8、什么是可重入锁（ReentrantLock）？

---

举例来说明锁的可重入性

```
public class UnReentrant{  
    Lock lock = new Lock();  
    public void outer(){  
        lock.lock();  
        inner();  
        lock.unlock();  
    }  
    public void inner(){  
        lock.lock();  
        //do something  
        lock.unlock();  
    }  
}
```

outer 中调用了 inner, outer 先锁住了 lock, 这样 inner 就不能再获取 lock。其实调用 outer 的线程已经获取了 lock 锁, 但是不能在 inner 中重复利用已经获取的锁资源, 这种锁即称之为不可重入可重入就意味着: 线程可以进入任何一个它已经拥有的锁所同步着的代码块。

synchronized、ReentrantLock 都是可重入的锁, 可重入锁相对来说简化了并发编程的开发。

## 9、创建线程的三个方法是什么？

- 通过继承 Thread 类创建线程类。
- 实现 Runnable 接口创建线程类。
- 通过 Callable 和 Future 接口创建线程。

## 10、Java 怎么获取多线程的返回值？

- 主线程等待。
- 使用 Thread 的 join 阻塞当前线程等待。
- 实现 Callable 接口（通过 FutureTask 或线程池的 Future）。

## 11、线程池有哪几种创建方式？

Java 通过 Executors (jdk1.5 并发包) 提供四种线程池, 分别为:

- newCachedThreadPool 创建一个可缓存线程池, 如果线程池长度超过处理需要, 可灵活回收空闲线程, 若无可回收, 则新建线程。
- newFixedThreadPool 创建一个定长线程池, 可控制线程最大并发数, 超出的线程会在队列中等待。

- 
- newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。
  - newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

## 12、线程池参数有哪些？

- corePoolSize 核心线程大小。
- maximumPoolSize 线程池最大线程数量。
- keepAliveTime 空闲线程存活时间。
- unit 空间线程存活时间单位。
- workQueue 工作队列。
- threadFactory 线程工厂。
- handler 拒绝策略。

## 13、线程池拒绝策略有哪些？

- ThreadPoolExecutor.AbortPolicy: 丢弃任务并抛出 RejectedExecutionException 异常（默认拒绝策略）。
- ThreadPoolExecutor.DiscardPolicy: 丢弃任务，但是不抛出异常。
- ThreadPoolExecutor.DiscardOldestPolicy: 丢弃队列最前面的任务，然后重新提交被拒绝的任务。
- ThreadPoolExecutor.CallerRunsPolicy: 由调用线程（提交任务的线程）处理该任务。

## 14、你认为对线程池的核心参数实现自定义可配置，三个核心参数是？

- corePoolSize : 核心线程数线程数定义了最小可以同时运行的线程数量。
- maximumPoolSize : 当队列中存放的任务达到队列容量的时候，当前可以同时运行的线程数量变为最大线程数。
- workQueue: 当新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，任务就会被存放在队列中。

## 15、ThreadPoolExecutor 线程池，corePoolSize=5， maximumPoolSize=10，queueCapacity=10，有 20 个耗时任务 交给这个 线程池执行，线程池会如何执行这 20 个任务？

- 如果当前线程数<corePoolSize，如果是则创建新的线程执行该任务。
- 如果当前线程数>=corePoolSize，则将任务存入 BlockingQueue。
- 如果阻塞队列已满，且当前线程数<maximumPoolSize，则新建线程执行该任务。
- 如果阻塞队列已满，且当前线程数>=maximumPoolSize，则抛出异常。

- 
- RejectedExecutionException，告诉调用者无法再接受任务了。

## 16、给用户发消息任务超出队列，你用哪个拒绝策略？有其他方法吗？

ThreadPoolExecutor.CallerRunsPolicy

- 无界队列（LinkedBlockingQuene），继续添加任务到阻塞队列中等待执行。
- 用消息队列存任务数据，线程池慢慢处理。

## 17、Java8 新特性有哪些了解？

- 接口的默认方法
- Lambda 表达式
- 函数式接口
- 方法和构造函数引用
- Lamda 表达式作用域
- 内置函数式接口
- Optional
- Streams(流)
- Parallel Streams(并行流)
- Maps
- Date API(日期相关 API)
- Annotations(注解)

## 18、什么时候用多线程、为什么要设计多线程？

### 高并发

系统接受实现多用户多请求的高并发时，通过多线程来实现。

### 线程后台处理大任务

一个程序是线性执行的。如果程序执行到要花大量时间处理的任务时，那主程序就得等待其执行完才能继续执行下面的。那用户就不得不等待它执行完。

这时候可以开线程把花大量时间处理的任务放在线程处理，这样线程在后台处理时，主程序也可以继续执行下去，用户就不需要等待。线程执行完后执行回调函数。

### 大任务

大任务处理起来比较耗时，这时候可以起到多个线程并行加快处理（例如：分片上传）。

好处：可以提高 CPU 的利用率。在多线程程序中，一个线程必须等待的时候，CPU 可以运行其他的线程而不是等待，这样就大大提高了程序的效率。也就是说允许单个程序创建多个并行执行的线程来完成各自的任务。

## 19、多线程越多效率越高吗？

---

不是

当线程总数较少时，线程越多，效率越高。

当线程总数较多时，由于线程本身调用耗时，线程越多，效率越低。

线程数越多会造成：

- 线程的生命周期开销非常高
- 消耗过多的 CPU 资源。

## 20、多线程会产生哪些并发问题？

安全性问题：在单线程系统上正常运行的代码，在多线程环境中可能会出现意料之外的结果。

活跃性问题：不正确的加锁、解锁方式可能会导致死锁 or 活锁问题。

性能问题：多线程并发即多个线程切换运行，线程切换会有一定的消耗并且不正确的加锁。

## 21、Mybatis 如何将对象转换成 SQL？

SQL 绑定是在加载 Mybatis 配置文件，然后扫描到哪个 mapper 子节点，再加载 mapper 映射文件，扫描里面的 SQL 节点，然后封装成对象（MappedStatement，在这个对象的 SqlSource 封装着 sql 语句）。所有的配置信息保存在 Configuration 类，最后动态代理执行的时候，取出来封装 sql 的对象，执行 sql。

## 22、虚拟内存是什么，虚拟内存的原理是什么？

虚拟内存是计算机系统内存管理的一种技术。

虚拟内存有以下两个优点：

- 虚拟内存地址空间是连续的，没有碎片。
- 虚拟内存的最大空间就是 cup 的最大寻址空间，不受内存大小的限制，能提供比内存更大的地址空间。

当每个进程创建的时候，内核会为每个进程分配虚拟内存，这个时候数据和代码还在磁盘上，当运行到对应的程序时，进程去寻找页表，如果发现页表中地址没有存放在物理内存上，而是在磁盘上，于是发生缺页异常，于是将磁盘上的数据拷贝到物理内存中并更新页表，下次再访问该虚拟地址时就能命中了。

## 23、栈会溢出吗？什么时候溢出？方法区会溢出吗？

栈是线程私有的，它的生命周期与线程相同，每个方法在执行的时候都会创建一个栈帧，用来存储局部变量表，操作数栈，动态链接，方法出口等信息。局部变量表又包含基本数据类型，对象引用类型。如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出

StackOverflowError 异常，方法递归调用产生这种结果。如果 Java 虚拟机栈可以动态扩展，并且扩展的动作已经尝试过，但是无法申请到足够的内存去完成扩展，或者在新建立线程的时

---

候没有足够的内存去创建对应的虚拟机栈，那么 Java 虚拟机将抛出一个 OutOfMemory 异常。(线程启动过多)。

方法区会发生溢出。

HotSpot jdk1.7 之前字符串常量池是方法区的一部分，方法区叫做“永久代”，在 1.7 之前无限的创建对象就会造成内存溢出，提示信息：PermGen space 而是用 jdk1.7 之后，开始逐步去永久代，就不会产生内存溢出。

方法区用于存放 Class 的相关信息，如类名、访问修饰符、常量池、字段描述、方法描述等，如果动态生成大量的 Class 文件，也会产生内存溢出。常见的场景还有：大量 JSP 或动态产生 JSP 文件的应用 (JSP 第一次运行时需要编译为 java 类)、基于 OSGi 的应用 (即使是同一个类文件，被不同的类加载器加载也会视为不同的类)。

## 24、JVM 如何加载类的？

JVM 类加载机制分为五个部分：加载，验证，准备，解析，初始化。

### 加载

加载是类加载过程中的一个阶段，这个阶段会在内存中生成一个代表这个类 `java.lang.Class` 对象，作为方法区这个类的各种数据的入口。注意这里不一定非得要从一个 Class 文件获取，这里既可以从 ZIP 包中读取（比如从 jar 包和 war 包中读取），也可以在运行时计算生成（动态代理），也可以由其它文件生成（比如将 JSP 文件转换成对应的 Class 类）。

### 验证

这一阶段的主要目的是为了确保 Class 文件的字节流中包含的信息是否符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

### 准备

准备阶段是正式为类变量分配内存并设置类变量的初始值阶段，即在方法区中分配这些变量所使用的内存空间。注意这里所说的初始值概念，比如一个类变量定义为：

实际上变量 v 在准备阶段过后的初始值为 0 而不是 8080，将 v 赋值为 8080 的 put static 指令是程序被编译后，存放于类构造器方法之中。

### 但是注意如果声明为：

```
public static final int v = 8080;
```

在编译阶段会为 v 生成 ConstantValue 属性，在准备阶段虚拟机会根据 ConstantValue 属性将 v 赋值为 8080。

### 解析

解析阶段是指虚拟机将常量池中的符号引用替换为直接引用的过程。符号引用就是 class 文件中的

```
public static int v = 8080;
```

实际上变量 v 在准备阶段过后的初始值为 0 而不是 8080，将 v 赋值为 8080 的 put static 指令是程序被编译后，存放于类构造器方法之中。但是注意如果声明为：

在编译阶段会为 v 生成 ConstantValue 属性，在准备阶段虚拟机会根据 ConstantValue 属性将 v 赋值为 8080。

## 初始化

初始化阶段是类加载最后一个阶段，前面的类加载阶段之后，除了在加载阶段可以自定义类加载器以外，其它操作都由 JVM 主导。到了初始阶段，才开始真正执行类中定义的 Java 程序代码。

## 25、自己写过 String 类能加载吗，之前的 String 是什么时候加载进去的？

不能加载，因为双亲委派机制，JVM 出于安全性的考虑，全限定类名相同的 String 是不能被加载的。

java.lang.String 会被顶级类加载器 Bootstrap Classloader 加载。当 class 文件被加载到内存中时，类文件常量池中的其他常量会加载到运行时常量池，但是字符串常量不会。它会首先在堆区中创建一个字符串对象，然后再把这个对象的引用保存到全局字符串常量池中。

## 26、描述 ThreadLocal（线程本地变量）的底层实现原理及常用场景？

**实现原理：**

- 每个 Thread 线程内部都有一个 ThreadLocalMap；以线程作为 key，泛型作为 value，可以理解为线程级别的缓存。每一个线程都会获得一个单独的 map。
- 提供了 set 和 get 等访问方法，这些方法为每个使用该变量的线程都存有一份独立的副本，因此 get 方法总是返回由当前执行线程在调用 set 时设置的最新值。

**应用场景：**

- JDBC 连接
- Session 管理
- Spring 事务管理
- 调用链，参数传递
- AOP

ThreadLocal 是一个解决线程并发问题的一个类，用于创建线程的本地变量，我们知道一个对象的所有线程会共享它的全局变量，所以这些变量不是线程安全的，我们可以使用同步技术。但是当我们不想使用同步的时候，我们可以选择 ThreadLocal 变量。例如，由于 JDBC 的连接对象不是线程安全的，因此，当多线程应用程序在没有协同的情况下，使用全局变量时，就不是线程安全的。通过将 JDBC 的连接对象保存到 ThreadLocal 中，每个线程都会拥有属于自己的连接对象副本。

## 27、什么是微服务架构？

微服务架构就是将单体的应用程序分成多个应用程序，这多个应用程序就成为微服务，每个微服务运行在自己的进程中，并使用轻量级的机制通信。这些服务围绕业务能力来划分，并通过自动化部署机制来独立部署。这些服务可以使用不同的编程语言，不同数据库，以保证最低限度的集中式管理。

## 28、微服务有哪些特点？

- **解耦** – 系统内的服务很大程度上是分离的。因此，整个应用程序可以轻松构建，更改和扩展
- **组件化** – 微服务被视为可以轻松更换和升级的独立组件
- **业务能力** – 微服务非常简单，专注于单一功能
- **自治** – 开发人员和团队可以彼此独立工作，从而提高速度
- **持续交付** – 通过软件创建，测试和批准的系统自动化，允许频繁发布软件
- **责任** – 微服务不关注应用程序作为项目。相反，他们将应用程序视为他们负责的产品
- **分散治理** – 重点是使用正确的工具来做正确的工作。这意味着没有标准化模式或任何技术模式。开发人员可以自由选择最有用的工具来解决他们的问题
- **敏捷** – 微服务支持敏捷开发。任何新功能都可以快速开发并再次丢弃

## 29、Lambda 表达式是啥？优缺点？

lambda 表达式，也被称为闭包，它是推动 Java 8 发布的最重要新特性。lambda 允许把函数作为一个方法的参数（函数作为参数传递进方法中），使用 Lambda 表达式可以使代码变得更加简洁紧凑。

**优点：**

- 代码更加简洁
- 减少匿名内部类的创建，节省资源
- 使用时不用去记忆所使用的接口和抽象函数

**缺点：**

- 不易于后期维护，必须熟悉 lambda 表达式和抽象函数中参数的类型
- 可读性差
- 若不用并行计算，很多时候计算速度没有比传统的 for 循环快。（并行计算有时需要预热才显示出效率优势）
- 不容易调试。
- 若其他程序员没有学过 lambda 表达式，代码不容易让其他语言的程序员看懂。

## 30、讲一下 Lambda 的表达式作用域（Lambda Scopes）。

**访问局部变量**

- 我们可以直接在 lambda 表达式中访问外部的局部变量：但是和匿名对象不同的是，这里的变量可以不用声明为 final，该代码同样正确，不过这里的变量必须不可被后面的代码修改（即隐性的具有 final 的语义）

**访问字段和静态变量**

- 与局部变量相比，我们对 lambda 表达式中的实例字段和静态变量都有读写访问权限。该行为和匿名对象是一致的。

## 访问默认接口方法

- 无法从 lambda 表达式中访问默认方法。

## 31、MySQL 事务的特性有什么，说一下分别是什么意思？

- 原子性：即不可分割性，事务要么全部被执行，要么就全部不被执行。
- 一致性或可串性。事务的执行使得数据库从一种正确状态转换成另一种正确状态。
- 隔离性。在事务正确提交之前，不允许把该事务对数据的任何改变提供给任何其他事务。  
持久性。事务正确提交后，其结果将永久保存在数据库中，即使在事务提交后有了其他故障，事务的处理结果也会得到保存。

2021 【京东】面试真题：

### 1、哪些情况下的对象会被垃圾回收机制处理掉？

利用可达性分析[算法](#)，虚拟机会将一些对象定义为 GCRoots，从 GCRoots 出发沿着引用链向下寻找，如果某个对象不能通过 GCRoots 寻找到，虚拟机就认为该对象可以被回收掉。

- 哪些对象可以被看做是 GCRoots 呢？
  - 1) 虚拟机栈（栈帧中的本地变量表）中引用的对象；
  - 2) 方法区中的类静态属性引用的对象，常量引用的对象；
  - 3) 本地方法栈中 JNI(Native 方法) 引用的对象；
- 对象不可达，一定会被垃圾收集器回收么？

即使不可达，对象也不一定会被垃圾收集器回收，1) 先判断对象是否有必要执行 finalize() 方法，对象必须重写 finalize() 方法且没有被运行过。2) 若有必要执行，会把对象放到一个队列中，JVM 会开一个线程去回收它们，这是对象最后一次可以逃逸清理的机会。

### 2、讲一下常见编码方式？

编码的意义：计算机中存储的最小单元是一个字节即 8bit，所能表示的字符范围是 255 个，而人类要表示的符号太多，无法用一个字节来完全表示，固需要将符号编码，将各种语言翻译成计算机能懂的语言。

- ASCII 码：总共 128 个，用一个字节的低 7 位表示，0~31 控制字符如换回车删除等；32~126 是打印字符，可通过键盘输入并显示出来；
- ISO-8859-1,用来扩展 ASCII 编码，256 个字符，涵盖了大多数西欧语言字符。
- GB2312:双字节编码，总编码范围是 A1-A7,A1-A9 是符号区，包含 682 个字符，B0-B7 是汉字区，包含 6763 个汉字；
- GBK 为了扩展 GB2312,加入了更多的汉字，编码范围是 8140~FEFE，有 23940 个码位，能表示 21003 个汉字。

- 
- UTF-16: ISO 试图想创建一个全新的超语言字典，世界上所有语言都可通过这本字典 Unicode 来相互翻译，而 UTF-16 定义了 Unicode 字符在计算机中存取方法，用两个字节来表示 Unicode 转化格式。不论什么字符都可用两字节表示，即 16bit，固叫 UTF-16。
  - UTF-8: UTF-16 统一采用两字节表示一个字符，但有些字符只用一个字节就可表示，浪费存储空间，而 UTF-8 采用一种变长技术，每个编码区域有不同的字码长度。不同类型的字符可以由 1~6 个字节组成。

### 3、utf-8 编码中的中文占几个字节；int 型几个字节？

utf-8 是一种变长编码技术，utf-8 编码中的中文占用的字节不确定，可能 2 个、3 个、4 个，int 型占 4 个字节。

### 4、静态代理和动态代理的区别，什么场景使用？

代理是一种常用的设计模式，目的是：为其他对象提供一个代理以控制对某个对象的访问，将两个类的关系解耦。代理类和委托类都要实现相同的接口，因为代理真正调用的是委托类的方法。

区别：

- 静态代理：由程序员创建或是由特定工具生成，在代码编译时就确定了被代理的类是哪一个，是静态代理。静态代理通常只代理一个类；
- 动态代理：在代码运行期间，运用反射机制动态创建生成。动态代理代理的是一个接口下的多个实现类；

实现步骤：

- a. 实现 InvocationHandler 接口创建自己的调用处理器；
- b. 给 Proxy 类提供 ClassLoader 和代理接口类型数组创建动态代理类；
- c. 利用反射机制得到动态代理类的构造函数；
- d. 利用动态代理类的构造函数创建动态代理类对象；

使用场景：Retrofit 中直接调用接口的方法；Spring 的 AOP 机制；

### 5、简述下 Java 的异常体系。

Java 中 Throwable 是所有异常和错误的超类，两个直接子类是 Error（错误）和 Exception（异常）：

- Error 是程序无法处理的错误，由 JVM 产生和抛出，如 OOM、ThreadDeath 等。这些异常发生时，JVM 一般会选择终止程序。
- Exception 是程序本身可以处理的异常，又分为运行时异常(RuntimeException)(也叫 Checked Exception)和非运行时异常(不检查异常 Unchecked Exception)。运行时异常有 NullPointerException\IndexOutOfBoundsException 等，这些异常一般

---

是由程序逻辑错误引起 的，应尽可能避免。非运行时异常有 IOException\SQLException\FileNotFoundException 以及 由用户自定义的 Exception 异常等。

## 6、谈谈你对解析与分派的认识。

解析指方法在运行前，即编译期间就可知的，有一个确定的版本，运行期间也不会改变。解析是静态的，在类加载的解析阶段就可将符号引用转变成直接引用。

分派可分为静态分派和动态分派，重载属于静态分派，覆盖属于动态分派。静态分派是指在重载时通过参数的静态类型而非实际类型作为判断依据，在编译阶段，编译器可根据参数的静态类型决定使用哪一个重载版本。动态分派则需要根据实际类型来调用相应的方法。

## 7、修改对象 A 的 equals 方法的签名，那么使用 HashMap 存放这个对象实例的时候，会用哪个 equals 方法？

会调用对象对象的 equals 方法。

“==”如果是基本类型的话就是看他们的数据值是否相等就可以。如果是引用类型的话，比较的是栈内存局部变量表中指向堆内存中的指针的值是否相等。“equals”如果对象的 equals 方法没有重写的话，equals 方法和 “==” 是同一种。hashcod 是返回对象实例内存地址的 hash 映射。理论上所有对象的 hash 映射都是不相同的。

## 8、Java 中实现多态的机制是什么？

多态是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编译时不确定，在运行期间才确定，一个引用变量到底会指向哪个类的实例。这样就可以不用 修改源程序，就可以让引用变量绑定到各种不同的类实现上。Java 实现多态有三个必要条件：继承、重定、向上转型，在多态中需要将子类的引用赋值给父类对象，只有这样该引用才能够具备调用父类方法和子类的方法。

## 9、如何将一个 Java 对象序列化到文件里？

ObjectOutputStream.writeObject()负责将指定的流写入，ObjectInputStream.readObject()从指 定流读取序列化数据。

```
//写入
try {
    ObjectOutputStream os = new ObjectOutputStream(new
FileOutputStream("D:/student.txt"));
    os.writeObject(studentList);
    os.close();
} catch (FileNotFoundExceptione) {
```

---

```
e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

## 10、说说你对 Java 反射的理解。

在运行状态中，对任意一个类，都能知道这个类的所有属性和方法，对任意一个对象，都能调用它的任意一个方法和属性。这种能动态获取信息及动态调用对象方法的功能称为 java 语言的反射机制。

反射的作用：开发过程中，经常会遇到某个类的某个成员变量、方法或属性是私有的，或只对系统应用开放，这里就可以利用 java 的反射机制通过反射来获取所需的私有成员或是方法。

- 获取类的 Class 对象实例 Class clz=Class.forName("com.zhenai.api.Apple");
- 根据 Class 对象实例获取 Constructor 对象 Constructor appConstructor = clz.getConstructor();
- 使用 Constructor 对象的 newInstance 方法获取反射类对象 Object appleObj = appConstructor.newInstance();
- 获取方法的 Method 对象  
Method setPriceMethod=clz.getMethod("setPrice",int.class);
- 利用 invoke 方法调用方法 setPriceMethod.invoke(appleObj,14);
- 通过 getFields() 可以获取 Class 类的属性，但无法获取私有属性，而 getDeclaredFields() 可以获取到包括私有属性在内的所有属性。带有 Declared 修饰的方法可以反射到私有的方法，没有 Declared 修饰的只能用来反射公有的方法，其他如 Annotation\Field\Constructor 也是如此。

## 11、说说你对 Java 注解的理解。

注解是通过@interface 关键字来进行定义的，形式和接口差不多，只是前面多了一个@  
public@interfaceTestAnnotation{  
}

使用时@TestAnnotation 来引用，要使注解能正常工作，还需要使用元注解，它是可以注解到注解上的注解。元标签有@Retention、@Documented、@Target、@Inherited 和 @Repeatable 五种。

@Retention 说明注解的存活时间，取值有 RetentionPolicy.SOURCE 注解只在源码阶段保留，在编译器进行编译时被丢弃；RetentionPolicy.CLASS 注解只保留到编译进行的时候，并不会被加载到 JVM 中。RetentionPolicy.RUNTIME 可以留到程序运行的时候，它会被加载进入到 JVM 中，所以在程序运行时可以获取到它们。

@Documented 注解中的元素包含到 javadoc 中去。

---

@Target 限定注解的应用场景，ElementType.FIELD 给属性进行注解； ElementType.LOCAL\_VARIABLE 可以给局部变量进行注解； ElementType.METHOD 可以给方法进行注解； ElementType.PACKAGE 可以给一个包进行注解 ElementType.TYPE 可以给一个类型进行注解，如类、接口、枚举。

@Inherited 若一个超类被@Inherited 注解过的注解进行注解，它的子类没有被任何注解应用的话，该子类就可继承超类的注解；

#### 注解的作用：

- 提供信息给编译器：编译器可利用注解来探测错误和警告信息
- 编译阶段：软件工具可以利用注解信息来生成代码、html 文档或做其它相应处理；
- 运行阶段：程序运行时可利用注解提取代码

注解是通过反射获取的，可以通过 Class 对象的 isAnnotationPresent() 方法判断它是否应用了某个注解，再通过 getAnnotation() 方法获取 Annotation 对象

## 12、说一下泛型原理，并举例说明。

泛型就是将类型变成参数传入，使得可以使用的类型多样化，从而实现解耦。Java 泛型是在 Java1.5 以后出现的，为保持对以前版本的兼容，使用了擦除的方法实现泛型。擦除是指在一定程度无视类型参数 T，直接从 T 所在的类开始向上 T 的父类去擦除，如调用泛型方法，传入类型参数 T 进入方法内部，若没在声明时做类似

public T methodName(TextendsFather t){}，Java 就进行了向上类型的擦除，直接把参数 t 当做 Object 类来处理，而不是传进去的 T。即在有泛型的任何类和方法内部，它都无法知道自己的泛型参数，擦除和转型都是在边界上发生，即传进去的参在进入类或方法时被擦除掉，但传出来的时候又被转成了我们设置的 T。在泛型类或方法内，任何涉及到具体类型（即擦除后的类型的子类）操作都不能进行，如 newT()，或者 T.play()（play 为某子类的方法而不是擦除后的类的方法）。

## 13、谈谈你对 Java 中 String 的了解。

- String 类是 final 型，固 String 类不能被继承，它的成员方法也都默认为 final 方法。String 对象一旦创建就固定不变了，对 String 对象的任何改变都不影响到原对象，相关的任何改变操作都会生成新的 String 对象。
- String 类是通过 char 数组来保存字符串的，String 对 equals 方法进行了重定，比较的是值相等。

String a="test";String b="test";String c=newString("test");

a、b 和字面上的 test 都是指向 JVM 字符串常量池中的"test"对象，他们指向同一个对象。而 new 关键字一定会产生一个对象 test，该对象存储在堆中。所以 newString("test") 产生了两个对象，保存在栈中的 c 和保存在堆中的 test。而在 java 中根本就不存在两个完全一模一样的字符串对象，故在堆中的 test 应该是引用字符串常量池中的 test。

例：

```
String str1="abc";//栈中开辟一块空间存放引用 str1, str1 指向池中 String 常量"abc"
String str2="def";//栈中开辟一块空间存放引用 str2, str2 指向池中 String 常量"def"
String str3=str1+str2;//栈中开辟一块空间存放引用 str3//str1+str2 通过 StringBuilder 的最后一步
toString()方法返回一个新的 String 对象"abcdef"
//会在堆中开辟一块空间存放此对象，引用 str3 指向堆中的(str1+str2)所返回的新 String 对象。
System.out.println(str3=="abcdef");//返回 false 因为 str3 指向堆中的"abcdef"对象，而
"abcdef"是字符池中的对象，所以结果为 false。JVM 对 Stringstr="abc"对象放在常量池是在编译
时做的， 而 Stringstr3=str1+str2 是在运行时才知道的，new 对象也是在运行时才做的。
```

## 14、String 为什么要设计成不可变的？

- 字符串常量池需要 String 不可变。因为 String 设计成不可变，当创建一个 String 对象时，若此字符串值已经存在于常量池中，则不会创建一个新的对象，而是引用已经存在的对象。如果字符串变量允许必变，会导致各种逻辑错误，如改变一个对象会影响到另一个独立对象。
- String 对象可以缓存 hashCode。字符串的不可变性保证了 hash 码的唯一性，因此可以缓存 String 的 hashCode，这样不用每次去重新计算哈希码。在进行字符串比较时，可以直接比较 hashCode，提高了比较性能；
- 安全性。String 被许多 java 类用来当作参数，如 url 地址，文件 path 路径，反射机制所需的 Strign 参数等，若 String 可变，将会引起各种安全隐患。

## 15、Redis 常见的几种数据结构说一下？各自的使用场景？

### string

介绍：string 数据结构是简单的 key-value 类型。

使用场景：一般常用在需要计数的场景，比如用户的访问次数、热点文章的点赞转发数量等等。

### list

介绍：list 即是 链表

使用场景：发布与订阅或者说消息队列、慢查询。

### hash

介绍：hash 类似于 JDK1.8 前的 HashMap，内部实现也差不多(数组 + 链表)。

使用场景：系统中对象数据的存储。

### set

介绍：set 类似于 Java 中的 HashSet。Redis 中的 set 类型是一种无序集合，集合中的元素没有先后顺序。当你需要存储一个列表数据，又不希望出现重复数据时，set 是一个很好的选择，并且 set 提供了判断某个成员是否在一个 set 集合内的重要接口，这个也是 list 所不能提供的。可以基于 set 轻易实现交集、并集、差集的操作

---

使用场景：需要存放的数据不能重复以及需要获取多个数据源交集和并集等场景。

### sorted set

介绍：和 set 相比，sorted set 增加了一个权重参数 score，使得集合中的元素能够按 score 进行有序排列，还可以通过 score 的范围来获取元素的列表。有点像是 Java 中 HashMap 和 TreeSet 的结合体。

使用场景：需要对数据根据某个权重进行排序的场景。比如在直播系统中，实时排行信息包含直播间在线用户列表，各种礼物排行榜，弹幕消息（可以理解为按消息维度的消息排行榜）等信息。

### bitmap

介绍：bitmap 存储的是连续的二进制数字（0 和 1），通过 bitmap，只需要一个 bit 位来表示某个元素对应的值或者状态，key 就是对应元素本身。我们知道 8 个 bit 可以组成一个 byte，所以 bitmap 本身会极大的节省储存空间。。

使用场景：适合需要保存状态信息（比如是否签到、是否登录...）并需要进一步对这些信息进行分析的场景。比如用户签到情况、活跃用户情况、用户行为统计（比如是否点赞过某个视频）。。

## 16、谈一谈缓存穿透、缓存击穿和缓存雪崩，以及各自的解决方案？

### 缓存穿透

- 问题：大量并发查询不存在的 KEY，在缓存和数据库中都不存在，同时给缓存和数据库带来压力。
- 原因：一般而言，缓存穿透有 2 种可能性：业务数据被误删，导致缓存和数据库中都没有数据。恶意进行 ddos 攻击。
- 分析：为什么会多次透传呢？不存在一直为空，需要注意让缓存能够区分 KEY 不存在和查询到一个空值。
- 解决办法：缓存空值的 KEY，这样第一次不存在也会被加载会记录，下次拿到有这个 KEY。Bloom 过滤或 RoaringBitmap 判断 KEY 是否存在，如果布隆过滤器中没有查到这个数据，就不去数据库中查。在处理请求前增加恶意请求检查，如果检测到是恶意攻击，则拒绝进行服务。完全以缓存为准，使用延迟异步加载的策略（异步线程负责维护缓存的数据，定期或根据条件触发更新），这样就不会触发更新。

### 缓存击穿

- 问题：某个 KEY 失效的时候，正好有大量并发请求访问这个 KEY。
- 分析：跟穿透其实很像，属于比较偶然的。
- 解决办法：KEY 的更新操作添加全局互斥锁。完全以缓存为准，使用延迟异步加载的策略（异步线程负责维护缓存的数据，定期或根据条件触发更新），这样就不会触发更新。

### 缓存雪崩

- 问题：当某一时刻发生大规模的缓存失效的情况，导致大量的请求无法获取数据，从而将流量压力传导到数据库上，导致数据库压力过大甚至宕机。

- 
- 原因：一般而言，缓存雪崩有 2 种可能性：大量的数据同一个时间失效：比如业务关系强相关的数据要求同时失效 Redis 容机
  - 分析：一般来说，由于更新策略、或者数据热点、缓存服务宕机等原因，可能会导致缓存数据同一个时间点大规模不可用，或者都更新。所以，需要我们的更新策略要在时间上合适，数据要均匀分享，缓存服务器要多台高可用。
  - 解决办法：更新策略在时间上做到比较平均。如果数据需要同一时间失效，可以给这批数据加上一些随机值，使得这批数据不要在同一个时间过期，降低数据库的压力。使用的热数据尽量分散到不同的机器上。多台机器做主从复制或者多副本，实现高可用。做好主从的部署，当主节点挂掉后，能快速的使用从结点顶上。实现熔断限流机制，对系统进行负载能力控制。对于非核心功能的业务，拒绝其请求，只允许核心功能业务访问数据库获取数据。服务降价：提供默认返回值，或简单的提示信息。

## 17、讲下 Kafka、RabbitMQ、RocketMQ 之间的区别是什么？

### 性能

消息中间件的性能主要衡量吞吐量，Kafka 的吞吐量比 RabbitMQ 要高出 1~2 个数量级，RabbitMQ 的单机 QPS 在万级别，Kafka 的单机 QPS 能够达到百万级别。RocketMQ 单机写入 TPS 单实例约 7 万条/秒，单机部署 3 个 Broker，可以跑到最高 12 万条/秒，消息大小 10 个字节，Kafka 如果开启幂等、事务等功能，性能也会有所降低。

### 数据可靠性

Kafka 与 RabbitMQ 都具备多副本机制，数据可靠性较高。RocketMQ 支持异步实时刷盘，同步刷盘，同步 Replication，异步 Replication。

### 服务可用性

Kafka 采用集群部署，分区与多副本的设计，使得单节点宕机对服务无影响，且支持消息容量的线性提升。RabbitMQ 支持集群部署，集群节点数量有多种规格。RocketMQ 是分布式架构，可用性高。

### 功能

Kafka 与 RabbitMQ 都是比较主流的两款消息中间件，具备消息传递的基本功能，但在一些特殊的功能方面存在差异，RocketMQ 在阿里集团内部有大量的应用在使用。

## 18、Kafka 的架构说一下？

整个架构中包括三个角色。

- 生产者（Producer）：消息和数据生产者。
- 代理（Broker）：缓存代理，Kafka 的核心功能。
- 消费者（Consumer）：消息和数据消费者。

Kafka 给 Producer 和 Consumer 提供注册的接口，数据从 Producer 发送到 Broker，Broker 承担一个中间缓存和分发的作用，负责分发注册到系统中的 Consumer。

## 19、Kafka 怎么保证消息是有序的？

消息在被追加到 Partition(分区)的时候都会分配一个特定的偏移量（offset）。Kafka 通过偏移量（offset）来保证消息在分区内的顺序性。发送消息的时候指定 key/Partition。

## 20、Kafka 怎么保证消息不丢失？

### 生产者丢失消息的情况

生产者(Producer) 调用 `send` 方法发送消息之后，消息可能因为网络问题并没有发送过去。为了确定消息是发送成功，我们要判断消息发送的结果，Kafka 生产者(Producer) 使用 `send` 方法发送消息实际上是异步的操作，我们可以通过 `get()`方法获取调用结果，但是这样也让它变为了同步操作，可以采用为其添加回调函数的形式，示例代码如下：

```
ListenableFuture<SendResult<String, Object>> future = kafkaTemplate.send(topic, o);
future.addCallback(result -> logger.info("生产者成功发送消息到 topic:{} partition:{} 的消息",
result.getRecordMetadata().topic(), result.getRecordMetadata().partition(),
ex -> logger.error("生产者发送消失败，原因：{}", ex.getMessage()));
```

Producer 的 `retries`（重试次数）设置一个比较合理的值，一般是 3，但是为了保证消息不丢失的话一般会设置比较大一点。设置完成之后，当出现网络问题之后能够自动重试消息发送，避免消息丢失。另外，建议还要设置重试间隔，因为间隔太小的话重试的效果就不明显了，网络波动一次你 3 次一下子就重试完了。

### 消费者丢失消息的情况

当消费者拉取到了分区的某个消息之后，消费者会自动提交了 offset。自动提交的话会有一个问题，试想一下，当消费者刚拿到这个消息准备进行真正消费的时候，突然挂掉了，消息实际上并没有被消费，但是 offset 却被自动提交了。

解决办法也比较粗暴，我们手动关闭自动提交 offset，每次在真正消费完消息之后再自己手动提交 offset。但是，细心的朋友一定会发现，这样会带来消息被重新消费的问题。比如你刚刚消费完消息之后，还没提交 offset，结果自己挂掉了，那么这个消息理论上就会被消费两次。

### Kafka 弄丢了消息

试想一种情况：假如 leader 副本所在的 broker 突然挂掉，那么就要从 follower 副本重新选出一个 leader，但是 leader 的数据还有一些没有被 follower 副本的同步的话，就会造成消息丢失。

当我们配置了 `unclean.leader.election.enable = false` 的话，当 leader 副本发生故障时就不会从 follower 副本中和 leader 同步程度达不到要求的副本中选择出 leader，这样降低了消息丢失的可能性。

## 21、Kafka 怎么解决重复消费？

- 
- 生产者发送每条数据的时候，里面加一个全局唯一的 id，消费到了之后，先根据这个 id 去比如 Redis 里查一下，之前消费过吗，如果没有消费过，就处理，然后这个 id 写 Redis。如果消费过就别处理了。
  - 基于数据库的唯一键来保证重复数据不会重复插入多条。因为有唯一键约束了，重复数据插入只会报错，不会导致数据库中出现脏数据。

## 22、介绍下 MySQL 聚簇索引与非聚簇索引的区别（InnoDB 与 Myisam 引擎）？

聚集索引是指数据库表行中数据的物理顺序与键值的逻辑（索引）顺序相同。一个表只能有一个聚簇索引，因为一个表的物理顺序只有一种情况，所以，对应的聚簇索引只能有一个。聚簇索引的叶子节点就是数据节点，既存储索引值，又在叶子节点存储行数据。

Innodb 创建表后生成的文件有：

frm:创建表的语句

idb:表里面的数据+索引文件

非聚集索引（MyISAM 引擎的底层实现）的逻辑顺序与磁盘上行的物理存储顺序不同。非聚簇索引的叶子节点仍然是索引节点，只不过有指向对应数据块的指针。索引命中后，需要回表查询。

Myisam 创建表后生成的文件有：

frm:创建表的语句 MYD:表里面的数据文件（myisam data）

MYI:表里面的索引文件（myisam index）

innodb 的次索引指向对主键的引用（聚簇索引）

myisam 的次索引和主索引都指向物理行（非聚簇索引）

## 23、然后给一个联合索引(a,b)和一个语句,select \* from table where b = 'xxx', 判断是否能命中索引？为什么？

不能命中。

对于查询 SELECT \* FROM TABLE WHERE a=xxx and b=xxx，显然是可以使用（a, b）这个联合索引的。

对于单个的 a 列查询 SELECT \* FROM TABLE WHERE a=xxx，也可以使用这个（a, b）索引。

但对于 b 列的查询 SELECT \* FROM TABLE WHERE b=xxx，则不可以使用这棵 B+树索引。

在 innodb 数据引擎中，可以发现叶子节点上的 b 值为 1、2、1、4、1、2，显然不是排序的，因此对于 b 列的查询使用不到（a, b）的索引

## 24、Java 多线程有哪几种实现方式？

- 
- 通过继承 Thread 类创建线程类
  - 实现 Runnable 接口创建线程类
  - 通过 Callable 和 Future 接口创建线程

## 25、用过 ConcurrentHashMap，讲一下他和 HashTable 的不同之处？

- HashTable 就是实现了 HashMap 加上了 synchronized，而 ConcurrentHashMap 底层采用分段的数组+链表实现，线程安全
- ConcurrentHashMap 通过把整个 Map 分为 N 个 Segment，可以提供相同的线程安全，但是效率提升 N 倍，默认提升 16 倍。
- 并且读操作不加锁，由于 HashEntry 的 value 变量是 volatile 的，也能保证读取到最新的值。
- Hashtable 的 synchronized 是针对整张 Hash 表的，即每次锁住整张表让线程独占，ConcurrentHashMap 允许多个修改操作并发进行，其关键在于使用了锁分离技术
- 扩容：段内扩容（段内元素超过该段对应 Entry 数组长度的 75% 触发扩容，不会对整个 Map 进行扩容），插入前检测需不需要扩容，有效避免无效扩容

## 26、Java 怎么实现线程安全？

- 使用同步代码块
- 使用同步方法
- 使用 Lock 锁机制，通过创建 Lock 对象，采用 lock() 加锁，unlock() 解锁，来保护指定的代码块

## 27、描述 ThreadLocal（线程本地变量）的底层实现原理及常用场景。

实现原理：

- 每个 Thread 线程内部都有一个 ThreadLocalMap；以线程作为 key，泛型作为 value，可以理解为线程级别的缓存。每一个线程都会获得一个单独的 map。
- 提供了 set 和 get 等访问方法，这些方法为每个使用该变量的线程都存有一份独立的副本，因此 get 方法总是返回由当前执行线程在调用 set 时设置的最新值。

应用场景：

- JDBC 连接
- Session 管理
- Spring 事务管理
- 调用链，参数传递
- AOP

ThreadLocal 是一个解决线程并发问题的一个类，用于创建线程的本地变量，我们知道一个对象的所有线程会共享它的全局变量，所以这些变量不是线程安全的，我们可以使用同步技术。

---

但是当我们不想使用同步的时候，我们可以选择 ThreadLocal 变量。例如，由于 JDBC 的连接对象不是线程安全的，因此，当多线程应用程序在没有协同的情况下，使用全局变量时，就不是线程安全的。通过将 JDBC 的连接对象保存到 ThreadLocal 中，每个线程都会拥有属于自己的连接对象副本。

## 28、介绍下 Spring Bean 都有哪些作用域？

- 单例 singleton : bean 在每个 Spring IOC 容器中只有一个实例。
- 原型 prototype: 一个 bean 的定义可以有多个实例。
- request: 每次 http 请求都会创建一个 bean。
- session: 在一个 HTTP Session 中，一个 bean 定义对应一个实例。
- globalsection
- application

## 29、注解 @Autowired 和 @Resource 有什么区别？

- Resource 是 JDK 提供的，而 Autowired 是 Spring 提供的
- Resource 不允许找不到 bean 的情况，而 Autowired 允许 (@Autowired(required = false))
- 指定 name 的方式不一样，@Resource(name = "baseDao"),@Autowired()@Qualifier("baseDao")

Resource 默认通过 name 查找，而 Autowired 默认通过 type 查找

- (1) @Autowired 与@Resource 都可以用来装配 bean，都可以写在字段或 setter 方法上
- (2) @Autowired 默认按类型装配，默认情况下必须要求依赖对象存在，如果要允许 null 值，可以设置它的 required 属性为 false。如果想使用名称装配可以结合@Qualifier 注解进行使用。
- (3) @Resource，默认按照名称进行装配，名称可以通过 name 属性进行指定，如果没有指定 name 属性，当注解写在字段上时，默认取字段名进行名称查找。如果注解写在 setter 方法上默认取属性名进行装配。当找不到与名称匹配的 bean 时才按照类型进行装配。但是需要注意的是，如果 name 属性一旦指定，就只会按照名称进行装配。

## 30、RPC 的实现基础？

- 需要有非常高效的网络通信，比如一般选择 Netty 作为网络通信框架；
- 需要有比较高效的序列化框架，比如谷歌的 Protobuf 序列化框架；
- 可靠的寻址方式（主要是提供服务的发现），比如可以使用 Zookeeper 来注册服务等等；
- 如果是带会话（状态）的 RPC 调用，还需要有会话和状态保持的功能；

## 31、CMS, G1 垃圾回收器中的三色标记了解吗？

### 三色标记算法思想

三色标记法是一种垃圾回收法，它可以让 JVM 不发生或仅短时间发生 STW(Stop The World)，从而达到清除 JVM 内存垃圾的目的。

三色标记法将对象的颜色分为了黑、灰、白，三种颜色。

**黑色：**该对象已经被标记过了，且该对象下的属性也全部都被标记过了。（程序所需要的对象）；

**灰色：**对象已经被垃圾收集器扫描过了，但是对象中还存在没有扫描的引用（GC 需要从此对象中去寻找垃圾）；

**白色：**表示对象没有被垃圾收集器访问过，即表示不可达。

### CMS 解决办法：增量更新

在应对漏标问题时，CMS 使用了增量更新(Increment Update)方法来做，在一个未被标记的对象（白色对象）被重新引用后，引用它的对象若为黑色则要变成灰色，在下次二次标记时让 GC 线程继续标记它的属性对象（但还是存在漏标的问题）。

CMS 另两个致命缺陷

CMS 采用了 **Mark-Sweep** 算法，最后会产生许多内存碎片，当到一定数量时，CMS 无法清理这些碎片了，CMS 会让 **Serial Old** 垃圾处理器来清理这些垃圾碎片，而 **Serial Old** 垃圾处理器是单线程操作进行清理垃圾的，效率很低。

所以使用 CMS 就会出现一种情况，硬件升级了，却越来越卡顿，其原因就是因为进行 **Serial Old GC** 时，效率过低。

解决方案：使用 **Mark-Sweep-Compact** 算法，减少垃圾碎片

调优参数（配套使用）：

`-XX:+UseCMSCompactAtFullCollection` 开启 CMS 的压缩

`-XX:CMSFullGCsBeforeCompaction` 默认为 0，指经过多少次 CMS FullGC 才进行压缩

当 JVM 认为内存不够，再使用 CMS 进行并发清理内存可能会发生 OOM 的问题，而不得不进行 **Serial Old GC**，**Serial Old** 是单线程垃圾回收，效率低

解决方案：降低触发 **CMS GC** 的阈值，让浮动垃圾不那么容易占满老年代

调优参数：

`-XX:CMSInitiatingOccupancyFraction 92%` 可以降低这个值，让老年代占用率达到该值就进行 CMS GC

### G1 解决办法：SATB

SATB(Snapshot At The Beginning)，在应对漏标问题时，G1 使用了 **SATB** 方法来做，具体流程：

- 在开始标记的时候生成一个快照图标记存活对象

- 
- 在一个引用断开后，要将此引用推到 GC 的堆栈里，保证白色对象（垃圾）还能被 GC 线程扫描到(在\*\*write barrier(写屏障)\*\*里把所有旧的引用所指向的对象都变成非白的)
  - 配合 Rset，去扫描哪些 Region 引用到当前的白色对象，若没有引用到当前对象，则回收

## G1 会不会进行 Full GC？

会，当内存满了的时候就会进行 Full GC；且 JDK10 之前的 Full GC，为单线程的，所以使用 G1 需要避免 Full GC 的产生。

解决方案：

- 加大内存；
- 提高 CPU 性能，加快 GC 回收速度，而对象增加速度赶不上回收速度，则 Full GC 可以避免；
- 降低进行 Mixed GC 触发的阈值，让 Mixed GC 提早发生（默认 45%）

2021 【腾讯】面试真题：

## 1、Kafka 是什么？主要应用场景有哪些？

Kafka 是一个分布式流式处理平台。这到底是什么意思呢？

流平台具有三个关键功能：

- **消息队列**：发布和订阅消息流，这个功能类似于消息队列，这也是 Kafka 也被归类为消息队列的原因。
- **容错的持久方式存储记录消息流**：Kafka 会把消息持久化到磁盘，有效避免了消息丢失的风险。
- **流式处理平台**：在消息发布的时候进行处理，Kafka 提供了一个完整的流式处理类库。

Kafka 主要有两大应用场景：

- **消息队列**：建立实时流数据管道，以可靠地在系统或应用程序之间获取数据。
- **数据处理**：构建实时的流数据处理程序来转换或处理数据流。

## 2、kafka 为什么有 topic 还要用 partition？

- Kafka 可以将主题划分为多个分区（Partition），会根据分区规则选择把消息存储到哪个分区中，只要分区规则设置的合理，那么所有的消息将会被均匀的分布到不同的分区中，这样就实现了负载均衡和水平扩展。另外，多个订阅者可以从一个或者多个分区中同时消费数据，以支撑海量数据处理能力。
- producer 只需要关心消息发往哪个 topic，而 consumer 只关心自己订阅哪个 topic，并不关心每条消息存于整个集群的哪个 broker。为了性能考虑，如果 topic 内的消息只存于一个 broker，那这个 broker 会成为瓶颈，无法做到水平扩展。所以把 topic 内的数据分布到整个集群就是一个自然而然的设计方式。

- 
- Partition 的引入就是解决水平扩展问题的一个方案。

### 3、客户端和服务器之间最多能建立多少个连接？

65535。

服务器的 ip，端口号，客户端的 ip 都是确定的。能变的只有客户端的端口号。  
加网卡，保证四元组唯一，理论上能是客户端和服务器之间建立 10 万以上的连接。

### 4、HashMap 结构，线程不安全举个例子？

- 多个线程同时操作一个 hashmap 就可能出现不安全的情况。
- 如果两个线程同时遇到 HashMap 的大小达到 12 的倍数时，就很有可能会出现在将 oldTable 转移到 newTable 的过程中遇到问题，从而导致最终的 HashMap 的值存储异常。
- 构造 entry<K,V>单链表时，也会出现不安全的情况。

### 5、MySQL 索引分类？

#### 单列索引

- 普通索引：MySQL 中基本索引类型，没有什么限制，允许在定义索引的列中插入重复值和空值，纯粹为了查询数据更快一点。
- 唯一索引：索引列中的值必须是唯一的，但是允许为空值，
- 主键索引：是一种特殊的唯一索引，不允许有空值。

#### 组合索引：

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，使用组合索引时遵循最左前缀集合。

#### 全文索引：

只有在 MyISAM 引擎上才能使用，只能在 CHAR,VARCHAR,TEXT 类型字段上使用全文索引，介绍了要求，说说什么是全文索引，就是在一堆文字中，通过其中的某个关键字等，就能找到该字段所属的记录行，比如有"你是个靓仔，靓女 ..." 通过靓仔，可能就可以找到该条记录

#### 空间索引：

空间索引是对空间数据类型的字段建立的索引，MySQL 中的空间数据类型有四种，GEOMETRY、POINT、LINESTRING、POLYGON。在创建空间索引时，使用 SPATIAL 关键字。要求，引擎为 MyISAM，创建空间索引的列，必须将其声明为 NOT NULL。

### 6、了解线程 & 进程的区别吗？

操作系统中可以拥有多个进程，一个进程里可以拥有多个线程，线程在进程内执行  
进程和线程的区别

- 
- 容易创建新线程。创建新进程需要重复父进程
  - 线程可以控制同一进程的其他线程。进程无法控制兄弟进程，只能控制其子进程
  - 进程拥有自己的内存空间。线程使用进程的内存空间，且要和该进程的其他线程共享这个空间；而不是在进程中给每个线程单独划分一点空间。
  - （同一进程中的）线程在共享内存空间中运行，而进程在不同的内存空间中运行
  - 线程可以使用 `wait()`, `notify()`, `notifyAll()` 等方法直接与其他线程（同一进程）通信；而，进程需要使用“进程间通信”（IPC）来与操作系统中的其他进程通信。

## 7、Java 进程间的几种通信方式？

- 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 (named pipe) : 有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。
- 信号量( semaphore ) : 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。
- 消息队列( message queue ) : 消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。
- 信号 ( signal ) : 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。
- 共享内存( shared memory ) : 共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。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。
- 套接字( socket ) : 套接字也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同及其间的进程通信。

## 8、多台服务器同时对一个数据定时任务，怎么处理？

对于一个定时任务，如果当前任务已经被某一个服务器处理后，另外一个服务器就不需要执行这个任务了

- 在定时任务里加锁机制，等某台服务器获取权限，其他服务器将不再执行此次定时任务。
- 在数据库的创建定时任务控制表 `job_controller`, 创建 `updated_by` 字段，用来存放执行代码的服务器生成的序列号。创建 `updateTime` 字段，用于记录标记更新 `update_by` 的时间戳，也可以理解为上一次任务执行的时间戳。
- 在代码层面，在执行任务的时候，首先生成一个序列号，然后将序列号存储在当前任务的记录上。然后再从数据库里查询当前记录的序列号，在做标记前，首先检查当前任务的上一次执行时间离当前时间超过阈值（自己定义），如果超过则表明还没有其他节点执行该

---

任务，然后为 task 保存标签和当前运行时间。当然如果上一次运行时间为空的情况下，也是允许标记的，如果数据库中的序列号与当前节点生成序列号相匹配，则执行任务的具体逻辑，反之，则什么都不做处理。

## 9、常见分布式锁的几种实现方式？

- 基于数据库实现分布式锁
- 基于缓存实现分布式锁
- 基于 Zookeeper 实现分布式锁

## 10、Redis 分布式锁实现原理？

- set px nx
- 守护线程，进行 renew
- Redis 分布式锁实现：先拿 setnx 来争抢锁，抢到之后，再用 expire(过期)给锁加一个过期时间防止锁忘记了释放。
- 如果在 setnx 之后执行 expire 之前进程意外 crash 或者要重启维护了，那会怎么样：set 指令有非常复杂的参数，这个应该是可以同时把 setnx 和 expire 合成一条指令来用的！

## 11、Redis 的数据类型及它们的使用场景？

string

- key/value；二进制安全的。意思是 redis 的 string 可以包含任何数据。比如 jpg 图片或者序列化的对象。一个键最大能存储 512MB。

hash

- 存储对象数据

list：简单的字符串列表

关注列表

- 队列

set：string 类型的无序集合

共同关注列表

- 统计独立 IP

zset：(sorted set：有序集合)，每个元素都会关联一个 double 类型的分数。redis 正是通过分数来为集合中的成员进行从小到大的排序。

排行

- 带权重的消息队列

## 12、信号量与信号的区别？

- 信号：（signal）是一种处理异步事件的方式。信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程外，还可以发送信号给进程本身。
- 信号量：（Semaphore）进程间通信处理同步互斥的机制。是在多线程环境下使用的一种设施，它负责协调各个线程，以保证它们能够正确、合理的使用公共资源。

简单地说，信号就是一种异步通信，通知进程某种事件的发生；信号量是进程/线程同步与互斥的一种机制，保证进程/线程间之间的有序执行或对公共资源的有序访问。

## 13、select 和 epoll 的底层结构是什么原理

select：支持阻塞操作的设备驱动通常会实现一组自身的等待队列如读/写等待队列用于支持上层(用户层)所需的 BLOCK 或 NONBLOCK 操作。当应用程序通过设备驱动访问该设备时(默认为 BLOCK 操作)，若该设备当前没有数据可读或写，则将该用户进程插入到该设备驱动对应的读/写等待队列让其睡眠一段时间，等到有数据可读/写时再将该进程唤醒。

select 就是巧妙的利用等待队列机制让用户进程适当在没有资源可读/写时睡眠，有资源可读/写时唤醒。

epoll：epoll 由三个系统调用组成，分别是 epoll\_create，epoll\_ctl 和 epoll\_wait。

epoll\_create 用于创建和初始化一些内部使用的数据结构；epoll\_ctl 用于添加，删除或者修改指定的 fd 及其期待的事件，epoll\_wait 就是用于等待任何先前指定的 fd 事件。

## 14、场景题：1亿个数据取出最大前100个有什么方法？

- 最容易想到的方法是将数据全部排序，然后在排序后的集合中进行查找，最快的排序算法的时间复杂度一般为  $O(n \log n)$ ，如快速排序。
- 局部淘汰法，该方法与排序方法类似，用一个容器保存前 10000 个数，然后将剩余的所有数字——与容器内的最小数字相比，如果所有后续的元素都比容器内的 10000 个数还小，那么容器内这个 10000 个数就是最大 10000 个数。如果某一后续元素比容器内最小数字大，则删掉容器内最小元素，并将该元素插入容器，最后遍历完这 1 亿个数，得到的结果容器中保存的数即为最终结果了。此时的时间复杂度为  $O(n+m^2)$ ，其中 m 为容器的大小，即 10000。
- 分治法，将 1 亿个数据分成 100 份，每份 100 万个数据，找到每份数据中最大的 10000 个，最后在剩下的 10010000 个数据里面找出最大的 10000 个。如果 100 万个数据选择足够理想，那么可以过滤掉 1 亿数据里面 99% 的数据。100 万个数据里面查找最大的 10000 个数据的方法如下：用快速排序的方法，将数据分为 2 堆，如果大的那堆个数 N 大于 10000 个，继续对大堆快速排序一次分成 2 堆，如果大的那堆个数 N 大于 10000 个，继续对大堆快速排序一次分成 2 堆，如果大堆个数 N 小于 10000 个，就在小的那堆里面快速排序一次，找第  $10000-n$  大的数字；递归以上过程，就可以找到第  $1w$  大的数。参考上面的找出第  $1w$  大数字，就可以类似的方法找到前 10000 大数字了。此种方法需要每次的内存空间为  $10^{64}=4MB$ ，一共需要 101 次这样的比较。

- 
- Hash 法，如果这 1 亿个数里面有很多重复的数，先通过 Hash 法，把这 1 亿个数字去重复，这样如果重复率很高的话，会减少很大的内存用量，从而缩小运算空间，然后通过分治法或最小堆法查找最大的 10000 个数。
  - 采用最小堆法，首先读入前 10000 个数来创建大小为 10000 的最小堆，建堆的时间复杂度为  $O(m \log m)$  ( $m$  为数组的大小即为 10000)，然后遍历后续的数字，并于堆顶（最小）数字进行比较。如果比最小的数小，则继续读取后续数字；如果比堆顶数字大，则替换堆顶元素并重新调整堆为最小堆。整个过程直至 1 亿个数全部遍历完为止。然后按照中序遍历的方式输出当前堆中的所有 10000 个数字。该算法的时间复杂度为  $O(nm \log m)$ ，空间复杂度是 10000（常数）。

## 15、kafka 如何保证消息可靠？

### 生产者丢失消息的情况

生产者(Producer) 调用 `send` 方法发送消息之后，消息可能因为网络问题并没有发送过去。为了确定消息是发送成功，我们要判断消息发送的结果，Kafka 生产者(Producer) 使用 `send` 方法发送消息实际上是异步的操作，我们可以通过 `get()`方法获取调用结果，但是这样也让它变为了同步操作，可以采用为其添加回调函数的形式，示例代码如下：

```
ListenableFuture<SendResult<String, Object>> future = kafkaTemplate.send(topic, o);
future.addCallback(result -> logger.info("生产者成功发送消息到 topic:{} partition:{} 的消息",
result.getRecordMetadata().topic(), result.getRecordMetadata().partition()),
ex -> logger.error("生产者发送消失败，原因： {}", ex.getMessage()));
```

Producer 的 `retries`（重试次数）设置一个比较合理的值，一般是 3，但是为了保证消息不丢失的话一般会设置比较大一点。设置完成之后，当出现网络问题之后能够自动重试消息发送，避免消息丢失。另外，建议还要设置重试间隔，因为间隔太小的话重试的效果就不明显了，网络波动一次你 3 次一下子就重试完了

### 消费者丢失消息的情况

当消费者拉取到了分区的某个消息之后，消费者会自动提交了 offset。自动提交的话会有一个问题，试想一下，当消费者刚拿到这个消息准备进行真正消费的时候，突然挂掉了，消息实际上并没有被消费，但是 offset 却被自动提交了。

解决办法也比较粗暴，我们手动关闭自动提交 offset，每次在真正消费完消息之后再自己手动提交 offset。但是，细心的朋友一定会发现，这样会带来消息被重新消费的问题。比如你刚刚消费完消息之后，还没提交 offset，结果自己挂掉了，那么这个消息理论上就会被消费两次。

### Kafka 弄丢了消息

试想一种情况：假如 leader 副本所在的 broker 突然挂掉，那么就要从 follower 副本重新选出一个 leader，但是 leader 的数据还有一些没有被 follower 副本的同步的话，就会造成消息丢失。

---

当我们配置了 `unclean.leader.election.enable = false` 的话，当 leader 副本发生故障时就不会从 follower 副本中和 leader 同步程度达不到要求的副本中选择出 leader，这样降低了消息丢失的可能性。

## 16、消息队列的使用场景？

消息队列在实际应用中包括如下四个场景：

- 应用耦合：多应用间通过消息队列对同一消息进行处理，避免调用接口失败导致整个过程失败；
- 异步处理：多应用对消息队列中同一消息进行处理，应用间并发处理消息，相比串行处理，减少处理时间；
- 限流削峰：广泛应用于秒杀或抢购活动中，避免流量过大导致应用系统挂掉的情况；
- 消息驱动的系统：系统分为消息队列、消息生产者、消息消费者，生产者负责产生消息，消费者(可能有多个)负责对消息进行处理；

## 17、乐观锁和悲观锁的理解及如何实现，有哪些实现方式？

**悲观锁：**

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。再比如 Java 里面的同步原语 `synchronized` 关键字的实现也是悲观锁。

**乐观锁：**

顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于 `write_condition` 机制，其实都是提供的乐观锁。在 Java 中 `java.util.concurrent.atomic` 包下面的原子变量类就是使用了乐观锁的一种实现方式 CAS 实现的。

## 18、ArrayList 和 LinkedList 的区别在哪里？

- 数据结构实现：ArrayList：基于数组，便于按 index 访问，超过数组需要扩容，扩容成本较高。LinkedList：使用链表实现，无需扩容。
- 随机访问效率：ArrayList 比 LinkedList 在随机访问的时候效率要高，因为 LinkedList 是线性的数据存储方式，所以需要移动指针从前往后依次查找。
- 增加和删除效率：在非首尾的增删操作，LinkedList 要比 ArrayList 效率要高，因为 ArrayList 增删操作要影响数组内的其他数据的下标。
- 内存空间占用：LinkedList 比 ArrayList 更占内存，因为 LinkedList 的节点除了存储数据，还存储了两个引用，一个指向下一个元素，一个指向后一个元素。

- 
- 线程安全：ArrayList 和 LinkList 都是不同步的，不保证线程安全。
  - 综合来说，需要频繁读取集合中的元素时，更推荐使用 ArrayList，而在增删操作较多时，更推荐使用 LinkedList。
  - LinkedList 的双向链表是链表的一种，它的每个数据结点中都有 2 个指针，分别指向直接后继和直接前驱。所以，从双向链表中的任意一个结点开始，都可以很方便的访问它的前驱结点和后继结点。

## 19、谈谈你对 SQL 注入式攻击的理解？

所谓 SQL 注入式攻击，就是攻击者把 SQL 命令插入到 Web 表单的输入域或页面请求的查询字符串，欺骗服务器执行恶意的 SQL 命令。

### 如何防范 SQL 注入式攻击？

在利用表单输入的内容构造 SQL 命令之前，把所有输入内容过滤一番就可以了。过滤输入内容可以按多种方式进行。

- 对于动态构造 SQL 查询的场合
  - a. 替换单引号，即把所有单独出现的单引号改成两个单引号，防止攻击者修改 SQL 命令的含义。
  - b. 删除用户输入内容中的所有连字符
  - c. 对于用来执行查询的数据库帐户，限制其权限。用不同的用户帐户执行查询、插入、更新、删除操作。
- 用存储过程来执行所有的查询。
- 限制表单或查询字符串输入的长度。
- 检查用户输入的合法性。
- 将用户登录名称、密码等数据加密保存。
- 检查提取数据的查询所返回的记录数量。

## 20、数据库事务的特性？

- 原子性：即不可分割性，事务要么全部被执行，要么就全部不被执行。
- 一致性或可串性。事务的执行使得数据库从一种正确状态转换成另一种正确状态
- 隔离性。在事务正确提交之前，不允许把该事务对数据的任何改变提供给任何其他事务，
- 持久性。事务正确提交后，其结果将永久保存在数据库中，即使在事务提交后有了其他故障，事务的处理结果也会得到保存。

## 21、Redis 如何做内存优化？

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。

## 22、缓存穿透，缓存击穿，缓存雪崩都是咋回事？解决办法？

### 缓存穿透

- 问题：大量并发查询不存在的 KEY，在缓存和数据库中都不存在，同时给缓存和数据库带来压力。
- 原因：一般而言，缓存穿透有 2 种可能性：业务数据被误删，导致缓存和数据库中都没有数据。恶意进行 ddos 攻击。
- 分析：为什么会多次透传呢？不存在一直为空，需要注意让缓存能够区分 KEY 不存在和查询到一个空值。
- 解决办法：缓存空值的 KEY，这样第一次不存在也会被加载会记录，下次拿到有这个 KEY。Bloom 过滤或 RoaringBitmap 判断 KEY 是否存在，如果布隆过滤器中没有查到这个数据，就不去数据库中查。在处理请求前增加恶意请求检查，如果检测到是恶意攻击，则拒绝进行服务。完全以缓存为准，使用延迟异步加载的策略（异步线程负责维护缓存的数据，定期或根据条件触发更新），这样就不会触发更新。

### 缓存击穿

- 问题：某个 KEY 失效的时候，正好有大量并发请求访问这个 KEY。
- 分析：跟穿透其实很像，属于比较偶然的。
- 解决办法：KEY 的更新操作添加全局互斥锁。完全以缓存为准，使用延迟异步加载的策略（异步线程负责维护缓存的数据，定期或根据条件触发更新），这样就不会触发更新。

### 缓存雪崩

- 问题：当某一时刻发生大规模的缓存失效的情况，导致大量的请求无法获取数据，从而将流量压力传导到数据库上，导致数据库压力过大甚至宕机。
- 原因：一般而言，缓存雪崩有 2 种可能性：大量的数据同一个时间失效：比如业务关系强相关的数据要求同时失效 Redis 宕机
- 分析：一般来说，由于更新策略、或者数据热点、缓存服务宕机等原因，可能会导致缓存数据同一个时间点大规模不可用，或者都更新。所以，需要我们的更新策略要在时间上合适，数据要均匀分享，缓存服务器要多台高可用。
- 解决办法：更新策略在时间上做到比较平均。如果数据需要同一时间失效，可以给这批数据加上一些随机值，使得这批数据不要在同一个时间过期，降低数据库的压力。使用的热数据尽量分散到不同的机器上。多台机器做主从复制或者多副本，实现高可用。做好主从的部署，当主节点挂掉后，能快速的使用从结点顶上。实现熔断限流机制，对系统进行负载能力控制。对于非核心功能的业务，拒绝其请求，只允许核心功能业务访问数据库获取数据。服务降价：提供默认返回值，或简单的提示信息。

## 23、数组和链表的区别？当数组内存过大时会出现什么问题？链表增删过多会出现的什么问题？

- 数组静态分配内存，链表动态分配内存；

- 
- 数组事先定义固定的长度，不能适应数据动态的增减的情况。当数据增加时，可能超出原先定义的元素个数；当数据减少时，造成内存浪费；
  - 链表动态地进行存储分配，可以适应数据动态地增减的情况
  - 数组在内存中连续，链表不连续；
  - 数组元素在栈区，链表元素在堆区；
  - （静态）数组从栈中分配空间，对于程序员方便快速，但是自由度小；
  - 链表从堆中分配空间，自由度大但是申请管理比较麻烦。
  - 数组利用下标定位，时间复杂度为  $O(1)$ ，链表定位元素时间复杂度  $O(n)$ ；
  - 数组插入或删除元素的时间复杂度  $O(n)$ ，链表的时间复杂度  $O(1)$ 。
  - 当数组内存过大时会出现什么问题（堆内存溢出），链表增删过多会出现的什么问题（大量内存碎片）

## 24、常见排序算法和分别的复杂度？

- 冒泡排序， $O(n^2)$ ，通过重复走完数组的所有元素，通过两两比较，直到没有数可以交换的时候结束这个数，再到下个数，直到整个数组排好顺序。
- 插入排序， $O(n^2)$ ，每次从未排好序的数据堆中拿出一个数，插入到已排好序的数据队列的正确位置。
- 选择排序， $O(n^2)$ ，每次从未排好序的数据堆中找到最小的数，插入到已排好序的数据队列的头部。
- 快速排序， $O(N \log N)$ ，以数据堆中的一个数为标准，将数据堆分为小于等于和大于该数的两堆，对于分割后的两堆数再分别利用上述方法进行分割，以此类推，直到堆中只有一个数为止。
- 堆排序， $O(N \log N)$ ，将数据堆中的数两两组队排序，对于排序好的这些子堆再两两组队排序，以此类推，直到只剩下一个堆。
- 归并排序， $O(N \log N)$ ，基于堆的排序算法，分为最大堆和最小堆。排序分为两个过程堆的构造和堆的排序。

## 25、jdk 1.8 的 JVM 内存划分模型，堆和栈的区别

- 方法区(method)：被所有的线程共享。方法区包含所有的类信息和静态变量。（运行时常量池）
- 堆(heap)：被所有的线程共享，存放对象实例以及数组，Java 堆是 GC 的主要区域。
- 栈(stack)：每个线程包含一个栈区，栈中保存一些局部变量等。（本地局部变量、操作数栈、动态链接、返回地址）
- 程序计数器：是当前线程执行的字节码的行指示器。
- 本地方法栈

## 26、简单描述 MySQL 中，索引，主键，唯一索引，联合索引的区别，对数据库的性能有什么影响（从读写两方面）？

- 索引是一种特殊的文件(InnoDB 数据表上的索引是表空间的一个组成部分)，它们包含着对数据表里所有记录的引用指针。
- 普通索引(由关键字 KEY 或 INDEX 定义的索引)的唯一任务是加快对数据的访问速度。
- 普通索引允许被索引的数据列包含重复的值。如果能确定某个数据列将只包含彼此各不相同的值，在为这个数据列创建索引的时候就应该用关键字 UNIQUE 把它定义为一个唯一索引。也就是说，唯一索引可以保证数据记录的唯一性。
- 主键，是一种特殊的唯一索引，在一张表中只能定义一个主键索引，主键用于唯一标识一条记录，使用关键字 PRIMARY KEY 来创建。
- 索引可以覆盖多个数据列，如像 INDEX(columnA, columnB)索引，这就是联合索引。
- 索引可以极大的提高数据的查询速度，但是会降低插入、删除、更新表的速度，因为在执行这些写操作时，还要操作索引文件。

## 27、I/O 模型有哪几种？

阻塞 I/O， 非阻塞 I/O 模型， I/O 复用模型，信号驱动 I/O 模型， 异步 I/O 模型。

## 28、当你用浏览器打开一个链接的时候，计算机做了哪些工作步骤？

域名解析 -> 发起 TCP 的 3 次握手 -> 建立 TCP 连接后发起 http 请求 -> 服务器响应 http 请求 -> 浏览器得到 html 代码 -> 浏览器解析 html 代码，并请求 html 代码中的资源（如 js、css、图片等） -> 浏览器对页面进行渲染呈现给用户。

## 29、虚拟 DOM 的优劣如何？

### 优点:

- 保证性能下限: 虚拟 DOM 可以经过 diff 找出最小差异,然后批量进行 patch,这种操作虽然比不上手动优化,但是比起粗暴的 DOM 操作性能要好很多,因此虚拟 DOM 可以保证性能下限
- 无需手动操作 DOM: 虚拟 DOM 的 diff 和 patch 都是在一次更新中自动进行的,我们无需手动操作 DOM,极大提高开发效率
- 跨平台: 虚拟 DOM 本质上是 JavaScript 对象,而 DOM 与平台强相关,相比之下虚拟 DOM 可以进行更方便地跨平台操作,例如服务器渲染、移动端开发等等

### 缺点:

无法进行极致优化: 在一些性能要求极高的应用中虚拟 DOM 无法进行针对性的极致优化,比如 VScode 采用直接手动操作 DOM 的方式进行极端的性能优化。

## 30、幻读是什么，用什么隔离级别可以防止幻读？

幻读是一个事务在前后两次查询同一个范围的时候、后一次查询看到了前一次查询未看到的行。在可重复读隔离级别下，普通的查询是快照读，是不会看到别的事务插入的数据的。因此，幻读在“当前读”下才会出现。

SERIALIZABLE(可串行化)可以防止幻读：最高的隔离级别，完全服从 ACID 的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰。

2021 【百度】面试真题：

### 1、SpringBoot 也有定时任务？是什么注解？

在 SpringBoot 中使用定时任务主要有两种不同的方式，一个就是使用 Spring 中的 [@Scheduled](#) 注解，另一个则是使用第三方框架 Quartz。

使用 Spring 中的 [@Scheduled](#) 的方式主要通过 [@Scheduled](#) 注解来实现。

使用 Quartz，则按照 Quartz 的方式，定义 Job 和 Trigger 即可。

### 2、请描述线程的生命周期，它们之间如何切换？

线程的生命周期包含 5 个阶段，包括：新建、就绪、运行、阻塞、销毁。

- 新建（NEW）：就是刚使用 new 方法，new 出来的线程；
- 就绪（RUNNABLE）：就是调用的线程的 start()方法后，这时候线程处于等待 CPU 分配资源阶段，谁先抢的 CPU 资源，谁开始执行；
- 运行（RUNNING）：当就绪的线程被调度并获得 CPU 资源时，便进入运行状态，run 方法定义了线程的操作和功能；
- 阻塞（BLOCKED）：在运行状态的时候，可能因为某些原因导致运行状态的线程变成了阻塞状态，比如 sleep()、wait()之后线程就处于了阻塞状态，这个时候需要其他机制将处于阻塞状态的线程唤醒，比如调用 notify 或者 notifyAll()方法。唤醒的线程不会立刻执行 run 方法，它们要再次等待 CPU 分配资源进入运行状态；
- Waiting（无限等待）：一个线程在等待另一个线程执行一个（唤醒）动作时，该线程进入 Waiting 状态。进入这个状态后不能自动唤醒，必须等待另一个线程调用 notify 方法或者 notifyAll 方法时才能够被唤醒。
- 销毁（TERMINATED）：如果线程正常执行完毕后或线程被提前强制性的终止或出现异常导致结束，那么线程就要被销毁，释放资源；

### 3、什么情况线程会进入 WAITING 状态？

---

一个线程在等待另一个线程执行一个（唤醒）动作时，该线程进入 Waiting 状态。进入这个状态后不能自动唤醒，必须等待另一个线程调用 notify 方法或者 notifyAll 方法时才能够被唤醒。

- 调用 Object 对象的 wait 方法，但没有指定超时值。
- 调用 Thread 对象的 join 方法，但没有指定超时值。
- 调用 LockSupport 对象的 park 方法。

## 4、简述多进程开发中 join 和 deamon 的区别？

join：当子线程调用 join 时，主线程会被阻塞，当子线程结束后，主线程才能继续执行。

deamon：当子进程被设置为守护进程时，主进程结束，不管子进程是否执行完毕，都会随着主进程的结束而结束。

## 5、异步和同步、阻塞和非阻塞之间的区别？

### 同步

当一个 request 发送出去以后，会得到一个 response，这整个过程就是一个同步调用的过程。哪怕 response 为空，或者 response 的返回特别快，但是针对这一次请求而言就是一个同步的调用。

### 异步

当一个 request 发送出去以后，没有得到想要的 response，而是通过后面的 callback、状态或者通知的方式获得结果。可以这么理解，对于异步请求分两步：

- 调用方发送 request 没有返回对应的 response（可能是一个空的 response）；
- 服务提供方将 response 处理完成以后通过 callback 的方式通知调用方。

对于 1) 而言是同步操作（调用方请求服务方），对于 2) 而言也是同步操作（服务方回掉调用方）。从请求的目的（调用方发送一个 request，希望获得对应的 response）来看，这两个步骤拆分开来没有任何意义，需要结合起来看，而这整个过程就是一次异步请求。异步请求有一个最典型的特点：需要 callback、状态或者通知的方式来告知调用方结果。

### 阻塞

阻塞调用是指调用方发出 request 的线程因为某种原因（如：等待系统资源）被服务方挂起，当服务方得到 response 后就唤醒挂起线程，并将 response 返回给调用方。

### 非阻塞

非阻塞调用是指调用方发出 request 的线程在没有等到结果时不会被挂起，并且直到得到 response 后才返回。

阻塞和非阻塞最大的区别就是看调用方线程是否会被挂起。

## 6、为什么要分内核态和用户态？

---

假设没有这种内核态和用户态之分，程序随随便便就能访问硬件资源，比如说分配内存，程序能随意的读写所有的内存空间，如果程序员一不小心将不适当的内容写到了不该写的地方，就很可能导致系统崩溃。用户程序是不可信的，不管程序员是有意的还是无意的，都很容易将系统干到崩溃。

正因为如此，Intel 就发明了 ring0-ring3 这些访问控制级别来保护硬件资源，ring0 的就是我们所说的内核级别，要想使用硬件资源就必须获取相应的权限（设置 PSW 寄存器，这个操作只能由操作系统设置）。操作系统对内核级别的指令进行封装，统一管理硬件资源，然后向用户程序提供系统服务，用户程序进行系统调用后，操作系统执行一系列的检查验证，确保这次调用是安全的，再进行相应的资源访问操作。\*\*内核态能有效保护硬件资源的安全。

## 7、说下类加载器与类加载？加载的类信息放在哪个区域？

一个类型从被加载到虚拟机内存开始，到卸载出内存为止，它的整个生命周期将会经历加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（Using）和卸载（Unloading）七个阶段。其中验证、准备、解析三个部分统称为连接（Linking）。

Java 虚拟机设计团队把类加载阶段中“通过一个类的全限定名来获取描述该类的二进制流”这个动作放到 Java 虚拟机外部去实现。以便让程序应用自己决定如何取获取所需的类。实现这个动作的代码被称为“类加载器”（Class Loader）。

对于任意一个类，都必须由加载它的类加载器和这个类本身一起共同确立其在 Java 虚拟机中的唯一性，每一个类加载器，都拥有一个独立的类名称空间。

## 8、UDP 协议和 TCP 协议的区别？

- TCP 基于连接，UDP 基于无连接
- TCP 要求系统资源较多，UDP 较少
- UDP 程序结构较简单
- TCP 保证数据正确性，UDP 可能丢包
- TCP 保证数据顺序，UDP 不保证

## 9、limit 1000000 加载很慢的话，你是怎么解决的呢？

**方案一：如果 id 是连续的，可以这样，返回上次查询的最大记录(偏移量)，再往下 limit**  
select id, name from employee where id>1000000 limit 10.

**方案二：在业务允许的情况下限制页数：**

建议跟业务讨论，有没有必要查这么后的分页啦。因为绝大多数用户都不会往后翻太多页。

**方案三：order by + 索引（id 为索引）**

select id, name from employee order by id limit 1000000, 10

---

**方案四：利用延迟关联或者子查询优化超多分页场景。（先快速定位需要获取的 id 段，然后再关联）**

```
SELECT a.* FROM employee a, (select id from employee where 条件 LIMIT 1000000,10 ) b  
where a.id=b.id
```

## 10、MySQL 的索引分类是什么？

### 单列索引

- 普通索引：MySQL 中基本索引类型，没有什么限制，允许在定义索引的列中插入重复值和空值，纯粹为了查询数据更快一点。
- 唯一索引：索引列中的值必须是唯一的，但是允许为空值，
- 主键索引：是一种特殊的唯一索引，不允许有空值。

### 组合索引：

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，使用组合索引时遵循最左前缀集合。

### 全文索引：

只有在 MyISAM 引擎上才能使用，只能在 CHAR,VARCHAR,TEXT 类型字段上使用全文索引，介绍了要求，说说什么是全文索引，就是在一堆文字中，通过其中的某个关键字等，就能找到该字段所属的记录行，比如有"你是个靓仔，靓女 ..." 通过靓仔，可能就可以找到该条记录

### 空间索引：

空间索引是对空间数据类型的字段建立的索引，MySQL 中的空间数据类型有四种，GEOMETRY、POINT、LINESTRING、POLYGON。在创建空间索引时，使用 SPATIAL 关键字。要求，引擎为 MyISAM，创建空间索引的列，必须将其声明为 NOT NULL。

## 11、什么是散列表？ select \* 和 select 1 ?

哈希表（Hash table，也叫散列表），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。

有时候为了提高效率，只是为了测试下某个表中是否存在记录，就用 1 来代替。

## 12、MySQL 的主从复制了解吗？

主库将变更写入 binlog 日志，然后从库连接到主库之后，从库有一个 IO 线程，将主库的 binlog 日志拷贝到自己本地，写入一个 relay 中继日志中接着从库中有一个 SQL 线程会从中继日志读取 binlog，然后执行 binlog 日志中的内容，也就是在自己本地再次执行一遍 SQL。

## 13、Spring 框架事务注解用什么注解？使用该注解的失效场景？

---

## @Transactional

- Transactional 注解应用在非 public 修饰的方法上 @Transactional 注解属性 propagation 设置错误
- @Transactional 注解属性 rollbackFor 设置错误
- 同一个类中方法调用，导致 @Transactional 失效
- 异常被 catch “吃了” 导致 @Transactional 失效

## 14、final、finally、finalize? finally 是在 return 之前执行还是之后？finally 块里的代码一定会执行吗？

- final 可以用来修饰类、方法、变量，分别有不同的意义，final 修饰的 class 代表不可以继承扩展，final 的变量是不可以修改的，而 final 的方法也是不可以重写的（override）。
- finally 是 Java 保证重点代码一定要被执行的一种机制。可以使用 try-finally 或者 try-catch-finally 来进行类似关闭 JDBC 连接、保证 unlock 锁等动作。
- finalize 是基础类 java.lang.Object 的一个方法，设计目的是保证对象在被垃圾收集前完成特定资源的回收。finalize 机制现在已经不推荐使用，并且在 JDK 9 开始被标记为 deprecated。

finally 块的语句在 try 或 catch 中的 return 语句执行之后返回之前执行且 finally 里的修改语句可能影响也可能不影响 try 或 catch 中 return 已经确定的返回值，若 finally 里也有 return 语句则覆盖 try 或 catch 中的 return 语句直接返回。

finally 块里的代码不一定会执行。比如：

- try 语句没有被执行到，如在 try 语句之前就返回了，这样 finally 语句就不会执行，这也说明了 finally 语句被执行的必要而非充分条件是：相应的 try 语句一定被执行到。
- 在 try 块中有 System.exit(0)\*\*

## 15、I/O 多路复用实现方式有哪些？

- select
- poll
- epoll

## 16、select、poll、epoll 区别有哪些？

select：它仅仅知道了，有 I/O 事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。所以 select 具有  $O(n)$  的无差别轮询复杂度，同时处理的流越多，无差别轮询时间就越长。

poll：poll 本质上和 select 没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个 fd 对应的设备状态，但是它没有最大连接数的限制，原因是它是基于链表来存储的。

---

epoll: epoll 可以理解为 event poll, 不同于忙轮询和无差别轮询, epoll 会把哪个流发生了怎样的 I/O 事件通知我们。所以我们说 epoll 实际上是事件驱动（每个事件关联上 fd）的，此时我们对这些流的操作都是有意义的。（复杂度降低到了 O(1)），通过红黑树和双链表数据结构，并结合回调机制，造就了 epoll 的高效，epoll\_create(), epoll\_ctl() 和 epoll\_wait() 系统调用。

## 17、哈希算法解决哈希冲突方式有哪些？

解决哈希冲突的方法一般有：开放寻址法、链地址法（拉链法）、再哈希法、建立公共溢出区等方法。

## 18、如何保证 Redis 中的数据不丢失？

### 单机单节点模式

#### 使用 AOF 和 RDB 结合的方式

RDB 做镜像全量持久化，AOF 做增量持久化。因为 RDB 会耗费较长时间，不够实时，在停机的时候会导致大量丢失数据，所以需要 AOF 来配合使用。

### Redis 集群模式

#### master 节点持久化

如果采用了主从架构，那么建议必须开启 master node 的持久化！不建议用 slave node 作为 master node 的数据热备，因为那样的话，如果你关掉 master 的持久化，可能在 master 宕机重启的时候数据是空的，然后可能一经过复制，slave node 数据也丢了，master 就会将空的数据集同步到 slave 上去，所有 slave 的数据全部清空。

#### Redis 断点续传

从 redis 2.8 开始，就支持主从复制的断点续传，如果主从复制过程中，网络连接断掉了，那么可以接着上次复制的地方，继续复制下去，而不是从头开始复制一份。

#### 主备切换的过程，可能会导致数据丢失

解决异步复制和脑裂导致的数据丢失

redis.conf 中

min-slaves-to-write 1

min-slaves-max-lag 10

要求至少有 1 个 slave，数据复制和同步的延迟不能超过 10 秒

如果说一旦所有的 slave，数据复制和同步的延迟都超过了 10 秒钟，那么这个时候，master 就不会再接收任何请求了

上面两个配置可以减少异步复制和脑裂导致的数据丢失。

## 19、如何保证 Redis 中的数据都是热点数据？

- 
- Redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。Redis 提供 6 种数据淘汰策略：
  - volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰
  - volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）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（server.db[i].expires）中任意选择数据淘汰
  - allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰
  - allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰
  - no-eviction（驱逐）：禁止驱逐数据

## 20、Redis 持久化机制是如何做的？

### RDB

RDB 持久化方式，是将 Redis 某一时刻的数据持久化到磁盘中，是一种快照式的持久化方法。

#### RDB 优点：

- RDB 是一个非常紧凑（有压缩）的文件,它保存了某个时间点的数据,非常适用于数据的备份。
- RDB 作为一个非常紧凑（有压缩）的文件，可以很方便传送到另一个远端数据中心，非常适用于灾难恢复.
- RDB 在保存 RDB 文件时父进程唯一需要做的就是 fork 出一个子进程,接下来的工作全部由子进程来做，父进程不需要再做其他 IO 操作，所以 RDB 持久化方式可以最大化 redis 的性能.
- 与 AOF 相比,在恢复大的数据集的时候，RDB 方式会更快一些.

#### RDB 缺点：

- Redis 意外宕机 时，会丢失部分数据
- 当 Redis 数据量比较大时，fork 的过程是非常耗时的，fork 子进程时是会阻塞的，在这期间 Redis 是不能响应客户端的请求的。

### AOF

AOF 方式是将执行过的写指令记录下来，在数据恢复时按照从前到后的顺序再将指令都执行一遍。

#### AOF 优点：

- 使用 AOF 会让你的 Redis 更加持久化。
- AOF 文件是一个只进行追加的日志文件，不需要在写入时读取文件。
- Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写。
- AOF 文件可读性高，分析容易。

---

### AOF 缺点：

- 对于相同的数据来说，AOF 文件大小通常要大于 RDB 文件
- 根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB

### 混合持久化方式

Redis 4.0 之后新增的方式，混合持久化是结合了 RDB 和 AOF 的优点，在写入的时候，先把

当前的数据以 RDB 的形式写入文件的开头，再将后续的操作命令以 AOF 的格式存入文件，这

样既能保证 Redis 重启时的速度，又能减低数据丢失的风险。

## 21、Redis 为什么在使用 RDB 进行快照时会通过子进程的方式进行实现？

- 通过 `fork` 创建的子进程能够获得和父进程完全相同的内存空间，父进程对内存的修改对于子进程是不可见的，两者不会相互影响；
- 通过 `fork` 创建子进程时不会立刻触发大量内存的拷贝，内存被修改时会以页为单位进行拷贝，这也就避免了大量拷贝内存而带来的性能问题；

## 22、介绍下 MySQL 的主从复制原理？产生主从延迟的原因？

- 主从复制原理：主库将变更写入 binlog 日志，然后从库连接到主库之后，从库有一个 IO 线程，将主库的 binlog 日志拷贝到自己本地，写入一个 relay 中继日志中。接着从库中有一个 SQL 线程会从中继日志读取 binlog，然后执行 binlog 日志中的内容，也就是在自己本地再次执行一遍 SQL。
- 主从延迟：
  - a. 主库的从库太多
  - b. 从库硬件配置比主库差
  - c. 慢 SQL 语句过多
  - d. 主从库之间的网络延迟
  - e. 主库读写压力大

## 23、父进程如果宕掉，子进程会怎样？

如果父进程是会话首进程，那么父进程退出后，子进程也会退出；反之如果父进程不是会话首进程，那么父进程退出后，子进程不会退出，而它的一个或多个子进程还在运行，那么这些子进程就成为孤儿进程。

## 24、孤儿进程和僵尸进程有什么区别？

孤儿进程：父进程结束了，而它的一个或多个子进程还在运行，那么这些子进程就成为孤儿进程(father died)。子进程的资源由 init 进程(进程号 PID = 1)回收。

---

僵尸进程：子进程退出了，但是父进程没有用 wait 或 waitpid 去获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中，这种进程称为僵死进程。

## 25、MySQL 中有哪几种锁？

- 表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低。
- 行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。
- 页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。

## 26、互斥锁（mutex）和自旋锁（spinlock）分别在什么场景使用？

在多核机器中，如果锁住的“事务”很简单，占用很少的时间，就应该使用 spinlock，这个时候 spinlock 的代价比 mutex 会小很多。“事务”很快执行完毕，自旋的消耗远远小于陷入 sleep 和 wake 的消耗。如果锁住“事务”粒度较大，就应该使用 mutex，因为如果用 spinlock，那么在“事务”执行过程中自旋很长时间还不如使得线程 sleep。

在单核机器中。spinlock 没有任何意义的，spinlock 只会浪费唯一核心的 cpu 时间片，这个时刻没有任何线程会运行的。所以单核机器中，不论锁住的“事务”的粒度大小都要使用。

## 27、描述 Synchronized、ReentrantLock 的区别？

- synchronized 是关键字，ReentrantLock 是 API 接口
- Lock 需要手动加锁，手动释放锁
- synchronized 不可中断，ReentrantLock 可中断、可超时
- synchronized 是非公平锁，ReentrantLock 公平、非公平皆可
- ReentrantLock 支持 Condition，多条件

## 28、HashMap 扩容操作是怎么实现的？

- 在 jdk1.8 中，resize 方法是在 hashmap 中的键值对大于阀值时或者初始化时，就调用 resize 方法进行扩容；
- 每次扩展的时候，都是扩展 2 倍；
- 扩展后 Node 对象的位置要么在原位置，要么移动到原偏移量两倍的位置。

## 29、ConcurrentHashMap 1.7 与 1.8 区别？

- 1.8 采用 synchronized 代替可重入锁 ReentrantLock（现代 JDK 中，synchronized 已经被不断优化，可以不再过分担心性能差异）

- 
- 1.8 取消了 Segment 分段锁的数据结构，使用数组+链表+红黑树的结构代替
  - 1.8 对每个数组元素加锁，1.7 对要操作的 Segment 数据段加锁

## 30、如何使用 Java 的反射？

- 通过一个全限类名创建一个对象

Class.forName(“全限类名”); 例如： com.mysql.jdbc.Driver Driver 类已经被加载到 jvm 中，并且完成了类的初始化工作就行了

- 类名.class; 获取 Class<? > clz 对象

对象.getClass();

- 获取构造器对象，通过构造器 new 出一个对象

Clazz.getConstructor([String.class]);

Con.newInstance([参数]);

- 通过 class 对象创建一个实例对象（就相当与 new 类名() 无参构造器）

Clz.newInstance();

- 通过 class 对象获得一个属性对象

Field c=cls.getFields(): 获得某个类的所有的公共（public）的字段，包括父类中的字段。

Field c=cls.getDeclaredFields(): 获得某个类的所有声明的字段，即包括 public、private 和 proteced，但是不包括父类的声明字段

- 通过 class 对象获得一个方法对象

Clz.getMethod(“方法名”,class……parameaType); (只能获取公共的)

Clz.getDeclareMethod(“方法名”); (获取任意修饰的方法，不能执行私有)

M.setAccessible(true); (让私有的方法可以执行)

- 让方法执行

Method.invoke(obj 实例对象,obj 可变参数);----- (是有返回值的)

## 2021 【华为】面试真题：

### 1、Java 常用集合及特点？

List: ArrayList、LinkedList、Vector、Stack Set: LinkedSet、HashSet、TreeSet  
Queue->Deque->LinkedList。

Map: HashMap、LinkedHashMap、TreeMap Dictionary->HashTable->Properties。

Vector: 底层数据结构是数组，查询快，增删慢，线程安全，效率低，默认长度为 10，超过会 100% 延长，变成 20，浪费空间。

ArrayList : 基于数组，便于按 index 访问，超过数组需要扩容，扩容成本较高。

LinkedList: 使用链表实现，无需扩容。

HashSet：底层数据结构是哈希表（无序，唯一），通过 hashCode() 和 equals() 保证元素唯一。

LinkedHashSet：底层数据结构是链表和哈希表（FIFO 插入有序，唯一），由链表保证元素有序，由哈希表保证元素唯一。

TreeSet：底层数据结构是红黑树（唯一，有序），通过自然排序和比较器排序保证元素有序，根据比较返回值是否是 0 来保证元素唯一性。

TreeMap 是有序的。

HashMap：空间换时间，哈希冲突不大的情况下查找数据性能很高。

LinkedHashMap 基本特点：继承自 HashMap，对 Entry 集合添加了一个双向链表。

## 2、开启一个线程的方法？

- 继承 Thread 类，新建一个当前类对象，并且运行其 start() 方法
- 实现 Runnable 接口，然后新建当前类对象，接着新建 Thread 对象时把当前类对象传进去，最后运行 Thread 对象的 start() 方法
- 实现 Callable 接口，新建当前类对象，在新建 FutureTask 类对象时传入当前类对象，接着新建 Thread 类对象时传入 FutureTask 类对象，最后运行 Thread 对象的 start() 方法

## 3、Java 面向对象包括哪些特性，怎么理解的？

- 封装：通常认为封装是把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。面向对象的本质就是将现实世界描绘成一系列完全自治、封闭的对象。我们在类中编写的方法就是对实现细节的一种封装；我们编写一个类就是对数据和数据操作的封装。可以说，封装就是隐藏一切可隐藏的东西，只向外界提供最简单的编程接口。
- 继承：继承是从已有类得到继承信息创建新类的过程。提供继承信息的类被称为父类（超类、基类）；得到继承信息的类被称为子类（派生类）。继承让变化中的软件系统有了一定的延续性，同时继承也是封装程序中可变因素的重要手段。
- 多态：多态性是指允许不同子类型的对象对同一消息作出不同的响应。简单的说就是用同样的对象引用调用同样的方法但是做了不同的事情。多态性分为编译时的多态性和运行时的多态性。如果将对象的方法视为对象向外界提供的服务，那么运行时的多态性可以解释为：当 A 系统访问 B 系统提供的服务时，B 系统有多种提供服务的方式，但一切对 A 系统来说都是透明的。方法重载（overload）实现的是编译时的多态性（也称为前绑定），而方法重写（override）实现的是运行时的多态性（也称为后绑定）。运行时的多态是面向对象最精髓的东西，要实现多态需要做两件事：

第一：方法重写（子类继承父类并重写父类中已有的或抽象的方法）；

第二：对象造型（用父类型引用指向子类型对象，这样同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。

- 
- 抽象：抽象是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方面。抽象只关注对象有哪些属性和行为，并不关注这些行为的细节是什么。

## 4、Java 如何保证线程安全？

- 使用同步代码块
- 使用同步方法
- 使用 Lock 锁机制，通过创建 Lock 对象，采用 lock()加锁，unlock()解锁，来保护指定的代码块。

## 5、介绍 Spring MVC 的工作流程？

- 用户向服务端发送一次请求，这个请求会先到前端控制器 DispatcherServlet。
- DispatcherServlet 接收到请求后会调用 HandlerMapping 处理器映射器。由此得知，该请求该由哪个 Controller 来处理（并未调用 Controller，只是得知）
- DispatcherServlet 调用 HandlerAdapter 处理器适配器，告诉处理器适配器应该要去执行哪个 Controller
- HandlerAdapter 处理器适配器去执行 Controller 并得到 ModelAndView(数据和视图)，并层层返回给 DispatcherServlet
- DispatcherServlet 将 ModelAndView 交给 ViewReslover 视图解析器解析，然后返回真正的视图。
- DispatcherServlet 将模型数据填充到视图中
- DispatcherServlet 将结果响应给用户

## 6、Spring 框架中用到了哪些设计模式？

- 工厂设计模式：Spring 使用工厂模式通过 BeanFactory、ApplicationContext 创建 bean 对象。
- 代理设计模式：Spring AOP 功能的实现。
- 单例设计模式：Spring 中的 Bean 默认都是单例的。
- 模板方法模式：Spring 中 jdbcTemplate、hibernateTemplate 等以 Template 结尾的对数据库操作的类，它们就使用到了模板模式。
- 包装器设计模式：我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。这种模式让我们可以根据客户的需求能够动态切换不同的数据源。
- 观察者模式：Spring 事件驱动模型就是观察者模式很经典的一个应用。
- 适配器模式：Spring AOP 的增强或通知(Advice)使用到了适配器模式、spring MVC 中也是用到了适配器模式适配 Controller。

## 7、Redis 的特点是什么？

Redis 本质上是一个 Key-Value 类型的内存数据库，很像 Memcached，整个数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据 flush 到硬盘上进行保存。

因为是纯内存操作，Redis 的性能非常出色，每秒可以处理超过 10 万次读写操作，是已知性能最快的 Key-Value DB。

Redis 的出色之处不仅仅是性能，Redis 最大的魅力是支持保存多种数据结构，此外单个 value 的最大限制是 1GB，不像 Memcached 只能保存 1MB 的数据，因此 Redis 可以用来实现很多有用的功能。

比方说用他的 List 来做 FIFO 双向链表，实现一个轻量级的高性能消息队列服务，用他的 Set 可以做高性能的 tag 系统等等。另外 Redis 也可以对存入的 Key-Value 设置 expire 时间，因此也可以被当作一个功能加强版的 Memcached 来用。

Redis 的主要缺点是数据库容量受到物理内存的限制，不能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高性能操作和运算上。

## 8、为什么使用 Redis，有什么好处？

- 速度快，因为数据存在内存中，类似于 HashMap，HashMap 的优势就是查找和操作的时间复杂度都是 O(1)
- 支持丰富数据类型，支持 string, list, set, sorted set, hash
- 支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行
- 丰富的特性：可用于缓存，消息，按 key 设置过期时间，过期后将会自动删除

## 9、Redis 雪崩和击穿了解吗？

### 缓存击穿

- 问题：某个 KEY 失效的时候，正好有大量并发请求访问这个 KEY。
- 分析：跟穿透其实很像，属于比较偶然的。
- 解决办法：KEY 的更新操作添加全局互斥锁。完全以缓存为准，使用延迟异步加载的策略（异步线程负责维护缓存的数据，定期或根据条件触发更新），这样就不会触发更新。

### 缓存雪崩

- 问题：当某一时刻发生大规模的缓存失效的情况，导致大量的请求无法获取数据，从而将流量压力传导到数据库上，导致数据库压力过大甚至宕机。
- 原因：一般而言，缓存雪崩有 2 种可能性：大量的数据同一个时间失效：比如业务关系强相关的数据要求同时失效 Redis 宕机
- 分析：一般来说，由于更新策略、或者数据热点、缓存服务宕机等原因，可能会导致缓存数据同一个时间点大规模不可用，或者都更新。所以，需要我们的更新策略要在时间上合适，数据要均匀分享，缓存服务器要多台高可用。

- 
- 解决办法：更新策略在时间上做到比较平均。如果数据需要同一时间失效，可以给这批数据加上一些随机值，使得这批数据不要在同一个时间过期，降低数据库的压力。使用的热数据尽量分散到不同的机器上。多台机器做主从复制或者多副本，实现高可用。做好主从的部署，当主节点挂掉后，能快速的使用从结点顶上。实现熔断限流机制，对系统进行负载能力控制。对于非核心功能的业务，拒绝其请求，只允许核心功能业务访问数据库获取数据。服务降价：提供默认返回值，或简单的提示信息。

## 10、什么是面向对象，谈谈你的理解？

世间万物都可以看成一个对象。每个物体包括动态的行为和静态的属性，这些就构成了一个对象。

## 11、访问数据库除了 JDBC 还有什么？

- 自己封装 JDBC 的工具类
- Commons-Dbutils+dbcp 【QueryRunner】
- SpringJDBC 【JdbcTemplate】
- JPA 【配置文件、domain 实体类+注解、EntityManager】
- SpringDataJpa
- Hibernate 框架
- Mybatis



## 12、你知道有哪些设计原则？

- 遵循单一职责原则
- 开放-封闭原则
- 里氏代换原则（LSP）
- 依赖倒置原则
- 接口隔离原则（Interface Segregation Principle）
- 迪米特法则（Law of Demeter）

## 13、在生产环境 Linux 服务器上，发现某台运行 Java 服务的服务器的 CPU100%，不借助任何可视化工具，怎么进行问题的定位？

- top 找出进程 CPU 比较高 PID
- top -Hp PID 打印该 PID 进程下哪条线程的 CPU 占用比较高 tid
- printf "%x\n" tid 将该 id 进行 16 进制转换 tidhex
- jstack PID |grep tidhex 打印线程的堆栈信息

## 14、JDK 里面带的工具你知道哪些？

- jstat: 虚拟机进程状况工具
- jinfo: Java 配置信息工具
- jmap: Java 内存映像工具
- jhat: 虚拟机堆转储快照分析工具
- jstack: Java 堆栈跟踪工具
- JConsole: Java 监视与管理控制台
- VisualVM: 多合一故障处理工具

## 15、基本数据类型 bit 长度？

- byte: 1\*8
- short: 2\*8
- int: 4\*8
- long: 8\*8
- float: 4\*8
- double: 8\*8
- char: 2\*8
- boolean: 1\*8

## 16、char 能不能存中文？

可以，不过，如果某个特殊的汉字没有被包含在 unicode 编码字符集中，那么，这个 char 型变量中就不能存储这个特殊汉字。

## 17、谈谈你对泛型的理解？

Java 中的泛型有 3 种形式，泛型方法，泛型类，泛型接口。Java 通过在编译时类型擦除的方式来实现泛型。擦除时使用 Object 或者界定类型替代泛型，同时在要调用具体类型方法或者成员变量的时候插入强转代码，为了保证多态特性，Java 编译器还会为泛型类的子类生成桥接方法。类型信息在编译阶段被擦除之后，程序在运行期间无法获取类型参数所对应的具体类型。

## 18、Java 程序是怎样运行的？

- 首先通过 Javac 编译器将 .java 转为 JVM 可加载的 .class 字节码文件。
- Javac 是由 Java 编写的程序，编译过程可以分为：① 词法解析，通过空格分割出单词、操作符、控制符等信息，形成 token 信息流，传递给语法解析器。② 语法解析，把

- 
- token 信息流按照 Java 语法规则组装成语法树。③ 语义分析，检查关键字使用是否合理、类型是否匹配、作用域是否正确等。④ 字节码生成，将前面各个步骤的信息转换为字节码。
- 字节码必须通过类加载过程加载到 JVM 后才可以执行，执行有三种模式，解释执行、JIT 编译执行、JIT 编译与解释器混合执行（主流 JVM 默认执行的方式）。混合模式的优势在于解释器在启动时先解释执行，省去编译时间。
  - 之后通过即时编译器 JIT 把字节码文件编译成本地机器码。
  - Java 程序最初都是通过解释器进行解释执行的，当虚拟机发现某个方法或代码块的运行特别频繁，就会认定其为“热点代码”，热点代码的检测主要有基于采样和基于计数器两种方式，为了提高热点代码的执行效率，虚拟机会把它们编译成本地机器码，尽可能对代码优化，在运行时完成这个任务的后端编译器被称为即时编译器。
  - 还可以通过静态的提前编译器 AOT 直接把程序编译成与目标机器指令集相关的二进制代码。

## 19、GC root 有哪些？

- Thread-存活的线程。
- Java 虚拟机栈中的引用的对象。
- 方法区中的类静态属性引用的对象。（一般指被 static 修饰的对象，加载类的时候就加载到内存中。）
- 方法区中的常量引用的对象。
- 本地方法栈中的 JNI (native 方法) 引用的对象。
- Monitor Used-用于同步监控的对象。

## 20、栈帧的大小什么时候确定？

有时候编译期能够确定，有些时候函数的栈帧的大小在编译期并不确定。比如用了 VLA。所以一般会有两个寄存器 (IA-32 上就是 ebp 和 esp) 来记录栈帧的首尾地址。当进入一个函数时，首先把上个栈帧的首尾地址分别保存起来（一般做法是将 ebp 压栈、并将 esp 写入 ebp），接着再分配新的栈帧大小（先给 esp 减一个常数，如果需要动态分配再接着减）。

## 21、静态 filed 声明和构造器哪个先执行？

filed 声明先执行。

## 22、线程创建方式是什么？

- 通过继承 Thread 类创建线程类
- 实现 Runnable 接口创建线程类
- 通过 Callable 和 Future 接口创建线程

## 23、传统 I/O 跟 NIO 的区别？

- 所有 I/O 都被视为单个的字节的移动，通过一个称为 Stream 的对象一次移动一个字节。流 I/O 用于与外部世界接触。它也在内部使用，用于将对象转换为字节，然后再转换回对象。传统流 IO 的好处是使用简单，将底层的机制都抽象成流，但缺点就是性能不足。而且 IO 的各种流是阻塞的。这意味着，当一个线程调用 read() 或 write() 时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。
- 原来的 I/O 库(在 java.io.\* 中) 与 NIO 最重要的区别是数据打包和传输的方式。原来的 I/O 以流的方式处理数据，而 NIO 以块的方式处理数据。
- NIO 性能的优势就来源于缓冲的机制 (buffer 机制)，不管是读或者写都需要以块的形式写入到缓冲区中。NIO 实际上让我们对 IO 的操作更接近于操作系统的实际过程。
- NIO 作为非阻塞式的 IO，它的优点就在于，1、它由一个专门的线程去处理所有的 IO 事件，并负责分发；2、事件驱动，只有事件到了才会触发，而不是同步的监听这个事件；3、线程之间通过 wait, notify 等方式通讯。保证每次上下文切换都是有意义的。减少无谓的线程切换。
- 当我们在执行持续性的操作（如上传下载）时，IO 的方式是要优于 NIO 的。分清情况，合理选用。
- NIO 相对于 IO 流的优势：  
非阻塞  
buffer 机制  
流替代块



## 24、消息队列在各种场景下如何选型？

- 优先级队列；队列设置最大的优先级，之后每条消息设置对应的优先级，队列根据消息优先级进行消费，（在有可能队列堆积的情况下才有意义）；应用场景：不同业务消息推送。
- 延迟队列：消息发送后，并不想让消费者立即拿到消息，等待特定的事件后，消费者才能拿到并消费；应用场景：订单系统中订单支付 30 分钟内没有支付成功，那么将这个订单进行异常处理；远程操作智能设备在指定时间进行工作等。（rabbit 中没有延迟队列，但可以借助死信队列与 TTL 设置来完成）
- 死信队列：当消息在一个队列中变成死信之后，它能被重新被发送到另一个交换器（DLX 交换器）中，绑定 DLX 的队列就称为死信队列。
- 重试队列：消费端，一直不回传消费的结果，rocketmq 认为消息没收到，consumer 下一次拉取，broker 依然会发送该消息（有次数限制）。重试队列其实可以看成是一种回退队列，具体指消费端消费消息失败时，为防止消息无故丢失而重新将消息回滚到 Broker 中。
- 消费模式：推模式：对于 kafka 而言，由 Broker 主动推送消息至消费端，实时性较好，不过需要一定的流制机制来确保服务端推送过来的消息不会压垮消费端。拉模式：对于

kafka 而言，消费端主动向 Broker 端请求拉取(一般是定时或者定量)消息，实时性较推模式差，但是可以根据自身的处理能力而控制拉取的消息量。

- 消息回溯：重置消息 offset (如： kafka、rokctMq ) 一般消息在消费完成之后就被处理了，之后再也不能消费到该条消息。消息回溯正好相反，是指消息在消费完成之后，还能消费到之前被消费掉的消息。对于消息而言，经常面临的问题是“消息丢失”，至于是真正由于消息中间件的缺陷丢失还是由于使用方的误用而丢失一般很难追查，如果消息中间件本身具备消息回溯功能的话，可以通过回溯消费复现“丢失的”消息进而查出问题的源头之所在。消息回溯的作用远不止于此，比如还有索引恢复、本地缓存重建，有些业务补偿方案也可以采用回溯的方式来实现。
- 消息堆积：流量削峰是消息中间件的一个非常重要的功能，而这个功能其实得益于其消息堆积能力。从某种意义上讲，如果一个消息中间件不具备消息堆积的能力，那么就不能把它看做是一个合格的消息中间件。消息堆积分内存式堆积和磁盘式堆积。
- 消息持久化：持久化确保 MQ 的使用不只是一个部分场景的辅助工具，而是让 MQ 能像数据库一样存储核心的数据。有些功能是默认不开启的，需要进行配置。
- 多租户：也可以称为多重租赁技术，是一种软件架构技术，主要用来实现多用户的环境下公用相同的系统或程序组件，并且仍可以确保各用户间数据的隔离性。RabbitMQ 就能够支持多租户技术，每一个租户表示为一个 vhost，其本质上是一个独立的小型 RabbitMQ 服务器，又有自己独立的队列、交换器及绑定关系等，并且它拥有自己独立的权限。  
vhost 就像是物理机中的虚拟机一样，它们在各个实例间提供逻辑上的分离，为不同程序安全保密地允许数据，它既能将同一个 RabbitMQ 中的众多客户区分开，又可以避免队列和交换器等命名冲突。
- 跨语言支持：对很多公司而言，其技术栈体系中会有多种编程语言，如 C/C++、JAVA、Go、PHP 等，消息中间件本身具备应用解耦的特性，如果能够进一步的支持多客户端语言，那么就可以将此特性的效能扩大。跨语言的支持力度也可以从侧面反映出一个消息中间件的流行程度。
- 消息顺序消息：先进先出、逐条进行消费顾名思义，消息顺序性是指保证消息有序。这个功能有个很常见的应用场景就是 CDC (Change Data Chapture)，以 MySQL 为例，如果其传输的 binlog 的顺序出错，比如原本是先对一条数据加 1，然后再乘以 2，发送错序之后就变成了先乘以 2 后加 1 了，造成了数据不一致。
- 安全机制：在 Kafka 0.9 版本之后就开始增加了身份认证和权限控制两种安全机制。身份认证是指客户端与服务端连接进行身份认证，包括客户端与 Broker 之间、Broker 与 Broker 之间、Broker 与 ZooKeeper 之间的连接认证，目前支持 SSL、SASL 等认证机制。权限控制是指对客户端的读写操作进行权限控制，包括对消息或 Kafka 集群操作权限控制。权限控制是可插拔的，并支持与外部的授权服务进行集成。对于 RabbitMQ 而言，其同样提供身份认证 (TLS/SSL、SASL) 和 权限控制 (读写操作) 的安全机制。
- 事务支持：事务本身是一个并不陌生的词汇，事务是由事务开始 (Begin Transaction) 和事务结束 (End Transaction) 之间执行的全体操作组成。支持事务的消息中间件并不在少数，Kafka 和 RabbitMQ 都支持，不过此两者的事务是指生产者发生消息的事务，

---

要么发送成功，要么发送失败。消息中间件可以作为用来实现分布式事务的一种手段，但其本身并不提供全局分布式事务的功能。

## 25、Java 的安全性体现在哪里？

- Java SE 安全性概述 Java SE
- 平台基于一个动态、可扩展、基于标准、可互操作的安全架构。加密、身份验证和授权、公共密钥基础架构等安全特性是内置的。Java
- 安全模型基于一个可定制的“沙盒”，Java 软件程序可在其中安全运行，对系统或用户无潜在风险。
- Java 编译器和虚拟机强制实施的内置的语言安全特性：
- 强大的数据类型管理
- 自动内存管理
- 字节码验证
- 安全的类加载

## 26、static 方法怎么访问非 static 变量？

类的静态成员（变量和方法）都属于类本身，在类加载的时候就会分配内存，可以通过类名直接访问



## 27、讲下你理解的 Java 多继承？

- 若子类继承的父类中拥有相同的成员变量，子类在引用该变量时将无法判断使用哪个父类的成员变量
- 若一个子类继承的多个父类拥有相同方法，同时子类并未覆盖该方法（若覆盖，则直接使用子类中该方法），那么调用该方法时将无法确定调用哪个父类的方法。

## 28、Java 基本类型有哪些？

- byte 1
- short 2
- int 4
- long 8
- float 4
- double 8
- char 2
- boolean 1

## 29、线程池如果满了会怎么样？

- 
- 如果使用的是无界队列 `LinkedBlockingQueue`, 也就是无界队列的话, 没关系, 继续添加任务到阻塞队列中等待执行, 因为 `LinkedBlockingQueue` 可以近乎认为是一个无穷大的队列, 可以无限存放任务
  - 如果使用的是有界队列比如 `ArrayBlockingQueue`, 任务首先会被添加到 `ArrayBlockingQueue` 中, `ArrayBlockingQueue` 满了, 会根据 `maximumPoolSize` 的值增加线程数量, 如果增加了线程数量还是处理不过来, `ArrayBlockingQueue` 继续满, 那么则会使用拒绝策略 `RejectedExecutionHandler` 处理满了的任务, 默认是 `AbortPolicy`。

## 30、什么是双亲委派机制，它有什么作用？

双亲委派机制的意思是除了顶层的启动类加载器以外, 其余的类加载器, 在加载之前, 都会委派给它的父加载器进行加载。这样一层层向上传递, 直到祖先们都无法胜任, 它才会真正的加载。

- 通过带有优先级的层级关可以避免类的重复加载;
- 保证 Java 程序安全稳定运行, Java 核心 API 定义类型不会被随意替换。



### 1、I/O 流的分类

按照读写的单位大小来分:

- 字符流: 以字符为单位, 每次次读入或读出是 16 位数据。其只能读取字符类型数据。(Java 代码接收数据一般为 `char` 数组, 也可以是别的)
- 字节流: 以字节为单位, 每次次读入或读出是 8 位数据。可以读任何类型数据, 图片、文件、音乐视频等。(Java 代码接收数据只能为 `byte` 数组)

按照实际 IO 操作来分:

- 输出流: 从内存读出到文件。只能进行写操作。
- 输入流: 从文件读入到内存。只能进行读操作。
- 注意: 输出流可以帮助我们创建文件, 而输入流不会。

按照读写时是否直接与硬盘, 内存等节点连接分:

- 节点流: 直接与数据源相连, 读入或读出。
- 处理流: 也叫包装流, 是对一个对于已存在的流的连接进行封装, 通过所封装的流的功能调用实现数据读写。如添加个 Buffering 缓冲区。(意思就是有个缓存区, 等于软件和 mysql 中的 redis)

- 
- 注意：为什么要有处理流？主要作用是在读入或写出时，对数据进行缓存，以减少 I/O 的次数，以便下次更好更快的读写文件，才有了处理流。

## 2、字节流如何转为字符流？

字节输入流转字符输入流通过 `InputStreamReader` 实现，该类的构造函数可以传入 `InputStream` 对象。

字节输出流转字符输出流通过 `OutputStreamWriter` 实现，该类的构造函数可以传入 `OutputStream` 对象。

## 3、字节流和字符流，你更喜欢使用哪一个？

个人来说，更喜欢使用字符流，因为他们更新一些。许多在字符流中存在的特性，字节流中不存在。比如使用 `BufferedReader` 而不是 `BufferedInputStreams` 或 `DataInputStream`，使用 `newLine()`方法来读取下一行，但是在字节流中我们需要做额外的操作。

## 4、`System.out.println` 是什么？

`println` 是 `PrintStream` 的一个方法。`out` 是一个静态 `PrintStream` 类型的成员变量，`System` 是一个 `java.lang` 包中的类，用于和底层的操作系统进行交互。

## 5、什么是 Filter 流？



Filter Stream 是一种 IO 流主要作用是用来对存在的流增加一些额外的功能，像给目标文件增加源文件中不存在的行数，或者增加拷贝的性能。

## 5、有哪些可用的 Filter 流？

在 `java.io` 包中主要由 4 个可用的 filter Stream。两个字节 filter stream，两个字符 filter stream. 分别是 `FilterInputStream`, `FilterOutputStream`, `FilterReader` 和 `FilterWriter`. 这些类是抽象类，不能被实例化的。

## 6、有哪些 Filter 流的子类？

- `LineNumberInputStream` 给目标文件增加行号
- `DataInputStream` 有些特殊的方法如 `readInt()`, `readDouble()` 和 `readLine()` 等可以读取一个 `int`, `double` 和一个 `string` 一次性的,

- 
- BufferedInputStream 增加性能
  - PushbackInputStream 推送要求的字节到系统中

## 7、NIO 和 I/O 的主要区别

### 面向流与面向缓冲

Java IO 和 NIO 之间第一个最大的区别是，IO 是面向流的，NIO 是面向缓冲区的。Java IO 面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。Java NIO 的缓冲导向方法略有不同。数据读取到一个稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

### 阻塞与非阻塞 IO

Java IO 的各种流是阻塞的。这意味着，当一个线程调用 `read()` 或 `write()` 时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。Java NIO 的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取，而不是保持线程阻塞，所以直至数据变为可以读取之前，该线程可以继续做其他的事情。非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。线程通常将非阻塞 IO 的空闲时间用于在其它通道上执行 IO 操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

### 选择器（Selectors）

Java NIO 的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

### NIO 提供了与标准 IO 不同的 IO 工作方式：

Channels and Buffers（通道和缓冲区）：标准的 IO 基于字节流和字符流进行操作的，而 NIO 是基于通道（Channel）和缓冲区（Buffer）进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。

Asynchronous IO（异步 IO）：Java NIO 可以让你异步的使用 IO，例如：当线程从通道读取数据到缓冲区时，线程还是可以进行其他事情。当数据被写入到缓冲区时，线程可以继续处理它。从缓冲区写入通道也类似。

Selectors（选择器）：Java NIO 引入了选择器的概念，选择器用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个的线程可以监听多个数据通道。

## 8、BIO、NIO、AIO 有什么区别？

---

**BIO (Blocking I/O):** 同步阻塞 I/O 模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机 1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。

**NIO (New I/O):** NIO 是一种同步非阻塞的 I/O 模型，在 Java 1.4 中引入了 NIO 框架，对应 java.nio 包，提供了 Channel, Selector, Buffer 等抽象。NIO 中的 N 可以理解为 Non-blocking，不单纯是 New。它支持面向缓冲的，基于通道的 I/O 操作方法。NIO 提供了与传统 BIO 模型中的 Socket 和 ServerSocket 相对应 de 的 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现，两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞 I/O 来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发。

**AIO (Asynchronous I/O):** AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2，它是异步非阻塞的 IO 模型。异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 是异步 IO 的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO 操作本身是同步的。查阅网上相关资料，我发现就目前来说 AIO 的应用还不是很广泛，Netty 之前也尝试使用过 AIO，不过又放弃了。

## 9、NIO 有哪些核心组件？

### Channel Buffer Selector

Channel 和流有点类似。通过 Channel，我们即可以从 Channel 把数据写到 Buffer 中，也可以把数据从 Buffer 写入到 Channel，每个 Channel 对应一个 Buffer 缓冲区，Channel 会注册到 Selector。Selector 根据 Channel 上发生的读写事件，将请求交由某个空闲的线程处理，Selector 对应一个或多个线程，Channnel 和 Buffer 是可读可写的。

## 10、select、poll 和 epoll 什么区别

它们是 NIO 多路复用的三种实现机制，是有 Linux 系统提供。

- **select:** 无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作，会维护一个文件描述符 FD 的集合 fd\_set，将 fd\_set 从用户空间复制到内核空间。x86 fd\_set 是数组结构
- **poll:** 与 select 机制相似，fd\_set 结构进行优化，突破操作系统限制，pollfd 代替 fd\_set，链表结构

- 
- epoll：不再扫描所以 fd，只将用户关心的事件放在内核的一个事件表中，减少用户空间和内核空间的数据拷贝。epoll 可以理解为 event poll，不同于忙轮询和无差别轮询，epoll 会把哪个流发生了怎样的 I/O 事件通知我们。

## 11、什么是 Java 序列化，如何实现 Java 序列化？

序列化就是一种用来处理对象流的机制，将对象的内容进行流化。可以对流化后的对象进行读写操作，可以将流化后的对象传输于网络之间。

序列化是为了解决在对象流读写操作时所引发的问题 序列化的实现：将需要被序列化的类实现 Serialize 接口，没有需要实现的方法，此接口只是为了标注对象可被序列化的，然后使用一个输出流（如：FileOutputStream）来构造一个 ObjectOutputStream(对象流)对象，再使用 ObjectOutputStream 对象的 write(Object obj)方法就可以将参数 obj 的对象写出。

## 12、如何实现对象克隆？

有两种方式：

1. 实现 Cloneable 接口并重写 Object 类中的 clone()方法；
2. 实现 Serializable 接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深度克隆。

## 13、什么是缓冲区？有什么作用？

缓冲区就是一段特殊的内存区域，很多情况下当程序需要频繁地操作一个资源（如文件或数据库）则性能会很低，所以为了提升性能就可以将一部分数据暂时读写到缓存区，以后直接从此区域中读写数据即可，这样就可以显著的提升性能。

对于 Java 字符流的操作都是在缓冲区操作的，所以如果我们想在字符流操作中主动将缓冲区刷新到文件则可以使用 flush() 方法操作。

## 14、什么是阻塞 IO？什么是非阻塞 IO？

IO 操作包括：对硬盘的读写、对 socket 的读写以及外设的读写。

当用户线程发起一个 IO 请求操作（本文以读请求操作为例），内核会去查看要读取的数据是否就绪，对于阻塞 IO 来说，如果数据没有就绪，则会一直在那等待，直到数据就绪；对于非阻塞 IO 来说，如果数据没有就绪，则会返回一个标志信息告知用户线程当前要读的数据没有就绪。当数据就绪之后，便将数据拷贝到用户线程，这样才完成了一个完整的 IO 读请求操作，也就是说一个完整的 IO 读请求操作包括两个阶段：

- 查看数据是否就绪；

- 
- 进行数据拷贝（内核将数据拷贝到用户线程）。

那么阻塞（blocking IO）和非阻塞（non-blocking IO）的区别就在于第一个阶段，如果数据没有就绪，在查看数据是否就绪的过程中是一直等待，还是直接返回一个标志信息。

Java 中传统的 IO 都是阻塞 IO，比如通过 socket 来读数据，调用 read()方法之后，如果数据没有就绪，当前线程就会一直阻塞在 read 方法调用那里，直到有数据才返回；而如果是非阻塞 IO 的话，当数据没有就绪，read()方法应该返回一个标志信息，告知当前线程数据没有就绪，而不是一直在那里等待。

## 15、请说一下 PrintStream BufferedWriter PrintWriter 有什么不同？

PrintStream 类的输出功能非常强大，通常如果需要输出文本内容，都应该将输出流包装成 PrintStream 后进行输出。它还提供其他两项功能。与其他输出流不同，PrintStream 永远不会抛出 IOException；而是，异常情况仅设置可通过 checkError 方法测试的内部标志。另外，为了自动刷新，可以创建一个 PrintStream

BufferedWriter: 将文本写入字符输出流，缓冲各个字符从而提供单个字符，数组和字符串的高效写入。通过 write()方法可以将获取到的字符输出，然后通过 newLine()进行换行操作。BufferedWriter 中的字符流必须通过调用 flush 方法才能将其刷出去。并且 BufferedWriter 只能对字符流进行操作。如果要对字节流操作，则使用 BufferedInputStream。

PrintWriter 的 println 方法自动添加换行，不会抛异常，若关心异常，需要调用 checkError 方法看是否有异常发生，PrintWriter 构造方法可指定参数，实现自动刷新缓存（autoflush）。

## Kafka

### 1、Kafka 是什么？主要应用场景有哪些？

Kafka 是一个分布式流式处理平台。

流平台具有三个关键功能：

- 消息队列：发布和订阅消息流，这个功能类似于消息队列，这也是 Kafka 也被归类为消息队列的原因。
- 容错的持久方式存储记录消息流：Kafka 会把消息持久化到磁盘，有效避免了消息丢失的风险。
- 流式处理平台：在消息发布的时候进行处理，Kafka 提供了一个完整的流式处理类库。

Kafka 主要有两大应用场景：

- 消息队列：建立实时流数据管道，以可靠地在系统或应用程序之间获取数据。
- 数据处理：构建实时的流数据处理程序来转换或处理数据流。

### 2、和其他消息队列相比，Kafka 的优势在哪里？

---

我们现在经常提到 Kafka 的时候就已经默认它是一个非常优秀的消息队列了，我们也会经常拿它跟 RocketMQ、RabbitMQ 对比。我觉得 Kafka 相比其他消息队列主要的优势如下：

- 极致的性能：基于 Scala 和 Java 语言开发，设计中大量使用了批量处理和异步的思想，最高可以每秒处理千万级别的消息。
- 生态系统兼容性无可匹敌：Kafka 与周边生态系统的兼容性是最好的没有之一，尤其在大数据和流计算领域。

实际上在早期的时候 Kafka 并不是一个合格的消息队列，早期的 Kafka 在消息队列领域就像是一个衣衫褴褛的孩子一样，功能不完备并且有一些小问题比如丢失消息、不保证消息可靠性等等。当然，这也和 LinkedIn 最早开发 Kafka 用于处理海量的日志有很大关系，哈哈哈，人家本来最开始就不是为了作为消息队列滴，谁知道后面误打误撞在消息队列领域占据了一席之地。

### 3、什么是 Producer、Consumer、Broker、Topic、Partition？

Kafka 将生产者发布的消息发送到 Topic（主题）中，需要这些消息的消费者可以订阅这些 Topic（主题）。Kafka 比较重要的几个概念：

- Producer（生产者）：产生消息的一方。
- Consumer（消费者）：消费消息的一方。
- Broker（代理）：可以看作是一个独立的 Kafka 实例。多个 Kafka Broker 组成一个 Kafka Cluster。
- Topic（主题）：Producer 将消息发送到特定的主题，Consumer 通过订阅特定的 Topic(主题) 来消费消息。
- Partition（分区）：Partition 属于 Topic 的一部分。一个 Topic 可以有多个 Partition，并且同一 Topic 下的 Partition 可以分布在不同的 Broker 上，这也就表明一个 Topic 可以横跨多个 Broker。这正如我上面所画的图一样。

### 4、Kafka 的多副本机制了解吗？

Kafka 为分区（Partition）引入了多副本（Replica）机制。分区（Partition）中的多个副本之间会有一个叫做 leader 的家伙，其他副本称为 follower。我们发送的消息会被发送到 leader 副本，然后 follower 副本才能从 leader 副本中拉取消息进行同步。

生产者和消费者只与 leader 副本交互。你可以理解为其他副本只是 leader 副本的拷贝，它们的存在只是为了保证消息存储的安全性。当 leader 副本发生故障时会从 follower 中选举出一个 leader,但是 follower 中如果有和 leader 同步程度达不到要求的参加不了 leader 的竞选。

### 5、Kafka 的多分区（Partition）以及多副本（Replica）机制有什么好处呢？

- Kafka 通过给特定 Topic 指定多个 Partition, 而各个 Partition 可以分布在不同的 Broker 上, 这样便能提供比较好的并发能力（负载均衡）。

- 
- Partition 可以指定对应的 Replica 数, 这也极大地提高了消息存储的安全性, 提高了容灾能力, 不过也相应的增加了所需要的存储空间。

## 6、Zookeeper 在 Kafka 中的作用知道吗？

- Broker 注册：在 Zookeeper 上会有一个专门用来进行 Broker 服务器列表记录的节点。每个 Broker 在启动时, 都会到 Zookeeper 上进行注册, 即到 /brokers/ids 下创建属于自己的节点。每个 Broker 就会将自己的 IP 地址和端口等信息记录到该节点中去
- Topic 注册：在 Kafka 中, 同一个 Topic 的消息会被分成多个分区并将其分布在多个 Broker 上, 这些分区信息及与 Broker 的对应关系也都是由 Zookeeper 在维护。比如我创建了一个名字为 my-topic 的主题并且它有两个分区, 对应到 zookeeper 中会创建这些文件夹: /brokers/topics/my-topic/Partitions/0、/brokers/topics/my-topic/Partitions/1
- 负载均衡：上面也说过了 Kafka 通过给特定 Topic 指定多个 Partition, 而各个 Partition 可以分布在不同的 Broker 上, 这样便能提供比较好的并发能力。对于同一个 Topic 的不同 Partition, Kafka 会尽力将这些 Partition 分布到不同的 Broker 服务器上。当生产者产生消息后也会尽量投递到不同 Broker 的 Partition 里面。当 Consumer 消费的时候, Zookeeper 可以根据当前的 Partition 数量以及 Consumer 数量来实现动态负载均衡。

## 7、Kafka 如何保证消息的消费顺序？

我们在使用消息队列的过程中经常有业务场景需要严格保证消息的消费顺序, 比如我们同时发了 2 个消息, 这 2 个消息对应的操作分别对应的数据操作是:

- 更改用户会员等级。
- 根据会员等级计算订单价格。

假如这两条消息的消费顺序不一样造成的最终结果就会截然不同。

Kafka 中 Partition(分区)是真正保存消息的地方, 我们发送的消息都被放在了这里。而我们的 Partition(分区) 又存在于 Topic(主题) 这个概念中, 并且我们可以给特定 Topic 指定多个 Partition。

每次添加消息到 Partition(分区) 的时候都会采用尾加法, 如上图所示。Kafka 只能为我们保证 Partition(分区) 中的消息有序。

消息在被追加到 Partition(分区)的时候都会分配一个特定的偏移量 (offset)。Kafka 通过偏移量 (offset) 来保证消息在分区内的顺序性。

所以, 我们就有一种很简单的保证消息消费顺序的方法: 1 个 Topic 只对应一个 Partition。这样当然可以解决问题, 但是破坏了 Kafka 的设计初衷。

Kafka 中发送 1 条消息的时候, 可以指定 topic, partition, key,data (数据) 4 个参数。如果你发送消息的时候指定了 Partition 的话, 所有消息都会被发送到指定的 Partition。并且, 同一个 key 的消息可以保证只发送到同一个 partition, 这个我们可以采用表/对象的 id 来作为 key。

---

总结一下，对于如何保证 Kafka 中消息消费的顺序，有了下面两种方法：

- 1 个 Topic 只对应一个 Partition。
- 发送消息的时候指定 key/Partition。

## 8、Kafka 如何保证消息不丢失？

### 生产者丢失消息的情况

生产者(Producer) 调用 send 方法发送消息之后，消息可能因为网络问题并没有发送过去。

所以，我们不能默认在调用 send 方法发送消息之后消息发送成功了。为了确定消息是发送成功，我们要判断消息发送的结果。但是要注意的是 Kafka 生产者(Producer) 使用 send 方法发送消息实际上是异步的操作，我们可以通过 get()方法获取调用结果，但是这样也让它变为了同步操作。

### 消费者丢失消息的情况

我们知道消息在被追加到 Partition(分区)的时候都会分配一个特定的偏移量 (offset)。偏移量 (offset) 表示 Consumer 当前消费到的 Partition(分区)的所在的位置。Kafka 通过偏移量 (offset) 可以保证消息在分区内的顺序性。

当消费者拉取到了分区的某个消息之后，消费者会自动提交了 offset。自动提交的话会有一个问题，试想一下，当消费者刚拿到这个消息准备进行真正消费的时候，突然挂掉了，消息实际上并没有被消费，但是 offset 却被自动提交了。

解决办法也比较粗暴，我们手动关闭自动提交 offset，每次在真正消费完消息之后再自己手动提交 offset。但是，细心的朋友一定会发现，这样会带来消息被重新消费的问题。比如你刚刚消费完消息之后，还没提交 offset，结果自己挂掉了，那么这个消息理论上就会被消费两次。

## 9、Kafka 判断一个节点是否还活着有那两个条件？

- 节点必须可以维护和 ZooKeeper 的连接，ZooKeeper 通过心跳机制检查每个节点的连接；
- 如果节点是个 follower，他必须能及时的同步 leader 的写操作，延时不能太久。

## 10、producer 是否直接将数据发送到 broker 的 leader(主节点)？

producer 直接将数据发送到 broker 的 leader(主节点)，不需要在多个节点进行分发，为了

帮助 producer 做到这点，所有的 Kafka 节点都可以及时的告知：哪些节点是活动的，目标 topic 目标分区的 leader 在哪。这样 producer 就可以直接将消息发送到目的地了。

## 11、Kafa consumer 是否可以消费指定分区消息吗？

---

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，customer 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。

## 12、Kafka 高效文件存储设计特点是什么？

- 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。

## 13、partition 的数据如何保存到硬盘？

topic 中的多个 partition 以文件夹的形式保存到 broker，每个分区序号从 0 递增，且消息有序。

Partition 文件下有多个 segment (xxx.index, xxx.log)

segment 文件里的 大小和配置文件大小一致可以根据要求修改，默认为 1g。如果大小大于 1g 时，会滚动一个新的 segment 并且以上一个 segment 最后一条消息的偏移量命名。

## 14、kafka 生产数据时数据的分组策略是怎样的？

生产者决定数据产生到集群的哪个 partition 中，每一条消息都是以 (key, value) 格式，Key 是由生产者发送数据传入，所以生产者 (key) 决定了数据产生到集群的哪个 partition。

## 15、consumer 是推还是拉？

customer 应该从 brokers 拉取消息还是 brokers 将消息推送到 consumer，也就是 pull 还 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。

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。

## 16、kafka 维护消费状态跟踪的方法有什么？

大部分消息系统在 broker 端的维护消息被消费的记录：一个消息被分发到 consumer 后 broker 就马上进行标记或者等待 consumer 的通知后进行标记。这样也可以在消息在消费后立马就删除以减少空间占用。

## MySQL

### 1、数据库三大范式是什么

- 第一范式：每个列都不可以再拆分。
- 第二范式：在第一范式的基础上，非主键列完全依赖于主键，而不能是依赖于主键的一部分。
- 第三范式：在第二范式的基础上，非主键列只依赖于主键，不依赖于其他非主键。

在设计数据库结构的时候，要尽量遵守三范式，如果不遵守，必须有足够的理由。比如性能。事实上我们经常会为了性能而妥协数据库的设计。

### 2、MySQL 有关权限的表都有哪些？

MySQL 服务器通过权限表来控制用户对数据库的访问，权限表存放在 mysql 数据库里，由 mysql\_install\_db 脚本初始化。这些权限表分别 user, db, table\_priv, columns\_priv 和 host。下面分别介绍一下这些表的结构和内容：

- user 权限表：记录允许连接到服务器的用户帐号信息，里面的权限是全局级的。
- db 权限表：记录各个帐号在各个数据库上的操作权限。
- table\_priv 权限表：记录数据表级的操作权限。
- columns\_priv 权限表：记录数据列级的操作权限。
- host 权限表：配合 db 权限表对给定主机上数据库级操作权限作更细致的控制。这个权限表不受 GRANT 和 REVOKE 语句的影响。

### 3、MySQL 的 Binlog 有几种录入格式？分别有什么区别？

有三种格式，statement, row 和 mixed。

- statement 模式下，每一条会修改数据的 sql 都会记录在 binlog 中。不需要记录每一行的变化，减少了 binlog 日志量，节约了 IO，提高性能。由于 sql 的执行是有上下文的，因此在保存的时候需要保存相关的信息，同时还有一些使用了函数之类的语句无法被记录复制。
- row 级别下，不记录 sql 语句上下文相关信息，仅保存哪条记录被修改。记录单元为每一行的改动，基本是可以全部记下来但是由于很多操作，会导致大量行的改动(比如 alter table)，因此这种模式的文件保存的信息太多，日志量太大。
- mixed，一种折中的方案，普通操作使用 statement 记录，当无法使用 statement 的时候使用 row。

### 4、MySQL 存储引擎 MyISAM 与 InnoDB 区别

- 锁粒度方面：由于锁粒度不同，InnoDB 比 MyISAM 支持更高的并发；InnoDB 的锁粒度为行锁、MyISAM 的锁粒度为表锁、行锁需要对每一行进行加锁，所以锁的开销更大，但是能解决脏读和不可重复读的问题，相对来说也更容易发生死锁

- 
- 可恢复性上：由于 InnoDB 是有事务日志的，所以在产生由于数据库崩溃等条件后，可以根据日志文件进行恢复。而 MyISAM 则没有事务日志。
  - 查询性能上：MyISAM 要优于 InnoDB 因为 InnoDB 在查询过程中，是需要维护数据缓存，而且查询过程是先定位到行所在的数据块，然后在从数据块中定位到要查找的行；而 MyISAM 可以直接定位到数据所在的内存地址，可以直接找到数据。
  - 表结构文件上：MyISAM 的表结构文件包括：.frm(表结构定义), .MYI(索引), .MYD(数据)；而 InnoDB 的表数据文件为：.ibd 和 .frm(表结构定义)。

## 5、MyISAM 索引与 InnoDB 索引的区别？

- InnoDB 索引是聚簇索引，MyISAM 索引是非聚簇索引。
- InnoDB 的主键索引的叶子节点存储着行数据，因此主键索引非常高效。
- MyISAM 索引的叶子节点存储的是行数据地址，需要再寻址一次才能得到数据。
- InnoDB 非主键索引的叶子节点存储的是主键和其他带索引的列数据，因此查询时做到覆盖索引会非常高效。

## 6、什么是索引？

索引是一种特殊的文件（InnoDB 数据表上的索引是表空间的一个组成部分），它们包含着对数据表里所有记录的引用指针。

索引是一种数据结构。数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用 B 树及其变种 B+树。

更通俗的说，索引就相当于目录。为了方便查找书中的内容，通过对内容建立索引形成目录。索引是一个文件，它是要占据物理空间的。

## 7、索引有哪些优缺点？

### 索引的优点

- 可以大大加快数据的检索速度，这也是创建索引的最主要的原因。
- 通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

### 索引的缺点

- 时间方面：创建索引和维护索引要耗费时间，具体地，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，会降低增/改/删的执行效率；
- 空间方面：索引需要占物理空间。

## 8、索引有哪几种类型？

- 主键索引：数据列不允许重复，不允许为 NULL，一个表只能有一个主键。
- 唯一索引：数据列不允许重复，允许为 NULL 值，一个表允许多个列创建唯一索引。

- 
- 可以通过 ALTER TABLE table\_name ADD UNIQUE (column); 创建唯一索引。
  - 可以通过 ALTER TABLE table\_name ADD UNIQUE (column1,column2); 创建唯一组合索引。
  - 普通索引: 基本的索引类型, 没有唯一性的限制, 允许为 NULL 值。
    - 可以通过 ALTER TABLE table\_name ADD INDEX index\_name (column); 创建普通索引
    - 可以通过 ALTER TABLE table\_name ADD INDEX index\_name(column1, column2, column3); 创建组合索引。
  - 全文索引: 是目前搜索引擎使用的一种关键技术。
    - 可以通过 ALTER TABLE table\_name ADD FULLTEXT (column); 创建全文索引。

## 9、MySQL 中有哪几种锁？

- 表级锁: 开销小, 加锁快; 不会出现死锁; 锁定粒度大, 发生锁冲突的概率最高, 并发度最低。
- 行级锁: 开销大, 加锁慢; 会出现死锁; 锁定粒度最小, 发生锁冲突的概率最低, 并发度也最高。
- 页面锁: 开销和加锁时间界于表锁和行锁之间; 会出现死锁; 锁定粒度界于表锁和行锁之间, 并发度一般。

## 10、MySQL 中 InnoDB 支持的四种事务隔离级别名称, 以及逐级之间的区别?

SQL 标准定义的四个隔离级别为:

- read uncommitted : 读到未提交数据
- read committed: 脏读, 不可重复读
- repeatable read: 可重读
- serializable : 串行事物

## 11、CHAR 和 VARCHAR 的区别?

- CHAR 和 VARCHAR 类型在存储和检索方面有所不同
- CHAR 列长度固定为创建表时声明的长度, 长度值范围是 1 到 255
- 当 CHAR 值被存储时, 它们被用空格填充到特定长度, 检索 CHAR 值时需删除尾随空格。

## 12、主键和候选键有什么区别?

表格的每一行都由主键唯一标识,一个表只有一个主键。

主键也是候选键。按照惯例, 候选键可以被指定为主键, 并且可以用于任何外键引用。

## 13、如何在 Unix 和 MySQL 时间戳之间进行转换?

---

UNIX\_TIMESTAMP 是从 Mysql 时间戳转换为 Unix 时间戳的命令 FROM\_UNIXTIME 是从 Unix 时间戳转换为 Mysql 时间戳的命令。

## 14、MyISAM 表类型将在哪里存储，并且还提供其存储格式？

每个 MyISAM 表格以三种格式存储在磁盘上：

- “.frm” 文件 存储表定义
- 数据文件具有 “.MYD” (MYData) 扩展名
- 索引文件具有 “.MYI” (MYIndex) 扩展名

## 15、MySQL 里记录货币用什么字段类型好

NUMERIC 和 DECIMAL 类型被 Mysql 实现为同样的类型，这在 SQL92 标准允许。他们被用于保存值，该值的准确精度是极其重要的值，例如与金钱有关的数据。当声明一个类是这些类型之一时，精度和规模的能被(并且通常是)指定。

例如：

```
salary DECIMAL(9,2)
```

在这个例子中，9(precision)代表将被用于存储值的总的小数位数，而 2(scale)代表将被用于存储小数点后的位数。

因此，在这种情况下，能被存储在 salary 列中的值的范围是从 -9999999.99 到 9999999.99。

## 16、创建索引时需要注意什么？

- 非空字段：应该指定列为 NOT NULL，除非你想存储 NULL。在 mysql 中，含有空值的列很难进行查询优化，因为它们使得索引、索引的统计信息以及比较运算更加复杂。应该用 0、一个特殊的值或者一个空串代替空值；
- 取值离散大的字段：（变量各个取值之间的差异程度）的列放到联合索引的前面，可以通过 count() 函数查看字段的差异值，返回值越大说明字段的唯一值越多字段的离散程度高；
- 索引字段越小越好：数据库的数据存储以页为单位一页存储的数据越多一次 IO 操作获取的数据越大效率越高。

## 17、使用索引查询一定能提高查询的性能吗？为什么

通常，通过索引查询数据比全表扫描要快。但是我们也必须注意到它的代价。

索引需要空间来存储，也需要定期维护，每当有记录在表中增减或索引列被修改时，索引本身也会被修改。这意味着每条记录的 INSERT, DELETE, UPDATE 将为此多付出 4, 5 次的磁盘 I/O。因为索引需要额外的存储空间和处理，那些不必要的索引反而会使查询反应时间变慢。使用索引查询不一定能提高查询性能，索引范围查询(INDEX RANGE SCAN)适用于两种情况：

- 基于一个范围的检索，一般查询返回结果集小于表中记录数的 30%

- 
- 基于非唯一性索引的检索

## 18、百万级别或以上的数据如何删除

关于索引：由于索引需要额外的维护成本，因为索引文件是单独存在的文件，所以当我们对数据的增加,修改,删除,都会产生额外的对索引文件的操作,这些操作需要消耗额外的 IO,会降低增/改/删的执行效率。所以，在我们删除数据库百万级别数据的时候，查询 MySQL 官方手册得知删除数据的速度和创建的索引数量是成正比的。

- 所以我们想要删除百万数据的时候可以先删除索引（此时大概耗时三分多钟）
- 然后删除其中无用数据（此过程需要不到两分钟）
- 删除完成后重新创建索引(此时数据较少)创建索引也非常快，约十分钟左右。
- 与之前的直接删除绝对是要快速很多，更别说万一删除中断,一切删除会回滚。那更是坑了。

## 19、什么是最左前缀原则？什么是最左匹配原则

顾名思义，就是最左优先，在创建多列索引时，要根据业务需求，where 子句中使用最频繁的一列放在最左边。

最左前缀匹配原则，非常重要的原则，mysql 会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如  $a = 1 \text{ and } b = 2 \text{ and } c > 3 \text{ and } d = 4$  如果建立(a,b,c,d)顺序的索引，d 是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d 的顺序可以任意调整。

=和 in 可以乱序，比如  $a = 1 \text{ and } b = 2 \text{ and } c = 3$  建立(a,b,c)索引可以任意顺序，mysql 的查询优化器会帮你优化成索引可以识别的形式。

## 20、什么是聚簇索引？何时使用聚簇索引与非聚簇索引

- 聚簇索引：将数据存储与索引放到了一块，找到索引也就找到了数据
- 非聚簇索引：将数据存储于索引分开结构，索引结构的叶子节点指向了数据的对应行，myisam 通过 key\_buffer 把索引先缓存到内存中，当需要访问数据时（通过索引访问数据），在内存中直接搜索索引，然后通过索引找到磁盘相应数据，这也就是为什么索引不在 key buffer 命中时，速度慢的原因。

## 21、MySQL 连接器

首先需要在 MySQL 客户端登陆才能使用，所以需要 个连接器 来连接用户和 MySQL 数据库，我们一般是使用 mysql-u 用户名-p 密码

来进行 MySQL 登陆，和服务端建立连接。在完成 TCP 握手后，连接器会根据你输入的用户名和密码验证你的登录身份。如果用户名或者密码错误，MySQL 就会提示 Access denied for user, 来结束执行。如果登录成功后，MySQL 会根据权限表中的记录来判定你的权限。

## 22、MySQL 查询缓存

连接完成后，你就可以执行 SQL 语句了，这行逻辑就会来到第二步：查询缓存。

MySQL 在得到一个执行请求后，会首先去查询缓存中查找，是否执行过这条 SQL 语句，之前执行过的语句以及结果会以 key-value 对的形式，被直接放在内存中。key 是查询语句，value 是查询的结果。

如果通过 key 能够查找到这条 SQL 语句，就直接返回 SQL 的执行结果。

如果语句不在查询缓存中，就会继续后面的执行阶段。执行完成后，执行结果就会被放入查询缓存中。

可以看到，如果查询命中缓存，MySQL 不需要执行后面的复杂操作，就可以直接返回结果，效率会很高。

## 23、MySQL 分析器

如果没有命中查询，就开始执行真正的 SQL 语句。

- 首先，MySQL 会根据你写的 SQL 语句进行解析，分析器会先做词法分析，你写的 SQL 就是由多个字符串和空格组成的一条 SQL 语句，MySQL 需要识别出里面的字符串是什么，代表什么。
- 然后进行语法分析，根据词法分析的结果，语法分析器会根据语法规则，判断你输入的这个 SQL 语句是否满足 MySQL 语法。如果 SQL 语句不正确，就会提示 You have an error in your SQL syntax。

## 24、MySQL 优化器

经过分析器的词法分析和语法分析后，你这条 SQL 就合法了，MySQL 就知道你要做什么了。但是在执行前，还需要进行优化器的处理，优化器会判断你使用了哪种索引，使用了何种连接，优化器的作用就是确定效率最高的执行方案。

## 25、MySQL 执行器

MySQL 通过分析器知道了你的 SQL 语句是否合法，你想要做什么操作，通过优化器知道了该怎么做效率最高，然后就进入了执行阶段，开始执行这条 SQL 语句。在执行阶段，MySQL 首先会判断你有没有执行这条语句的权限，没有权限的话，就会返回没有权限的错误。如果有权限，就打开表继续执行。打开表的时候，执行器就会根据表的引擎定义，去使用这个引擎提供的接口。对于有索引的表，执行的逻辑也差不多。

## 26、什么是临时表，何时删除临时表？

什么是临时表？MySQL 在执行 SQL 语句的过程中通常会临时创建一些存储中间结果集的表，临时表只对当前连接可见，在连接关闭时，临时表会被删除并释放所有表空间。

临时表分为两种：一种是内存临时表，一种是磁盘临时表，什么区别呢？内存临时表使用的是 MEMORY 存储引擎，而临时表采用的是 MyISAM 存储引擎。

MySQL 会在下面这几种情况产生临时表。

- 使用 UNION 查询:UNION 有两种,一种是 UNION,一种是 UNION ALL,它们都用于联合查询;区别是使用 UNION 会去掉两个表中的重复数据,相当于对结果集做了一下去重(distinct)。使用 UNIONALL,则不会排重,返回所有的行。使用 UNION 查询会产生临时表。
- 使用 TEMPTABLE 算法或者是 UNION 查询中的视图。TEMPTABLE 算法是一种创建临时表的算法,它是将结果放置到临时表中,意味这要 MySQL 要先创建好一个临时表,然后将结果放到临时表中去,然后再使用这个临时表进行相应的查询。
- ORDER BY 和 GROUPBY 的子句不一样时也会产生临时表。
- DISTINCT 查询并且加上 ORDER BY 时;
- SQL 中用到 SQL\_SMALL\_RESULT 选项时;如果查询结果比较小的时候,可以加上 SQL\_SMALL\_RESULT 来优化,产生临时表
- FROM 中的子查询;
- EXPLAIN 查看执行计划结果的 Extra 列中,如果使用 Using Temporary 就表示会用到临时表。

## 27、谈谈 SQL 优化的经验

- 查询语句无论是使用哪种判断条件等于、小于、大于, WHERE 左侧的条件查询字段不要使用函数或者表达式
- 使用 EXPLAIN 命令优化你的 SELECT 查询,对于复杂、效率低的 sql 语句,我们通常是使用 explainsql 来分析这条 sql 语句,这样方便我们分析,进行优化。
- 当你的 SELECT 查询语句只需要使用一条记录时,要使用 LIMIT 1。不要直接使用 SELECT\*,而应该使用具体需要查询的表字段,因为使用 EXPLAIN 进行分析时,SELECT"使用的是全表扫描,也就是 type =all 。
- 为每一张表设置一个 ID 属性。
- 避免在 WHERE 字句中对字段进行 NULL
- 判断避免在 WHERE 中使用!或>操作符
- 使用 BETWEEN AND 替代 IN
- 为搜索字段创建索引
- 选择正确的存储引擎, InnoDB、MyISAM、MEMORY 等
- 使用 LIKE%abc%不会走索引,而使用 LIKE abc%会走索引。
- 对于枚举类型的字段(即有固定罗列值的字段),建议使用 ENUM 而不是 VARCHAR,如性别、星期、类型、类别等。
- 拆分大的 DELETE 或 INSERT 语句
- 选择合适的字段类型,选择标准是尽可能小、尽可能定长、尽可能使用整数。
- 字段设计尽可能使用 NOT NULL
- 进行水平切割或者垂直分割

## 28、什么叫外链接?

外连接分为三种,分别是是左外连接(LEFT OUTER JOIN 或 LEFT JOIN 右外连接(RIGHT OUTER JOIN 或 RIGHT JOIN)、全外连接(FULL OUTER JOIN 或 FULLJOIN)。

左外连接:又称为左连接,这种连接方式会显示左表不符合条件的数据行,右边不符合条件的数据行直接显示 NULL。

右外连接:也被称为右连接,他与左连接相对,这种连接方式会显示右表不符合条件的数据行,左表不符合条件的数据行直接显示 NULL。

## 29、什么叫内链接？

结合两个表中相同的字段，返回关联字段相符的记录就是内链接。



## 30、使用 union 和 union all 时需要注意些什么？

通过 union 连接的 SQL 分别单独取出的列数必须相同。

使用 union 时，多个相等的行将会被合并，由于合升比较耗时，一般不直接使用 union 进行合并，而是通常采用 union all 进行合并。

## 31、MyISAM 存储引擎的特点

在 5.1 版本之前，MyISAM 是 MySQL 的默认存储引擎，MyISAM 并发性比较差，使用的场景比较少主要特点是：

- 不支持事务操作，ACID 的特性也就不存在了，这一设计是为了性能和效率考虑的，
- 不支持外键操作，如果强行增加外键，MySQL 不会报错，只不过外键不起作用。
- MyISAM 默认的锁粒度是表级锁，所以并发性能比较差，加锁比较快，锁冲突比较少，不太容易发生死锁的情况。
- MyISAM 会在磁盘上存储三个文件，文件名和表名相同，扩展名分别是 frm(存储表定义)、MYD(MYData，存储数据)、MYI(MyIndex，存储索引)。这里需要特别注意的是 MyISAM 只缓存索引文件，并不缓存数据文件。
- MyISAM 支持的索引类型有全局索引(Full-Text)、B-Tree 索引、R-Tree 索引
  - Full-Text 紴引:它的出现是为了解决针对文本的模糊查询效率较低的问题。
  - B-Tree 紴引:所有的索引节点都按照平衡树的数据结构来存储，所有的索引数据节点都在叶节点
  - R-Tree 紴引:它的存储方式和 B-Tree 紹引有一些区别，主要设计用于存储空间和多维数据的字段做索引目前的 MySQL 版本仅支持 geometry 类型的字段作索引，相对于 BTREE,RTREE 的优势在于范围查找。
- 数据库所在主机如果宕机，MyISAM 的数据文件容易损坏，而且难以恢复。
- 增删改查性能方面:SELECT 性能较高，适用于查询较多的情况

## 32、InnoDB 存储引擎的特点

---

自从 MySQL5.1 之后，默认的存储引擎变成了 InnoDB 存储引擎，相对于 MyISAM，InnoDB 存储引擎有了较大的改变，它的主要特点是

- 支持事务操作，具有事务 ACID 隔离特性，默认的隔离级别是可重复读(repeatable-read)、通过 MVCC(并发版本控制)来实现的。能够解决 脏读 和 不可重复读 的问题。InnoDB 支持外键操作。
- InnoDB 默认的锁粒度行级锁，并发性能比较好，会发生死锁的情况。
- 和 MyISAM 一样的是，InnoDB 存储引擎也有 frm 文件存储表结构定义，但是不同的是，InnoDB 的表数据与索引数据是存储在一起的，都位于 B+树的叶子节点上，而 MyISAM 的表数据和索引数据是分开的。
- InnoDB 有安全的日志文件，这个日志文件用于恢复因数据库崩溃或其他情况导致的数据丢失问题，保证数据的一致性。
- InnoDB 和 MyISAM 支持的索引类型相同，但具体实现因为文件结构的不同有很大差异。
- 增删改查性能方面，果执行大量的增删改操作，推荐使用 InnoDB 存储引擎，它在删除操作时是对行删除，不会重建表。

## Netty

### 1、为什么要用 Netty 呢？

因为 Netty 具有下面这些优点，并且相比于直接使用 JDK 自带的 NIO 相关的 API 来说更加易用。

- 统一的 API，支持多种传输类型，阻塞和非阻塞的。
- 简单而强大的线程模型。
- 自带编解码器解决 TCP 粘包/拆包问题。
- 自带各种协议栈。
- 真正的无连接数据包套接字支持。
- 比直接使用 Java 核心 API 有更高的吞吐量、更低的延迟、更低的资源消耗和更少的内存复制。
- 安全性不错，有完整的 SSL/TLS 以及 StartTLS 支持。
- 社区活跃
- 成熟稳定，经历了大型项目的使用和考验，而且很多开源项目都使用到了 Netty，比如我们经常接触的 Dubbo、RocketMQ 等等。

### 2、Netty 有哪些应用场景？

理论上来说，NIO 可以做的事情，使用 Netty 都可以做并且更好。Netty 主要用来做网络通信：

- 作为 RPC 框架的网络通信工具：我们在分布式系统中，不同服务节点之间经常需要相互调用，这个时候就需要 RPC 框架了。不同服务节点之间的通信是如何做的呢？可以使用 Netty 来做。比如我调用另外一个节点的方法的话，至少是要让对方知道我调用的是哪个类中的哪个方法以及相关参数吧！

- 
- 实现一个自己的 HTTP 服务器：通过 Netty 我们可以自己实现一个简单的 HTTP 服务器，这个大家应该不陌生。说到 HTTP 服务器的话，作为 Java 后端开发，我们一般使用 Tomcat 比较多。一个最基本的 HTTP 服务器可以处理常见的 HTTP Method 的请求，比如 POST 请求、GET 请求等等。
  - 实现一个即时通讯系统：使用 Netty 我们可以实现一个可以聊天类似微信的即时通讯系统，这方面的开源项目还蛮多的，可以自行去 Github 找一找。
  - 实现消息推送系统：市面上有很多消息推送系统都是基于 Netty 来做的。

### 3、Netty 的优势有哪些？

- 使用简单：封装了 NIO 的很多细节，使用更简单。
- 功能强大：预置了多种编解码功能，支持多种主流协议。
- 定制能力强：可以通过 ChannelHandler 对通信框架进行灵活地扩展。
- 性能高：通过与其他业界主流的 NIO 框架对比，Netty 的综合性能最优。
- 稳定：Netty 修复了已经发现的所有 NIO 的 bug，让开发人员可以专注于业务本身。
- 社区活跃：Netty 是活跃的开源项目，版本迭代周期短，bug 修复速度快。

### 3、Netty 核心组件有哪些？分别有什么作用？

#### Channel

Channel 接口是 Netty 对网络操作抽象类，它除了包括基本的 I/O 操作，如 bind()、connect()、read()、write() 等。

比较常用的 Channel 接口实现类是 NioServerSocketChannel（服务端）和 NioSocketChannel（客户端），这两个 Channel 可以和 BIO 编程模型中的 ServerSocket 以及 Socket 两个概念对应上。Netty 的 Channel 接口所提供的 API，大大地降低了直接使用 Socket 类的复杂性。

#### EventLoop

EventLoop 的主要作用实际就是负责监听网络事件并调用事件处理器进行相关 I/O 操作的处理。

#### ChannelFuture

Netty 是异步非阻塞的，所有的 I/O 操作都为异步的。因此，我们不能立刻得到操作是否执行成功，但是，你可以通过 ChannelFuture 接口的 addListener() 方法注册一个 ChannelFutureListener，当操作执行成功或者失败时，监听就会自动触发返回结果。

#### ChannelHandler 和 ChannelPipeline

---

ChannelHandler 是消息的具体处理器。他负责处理读写操作、客户端连接等事情。

ChannelPipeline 为 ChannelHandler 的链，提供了一个容器并定义了用于沿着链传播入站和出站事件流的 API 。当 Channel 被创建时，它会被自动地分配到它专属的 ChannelPipeline 。

我们可以在 ChannelPipeline 上通过 addLast() 方法添加一个或者多个 ChannelHandler ，因为一个数据或者事件可能会被多个 Handler 处理。当一个 ChannelHandler 处理完之后就将数据交给下一个 ChannelHandler 。

## 5、EventloopGroup 了解么?和 EventLoop 啥关系?

EventLoopGroup 包含多个 EventLoop (每一个 EventLoop 通常内部包含一个线程) ,上面我们已经说了 EventLoop 的主要作用实际就是负责监听网络事件并调用事件处理器进行相关 I/O 操作的处理。

并且 EventLoop 处理的 I/O 事件都将在它专有的 Thread 上被处理，即 Thread 和 EventLoop 属于 1 : 1 的关系，从而保证线程安全。

Boss EventloopGroup 用于接收连接，Worker EventloopGroup 用于具体的处理 (消息的读写以及其他逻辑处理) 。

当客户端通过 connect 方法连接服务端时，bossGroup 处理客户端连接请求。当客户端处理完成后，会将这个连接提交给 workerGroup 来处理，然后 workerGroup 负责处理其 IO 相关操作。

## 6、请说下对 Bootstrap 和 ServerBootstrap 的了解?

- Bootstrap 通常使用 connet() 方法连接到远程的主机和端口，作为一个 Netty TCP 协议通信中的客户端。另外，Bootstrap 也可以通过 bind() 方法绑定本地的一个端口，作为 UDP 协议通信中的一端。
- ServerBootstrap 通常使用 bind() 方法绑定本地的端口上，然后等待客户端的连接。
- Bootstrap 只需要配置一个线程组— EventLoopGroup ,而 ServerBootstrap 需要配置两个线程组— EventLoopGroup ，一个用于接收连接，一个用于具体的处理。

## 7、请说下 Netty 线程模型?

在 Netty 主要靠 NioEventLoopGroup 线程池来实现具体的线程模型的。

我们实现服务端的时候，一般会初始化两个线程组：

- bossGroup：接收连接。
- workerGroup：负责具体的处理，交由对应的 Handler 处理。

**单线程模型：**

一个线程需要执行处理所有的 accept、read、decode、process、encode、send 事件。对于高负载、高并发，并且对性能要求比较高的场景不适用。

## 多线程模型

一个 Acceptor 线程只负责监听客户端的连接，一个 NIO 线程池负责具体处理 accept、read、decode、process、encode、send 事件。满足绝大部分应用场景，并发连接量不大的时候没啥问题，但是遇到并发连接大的时候就可能会出现问题，成为性能瓶颈。

## 主从多线程模型

从一个 主线程 NIO 线程池中选择一个线程作为 Acceptor 线程，绑定监听端口，接收客户端连接的连接，其他线程负责后续的接入认证等工作。连接建立完成后，Sub NIO 线程池负责具体处理 I/O 读写。如果多线程模型无法满足你的需求的时候，可以考虑使用主从多线程模型。

## 8、Netty 服务端和客户端的启动过程是怎样的？

### 服务端

1. 首先你创建了两个 NioEventLoopGroup 对象实例：bossGroup 和 workerGroup。
  - bossGroup : 用于处理客户端的 TCP 连接请求。
  - workerGroup : 负责每一条连接的具体读写数据的处理逻辑，真正负责 I/O 读写操作，交由对应的 Handler 处理。
2. 接下来 我们创建了一个服务端启动引导/辅助类：ServerBootstrap，这个类将引导我们进行服务端的启动工作。
3. 通过 .group() 方法给引导类 ServerBootstrap 配置两大线程组，确定了线程模型。
4. 通过 channel()方法给引导类 ServerBootstrap 指定了 IO 模型为 NIO
  - NioServerSocketChannel : 指定服务端的 IO 模型为 NIO，与 BIO 编程模型中的 ServerSocket 对应。
  - NioSocketChannel : 指定客户端的 IO 模型为 NIO，与 BIO 编程模型中的 Socket 对应
5. 通过 .childHandler() 给引导类创建一个 ChannelInitializer，然后制定了服务端消息的业务处理逻辑 HelloServerHandler 对象
6. 调用 ServerBootstrap 类的 bind()方法绑定端口

### 客户端

- 创建一个 NioEventLoopGroup 对象实例
- 创建客户端启动的引导类是 Bootstrap
- 通过 .group() 方法给引导类 Bootstrap 配置一个线程组
- 通过 channel()方法给引导类 Bootstrap 指定了 IO 模型为 NIO
- 通过 .childHandler()给引导类创建一个 ChannelInitializer，然后制定了客户端消息的业务处理逻辑 HelloClientHandler 对象

- 
- 调用 Bootstrap 类的 connect()方法进行连接，这个方法需要指定两个参数：
  - inetHost : ip 地址
  - inetPort : 端口号

## 9、什么是 TCP 粘包/拆包？

TCP 粘包/拆包 就是你基于 TCP 发送数据的时候，出现了多个字符串“粘”在了一起或者一个字符串被“拆”开的问题。

## 10、如何在 Netty 中解决 TCP 粘包问题？

### 1. 使用 Netty 自带的解码器

- LineBasedFrameDecoder：发送端发送数据包的时候，每个数据包之间以换行符作为分隔，LineBasedFrameDecoder 的工作原理是它依次遍历 ByteBuf 中的可读字节，判断是否有换行符，然后进行相应的截取。
- DelimiterBasedFrameDecoder：可以自定义分隔符解码器，LineBasedFrameDecoder 实际上是一种特殊的 DelimiterBasedFrameDecoder 解码器。
- FixedLengthFrameDecoder：固定长度解码器，它能够按照指定的长度对消息进行相应的拆包。LengthFieldBasedFrameDecoder：

### 2. 自定义序列化编解码器

在 Java 中自带的有实现 Serializable 接口来实现序列化，但由于它性能、安全性等原因一般情况下是不会被使用到的。

通常情况下，我们使用 Protostuff、Hessian2、json 序列方式比较多，另外还有一些序列化性能非常好的序列化方式也是很好的选择：

- 专门针对 Java 语言的：Kryo, FST 等等
- 跨语言的：Protostuff（基于 protobuf 发展而来），ProtoBuf, Thrift, Avro, MsgPack 等等

## 11、TCP 长连接和短连接了解么？

我们知道 TCP 在进行读写之前，server 与 client 之间必须提前建立一个连接。建立连接的过程，需要我们常说的三次握手，释放/关闭连接的话需要四次挥手。这个过程是比较消耗网络资源并且有时间延迟的。

所谓，短连接说的就是 server 端与 client 端建立连接之后，读写完成之后就关闭掉连接，如果下一次再要互相发送消息，就要重新连接。短连接的优点很明显，就是管理和实现都比较简单，缺点也很明显，每一次的读写都要建立连接必然会带来大量网络资源的消耗，并且连接的建立也需要耗费时间。

长连接说的就是 client 向 server 双方建立连接之后，即使 client 与 server 完成一次读写，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。长连接的可以省

---

去较多的 TCP 建立和关闭的操作，降低对网络资源的依赖，节约时间。对于频繁请求资源的客户来说，非常适用长连接。

## 12、为什么需要心跳机制？Netty 中心跳机制了解么？

在 TCP 保持长连接的过程中，可能会出现断网等网络异常出现，异常发生的时候，client 与 server 之间如果没有交互的话，他们是无法发现对方已经掉线的。为了解决这个问题，我们就需要引入 心跳机制 。

心跳机制的工作原理是：在 client 与 server 之间在一定时间内没有数据交互时，即处于 idle 状态时，客户端或服务器就会发送一个特殊的数据包给对方，当接收方收到这个数据报文后，也立即发送一个特殊的数据报文，回应发送方，此即一个 PING-PONG 交互。所以，当某一段收到心跳消息后，就知道了对方仍然在线，这就确保 TCP 连接的有效性。

TCP 实际上自带的就有长连接选项，本身是也有心跳包机制，也就是 TCP 的选项：SO\_KEEPALIVE。但是，TCP 协议层面的长连接灵活性不够。所以，一般情况下我们都是在应用层协议上实现自定义心跳机制的，也就是在 Netty 层面通过编码实现。通过 Netty 实现心跳机制的话，核心类是 IdleStateHandler 。

## 13、讲讲 Netty 的零拷贝

零复制（英语：Zero-copy；也译零拷贝）技术是指计算机执行操作时，CPU 不需要先将数据从某处内存复制到另一个特定区域。这种技术通常用于通过网络传输文件时节省 CPU 周期和内存带宽。

在 OS 层面上的 Zero-copy 通常指避免在 用户态(User-space) 与 内核态(Kernel-space) 之间来回拷贝数据。而在 Netty 层面，零拷贝主要体现在对于数据操作的优化。Netty 中的零拷贝体现在以下几个方面：

- 使用 Netty 提供的 CompositeByteBuf 类，可以将多个 ByteBuf 合并为一个逻辑上的 ByteBuf，避免了各个 ByteBuf 之间的拷贝。
- ByteBuf 支持 slice 操作，因此可以将 ByteBuf 分解为多个共享同一个存储区域的 ByteBuf，避免了内存的拷贝。
- 通过 FileRegion 包装的 FileChannel.transferTo 实现文件传输，可以直接将文件缓冲区的数据发送到目标 Channel，避免了传统通过循环 write 方式导致的内存拷贝问题。

## 14、Netty 和 Tomcat 的区别？

- 作用不同：Tomcat 是 Servlet 容器，可以视为 Web 服务器，而 Netty 是异步事件驱动的网络应用程序框架和工具用于简化网络编程，例如 TCP 和 UDP 套接字服务器。
- 协议不同：Tomcat 是基于 http 协议的 Web 服务器，而 Netty 能通过编程自定义各种协议，因为 Netty 本身自己能编码/解码字节流，所有 Netty 可以实现，HTTP 服务器、FTP 服务器、UDP 服务器、RPC 服务器、WebSocket 服务器、Redis 的 Proxy 服务器、MySQL 的 Proxy 服务器等等。

## 15、Netty 发送消息有几种方式？

Netty 有两种发送消息的方式：

- 直接写入 Channel 中，消息从 ChannelPipeline 当中尾部开始移动；
- 写入和 ChannelHandler 绑定的 ChannelHandlerContext 中，消息从 ChannelPipeline 中的下一个 ChannelHandler 中移动。

### 分布式

#### 1、分布式服务接口的幂等性如何设计？

所谓幂等性，就是说一个接口，多次发起同一个请求，你这个接口得保证结果是准确得。比如不能多扣款。不能多插入一条数据，不能将统计值多加了 1，这就是幂等性。

其实保证幂等性主要是三点：

- 对于每个请求必须有一个唯一的标识，举个例子：订单支付请求，肯定得包含订单 ID，一个订单 ID 最多支付一次。
- 每次处理完请求之后，必须有一个记录标识这个请求处理过了，比如说常见的方案是再 mysql 中记录个状态啥得，比如支付之前记录一条这个订单得支付流水，而且支付流水采用 order id 作为唯一键（unique key）。只有成功插入这个支付流水，才可以执行实际得支付扣款
- 每次接收请求需要进行判断之前是否处理过得逻辑处理，比如说，如果有一个订单已经支付了，就已经有了一条支付流水，那么如果重复发送这个请求，则此时先插入支付流水，order id 已经存在了，唯一键约束生效，报错插入不进去得。然后你就不用再扣款了。

#### 2、分布式系统中的接口调用如何保证顺序性？

可以接入 MQ，如果是系统 A 使用多线程处理的话，可以使用内存队列，来保证顺序性，如果你要 100% 的顺序性，当然可以使用分布式锁来搞，会影响系统的并发性。

#### 3、说说 ZooKeeper 一般都有哪些使用场景？

- 分布式协调：这个其实就是 zk 很经典的一个用法，简单来说，就好比，你系统 A 发送个请求到 mq，然后 B 消费了之后处理。那 A 系统如何指导 B 系统的处理结果？用 zk 就可以实现分布式系统之间的协调工作。A 系统发送请求之后可以在 zk 上对某个节点的值注册个监听器，一旦 B 系统处理完了就修改 zk 那个节点的值，A 立马就可以收到通知，完美解决。
- 分布锁：对某一个数据联系发出两个修改操作，两台机器同时收到请求，但是只能一台机器先执行另外一个机器再执行，那么此时就可以使用 zk 分布式锁，一个机器接收到了请求之后先获取 zk 上的一把分布式锁，就是可以去创建一个 znode，接着执行操作，

---

然后另外一个机器也尝试去创建那个 znode，结果发现自己创建不了，因为被别人创建了，那只能等着，等等一个机器执行完了自己再执行。

- 配置信息管理：zk 可以用作很多系统的配置信息的管理，比如 kafka, storm 等等很多分布式系统都会选用 zk 来做一些元数据，配置信息的管理，包括 dubbo 注册中心不也支持 zk 么。
- HA 高可用性：这个应该是很常见的，比如 hdfs, yarn 等很多大数据系统，都选择基于 zk 来开发 HA 高可用机制，就是一个重要进程一般会主备两个，主进程挂了立马通过 zk 感知到切换到备份进程。

#### 4、说说你们的分布式 session 方案是啥？怎么做的？

- Tomcat + redis

其实还挺方便的，就是使用 session 的代码跟以前一样，还是基于 tomcat 原生的 session 支持即可，然后就是用一个叫做 tomcat RedisSessionManager 的东西，让我们部署的 tomcat 都将 session 数据存储到 redis 即可。

- Spring Session + redis

分布式会话的这个东西重耦合在 tomcat，如果我要将 web 容器迁移成 jetty，不能重新把 jetty 都配置一遍。

所以现在比较好用的还是基于 java 的一站式解决方案，使用 spring session 是一个很好的选择，给 spring session 配置基于 redis 来存储 session 数据，然后配置一个 spring session 的过滤器，这样的话，session 相关操作都会交给 spring session 来管了。接着在代码中，就是用原生的 session 操作，就是直接基于 spring session 从 redis 中获取数据了。

#### 5、分布式事务了解吗？

- XA 方案/两阶段提交方案

第一个阶段（先询问）

第二个阶段（再执行）

- TCC 方案

TCC 的全程是：Try、Confirm、Cancel

这个其实是用到了补偿的概念，分为了三个阶段

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，就是执行已经成功的业务逻辑的回滚操作

- 本地消息表

- 可靠消息最终一致性方案

- 
- 最大努力通知方案

## 6、那常见的分布式锁有哪些解决方案？

- Reids 的分布式锁，很多大公司会基于 Reids 做扩展开发
- 基于 Zookeeper
- 基于数据库，比如 Mysql

## 7、ZK 和 Redis 的区别，各自有什么优缺点？

先说 Redis：

- Redis 只保证最终一致性，副本间的数据复制是异步进行（Set 是写，Get 是读，Reids 集群一般是读写分离架构，存在主从同步延迟情况），主从切换之后可能有部分数据没有复制过去可能会丢失锁情况，故强一致性要求的业务不推荐使用 Reids，推荐使用 zk。
- Redis 集群各方法的响应时间均为最低。随着并发量和业务数量的提升其响应时间会有明显上升（公有集群影响因素偏大），但是极限 qps 可以达到最大且基本无异常

再说 ZK：

- 使用 ZooKeeper 集群，锁原理是使用 ZooKeeper 的临时节点，临时节点的生命周期在 Client 与集群的 Session 结束时结束。因此如果某个 Client 节点存在网络问题，与 ZooKeeper 集群断开连接，Session 超时同样会导致锁被错误的释放（导致被其他线程错误地持有），因此 ZooKeeper 也无法保证完全一致。
- ZK 具有较好的稳定性；响应时间抖动很小，没有出现异常。但是随着并发量和业务数量的提升其响应时间和 qps 会明显下降。

## 8、MySQL 如何做分布式锁？

方法一：

利用 Mysql 的锁表，创建一张表，设置一个 UNIQUE KEY 这个 KEY 就是要锁的 KEY，所以同一个 KEY 在 mysql 表里只能插入一次了，这样对锁的竞争就交给了数据库，处理同一个 KEY 数据库保证了只有一个节点能插入成功，其他节点都会插入失败。

DB 分布式锁的实现：通过主键 id 的唯一性进行加锁，说白了就是加锁的形式是向一张表中插入一条数据，该条数据的 id 就是一把分布式锁，例如当一次请求插入了一条 id 为 1 的数据，其他想要进行插入数据的并发请求必须等第一次请求执行完成后删除这条 id 为 1 的数据才能继续插入，实现了分布式锁的功能。

方法二：

使用流水号+时间戳做幂等操作，可以看作是一个不会释放的锁。

## 9、你了解业界哪些大公司的分布式锁框架

---

- **Google:Chubby**

Chubby 是一套分布式协调系统，内部使用 Paxos 协调 Master 与 Replicas。

Chubby lock service 被应用在 GFS, BigTable 等项目中，其首要设计目标是高可靠性，而不是高性能。

Chubby 被作为粗粒度锁使用，例如被用于选主。持有锁的时间跨度一般为小时或天，而不是秒级。

Chubby 对外提供类似于文件系统的 API，在 Chubby 创建文件路径即加锁操作。

Chubby 使用 Delay 和 SequenceNumber 来优化锁机制。Delay 保证客户端异常释放锁时，Chubby 仍认为该客户端一直持有锁。Sequence number 指锁的持有者向 Chubby 服务端请求一个序号（包括几个属性），然后之后在需要使用锁的时候将该序号一并发给 Chubby 服务器，服务端检查序号的合法性，包括 number 是否有效等。

- **京东 SharkLock**

SharkLock 是基于 Redis 实现的分布式锁。锁的排他性由 SETNX 原语实现，使用 timeout 与续租机制实现锁的强制释放。

- **蚂蚁金服 SOFAJRaft-RheaKV 分布式锁**

RheaKV 是基于 SOFAJRaft 和 RocksDB 实现的嵌入式、分布式、高可用、强一致的 KV 存储类库。

RheaKV 对外提供 lock 接口，为了优化数据的读写，按不同的存储类型，提供不同的锁特性。RheaKV 提供 watchdog 调度器来控制锁的自动续租机制，避免锁在任务完成前前提释放，和锁永不释放造成死锁。

- **Netflix: Curator**

Curator 是 ZooKeeper 的客户端封装，其分布式锁的实现完全由 ZooKeeper 完成。

在 ZooKeeper 创建 EPHEMERAL\_SEQUENTIAL 节点视为加锁，节点的 EPHEMERAL 特性保证了锁持有者与 ZooKeeper 断开时强制释放锁；节点的 SEQUENTIAL 特性避免了加锁较多时的惊群效应。

## 10、请讲一下你对 CAP 理论的理解

在理论计算机科学中，CAP 定理（CAP theorem），又被称作布鲁尔定理（Brewer's theorem），它指出对于一个分布式计算系统来说，不可能同时满足以下三点：

- Consistency（一致性） 指数据在多个副本之间能够保持一致的特性（严格的一致性）
- Availability（可用性） 指系统提供的服务必须一直处于可用的状态，每次请求都能获取到非错的响应（不保证获取的数据为最新数据）
- Partition tolerance（分区容错性） 分布式系统在遇到任何网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务，除非整个网络环境都发生了故障

Spring Cloud 在 CAP 法则上主要满足的是 A 和 P 法则，Dubbo 和 Zookeeper 在 CAP 法则主要满足的是 C 和 P 法则。

---

CAP 仅适用于原子读写的 NOSQL 场景中，并不适合数据库系统。现在的分布式系统具有更多特性比如扩展性、可用性等等，在进行系统设计和开发时，我们不应该仅仅局限在 CAP 问题上。

现实生活中，大部分人解释这一定律时，常常简单的表述为：“一致性、可用性、分区容忍性三者你只能同时达到其中两个，不可能同时达到”。实际上这是一个非常具有误导性质的说法，而且在 CAP 理论诞生 12 年之后，CAP 之父也在 2012 年重写了之前的论文。

当发生网络分区的时候，如果我们要继续服务，那么强一致性和可用性只能 2 选 1。也就是说当网络分区之后 P 是前提，决定了 P 之后才有 C 和 A 的选择。也就是说分区容错性（Partition tolerance）我们是必须要实现的。

## 11、请讲一下你对 BASE 理论的理解

BASE 理论由 eBay 架构师 Dan Pritchett 提出，在 2008 年上被发表为论文，并且 eBay 给出了他们在实践中总结的基于 BASE 理论的一套新的分布式事务解决方案。

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。BASE 理论是对 CAP 中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的总结，是基于 CAP 定理逐步演化而来的，它大大降低了我们对系统的要求。

BASE 理论的核心思想是即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性。也就是牺牲数据的一致性来满足系统的高可用性，系统中一部分数据不可用或者不一致时，仍需要保持系统整体“主要可用”。

针对数据库领域，BASE 思想的主要实现是对业务数据进行拆分，让不同的数据分布在不同的机器上，以提升系统的可用性，当前主要有以下两种做法：

- 按功能划分数据库
- 分片（如开源的 Mycat、Amoeba 等）。

## 12、分布式与集群的区别是什么？

分布式：一个业务分拆多个子业务，部署在不同的服务器上

集群：同一个业务，部署在多个服务器上。比如之前做电商网站搭的 redis 集群以及 solr 集群都是属于将 redis 服务器提供的缓存服务以及 solr 服务器提供的搜索服务部署在多个服务器上以提高系统性能、并发量解决海量存储问题。

## 13、请讲一下 BASE 理论的三要素

### 基本可用

基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性。但是，这绝不等价于系统不可用。

比如：

- 
- 响应时间上的损失：正常情况下，一个在线搜索引擎需要在 0.5 秒之内返回给用户相应的查询结果，但由于出现故障，查询结果的响应时间增加了 1~2 秒
  - 系统功能上的损失：正常情况下，在一个电子商务网站上进行购物的时候，消费者几乎能够顺利完成每一笔订单，但是在一些节日大促购物高峰的时候，由于消费者的购物行为激增，为了保护购物系统的稳定性，部分消费者可能会被引导到一个降级页面

## 软状态

软状态指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时。

## 最终一致性

强调的是系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

## 14、请说一下对两阶段提交协议的理解

分布式系统的一个难点是如何保证架构下多个节点在进行事务性操作的时候保持一致性。为实现这个目的，二阶段提交算法成立基于以下假设：

- 该分布式系统中，存在一个节点作为协调者(Coordinator)，其他节点作为参与者(Cohorts)。且节点之间可以进行网络通信。
- 所有节点都采用预写式日志，且日志被写入后即被保持在可靠的存储设备上，即使节点损坏不会导致日志数据的消失。
- 所有节点不会永久性损坏，即使损坏后仍然可以恢复。

### 第一阶段（投票阶段）

- 协调者节点向所有参与者节点询问是否可以执行提交操作(vote)，并开始等待各参与者节点的响应。
- 参与者节点执行询问发起为止的所有事务操作，并将 Undo 信息和 Redo 信息写入日志。  
(注意：若成功这里其实每个参与者已经执行了事务操作)
- 各参与者节点响应协调者节点发起的询问。如果参与者节点的事务操作实际执行成功，则它返回一个“同意”消息；如果参与者节点的事务操作实际执行失败，则它返回一个“中止”消息。

### 第二阶段（提交执行阶段）

当协调者节点从所有参与者节点获得的相应消息都为“同意”：

- 协调者节点向所有参与者节点发出“正式提交(commit)”的请求。
- 参与者节点正式完成操作，并释放在整个事务期间内占用的资源。
- 参与者节点向协调者节点发送“完成”消息。
- 协调者节点收到所有参与者节点反馈的“完成”消息后，完成事务。

如果任一参与者节点在第一阶段返回的响应消息为“中止”：

- 协调者节点向所有参与者节点发出“回滚操作(rollback)”的请求。
- 参与者节点利用之前写入的 Undo 信息执行回滚，并释放在整个事务期间内占用的资源。

- 
- 参与者节点向协调者节点发送“回滚完成”消息。
  - 协调者节点受到所有参与者节点反馈的“回滚完成”消息后，取消事务。

## 15、请讲一下对 TCC 协议的理解

Try Confirm Cancel

- Try：尝试待执行的业务，这个过程并未执行业务，只是完成所有业务的一致性检查，并预留好执行所需的全部资源。
- Confirm：执行业务，这个过程真正开始执行业务，由于 Try 阶段已经完成了一致性检查，因此本过程直接执行，而不做任何检查。并且在执行的过程中，会使用到 Try 阶段预留的业务资源。
- Cancel：取消执行的业务，若业务执行失败，则进入 Cancel 阶段，它会释放所有占用的业务资源，并回滚 Confirm 阶段执行的操作。

## 微服务

### 1、你对微服务是怎么理解的？

- 微服务，又名微服务架构，是一种架构风格，它将应用构建为一个小型自治服务的集合，**以业务领域为模型**。
- 通俗地说，就像蜜蜂通过对蜡制的等边六角形单元来构建它们的蜂巢。
- 他们最初从使用各种材料的小单元开始，一点点的搭建出一个大型蜂巢。
- 这些小单元组成坚固的结构，将蜂窝的特定部分固定在一起。
- 这里，每个小单元都独立于另一个，但它也与其他小单元相关。
- 这意味着对一个小单元的损害不会损害其他的单元，因此，蜜蜂可以在不影响完整蜂巢的情况下重建这些单元。

### 2、说说微服务架构的优势。

- 独立开发
- 独立部署
- 故障隔离
- 混合技术栈
- 粒度缩放

### 3、微服务有哪些特点？

- **解耦（Decoupling）** - 系统内的服务很大程度上是分离的。因此整个应用可以被轻松构建、修改和扩展
- **组件化（Componentization）** - 微服务被视为可以被轻松替换和升级的独立组件
- **业务能力（Business Capabilities）** - 微服务非常简单，专注于单一功能

- 
- **自治 (Autonomy)** - 开发人员和团队可以相互独立工作，从而提高效率
  - **持续交付 (ContinuousDelivery)** - 允许频繁发版，通过系统自动化完成对软件的创建、测试和审核，
  - **责任 (Responsibility)** - 微服务不把程序作为项目去关注。相反，他们将程序视为自己负责的产品
  - **分散治理 (Decentralized Governance)** - 重点是用正确的工具去做正确的事。这意味着没有任何标准化模式或着技术模式。开发人员可以自由选择最合适的工具来解决自己的问题
  - **敏捷性 (Agility)** - 微服务支持敏捷开发。任何新功能都可以快速开发并被再次丢弃

#### 4、单体应用、SOA 和微服务架构有什么区别？

- **单体应用**类似于一个大容器，其中程序的所有组件都被组装在一起并紧密包装。
- **SOA**是一组相互通信的服务。通信可以涉及简单的数据传送，也可以涉及两个或多个协调某些活动的服务。
- **微服务架构**是一种架构风格，它将应用程序构建为以业务域为模型的小型自治服务集合。

#### 5、在使用微服务架构时，你面临的挑战是什么？

- **自动化组件**: 难以自动化，因为有许多较小的组件。对于每个组件，都必须采取构建、发布和监控的步骤。
- **可感知性**: 将大量组件维持在一起会带来难以部署、维护、监控和识别的问题。它需要在所有组件周围具有很好的感知能力。
- **配置管理**: 有时在各种环境中维护组件的配置会很困难。
- **调试**: 很难找到与产生的错误相关的每一项服务。维护一个集中式的日志和控制面板对调试问题至关重要。

#### 6、什么是 Spring Boot?

多年来，随着新功能的增加，spring 变得越来越复杂。访问 spring 官网页面，我们就会看到可以在我们的应用程序中使用的所有 Spring 项目的不同功能。如果必须启动一个新的 Spring 项目，我们必须添加构建路径或添加 Maven 依赖关系，配置应用程序服务器，添加 spring 配置。因此，开始一个新的 spring 项目需要很多努力，因为我们现在必须从头开始做所有事情。

Spring Boot 是解决这个问题的方法。

Spring Boot 已经建立在现有 spring 框架之上。使用 spring 启动，我们避免了之前我们必须做的所有样板代码和配置。因此，Spring Boot 可以帮助我们以最少的工作量，更加健壮地使用现有的 Spring 功能。

#### 7、Spring Boot 有哪些优点？

---

Spring Boot 的优点有：

- 减少开发，测试时间和努力。
- 使用 JavaConfig 有助于避免使用 XML。
- 避免大量的 Maven 导入和各种版本冲突。
- 提供意见发展方法。
- 通过提供默认值快速开始开发。
- 没有单独的 Web 服务器需要。这意味着你不再需要启动 Tomcat, Glassfish 或其他任何东西。
- 需要更少的配置 因为没有 web.xml 文件。只需添加用 @Configuration 注释的类，然后添加用 @Bean 注释的方法，Spring 将自动加载对象并像以前一样对其进行管理。您甚至可以将 @Autowired 添加到 bean 方法中，以使 Spring 自动装入需要的依赖关系中。
- 基于环境的配置 使用这些属性，您可以将您正在使用的环境传递到应用程序： -  
Dspring.profiles.active = {enviornment}。在加载主应用程序属性文件后，Spring 将在 (application{environment}.properties) 中加载后续的应用程序属性文件。

## 8、什么是 JavaConfig？

Spring JavaConfig 是 Spring 社区的产品，它提供了配置 Spring IoC 容器的纯 Java 方法。因此它有助于避免使用 XML 配置。使用 JavaConfig 的优点在于：

- 面向对象的配置。由于配置被定义为 JavaConfig 中的类，因此用户可以充分利用 Java 中的面向对象功能。一个配置类可以继承另一个，重写它的@Bean 方法等。
- 减少或消除 XML 配置。基于依赖注入原则的外化配置的好处已被证明。但是，许多开发人员不希望在 XML 和 Java 之间来回切换。JavaConfig 为开发人员提供了一种纯 Java 方法来配置与 XML 配置概念相似的 Spring 容器。从技术角度来讲，只使用 JavaConfig 配置类来配置容器是可行的，但实际上很多人认为将 JavaConfig 与 XML 混合匹配是理想的。
- 类型安全和重构友好。JavaConfig 提供了一种类型安全的方法来配置 Spring 容器。由于 Java 5.0 对泛型的支持，现在可以按类型而不是按名称检索 bean，不需要任何强制转换或基于字符串的查找。

## 9、什么是 Spring Cloud？

Spring cloud 流应用程序启动器是基于 Spring Boot 的 Spring 集成应用程序，提供与外部系统的集成。Spring cloud Task，一个生命周期短暂的微服务框架，用于快速构建执行有限数据处理的应用程序。

## 10、使用 Spring Boot 开发分布式微服务时，我们需要关注哪些问题？

- 与分布式系统相关的复杂性-这种开销包括网络问题，延迟开销，带宽问题，安全问题。

- 
- 服务发现-服务发现工具管理群集中的流程和服务如何查找和互相交谈。它涉及一个服务目录，在该目录中注册服务，然后能够查找并连接到该目录中的服务。
  - 冗余-分布式系统中的冗余问题。
  - 负载平衡 --负载平衡改善跨多个计算资源的工作负荷，诸如计算机，计算机集群，网络链路，中央处理单元，或磁盘驱动器的分布。
  - 性能-问题 由于各种运营开销导致的性能问题。
  - 部署复杂性-Devops 技能的要求。

## 11、服务注册和发现是什么意思？Spring Cloud 如何实现？

当我们开始一个项目时，我们通常在属性文件中进行所有的配置。随着越来越多的服务开发和部署，添加和修改这些属性变得更加复杂。有些服务可能会下降，而某些位置可能会发生变化。手动更改属性可能会产生问题。

Eureka 服务注册和发现可以在这种情况下提供帮助。由于所有服务都在 Eureka 服务器上注册并通过调用 Eureka 服务器完成查找，因此无需处理服务地点的任何更改和处理。

## 12、负载平衡的意义什么？

在计算中，负载平衡可以改善跨计算机，计算机集群，网络链接，中央处理单元或磁盘驱动器等多种计算资源的工作负载分布。负载平衡旨在优化资源使用，最大化吞吐量，最小化响应时间并避免任何单一资源的过载。使用多个组件进行负载平衡而不是单个组件可能会通过冗余来提高可靠性和可用性。负载平衡通常涉及专用软件或硬件，例如多层交换机或域名系统服务器进程。

## 13、什么是 Hystrix？

Hystrix 是一个延迟和容错库，旨在隔离远程系统，服务和第三方库的访问点，当出现故障是不可避免的故障时，停止级联故障并在复杂的分布式系统中实现弹性。

## 14、什么是 Netflix Feign？它的优点是什么？

Feign 是受到 Retrofit，JAXRS-2.0 和 WebSocket 启发的 java 客户端联编程序。

优点：Feign 的第一个目标是将约束分母的复杂性统一到 http apis，而不考虑其稳定性。

## 15、Spring Cloud 断路器的作用

当一个服务调用另一个服务由于网络原因或自身原因出现问题，调用者就会等待被调用者的响应 当更多的服务请求到这些资源导致更多的请求等待，发生连锁效应（雪崩效应）。

断路器有完全打开状态：一段时间内达到一定的次数无法调用，并且多次监测没有恢复的迹象，断路器完全打开，那么下次请求就不会请求到该服务。

半开：短时间内有恢复迹象，断路器会将部分请求发给该服务，正常调用时 断路器关闭。

---

关闭：当服务一直处于正常状态，能正常调用。

## 消息队列

### 1、为什么使用消息队列？

消息队列常见的使用场景其实有很多，但是比较核心的有 3 个：解耦、异步、削峰。

### 2、消息队列有什么优点和缺点？

优点就是在特殊场景下有其对应的好处，解耦、异步、削峰。

缺点有以下几个：

- 系统可用性降低

系统引入的外部依赖越多，越容易挂掉。本来你就是 A 系统调用 BCD 三个系统的接口就好了，人 ABCD 四个系统好好的，没啥问题，你偏加个 MQ 进来，万一 MQ 挂了咋整，MQ 一挂，整套系统崩溃的，你不就完了？

- 系统复杂度提高

硬生生加个 MQ 进来，你怎么保证消息没有重复消费？怎么处理消息丢失的情况？怎么保证消息传递的顺序性？头大头大，问题一大堆，痛苦不已。

- 一致性问题

A 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是 BCD 三个系统那里，BD 两个系统写库成功了，结果 C 系统写库失败了，咋整？你这数据就不一致了。

### 3、Kafka、ActiveMQ、RabbitMQ、RocketMQ 都有什么区别，以及适合哪些场景？

## Kafka、ActiveMQ、RabbitMQ、RocketMQ 有什么优缺点？

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
单机吞吐量	万级，比 RocketMQ、Kafka 低一个数量级	同 ActiveMQ	10 万级，支撑高吞吐	10 万级，高吞吐，一般配合大数据类的系统来进行实时数据计算、日志采集等场景
topic 数量对吞吐量的影响			topic 可以达到几百/几千的级别，吞吐量会有较小幅度的下降，这是 RocketMQ 的一大优势，在同等机器下，可以支撑大量的 topic	topic 从几十到几百个时候，吞吐量会大幅度下降，在同等机器下，Kafka 尽量保证 topic 数量不要过多，如果要支撑大规模的 topic，需要增加更多的机器资源
时效性	ms 级	微秒级，这是 RabbitMQ 的一大特点，延迟最低	ms 级	延迟在 ms 级以内
可用性	高，基于主从架构实现高可用	同 ActiveMQ	非常高，分布式架构	非常高，分布式，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用
消息可靠性	有较低的概率丢失数据	基本不丢	经过参数优化配置，可以做到 0 丢失	同 RocketMQ
功能支持	MQ 领域的功能极其完备	基于 erlang 开发，并发能力很强，性能极好，延时很低	MQ 功能较为完善，还是分布式的，扩展性好	功能较为简单，主要支持简单的 MQ 功能，在大数据领域的实时计算以及日志采集被大规模使用

## 4、如何保证消息不被重复消费？

首先，比如 RabbitMQ、RocketMQ、Kafka，都有可能会出现消息重复消费的问题，正常。因为这问题通常不是 MQ 自己保证的，是由我们开发来保证的。

Kafka 实际上有个 offset 的概念，就是每个消息写进去，都有一个 offset，代表消息的序号，然后 consumer 消费了数据之后，每隔一段时间（定时定期），会把自己消费过的消息的 offset 提交一下，表示“我已经消费过了，下次我要是重启啥的，你就让我继续从上次消费到的 offset 来继续消费吧”。

但是凡事总有意外，比如我们之前生产经常遇到的，就是你有时候重启系统，看你怎么重启了，如果碰到点着急的，直接 kill 进程了，再重启。这会导致 consumer 有些消息处理了，但是没来得及提交 offset，尴尬了。重启之后，少数消息会再次消费一次。

## 5、如何保证消息消费的幂等性？

其实重复消费不可怕，可怕的是你没考虑到重复消费之后，怎么保证幂等性。一条数据重复出现两次，数据库里就只有一条数据，这就保证了系统的幂等性。

---

幂等性，通俗点说，就一个数据，或者一个请求，给你重复来多次，你得确保对应的数据是不会改变的，不能出错。

其实还是得结合业务来思考：

- 比如你拿个数据要写库，你先根据主键查一下，如果这数据都有了，你就别插入了，`update` 一下好吧。
- 比如你是写 Redis，那没问题了，反正每次都是 `set`，天然幂等性。
- 比如你不是上面两个场景，那做的稍微复杂一点，你需要让生产者发送每条数据的时候，里面加一个全局唯一的 id，类似订单 id 之类的东西，然后你这里消费到了之后，先根据这个 id 去比如 Redis 里查一下，之前消费过吗？如果没有消费过，你就处理，然后这个 id 写 Redis。如果消费过了，那你就别处理了，保证别重复处理相同的消息即可。
- 比如基于数据库的唯一键来保证重复数据不会重复插入多条。因为有唯一键约束了，重复数据插入只会报错，不会导致数据库中出现脏数据。

## 6、如何保证消息的可靠性传输？（如何处理消息丢失的问题）

### RabbitMQ

生产者将数据发送到 RabbitMQ 的时候，可能数据就在半路给搞丢了，因为网络问题啥的，都有可能。

此时可以选择用 RabbitMQ 提供的事务功能，就是生产者发送数据之前开启 RabbitMQ 事务 `channel.txSelect`，然后发送消息，如果消息没有成功被 RabbitMQ 接收到，那么生产者会收到异常报错，此时就可以回滚事务 `channel.txRollback`，然后重试发送消息；如果收到了消息，那么可以提交事务 `channel.txCommit`。

### RabbitMQ 弄丢了数据

就是 RabbitMQ 自己弄丢了数据，这个你必须开启 RabbitMQ 的持久化，就是消息写入之后会持久化到磁盘，哪怕是 RabbitMQ 自己挂了，恢复之后会自动读取之前存储的数据，一般数据不会丢。除非极其罕见的是，RabbitMQ 还没持久化，自己就挂了，可能导致少量数据丢失，但是这个概率较小。

### 消费端弄丢了数据

RabbitMQ 如果丢失了数据，主要是因为你消费的时候，刚消费到，还没处理，结果进程挂了，比如重启了，那么就尴尬了，RabbitMQ 认为你都消费了，这数据就丢了。

---

这个时候得用 RabbitMQ 提供的 `ack` 机制，简单来说，就是你必须关闭 RabbitMQ 的自动 `ack`，可以通过一个 api 来调用就行，然后每次你自己代码里确保处理完的时候，再在程序里 `ack` 一把。这样的话，如果你还没处理完，不就没有 `ack` 了？那 RabbitMQ 就认为你还没处理完，这个时候 RabbitMQ 会把这个消费分配给别的 consumer 去处理，消息是不会丢的。

## Kafka

### 消费端弄丢了数据

唯一可能导致消费者弄丢数据的情况，就是说，你消费到了这个消息，然后消费者那边自动提交了 offset，让 Kafka 以为你已经消费好了这个消息，但其实你才刚准备处理这个消息，你还没处理，你自己就挂了，此时这条消息就丢咯。

### Kafka 弄丢了数据

Kafka 的 leader 机器宕机了，将 follower 切换为 leader 之后，就会发现说这个数据就丢了。

所以此时一般是要起起码设置如下 4 个参数：

- 给 topic 设置 `replication.factor` 参数：这个值必须大于 1，要求每个 partition 必须有至少 2 个副本。
- 在 Kafka 服务端设置 `min.insync.replicas` 参数：这个值必须大于 1，这是要求一个 leader 至少感知到有至少一个 follower 还跟自己保持联系，没掉队，这样才能确保 leader 挂了还有一个 follower 吧。
- 在 producer 端设置 `acks=all`：这个是要求每条数据，必须是写入所有 replica 之后，才能认为是写成功了。
- 在 producer 端设置 `retries=MAX`（很大很大的一个值，无限次重试的意思）：这个是要求一旦写入失败，就无限重试，卡在这里了。

## 7、如果让你写一个消息队列，该如何进行架构设计啊？说一下你的思路。

- 首先这个 mq 得支持可伸缩性吧，就是需要的时候快速扩容，就可以增加吞吐量和容量，那怎么搞？设计个分布式的系统呗，参照一下 kafka 的设计理念，broker -> topic -> partition，每个 partition 放一个机器，就存一部分数据。如果现在资源不够了，简单啊，

---

给 topic 增加 partition，然后做数据迁移，增加机器，不就可以存放更多数据，提供更高的吞吐量了？

- 其次你得考虑一下这个 mq 的数据要不要落地磁盘吧？那肯定要了，落磁盘才能保证别进程挂了数据就丢了。那落磁盘的时候怎么落啊？顺序写，这样就没有磁盘随机读写的寻址开销，磁盘顺序读写的性能是很高的，这就是 kafka 的思路。
- 其次你考虑一下你的 mq 的可用性啊？这个事儿，具体参考之前可用性那个环节讲解的 kafka 的高可用保障机制。多副本 -> leader & follower -> broker 挂了重新选举 leader 即可对外服务。

## 8、如何解决消息队列的延时以及过期失效问题？

### 大量消息在 mq 里积压了几个小时了还没解决

一个消费者一秒是 1000 条，一秒 3 个消费者是 3000 条，一分钟是 18 万条，1000 多万条，所以如果你积压了几百万到上千万的数据，即使消费者恢复了，也需要大概 1 小时的时间才能恢复过来。

一般这个时候，只能操作临时紧急扩容了，具体操作步骤和思路如下：

- 先修复 consumer 的问题，确保其恢复消费速度，然后将现有 consumer 都停掉。
- 新建一个 topic，partition 是原来的 10 倍，临时建立好原先 10 倍或者 20 倍的 queue 数量。
- 然后写一个临时的分发数据的 consumer 程序，这个程序部署上去消费积压的数据，消费之后不做耗时的处理，直接均匀轮询写入临时建立好的 10 倍数量的 queue。
- 接着临时征用 10 倍的机器来部署 consumer，每一批 consumer 消费一个临时 queue 的数据。

这种做法相当于是临时将 queue 资源和 consumer 资源扩大 10 倍，以正常的 10 倍速度来消费数据。等快速消费完积压数据之后，得恢复原先部署架构，重新用原先的 consumer 机器来消费消息。

### 消息队列过期失效问题

假设你用的是 rabbitmq，rabbitmq 是可以设置过期时间的，就是 TTL，如果消息在 queue 中积压超过一定的时间就会被 rabbitmq 给清理掉，这个数据就没了。那这就是第二个坑了。这就不是说数据会大量积压在 mq 里，而是大量的数据会直接搞丢。

这个情况下，就不是说要增加 consumer 消费积压的消息，因为实际上没啥积压，而是丢了大量的消息。我们可以采取一个方案，就是批量重导，

这个时候我们就开始写程序，将丢失的那批数据，写个临时程序，一点一点的查出来，然后重新灌入 mq 里面去，把白天丢的数据给他补回来。也只能是这样了。

## 9、消息队列满了以后该怎么处理？

如果走的方式是消息积压在 mq 里，那么如果你很长时间都没处理掉，此时导致 mq 都快写满了，咋办？这个还有别的办法吗？没有，谁让你第一个方案执行的太慢了，你临时写程序，接入数据来消费，消费一个丢弃一个，都不要了，快速消费掉所有的消息。然后走第二个方案，到了晚上再补数据吧。

## 10、消息如何分发？

若该队列至少有一个消费者订阅，消息将以循环（round-robin）的方式发送给消费者。每条消息只会分发给一个订阅的消费者（前提是消费者能够正常处理消息并进行确认）。通过路由可实现多消费的功能。

## 11、消息怎么路由？

消息提供方->路由->一至多个队列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。通过队列路由键，可以把队列绑定到交换器上。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）；

常用的交换器主要分为一下三种：

- fanout：如果交换器收到消息，将会广播到所有绑定的队列上
- direct：如果路由键完全匹配，消息就被投递到相应的队列
- topic：可以使来自不同源头的消息能够到达同一个队列。使用 topic 交换器时，可以使用通配符

## 12、消息基于什么传输？

由于 TCP 连接的创建和销毁开销较大，且并发数受系统资源限制，会造成性能瓶颈。RabbitMQ 使用信道的方式来传输数据。信道是建立在真实的 TCP 连接内的虚拟连接，且每条 TCP 连接上的信道数量没有限制。

## 13、为什么不应该对所有的 message 都使用持久化机制？

首先，必然导致性能的下降，因为写磁盘比写 RAM 慢的多，message 的吞吐量可能有 10 倍的差距。

其次，message 的持久化机制用在 RabbitMQ 的内置 cluster 方案时会出现“坑爹”问题。矛盾点在于，若 message 设置了 persistent 属性，但 queue 未设置 durable 属性，那么当该 queue 的 owner node 出现异常后，在未重建该 queue 前，发往该 queue 的 message 将被 blackholed；若 message 设置了 persistent 属性，同时 queue 也设置了 durable 属性，那么当 queue 的 owner node 异常且无法重启的情况下，则该 queue 无法

---

在其他 node 上重建，只能等待其 owner node 重启后，才能恢复该 queue 的使用，而在这段时间内发送给该 queue 的 message 将被 blackholed。

所以，是否要对 message 进行持久化，需要综合考虑性能需要，以及可能遇到的问题。若想达到 100,000 条/秒以上的消息吞吐量（单 RabbitMQ 服务器），则要么使用其他的方式来确保 message 的可靠 delivery，要么使用非常快速的存储系统以支持全持久化（例如使用 SSD）。另外一种处理原则是：仅对关键消息作持久化处理（根据业务重要程度），且应该保证关键消息的量不会导致性能瓶颈。

## 14、如何保证高可用的？RabbitMQ 的集群

RabbitMQ 是比较有代表性的，因为是基于主从（非分布式）做高可用性的，我们就以 RabbitMQ 为例子讲解第一种 MQ 的高可用性怎么实现。RabbitMQ 有三种模式：单机模式、普通集群模式、镜像集群模式。

单机模式，就是 Demo 级别的，一般就是你本地启动了玩玩儿的？，没人生产用单机模式  
普通集群模式，意思就是在多台机器上启动多个 RabbitMQ 实例，每个机器启动一个。  
你创建的 queue，只会放在一个 RabbitMQ 实例上，但是每个实例都同步 queue 的元数据  
(元数据可以认为是 queue 的一些配置信息，通过元数据，可以找到 queue 所在实例)。  
你消费的时候，实际上如果连接到了另外一个实例，那么那个实例会从 queue 所在实例上拉  
取数据过来。这方案主要是提高吞吐量的，就是说让集群中多个节点来服务某个 queue 的读  
写操作。

镜像集群模式：这种模式，才是所谓的 RabbitMQ 的高可用模式。跟普通集群模式不一样的是，在镜像集群模式下，你创建的 queue，无论元数据还是 queue 里的消息都会存在于多个实例上，就是说，每个 RabbitMQ 节点都有这个 queue 的一个完整镜像，包含 queue 的全部数据的意思。然后每次你写消息到 queue 的时候，都会自动把消息同步到多个实例的 queue 上。RabbitMQ 有很好的管理控制台，就是在后台新增一个策略，这个策略是镜像集群模式的策略，指定的时候是可以要求数据同步到所有节点的，也可以要求同步到指定数量的节点，再次创建 queue 的时候，应用这个策略，就会自动将数据同步到其他的节点上去了。这样的话，好处在于，你任何一个机器宕机了，没事儿，其它机器（节点）还包含了这个 queue 的完整数据，别的 consumer 都可以到其它节点上去消费数据。坏处在于，第一，这个性能开销也太大了吧，消息需要同步到所有机器上，导致网络带宽压力和消耗很重！  
RabbitMQ 一个 queue 的数据都是放在一个节点里的，镜像集群下，也是每个节点都放这个 queue 的完整数据。

## 15、RabbitMQ 的工作模式

- simple 模式（即最简单的收发模式）
- work 工作模式(资源的竞争)
- publish/subscribe 发布订阅(共享资源)
- routing 路由模式
- topic 主题模式

## 16、为什么需要消息系统，MySQL 不能满足需求吗？

- **解耦：**  
允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。
- **冗余：**  
消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。许多消息队列所采用的”插入-获取-删除”范式中，在把一个消息从队列中删除之前，需要你的处理系统明确的指出该消息已经被处理完毕，从而确保你的数据被安全的保存直到你使用完毕。
- **扩展性：**  
因为消息队列解耦了你的处理过程，所以增大消息入队和处理的频率是很容易的，只要另外增加处理过程即可。
- **灵活性 & 峰值处理能力：**  
在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见。如果为以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。
- **可恢复性：**  
系统的一部分组件失效时，不会影响到整个系统。消息队列降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。
- **顺序保证：**  
在大多使用场景下，数据处理的顺序都很重要。大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。（Kafka 保证一个 Partition 内的消息的有序性）
- **缓冲：**  
有助于控制和优化数据流经过系统的速度，解决生产消息和消费消息的处理速度不一致的情况。
- **异步通信：**  
很多时候，用户不想也需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

## Dubbo

### 1、说说核心的配置有哪些？

标签	用途	解释
<a href="#">dubbo:service/</a>	服务配置	用于暴露一个服务，定义服务的元信息，一个服务可以用多个协议暴露，一个服务也可以注册到多个注册中心

<a href="#">dubbo:reference/</a>	引用配置	用于创建一个远程服务代理，一个引用可以指向多个注册中心	
<a href="#">dubbo:protocol/</a>	协议配置	用于配置提供服务的协议信息，协议由提供方指定，消费方被动接受	
<a href="#">dubbo:application/</a>	应用配置	用于配置当前应用信息，不管该应用是提供者还是消费者	
<a href="#">dubbo:module/</a>	模块配置	用于配置当前模块信息，可选	
<a href="#">dubbo:registry/</a>	注册中心配置	用于配置连接注册中心相关信息	
<a href="#">dubbo:monitor/</a>	监控中心配置	用于配置连接监控中心相关信息，可选	
<a href="#">dubbo:provider/</a>	提供方配置	当 ProtocolConfig 和 ServiceConfig 某属性没有配置时，采用此缺省值，可选	
<a href="#">dubbo:consumer/</a>	消费方配置	当 ReferenceConfig 某属性没有配置时，采用此缺省值，可选	
<a href="#">dubbo:method/</a>	方法配置	用于 ServiceConfig 和 ReferenceConfig 指定方法级的配置信息	
<a href="#">dubbo:argument</a>	参数配置	用于指定方法参数配置	

## 2、Dubbo 支持哪些协议，每种协议的应用场景，优缺点？

- dubbo：单一长连接和 NIO 异步通讯，适合大并发小数据量的服务调用，以及消费者远大于提供者。传输协议 TCP，异步，Hessian 序列化；
- rmi：采用 JDK 标准的 rmi 协议实现，传输参数和返回参数对象需要实现 Serializable 接口，使用 java 标准序列化机制，使用阻塞式短连接，传输数据包大小混合，消费者和提供者个数差不多，可传文件，传输协议 TCP。多个短连接，TCP 协议传输，同步传输，适用常规的远程服务调用和 rmi 互操作。在依赖低版本的 Common-Collections 包，java 序列化存在安全漏洞；
- http：基于 Http 表单提交的远程调用协议，使用 Spring 的 HttpInvoke 实现。多个短连接，传输协议 HTTP，传入参数大小混合，提供者个数多于消费者，需要给应用程序和浏览器 JS 调用；
- webservice：基于 WebService 的远程调用协议，集成 CXF 实现，提供和原生 WebService 的互操作。多个短连接，基于 HTTP 传输，同步传输，适用系统集成和跨语言调用；

- 
- hessian: 集成 Hessian 服务，基于 HTTP 通讯，采用 Servlet 暴露服务，Dubbo 内嵌 Jetty 作为服务器时默认实现，提供与 Hessian 服务互操作。多个短连接，同步 HTTP 传输，Hessian 序列化，传入参数较大，提供者大于消费者，提供者压力较大，可传文件；
  - Redis: 基于 Redis 实现的 RPC 协议

### 3、服务提供者能实现失效踢出是什么原理？

服务失效踢出基于 zookeeper 的临时节点原理。

### 4、RPC 架构有哪些组件？

一个基本的 RPC 架构里面应该至少包含以下 4 个组件：

- 客户端（Client）：服务调用方（服务消费者）；
- 客户端存根（Client Stub）：存放服务端地址信息，将客户端的请求参数数据信息打包成网络消息，再通过网络传输发送给服务端；
- 服务端存根（Server Stub）：接收客户端发送过来的请求消息并进行解包，然后再调用本地服务进行处理；
- 服务端（Server）：服务的真正提供者。

#### 具体调用过程

- 服务消费者（client 客户端）通过调用本地服务的方式调用需要消费的服务；
- 客户端存根（client stub）接收到调用请求后负责将方法、入参等信息序列化（组装）成能够进行网络传输的消息体；
- 客户端存根（client stub）找到远程的服务地址，并且将消息通过网络发送给服务端；
- 服务端存根（server stub）收到消息后进行解码（反序列化操作）；
- 服务端存根（server stub）根据解码结果调用本地的服务进行相关处理；
- 本地服务执行具体业务逻辑并将处理结果返回给服务端存根（server stub）；
- 服务端存根（server stub）将返回结果重新打包成消息（序列化）并通过网络发送至消费方；
- 客户端存根（client stub）接收到消息，并进行解码（反序列化）；
- 服务消费方得到最终结果。

### 5、Dubbo 服务调用是阻塞的吗？

默认是阻塞的，可以异步调用，没有返回值的可以这么做。

### 6、Dubbo 核心功能有哪些？

- Remoting：网络通信框架，提供对多种 NIO 框架抽象封装，包括“同步转异步”和“请求-响应”模式的信息交换方式。

- 
- Cluster：服务框架，提供基于接口方法的透明远程过程调用，包括多协议支持，以及软负载均衡，失败容错，地址路由，动态配置等集群支持。
  - Registry：服务注册，基于注册中心目录服务，使服务消费方能动态的查找服务提供方，使地址透明，使服务提供方可以平滑增加或减少机器。

## 7、Dubbo 服务器注册与发现的流程？

- 服务容器 Container 负责启动，加载，运行服务提供者。
- 服务提供者 Provider 在启动时，向注册中心注册自己提供的服务。
- 服务消费者 Consumer 在启动时，向注册中心订阅自己所需的服务。
- 注册中心 Registry 返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
- 服务消费者 Consumer，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。
- 服务消费者 Consumer 和提供者 Provider，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心 Monitor。

## 8、Dubbo Monitor 实现原理？

Consumer 端在发起调用之前会先走 filter 链；provider 端在接收到请求时也是先走 filter 链，然后才进行真正的业务逻辑处理。默认情况下，在 consumer 和 provider 的 filter 链中都会有 Monitorfilter。

- MonitorFilter 向 DubboMonitor 发送数据
- DubboMonitor 将数据进行聚合后（默认聚合 1min 中的统计数据）暂存到 ConcurrentMap<Statistics, AtomicReference> statisticsMap，然后使用一个含有 3 个线程（线程名字：DubboMonitorSendTimer）的线程池每隔 1min 钟，调用 SimpleMonitorService 遍历发送 statisticsMap 中的统计数据，每发送完毕一个，就重置当前的 Statistics 的 AtomicReference
- SimpleMonitorService 将这些聚合数据塞入 BlockingQueue queue 中（队列大写为 100000）
- SimpleMonitorService 使用一个后台线程（线程名为：DubboMonitorAsyncWriteLogThread）将 queue 中的数据写入文件（该线程以死循环的形式来写）
- SimpleMonitorService 还会使用一个含有 1 个线程（线程名字：DubboMonitorTimer）的线程池每隔 5min 钟，将文件中的统计数据画成图表

## 9、Dubbo 和 Spring Cloud 有什么关系？

Dubbo 是 SOA 时代的产物，它的关注点主要在于服务的调用，流量分发、流量监控和熔断。而 Spring Cloud 诞生于微服务架构时代，考虑的是微服务治理的方方面面，另外由于依

---

托了 Spring、Spring Boot 的优势之上，两个框架在开始目标就不一致，Dubbo 定位服务治理、Spring Cloud 是打造一个生态。

## 10、Dubbo 和 Spring Cloud 有什么哪些区别？

- Dubbo 底层是使用 Netty 这样的 NIO 框架，是基于 TCP 协议传输的，配合以 Hession 序列化完成 RPC 通信。
- Spring Cloud 是基于 Http 协议 Rest 接口调用远程过程的通信，相对来说 Http 请求会有更大的报文，占的带宽也会更多。但是 REST 相比 RPC 更为灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖，这在强调快速演化的微服务环境下，显得更为合适，至于注重通信速度还是方便灵活性，具体情况具体考虑。

## 11、Dubbo 有哪些注册中心？

- Multicast 注册中心：Multicast 注册中心不需要任何中心节点，只要广播地址，就能进行服务注册和发现，基于网络中组播传输实现。
- Zookeeper 注册中心：基于分布式协调系统 Zookeeper 实现，采用 Zookeeper 的 watch 机制实现数据变更。
- Redis 注册中心：基于 Redis 实现，采用 key/map 存储，key 存储服务名和类型，map 中 key 存储服务 url，value 服务过期时间。基于 Redis 的发布/订阅模式通知数据变更。
- Simple 注册中心。
- 推荐使用 Zookeeper 作为注册中心

## 12、Dubbo 的注册中心集群挂掉，发布者和订阅者之间还能通信么？

可以通讯。启动 Dubbo 时，消费者会从 Zookeeper 拉取注册的生产者的地址接口等数据，缓存在本地。每次调用时，按照本地存储的地址进行调用。

## 13、Dubbo 集群提供了哪些负载均衡策略？

- Random LoadBalance: 随机选取提供者策略，有利于动态调整提供者权重。截面碰撞率高，调用次数越多，分布越均匀。
- RoundRobin LoadBalance: 轮循选取提供者策略，平均分布，但是存在请求累积的问题。
- LeastActive LoadBalance: 最少活跃调用策略，解决慢提供者接收更少的请求。
- ConstantHash LoadBalance: 一致性 Hash 策略，使相同参数请求总是发到同一提供者，一台机器宕机，可以基于虚拟节点，分摊至其他提供者，避免引起提供者的剧烈变动。

## 14、Dubbo 的集群容错方案有哪些？

- Failover Cluster: 失败自动切换，当出现失败，重试其它服务器。通常用于读操作，但重试会带来更长延迟。

- 
- Failfast Cluster: 快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。
  - Failsafe Cluster: 失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。
  - Fallback Cluster: 失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。
  - Forking Cluster: 并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks="2" 来设置最大并行数。
  - Broadcast Cluster: 广播调用所有提供者，逐个调用，任意一台报错则报错。通常用于通知所有提供者更新缓存或日志等本地资源信息。

## 15、Dubbo 超时设置有哪些方式？

Dubbo 超时设置有两种方式：

- 服务提供者端设置超时时间，在 Dubbo 的用户文档中，推荐如果能在服务端多配置就尽量多配置，因为服务提供者比消费者更清楚自己提供的服务特性。
- 服务消费者端设置超时时间，如果在消费者端设置了超时时间，以消费者端为主，即优先级更高。因为服务调用方设置超时时间控制性更灵活。如果消费方超时，服务端线程不会定制，会产生警告

## Java 集合

### 1、说说 List, Set, Queue, Map 四者的区别？

- List(对付顺序的好帮手): 存储的元素是有序的、可重复的。
- Set(注重独一无二的性质): 存储的元素是无序的、不可重复的。
- Queue(实现排队功能的叫号机): 按特定的排队规则来确定先后顺序，存储的元素是有序的、可重复的。
- Map(用 key 来搜索的专家): 使用键值对 (key-value) 存储，类似于数学上的函数  $y=f(x)$ ，"x" 代表 key，"y" 代表 value，key 是无序的、不可重复的，value 是无序的、可重复的，每个键最多映射到一个值

### 2、如何选用集合？

主要根据集合的特点来选用，比如我们需要根据键值获取到元素值时就选用 Map 接口下的集合，需要排序时选择 TreeMap，不需要排序时就选择 HashMap，需要保证线程安全就选用 ConcurrentHashMap。

当我们只需要存放元素值时，就选择实现 Collection 接口的集合，需要保证元素唯一时选择实现 Set 接口的集合比如 TreeSet 或 HashSet，不需要就选择实现 List 接口的比如 ArrayList 或 LinkedList，然后再根据实现这些接口的集合的特点来选用。

### 3、为什么要使用集合？

当我们需要保存一组类型相同的数据的时候，我们应该是用一个容器来保存，这个容器就是数组，但是，使用数组存储对象具有一定的弊端，因为我们在实际开发中，存储的数据的类型是多种多样的，于是，就出现了“集合”，集合同样也是用来存储多个数据的。

数组的缺点是一旦声明之后，长度就不可变了；同时，声明数组时的数据类型也决定了该数组存储的数据的类型；而且，数组存储的数据是有序的、可重复的，特点单一。但是集合提高了数据存储的灵活性，Java 集合不仅可以用来存储不同类型不同数量的对象，还可以保存具有映射关系的数据。

### 4、ArrayList 和 Vector 的区别？

- ArrayList 是 List 的主要实现类，底层使用 Object[] 存储，适用于频繁的查找工作，线程不安全；
- Vector 是 List 的古老实现类，底层使用 Object[] 存储，线程安全的。

### 5、ArrayList 与 LinkedList 区别？

- 是否保证线程安全：ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；
- 底层数据结构：ArrayList 底层使用的是 Object 数组；LinkedList 底层使用的是双向链表 数据结构（JDK1.6 之前为循环链表，JDK1.7 取消了循环。注意双向链表和双向循环链表的区别，下面有介绍到！）
- 插入和删除是否受元素位置的影响：
  - a. ArrayList 采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。比如：执行 add(E e) 方法的时候，ArrayList 会默认在将指定的元素追加到此列表的末尾，这种情况时间复杂度就是 O(1)。但是如果要在指定位置 i 插入和删除元素的话（add(int index, E element））时间复杂度就为 O(n-i)。因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的(n-i)个元素都要执行向后位/向前移一位的操作。
  - b. LinkedList 采用链表存储，所以，如果是在头尾插入或者删除元素不受元素位置的影响（add(E e)、addFirst(E e)、addLast(E e)、removeFirst()、removeLast()），近似 O(1)，如果是要在指定位置 i 插入和删除元素的话（add(int index, E element），remove(Object o））时间复杂度近似为 O(n)，因为需要先移动到指定位置再插入。
- 是否支持快速随机访问：LinkedList 不支持高效的随机元素访问，而 ArrayList 支持。快速随机访问就是通过元素的序号快速获取元素对象(对应于 get(int index)方法)。
- 内存空间占用：ArrayList 的空间浪费主要体现在在 list 列表的结尾会预留一定的容量空间，而 LinkedList 的空间花费则体现在它的每一个元素都需要消耗比 ArrayList 更多的空间（因为要存放直接后继和直接前驱以及数据）。

### 6、Comparable 和 Comparator 的区别？

- comparable 接口实际上是出自 java.lang 包 它有一个 compareTo(Object obj)方法用来排序
  - comparator 接口实际上是出自 java.util 包它有一个 compare(Object obj1, Object obj2)方法用来排序
- 一般我们需要对一个集合使用自定义排序时，我们就要重写 compareTo()方法或 compare()方法，当我们需要对某一个集合实现两种排序方式，比如一个 song 对象中的歌名和歌手名分别采用一种排序方法的话，我们可以重写 compareTo()方法和使用自制的 Comparator 方法或者以两个 Comparator 来实现歌名排序和歌星名排序，第二种代表我们只能使用两个参数版的 Collections.sort().

## 7、无序性和不可重复性的含义是什么

- 什么是无序性？无序性不等于随机性，无序性是指存储的数据在底层数组中并非按照数组索引的顺序添加，而是根据数据的哈希值决定的。
- 什么是不可重复性？不可重复性是指添加的元素按照 equals()判断时，返回 false，需要同时重写 equals()方法和 hashCode()方法。

## 8、比较 HashSet、LinkedHashSet 和 TreeSet 三者的异同

- 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，元素是有序的，排序的方式有自然排序和定制排序。

## 9、试比较 Queue 与 Deque 的区别

Queue 是单端队列，只能从一端插入元素，另一端删除元素，实现上一般遵循 先进先出 (FIFO) 规则。

Queue 扩展了 Collection 的接口，根据因为容量问题而导致操作失败后处理方式的不同可以分为两类方法：一种在操作失败后会抛出异常，另一种则会返回特殊值。

Queue 接口	抛出异常	返回特殊值
插入队尾	add(E e)	offer(E e)
删除队首	remove()	poll()
查询队首元素	element()	peek()

Deque 是双端队列，在队列的两端均可以插入或删除元素。

Deque 扩展了 Queue 的接口，增加了在队首和队尾进行插入和删除的方法，同样根据失败后处理方式的不同分为两类：

Deque 接口	抛出异常	返回特殊值
----------	------	-------

插入队首	addFirst(E e)	offerFirst(E e)
插入队尾	addLast(E e)	offerLast(E e)
删除队首	removeFirst()	pollFirst()
删除队尾	removeLast()	pollLast()
查询队首元素	getFirst()	peekFirst()
查询队尾元素	getLast()	peekLast()

事实上，Deque 还提供有 push() 和 pop() 等其他方法，可用于模拟栈。

## 10、请谈一下对 PriorityQueue 的认识？

PriorityQueue 是在 JDK1.5 中被引入的，其与 Queue 的区别在于元素出队顺序是与优先级相关的，即总是优先级最高的元素先出队。

这里列举其相关的一些要点：

- PriorityQueue 利用了二叉堆的数据结构来实现的，底层使用可变长的数组来存储数据
- PriorityQueue 通过堆元素的上浮和下沉，实现了在  $O(\log n)$  的时间复杂度内插入元素和删除堆顶元素。
- PriorityQueue 是非线程安全的，且不支持存储 NULL 和 non-comparable 的对象。
- PriorityQueue 默认是小顶堆，但可以接收一个 Comparator 作为构造参数，从而来自定义元素优先级的先后。

## 11、HashMap 和 Hashtable 的区别？

- 线程是否安全：HashMap 是非线程安全的，Hashtable 是线程安全的，因为 Hashtable 内部的方法基本都经过 synchronized 修饰。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；
- 效率：因为线程安全的问题，HashMap 要比 Hashtable 效率高一点。另外，Hashtable 基本被淘汰，不要在代码中使用它；
- 对 Null key 和 Null value 的支持：HashMap 可以存储 null 的 key 和 value，但 null 作为键只能有一个，null 作为值可以有多个；Hashtable 不允许有 null 键和 null 值，否则会抛出 NullPointerException。
- 初始容量大小和每次扩充容量大小的不同：① 创建时如果不指定容量初始值，Hashtable 默认的初始大小为 11，之后每次扩充，容量变为原来的  $2n+1$ 。HashMap 默认的初始化大小为 16。之后每次扩充，容量变为原来的 2 倍。② 创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为 2 的幂次方大小（HashMap 中的 tableSizeFor() 方法保证，下面给出了源代码）。也就是说 HashMap 总是使用 2 的幂作为哈希表的大小，后面会介绍到为什么是 2 的幂次方。

- 
- 底层数据结构：JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）（将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制。

## 12、HashSet 如何检查重复？

当你把对象加入 HashSet 时，HashSet 会先计算对象的 hashCode 值来判断对象加入的位置，同时也会与其他加入的对象的 hashCode 值作比较，如果没有相符的 hashCode，HashSet 会假设对象没有重复出现。但是如果发现有相同 hashCode 值的对象，这时会调用 equals() 方法来检查 hashCode 相等的对象是否真的相同。如果两者相同，HashSet 就不会让加入操作成功。

在 openjdk8 中，HashSet 的 add() 方法只是简单的调用了 HashMap 的 put() 方法，并且判断了一下返回值以确保是否有重复元素。也就是说，在 openjdk8 中，实际上无论 HashSet 中是否已经存在了某元素，HashSet 都会直接插入，只是会在 add() 方法的返回值处告诉我们插入前是否存在相同元素。

## 13、HashMap 的长度为什么是 2 的幂次方？

为了能让 HashMap 存取高效，尽量较少碰撞，也就是要尽量把数据分配均匀。我们上面也讲到了过了，Hash 值的范围值 -2147483648 到 2147483647，前后加起来大概 40 亿的映射空间，只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个 40 亿长度的数组，内存是放不下的。所以这个散列值是不能直接拿来用的。用之前还要先做对数组的长度取模运算，得到的余数才能用来要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“ $(n - 1) \& hash$ ”。（ $n$  代表数组长度）。这也解释了 HashMap 的长度为什么是 2 的幂次方。

### 这个算法应该如何设计呢？

我们首先可能会想到采用 % 取余的操作来实现。但是，重点来了：“取余(%)操作中如果除数是 2 的幂次则等价于与其除数减一的与(&)操作（也就是说  $hash \% length == hash \& (length - 1)$  的前提是 length 是 2 的 n 次方；）。”并且采用二进制位操作 &，相对于 % 能够提高运算效率，这就解释了 HashMap 的长度为什么是 2 的幂次方。

## 14、ConcurrentHashMap 和 Hashtable 的区别？

ConcurrentHashMap 和 Hashtable 的区别主要体现在实现线程安全的方式上不同。

- 底层数据结构：JDK1.7 的 ConcurrentHashMap 底层采用分段的数组+链表实现，JDK1.8 采用的数据结构跟 HashMap1.8 的结构一样，数组+链表/红黑二叉树。

- 
- Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 数组+链表 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；
- 实现线程安全的方式（重要）：① 在 JDK1.7 的时候，ConcurrentHashMap（分段锁）对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。到了 JDK1.8 的时候已经摒弃了 Segment 的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，并发控制使用 synchronized 和 CAS 来操作。（JDK1.6 以后对 synchronized 锁做了很多优化）整个看起来就像是优化过且线程安全的 HashMap，虽然在 JDK1.8 中还能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本；② Hashtable(同一把锁) : 使用 synchronized 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。

## 15、ConcurrentHashMap 线程安全的具体实现方式是怎样的？

### JDK1.7

首先将数据分为一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据时，其他段的数据也能被其他线程访问。

ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成。  
Segment 实现了 ReentrantLock, 所以 Segment 是一种可重入锁，扮演锁的角色。  
HashEntry 用于存储键值对数据。

```
static class Segment<K,V> extends ReentrantLock implements Serializable { }
```

一个 ConcurrentHashMap 里包含一个 Segment 数组。Segment 的结构和 HashMap 类似，是一种数组和链表结构，一个 Segment 包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素，每个 Segment 守护着一个 HashEntry 数组里的元素，当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment 的锁。

### JDK1.8

ConcurrentHashMap 取消了 Segment 分段锁，采用 CAS 和 synchronized 来保证并发安全。数据结构跟 HashMap1.8 的结构类似，数组+链表/红黑二叉树。Java 8 在链表长度超过一定阈值（8）时将链表（寻址时间复杂度为 O(N)）转换为红黑树（寻址时间复杂度为 O(log(N))）

synchronized 只锁定当前链表或红黑二叉树的首节点，这样只要 hash 不冲突，就不会产生并发，效率又提升 N 倍。

## 16、TreeMap 和 TreeSet 在排序时如何比较元素？Collections 工具类中的 sort()方法如何比较元素？

---

TreeSet 要求存放的对象所属的类必须实现 Comparable 接口，该接口提供了比较元素的 compareTo()方法，当插入元素时会回调该方法比较元素的大小。TreeMap 要求存放的键值对映射的键必须实现 Comparable 接口从而根据键对元素进行排序。

Collections 工具类的 sort 方法有两种重载的形式：

第一种要求传入的待排序容器中存放的对象比较实现 Comparable 接口以实现元素的比较。

第二种不强制性的要求容器中的元素必须可比较，但是要求传入第二个参数，参数是 Comparator 接口的子类型（需要重写 compare 方法实现元素的比较），相当于一个临时定义的排序规则，其实就是通过接口注入比较元素大小的算法，也是对回调模式的应用（Java 中对函数式编程的支持）。

## 17、Collection 和 Collections 有什么区别？

java.util.Collection 是一个集合接口（集合类的一个顶级接口）。它提供了对集合对象进行基本操作的通用接口方法。Collection 接口在 Java 类库中有很多具体的实现。Collection 接口的意义是为各种具体的集合提供了最大化的统一操作方式，其直接继承接口有 List 与 Set。

Collections 则是集合类的一个工具类/帮助类，其中提供了一系列静态方法，用于对集合中元素进行排序、搜索以及线程安全等各种操作。

## 18、Array 和 ArrayList 有何区别？

- Array 可以存储基本数据类型和对象，ArrayList 只能存储对象。
  - Array 是指定固定大小的，而 ArrayList 大小是自动扩展的。
- Array 内置方法没有 ArrayList 多，比如 addAll、removeAll、iteration 等方法只有 ArrayList 有。

对于基本类型数据，集合使用自动装箱来减少编码工作量。但是，当处理固定大小的基本数据类型的时候，这种方式相对比较慢。

## 19、HashMap 和 ConcurrentHashMap 的区别

ConcurrentHashMap 对整个桶数组进行了分割分段(Segment)，然后在每一个分段上都用 lock 锁进行保护，相对于 HashTable 的 synchronized 锁的粒度更精细了一些，并发性能更好，而 HashMap 没有锁机制，不是线程安全的。（JDK1.8 之后 ConcurrentHashMap 启用了一种全新的方式实现，利用 CAS 算法。）

HashMap 的键值对允许有 null，但是 ConcurrentHashMap 都不允许。

## 20、如果使用 Object 作为 HashMap 的 Key，应该怎么办呢？

重写 hashCode() 和 equals() 方法

- 
- 重写 hashCode() 是因为需要计算存储数据的存储位置，需要注意不要试图从散列码计算中排除掉一个对象的关键部分来提高性能，这样虽然能更快但可能会导致更多的 Hash 碰撞。
  - 重写 equals() 方法，需要遵守自反性、对称性、传递性、一致性以及对于任何非 null 的引用值 x，x.equals(null) 必须返回 false 的这几个特性，目的是为了保证 key 在哈希表中的唯一性。

## 21、为什么 HashMap 中 String、Integer 这样的包装类适合作为 K？

String、Integer 等包装类的特性能够保证 Hash 值的不可更改性和计算准确性，能够有效的减少 Hash 碰撞的几率。

- 都是 final 类型，即不可变性，保证 key 的不可更改性，不会存在获取 hash 值不同的情况
- 内部已重写了 equals()、hashCode() 等方法，遵守了 HashMap 内部的规范（不清楚可以去上面看看 putValue 的过程），不容易出现 Hash 值计算错误的情况；

## 22、什么是哈希冲突？

当两个不同的输入值，根据同一散列函数计算出相同的散列值的现象，我们就把它叫做碰撞（哈希碰撞）。

## 23、你知道 fail-fast 和 fail-safe 吗？

fail-fast 是 Java 中的一种快速失败机制，java.util 包下所有的集合都是快速失败的，快速失败会抛出 ConcurrentModificationException 异常，fail-fast 你可以把它理解为一种快速检测机制它只能用来检测错误，不会对错误进行恢复，fail-fast 不一定只在多线程环境下存在，ArrayList 也会抛出这个异常，主要原因是由于 modCount 不等于 expectedModCount。

fail-safe 是 Java 中的一种安全失败机制，它表示的是在遍历时不是直接在原集合上进行访问，而是先复制原有集合内容，在拷贝的集合上进行遍历。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 ConcurrentModificationException。java.util.concurrent 包下的容器都是安全失败的，可以在多线程条件下使用，并发修改。

## 24、Arrays.asList 获得的 List 应该注意什么？

Arrays.asList 是 Array 中的一个静态方法，它能够实现把数组转换成为 List 序列，需要注意下面几点：

- Arrays.asList 转换完成后的 List 不能再进行结构化的的修改，什么是结构化的修改？就是不能再进行任何 List 元素的增加或者减少的操作。

```
public static void main(String[] args) {
```

```
Integer[] integer = new Integer[] { 1, 2, 3, 4 };
List integelist = Arrays.asList(integer);
integelist.add(5);
}

// 结果抛出异常
// Exception in thread "main" java.lang.UnsupportedOperationException
```

我们看源码就很容易发现问题：  
// 这是 java.util.Arrays 内部类，  
// 而不是 java.util.ArrayList  
private static class ArrayList<E> extends AbstractList<E>  
 implements RandomAccess, java.io.Serializable

继承 AbstractList 中对 add、remove、set 方法是直接抛异常的，也就是说如果继承的子类没有去重写这些方法，那么子类的实例去调用这些方法是会直接抛异常的。

## 25、final、finally 和 finalize() 的区别

finally 是一个关键字，它经常和 try 块一起使用，用于异常处理。使用 try...finally 的代码块种，finally 部分的代码一定会被执行，所以我们经常在 finally 方法中用于资源的关闭操作。

JDK1.7 中，推荐使用 try-with-resources 优雅的关闭资源，它直接使用 try 语句进行资源的关闭即可，就不用写 finally 关键字了。

finalize 是 Object 对象中的一个方法，用于对象的回收方法，这个方法我们一般不推荐使用， finalize 是和垃圾回收关联在一起的，在 Java9 中，将 finalize 标记为了 deprecated，如果没有特别原因，不要实现 finalize 方法，也不要指望他来进行垃圾回收。

## 26、内部类有哪些分类，分别解释一下

在 Java 中，可以将一个类的定义放在另外一个类的定义内部，这就是内部类。内部类本身就是类的一个属性，与其他属性定义方式一致。内部类的分类一般主要有四种：

- 成员内部类
- 局部内部类
- 匿名内部类
- 静态内部类

静态内部类就是定义在类内部的静态类，静态内部类可以访问外部类所有的静态变量，而不可访问外部类的非静态变量；

成员内部类就是定义在类内部，成员位置上的非静态类，就是成员内部类。成员内部类可以访问外部类所有的变量和方法，包括静态和非静态，私有和公有。

定义在方法中的内部类，就是局部内部类。定义在实例方法中的局部类可以访问外部类的所有变量和方法，定义在静态方法中的局部类只能访问外部类的静态变量和方法。

匿名内部类就是没有名字的内部类，除了没有名字，匿名内部类还有以下特点：

- 匿名内部类必须继承一个抽象类或者实现一个接口
- 匿名内部类不能定义任何静态成员和静态方法。

- 
- 当所在的方法的形参需要被匿名内部类使用时，必须声明为 final。
  - 匿名内部类不能是抽象的，它必须要实现继承的类或者实现的接口的所有抽象方法。

## 27、项目为 UTF-8 环境，charc=中，是否合法

可以，因为 Unicode 编码采用 2 个字节的编码，UTF-8 是 Unicode 的一种实现，它使用可变长度的字符集进行编码，charc=中是两个字节，所以能够存储。合法。

## 28、动态代理是基于什么原理

代理一般分为静态代理和动态代理，它们都是代理模式的一种应用，静态代理指的是在程序运行前已经编译好，程序知道由谁来执行代理方法。

而动态代理只有在程序运行期间才能确定，相比于静态代理，动态代理的优势在于可以很方便的对代理类的函数进行统一的处理，而不用修改每个代理类中的方法。可以说动态代理是基于反射实现的。通过反射我们可以直接操作类或者对象，比如获取类的定义，获取声明的属性和方法，调用方法，在运行时可以修改类的定义。

动态代理是一种在运行时构建代理、动态处理方法调用的机制。动态代理的实现方式有很多，Java 提供的代理被称为 JDK 动态代理，JDK 动态代理是基于类的继承。

## 29、Exception 和 Error 有什么区别

Exception 泛指的是异常，Exception 主要分为两种异常，一种是编译期出现的异常，称为 checkedException，一种是程序运行期间出现的异常，称为 uncheckedException，常见的 checkedException 有 IOException，uncheckedException 统称为 RuntimeException，常见的 RuntimeException 主要有 NullPointerException、IllegalArgumentException、  
ArrayIndexOutOfBoundsException 等，Exception 可以被捕获。

Error 是指程序运行过程中出现的错误，通常情况下会造成程序的崩溃，Error 通常是不可恢复的，Error 不能被捕获。

## 30、反射的基本原理，反射创建类实例的三和方式是什么

反射机制就是使 Java 程序在运行时具有自省(introspect)的能力，通过反射我们可以直接操作类和对象，比如获取某个类的定义，获取类的属性和方法构造方法等。

创建类实例的三种方式是

- 对象实例.getClass();
- 通过 Class.forName() 创建
- 对象实例.newInstance() 方法创建

## ZooKeeper

### 1、ZooKeeper 是什么？

---

ZooKeeper 是一个开放源码的分布式协调服务，它是集群的管理者，监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

分布式应用程序可以基于 Zookeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。

Zookeeper 保证了如下分布式一致性特性：

- (1) 顺序一致性
- (2) 原子性
- (3) 单一视图
- (4) 可靠性
- (5) 实时性（最终一致性）

客户端的读请求可以被集群中的任意一台机器处理，如果读请求在节点上注册了监听器，这个监听器也是由所连接的 zookeeper 机器来处理。对于写请求，这些请求会同时发给其他 zookeeper 机器并且达成一致后，请求才会返回成功。因此，随着 zookeeper 的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降。

有序性是 zookeeper 中非常重要的一个特性，所有的更新都是全局有序的，每个更新都有一个唯一的时间戳，这个时间戳称为 zxid（Zookeeper Transaction Id）。而读请求只会相对于更新有序，也就是读请求的返回结果中会带有这个 zookeeper 最新的 zxid。

## 2、ZooKeeper 提供了什么？

- 文件系统
- 通知机制

## 3、Zookeeper 文件系统

Zookeeper 提供一个多层级的节点命名空间（节点称为 znode）。与文件系统不同的是，这些节点都可以设置关联的数据，而文件系统中只有文件节点可以存放数据而目录节点不行。

Zookeeper 为了保证高吞吐和低延迟，在内存中维护了这个树状的目录结构，这种特性使得 Zookeeper 不能用于存放大量的数据，每个节点的存放数据上限为 1M。

## 4、ZAB 协议？

ZAB 协议是为分布式协调服务 Zookeeper 专门设计的一种支持崩溃恢复的原子广播协议。

ZAB 协议包括两种基本的模式：崩溃恢复和消息广播。

当整个 zookeeper 集群刚刚启动或者 Leader 服务器宕机、重启或者网络故障导致不存在过半的服务器与 Leader 服务器保持正常通信时，所有进程（服务器）进入崩溃恢复模式，首

---

先选举产生新的 Leader 服务器，然后集群中 Follower 服务器开始与新的 Leader 服务器进行数据同步，当集群中超过半数 机器与该 Leader 服务器完成数据同步之后，退出恢复模式进入消息广播模 式，Leader 服务器开始接收客户端的事务请求生成事物提案来进行事务请求处理。

## 5、四种类型的数据节点 Znode

### (1) PERSISTENT-持久节点

除非手动删除，否则节点一直存在于 Zookeeper 上

### (2) EPHEMERAL-临时节点

临时节点的生命周期与客户端会话绑定，一旦客户端会话失效（客户端与 zookeeper 连接断开不一定会话失效），那么这个客户端创建的所有临时节点 都会被移除。

### (3) PERSISTENT\_SEQUENTIAL-持久顺序节点

基本特性同持久节点，只是增加了顺序属性，节点名后边会追加一个由父节点 维护的自增整型数字。

### (4) EPHEMERAL\_SEQUENTIAL-临时顺序节点

基本特性同临时节点，增加了顺序属性，节点名后边会追加一个由父节点维护 的自增整型数字。

## 6、Zookeeper Watcher 机制 -- 数据变更通知

Zookeeper 允许客户端向服务端的某个 Znode 注册一个 Watcher 监听，当服 务端的一些指定事件触发了这个 Watcher，服务端会向指定客户端发送一个事 件通知来实现分布式的通知功能，然后客户端根据 Watcher 通知状态和事件类 型做出业务上的改变。

工作机制：

### (1) 客户端注册 watcher

### (2) 服务端处理 watcher

### (3) 客户端回调 watcher

Watcher 特性总结：

### (1) 一次性

无论是服务端还是客户端，一旦一个 Watcher 被 触发，Zookeeper 都会将 其从相应的存 储中移除。这样的设计有效的减轻了服务端的压力，不然对于更 新非常频繁的节点，服务端 会不断的向客户端发送事件通知，无论对于网络还 是服务端的压力都非常大。

### (2) 客户端串行执行

客户端 Watcher 回调的过程是一个串行同步的过程。

### (3) 轻量

- Watcher 通知非常简单，只会告诉客户端发生了事件，而不会说明事件的 具体内容。

- 
- 客户端向服务端注册 Watcher 的时候，并不会把客户端真实的 Watcher 对象实体传递到服务端，仅仅是在客户端请求中使用 boolean 类型属性进行了标记。

(4) watcher event 异步发送 watcher 的通知事件从 server 发送到 client 是异步的，这就存在一个问题，不同的客户端和服务器之间通过 socket 进行通信，由于网络延迟或其他因素导致客户端在不通的时刻监听到事件，由于 Zookeeper 本身提供了 ordering guarantee，即客户端监听事件后，才会感知它所监视 znode 发生了变化。所以我们使用 Zookeeper 不能期望能够监控到 节点每次的变化。Zookeeper 只能保证最终的一致性，而无法保证强一致性。（5）注册 watcher getData、exists、getChildren

(6) 触发 watcher create、delete、setData

(7) 当一个客户端连接到一个新的服务器上时，watch 将会被以任意会话事件触发。当与一个服务器失去连接的时候，是无法接收到 watch 的。而当 client 重新连接时，如果需要的话，所有先前注册过的 watch，都会被重新注册。通常这是完全透明的。只有在一个特殊情况下，watch 可能会丢失：对于一个未创建的 znode 的 exist watch，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个 watch 事件可能会丢失。

## 7、客户端注册 Watcher 实现

(1) 调用 getData()/getChildren()/exist()三个 API，传入 Watcher 对象 (2) 标记请求 request，封装 Watcher 到 WatchRegistration

(3) 封装成 Packet 对象，发服务端发送 request

(4) 收到服务端响应后，将 Watcher 注册到 ZKWatcherManager 中进行管理 (5) 请求返回，完成注册。

## 8、服务端处理 Watcher 实现

(1) 服务端接收 Watcher 并存储

接收到客户端请求，处理请求判断是否需要注册 Watcher，需要的话将数据节点的节点路径和 ServerCnxn（ServerCnxn 代表一个客户端和服务端的连接，实现了 Watcher 的 process 接口，此时可以看成一个 Watcher 对象）存储在 WatcherManager 的 WatchTable 和 watch2Paths 中去。

(2) Watcher 触发

以服务端接收到 setData() 事务请求触发 NodeDataChanged 事件为例： 2.1 封装 WatchedEvent

---

将通知状态（SyncConnected）、事件类型（NodeDataChanged）以及节点路径 封装成一个 WatchedEvent 对象

查询 Watcher

- 从 WatchTable 中根据节点路径查找 Watcher
- 没找到；说明没有客户端在该数据节点上注册过 Watcher
- 找到；提取并从 WatchTable 和 Watch2Paths 中删除对应 Watcher（从这里可以看出 Watcher 在服务端是一次性的，触发一次就失效了）

(3) 调用 process 方法来触发 Watcher

这里 process 主要就是通过 ServerCnxn 对应的 TCP 连接发送 Watcher 事件 通知。

## 9、客户端回调 Watcher

客户端 SendThread 线程接收事件通知，交由 EventThread 线程回调 Watcher。

客户端的 Watcher 机制同样是一次性的，一旦被触发后，该 Watcher 就失效了。

## 10、ACL 权限控制机制

UGO (User/Group/Others)

目前在 Linux/Unix 文件系统中使用，也是使用最广泛的权限控制方式。是一种粗粒度的文件系统权限控制模式。

ACL (Access Control List) 访问控制列表

包括三个方面：

权限模式 (Scheme)

- (1) IP：从 IP 地址粒度进行权限控制
- (2) Digest：最常用，用类似于 username:password 的权限标识来进行权限 配置，便于区分不同应用来进行权限控制
- (3) World：最开放的权限控制方式，是一种特殊的 digest 模式，只有一个 权限标识 “world:anyone”
- (4) Super：超级用户

授权对象

授权对象指的是权限赋予的用户或一个指定实体，例如 IP 地址或是机器灯。权限 Permission

- (1) CREATE：数据节点创建权限，允许授权对象在该 Znode 下创建子节点
- (2) DELETE：子节点删除权限，允许授权对象删除该数据节点的子节点
- (3) READ：数据节点的读取权限，允许授权对象访问该数据节点并读取其数据 内容或子节点列表等

---

(4) WRITE: 数据节点更新权限, 允许授权对象对该数据节点进行更新操作 (5) ADMIN: 数据节点管理权限, 允许授权对象对该数据节点进行 ACL 相关设置操作

## 11、Chroot 特性

3.2.0 版本后, 添加了 Chroot 特性, 该特性允许每个客户端为自己设置一个命名空间。如果一个客户端设置了 Chroot, 那么该客户端对服务器的任何操作, 都将会被限制在其自己的命名空间下。

通过设置 Chroot, 能够将一个客户端应用于 Zookeeper 服务端的一颗子树相对应, 在那些多个应用公用一个 Zookeeper 进群的场景下, 对实现不同应用间的相互隔离非常有帮助。

## 12、会话管理

分桶策略: 将类似的会话放在同一区块中进行管理, 以便于 Zookeeper 对会话进行不同区块的隔离处理以及同一区块的统一处理。

分配原则: 每个会话的“下次超时时间点”(ExpirationTime)

计算公式:

$\text{ExpirationTime}_\text{current} = \text{currentTime} + \text{sessionTimeout}$

$\text{ExpirationTime} = (\text{ExpirationTime}_\text{current} / \text{ExpirationInterval} + 1) * \text{ExpirationInterval}$

ExpirationInterval, ExpirationInterval 是指 Zookeeper 会话超时检查时间间隔, 默认 tickTime

## 13、服务器角色

Leader

- (1) 事务请求的唯一调度和处理器, 保证集群事务处理的顺序性
- (2) 集群内部各服务的调度者

Follower

- (1) 处理客户端的非事务请求, 转发事务请求给 Leader 服务器
- (2) 参与事务请求 Proposal 的投票
- (3) 参与 Leader 选举投票

Observer

- (1) 3.0 版本以后引入的一个服务器角色, 在不影响集群事务处理能力的基础上提升集群的非事务处理能力
- (2) 处理客户端的非事务请求, 转发事务请求给 Leader 服务器
- (3) 不参与任何形式的投票

## 14、Zookeeper 下 Server 工作状态

---

服务器具有四种状态，分别是 LOOKING、FOLLOWING、LEADING、OBSERVING。 (1)

LOOKING：寻找 Leader 状态。当服务器处于该状态时，它会认为当前集群中没有 Leader，因此需要进入 Leader 选举状态。

(2) FOLLOWING：跟随者状态。表明当前服务器角色是 Follower。

(3) LEADING：领导者状态。表明当前服务器角色是 Leader。

(4) OBSERVING：观察者状态。表明当前服务器角色是 Observer。

## 15、数据同步

整个集群完成 Leader 选举之后，Learner（Follower 和 Observer 的统称）向 Leader 服务器进行注册。当 Learner 服务器向 Leader 服务器完成注册后，进入数据同步环节。

数据同步流程：（均以消息传递的方式进行）

Learner 向 Leader 注册

数据同步

同步确认

Zookeeper 的数据同步通常分为四类：

- (1) 直接差异化同步 (DIFF 同步)
- (2) 先回滚再差异化同步 (TRUNC+DIFF 同步)
- (3) 仅回滚同步 (TRUNC 同步)
- (4) 全量同步 (SNAP 同步)

在进行数据同步前，Leader 服务器会完成数据同步初始化：

peerLastZxid：

- 从 learner 服务器注册时发送的 ACKEPOCH 消息中提取 lastZxid（该 Learner 服务器最后处理的 ZXID）

minCommittedLog：

- Leader 服务器 Proposal 缓存队列 committedLog 中最小

ZXIDmaxCommittedLog：

- Leader 服务器 Proposal 缓存队列 committedLog 中最大 ZXID 直接差异化同步 (DIFF 同步)

- 场景：peerLastZxid 介于 minCommittedLog 和 maxCommittedLog 之间先回滚再差异化同步 (TRUNC+DIFF 同步)

- 场景：当新的 Leader 服务器发现某个 Learner 服务器包含了一条自己没有的事务记录，那么就需要让该 Learner 服务器进行事务回滚--回滚到 Leader 服务器上存在的，同时也是最接近于 peerLastZxid 的 ZXID 仅回滚同步 (TRUNC 同步)

- 场景：peerLastZxid 大于 maxCommittedLog

全量同步 (SNAP 同步)

- 
- 场景一：peerLastZxid 小于 minCommittedLog
  - 场景二：Leader 服务器上没有 Proposal 缓存队列且 peerLastZxid 不等于 lastProcessZxid

## 16、zookeeper 是如何保证事务的顺序一致性的？

zookeeper 采用了全局递增的事务 Id 来标识，所有的 proposal（提议）都在被提出的时候加上了 zxid，zxid 实际上是一个 64 位的数字，高 32 位是 epoch（时期；纪元；世；新时代）用来标识 leader 周期，如果有新的

leader 产生出来，epoch 会自增，低 32 位用来递增计数。当新产生 proposal 的时候，会依据数据库的两阶段过程，首先会向其他的 server 发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

## 17、分布式集群中为什么会有 Master？

在分布式环境中，有些业务逻辑只需要集群中的某一台机器进行执行，其他的机器可以共享这个结果，这样可以大大减少重复计算，提高性能，于是就需要进行 leader 选举。

## 18、zk 节点宕机如何处理？

Zookeeper 本身也是集群，推荐配置不少于 3 个服务器。Zookeeper 自身也要保证当一个节点宕机时，其他节点会继续提供服务。

如果是一个 Follower 宕机，还有 2 台服务器提供访问，因为 Zookeeper 上的数据是多个副本的，数据并不会丢失；

如果是一个 Leader 宕机，Zookeeper 会选举出新的 Leader。

ZK 集群的机制是只要超过半数的节点正常，集群就能正常提供服务。只有在 ZK 节点挂得太多，只剩一半或不到一半节点能工作，集群才失效。

所以

3 个节点的 cluster 可以挂掉 1 个节点 (leader 可以得到 2 票  $> 1.5$ )

2 个节点的 cluster 就不能挂掉任何 1 个节点了 (leader 可以得到 1 票  $\leq 1$ )

## 19、zookeeper 负载均衡和 nginx 负载均衡区别

---

zk 的负载均衡是可以调控，nginx 只是能调权重，其他需要可控的都需要自己写插件；但是 nginx 的吞吐量比 zk 大很多，应该说按业务选择用哪种方式。

## 20. Zookeeper 有哪几种几种部署模式？

部署模式：单机模式、伪集群模式、集群模式。

## 21. 集群最少要几台机器，集群规则是怎样的？

集群规则为  $2N+1$  台， $N>0$ ，即 3 台。

## 22. 集群支持动态添加机器吗？

其实就是水平扩容了，Zookeeper 在这方面不太好。两种方式：

全部重启：关闭所有 Zookeeper 服务，修改配置之后启动。不影响之前客户端的会话。

逐个重启：在过半存活即可用的原则下，一台机器重启不影响整个集群对外提供服务。这是比较常用的方式。

3.5 版本开始支持动态扩容。

## 23. Zookeeper 对节点的 watch 监听通知是永久的吗？为什么不是永久的？

不是。官方声明：一个 Watch 事件是一个一次性的触发器，当被设置了 Watch 的数据发生了改变的时候，则服务器将这个改变发送给设置了 Watch 的客户端，以便通知它们。

为什么不是永久的，举个例子，如果服务端变动频繁，而监听的客户端很多情况下，每次变动都要通知到所有的客户端，给网络和服务器造成很大压力。一般是客户端执行 `getData(“/节点 A”,true)`，如果节点 A 发生了变更或删除，客户端会得到它的 watch 事件，但是在之后节点 A 又发生了变更，而客户端又没有设置 watch 事件，就不再给客户端发送。

在实际应用中，很多情况下，我们的客户端不需要知道服务端的每一次变动，我只要最新的数据即可。

## 24. Zookeeper 的 java 客户端都有哪些？

java 客户端：zk 自带的 zkclient 及 Apache 开源的 Curator。

## 25. chubby 是什么，和 zookeeper 比你怎么看？

---

chubby 是 google 的，完全实现 paxos 算法，不开源。zookeeper 是 chubby 的开源实现，使用 zab 协议，paxos 算法的变种。

## 26. 说几个 zookeeper 常用的命令。

常用命令：ls get set create delete 等。

## 27. ZAB 和 Paxos 算法的联系与区别？

相同点：

- (1) 两者都存在一个类似于 Leader 进程的角色，由其负责协调多个 Follower 进程的运行
- (2) Leader 进程都会等待超过半数的 Follower 做出正确的反馈后，才会将一个提案进行提交
- (3) ZAB 协议中，每个 Proposal 中都包含一个 epoch 值来代表当前的 Leader 周期，Paxos 中名字为 Ballot

不同点：

ZAB 用来构建高可用的分布式数据主备系统（Zookeeper），Paxos 是用来构建分布式一致性状态机系统。

## 28. Zookeeper 的典型应用场景

Zookeeper 是一个典型的发布/订阅模式的分布式数据管理与协调框架，开发人员可以使用它来进行分布式数据的发布和订阅。

通过对 Zookeeper 中丰富的数据节点进行交叉使用，配合 Watcher 事件通知机制，可以非常方便的构建一系列分布式应用中都会涉及的核心功能，如：

- (1) 数据发布/订阅
- (2) 负载均衡
- (3) 命名服务
- (4) 分布式协调/通知
- (5) 集群管理
- (6) Master 选举
- (7) 分布式锁
- (8) 分布式队列

数据发布/订阅

介绍

---

数据发布/订阅系统，即所谓的配置中心，顾名思义就是发布者发布数据供订阅者进行数据订阅。

## 目的

动态获取数据（配置信息）

实现数据（配置信息）的集中式管理和数据的动态更新

## 设计模式

Push 模式

Pull 模式

数据（配置信息）特性

- (1) 数据量通常比较小
- (2) 数据内容在运行时会发生动态更新
- (3) 集群中各机器共享，配置一致

如：机器列表信息、运行时开关配置、数据库配置信息等

基于 Zookeeper 的实现方式

- 数据存储：将数据（配置信息）存储到 Zookeeper 上的一个数据节点
  
- 数据获取：应用在启动初始化节点从 Zookeeper 数据节点读取数据，并在该节点上注册一个数据变更 Watcher
- 数据变更：当变更数据时，更新 Zookeeper 对应节点数据，Zookeeper 会将数据变更通知发到各客户端，客户端接到通知后重新读取变更后的数据即可。

## 负载均衡

### zk 的命名服务

命名服务是指通过指定的名字来获取资源或者服务的地址，利用 zk 创建一个全局的路径，这个路径就可以作为一个名字，指向集群中的集群，提供的服务的地址，或者一个远程的对象等等。

### 分布式通知和协调

对于系统调度来说：操作人员发送通知实际是通过控制台改变某个节点的状态，然后 zk 将这些变化发送给注册了这个节点的 watcher 的所有客户端。对于执行情况汇报：每个工作进程都在某个目录下创建一个临时节点。并携带工作的进度数据，这样汇总的进程可以监控目录子节点的变化获得工作进度的实时的全局情况。

### zk 的命名服务（文件系统）

---

命名服务是指通过指定的名字来获取资源或者服务的地址，利用 zk 创建一个全局的路径，即是唯一的路径，这个路径就可以作为一个名字，指向集群中的集群，提供的服务的地址，或者一个远程的对象等等。

### zk 的配置管理（文件系统、通知机制）

程序分布式的部署在不同的机器上，将程序的配置信息放在 zk 的 znode 下，当有配置发生改变时，也就是 znode 发生变化时，可以通过改变 zk 中某个目录节点的内容，利用 watcher 通知给各个客户端，从而更改配置。Zookeeper 集群管理（文件系统、通知机制）

所谓集群管理无在乎两点：是否有机器退出和加入、选举 master。

对于第一点，所有机器约定在父目录下创建临时目录节点，然后监听父目录节点的子节点变化消息。一旦有机器挂掉，该机器与 zookeeper 的连接断开，其所创建的临时目录节点被删除，所有其他机器都收到通知：某个兄弟目录被删除，于是，所有人都知道：它上船了。

新机器加入也是类似，所有机器收到通知：新兄弟目录加入，highcount 又有了，对于第二点，我们稍微改变一下，所有机器创建临时顺序编号目录节点，每次选取编号最小的机器作为 master 就好。

### Zookeeper 分布式锁（文件系统、通知机制）

有了 zookeeper 的一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是保持独占，另一个是控制时序。

对于第一类，我们将 zookeeper 上的一个 znode 看作是一把锁，通过 createznode 的方式来实现。所有客户端都去创建 /distribute\_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。用完删除掉自己创建的 distribute\_lock 节点就释放出锁。

对于第二类，/distribute\_lock 已经预先存在，所有客户端在它下面创建临时顺序编号目录节点，和选 master 一样，编号最小的获得锁，用完删除，依次方便。

### Zookeeper 队列管理（文件系统、通知机制）

两种类型的队列：

- (1) 同步队列，当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达。
- (2) 队列按照 FIFO 方式进行入队和出队操作。

第一类，在约定目录下创建临时目录节点，监听节点数目是否是我们要求的数目。

第二类，和分布式锁服务中的控制时序场景基本原理一致，入列有编号，出列按编号。在特定的目录下创建 PERSISTENT\_SEQUENTIAL 节点，创建成功时 Watcher 通知等待的队列，队列删除序列号最小的节点用以消费。此场景下 Zookeeper 的 znode 用于消息存储，znode 存储的数据就是消息队列中的消息内容，SEQUENTIAL 序列号就是消息的编号，按序取出即可。由于创建的节点是持久化的，所以不必担心队列消息的丢失问题。

---

作者：程序员追风

链接：<https://juejin.cn/post/6844904047309225991>

来源：掘金

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## Java 并发编程

### 1、在 java 中守护线程和本地线程区别？

java 中的线程分为两种：守护线程（Daemon）和用户线程（User）。任何线程都可以设置为守护线程和用户线程，通过方法

`Thread.setDaemon(boolean);` true 则把该线程设置为守护线程，反之则为用户线程。

`Thread.setDaemon()` 必须在 `Thread.start()` 之前调用，否则运行时会抛出异常。

两者的区别：

唯一的区别是判断虚拟机(JVM)何时离开，Daemon 是为其他线程提供服务，如果全部的 User Thread 已经撤离，Daemon 没有可服务的线程，JVM 撤离。也可以理解为守护线程是 JVM 自动创建的线程（但不一定），用户线程是程序创建的线程；比如 JVM 的垃圾回收线程是一个守护线程，当所有线程已经撤离，不再产生垃圾，守护线程自然就没事可干了，当垃圾回收线程是 Java 虚拟机上仅剩的线程时，Java 虚拟机会自动离开。

扩展：Thread Dump 打印出来的线程信息，含有 `daemon` 字样的线程即为守护进程，可能会有：服务守护进程、编译守护进程、windows 下的监听 `Ctrl+break` 的守护进程、`Finalizer` 守护进程、引用处理守护进程、GC 守护进程。

### 2、线程与进程的区别？

进程是操作系统分配资源的最小单元，线程是操作系统调度的最小单元。一个程序至少有一个进程，一个进程至少有一个线程。

### 3、什么是多线程中的上下文切换？

---

多线程会共同使用一组计算机上的 CPU，而线程数大于给程序分配的 CPU 数量时，为了让各个线程都有执行的机会，就需要轮转使用 CPU。不同的线程切换使用 CPU 发生的切换数据等就是上下文切换。

## 4、死锁与活锁的区别，死锁与饥饿的区别？

死锁：是指两个或两个以上的进程（或线程）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。

产生死锁的必要条件：

- 互斥条件：所谓互斥就是进程在某一时间内独占资源。
- 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
- 不剥夺条件：进程已获得资源，在未使用完之前，不能强行剥夺。
- 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

活锁：任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。

活锁和死锁的区别在于，处于活锁的实体是在不断的改变状态，所谓的

“活”，而处于死锁的实体表现为等待；活锁有可能自行解开，死锁则不能。饥饿：一个或者多个线程因为种种原因无法获得所需要的资源，导致一直无法执行的状态。

Java 中导致饥饿的原因：

- 高优先级线程吞噬所有的低优先级线程的 CPU 时间。
- 线程被永久堵塞在一个等待进入同步块的状态，因为其他线程总是能在它之前持续地对该同步块进行访问。
- 线程在等待一个本身也处于永久等待完成的对象(比如调用这个对象的 wait 方法)，因为其他线程总是被持续地获得唤醒。
- Java 中用到的线程调度算法是什么？
- 采用时间片轮转的方式。可以设置线程的优先级，会映射到下层的系统上面的优先级上，如非特别需要，尽量不要用，防止线程饥饿。

## 5、什么是线程组，为什么在 Java 中不推荐使用？

---

ThreadGroup 类，可以把线程归属到某一个线程组中，线程组中可以有线程对象，也可以有线程组，组中还可以有线程，这样的组织结构有点类似于树的形式。  
为什么不推荐使用？因为使用有很多的安全隐患吧，没有具体追究，如果需要使用，推荐使用线程池。

## 6、为什么使用 Executor 框架？

每次执行任务创建线程 new Thread() 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，被称为野线程，而且可以无限制的创建，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，还有线程之间的频繁交替也会消耗很多系统资源。

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、线程中断等都不便实现。

## 7、在 Java 中 Executor 和 Executors 的区别？

Executors 工具类的不同方法按照我们的需求创建了不同的线程池，来满足业务的需求。

Executor 接口对象能执行我们的线程任务。

ExecutorService 接口继承了 Executor 接口并进行了扩展，提供了更多的方法我们能获得任务执行的状态并且可以获取任务的返回值。

使用 ThreadPoolExecutor 可以创建自定义线程池。

Future 表示异步计算的结果，他提供了检查计算是否完成的方法，以等待计算的完成，并可以使用 get() 方法获取计算的结果。

## 8、什么是原子操作？在 Java Concurrency API 中有哪些原子类(atomic classes)？

原子操作（atomic operation）意为“不可被中断的一个或一系列操作”。处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。在 Java 中可以通过锁和循环 CAS 的方式来实现原子操作。CAS 操作—  
—Compare & Set，或是 Compare & Swap，现在几乎所有的 CPU 指令都支持 CAS 的原子操作。

---

原子操作是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境下避免数据不一致必须的手段。

int++并不是一个原子操作，所以当一个线程读取它的值并加 1 时，另外一个线程有可能会读到之前的值，这就会引发错误。

为了解决这个问题，必须保证增加操作是原子的，在 JDK1.5 之前我们可以使用同步技术来做到这一点。到 JDK1.5，java.util.concurrent.atomic 包提供了 int 和 long 类型的原子包装类，它们可以自动的保证对于他们的操作是原子的并且不需要使用同步。

java.util.concurrent 这个包里面提供了一组原子类。其基本的特性就是在多线程环境下，当有多个线程同时执行这些类的实例包含的方法时，具有排他

性，即当某个线程进入方法，执行其中的指令时，不会被其他线程打断，而别的线程就像自旋锁一样，一直等到该方法执行完成，才由 JVM 从等待队列中选择一个另一个线程进入，这只是一个逻辑上的理解。

原子类：AtomicBoolean, AtomicInteger, AtomicLong, AtomicReference 原子数组：

AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray 原子属性更新器：

AtomicLongFieldUpdater, AtomicIntegerFieldUpdater,

AtomicReferenceFieldUpdater

解决 ABA 问题的原子类：AtomicMarkableReference（通过引入一个 boolean 来反映中间有没有变过），AtomicStampedReference（通过引入一个 int 来累加来反映中间有没有变过）



## 9、Java Concurrency API 中的 Lock 接口(Lock interface)是什么？对比同步它有什么优势？

Lock 接口比同步方法和同步块提供了更具扩展性的锁操作。

他们允许更灵活的结构，可以具有完全不同的性质，并且可以支持多个相关类的条件对象。

它的优势有：可以使锁更公平

可以使线程在等待锁的时候响应中断

可以让线程尝试获取锁，并在无法获取锁的时候立即返回或者等待一段时间可以在不同的范围，以不同的顺序获取和释放锁

整体上来说 Lock 是 synchronized 的扩展版，Lock 提供了无条件的、可轮询的(tryLock 方法)、定时的(tryLock 带参方法)、可中断的(lockInterruptibly)、可多条件队列的(newCondition 方法)锁操作。另外 Lock 的实现类基本都支持非公平锁(默认)和公平锁，synchronized 只支持非公平锁，当然，在大部分情况下，非公平锁是高效的选择。

## 10、什么是 Executors 框架？

---

Executor 框架是一个根据一组执行策略调用，调度，执行和控制的异步任务的框架。无限制的创建线程会引起应用程序内存溢出。所以创建一个线程池是个更好的的解决方案，因为可以限制线程的数量并且可以回收再利用这些线程。利用 Executors 框架可以非常方便的创建一个线程池。

## 11、什么是阻塞队列？阻塞队列的实现原理是什么？如何使用阻塞队列来实现生产者-消费者模型？

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。

这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。

阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

JDK7 提供了 7 个阻塞队列。分别是：

ArrayBlockingQueue：一个由数组结构组成的有界阻塞队列。LinkedBlockingQueue：一个由链表结构组成的有界阻塞队列。PriorityBlockingQueue：一个支持优先级排序的无界阻塞队列。DelayQueue：一个使用优先级队列实现的无界阻塞队列。SynchronousQueue：一个不存储元素的阻塞队列。LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

Java 5 之前实现同步存取时，可以使用普通的一个集合，然后在使用线程的协作和线程同步可以实现生产者，消费者模式，主要的技术就是用好，

wait ,notify,notifyAll,synchronized 这些关键字。而在 java 5 之后，可以

使用阻塞队列来实现，此方式大大简少了代码量，使得多线程编程更加容易，安全方面也有保障。

BlockingQueue 接口是 Queue 的子接口，它的主要用途并不是作为容器，而是作为线程同步的工具，因此他具有一个很明显的特性，当生产者线程试图向 BlockingQueue 放入元素时，如果队列已满，则线程被阻塞，当消费者线程试图从中取出一个元素时，如果队列为空，则该线程会被阻塞，正是因为它所具有这个特性，所以在程序中多个线程交替向 BlockingQueue 中放入元素，取出元素，它可以很好的控制线程之间的通信。

阻塞队列使用最经典的场景就是 socket 客户端数据的读取和解析，读取数据的线程不断将数据放入队列，然后解析线程不断从队列取数据解析。

## 12、什么是 Callable 和 Future?

---

Callable 接口类似于 Runnable，从名字就可以看出来了，但是 Runnable 不会返回结果，并且无法抛出返回结果的异常，而 Callable 功能更强大一些，被线程执行后，可以返回值，这个返回值可以被 Future 拿到，也就是说，Future 可以拿到异步执行任务的返回值。

可以认为是带有回调的 Runnable。

Future 接口表示异步任务，是还没有完成的任务给出的未来结果。所以说 Callable 用于产生结果，Future 用于获取结果。

## 13、什么是 FutureTask?使用 ExecutorService 启动任务。

在 Java 并发程序中 FutureTask 表示一个可以取消的异步运算。它有启动和取消运算、查询运算是否完成和取回运算结果等方法。只有当运算完成的时候结果才能取回，如果运算尚未完成 get 方法将会阻塞。一个 FutureTask 对象可以对调用了 Callable 和 Runnable 的对象进行包装，由于 FutureTask 也是调用了 Runnable 接口所以它可以提交给 Executor 来执行。

## 14、什么是并发容器的实现？



何为同步容器：可以简单地理解为通过 synchronized 来实现同步的容器，如果有多个线程调用同步容器的方法，它们将会串行执行。比如 Vector, Hashtable，以及 Collections.synchronizedSet, synchronizedList 等方法返回的容器。可以通过查看 Vector, Hashtable 等这些同步容器的实现代码，可以看到这些容器实现线程安全的方式就是将它们的状态封装起来，并在需要同步的方法上加上关键字 synchronized。

并发容器使用了与同步容器完全不同的加锁策略来提供更高的并发性和伸缩

性，例如在 ConcurrentHashMap 中采用了一种粒度更细的加锁机制，可以称为分段锁，在这种锁机制下，允许任意数量的读线程并发地访问 map，并且执行

读操作的线程和写操作的线程也可以并发的访问 map，同时允许一定数量的写操作线程并发地修改 map，所以它可以在并发环境下实现更高的吞吐量。

## 15、多线程同步和互斥有几种实现方法，都是什么？

线程同步是指线程之间所具有的一种制约关系，一个线程的执行依赖另一个线程的消息，当它没有得到另一个线程的消息时应等待，直到消息到达时才被唤醒。线程互斥是指对于共享的进程系统资源，在各单个线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何

---

时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步。

线程间的同步方法大体可分为两类：用户模式和内核模式。顾名思义，内核模式就是指利用系统内核对象的单一性来进行同步，使用时需要切换内核态与用户态，而用户模式就是不需要切换到内核态，只在用户态完成操作。

用户模式下的方法有：原子操作（例如一个单一的全局变量），临界区。内核模式下的方法有：事件，信号量，互斥量。

## 16、什么是竞争条件？你怎样发现和解决竞争？

当多个进程都企图对共享数据进行某种处理，而最后的结果又取决于进程运行的顺序时，则我们认为这发生了竞争条件（race condition）。

## 17、你将如何使用 thread dump？你将如何分析 Thread dump？

### 新建状态（New）

用 new 语句创建的线程处于新建状态，此时它和其他 Java 对象一样，仅仅在堆区中被分配了内存。就绪状态（Runnable）

当一个线程对象创建后，其他线程调用它的 start() 方法，该线程就进入就绪状态，Java 虚拟机会为它创建方法调用栈和程序计数器。处于这个状态的线程位于可运行池中，等待获得 CPU 的使用权。

### 运行状态（Running）

处于这个状态的线程占用 CPU，执行程序代码。只有处于就绪状态的线程才有机会转到运行状态。

### 阻塞状态（Blocked）

阻塞状态是指线程因为某些原因放弃 CPU，暂时停止运行。当线程处于阻塞状态时，Java 虚拟机不会给线程分配 CPU。直到线程重新进入就绪状态，它才有机会转到运行状态。

阻塞状态可分为以下 3 种：

位于对象等待池中的阻塞状态（Blocked in object's wait pool）：

当线程处于运行状态时，如果执行了某个对象的 wait() 方法，Java 虚拟机就会把线程放到这个对象的等待池中，这涉及到“线程通信”的内容。

位于对象锁池中的阻塞状态（Blocked in object's lock pool）：

当线程处于运行状态时，试图获得某个对象的同步锁时，如果该对象的同步锁已经被其他线程占用，Java 虚拟机就会把这个线程放到这个对象的锁池中，这涉及到“线程同步”的内容。

其他阻塞状态（Otherwise Blocked）：

---

当前线程执行了 sleep()方法，或者调用了其他线程的 join()方法，或者发出了 I/O 请求时，就会进入这个状态。

死亡状态（Dead）

当线程退出 run()方法时，就进入死亡状态，该线程结束生命周期。

## 18、为什么我们调用 start()方法时会执行 run() 方法，为什么我们不能直接调用 run()方法？

当你调用 start()方法时你将创建新的线程，并且执行在 run()方法里的代码。

但是如果你直接调用 run()方法，它不会创建新的线程也不会执行调用线程的代码，只会把 run 方法当作普通方法去执行。

## 19、Java 中你怎样唤醒一个阻塞的线程？

在 Java 发展史上曾经使用 suspend()、resume()方法对于线程进行阻塞唤醒，但随之出现很多问题，比较典型的还是死锁问题。

解决方案可以使用以对象为目标的阻塞，即利用 Object 类的 wait()和 notify()方法实现线程阻塞。

首先，wait、notify 方法是针对对象的，调用任意对象的 wait()方法都将导致线程阻塞，阻塞的同时也将释放该对象的锁，相应地，调用任意对象的 notify()方法则将随机解除该对象阻塞的线程，但它需要重新获取改对象的 锁，直到获取成功才能往下执行；其次，wait、notify 方法必须在 synchronized 块或方法中被调用，并且要保证同步块或方法的锁对象与调用 wait、notify 方法的对象是同一个，如此一来在调用 wait 之前当前线程就已经成功获取某对象的锁，执行 wait 阻塞后当前线程就将之前获取的对象锁释放。

## 20、在 Java 中 CyclicBarrier 和 CountdownLatch 有什么区别？

CyclicBarrier 可以重复使用，而 CountdownLatch 不能重复使用。

Java 的 concurrent 包里面的 CountDownLatch 其实可以把它看作一个计数器，只不过这个计数器的操作是原子操作，同时只能有一个线程去操作这个计数器，也就是同时只能有一个线程去减这个计数器里面的值。你可以向 CountDownLatch 对象设置一个初始的数字作为计数值，任何调用这个对象上的 await()方法都会阻塞，直到这个计数器的计数值被其他的线程减为 0 为止。所以在当前计数到达零之前，await 方法会一直受阻塞。之后，会释放所有等

---

待的线程，await 的所有后续调用都将立即返回。这种现象只出现一次——计数无法被重置。如果需要重置计数，请考虑使用 CyclicBarrier。CountDownLatch 的一个非常典型的应用场景是：有一个任务想要往下执行，但必须要等到其他的任务执行完毕后才可以继续往下执行。假如我们这个想要继续往下执行的任务调用一个 CountDownLatch 对象的 await()方法，其他的任务执行完自己的任务后调用同一个 CountDownLatch 对象上的 countDown()方法，这个调用 await()方法的任务将一直阻塞等待，直到这个 CountDownLatch 对象的计数值减到 0 为止。

CyclicBarrier 一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)。在涉及一组固定大小的线程的程序中，这些线程必须不断地互相等待，此时 CyclicBarrier 很有用。因为该 barrier 在释放等待线程后可以重用，所以称它为循环的 barrier。

## 21、什么是不可变对象，它对写并发应用有什么帮助？

不可变对象(Immutable Objects)即对象一旦被创建它的状态（对象的数据，也即对象属性值）就不能改变，反之即为可变对象(Mutable Objects)。

不可变对象的类即为不可变类(Immutable Class)。Java 平台类库中包含许多不可变类，如 String、基本类型的包装类、BigInteger 和 BigDecimal 等。不可变对象天生是线程安全的。它们的常量（域）是在构造函数中创建的。既然它们的状态无法修改，这些常量永远不会变。不可变对象永远是线程安全的。

只有满足如下状态，一个对象才是不可变的；它的状态不能在创建后再被修改；所有域都是 final 类型；并且，它被正确创建（创建期间没有发生 this 引用的逸出）。

## 22、什么是多线程中的上下文切换？

在上下文切换过程中，CPU 会停止处理当前运行的程序，并保存当前程序运行的具体位置以便之后继续运行。从这个角度来看，上下文切换有点像我们同时阅读几本书，在来回切换书本的同时我们需要记住每本书当前读到的页码。在程序中，上下文切换过程中的“页码”信息是保存在进程控制块（PCB）中的。

PCB 还经常被称作“切换桢”（switchframe）。 “页码”信息会一直保存到 CPU 的内存中，直到他们被再次使用。

上下文切换是存储和恢复 CPU 状态的过程，它使得线程执行能够从中断点恢复执行。上下文切换是多任务操作系统和多线程环境的基本特征。

## 23、Java 中用到的线程调度算法是什么？

计算机通常只有一个 CPU，在任意时刻只能执行一条机器指令，每个线程只有获得 CPU 的使用权才能执行指令。所谓多线程的并发运行，其实是指从宏观上看，各个线程轮流获得 CPU 的使用权，分别执行各自的任务。在运行池中，会有多个处于就绪状态的线程在等待 CPU，JAVA 虚拟机的一项任务就是负责线程的调度，线程调度是指按照特定机制为多个线程分配 CPU 的使用权。有两种调度模型：分时调度模型和抢占式调度模型。

分时调度模型是指让所有的线程轮流获得 cpu 的使用权，并且平均分配每个线程占用的 CPU 的时间片这个也比较好理解。

Java 虚拟机采用抢占式调度模型，是指优先让可运行池中优先级高的线程占用 CPU，如果可运行池中的线程优先级相同，那么就随机选择一个线程，使其占用 CPU。处于运行状态的线程会一直运行，直至它不得不放弃 CPU。

## 24、什么是线程组，为什么在 Java 中不推荐使用？

线程组和线程池是两个不同的概念，他们的作用完全不同，前者是为了方便线程的管理，后者是为了管理线程的生命周期，复用线程，减少创建销毁线程的开销。



## 25、为什么使用 Executor 框架比使用应用创建和管理线程好？

为什么要使用 Executor 线程池框架

- 每次执行任务创建线程 new Thread() 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，被称为野线程，而且可以无限制的创建，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，还有线程之间的频繁交替也会消耗很多系统资源。
- 
- 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、线程中断等都不便实现。

使用 Executor 线程池框架的优点

- 能复用已存在并空闲的线程从而减少线程对象的创建从而减少了消亡线程的开销。
- 可有效控制最大并发线程数，提高系统资源使用率，同时避免过多资源竞争。
- 框架中已经有定时、定期、单线程、并发数控制等功能。

综上所述使用线程池框架 Executor 能更好的管理线程、提供系统资源使用率。

---

## 26、Java 中有几种方法可以实现一个线程？

继承 Thread 类

实现 Runnable 接口

实现 Callable 接口，需要实现的是 call() 方法

## 27、如何停止一个正在运行的线程？

使用共享变量的方式

在这种方式中，之所以引入共享变量，是因为该变量可以被多个执行相同任务的线程用来作为是否中断的信号，通知中断线程的执行。

使用 interrupt 方法终止线程

如果一个线程由于等待某些事件的发生而被阻塞，又该怎样停止该线程呢？这种情况经常会发生，比如当一个线程由于需要等候键盘输入而被阻塞，或者调用 Thread.join()方法，或者 Thread.sleep()方法，在网络中调用 ServerSocket.accept()方法，或者调用了 DatagramSocket.receive()方法时，都有可能导致线程阻塞，使线程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。这里我们给出的建议是，不要使用 stop()方法，而是使用 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，退出堵塞代码。

## 28、notify()和 notifyAll()有什么区别？

当一个线程进入 wait 之后，就必须等其他线程 notify/notifyall, 使用 notifyall, 可以唤醒所有处于 wait 状态的线程，使其重新进入锁的争夺队列中，而 notify 只能唤醒一个。

如果没把握，建议 notifyAll，防止 notigy 因为信号丢失而造成程序异常。

## 29、什么是 Daemon 线程？它有什么意义？

所谓后台(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。

## 30、Java 如何实现多线程之间的通讯和协作？

### 31、什么是可重入锁（ReentrantLock）？

举例来说明锁的可重入性

```
public class UnReentrant{  
    Lock lock = new Lock();  
    public void outer(){  
        lock.lock();  
        inner(); lock.unlock();  
    }  
    public void inner(){  
        lock.lock();  
        //do something lock.unlock();  
    }  
}
```

}复制代码

outer 中调用了 inner, outer 先锁住了 lock, 这样 inner 就不能再获取 lock。其实调用 outer 的线程已经获取了 lock 锁，但是不能在 inner 中重复利用已经获取的锁资源，这种锁即称之为 不可重入可重入就意味着：线程可以进入任何一个它已经拥有的锁所同步着的代码块。

synchronized、ReentrantLock 都是可重入的锁，可重入锁相对来说简化了并发编程的开发

### 32、当一个线程进入某个对象的一个 synchronized 的实例方法后，其它线程是否可进入此对象的其它 方法？

如果其他方法没有 synchronized 的话，其他线程是可以进入的。

所以要开放一个线程安全的对象时，得保证每个方法都是线程安全的。

### 33、乐观锁和悲观锁的理解及如何实现，有哪些实现方式？

悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到 锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等， 读锁，写锁等，都是在做操作之前先上锁。再比如 Java 里面的同步原语 synchronized 关键字的实现也是悲观锁。

乐观锁：顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改， 所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。

乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于 write\_condition 机制，其实都是提供的乐观锁。在 Java 中 java.util.concurrent.atomic 包下面的原子变量类就是使用了乐观锁的一种实现方式 CAS 实现的。

---

乐观锁的实现方式：

1、使用版本标识来确定读到的数据与提交时的数据是否一致。提交后修改版本标识，不一致时可以采取丢弃和再次尝试的策略。

2、java 中的 Compare and Swap 即 CAS，当多个线程尝试使用 CAS 同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。CAS 操作中包含三个操作数——需要读写的内存位置 (V)、进行比较的预期原值

(A) 和拟写入的新值(B)。如果内存位置 V 的值与预期原值 A 相匹配，那么处理器会自动将该位置值更新为新值 B。否则处理器不做任何操作。

CAS 缺点：

ABA 问题：

比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但可能存在潜藏的问题。从 Java1.5 开始 JDK 的 atomic 包里提供了一个类 AtomicStampedReference 来解决 ABA 问题。

循环时间长开销大：

对于资源竞争严重（线程冲突严重）的情况，CAS 自旋的概率会比较大，从而浪费更多的 CPU 资源，效率低于 synchronized。

只能保证一个共享变量的原子操作：

当对一个共享变量执行操作时，我们可以使用循环 CAS 的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS 就无法保证操作的原子性，这个时候就可以用锁。

## 34、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为 map。ConcurrentHashMap 使用分段锁来保证在多线程下的性能。ConcurrentHashMap 中则是一次锁住一个桶。ConcurrentHashMap 默认将 hash 表分为 16 个桶，诸如 get,put,remove 等常用操作只锁当前需要用到的桶。这样，原来只能一个线程进入，现在却能同时有 16 个写线程执行，并发性能的提升是显而易见的。

另外 ConcurrentHashMap 使用了一种不同的迭代方式。在这种迭代方式中，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 新的数据从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。

## 35、CopyOnWriteArrayList 可以用于什么应用场景？

---

CopyOnWriteArrayList(免锁容器)的好处之一是当多个迭代器同时遍历和修改这个列表时，不会抛出 ConcurrentModificationException。在 CopyOnWriteArrayList 中，写入将导致创建整个底层数组的副本，而源数组将保留在原地，使得复制的数组在被修改时，读取操作可以安全地执行。

- 1、由于写操作的时候，需要拷贝数组，会消耗内存，如果原数组的内容比较多的情况下，可能导致 young gc 或者 full gc；
- 2、不能用于实时读的场景，像拷贝数组、新增元素都需要时间，所以调用一个 set 操作后，读取到数据可能还是旧的，虽然 CopyOnWriteArrayList 能做到最终一致性，但是还是没法满足实时性要求；

CopyOnWriteArrayList 透露的思想

- 1、读写分离，读和写分开
- 2、最终一致性
- 3、使用另外开辟空间的思路，来解决并发冲突

## 36、什么叫线程安全？servlet 是线程安全吗？

线程安全是编程中的术语，指某个函数、函数库在多线程环境中被调用时，能够正确地处理多个线程之间的共享变量，使程序功能正确完成。

Servlet 不是线程安全的，servlet 是单实例多线程的，当多个线程同时访问同一个方法，是不能保证共享变量的线程安全性的。

Struts2 的 action 是多实例多线程的，是线程安全的，每个请求过来都会 new 一个新的 action 分配给这个请求，请求完成后销毁。

SpringMVC 的 Controller 是线程安全的吗？不是的，和 Servlet 类似的处理流程。

Struts2 好处是不用考虑线程安全问题；Servlet 和 SpringMVC 需要考虑线程安全问题，但是性能可以提升不用处理太多的 gc，可以使用 ThreadLocal 来处理多线程的问题。

## 37、volatile 有什么用？能否用一句话说明下 volatile 的应用场景？

volatile 保证内存可见性和禁止指令重排。

volatile 用于多线程环境下的单次操作(单次读或者单次写)。

## 38、为什么代码会重排序？

在执行程序时，为了提供性能，处理器和编译器常常会对指令进行重排序，但是不能随意重排序，不是你想怎么排序就怎么排序，它需要满足以下两个条件：

在单线程环境下不能改变程序运行的结果；

存在数据依赖关系的不允许重排序

---

需要注意的是：重排序不会影响单线程环境的执行结果，但是会破坏多线程的执行语义。

## 39、在 Java 中 wait 和 sleep 方法的不同？

最大的不同是在等待时 wait 会释放锁，而 sleep 一直持有锁。Wait 通常被用于线程间交互，sleep 通常被用于暂停执行。

## 40、一个线程运行时发生异常会怎样？

如果异常没有被捕获该线程将会停止执行。Thread.UncaughtExceptionHandler 是用于处理未捕获异常造成线程突然中断情况的一个内嵌接口。当一个未捕获异常将造成线程中断的时候 JVM 会使用 Thread.getUncaughtExceptionHandler() 来查询线程的 UncaughtExceptionHandler 并将线程和异常作为参数传递给 handler 的 uncaughtException() 方法进行处理。

## 41、如何在两个线程间共享数据？

在两个线程间共享变量即可实现共享。

一般来说，共享变量要求变量本身是线程安全的，然后在线程内使用的时候，如果有对共享变量的复合操作，那么也得保证复合操作的线程安全性。

## 42、Java 中 notify 和 notifyAll 有什么区别？

notify() 方法不能唤醒某个具体的线程，所以只有一个线程在等待的时候它才有用武之地。而 notifyAll() 唤醒所有线程并允许他们争夺锁确保了至少有一个线程能继续运行。

## 43、为什么 wait, notify 和 notifyAll 这些方法不在 thread 类里面？

一个很明显的原因是 JAVA 提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。由于 wait, notify 和 notifyAll 都是锁级别的操作，所以把他们定义在 Object 类中因为锁属于对象。

## 44、什么是 ThreadLocal 变量？

---

ThreadLocal 是 Java 里一种特殊的变量。每个线程都有一个 ThreadLocal 就是每个线程都拥有了自己独立的一个变量，竞争条件被彻底消除了。它是为创建代价高昂的对象获取线程安全的好方法，比如你可以用 ThreadLocal 让 SimpleDateFormat 变成线程安全的，因为那个类创建代价高昂且每次调用都需要创建不同的实例所以不值得在局部范围使用它，如果为每个线程提供一个自己独有的变量拷贝，将大大提高效率。首先，通过复用减少了代价高昂的对象的创建个数。其次，你在没有使用高代价的同步或者不变性的情况下获得了线程安全。

## 45、Java 中 interrupted 和 isInterrupted 方法的区别？

interrupt

interrupt 方法用于中断线程。调用该方法的线程的状态为将被置为“中断”状态。

注意：线程中断仅仅是置线程的中断状态位，不会停止线程。需要用户自己去监视线程的状态为并做处理。支持线程中断的方法（也就是线程中断后会抛出 InterruptedException 的方法）就是在监视线程的中断状态，一旦线程的中断状态被置为“中断状态”，就会抛出中断异常。

interrupted

查询当前线程的中断状态，并且清除原状态。如果一个线程被中断了，第一次调用

interrupted 则返回 true，第二次和后面的就返回 false 了。isInterrupted

仅仅是查询当前线程的中断状态

## 46、为什么 wait 和 notify 方法要在同步块中调用？

Java API 强制要求这样做，如果你不这么做，你的代码会抛出

IllegalMonitorStateException 异常。还有一个原因是为了避免 wait 和 notify 之间产生竞态条件。

## 47、为什么你应该在循环中检查等待条件？

处于等待状态的线程可能会收到错误警报和伪唤醒，如果不在循环中检查等待条件，程序就会在没有满足结束条件的情况下退出。

## 48、Java 中的同步集合与并发集合有什么区别？

---

同步集合与并发集合都为多线程和并发提供了合适的线程安全的集合，不过并发集合的可扩展性更高。在 Java1.5 之前程序员们只有同步集合来用且在多线程并发的时候会导致争用，阻碍了系统的扩展性。Java5 介绍了并发集合像 ConcurrentHashMap，不仅提供线程安全还用锁分离和内部分区等现代技术提高了可扩展性。

## 49、什么是线程池？为什么要使用它？

创建线程要花费昂贵的资源和时间，如果任务来了才创建线程那么响应时间会变长，而且一个进程能创建的线程数有限。为了避免这些问题，在程序启动的时候就创建若干线程来响应处理，它们被称为线程池，里面的线程叫工作线程。从 JDK1.5 开始，Java API 提供了 Executor 框架让你可以创建不同的线程池。

## 50、怎么检测一个线程是否拥有锁？

在 java.lang.Thread 中有一个方法叫 holdsLock()，它返回 true 如果当且仅当当前线程拥有某个具体对象的锁。

## 51、你如何在 Java 中获取线程堆栈？

kill -3 [java pid]

不会在当前终端输出，它会输出到代码执行的或指定的地方去。比如，kill -3 tomcat pid，输出堆栈到 log 目录下。Jstack [java pid]

这个比较简单，在当前终端显示，也可以重定向到指定文件中。

-JvisualVM: Thread Dump

不做说明，打开 JvisualVM 后，都是界面操作，过程还是很简单的。

## 52、JVM 中哪个参数是用来控制线程的栈堆栈小的？

-Xss 每个线程的栈大小

---

## 53、Thread 类中的 yield 方法有什么作用？

使当前线程从执行状态（运行状态）变为可执行态（就绪状态）。

当前线程到了就绪状态，那么接下来哪个线程会从就绪状态变成执行状态呢？可能是当前线程，也可能是其他线程，看系统的分配了。

## 54、Java 中 ConcurrentHashMap 的并发度是什么？

ConcurrentHashMap 把实际 map 划分成若干部分来实现它的可扩展性和线程安全。这种划分是使用并发度获得的，它是 ConcurrentHashMap 类构造函数的一个可选参数，默认值为 16，这样在多线程情况下就能避免争用。

在 JDK8 后，它摒弃了 Segment（锁段）的概念，而是启用了一种全新的方式实现，利用 CAS 算法。同时加入了更多的辅助变量来提高并发度，具体内容还是查看源码吧。

## 55、Java 中 Semaphore 是什么？



Java 中的 Semaphore 是一种新的同步类，它是一个计数信号。从概念上讲，从概念上讲，信号量维护了一个许可集合。如有必要，在许可可用前会阻塞每一个 acquire()，然后再获取该许可。每个 release()添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，Semaphore 只对可用许可的号码进行计数，并采取相应的行动。信号量常用于多线程的代码中，比如数据库连接池。

## 56、Java 线程池中 submit() 和 execute()方法有什么区别？

两个方法都可以向线程池提交任务，execute()方法的返回类型是 void，它定义在 Executor 接口中。

而 submit()方法可以返回持有计算结果的 Future 对象，它定义在 ExecutorService 接口中，它扩展了 Executor 接口，其它线程池类像 ThreadPoolExecutor 和 ScheduledThreadPoolExecutor 都有这些方法。

## 57、什么是阻塞式方法？

---

阻塞式方法是指程序会一直等待该方法完成期间不做其他事情，`ServerSocket` 的 `accept()` 方法就是一直等待客户端连接。这里的阻塞是指调用结果返回之前，当前线程会被挂起，直到得到结果之后才会返回。此外，还有异步和非阻塞式方法在任务完成前就返回。

## 58、Java 中的 `ReadWriteLock` 是什么？

读写锁是用来提升并发程序性能的锁分离技术的成果。

## 59、`volatile` 变量和 `atomic` 变量有什么不同？

`Volatile` 变量可以确保先行关系，即写操作会发生在后续的读操作之前，但它并不能保证原子性。例如用 `volatile` 修饰 `count` 变量那么 `count++` 操作就不是原子性的。

而 `AtomicInteger` 类提供的 `atomic` 方法可以让这种操作具有原子性如 `getAndIncrement()` 方法会原子性的进行增量操作把当前值加一，其它数据类型和引用变量也可以进行相似操作。

## 60、可以直接调用 `Thread` 类的 `run()` 方法么？

当然可以。但是如果我们调用了 `Thread` 的 `run()` 方法，它的行为就会和普通的方法一样，会在当前线程中执行。为了在新的线程中执行我们的代码，必须使用 `Thread.start()` 方法。

## 61、如何让正在运行的线程暂停一段时间？

我们可以使用 `Thread` 类的 `Sleep()` 方法让线程暂停一段时间。需要注意的是，这并不会让线程终止，一旦从休眠中唤醒线程，线程的状态将会被改变为 `Runnable`，并且根据线程调度，它将得到执行。

## 62、你对线程优先级的理解是什么？

每一个线程都是有优先级的，一般来说，高优先级的线程在运行时会具有优先权，但这依赖于线程调度的实现，这个实现是和操作系统相关的(OS dependent)。我们可以定义线程的优先

---

级，但是这并不能保证高优先级的线程会在低优先级的线程前执行。线程优先级是一个 int 变量(从 1-10)，1 代表最低优先级，10 代表最高优先级。

java 的线程优先级调度会委托给操作系统去处理，所以与具体的操作系统优先级有关，如非特别需要，一般无需设置线程优先级。

## 63、什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing )？

线程调度器是一个操作系统服务，它负责为 Runnable 状态的线程分配 CPU 时间。一旦我们创建一个线程并启动它，它的执行便依赖于线程调度器的实现。同上一个问题，线程调度并不受到 Java 虚拟机控制，所以由应用程序来控制它是更好的选择（也就是说不要让你的程序依赖于线程的优先级）。

时间分片是指将可用的 CPU 时间分配给可用的 Runnable 线程的过程。分配 CPU 时间可以基于线程优先级或者线程等待的时间。

## 64、你如何确保 main()方法所在的线程是 Java 程序最后结束的线程？

我们可以使用 Thread 类的 join()方法来确保所有程序创建的线程在 main() 方法退出前结束。

## 65、线程之间是如何通信的？

当线程间是可以共享资源时，线程间通信是协调它们的重要的手段。Object 类中 wait()\notify()\notifyAll()方法可以用于线程间通信关于资源的锁的状态。

## 66、为什么线程通信的方法 wait(), notify()和 notifyAll()被定义在 Object 类里？

Java 的每个对象中都有一个锁(monitor，也可以成为监视器) 并且 wait(), notify()等方法用于等待对象的锁或者通知其他线程对象的监视器可用。在 Java 的线程中并没有可供任何对象使用的锁和同步器。这就是为什么这些方法是 Object 类的一部分，这样 Java 的每一个类都有用于线程间通信的基本方法。

---

## 67、为什么 `wait()`, `notify()`和 `notifyAll ()`必须在同步方法或者同步块中被调用？

当一个线程需要调用对象的 `wait()`方法的时候，这个线程必须拥有该对象的锁，接着它就会释放这个对象锁并进入等待状态直到其他线程调用这个对象上的 `notify()`方法。同样的，当一个线程需要调用对象的 `notify()`方法时，它会释放这个对象的锁，以便其他在等待的线程就可以得到这个对象锁。由于所有的这些方法都需要线程持有对象的锁，这样就只能通过同步来实现，所以他们只能在同步方法或者同步块中被调用。

## 68、为什么 `Thread` 类的 `sleep()`和 `yield ()`方法是静态的？

`Thread` 类的 `sleep()`和 `yield()`方法将在当前正在执行的线程上运行。所以在其他处于等待状态的线程上调用这些方法是没有意义的。这就是为什么这些方

法是静态的。它们可以在当前正在执行的线程中工作，并避免程序员错误的认为可以在其他非运行线程调用这些方法。

## 69、如何确保线程安全？



在 Java 中可以有很多方法来保证线程安全——同步，使用原子类(atomic concurrent classes)，实现并发锁，使用 `volatile` 关键字，使用不变类和线程安全类。

## 70、同步方法和同步块，哪个是更好的选择？

同步块是更好的选择，因为它不会锁住整个对象（当然你也可以让它锁住整个对象）。同步方法会锁住整个对象，哪怕这个类中有多个不相关联的同步块，这通常会导致他们停止执行并需要等待获得这个对象上的锁。

同步块更要符合开放调用的原则，只在需要锁住的代码块锁住相应的对象，这样从侧面来说也可以避免死锁。

## 71、如何创建守护线程？

---

使用 Thread 类的 setDaemon(true)方法可以将线程设置为守护线程，需要注意的是，需要在调用 start()方法前调用这个方法，否则会抛出 IllegalThreadStateException 异常。

## 72、什么是 Java Timer 类？如何创建一个有特定时间间隔的任务？

java.util.Timer 是一个工具类，可以用于安排一个线程在未来的某个特定时间执行。Timer 类可以用安排一次性任务或者周期任务。java.util.TimerTask 是一个实现了 Runnable 接口的抽象类，我们需要去继承这个类来创建我们自己的定时任务并使用 Timer 去安排它的执行。

作者：程序员追风

链接：<https://juejin.cn/post/6844904063687983111>

来源：掘金

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## MyBatis

### 1、什么是 Mybatis？

- Mybatis 是一个半 ORM（对象关系映射）框架，它内部封装了 JDBC，开发时只需要关注 SQL 语句本身，不需要花费精力去处理加载驱动、创建连接、创建 statement 等繁杂的过程。程序员直接编写原生态 sql，可以严格控制 sql 执行性能，灵活度高。
- MyBatis 可以使用 XML 或注解来配置和映射原生信息，将 POJO 映射成数据库中的记录，避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。
- 通过 xml 文件或注解的方式将要执行的各种 statement 配置起来，并通过 java 对象和 statement 中 sql 的动态参数进行映射生成最终执行的 sql 语句，最后由 mybatis 框架执行 sql 并将结果映射为 java 对象并返回。  
(从执行 sql 到返回 result 的过程)。

### 2、Mybatis 的优点：

- 基于 SQL 语句编程，相当灵活，不会对应用程序或者数据库的现有设计造成任何影响，SQL 写在 XML 里，解除 sql 与程序代码的耦合，便于统一管理；提供 XML 标签，支持编写动态 SQL 语句，并可重用。
- 与 JDBC 相比，减少了 50%以上的代码量，消除了 JDBC 大量冗余的代码，不需要手动开关连接；

- 
- 很好的与各种数据库兼容（因为 MyBatis 使用 JDBC 来连接数据库，所以只要 JDBC 支持的数据库 MyBatis 都支持）。
  - 能够与 Spring 很好的集成；
  - 提供映射标签，支持对象与数据库的 ORM 字段关系映射；提供对象关系映射标签，支持对象关系组件维护。

### 3、MyBatis 框架的缺点：

- SQL 语句的编写工作量较大，尤其当字段多、关联表多时，对开发人员编写 SQL 语句的功底有一定要求。
- SQL 语句依赖于数据库，导致数据库移植性差，不能随意更换数据库。

### 4、MyBatis 框架适用场合：

- MyBatis 专注于 SQL 本身，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，如互联网项目，MyBatis 将是不错的选择。

### 5、MyBatis 与 Hibernate 有哪些不同？

- Mybatis 和 hibernate 不同，它不完全是一个 ORM 框架，因为 MyBatis 需要程序员自己编写 Sql 语句。
- Mybatis 直接编写原生态 sql，可以严格控制 sql 执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，因为这类软件需求变化频繁，一但需求变化要求迅速输出成果。但是灵活的前提是 mybatis 无法做到数据库无关性，如果需要实现支持多种数据库的软件，则需要自定义多套 sql 映射文件，工作量大。
- Hibernate 对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件，如果用 hibernate 开发可以节省很多代码，提高效率。

### 6、#{ }和\${ }的区别是什么？

#{ }是预编译处理，\${ }是字符串替换。

Mybatis 在处理#{ }时，会将 sql 中的#{ }替换为?号，调用 PreparedStatement 的 set 方法来赋值；

Mybatis 在处理\${ }时，就是把\${ }替换成变量的值。使用#{ }可以有效的防止 SQL 注入，提高系统安全性。

## 7、当实体类中的属性名和表中的字段名不一样，怎么办？

第 1 种：通过在查询的 sql 语句中定义字段名的别名，让字段名的别名和实体类的属性名一致。

```
<select id="selectorder" parameterType="int" resultType="me.gacl.domain.order">
    select order_id id, order_no orderno ,order_price price from orders where
    order_id=#{id};
</select>
```

复制代码

第 2 种：通过<resultMap>来映射字段名和实体类属性名的一一对应的关系。

```
<select id="getOrder" parameterType="int" resultMap="orderresultmap">
    select * from orders where order_id=#{id}
</select>

<resultMap type="me.gacl.domain.order" id="orderresultmap" >
    <!- 用 id 属性来映射主键字段 ->
    <id property="id" column="order_id" >
        <!- 用 result 属性来映射非主键字段，property 为实体类属性名，column 为数据表中的属性 ->
        <result property="orderno" column="order_no" />
        <result property="price" column="order_price" />
    </resultMap>
```

复制代码

## 8、模糊查询 like 语句该怎么写？

第 1 种：在 Java 代码中添加 sql 通配符。string wildcardname = “%smi%”；

```
list<name> names = mapper.selectlike(wildcardname);
<select id="selectlike" >
    select * from foo where bar like #{value}
</select>
```

复制代码

第 2 种：在 sql 语句中拼接通配符，会引起 sql 注入 string wildcardname = “smi”；

```
list<name> names = mapper.selectlike(wildcardname);
<select id="selectlike" >
    select * from foo where bar like "%"#{value}%""
</select>
```

复制代码

---

## 9、通常一个 Xml 映射文件，都会写一个 Dao 接口与之对应，请问，这个 Dao 接口的工作原理是什么？Dao 接口里的方法，参数不同时，方法能重载吗？

Dao 接口即 Mapper 接口。接口的全限名，就是映射文件中的 namespace 的值；接口的方法名，就是映射文件中 Mapper 的 Statement 的 id 值；接口方法内的参数，就是传递给 sql 的参数。

Mapper 接口是没有实现类的，当调用接口方法时，接口全限名+方法名拼接字符串作为 key 值，可唯一定位一个 MapperStatement。在 Mybatis 中，每一个<select>、<insert>、<update>、<delete>标签，都会被解析为一个 MapperStatement 对象。

举例：com.mybatis3.mappers.StudentDao.findStudentById，可以唯一找到 namespace 为 com.mybatis3.mappers.StudentDao 下面 id 为 findStudentById 的 MapperStatement。

Mapper 接口里的方法，是不能重载的，因为是使用 全限名+方法名 的保存和寻找策略。

Mapper 接口的工作原理是 JDK 动态代理，Mybatis 运行时会使用 JDK 动态代理为 Mapper 接口生成代理对象 proxy，代理对象会拦截接口方法，转而执行 MapperStatement 所代表的 sql，然后将 sql 执行结果返回。



## 10、Mybatis 是如何进行分页的？分页插件的原理是什么？

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。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。

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，根据 dialect 方言，添加对应的物理分页语句和物理分页参数。

## 11、Mybatis 是如何将 sql 执行结果封装为目标对象并返回的？都有哪些映射形式？

第一种是使用<resultMap>标签，逐一定义数据库列名和对象属性名之间的映射关系。

第二种是使用 sql 列的别名功能，将列的别名书写为对象属性名。

有了列名与属性名的映射关系后，Mybatis 通过反射创建对象，同时使用反射给对象的属性逐一赋值并返回，那些找不到映射关系的属性，是无法完成赋值的。

---

## 12、如何执行批量插入？

首先,创建一个简单的 insert 语句:

```
<insert id=" insertname " >  
insert into names (name) values (#  
value  
)  
)
```

</insert>复制代码

然后在 java 代码中像下面这样执行批处理插入: list < string > names = new arraylist();  
names.add( “fred” );  
names.add( “barney” ); names.add( “betty” );  
names.add( “wilma” );  
// 注意这里 executortype.batch sqlsession sqlsession =  
sqlsessionfactory.opensession(executortype.batch); try {  
namemapper mapper = sqlsession.getmapper(namemapper.class); for (string name:  
names) {  
mapper.insertname(name);  
}  
sqlsession.commit();  
}  
catch (Exception e) {  
e.printStackTrace(); sqlSession.rollback(); throw e;  
}  
finally {  
sqlsession.close();  
}  
}复制代码

## 13、如何获取自动生成的(主)键值？

insert 方法总是返回一个 int 值，这个值代表的是插入的行数。

如果采用自增长策略，自动生成的键值在 insert 方法执行完后可以被设置到传入的参数对象中。示例：

```
<insert id=" insertname " usegeneratedkeys=" true " keyproperty="  
id " >
```

```
insert into names (name) values (#{  
    name  
}  
)  
</insert>  
name name = new name(); name.setname( "fred" );  
int rows = mapper.insertname(name);  
// 完成后,id 已经被设置到对象中 system.out.println( "rows inserted = " + rows);  
system.out.println( "generated key value = " + name.getId());复制代码
```

## 14、在 mapper 中如何传递多个参数?

第一种：DAO 层的函数

```
public UserselectUser(String name, String area);
```

对应的 xml,#{0}代表接收的是 dao 层中的第一个参数, #{1}代表 dao 层中第二参数, 更多参数一致往后加即可。

```
<select id="selectUser" resultMap="BaseResultMap"> select * from user_user_t  
where user_name = #{0} and user_area = #{1}
```

</select>复制代码

第二种： 使用 @param 注解:

```
public interface userMapper {
```

```
    User selectUser(@param("username") String
```

```
        username, @param("hashedpassword") String hashedpassword);
```

}复制代码

然后,就可以在 xml 像下面这样使用(推荐封装为一个 map,作为单个参数传递给 mapper):

```
<select id="selectUser" resultType="User" > select id, username, hashedpassword  
from some_table
```

```
where username = #{username}
```

```
and hashedpassword = #{hashedpassword}
```

</select>复制代码

第三种：多个参数封装成 map try {

```
// 映射文件的命名空间.SQL 片段的 ID, 就可以调用对应的映射文件
```

中的

SQL

```
// 由于我们的参数超过了两个, 而方法中只有一个 Object 参数收  
集, 因此
```

我们使用 Map 集合来装载我们的参数

```
Map < String, Object > map = new HashMap();
map.put("start", start);
map.put("end", end);
return sqlSession.selectList("StudentID.pagination", map);
}
catch (Exception e) {
e.printStackTrace(); sqlSession.rollback(); throw e;
}
finally {
MybatisUtil.closeSqlSession();
}复制代码
```

## 15、Mybatis 动态 sql 有什么用？执行原理？有哪些动态 sql？

Mybatis 动态 sql 可以在 Xml 映射文件内，以标签的形式编写动态 sql，执行原理是根据表达式的值 完成逻辑判断并动态拼接 sql 的功能。

Mybatis 提供了 9 种动态 sql 标签：trim | where | set | foreach | if | choose| when | otherwise | bind。



## 16、Xml 映射文件中，除了常见的 select|insert|update|delete 标签之外，还有哪些标签？

答：<resultMap>、<parameterMap>、<sql>、<include>、<selectKey>，加上动态 sql 的 9 个标签，其中<sql>为 sql 片段标签，通过<include>标签引入 sql 片段，<selectKey>为不支持自增的主键生成策略标签。

## 17、Mybatis 的 Xml 映射文件中，不同的 Xml 映射文件，id 是否可以重复？

不同的 Xml 映射文件，如果配置了 namespace，那么 id 可以重复；如果没有配置 namespace，那么 id 不能重复；

原因就是 namespace+id 是作为 Map<String, MapperStatement>的 key 使用的，如果没有 namespace，就剩下 id，那么，id 重复会导致数据互相覆盖。有了 namespace，自然 id 就可以重复，namespace 不同，namespace+id 自然也就不同。

## 18、为什么说 Mybatis 是半自动 ORM 映射工具？它与全自动的区别在哪里？

---

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。

## 19、一对一、一对多的关联查询？

```
<mapper namespace="com.lcb.mapping.userMapper">
<!--association 一对一关联查询 -->
<select id="getClass" parameterType="int" resultMap="ClassesResultMap">
select * from class c,teacher t where c.teacher_id=t.t_id and c.c_id=#{id}
</select>
<resultMap type="com.lcb.user.Classes" id="ClassesResultMap">
<!-- 实体类的字段名和数据表的字段名映射 -->
<id property="id" column="c_id"/>
<result property="name" column="c_name"/>
<association property="teacher" javaType="com.lcb.user.Teacher">
<id property="id" column="t_id"/>
<result property="name" column="t_name"/>
</association>
</resultMap>
<!--collection 一对多关联查询 -->
<select id="getClass2" parameterType="int" resultMap="ClassesResultMap2">
select * from class c,teacher t,student s where c.teacher_id=t.t_id
and c.c_id=s.class_id and c.c_id=#{id}
</select>
<resultMap type="com.lcb.user.Classes" id="ClassesResultMap2">
<id property="id" column="c_id"/>
<result property="name" column="c_name"/>
<association property="teacher" javaType="com.lcb.user.Teacher">
<id property="id" column="t_id"/>
<result property="name" column="t_name"/>
</association>
<collection property="student" ofType="com.lcb.user.Student">
<id property="id" column="s_id"/>
<result property="name" column="s_name"/>
</collection>
</resultMap>
```

---

</mapper>复制代码

## 20、MyBatis 实现一对一有几种方式?具体怎么操作的?

有联合查询和嵌套查询,联合查询是几个表联合查询,只查询一次,通过在 resultMap 里面配置 association 节点配置一对一的类就可以完成; 嵌套查询是先查一个表,根据这个表里面的 结果的 外键 id,去再另外一个表里面查询数据,也是通过 association 配置,但另外一个表的查询通过 select 属性配置。

## 21、MyBatis 实现一对多有几种方式,怎么操作的?

有联合查询和嵌套查询。联合查询是几个表联合查询,只查询一次,通过在 resultMap 里面的 collection 节点配置一对多的类就可以完成; 嵌套查询是先查一个表,根据这个表里面的 结果的外键 id,去再另外一个表里面查询数据,也是通过配置 collection,但另外一个表的查询通过 select 节点配置。

## 22、Mybatis 是否支持延迟加载?如果支持,它的实现原理是什么?

答: 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。

它的原理是, 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。这就是延迟加载的基本原理。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。

## 23、Mybatis 的一级、二级缓存:

- 一级缓存: 基于 PerpetualCache 的 HashMap 本地缓存, 其存储作用域为 Session, 当 Session flush 或 close 之后, 该 Session 中的所有 Cache 就将清空, 默认打开一级缓存。
- 二级缓存与一级缓存其机制相同, 默认也是采用 PerpetualCache, HashMap 存储, 不同在于其存储作用域为 Mapper(Namespace), 并且可自定义存储源, 如 Ehcache。默认不打开二级缓存, 要开启二级缓存, 使用二级缓存属性类需要实现 Serializable 序列化接口(可用来保存对象的状态),可在它的映射文件中配置<cache> ;

- 
- 对于缓存数据更新机制，当某一个作用域(一级缓存 Session/二级缓存 Namespaces)的进行了 C/U/D 操作后，默认该作用域下所有 select 中的缓存将被 clear。

## 24、什么是 MyBatis 的接口绑定？有哪些实现方式？

接口绑定，就是在 MyBatis 中任意定义接口,然后把接口里面的方法和 SQL 语句绑定, 我们直接调用接口方法就可以,这样比起原来来了 SqlSession 提供的方法我们可以有更加灵活的选择和设置。

接口绑定有两种实现方式,一种是通过注解绑定，就是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；另外一种就是通过 xml 里面写 SQL 来绑定，在这种情况下,要指定 xml 映射文件里面的 namespace 必须为接口的全路径名。当 Sql 语句比较简单时候,用注解绑定, 当 SQL 语句比较复杂时候,用 xml 绑定,一般用 xml 绑定的比较多。

## 25、使用 MyBatis 的 mapper 接口调用时有哪些要求？

- Mapper 接口方法名和 mapper.xml 中定义的每个 sql 的 id 相同；
- Mapper 接口方法的输入参数类型和 mapper.xml 中定义的每个 sql 的 parameterType 的类型相同；
- Mapper 接口方法的输出参数类型和 mapper.xml 中定义的每个 sql 的 resultType 的类型相同；
- Mapper.xml 文件中的 namespace 即是 mapper 接口的类路径。

## 26、Mapper 编写有哪几种方式？

第一种：接口实现类继承 SqlSessionDaoSupport：使用此种方法需要编写 mapper 接口，mapper 接口实现类、mapper.xml 文件。

(1) 在 sqlMapConfig.xml 中配置 mapper.xml 的位置

```
<mappers>
<mapper resource="mapper.xml 文件的地址" />
<mapper resource="mapper.xml 文件的地址" />
</mappers>
```

复制代码

(2) 定义 mapper 接口

(3) 实现类集成 SqlSessionDaoSupport

mapper 方法中可以 this.getSqlSession() 进行数据增删改查。

(4) spring 配置

```
<bean id=" " class="mapper 接口的实现">
<property name="sqlSessionFactory" ref="sqlSessionFactory"></property>
</bean>
```

复制代码

---

第二种：使用 org.mybatis.spring.mapper.MapperFactoryBean：

(1) 在 sqlMapConfig.xml 中配置 mapper.xml 的位置，如果 mapper.xml 和 mappe 接口的名称相同且在同一个目录，这里可以不用配置

```
<mappers>
<mapper resource="mapper.xml 文件的地址" />
<mapper resource="mapper.xml 文件的地址" />
</mappers>复制代码
```

(2) 定义 mapper 接口：

(3) mapper.xml 中的 namespace 为 mapper 接口的地址

(4) mapper 接口中的方法名和 mapper.xml 中的定义的 statement 的 id 保持一致

(5) Spring 中定义

```
<bean id="" class="org.mybatis.spring.mapper.MapperFactoryBean">
<property name="mapperInterface" value="mapper 接口地址" />
<property name="sqlSessionFactory" ref="sqlSessionFactory" />
</bean>复制代码
```

第三种：使用 mapper 扫描器：

(1) mapper.xml 文件编写：

mapper.xml 中的 namespace 为 mapper 接口的地址；

mapper 接口中的方法名和 mapper.xml 中的定义的 statement 的 id 保持一致；

如果将 mapper.xml 和 mapper 接口的名称保持一致则不用在 sqlMapConfig.xml 中进行配置。

(2) 定义 mapper 接口：

注意 mapper.xml 的文件名和 mapper 的接口名称保持一致，且放在同一个目录

(3) 配置 mapper 扫描器：

```
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
<property name="basePackage" value="mapper 接口包地址
"></property>
<property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>
</bean>复制代码
```

(4) 使用扫描器后从 spring 容器中获取 mapper 的实现对象。

## 27、简述 Mybatis 的插件运行原理，以及如何编写一个插件。

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。编写插件：实现 Mybatis 的 Interceptor 接口并复写 intercept()方法，然后在给插件编写注解，指定要拦截哪一个接口的哪些方法即可，记住，别忘了在配置文件中配置你编写的插件。

作者：程序员追风

链接：<https://juejin.cn/post/6844904040380235784> 来源：掘金

著作版权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## Redis

### 1、什么是 Redis?

Redis 是完全开源免费的，遵守 BSD 协议，是一个高性能的 key-value 数据库。

**Redis 与其他 key - value 缓存产品有以下三个特点：**

- Redis 支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。
- Redis 不仅仅支持简单的 key-value 类型的数据，同时还提供 list, set, zset, hash 等数据结构的存储。
- Redis 支持数据的备份，即 master-slave 模式的数据备份。

**Redis 优势**

- 性能极高 – Redis 能读的速度是 110000 次/s,写的速度是 81000 次 /s 。
- 丰富的数据类型 – Redis 支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。
- 原子 – Redis 的所有操作都是原子性的，意思就是要么成功执行要么失败完全不执行。单个操作是原子性的。多个操作也支持事务，即原子性，通过 MULTI 和 EXEC 指令包起来。
- 丰富的特性 – Redis 还支持 publish/subscribe, 通知, key 过期等等特性。

**Redis 与其他 key-value 存储有什么不同？**

- Redis 有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。Redis 的数据类型都是基于基本数据结构的同时对程序员透明，无需进行额外的抽象。
- Redis 运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，因为数据量不能大于硬件内存。在内存数据库方面的另一个优点是，相比在磁盘

---

上相同的复杂的数据结构，在内存中操作起来非常简单，这样 Redis 可以做很多内部复杂性很强的事情。同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

## 2、Redis 的数据类型？

Redis 支持五种数据类型：string（字符串），hash（哈希），list（列表），set（集合）及 zsetsorted set：有序集合）。

我们实际项目中比较常用的是 string，hash 如果你是 Redis 中高级用户，还需要加上下面几种数据结构 HyperLogLog、Geo、Pub/Sub。

如果说还玩过 Redis Module，像 BloomFilter，RedisSearch，Redis-ML，面试官得眼睛就开始发亮了。

## 3、使用 Redis 有哪些好处？

- 速度快，因为数据存在内存中，类似于 HashMap，HashMap 的优势就是查找和操作的时间复杂度都是 O1)
- 支持丰富数据类型，支持 string，list，set，Zset，hash 等
- 支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行
- 丰富的特性：可用于缓存，消息，按 key 设置过期时间，过期后将会自动删除

## 4、Redis 相比 Memcached 有哪些优势？

- Memcached 所有的值均是简单的字符串，redis 作为其替代者，支持更为丰富的数据类型
- Redis 的速度比 Memcached 快很多
- Redis 可以持久化其数据

## 5、Memcache 与 Redis 的区别都有哪些？

- 存储方式 Memecache 把数据全部存在内存之中，断电后会挂掉，数据不能超过内存大小。Redis 有部分存在硬盘上，这样能保证数据的持久性。
- 数据支持类型 Memcache 对数据类型支持相对简单。Redis 有复杂的数据类型。

- 
- 使用底层模型不同 它们之间底层实现方式 以及与客户端之间通信的应用 协议不一样。Redis 直接自己构建了 VM 机制，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。

## 6、Redis 是单进程单线程的？

Redis 是单进程单线程的，redis 利用队列技术将并发访问变为串行访问，消除了传统数据库串行控制的开销。

### 7、一个字符串类型的值能存储最大容量是多少？

答：512M

## 8、Redis 的持久化机制是什么？各自的优缺点？

Redis 提供两种持久化机制 RDB 和 AOF 机制：

**RDBRedis DataBase)持久化方式：**

是指用数据集快照的方式半持久化模式)记录 redis 数据库的所有键值对,在某个时间点将数据写入一个临时文件，持久化结束后，用这个临时文件替换上次 持久化的文件，达到数据恢复。

优点：

- 只有一个文件 dump.rdb，方便持久化。
- 容灾性好，一个文件可以保存到安全的磁盘。
- 性能最大化，fork 子进程来完成写操作，让主进程继续处理命令，所以是 IO 最大化。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)
- 相对于数据集大时，比 AOF 的启动效率更高。

缺点：

数据安全性低。RDB 是间隔一段时间进行持久化，如果持久化之间 redis 发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候

**AOFAppend-only file)持久化方式：**

是指所有的命令行记录以 redis 命令请求协议的格式完全持久化存储)保存为 aof 文件。

优点：

- 数据安全，aof 持久化可以配置 appendfsync 属性，有 always，每进行一次命令操作就记录到 aof 文件中一次。
- 通过 append 模式写文件，即使中途服务器宕机，可以通过 redis- check-aof 工具解决数据一致性问题。
- AOF 机制的 rewrite 模式。AOF 文件没被 rewrite 之前（文件过大时会 对命令进行合并重写），可以删除其中的某些命令（比如误操作的 flushall））

---

缺点：

- AOF 文件比 RDB 文件大，且恢复速度慢。
- 数据集大的时候，比 rdb 启动效率低。

## 9、Redis 常见性能问题和解决方案：

- (1) Master 最好不要写内存快照，如果 Master 写内存快照，save 命令调度 rdbSave 函数，会阻塞主线程的工作，当快照比较大时对性能影响是非常大的，会间断性暂停服务
- (2) 如果数据比较重要，某个 Slave 开启 AOF 备份数据，策略设置为每秒同步
- (3) 为了主从复制的速度和连接的稳定性，Master 和 Slave 最好在同一个局域网
- (4) 尽量避免在压力很大的主库上增加从
- (5) 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3…这样的结构方便解决单点故障问题，实现 Slave 对 Master 的替换。如果 Master 挂了，可以立刻启用 Slave1 做 Master，其他不变。

## 10、redis 过期键的删除策略？

- 
- (1) 定时删除：在设置键的过期时间的同时，创建一个定时器 timer。让定时器在键的过期时间来临时，立即执行对键的删除操作。
  - (2) 惰性删除：放任键过期不管，但是每次从键空间中获取键时，都检查取得的键是否过期，如果过期的话，就删除该键；如果没有过期，就返回该键。
  - (3) 定期删除：每隔一段时间程序就对数据库进行一次检查，删除里面的过期键。至于要删除多少过期键，以及要检查多少个数据库，则由算法决定。

## 11、Redis 的回收策略（淘汰策略）？

volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰

volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）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（server.db[i].expires）中任意选择数据淘汰

allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰

allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰

no-eviction（驱逐）：禁止驱逐数据

---

注意这里的 6 种机制，volatile 和 allkeys 规定了是对已设置过期时间的数据集淘汰数据还是从全部数据集淘汰数据，后面的 lru、ttl 以及 random 是三种不同的淘汰策略，再加上一种 no-eviction 永不回收的策略。

使用策略规则：

(1) 如果数据呈现幂律分布，也就是一部分数据访问频率高，一部分数据访问频率低，则使用 allkeys-lr

(2) 如果数据呈现平等分布，也就是所有的数据访问频率都相同，则使用 allkeys-random

## 12、为什么 Redis 需要把所有数据放到内存中？

Redis 为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以 redis 具有快速和数据持久化的特征。如果不将数据放在内存中，磁盘 I/O 速度为严重影响 redis 的性能。在内存越来越便宜的今天，redis 将会越来越受欢迎。如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。

## 13、Redis 的同步机制了解么？

Redis 可以使用主从同步，从从同步。第一次同步时，主节点做一次 bgsave，并同时将后续修改操作记录到内存 buffer，待完成后将 rdb 文件全量同步到复制节点，复制节点接受完成后将 rdb 镜像加载到内存。加载完成后，再通知主节点将期间修改的操作记录同步到复制节点进行重放就完成了同步过程。

## 14、Pipeline 有什么好处，为什么要用 pipeline？

可以将多次 IO 往返的时间缩减为一次，前提是 pipeline 执行的指令之间没有因果相关性。使用 redis-benchmark 进行压测的时候可以发现影响 redis 的 QPS 峰值的一个重要因素是 pipeline 批次指令的数目。

## 15、是否使用过 Redis 集群，集群的原理是什么？

(1) Redis Sentinel 着眼于高可用，在 master 宕机时会自动将 slave 提升为 master，继续提供服务。

(2) Redis Cluster 着眼于扩展性，在单个 redis 内存不足时，使用 Cluster 进行分片存储。

## 16、Redis 集群方案什么情况下会导致整个集群不可用？

---

有 A, B, C 三个节点的集群,在没有复制模型的情况下,如果节点 B 失败了,那么整个集群就会以为缺少 5501-11000 这个范围的槽而不可用。

## 17、Redis 支持的 Java 客户端都有哪些? 官方推荐用哪个?

Redisson、Jedis、lettuce 等等,官方推荐使用 Redisson。

## 18、Jedis 与 Redisson 对比有什么优缺点?

Jedis 是 Redis 的 Java 实现的客户端,其 API 提供了比较全面的 Redis 命令的支持; Redisson 实现了分布式和可扩展的 Java 数据结构,和 Jedis 相比,功能较为简单,不支持字符串操作,不支持排序、事务、管道、分区等 Redis 特性。 Redisson 的宗旨是促进使用者对 Redis 的关注分离,从而让使用者能够将精力更集中地放在处理业务逻辑上。

## 19、Redis 如何设置密码及验证密码?

设置密码: config set requirepass 123456

授权密码: auth 123456

## 20、说说 Redis 哈希槽的概念?

Redis 集群没有使用一致性 hash,而是引入了哈希槽的概念,Redis 集群有 16384 个哈希槽,每个 key 通过 CRC16 校验后对 16384 取模来决定放置哪个槽,集群的每个节点负责一部分 hash 槽。

## 21、Redis 集群的主从复制模型是怎样的?

为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用,所以集群使用了主从复制模型,每个节点都会有 N-1 个复制品。

## 22、Redis 集群会有写操作丢失吗?为什么?

Redis 并不能保证数据的强一致性,这意味着在实际中集群在特定的条件下可能会丢失写操作。

## 23、Redis 集群之间是如何复制的?

---

异步复制

## 24、Redis 集群最大节点个数是多少？

16384 个。

## 25、Redis 集群如何选择数据库？

Redis 集群目前无法做数据库选择， 默认在 0 数据库。

## 26、怎么测试 Redis 的连通性

使用 ping 命令。

## 27、怎么理解 Redis 事务？

- (1) 事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。
- (2) 事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

## 28、Redis 事务相关的命令有哪几个？

MULTI、EXEC、DISCARD、WATCH

## 29、Redis key 的过期时间和永久有效分别怎么设置？

EXPIRE 和 PERSIST 命令。

## 30、Redis 如何做内存优化？

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。

## 31、Redis 回收进程如何工作的？

---

一个客户端运行了新的命令，添加了新的数据。Redis 检查内存使用情况，如果大于 maxmemory 的限制，则根据设定好的策略进行回收。一个新的命令被执行，等等。所以我们不断地穿越内存限制的边界，通过不断达到边界然后不断地收回回到边界以下。如果一个命令的结果导致大量内存被使用（例如很大的集合的交集保存到一个新的键），不用多久内存限制就会被这个内存使用量超越。

### 32、都有哪些办法可以降低 Redis 的内存使用情况呢？

如果你使用的是 32 位的 Redis 实例，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在一起。

### 33、Redis 的内存用完了会发生什么？

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。）或者你可以将 Redis 当缓存来使用配置淘汰机制，当 Redis 达到内存上限时会冲刷掉旧的内容。

### 34、一个 Redis 实例最多能存放多少的 keys？List、Set、Sorted Set 他们最多能存放多少元素？

理论上 Redis 可以处理多达 232 的 keys，并且在实际中进行了测试，每个实例至少存放了 2 亿 5 千万的 keys。我们正在测试一些较大的值。任何 list、set、和 sorted set 都可以放 232 个元素。换句话说，Redis 的存储极限是系统中的可用内存值。

### 35、MySQL 里有 2000w 数据，redis 中只存 20w 的数据，如何保证 redis 中的数据都是热点数据？Redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。相关知识：Redis 提供 6 种数据淘汰策略：

volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰

---

volatile-ttl: 从已设置过期时间的数据集 (server.db[i].expires) 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 (server.db[i].expires) 中任 意选择数据淘汰  
allkeys-lru: 从数据集 (server.db[i].dict) 中挑选最近最少使用的数据淘 汰  
allkeys-random: 从数据集 (server.db[i].dict) 中任意选择数据淘汰  
no-eviction (驱逐) : 禁止驱逐数据

## 36、Redis 最适合的场景？

### 会话缓存 (Session Cache)

最常用的一种使用 Redis 的情景是会话缓存 (session cache)。用 Redis 缓 存会话比其他存储 (如 Memcached) 的优势在于: Redis 提供持久化。当维护 一个不是严格要求一致性的缓存时, 如果用户的购物车信息全部丢失, 大部分 人都会不高兴的, 现在, 他们还会这样吗? 幸运的是, 随着 Redis 这些年的 改进, 很容易找到怎么恰当的使用 Redis 来缓存会话的文档。甚至广为人知的 商业平台 Magento 也提供 Redis 的插件。

### 全页缓存 (FPC)

除基本的会话 token 之外, Redis 还提供很简便的 FPC 平台。回到一致性问 题, 即使重启了 Redis 实例, 因为有磁盘的持久化, 用户也不会看到页面加载 速度的下降, 这是一个极大改进, 类似 PHP 本地 FPC。再次以 Magento 为 例, Magento 提供一个插件来使用 Redis 作为全页缓存后端。此外, 对 WordPress 的用户来说, Pantheon 有一个非常好的插件 wp-redis, 这个插件 能帮助你以最快速度加载你曾浏览过的页面。

### 队列

Reids 在内存存储引擎领域的一大优点是提供 list 和 set 操作, 这使得 Redis 能作为一个很 好的消息队列平台来使用。Redis 作为队列使用的操作, 就类似于本地程序语言 (如 Python) 对 list 的 push/pop 操作。如果你快 速的在 Google 中搜索 “Redis queues”, 你马上就能找到大量的开源项目, 这些项目的目的就是利用 Redis 创建非常好的后端工具, 以满足各 种队列需 求。例如, Celery 有一个后台就是使用 Redis 作为 broker, 你可以从这里去 查看。

### 排行榜/计数器

Redis 在内存中对数字进行递增或递减的操作实现的非常好。集合 (Set) 和有 序集合 (Sorted Set) 也使得我们在执行这些操作的时候变的非常简单, Redis 只是正好提供了这两 种数据结构。所以, 我们要从排序集合中获取到排名最靠 前的 10 个用户 – 我们称之为 “user\_scores”, 我们只需要像下面一样执行即 可: 当然, 这是假定你是根据你用户的分 数做递增的排序。如果你想返回用户 及用户的分数, 你需要这样执行: ZRANGE

---

user\_scores 0 10 WITHSCORES Agora Games 就是一个很好的例子，用 Ruby 实现的，它的排行榜就是使用 Redis 来存储数据的，你可以在这里看到。

## 发布/订阅

最后（但肯定不是最不重要的）是 Redis 的发布/订阅功能。发布/订阅的使用 场景确实非常多。我已看见人们在社交网络连接中使用，还可作为基于发布/订阅的脚本触发器，甚至用 Redis 的发布/订阅功能来建立聊天系统！

## 37、假如 Redis 里面有 1 亿个 key，其中有 10w 个 key 是以某个固定的已知的前缀开头的，如果将 它们全部找出来？

使用 keys 指令可以扫出指定模式的 key 列表。

对方接着追问：如果这个 redis 正在给线上的业务提供服务，那使用 keys 指令会有什么问题？

这个时候你要回答 redis 关键的一个特性：redis 的单线程的。keys 指令会 导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用 scan 指令，scan 指令可以无阻塞的提取出指定模式的 key 列表，但是会有一定的重复概率，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用 keys 指令长。

## 38、如果有大量的 key 需要设置同一时间过期，一般需要注意什么？

如果大量的 key 过期时间设置的过于集中，到过期的那个时间点，redis 可能会出现短暂的卡顿现象。一般需要在时间上加一个随机值，使得过期时间 分散一些。

## 39、使用过 Redis 做异步队列么，你是怎么用的？

一般使用 list 结构作为队列，rpush 生产消息，lpop 消费消息。当 lpop 没有消息的时候，要适当 sleep 一会再重试。如果对方追问可不可以不用 sleep 呢？list 还有个指令叫 blpop，在没有消息的时候，它会阻塞住直 到消息到来。如果对方追问能不能生产一次消费多次呢？使用 pub/sub 主题订 阅者模式，可以实现 1:N 的消息队列。

如果对方追问 pub/sub 有什么缺点？

在消费者下线的情况下，生产的消息会丢失，得使用专业的消息队列如 RabbitMQ 等。

如果对方追问 redis 如何实现延时队列？

我估计现在你很想把面试官一棒打死如果你手上有一根棒球棍的话，怎么问的 这么详细。但是你很克制，然后神态自若的回答道：使用 sortedset，拿时间 截作为 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。到这里，面试官暗地里已经对你竖起了大拇指。但是他不知道的是此刻你却竖起了中指，在椅子背后。

## 40、使用过 Redis 分布式锁么，它是什么回事

先拿 `setnx` 来争抢锁，抢到之后，再用 `expire` 给锁加一个过期时间防止锁忘记释放。这时候对方会告诉你说你回答得不错，然后接着问如果在 `setnx` 之后执行 `expire` 之前进程意外 `crash` 或者要重启维护了，那会怎么样？这时候你要给予惊讶的反馈：唉，是喔，这个锁就永远得不到释放了。紧接着你需要抓一抓自己的脑袋，故作思考片刻，好像接下来的结果是你主动思考出来的，然后回答：我记得 `set` 指令有非常复杂的参数，这个应该是可以同时把 `setnx` 和 `expire` 合成一条指令来用的！对方这时会显露笑容，心里开始默念：嗯，这小子还不错。

作者：程序员追风

链接：<https://juejin.cn/post/6844904006121160711>

来源：掘金

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## Spring

## 1、不同版本的 Spring Framework 有哪些主要功能？

Version	Feature
Spring 2.5	发布于 2007 年。这是第一个支持注解的版本。
Spring 3.0	发布于 2009 年。它完全利用了 Java5 中的改进，并为 JEE6 提供了支持。
Spring 4.0	发布于 2013 年。这是第一个完全支持 Java8 的版本。

## 2、什么是 Spring Framework？

---

Spring 是一个开源应用框架，旨在降低应用程序开发的复杂度。它是轻量级、松散耦合的。它具有分层体系结构，允许用户选择组件，同时还为 J2EE 应用程序开发提供了一个有凝聚力的框架。它可以集成其他框架，如 Structs、Hibernate、EJB 等，所以又称为框架的框架。

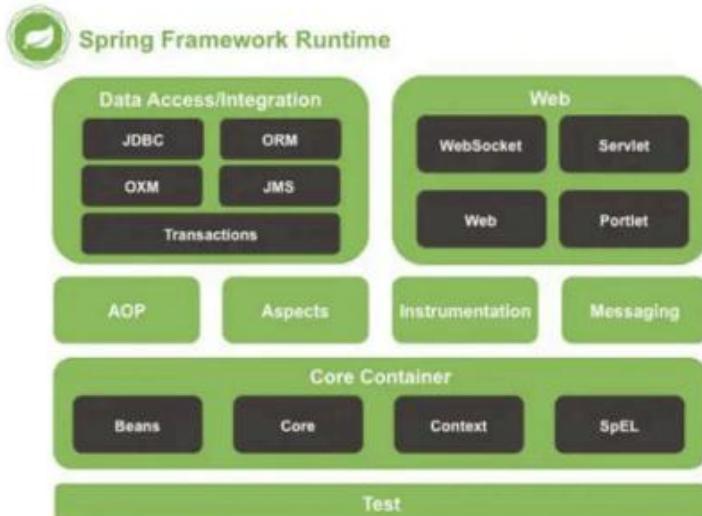
### 3、列举 Spring Framework 的优点。

由于 Spring Framework 的分层架构，用户可以自由选择自己需要的组件。Spring Framework 支持 POJO(Plain Old Java Object) 编程，从而具备持续集成和可测试性。由于依赖注入和控制反转，JDBC 得以简化。它是开源免费的。

### 4、Spring Framework 有哪些不同的功能？

轻量级 - Spring 在代码量和透明度方面都很轻便。IOC - 控制反转 AOP - 面向切面编程 可以将应用业务逻辑和系统服务分离，以实现高内聚。容器 - Spring 负责创建和管理对象 (Bean) 的生命周期和配置。MVC - 对 web 应用提供了高度可配置性，其他框架的集成 也十分方便。事务管理 - 提供了用于事务管理的通用抽象层。Spring 的事务支持也可用于 容器较少的环境。JDBC 异常 - Spring 的 JDBC 抽象层提供了一个异常层次结构，简化了 错误处理策略。

### 5、Spring Framework 中有多少个模块，它们分别是什么？



Spring 核心容器 - 该层基本上是 Spring Framework 的核心。它包含以下模块：

- Spring Core
- Spring Bean
- SpEL (Spring Expression Language)
- Spring Context

---

数据访问/集成 – 该层提供与数据库交互的支持。它包含以下模块：

- JDBC (Java DataBase Connectivity)
- ORM (Object Relational Mapping)
- OXM (Object XML Mappers) · JMS (Java Messaging Service)
- Transaction

Web – 该层提供了创建 Web 应用程序的支持。它包含以下模块：

- Web
- Web – Servlet
- Web – Socket
- Web – Portlet

AOP

- 该层支持面向切面编程

Instrumentatio

- 该层为类检测和类加载器实现提供支持。

Test

- 该层为使用 JUnit 和 TestNG 进行测试提供支持。

几个杂项模块: Messaging – 该模块为 STOMP 提供支持。它还支持注解编程模型，该模型用于从 WebSocket 客户端路由和处理 STOMP 消息。Aspects – 该模块为与 AspectJ 的集成提供支持。

## 6、什么是 Spring 配置文件？

Spring 配置文件是 XML 文件。该文件主要包含类信息。它描述了这些类是如何配置以及相互引入的。但是，XML 配置文件冗长且更加干净。如果没有正确规划和编写，那么在大项目中管理变得非常困难。

## 7、Spring 应用程序有哪些不同组件？

- Spring 应用一般有以下组件：
- 接口 – 定义功能。
- Bean 类 – 它包含属性，setter 和 getter 方法，函数等。 · Spring 面向切面编程 (AOP) – 提供面向切面编程的功能。 · Bean 配置文件 – 包含类的信息以及如何配置它们。
- 用户程序 – 它使用接口。

## 8、使用 Spring 有哪些方式？

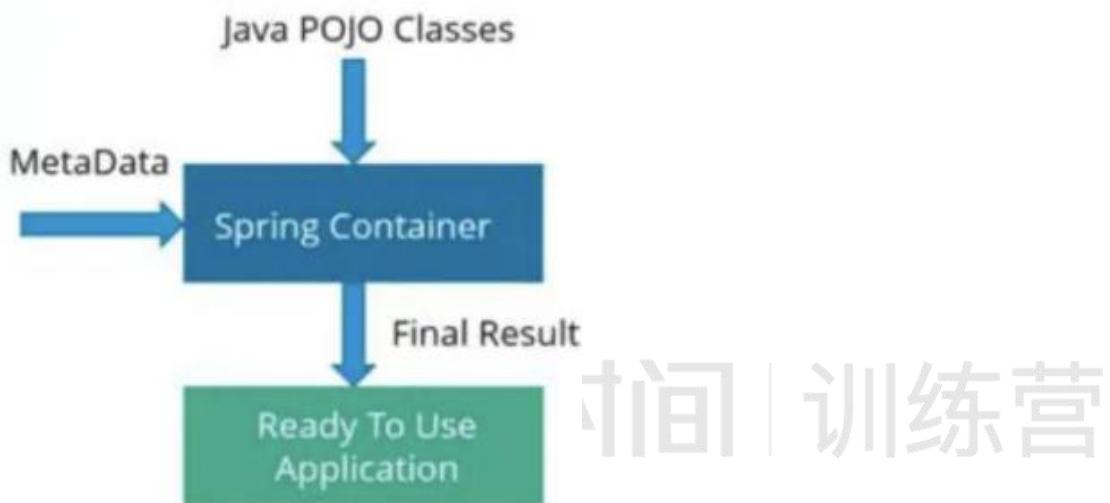
---

使用 Spring 有以下方式：

- 作为一个成熟的 Spring Web 应用程序。
- 作为第三方 Web 框架，使用 Spring Frameworks 中间层。 · 用于远程使用。
- 作为企业级 Java Bean，它可以包装现有的 POJO (Plain Old JavaObjects)。

## 9、什么是 Spring IOC 容器？

Spring 框架的核心是 Spring 容器。容器创建对象，将它们装配在一起，配置它们并管理它们的完整生命周期。Spring 容器使用依赖注入来管理组成应用程序的组件。容器通过读取提供的配置元数据来接收对象进行实例化，配置和组装的指令。该元数据可以通过 XML，Java 注解或 Java 代码提供。



## 10、什么是依赖注入？

在依赖注入中，您不必创建对象，但必须描述如何创建它们。您不是直接在代码 中将组件 和服务连接在一起，而是描述配置文件中哪些组件需要哪些服务。由 IoC 容器将它们装配在一起。

## 11、可以通过多少种方式完成依赖注入？

通常，依赖注入可以通过三种方式完成，即：

- 构造函数注入
- setter 注入
- 接口注入 在 Spring Framework 中，仅使用构造函数和 setter 注入

## 12、区分构造函数注入和 setter 注入

构造函数注入	setter 注入
没有部分注入	有部分注入
不会覆盖 setter 属性	会覆盖 setter 属性
任意修改都会创建一个新实例	任意修改不会创建一个新实例
适用于设置很多属性	适用于设置少量属性

## 13、Spring 中有多少种 IOC 容器？

BeanFactory - BeanFactory 就像一个包含 bean 集合的工厂类。它会在客户端要求时实例化 bean。 ApplicationContext - ApplicationContext 接口扩展了 BeanFactory 接口。它在 BeanFactory 基础上提供了一些额外的功能。

## 14、区分 BeanFactory 和 ApplicationContext。

BeanFactory	ApplicationContext
它使用懒加载	它使用即时加载
它使用语法显式提供资源对象	它自己创建和管理资源对象
不支持国际化	支持国际化
不支持基于依赖的注解	支持基于依赖的注解

## 15、列举 IoC 的一些好处。

IoC 的一些好处是：

- 它将最小化应用程序中的代码量。
- 它将使您的应用程序易于测试，因为它不需要单元测试用例中的任何单例或 JNDI 查找机制。
- 它以最小的影响和最少的侵入机制促进松耦合。
- 它支持即时的实例化和延迟加载服务。

## 16、Spring IoC 的实现机制。

Spring 中的 IoC 的实现原理就是工厂模式加反射机制。示例：

```
interface Fruit {  
    public abstract void eat();  
}  
  
class Apple implements Fruit {  
    public void eat(){  
        System.out.println("Apple");  
    }  
}  
  
class Orange implements Fruit {  
    public void eat(){  
        System.out.println("Orange");  
    }  
}  
  
class Factory {  
    public static Fruit getInstance(String ClassName) {  
        Fruit f=null;  
        try {  
            f=(Fruit)Class.forName(ClassName).newInstance();  
        }  
        catch (Exception e) {  
            e.printStackTrace();  
        }  
        return f;  
    }  
}  
  
class Client {  
    public static void main(String[] a) {  
        Fruit  
        f=Factory.getInstance("io.github.dunwu.spring.Apple");  
        if(f!=null){  
            f.eat();  
        }  
    }  
}
```

---

}复制代码

## 17、什么是 Spring Bean？

- 它们是构成用户应用程序主干的对象。
- Bean 由 Spring IoC 容器管理。
- 它们由 Spring IoC 容器实例化，配置，装配和管理。
- Bean 是基于用户提供给容器的配置元数据创建。

## 18、Spring 提供了哪些配置方式？

基于 xml 配置 bean 所需的依赖项和服务在 XML 格式的配置文件中指定。这些配置文件通常包含许多 bean 定义和特定于应用程序的配置选项。它们通常以 bean 标签开头。例如：

```
<bean id="studentbean" class="org.edureka.firstSpring.StudentBean">
<property name="name" value="Edureka"></property>
</bean>
```

基于注解配置 您可以通过在相关的类，方法或字段声明上使用注解，将 bean 配置为组件类本身，而不是使用 XML 来描述 bean 装配。默认情况下，Spring 容器中未打开注解装配。因此，您需要在使用它之前在 Spring 配置文件中启用它。例如：

```
<beans>
<context:annotation-config/>
<!-- bean definitions go here -->
</beans>
```

基于 Java API 配置

Spring 的 Java 配置是通过使用 @Bean 和 @Configuration 来实现。（1）@Bean 注解扮演与 <bean/> 元素相同的角色。（2）@Configuration 类允许通过简单地调用同一个类中的其他 @Bean 方法来定义 bean 间依赖关系。例如：

```
@Configuration
public class StudentConfig {
    @Bean
    public StudentBean myStudent() {
        return new StudentBean();
    }
}
```

## 19、Spring 支持集中 bean scope？

---

Spring bean 支持 5 种 scope： Singleton - 每个 Spring IoC 容器仅有一个单实例。 Prototype - 每次请求都会产生一个新的实例。 Request - 每一次 HTTP 请求都会产生一个新的实例，并且该 bean 仅在当前 HTTP 请求内有效。 Session - 每一次 HTTP 请求都会产生一个新的 bean，同时该 bean 仅在当前 HTTP session 内有效。 Global-session - 类似于标准的 HTTP Session 作用域，不过它仅仅在基于 portlet 的 web 应用中才有意义。 Portlet 规范定义了全局 Session 的概念，它被所有构成某个 portlet web 应用的各种不同的 portlet 所共享。在 globalSession 作用域中定义的 bean 被限定于全局 portlet Session 的生命周期范围内。如果你在 web 中使用 global session 作用域来标识 bean，那么 web 会自动当成 session 类型来使用。仅当用户使用支持 Web 的 ApplicationContext 时，最后三个才可用。

## 20、Spring Bean 容器的生命周期是什么样的

spring bean 容器的生命周期流程如下：

- (1) Spring 容器根据配置中的 bean 定义中实例化 bean。
- (2) Spring 使用依赖注入填充所有属性，如 bean 中所定义的配置。
- (3) 如果 bean 实现 BeanNameAware 接口，则工厂通过传递 bean 的 ID 来调用 setBeanName()。
- (4) 如果 bean 实现 BeanFactoryAware 接口，工厂通过传递自身的实例来调用 setBeanFactory()。
- (5) 如果存在与 bean 关联的任何 BeanPostProcessors，则调用 preProcessBeforeInitialization() 方法。
- (6) 如果为 bean 指定了 init 方法（<bean> 的 init-method 属性），那么将调用它。
- (7) 最后，如果存在与 bean 关联的任何 BeanPostProcessors，则将调用 postProcessAfterInitialization() 方法。
- (8) 如果 bean 实现 DisposableBean 接口，当 spring 容器关闭时，会调用 destory()。
- (9) 如果为 bean 指定了 destroy 方法（<bean> 的 destroy-method 属性），那么将调用它。

## 21、什么是 Spring 的内部 Bean？

只有将 bean 用作另一个 bean 的属性时，才能将 bean 声明为内部 bean。为了定义 bean，Spring 的基于 XML 的配置元数据在 <property> 或 <constructor-arg> 中提供了 <bean> 元素的使用。内部 bean 总是匿名的，它们总是作为原型。例如，假设我们有一个 Student 类，其中引用了 Person 类。这里我们将只创建一个 Person 类实例并在 Student 中使用它。

---

## Student.java

```
public class Student {  
    private Person person;  
    //Setters and Getters  
}
```

```
public class Person {  
    private String name;  
    private String address;  
    //Setters and Getters  
}
```

复制代码

bean.xml

```
<bean id=“StudentBean” class="com.edureka.Student">  
<property name="person">  
    <!--This is inner bean -->  
    <bean class="com.edureka.Person">  
        <property name="name" value=“Scott”></property>  
        <property name="address" value=“Bangalore”></property>  
    </bean>  
</property>  
</bean>
```

## 22、什么是 Spring 装配

当 bean 在 Spring 容器中组合在一起时，它被称为装配或 bean 装配。Spring 容器需要知道需要什么 bean 以及容器应该如何使用依赖注入来将 bean 绑定在一起，同时装配 bean。

## 23、自动装配有哪些方式？

Spring 容器能够自动装配 bean。也就是说，可以通过检查 BeanFactory 的内容让 Spring 自动解析 bean 的协作者。自动装配的不同模式：

no - 这是默认设置，表示没有自动装配。应使用显式 bean 引用进行装配。 byName - 它根据 bean 的名称注入对象依赖项。它匹配并装配其属性与 XML

---

文件中由相同名称定义的 bean。

byType - 它根据类型注入对象依赖项。如果属性的类型与 XML 文件中的一个 bean 名称匹配，则匹配并装配属性。构造函数- 它通过调用类的构造函数来注入依赖项。它有大量的参数。

autodetect - 首先容器尝试通过构造函数使用 autowire 装配，如果不能，则尝试通过 byType 自动装配。

## 24、自动装配有什么局限？

覆盖的可能性 - 您始终可以使用 <constructor-arg> 和 <property> 设置指定依赖项，这将覆盖自动装配。基本元数据类型 - 简单属性（如原数据类型，字符串和类）无法自动装配。令人困惑的性质 - 总是喜欢使用明确的装配，因为自动装配不太精确。

## 25、什么是基于注解的容器配置

不使用 XML 来描述 bean 装配，开发人员通过在相关的类，方法或字段声明上使用注解将配置移动到组件类本身。它可以作为 XML 设置的替代方案。例如：Spring 的 Java 配置是通过使用 @Bean 和 @Configuration 来实现。@Bean 注解扮演与元素相同的角色。@Configuration 类允许通过简单地调用同一个类中的其他 @Bean 方法来定义 bean 间依赖关系。例如：

```
@Configuration  
public class StudentConfig {  
    @Bean  
    public StudentBean myStudent() {  
        return new StudentBean();  
    }  
}
```

[复制代码](#)

## 26、如何在 Spring 中启动注解装配？

默认情况下，Spring 容器中未打开注解装配。因此，要使用基于注解装配，我们必须通过配置 <context: annotation-config/> 元素在 Spring 配置文件中启用它。

## 27、@Component, @Controller, @Repository, @Service 有何区别

---

@Component : 这将 java 类标记为 bean。它是任何 Spring 管理组件的通用构造型。spring 的组件扫描机制现在可以将其拾取并将其拉入应用程序环境中。

@Controller : 这将一个类标记为 Spring Web MVC 控制器。标有它的 Bean 会自动导入到 IoC 容器中。

@Service : 此注解是组件注解的特化。它不会对 @Component 注解提供任何其他行为。您可以在服务层类中使用 @Service 而不是 @Component，因为它以更好的方式指定了意图。

@Repository : 这个注解是具有类似用途和功能的 @Component 注解的特化。它为 DAO 提供了额外的好处。它将 DAO 导入 IoC 容器，并使未经检查的异常有资格转换为 Spring DataAccessException。

## 28、@Required 注解有什么用？

@Required 应用于 bean 属性 setter 方法。此注解仅指示必须在配置时使用 bean 定义中的显式属性值或使用自动装配填充受影响的 bean 属性。如果尚未填充受影响的 bean 属性，则容器将抛出 BeanInitializationException。示例：

```
public class Employee {  
    private String name;  
  
    @Required  
    public void setName(String name){  
        this.name=name;  
    }  
    public string getName(){  
        return name;  
    }  
}
```

[复制代码](#)

## 29、@Autowired 注解有什么用？

@Autowired 可以更准确地控制应该在何处以及如何进行自动装配。此注解用于在 setter 方法，构造函数，具有任意名称或多个参数的属性或方法上自动装配 bean。默认情况下，它是类型驱动的注入。

```
public class Employee {  
    private String name;  
    @Autowired
```

```
public void setName(String name) {  
    this.name=name;  
}  
public string getName(){  
    return name;  
}  
}
```

}  
}复制代码

### 30、@Qualifier 注解有什么用？

当您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，您可以使用 @Qualifier 注解和 @Autowired 通过指定应该装配哪个确切的 bean 来消除歧义。例如，这里我们分别有两个类，Employee 和 EmpAccount。在 EmpAccount 中，使用@Qualifier 指定了必须装配 id 为 emp1 的 bean。

Employee.java

```
public class Employee {  
    private String name;  
    @Autowired  
    public void setName(String name) {  
        this.name=name;  
    }  
    public string getName() {  
        return name;  
    }  
}
```

}  
}复制代码

EmpAccount.java

```
public class EmpAccount {  
    private Employee emp;  
    @Autowired  
    @Qualifier(emp1)  
    public void showName() {  
    }
```

```
System.out.println( "Employee name : " +emp.getName); }  
}复制代码
```

### 31、@RequestMapping 注解有什么用？

@RequestMapping 注解用于将特定 HTTP 请求方法映射到将处理相应请求的控制器中的特定类/方法。此注释可应用于两个级别：类级别：映射请求的 URL 方法级别：映射 URL 以及 HTTP 请求方法

### 32、Spring DAO 有什么用？

Spring DAO 使得 JDBC，Hibernate 或 JDO 这样的数据访问技术更容易以一种统一的方式工作。这使得用户容易在持久性技术之间切换。它还允许您在编写代码时，无需考虑捕获每种技术不同的异常。

### 33、列举 Spring DAO 抛出的异常。



训练营

### 34、Spring JDBC API 中存在哪些类？

- JdbcTemplate
- SimpleJdbcTemplate
- NamedParameterJdbcTemplate
- SimpleJdbcInsert
- SimpleJdbcCall

### 35、使用 Spring 访问 Hibernate 的方法有哪些？

---

我们可以通过两种方式使用 Spring 访问 Hibernate:

- 使用 Hibernate 模板和回调进行 控制反转
- 扩展 HibernateDAOsupport 并应用 AOP 拦截器节点

## 36、列举 spring 支持的事务管理类型

Spring 支持两种类型的事务管理:

- 程序化事务管理: 在此过程中, 在编程的帮助下管理事务。它为您提供极大的灵活性, 但维护起来非常困难。
- 声明式事务管理: 在此, 事务管理与业务代码分离。仅使用注解或基于 XML 的配置来管理事务。

## 37、Spring 支持哪些 ORM 框架

- Hibernate
- iBatis
- JPA
- JDO
- OJB



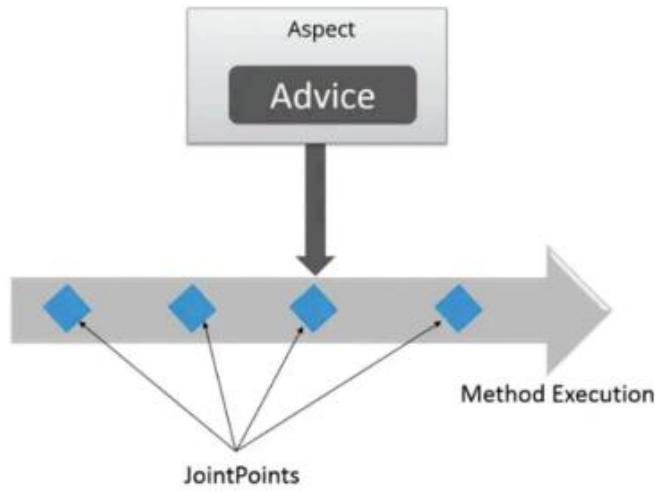
## 38、什么是 AOP?

AOP(Aspect-Oriented Programming), 即 面 向 切 面 编 程 , 它 与 OOP( Object-Oriented Programming, 面 向 对 象 编 程 ) 相 辅 相 成, 提 供 了 与 OOP 不 同 的 抽 象 软 件 结 构 的 视 角 . 在 OOP 中, 我 们 以 类 (class) 作 为 我 们 的 基 本 单 元, 而 AOP 中 的 基 本 单 元 是 Aspect(切面)

## 39、什么是 Aspect?

Aspect 由 pointcut 和 advice 组成, 它 既 包 含 了 横 切 逻 辑 的 定 义, 也 包 括 了 连 接 点 的 定 义 . Spring AOP 就 是 负 责 实 施 切 面 的 框 架, 它 将 切 面 所 定 义 的 横 切 逻 辑 编 织 到 切 面 所 指 定 的 连 接 点 中 . AOP 的 工 作 重 心 在 于 如 何 将 增 强 编 织 目 标 对 象 的 连 接 点 上, 这 里 包 含 两 个 工 作:

- 如何通过 pointcut 和 advice 定位到特定的 joinpoint 上
- 如何在 advice 中编写切面代码.



可以简单地认为，使用 @Aspect 注解的类就是切面。

## 40、什么是切点（JoinPoint）

程序运行中的一些时间点，例如一个方法的执行，或者是一个异常的处理。在 Spring AOP 中，join point 总是方法的执行点。

## 41、什么是通知（Advice）？

特定 JoinPoint 处的 Aspect 所采取的动作称为 Advice。Spring AOP 使用一个 Advice 作为拦截器，在 JoinPoint “周围” 维护一系列的拦截器。

## 42、有哪些类型的通知（Advice）？

- Before – 这些类型的 Advice 在 joinpoint 方法之前执行，并使用 @Before 注解标记进行配置。
- After Returning – 这些类型的 Advice 在连接点方法正常执行后执行，并使用 @AfterReturning 注解标记进行配置。
- After Throwing – 这些类型的 Advice 仅在 joinpoint 方法通过抛出异常退出并使用 @AfterThrowing 注解标记配置时执行。
- After (finally) – 这些类型的 Advice 在连接点方法之后执行，无论方法退出是正常还是异常返回，并使用 @After 注解标记进行配置。
- Around – 这些类型的 Advice 在连接点之前和之后执行，并使用 @Around 注解标记进行配置。

---

## 43、指出在 Spring AOP 中 concern 和 cross-cuttingconcern 的不同之处。

concern 是我们想要在应用程序的特定模块中定义的行为。它可以定义为我们想要实现的功能。cross-cutting concern 是一个适用于整个应用的行为，这会影响整个应用程序。例如，日志记录，安全性和数据传输是应用程序几乎每个模块都需要关注的问题，因此它们是跨领域的问题。

## 44、AOP 有哪些实现方式？

实现 AOP 的技术，主要分为两大类：

静态代理 指使用 AOP 框架提供的命令进行编译，从而在编译阶段就可生成 AOP 代理类，因此也称为编译时增强；

- 编译时编织（特殊编译器实现）
- 类加载时编织（特殊的类加载器实现）。

动态代理 在运行时在内存中“临时”生成 AOP 动态代理类，因此也被称为运行时增强。

JDK 动态代理

CGLIB

## 45、Spring AOP and AspectJ AOP 有什么区别？

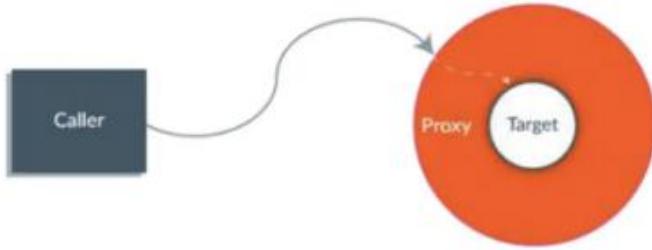
Spring AOP 基于动态代理方式实现；AspectJ 基于静态代理方式实现。SpringAOP 仅支持方法级别的 PointCut；提供了完全的 AOP 支持，它还支持属性级别的 PointCut。

## 46、如何理解 Spring 中的代理？

将 Advice 应用于目标对象后创建的对象称为代理。在客户端对象的情况下，目标对象和代理对象是相同的。Advice + Target Object = Proxy

## 47、什么是编织（Weaving）？

为了创建一个 advice 对象而链接一个 aspect 和其它应用类型或对象，称为编织（Weaving）。在 Spring AOP 中，编织在运行时执行。请参考下图：



## 48、Spring MVC 框架有什么用？

Spring Web MVC 框架提供 模型-视图-控制器 架构和随时可用的组件，用于开发灵活且松散耦合的 Web 应用程序。MVC 模式有助于分离应用程序的不同方面，如输入逻辑，业务 逻辑和 UI 逻辑，同时在所有这些元素之间提供松散耦合。

## 49、描述一下 DispatcherServlet 的工作流程

DispatcherServlet 的工作流程可以用一幅图来说明：



- (1) 向服务器发送 HTTP 请求，请求被前端控制器 DispatcherServlet 捕获。
- (2) DispatcherServlet 根据 -servlet.xml 中的配置对请求的 URL 进行解 析，得到请求资源标识符（URI）。然后根据该 URI，调用 HandlerMapping 获 得该 Handler 配置的所有相关的对象（包括 Handler 对象以及 Handler 对象 对应的拦截器），最后以 HandlerExecutionChain 对象的形式返回。
- (3) DispatcherServlet 根据获得的 Handler，选择一个合适的 HandlerAdapter。（附注：如果成功获得 HandlerAdapter 后，此时将开始执 行拦截器的 preHandler(...)方法）。

---

(4) 提取 Request 中的模型数据，填充 Handler 入参，开始执行 Handler ( Controller)。在填充 Handler 的入参过程中，根据你的配置，Spring 将帮你做一些额外的工作：  
· HttpMessageConvester: 将请求消息（如 Json、xml 等数据）转换成一个对象，将对象转换为指定的响应信息。  
· 数据转换：对请求消息进行数据转换。如 String 转换成 Integer、Double 等。  
· 数据格式化：对请求消息进行数据格式化。如将字符串转换成格式化数字或格式化日期等。  
· 数据验证：验证数据的有效性（长度、格式等），验证结果存储到 BindingResult 或 Error 中。

(5) Handler(Controller)执行完成后，向 DispatcherServlet 返回一个 ModelAndView 对象；

(6) 根据返回的 ModelAndView，选择一个适合的 ViewResolver（必须是已经注册到 Spring 容器中的 ViewResolver)返回给 DispatcherServlet。(7) ViewResolver 结合 Model 和 View，来渲染视图。

(8) 视图负责将渲染结果返回给客户端。

## 50、介绍一下 WebApplicationContext

WebApplicationContext 是 ApplicationContext 的扩展。它具有 Web 应用程序所需的一些额外功能。它与普通的 ApplicationContext 在解析主题和决定与哪个 servlet 关联的能力方面有所不同。

## 51、什么是 Spring?

Spring 是个 java 企业级应用的开源开发框架。Spring 主要用来开发 Java 应用，但是有些扩展是针对构建 J2EE 平台的 web 应用。Spring 框架目标是简化 Java 企业级应用开发，并通过 POJO 为基础的编程模型促进良好的编程习惯。

## 52、使用 Spring 框架的好处是什么？

- 轻量：Spring 是轻量的，基本的版本大约 2MB。
- 控制反转：Spring 通过控制反转实现了松散耦合，对象们给出它们的依赖，而不是创建或查找依赖的对象们。
- 面向切面的编程(AOP)：Spring 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。
- 容器：Spring 包含并管理应用中对象的生命周期和配置。

- 
- MVC 框架：Spring 的 WEB 框架是个精心设计的框架，是 Web 框架的一个很好的替代品。
  - 事务管理：Spring 提供一个持续的事务管理接口，可以扩展到上至本地事务下至全局事务（JTA）。
  - 异常处理：Spring 提供方便的 API 把具体技术相关的异常（比如由 JDBC, Hibernate or JDO 抛出的）转化为一致的 unchecked 异常。

## 53、Spring 由哪些模块组成？

以下是 Spring 框架的基本模块：

- Core module · Bean module · Context module
- Expression Language module
- JDBC module · ORM module · OXM module
- Java Messaging Service(JMS) module
- Transaction module · Web module · Web-Servlet module
- Web-Struts module · Web-Portlet module

## 54、核心容器（应用上下文）模块。

这是基本的 Spring 模块，提供 spring 框架的基础功能，BeanFactory 是任何以 spring 为基础的应用的核心。Spring 框架建立在此模块之上，它使 Spring 成为一个容器。

## 55、BeanFactory – BeanFactory 实现举例。

Bean 工厂是工厂模式的一个实现，提供了控制反转功能，用来把应用的配置和依赖从真正的应用代码中分离。最常用的 BeanFactory 实现是 XmlBeanFactory 类。

## 66、XMLBeanFactory

最常用的就是 org.springframework.beans.factory.xml.XmlBeanFactory，它根据 XML 文件中的定义加载 beans。该容器从 XML 文件读取配置元数据并用它去创建一个完全配置的系统或应用。

## 67、解释 AOP 模块

---

AOP 模块用于发给我们的 Spring 应用做面向切面的开发，很多支持由 AOP 联盟提供，这样就确保了 Spring 和其他 AOP 框架的共通性。这个模块将元数据编程引入 Spring。

## 68、解释 JDBC 抽象和 DAO 模块。

通过使用 JDBC 抽象和 DAO 模块，保证数据库代码的简洁，并能避免数据库资源错误关闭导致的问题，它在各种不同的数据库的错误信息之上，提供了一个统一的异常访问层。它还利用 Spring 的 AOP 模块给 Spring 应用中的对象提供事务管理服务。

## 69、解释对象/关系映射集成模块。

Spring 通过提供 ORM 模块，支持我们在直接 JDBC 之上使用一个对象 / 关系映射映射 (ORM) 工具，Spring 支持集成主流的 ORM 框架，如 Hibernate, JDO 和 iBATIS SQL Maps。Spring 的事务管理同样支持以上所有 ORM 框架及 JDBC。

## 70、解释 WEB 模块。

Spring 的 WEB 模块是构建在 application context 模块基础之上，提供一个适合 web 应用的上下文。这个模块也包括支持多种面向 web 的任务，如透明地处理多个文件上传请求和程序级请求参数的绑定到你的业务对象。它也有对 JakartaStruts 的支持。

## 72、Spring 配置文件

Spring 配置文件是个 XML 文件，这个文件包含了类信息，描述了如何配置它们，以及如何相互调用。

## 73、什么是 Spring IOC 容器

Spring IOC 负责创建对象，管理对象（通过依赖注入（DI），装配对象，配置对象，并且管理这些对象的整个生命周期）。

## 74、IOC 的优点是什么？

---

IOC 或 依赖注入把应用的代码量降到最低。它使应用容易测试，单元测试不再需要单例和 JNDI 查找机制。最小的代价和最小的侵入性使松散耦合得以实现。IOC 容器支持加载服务 时的饿汉式初始化和懒加载。

## 75、ApplicationContext 通常的实现是什么？

- FileSystemXmlApplicationContext：此容器从一个 XML 文件中加载 beans 的定义， XML Bean 配置文件的全路径名必须提供给它的构造函数。
- ClassPathXmlApplicationContext：此容器也从一个 XML 文件中加载 beans 的定义， 这里，你需要正确设置 classpath 因为这个容器将在 classpath 里找 bean 配置。
- WebXmlApplicationContext：此容器加载一个 XML 文件，此文件定义了一个 WEB 应用的 所有 bean。

## 76、Bean 工厂和 Application contexts 有什么区别？

Application contexts 提供一种方法处理文本消息，一个通常的做法是加载文件资源（比如镜像），它们可以向注册为监听器的 bean 发布事件。另外，在容器或容器内的对象上执行的那些不得不由 bean 工厂以程序化方式处理的操作，可以在 Application contexts 中以声明的方式处理。Application contexts 实现了 MessageSource 接口，该接口的实现以可插拔的方式提供获取本地化消息的方法。

## 77、一个 Spring 的应用看起来象什么？

- 一个定义了一些功能的接口。
- 这实现包括属性，它的 Setter，getter 方法和函数等。
- Spring AOP。 · Spring 的 XML 配置文件。
- 使用以上功能的客户端程序。依赖注入

## 78、什么是 Spring 的依赖注入？

依赖注入，是 IOC 的一个方面，是个通常的概念，它有多种解释。这概念是说你不用创建 对象，而只需要描述它如何被创建。你不在代码里直接组装你的组件和服务，但是要在配置 文件里描述哪些组件需要哪些服务，之后一个容器（IOC 容器）负责把他们组装起来。

## 79、有哪些不同类型的 IOC（依赖注入）方式？

- 
- 构造器依赖注入：构造器依赖注入通过容器触发一个类的构造器来实现的，该类有一系列参数，每个参数代表一个对其他类的依赖。
  - Setter 方法注入：Setter 方法注入是容器通过调用无参构造器或无参 static 工厂方法实例化 bean 之后，调用该 bean 的 setter 方法，即实现了基于 setter 的依赖注入。

## 80、哪种依赖注入方式你建议使用，构造器注入，还是 Setter 方法注入？

你两种依赖方式都可以使用，构造器注入和 Setter 方法注入。最好的解决方案是用构造器参数实现强制依赖，setter 方法实现可选依赖。

## 81、什么是 Spring Beans？

Spring beans 是那些形成 Spring 应用的主干的 java 对象。它们被 Spring IOC 容器初始化，装配，和管理。这些 beans 通过容器中配置的元数据创建。比如，以 XML 文件中的形式定义。Spring 框架定义的 beans 都是单件 beans。在 bean tag 中有个属性“singleton”，如果它被赋为 TRUE，bean 就是单件，否则就是一个 prototype bean。默认是 TRUE，所以所有在 Spring 框架中的 beans 缺省都是单件。

## 82、一个 Spring Bean 定义包含什么？

一个 Spring Bean 的定义包含容器必知的所有配置元数据，包括如何创建一个 bean，它的生命周期详情及它的依赖。

## 83、如何给 Spring 容器提供配置元数据？

这里有三种重要的方法给 Spring 容器提供配置元数据。

- XML 配置文件。
- 基于注解的配置。
- 基于 java 的配置。

## 84、你怎样定义类的作用域？

当定义一个在 Spring 里，我们还能给这个 bean 声明一个作用域。它可以通过 bean 定义中的 scope 属性来定义。如，当 Spring 要在需要的时候每次生产一个新的 bean 实例，

---

bean 的 scope 属性被指定为 prototype。另一方面，一个 bean 每次使用的时候必须返回同一个实例，这个 bean 的 scope 属性必须设为 singleton。

## 85、解释 Spring 支持的几种 Bean 的作用域

Spring 框架支持以下五种 bean 的作用域：

- singleton : bean 在每个 Spring ioc 容器中只有一个实例。
- prototype: 一个 bean 的定义可以有多个实例。
- request : 每次 http 请求都会创建一个 bean , 该作用域仅在基于 web 的 Spring ApplicationContext 情形下有效。
- session: 在一个 HTTP Session 中，一个 bean 定义对应一个实例。该作用域仅在基于 web 的 Spring ApplicationContext 情形下有效。
- global-session: 在一个全局的 HTTP Session 中，一个 bean 定义对应一个实例。该作用域仅在基于 web 的 Spring ApplicationContext 情形下有效。缺省的 Spring bean 的作用域是 Singleton。

## 86、Spring 框架中的单例 Bean 是线程安全的吗？

不，Spring 框架中的单例 bean 不是线程安全的。

## 87、解释 Spring 框架中 Bean 的生命周期。

- Spring 容器从 XML 文件中读取 bean 的定义，并实例化 bean。
- Spring 根据 bean 的定义填充所有的属性。
- 如果 bean 实现了 BeanNameAware 接口，Spring 传递 bean 的 ID 到 setBeanName 方法。
- 如果 Bean 实现了 BeanFactoryAware 接口，Spring 传递 beanfactory 给 setBeanFactory 方法。
- 如果有任何与 bean 相关联的 BeanPostProcessors，Spring 会在 postProcesserBeforeInitialization()方法内调用它们。
- 如果 bean 实现 IntializingBean 了，调用它的 afterPropertySet 方法，如果 bean 声明了 初始化方法，调用此初始化方法。
- 如果有 BeanPostProcessors 和 bean 关联，这些 bean 的 postProcessAfterInitialization() 方法将被调用。
- 如果 bean 实现了 DisposableBean，它将调用 destroy()方法。

---

## 88、哪些是重要的 Bean 生命周期方法？你能重载它们吗？

有两个重要的 bean 生命周期方法，第一个是 setup ， 它是在容器加载 bean 的时候被调用。第二个方法是 teardown 它是在容器卸载类的时候被调用。

The bean 标签有两个重要的属性（init-method 和 destroy-method）。用它们你可以自己定制初始化和注销方法。它们也有相应的注解（@PostConstruct 和@PreDestroy）。

## 89、什么是 Spring 的内部 Bean？

当一个 bean 仅被用作另一个 bean 的属性时，它能被声明为一个内部 bean，为了定义 inner bean，在 Spring 的基于 XML 的配置元数据中，可以在或元素内使用元素，内部 bean 通常是匿名的，它们的 Scope 一般是 prototype。

## 90、在 Spring 中如何注入一个 Java 集合？

Spring 提供以下几种集合的配置元素：

- 类型用于注入一列值，允许有相同的值。
- 类型用于注入一组值，不允许有相同的值。
- 类型用于注入一组键值对，键和值都可以为任意类型。
- 类型用于注入一组键值对，键和值都只能为 String 类型。

## 91、什么是 Bean 装配？

装配，或 bean 装配是指在 Spring 容器中把 bean 组装到一起，前提是容器需要知道 bean 的依赖关系，如何通过依赖注入来把它们装配到一起。

## 92、什么是 Bean 的自动装配？

Spring 容器能够自动装配相互合作的 bean，这意味着容器不需要手动配置，能通过 Bean 工厂自动处理 bean 之间的协作。

## 93、解释不同方式的自动装配

有五种自动装配的方式，可以用来指导 Spring 容器用自动装配方式进行依赖注入。

- no：默认的方式是不进行自动装配，通过显式设置 ref 属性来进行装配。

- 
- byName: 通过参数名 自动装配, Spring 容器在配置文件中发现 bean 的 autowire 属性被设置成 byname , 之后容器试图匹配、装配和该 bean 的属性 具有相同名字的 bean。
  - byType:: 通过参数类型自动装配, Spring 容器在配置文件中发现 bean 的 autowire 属性被设置成 byType , 之后容器试图匹配、装配和该 bean 的属性 具有相同类型的 bean。如果有多个 bean 符合条件, 则抛出错误。
  - constructor: 这个方式类似于 byType, 但是要提供给构造器参数, 如果没有确定的带参数的构造器参数类型, 将会抛出异常。
  - autodetect: 首先尝试使用 constructor 来自动装配, 如果无法工作, 则使用 byType 方式。

## 94、自动装配有哪些局限性 ?

自动装配的局限性是:

- 重写: 你仍需用 和 配置来定义依赖, 意味着总要重写自动装配。
- 基本数据类型: 你不能自动装配简单的属性, 如基本数据类型, String 字符串, 和类。
- 模糊特性: 自动装配不如显式装配精确, 如果有可能, 建议使用显式装配。

## 95、你可以在 Spring 中注入一个 null 和一个空字符串吗?

可以。

## 96、什么是基于 Java 的 Spring 注解配置? 给一些注解的例子

基于 Java 的配置, 允许你在少量的 Java 注解的帮助下, 进行你的大部分 Spring 配置而非通过 XML 文件。以@Configuration 注解为例, 它用来标记类可以当做一个 bean 的定义, 被 Spring IOC 容器使用。另一个例子是@Bean 注解, 它表示此方法将要返回一个对象, 作为一个 bean 注册进 Spring 应用上下文。

## 97、什么是基于注解的容器配置?

相对于 XML 文件, 注解型的配置依赖于通过字节码元数据装配组件, 而非尖括号的声明。开发者通过在相应的类, 方法或属性上使用注解的方式, 直接组件类中进行配置, 而不是使用 xml 表达 bean 的装配关系。

---

## 98、怎样开启注解装配？

注解装配在默认情况下是不开启的，为了使用注解装配，我们必须在 Spring 配置文件中配置 context:annotation-config/元素。

## 99、@Required 注解

这个注解表明 bean 的属性必须在配置的时候设置，通过一个 bean 定义的显式的属性值 或通过自动装配，若 @Required 注解的 bean 属性未被设置，容器将抛出 BeanInitializationException。

## 100、@Autowired 注解

@Autowired 注解提供了更细粒度的控制，包括在何处以及如何完成自动装配。它的用法和 @Required 一样，修饰 setter 方法、构造器、属性或者具有任意名称和/或多个参数的 PN 方法。

## 101、@Qualifier 注解

当有多个相同类型的 bean 却只有一个需要自动装配时，将@Qualifier 注解和 @Autowire 注解结合使用以消除这种混淆，指定需要装配的确切的 bean。

## 102、在 Spring 框架中如何更有效地使用 JDBC？

使用 SpringJDBC 框架，资源管理和错误处理的代价都会被减轻。所以开发者只需写 statements 和 queries 从数据存取数据，JDBC 也可以在 Spring 框架提供的模板类的帮助下更有效地被使用，这个模板叫 JdbcTemplate（例子见[这里](#) here）

## 103、JdbcTemplate

JdbcTemplate 类提供了很多便利的方法解决诸如把数据库数据转变成基本数据类型或对象，执行写好的或可调用的数据库操作语句，提供自定义的数据错误处理。

## 104、Spring 对 DAO 的支持

---

Spring 对数据访问对象（DAO）的支持旨在简化它和数据访问技术如 JDBC，Hibernate or JDO 结合使用。这使我们可以方便切换持久层。编码时也不用担心会捕获每种技术特有的异常。

## 105、使用 Spring 通过什么方式访问 Hibernate？

在 Spring 中有两 种 方 式 访 问 Hibernate :

- 控 制 反 转 Hibernate Template 和 Callback。
- 继 承 HibernateDAOsupport 提供一个 AOP 拦 截 器。

## 106、Spring 支持的 ORM

Spring 支持以下 ORM:

- Hibernate
- iBatis
- JPA (Java Persistence API)
- TopLink
- JDO (Java Data Objects)
- OJB

## 107. 如 何 通 过 HibernateDaoSupport 将 Spring 和 Hibernate 结合起来？

用 Spring 的 SessionFactory 调用 LocalSessionFactory。集成过程分三步：

- 配 置 the Hibernate SessionFactory。
- 继 承 HibernateDaoSupport 实现一个 DAO。
- 在 AOP 支持的事务中装配。

## 108、Spring 支持的事务管理类型

Spring 支持两种类型的事务管理：

- 编 程 式 事 务 管 理：这 意 味 你 通 过 编 程 的 方 式 管 理 事 务，给 你 带 来 极 大 的 灵 活 性，但 是 难 维 护。
- 声 明 式 事 务 管 理：这 意 味 着 你 可 以 将 业 务 代 码 和 事 务 管 理 分 离，你 只 需 用 注 解 和 XML 配 置 来 管 理 事 务。

## 109、Spring 框架的事务管理有哪些优点？

- 
- 它为不同的事务 API 如 JTA, JDBC, Hibernate, JPA 和 JDO, 提供一个不变的编程模式。
  - 它为编程式事务管理提供了一套简单的 API 而不是一些复杂的事务 API 如 · 它支持声明式事务管理。
  - 它和 Spring 各种数据访问抽象层很好得集成。

## 110、你更倾向用那种事务管理类型？

大多数 Spring 框架的用户选择声明式事务管理，因为它对应用代码的影响最小，因此更符合一个无侵入的轻量级容器的思想。声明式事务管理要优于编程式事务管理，虽然比编程式事务管理（这种方式允许你通过代码控制事务）少了一点灵活性。

## 111、解释 AOP

面向切面的编程，或 AOP，是一种编程技术，允许程序模块化横向切割关注点，或横切典型的责任划分，如日志和事务管理。

## 112、Aspect 切面

AOP 核心就是切面，它将多个类的通用行为封装成可重用的模块，该模块含有一组 API 提供横切功能。比如，一个日志模块可以被称作日志的 AOP 切面。根据需求的不同，一个应用程序可以有若干切面。在 Spring AOP 中，切面通过带有@Aspect 注解的类实现。

## 113、在 Spring AOP 中，关注点和横切关注的区别是什么？

关注点是应用中一个模块的行为，一个关注点可能会被定义成一个我们想实现的一个功能。横切关注点是一个关注点，此关注点是整个应用都会使用的功能，并影响整个应用，比如日志，安全和数据传输，几乎应用的每个模块都需要的功能。因此这些都属于横切关注点。

## 114、连接点

连接点代表一个应用程序的某个位置，在这个位置我们可以插入一个 AOP 切面，它实际上是个应用程序执行 Spring AOP 的位置。

## 115、通知

---

通知是个在方法执行前或执行后要做的动作，实际上是程序执行时要通过 SpringAOP 框架触发的代码段。Spring 切面可以应用五种类型的通知：

- before: 前置通知，在一个方法执行前被调用。
- after: 在方法执行之后调用的通知，无论方法执行是否成功。
- after-returning: 仅当方法成功完成后执行的通知。
- after-throwing: 在方法抛出异常退出时执行的通知。
- around: 在方法执行之前和之后调用的通知。

## 116、切点

切入点是一个或一组连接点，通知将在这些位置执行。可以通过表达式或匹配的方式指明切入点。

## 117、什么是引入？

引入允许我们在已存在的类中增加新的方法和属性。

## 118、什么是目标对象？

被一个或者多个切面所通知的对象。它通常是一个代理对象。也指被通知（advised）对象。

## 119、什么是代理？

代理是通知目标对象后创建的对象。从客户端的角度看，代理对象和目标对象是一样的。

## 120、有几种不同类型的自动代理

- BeanNameAutoProxyCreator
- DefaultAdvisorAutoProxyCreator (3) Metadata autoproxying

## 121、什么是织入。什么是织入应用的不同点？

织入是将切面和到其他应用类型或对象连接或创建一个被通知对象的过程。织入可以在编译时，加载时，或运行时完成。

## 122、解释基于 XML Schema 方式的切面实现。

---

在这种情况下，切面由常规类以及基于 XML 的配置实现。

## 123、解释基于注解的切面实现

在这种情况下(基于@AspectJ 的实现)，涉及到的切面声明的风格与带有 java5 标注的普通 java 类一致。

## 124、什么是 Spring 的 MVC 框架？

Spring 配备构建 Web 应用的全功能 MVC 框架。Spring 可以很便捷地和其他 MVC 框架 集成，如 Struts，Spring 的 MVC 框架用控制反转把业务对象和控制逻辑清晰地隔离。它也允许以声明的方式把请求参数和业务对象绑定。

## 125、DispatcherServlet

Spring 的 MVC 框架是围绕 DispatcherServlet 来设计的，它用来处理所有的 HTTP 请求和响应。

## 126、WebApplicationContext

WebApplicationContext 继承了 ApplicationContext 并增加了一些 WEB 应用必备的特有功能，它不同于一般的 ApplicationContext，因为它能处理主题，并找到被关联的 servlet。

## 127、什么是 Spring MVC 框架的控制器？

控制器提供一个访问应用程序的行为，此行为通常通过服务接口实现。控制器解析用户输入并将其转换为一个由视图呈现给用户的模型。Spring 用一个非常抽象的方式实现了一个控制层，允许用户创建多种用途的控制器。

## 128、@Controller 注解

该注解表明该类扮演控制器的角色，Spring 不需要你继承任何其他控制器基类或引用 Servlet API。

## 129、@RequestMapping 注解

---

该注解是用来映射一个 URL 到一个类或一个特定的方处理法上。

作者：程序员追风

链接：<https://juejin.cn/post/6844904081857708045>

来源：掘金

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## JVM

面试和笔试的要点其实差不多，基础知识和实战经验都是最重要的关注点（当然，面试时的态度和眼缘也很重要）。

实际面试时，因为时间有限，不可能所有问题都问一遍，一般根据简历上涉及的内容，抽一部分话题来聊一聊。看看面试者的经验，态度，以及面对一层层的深入问题时的处理思路。借此了解面试者的技术水平，对深度、广度，以及思考和解决问题的能力。

常见的面试套路是什么呢？

- XXX是什么？
- 实现原理是什么？
- 为什么这样实现？
- 如果让你实现你会怎么做？
- 分析下你的实现有什么优缺点？

- 
- 有哪些需要改进的地方？

下面总结一些比较常见的面试题，供大家参考。

针对这些问题，大家可以给自己打一个分。



- 
- 0分：不清楚相关知识。
  - 30分：有一点印象，知道一些名词。
  - 60分：知道一些概念以及含义，了解功能和常见用途。
  - 80分：能在参考答案的基础上进行补充。
  - 100分：发现参考答案的问题。

下面我们来看看JVM相关面试问题。

## 1. 什么是JVM？

JVM全称是 Java Virtual Machine，中文称为 Java 虚拟机。

JVM是Java程序运行的底层平台，与Java支持库一起构成了Java程序的执行环境。

分为JVM规范和JVM实现两个部分。简单来说，Java虚拟机就是指能执行标准Java字节码的虚拟计算机。

### 1.1 请问JDK与JVM有什么区别？

现在的JDK、JRE和JVM一般是整套出现的。

JDK = JRE + 开发调试诊断工具

JRE = JVM + Java标准库

### 1.2 你认识哪些JVM厂商？

常见的JDK厂商包括：

- Oracle公司，包括 Hotspot虚拟机、GraalVM；分为OpenJDK和OracleJDK两种版本。
- IBM 公司，J9虚拟机，用在IBM的产品套件中
- Azul Systems公司，高性能的Zing和开源的Zulu
- 阿里巴巴，Dragonwell 是阿里开发的OpenJDK定制版
- 亚马逊，Corretto OpenJDK
- Red Hat公司的OpenJDK

- 
- Adopt OpenJDK
  - 此外，还有一些开源和试验性质的JVM实现，比如Go.JVM

## 1.3 OracleJDK与OpenJDK有什么区别？

各种版本的JDK一般来说都会符合Java虚拟机规范。

两者区别一般来说包括：

- 两种JDK提供的工具套件略有差别，比如jmc等有版权的工具。
- 某些协议或配置不一样，比如美国限制出口的加密算法。
- 其他细微差别，比如JRE中某些私有的API不一样。

## 1.4 开发中使用哪个版本的JDK？生产环境呢？为什么这么选？

有一说一。选择哪个版本需要考虑研发团队的具体情况：比如机器的操作系统，团队成员的掌握情况，兼顾遗留项目等等。

当前Java最受欢迎的长期维护版本是Java8和Java11。

- Java8是经典LTS版本，性能优秀，系统稳定，良好支持各种CPU架构和操作系统平台。
- Java11是新的长期支持版，性能更强，支持更多新特性，而且经过几年的维护已经很稳定。

有的企业在开发环境使用OracleJDK，在生产环境使用OpenJDK。

也有的企业恰好相反，在开发环境使用OpenJDK，在生产环境使用OracleJDK。

也有的公司使用同样的打包版本。

开发和部署时只要进行过测试就没问题。

一般来说。测试环境、预上线环境的JDK配置需要和生产环境一致。

## 2. 什么是Java字节码？

Java 中的字节码，是值 Java 源代码编译后的中间代码格式，一般称为字节码文件。

### 2.1 字节码文件中包含哪些内容？

字节码文件中，一般包含以下部分：

- 版本号信息

- 
- 静态常量池（符号常量）
  - 类相关的信息
  - 字段相关的信息
  - 方法相关的信息
  - 调试相关的信息
  -

可以说，大部分信息都是通过常量池中的符号常量来表述的。

## 2.2 什么是常量？

常量是指不变的量，字母 `'K'` 或者数字 `1024` 在UTF8编码中对应到对应的二进制格式都是不变的。同样地，字符串在Java中的二进制表示也是不变的，比如 `"KK"`。在Java中需要注意的是，`final` 关键字修饰的字段和变量，表示最终变量，只能赋值1次，不允许再次修改，由编译器和执行引擎共同保证。

## 2.3 你怎么理解常量池？

在Java中，常量池包括两层含义：

- 静态常量池，class文件中的一个部分，里面保存的是类相关的各种符号常量。
- 运行时常量池，其内容主要由静态常量池解析得到，但也可以由程序添加。

## 3. JVM的运行时数据区有哪些？

根据 [JVM规范](#)，标准的JVM运行时数据区包括以下部分：

- 程序计数器
- Java虚拟机栈
- 堆内存
- 方法区
- 运行时常量池
- 本地方法栈

具体的JVM实现可根据实际情况进行优化或者合并，满足规范的要求即可。

### 3.1 什么是堆内存？

堆内存是指由程序代码自由分配的内存，与栈内存作区分。

---

在Java中，堆内存主要用于分配对象的存储空间，只要拿到对象引用，所有线程都可以访问堆内存。

## 3.2 堆内存包括哪些部分？

以Hotspot为例，堆内存（HEAP）主要由GC模块进行分配和管理，可分为以下部分：

- 新生代
- 存活区
- 老年代
- 

其中，新生代和存活区一般称为年轻代。

## 3.3 什么是非堆内存？

除堆内存之外，JVM的内存池还包括非堆（NON\_HEAP），对应于JVM规范中的方法区，常量池等部分：

- MetaSpace
- CodeCache
- Compressed Class Space

## 4. 什么是内存溢出？

内存溢出（OOM）是指可用内存不足。

程序运行需要使用的内存超出最大可用值，如果不进行处理就会影响到其他进程，所以现在操作系统的处理办法是：只要超出立即报错，比如抛出 **内存溢出错误**。

就像杯子装不下，满了要溢出来一样，比如一个杯子只有500ml的容量，却倒进去600ml，于是水就溢出造成破坏。

## 4.1 什么是内存泄漏？

内存泄漏（Memory Leak）是指本来无用的对象却继续占用内存，没有再恰当的时机释放占用的内存。

不使用的内存，却没有被释放，称为 **内存泄漏**。也就是该释放的没释放，该回收的没回收。

比较典型的场景是：每一个请求进来，或者每一次操作处理，都分配了内存，却有一

---

部分不能回收（或未释放），那么随着处理的请求越来越多，内存泄漏也就越来越严重。

在Java中一般是指无用的对象却因为错误的引用关系，不能被GC回收清理。

## 4.2 两者有什么关系？

如果存在严重的内存泄漏问题，随着时间的推移，则必然会引起内存溢出。  
内存泄漏一般是资源管理问题和程序BUG，内存溢出则是内存空间不足和内存泄漏的最终结果。

## 5. 给定一个具体的类，请分析对象的内存占用

```
1 public class MyOrder{ private  
2     long orderId; private long  
3     userId; private byte state;  
4     private long createMillis;  
5 }
```

一般来说，MyOrder类的每个对象会占用40个字节。

### 5.1 怎么计算出来的？

计算方式为：

- 对象头占用12字节。
- 每个long类型的字段占用8字节，3个long字段占用24字节。
- byte 字段占用1个字节。
- 以上合计 37字节，加上以8字节对齐，则实际占用40个字节。

### 5.2 对象头中包含哪些部分？

对象头中一般包含两个部分：

- 标记字，占用一个机器字，也就是8字节。
- 类型指针，占用一个机器字，也就是8个字节。

- 
- 如果堆内存小于32GB，JVM默认会开启指针压缩，则只占用4个字节。

所以前面的计算中，对象头占用12字节。

如果是数组，对象头中还会多出一个部分：

- 数组长度，int值，占用4字节。

## 6. 常用的JVM启动参数有哪些？

截止目前（2020年3月），JVM可配置参数已经达到1000多个，其中GC和内存配置相关的JVM参数就有600多个。

但在绝大部分业务场景下，常用的JVM配置参数也就10来个。

例如：

```
1 # JVM启动参数不换行
2 # 设置堆内存
3 -Xmx4g -Xms4g
4 # 指定GC算法
5 -XX:+UseG1GC -XX:MaxGCPauseMillis=50
6 # 指定GC并行线程数
7 -XX:ParallelGCThreads=4
8 # 打印GC日志
9 -XX:+PrintGCDetails -XX:+PrintGCDateStamps
10 # 指定GC日志文件
11 -Xloggc:gc.log
12 # 指定Meta区的最大值
13 -XX:MaxMetaspaceSize=2g
14 # 设置单个线程栈的大小
15 -Xss1m
16 # 指定堆内存溢出时自动进行Dump
17 -XX:+HeapDumpOnOutOfMemoryError
18 -XX:HeapDumpPath=/usr/local/
```

此外，还有一些常用的属性配置：

```
1 # 指定默认的连接超时时间  
2 -Dsun.net.client.defaultConnectTimeout=2000
```



```
3 -Dsun.net.client.defaultReadTimeout=2000
4 # 指定时区
5 -Duser.timezone=GMT+08
6 # 设置默认的文件编码为UTF-8
7 -Dfile.encoding=UTF-8
8 # 指定随机数熵源(Entropy Source)
9 -Djava.security.egd=file:/dev/.urandom
```

## 6.1 设置堆内存XMX应该考虑哪些因素？

需要根据系统的配置来确定，要给操作系统和JVM本身留下一定的剩余空间。  
推荐配置系统或容器里可用内存的 70-80% 最好。

## 6.2 假设物理内存是8G，设置多大堆内存比较合适？

比如说系统有 8G 物理内存，系统自己可能会用掉一点，大概还有 7.5G 可以用，那么建议配置 `-Xmx6`。

- 说明： $7.5G \times 0.8 = 6G$ ，如果知道系统里有明确使用堆外内存的地方，还需要进一步降低这个值。

## 6.3 `-Xm` 设置的值与JVM进程所占用的内存有什么关系？

JVM总内存=栈+堆+非堆+堆外+Native

## 6.4 怎样开启GC日志？

一般来说，JDK8及以下版本通过以下参数来开启GC日志：

```
1 -XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:gc.log
```

如果是在JDK9及以上的版本，则格式略有不同：

```
1 -Xlog:gc*=info:file=gc.log:time:filecount=0
```

## 6.5 请指定使用G1垃圾收集器来启动 Hello 程序

```
1 java -XX:+UseG1GC  
2 -Xms4g  
3 -Xmx4g  
4 -Xloggc:gc.log  
5 -XX:+PrintGCDetails  
6 -XX:+PrintGCDateStamps Hello
```

## 7. Java8默认使用的垃圾收集器是什么？

Java8版本的Hotspot JVM， 默认情况下使用的是并行垃圾收集器（Parallel GC）。其他厂商提供的JDK8基本上也默认使用并行垃圾收集器。

### 7.1 Java11的默认垃圾收集器是什么？

Java9之后，官方JDK默认使用的垃圾收集器是G1。

### 7.2 常见的垃圾收集器有哪些？

常见的垃圾收集器包括：

- 串行垃圾收集器：`-XX:+UseSerialGC`
- 并行垃圾收集器：`-XX:+UseParallelGC`
- CMS垃圾收集器：`-XX:+UseConcMarkSweepGC`
- G1垃圾收集器：`-XX:+UseG1GC`

### 7.3 什么是串行垃圾收集？

就是只有单个worker线程来执行GC工作。

### 7.4 什么是并行垃圾收集？

并行垃圾收集，是指使用多个GC worker 线程并行地执行垃圾收集，能充分利用多核CPU的能力，缩短垃圾收集的暂停时间。

除了单线程的GC，其他的垃圾收集器，比如 PS， CMS， G1等新的垃圾收集器都使

---

用了多个线程来并行执行GC工作。

## 7.5 什么是并发垃圾收集器？

并发垃圾收集器，是指在应用程序在正常执行时，有一部分GC任务，由GC线程在应用线程一起并发执行。

例如 CMS/G1的各种并发阶段。

## 7.6 什么是增量式垃圾收集？

首先，G1的堆内存不再单纯划分为年轻代和老年代，而是划分为多个（通常是2048个）可以存放对象的小块堆区域（smaller heap regions）。

每个小块，可能一会被定义成 Eden 区，一会被指定为 Survivor 区或者 Old 区。这样划分之后，使得 G1 不必每次都去回收整个堆空间，而是以增量的方式来进行处理：每次只处理一部分内存块，称为此次 GC 的回收集（collection set）。

下一次GC时在本次的基础上，再选定一定的区域来进行回收。增量式垃圾收集的好处是大大降低了单次GC暂停的时间。

## 7.7 什么是年轻代？

年轻代是分来垃圾收集算法中的一个概念，相对于老年代而言，年轻代一般包括：

- 新生代，Eden区。
- 存活区，执行年轻代GC时，用存活区来保存活下来的对象。存活区也是年轻代的一部分，但一般有2个存活区，所以可以来回倒腾。

## 7.8 什么是GC停顿(GC pause)？

因为GC过程中，有一部分操作需要等所有应用线程都到达安全点，暂停之后才能执行，这时候就叫做GC停顿，或者叫做GC暂停。

## 7.9 GC停顿与STW停顿有什么区别？

这两者一般可以认为就是同一个意思。

## 8. 如果CPU使用率突然飙升，你会怎么排查？

缺乏经验的话，针对当前问题，往往需要使用不同的工具来收集信息，例如：

- 
- 收集不同的指标 (CPU, 内存, 磁盘IO, 网络等等)
  - 分析应用日志
  - 分析GC日志
  - 获取线程转储并分析
  - 获取堆转储来进行分析

## 8.1 如果系统响应变慢，你会怎么排查？

一般根据APM监控来排查应用系统本身的问题。

有时候也可以使用Chrome浏览器等工具来排查外部原因，比如网络问题。

## 8.2 系统性能一般怎么衡量？

可量化的3个性能指标：

- 系统容量：比如硬件配置，设计容量；
- 吞吐量：最直观的指标是TPS；
- 响应时间：也就是系统延迟，包括服务端延时和网络延迟。

这些指标。可以具体拓展到单机并发，总体并发，数据量，用户数，预算成本等等。

## 9. 使用过哪些JVM相关的工具？

这个问题请根据实际情况回答，比如Linux命令，或者JDK提供的工具等。

### 9.1 查看JVM进程号的命令是什么？

可以使用 `ps -ef` 和 `jps -v` 等等。

### 9.2 怎么查看剩余内存？

比如：`free -m`，`free -h`，`top` 命令等等。

### 9.3 查看线程栈的工具是什么？

一般先使用 `jps`命令，再使用 `jstack -l`

### 9.4 用什么工具来获取堆内存转储？

---

一般使用 jmap 工具来获取堆内存快照。

## 9.5 内存Dump时有哪些注意事项？

根据实际情况来看，获取内存快照可能会让系统暂停或阻塞一段时间，根据内存量决定。

使用jmap时，如果指定 `live` 参数，则会触发一次FullGC，需要注意。

## 9.6 使用JMAP转储堆内存大致的参数怎么处理？

示例：

```
1 jmap -dump:format=b,file=3826.hprof 3826
```

## 9.7 为什么转储文件以 `.hprof` 结尾？

JVM有一个内置的分析器叫做HPROF，堆内存转储文件的格式，最早就是这款工具定义的。

## 9.8 内存Dump完成之后，用什么工具来分析？

一般使用 Eclipse MAT工具，或者 jhat 工具来处理。

## 9.9 如果忘记了使用什么参数你一般怎么处理？

上网搜索是比较笨的办法，但也是一种办法。

另外就是，各种JDK工具都支持 `-h` 选项来查看帮助信息，只要用得比较熟练，即使忘记了也很容易根据提示进行操作。

## 10. 开发性问题：你碰到过哪些JVM问题？

比如GC问题、内存泄漏问题、或者其他疑难杂症等等。

然后可能还有一些后续的问题。例如：

- 你遇到过的印象最深的JVM问题是什么？
- 这个问题是怎么分析和解决的？
- 这个过程中有哪些值得分享的经验？

---

此问题为开放性问题，请根据自身情况进行回答，可以把自己思考的答案发到本课程的微信群里，我们会逐个进行分析点评。

