目录

[1：2021-12-09 阿里面试](#_Toc572251773_WPSOffice_Level1) [3](#_Toc572251773_WPSOffice_Level1)

[一、阿里巴巴-4:00](#_Toc313297703_WPSOffice_Level2) [3](#_Toc313297703_WPSOffice_Level2)

[二、阿里巴巴-5:00](#_Toc2112075524_WPSOffice_Level2) [4](#_Toc2112075524_WPSOffice_Level2)

[1、B树和B+树的区别](#_Toc313297703_WPSOffice_Level3) [4](#_Toc313297703_WPSOffice_Level3)

[2、Buffer和Cache的区别](#_Toc2112075524_WPSOffice_Level3) [4](#_Toc2112075524_WPSOffice_Level3)

[3、内存与缓存的区别](#_Toc1896130605_WPSOffice_Level3) [4](#_Toc1896130605_WPSOffice_Level3)

[4、CPU荷载和使用率的区别](#_Toc1757240402_WPSOffice_Level3) [5](#_Toc1757240402_WPSOffice_Level3)

[5、TCP怎样保证可靠性？](#_Toc1744322870_WPSOffice_Level3) [5](#_Toc1744322870_WPSOffice_Level3)

[6、Redo和Undo的区别](#_Toc1535210893_WPSOffice_Level3) [6](#_Toc1535210893_WPSOffice_Level3)

[7、怎样去排查内存溢出的问题？](#_Toc273459946_WPSOffice_Level3) [6](#_Toc273459946_WPSOffice_Level3)

[8、Kafka是怎样保证消息的准确发送？](#_Toc426307842_WPSOffice_Level3) [6](#_Toc426307842_WPSOffice_Level3)

[9、消息丢失问题怎么解决？](#_Toc950454102_WPSOffice_Level3) [6](#_Toc950454102_WPSOffice_Level3)

[10、消息积压问题怎么解决？](#_Toc1298725928_WPSOffice_Level3) [7](#_Toc1298725928_WPSOffice_Level3)

[11、高峰期接口2%的失败率是什么原因导致的？](#_Toc662883788_WPSOffice_Level3) [7](#_Toc662883788_WPSOffice_Level3)

[12、CPU负荷变大可能的原因？](#_Toc2090147927_WPSOffice_Level3) [7](#_Toc2090147927_WPSOffice_Level3)

[13、如何自定义一个类加载器？](#_Toc578711463_WPSOffice_Level3) [7](#_Toc578711463_WPSOffice_Level3)

[14、怎样排查CPU飙高的问题？](#_Toc450121378_WPSOffice_Level3) [7](#_Toc450121378_WPSOffice_Level3)

[15、什么场景会出发Full GC？](#_Toc1752595312_WPSOffice_Level3) [8](#_Toc1752595312_WPSOffice_Level3)

[2：2021-12-10  北京掌上先机（慧策）](#_Toc313297703_WPSOffice_Level1) [8](#_Toc313297703_WPSOffice_Level1)

[1、易上手](#_Toc2112075524_WPSOffice_Level1) [9](#_Toc2112075524_WPSOffice_Level1)

[2、配置少，大部分场景下不需要写sql，但同时也提供xml标签传入动态sql](#_Toc1896130605_WPSOffice_Level1) [9](#_Toc1896130605_WPSOffice_Level1)

[3、通过注解完成对象与表的映射关系](#_Toc1757240402_WPSOffice_Level1) [9](#_Toc1757240402_WPSOffice_Level1)

[1、工作量较大，尤其是在字段较多的时候](#_Toc1744322870_WPSOffice_Level1) [9](#_Toc1744322870_WPSOffice_Level1)

[2、动态sql的查询不太好支持](#_Toc1535210893_WPSOffice_Level1) [9](#_Toc1535210893_WPSOffice_Level1)

[4、对未来工作的期望和愿景](#_Toc273459946_WPSOffice_Level1) [9](#_Toc273459946_WPSOffice_Level1)

[5、如何看待面试造火箭，工作拧螺丝？](#_Toc426307842_WPSOffice_Level1) [9](#_Toc426307842_WPSOffice_Level1)

[6、JWT有什么缺点？](#_Toc950454102_WPSOffice_Level1) [9](#_Toc950454102_WPSOffice_Level1)

[7、JWT中一般包含什么？](#_Toc1298725928_WPSOffice_Level1) [9](#_Toc1298725928_WPSOffice_Level1)

[3：2021-12-12  腾讯CSID面试复盘](#_Toc662883788_WPSOffice_Level1) [10](#_Toc662883788_WPSOffice_Level1)

[1、什么是AQS](#_Toc2090147927_WPSOffice_Level1) [10](#_Toc2090147927_WPSOffice_Level1)

[2、分布式事务的处理方式](#_Toc578711463_WPSOffice_Level1) [10](#_Toc578711463_WPSOffice_Level1)

[3、除了锁还有什么保证线程安全的方法？](#_Toc450121378_WPSOffice_Level1) [11](#_Toc450121378_WPSOffice_Level1)

[4、了解NIO和BIO吗？](#_Toc1752595312_WPSOffice_Level1) [12](#_Toc1752595312_WPSOffice_Level1)

[13：2021-12-23 OPPO二面](#_Toc983706532_WPSOffice_Level1) [12](#_Toc983706532_WPSOffice_Level1)

[1、说说你总结的技术经验？](#_Toc1826568718_WPSOffice_Level1) [12](#_Toc1826568718_WPSOffice_Level1)

[2、线程池相关](#_Toc861709561_WPSOffice_Level1) [12](#_Toc861709561_WPSOffice_Level1)

[1、如果工作线程数小于等于核心线程数](#_Toc122876359_WPSOffice_Level1) [12](#_Toc122876359_WPSOffice_Level1)

[2、如果工作线程数大于核心线程数](#_Toc1451180946_WPSOffice_Level1) [13](#_Toc1451180946_WPSOffice_Level1)

[3、ABA问题怎么解决？](#_Toc1026380443_WPSOffice_Level1) [13](#_Toc1026380443_WPSOffice_Level1)

[4、HashMap的容量为什么是2的n次方？](#_Toc1787452797_WPSOffice_Level1) [14](#_Toc1787452797_WPSOffice_Level1)

[5、HashMap为什么不用平衡二叉树而使用红黑树？](#_Toc570421296_WPSOffice_Level1) [14](#_Toc570421296_WPSOffice_Level1)

[6、ThreadLocal的底层实现原理？](#_Toc703721664_WPSOffice_Level1) [14](#_Toc703721664_WPSOffice_Level1)

[7、克隆与new()的效率哪个更高？](#_Toc1257562819_WPSOffice_Level1) [15](#_Toc1257562819_WPSOffice_Level1)

[知识点1:类的创建过程](#_Toc324245159_WPSOffice_Level1) [15](#_Toc324245159_WPSOffice_Level1)

[知识点2:类的初始化顺序](#_Toc1422374874_WPSOffice_Level1) [16](#_Toc1422374874_WPSOffice_Level1)

[1. 父类静态变量](#_Toc66548914_WPSOffice_Level1) [16](#_Toc66548914_WPSOffice_Level1)

[2. 父类静态代码块](#_Toc1796101158_WPSOffice_Level1) [16](#_Toc1796101158_WPSOffice_Level1)

[3. 子类静态变量](#_Toc2042020274_WPSOffice_Level1) [16](#_Toc2042020274_WPSOffice_Level1)

[4. 子类静态代码块](#_Toc1298582411_WPSOffice_Level1) [16](#_Toc1298582411_WPSOffice_Level1)

[5. 父类非静态变量](#_Toc398277216_WPSOffice_Level1) [16](#_Toc398277216_WPSOffice_Level1)

[6. 父类非静态代码块](#_Toc138641613_WPSOffice_Level1) [17](#_Toc138641613_WPSOffice_Level1)

[7. 父类构造方法](#_Toc129832696_WPSOffice_Level1) [17](#_Toc129832696_WPSOffice_Level1)

[8. 子类非静态变量](#_Toc254736320_WPSOffice_Level1) [17](#_Toc254736320_WPSOffice_Level1)

[9. 子类非静态代码块](#_Toc1418421769_WPSOffice_Level1) [17](#_Toc1418421769_WPSOffice_Level1)

[10. 子类构造方法](#_Toc198706236_WPSOffice_Level1) [17](#_Toc198706236_WPSOffice_Level1)

**1：2021-12-09 阿里面试**

**一、阿里巴巴-4:00**

**1、StringBuffer和StringBuilder除了线程安全还有哪些区别？**

 相同点：都继承了AbstractStringBuilder这个类

区别：

1⃣️StringBuffer线程安全，StringBuilder线程不安全

2⃣️在单线程中StringBuilder更快，因为它不需要阻塞

**2、各种排序算法的时间复杂度？哪种最好？**

**二、阿里巴巴-5:00**

**1、B树和B+树的区别**

1⃣️都是多叉树，B树所有节点都会存储索引和数据

2⃣️B+树分内部节点和叶子结点，内部节点只存储索引，数据部分存在叶子结点中

3⃣️B+树中会存储相邻两个叶子结点的索引，以此来支持范围查找（利用了内存局部性原理

**2、Buffer和Cache的区别**

Buffer是写缓冲：为了解决磁盘写入数据比内存慢的问题，内存向磁盘写入之前会将数据放入缓冲区，以一个磁盘可接受的速度进行写入；

Cache是读缓存：为了解决从磁盘读取数据比内存慢的问题，将磁盘中的数据缓存到内存中，加快响应速度。

**3、内存与缓存的区别**

计算机是由CPU(运算器和控制器)、存储器、IO设备、电源等组成

存储器又分为内部存储，即内存和外部存储，即磁盘两部分

而缓存是CPU中的一部分，它的访问速度要比内存高许多，是为了解决CPU与内存之间存取差异而存在的缓冲区，缓存一般有RAM和ROM两种，RAM是断电即丢数，ROM是断电不丢数，缓存使用的就是RAM，它会将内存中访问频率高的一部分数据缓存下来，提高访问存取效率，但是随着时间的推进，这些高频访问的数据会越来越多，所以就会再增加一级缓存，这样就形成了一级缓存、二级缓存，CPU处理数据时以一级缓存—>二级缓存->内存的方式进行访问！

**4、CPU荷载和使用率的区别**

荷载是指CPU需要执行的任务总数，包括正在执行的和等待执行的

使用率是指正在执行的任务或进程对CPU资源的占有率情况，使用率高荷载不一定高，因为荷载为1的情况下也可能产生100%的利用率

**5、TCP怎样保证可靠性？**

1⃣️序列号seq和确认号ack机制：发送方和接收方之间的数据传输都会加上数据的序列号以及对该序列号的确认机制，保证消息被准确的发送和接受，有求必应

2⃣️超时重发机制：发送方在发送数据后，会启动一个监控程序检查本次请求是否超时，超时后需要重新发送；

3⃣️乱序重排机制：消息从IP网络层到TCP传输层的过程中是乱序的，TCP层会对接受到的数据通过IP数据报文的头部序列号进行重新排序，再发送到接受方；

4⃣️去重机制：消息从IP网络层到TCP传输层到过程中可能会存在某些数据包被重复发送，TCP层会在发送之前进行去重；

5⃣️流量控制：为避免发送方发送过快导致接受方处理不过来，发送方和接受方都有一个缓冲区，用来维护自己能够发送和接受的一个度；

**6、Redo和Undo的区别**

Redo会保存每一次的操作，会从缓冲区写入的磁盘进行持久化，方便重启时进行恢复，可以保证事务的原子性和持久性；

Undo会对事务中的每一次修改操作都生成一个回滚操作，可以理解为每次插入操作都会生成一条对应的删除操作，以此来完成回滚操作，保证事务的一致性

**7、怎样去排查内存溢出的问题？**

内存泄漏：可能是因为老年代持有对年轻代对象的引用，导致年轻代无法被回收！

内存溢出：堆空间不足！

1⃣️jps找到Java进程

2⃣️Top -Hp pid查看最占用CPU的线程，并将线程ID转成16进制

3⃣️jstack pid 查看进程中的线程堆栈信息，找到转换后的线程ID

4⃣️jmap -heap pid 查看jvm内存使用情况

5⃣️jmap -histo pid |head 20 过滤存活最多的前20个对象

6⃣️查看当前虚拟机内存使用情况和垃圾回收情况

结合GC日志和一些可视化工具进行分析！！！

**8、Kafka是怎样保证消息的准确发送？**

ack机制，ISR

**9、消息丢失问题怎么解决？**

**10、消息积压问题怎么解决？**

从指定消费位移批量消费；

**11、高峰期接口2%的失败率是什么原因导致的？**

网络原因：新疆商博会注册后登录失败的问题

限流问题服务拒绝：403

参数错误问题：400

1**2、CPU负荷变大可能的原因？**

频繁GC

**13、如何自定义一个类加载器？**

继承ClassLoader类—>重写findClass方法

**14、怎样排查CPU飙高的问题？**

1、jps：查看java进程. ps -ef |grep java.  ps aux |grep java

2、top：命令查看最占用CPU的进程

3、Top -Hp 进程ID查看进程中吃CPU的线程

4、printf ‘0x%x\n’ 线程pid ：将线程ID转化成16进制

5、jstack pid：打印指定进程中的堆栈信息，找到上面的线程ID

6、jmap -heap pid：查看当前jvm堆内存的使用情况

7、jmap -histo pid |head 20 查看对象内存中存活对象的数量和大小

8、jstat -gc pid ：查看当前虚拟机内存使用情况和gc垃圾回收统计情况

9、jstat -gcutil pid 10000 10

可以使用arthas工具，不让用的话只能使用命令了！！！

**15、什么场景会出发Full GC？**

System.gc();

堆伸缩会触发

分配担保失败

老年代对象过多

**2：2021-12-10  北京掌上先机（慧策）**

**1、线程结束之后还能再执行start吗？即一个线程能调用两次start方法吗？**

不能！第二次启动的时候会报IllegalThreadStateException错误

源码：

if (threadStatus != 0)

    throw new IllegalThreadStateException();

new ：初始化线程，将线程加入到线程组的unStart列表中（如果不存在则用系统的安全管理器创建）

start：启动线程，调用native start0()方法执行当前线程

**2、非对称加密算法的缺点**

加解密的过程比较耗时

**3、Mybatis Plus的缺点**

优点：

1、易上手

2、配置少，大部分场景下不需要写sql，但同时也提供xml标签传入动态sql

3、通过注解完成对象与表的映射关系

缺点：

1、工作量较大，尤其是在字段较多的时候

2、动态sql的查询不太好支持

4、对未来工作的期望和愿景

5、如何看待面试造火箭，工作拧螺丝？

**6、JWT有什么缺点？**

1⃣️Jwt比较长，传输过程需要更多的带宽资源

2⃣️数据使用RSA加密算法，加解密过程会增大接口的响应时长

**7、JWT中一般包含什么？**

**通过第三方库JOSE进行签名或加密**

颁发者

主体

受众

过期时间戳

生效时间戳

颁发时间戳

令牌唯一标识

**3：2021-12-12  腾讯CSID面试复盘**

**1、什么是AQS**

AbstractQueueSunchronizer抽象队列同步器，提供独占锁和共享锁两种方式

ReentrantLock内部通过Sync静态内部类的方式继承AQS，实现了可重入锁

可重入锁又支持公平锁( FairSync)和非公平锁(NofairSync)两种方式:

公平锁先尝试获取锁，获取失败时加入等待队列

非公平锁先尝试获取资源，获取成功直接占有锁，获取失败进行尝试获取流程

**2、分布式事务的处理方式**

**ACID ：**

原子性(Atomicity)、一致性(Consistency) 、隔离性(Isolation)、持久性(Durability)

**CAP：**

CAP原则又称CAP定理，指的是在一个分布式系统中，一致性(Consistency)、可用性 (Availability)、分区容忍性(Partition tolerance)。

CAP 原则指的是，这三个要素最多只能同时实现两点，不可能三者兼顾。

一致性:在分布式系统中的所有数据备份，在同一时刻是否同样的值。

可用性:在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。

分区容忍性:以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数 据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。

**BASE理论：**

是对CAP中的一致性和可用性进行一个权衡的结果，理论的核心思想就是:我们无法做到 强一致，但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性。

Basically Available(基本可用)  
Soft state(软状态)  
Eventually consistent(最终一致性)

1⃣️两阶段提交

2⃣️三阶段提交

3⃣️TCC机制

4⃣️本地消息队列表(MQ)

5⃣️Sagas事务模型(最终一致性)

**3、除了锁还有什么保证线程安全的方法？**

volatile关键字

CSA机制，比如使用原子类

**4、了解NIO和BIO吗？**

BIO是一个连接一个线程：每次请求过来，都会等待数据处理完毕，才开始执行，线程空等待；

NIO是一个请求一个线程：每次都是先处理数据，线程处理已经传输完的数据，等数据处理完毕selector线程会通知任务线程执行，与线程池搭配使用

NIO里面主要是通过channel和buffer处理数据！

**4：2021-12-13 腾讯音乐面试**

心得：此面试官不按常理出牌，上来先问项目中解决的技术问题，然后从排查问题的角度来考知识点，还会给很多的友好提示，一步步引导你得出答案，虽然我很菜，但我从新的角度看到了自己的漏洞，哎，举一反三的能力还是太差了！！！

后补：后来一想，感觉他问的都是自己擅长的东西，而且他就是拿自己处理过的问题来考你，还说学东西不能太死，这TM是你说了就能解决的吗！！

**1、JMM内存模型以及存在的问题和解决办法？**

考察JMM和线程安全性问题

内存模型：为了解决因为共享内存导致的**缓存一致性问题**和处理器优化的指令重排序等问题而定制的一套规范，主要包含：**限制处理器优化**和**禁止指令重排序**

JMM：Java语言实现的跨平台解决方案，屏蔽了硬件和操作系统的差异，使不同平台上都能共用同一套解决方案，主要是就是解决**原子性、可见行和有序性**三个并发问题！

**2、单例模式双重锁机制中，变量为什么要用volatile关键字修饰？**

创建一个对象的过程：

1⃣️分配内存空间

2⃣️实例化

3⃣️将实例指向内存空间

如果2和3发生重排序，就会导致第二次检查时单例对象不为空（因为指向了真实的内存空间），直接返回一个未初始化的对象，如果使用，就会报空指针异常。

**3、静态代理和动态代理的区别？**

静态代理：代理类手动创建，在程序编译阶段就能生成代理类的.class文件

动态代理：定义好代理类的生成逻辑，在程序运行期间动态地生成代理类的.class文件

**4、抛开JDK动态代理和CGLIB动态代理，怎样手写一个动态代理？**

      即如何实现代理模式？

1⃣️通过继承被代理类，重写被代理方法

2⃣️被代理类和代理类实现相同的接口，在代理类中依赖被代理对象，调用其被代理方法

5、一个service层的方法中调用了三个dao层的方法，那么service层的方法在开启事务和不开启事务的情况下执行，分别会创建多少个连接？

**6、一个spring boot项目中连接了远程mysql服务，已知IP和端口号，怎样查看mysql连接数？**

首先netstat -anp |grep 3306

查看某个Java进程中的数据库连接数：

jmap -histo:live pid|grep  mysql.jdbc.driver.JDBC4Connection，如：jmap -histo:live 80613|grep mysql.jdbc.driver.JDBC4Connection

**13：2021-12-23 OPPO二面**

面试心得：先问自己在工作中的沉淀总结，再问底层基础，我为啥没答上来？

**1、说说你总结的技术经验？**

不知道怎么说，主要是考察总结能力

**2、线程池相关**

* 线程池中的线程是怎么回收的？

每个线程执行完之后，都会执行processWorkerExit操作，判断工作线程数是否大于核心线程数：

1、如果工作线程数小于等于核心线程数

1⃣️corePoolKeepAlive=false，则不执行回收

2⃣️corePoolKeepAlive=true，则过期回收

2、如果工作线程数大于核心线程数

1⃣️keepAliveTime=0，立即回收

2⃣️keepAliveTime>0，延迟回收

回收方法：

工作线程数减1；

将任务移出队列

* 核心线程会一直存活吗？

这个问题我居然说会，是我错了！！！

其实是有一个配置参数**allowCoreThreadTimeOut**，如果为false，则核心线程会一直存活；如果为true，则核心线程数也会在等待keepAliveTime后销毁

* 线程执行完毕后是如何保证继续存活的？

如果核心线程数的

**3、ABA问题怎么解决？**

ABA问题就是指在CAS中，检查某个值是否被修改时，被检查值可能从A-B-A，但是只检查是否为A，无法知道曾经成为过B

解决办法：加版本号，1A-2B-2A，修改时先检查地址，再检查值

JUC中提供了AtomicStampedReference和AtomicMarkableReference两种原子类解决ABA问题：

AtomicStampedReference：基于版本号的实现，对象——〉版本号

AtomicMarkableReference：基于修改次数的实现，对象——〉是否被修改

**4、HashMap的容量为什么是2的n次方？**

一方面是提升效率：与运算效率高于取模运算

另一方面是与（2的n次方-1）与运算，能够使数组下标分布在低位

**5、HashMap为什么不用平衡二叉树而使用红黑树？**

平衡二叉树是严格平衡的，层高不超过1，在插入、删除元素时为了平衡需要更多reblance（旋转）操作，而红黑树不是严格平衡，只需要保证局部平衡即可，旋转次数较少

**6、ThreadLocal的底层实现原理？**

我没答上来，事实证明平时看的公众号并没有什么卵用！

每个Thread中都维护了一个ThreadLocal.ThreadLocalMap 类型的threadLocals变量

ThreadLocal中有一个静态内部类ThreadLocalMap，含有一个Entry类型的数组，每个Entry都是一个键值对的方式存储，基于哈希表的实现，通过开放地址法解决哈希冲突，

其中Entry的key为指向ThreadLocal的一个弱引用，value就是线程共享变量的值，因为是弱引用，在GC的时候一定会被回收，这样key的值就会为空，但是value的值

是强引用，所以就会出现内存泄漏的问题，这一点ThreadLocal的实现方式为：每次操作ThreadLocal之前都要调用get()方法，在get方法中会判断某个key是否为空，如果为空就会将其

Value也设置为null，这样这些value也可以被回收掉。

相关知识：

强引用：任何时候都不会被回收

软引用：只有在内存不足的时候才会被回收，在内存足够的情况下，触发GC也不会回收，适合做缓存

弱引用：只要触发了GC，就会被回收

虚引用：GC时会被移到一个队列，通过直接内存访问类 DirectByteBuffer用来管理堆外内存，在回收堆外内存时使用的Cleaner就是一个虚引用

**7、克隆与new()的效率哪个更高？**

**知识点1:类的创建过程**

通过new的方式创建一个对象的过程为：

加载—》验证—》准备—》解析—》初始化—》使用—》卸载

* 加载：将class文件读入内存，并将类变量（静态变量、静态代码块）、常量等放入方法区
* 验证：验证class文件的语法等合法性
* 准备：给类变量（类中的静态变量）分配内存空间，并赋默认值
* 解析：将常量池中的符号引用变成直接引用，即从符号替换为方法区中的地址
* 初始化：给类变量赋真实值，执行静态代码块，如果存在继承，则要先初始化父类，再初始化子类

通过new关键字创建对象的过程，需要经过完整的初始化过程，调用构造方法

通过clone创建对象的过程，只是在内存中复制了已存在的对象放入新的地址，不需要执行初始化过程，不调用构造方法

效率比较：

如果一个类的构造方法中没有太多逻辑，则new会比clone效率高，一旦构造方法中包含复杂逻辑，clone的效率就会比new高，clone只是将内存中已经存在的对象复制一份分配到新的内存空间，不需要进行对象的初始化过程，而new每次都要执行初始化

所以这两者的效率还得分具体的业务场景！

**知识点2:类的初始化顺序**

类的初始化顺序为：

1. 父类静态变量
2. 父类静态代码块
3. 子类静态变量
4. 子类静态代码块
5. 父类非静态变量
6. 父类非静态代码块
7. 父类构造方法
8. 子类非静态变量
9. 子类非静态代码块
10. 子类构造方法

**5 ：2021-12-14 深信服面试**

面试体验：

1、面试官迟到10分钟，早退2分钟，也没问有没有什么想问的，但人事那边还好，会发面试结果邮件，挂了！

2、先进行自我介绍，然后开始问项目，项目问题问了以下几点：

Oauth2.0授权码模式的整个过程；

Kafka的使用场景

Kafka的topic在项目中创建了多少个分区？

Pg和mysql的区别：我说pg支持递归查询，他说mysql也支持，其实我后来查了，pg是有递归函数，但是mysql需要自己实现

3、写算法题

我觉得我的思路没错，只是被他误导了

**1、如何给一个1000万的表加字段?**

第一步：创建一个在原表上新增字段的表结构

第二步：将原表中的数据插入新表insert into new\_table (select \* from old\_table)

第三步：修改旧表名old\_table\_tmp，修改新表名为old\_table

**2、如何优化下面的SQL语句？**

Select \* from product p where p.order\_id = “1” and p.order\_name like ‘123%’  order by create\_time desc;

1⃣️ 不用使用\*

优化SQL的一些规则：

* 尽量避免使用select \*
* 尽量用关联查询代替子查询
* 避免在where子句中使用!=、<>等，因为索引会失效
* 尽量用 exist和not exist代替 in 和not in

避免索引失效的一些规则：

* Like查询时左边不用%
* 索引字段上不加表达式，如!=等
* 索引字段上不要出现类型隐式转换，如一个int类型的手机号查询时使用’12344’
* 索引字段上不要出现函数计算
* 组合索引尽量满足最左匹配，即如果(a,b)是一个组合索引，查询时条件中尽量包含a,b
* 使用组合索引时，如果中间某个字段使用了范围查找，则该字段之后的索引会失效

**3、spring怎么解决循环依赖？**

我感觉我虽然答了，但是答的不太好，面试官貌似也不是很懂，我说完就直接跳到下一个问题了

**4、如何自定义一个spring-boot-starter**

先自定义一个自动配置类，加上@configuration注解

接下来定义该配置的属性类：定义前缀和属性名，这些属性名都是某个配置的属性，并将属性文件注解到配置类上

在META-INF目录下创建spring.factory文件，在该文件中加上自动配置类的权限定名：

5、如何理解spring和spring boot？

6、怎样排查CPU飙高的问题？

**6：2021-12-14 阿里二面**

面试体验：面试官很准时，问的问题也比较有深度，但是不知道为啥聊到最后会问我跟男朋友几年异地，有没有结婚之类的？难道是怕我去深圳就马上结婚生子？

身边的男同事都说女程序员好找工作，但我好像并没有感觉到，我能感觉到的都是大家觉得你的工作年限与你的能力不匹配，哎，难过！

**1、指针碰撞法如何知道一次移动多大的内存？**

Object o = new Object();

一个对象包含【对象头+实例数据+对齐填充】

对象头：对象头包含mark work+class pointer+数组长度（数组才有）

实例数据：对象属性

对齐填充：当整体大小不足8字节的整数倍时需要进行填充，如果是8字节整数倍则不需要

* 在未开启指针压缩的情况下对象的内存布局如下：
* 在开启指针压缩的情况下对象的内存布局如下：

从上面截图看到，new一个object时，总共会占用16字节！！！

如果new一个有属性的对象，就会计算其属性所占的字节，会有实例数据部分，下面来看看，我在ReadObject中增加一个**byte**类型的属性！

* 在未开启指针压缩的情况下对象的内存布局如下：
* 在开启指针压缩的情况下对象的内存布局如下：

**总结：**

* **没有属性时，一个对象只占16字节，开启压缩指针会填充，不开启压缩不会填充；**
* **有属性时，一个对象开启压缩比不开启压缩占用的空间更少；**
* **默认是开启的，可以通过-XX:+UseCompressedOops开启指针压缩；通过 关闭指针压缩；**

**2、如何排查线上性能问题？**

我扯到了jprofile这个工具

3、知道jprofile的实现原理吗？

**4、读spring源码的感受？**

* 一切皆对象：比如会将bean的定义信息封装成BeanDefinition
* 设计模式最佳实践：如模板方法模式、工厂方法模式和建造者模式

**5、spring中用到了哪些设计模式？**

单例模式：其实我觉得spring中的单例模式更确切的说应该是享元模式

**6、如何手写一个观察者模式？**

观察者模式又叫发布-订阅模式，以微信公众号为例，用户订阅公众号，开发者发布文章

首先，定义一个抽象订阅者（观察者），用来对事件做出响应，比如用户User

接下来，定义一个具体订阅者，用来订阅文章，比如ZhangSan

然后，定义一个抽象目标，比如个人账号PersonalAccount，负责维护订阅者信息，同时支持发布文章

最后，定义一个具体目标，即具体的个人公众号，如Java精选，实现抽象目标中的方法，每来一个订阅者，就将其加入到订阅者列表中，当发布一篇文章的时候，将

通知所有订阅者，即群发！

**7、项目中遇到了什么问题？怎么解决的？**

就说续费自动化配置和多线程优化的问题，引导面试官到线程池，再给他解析源码

8、zookeeper了解吗？

**9、BeanFactory和FactoryBean的区别？**

BeanFactory主要是管理普通bean的创建

FactoryBean主要是解决复杂Bean的创建，将具体的创建流程交给用户实现，用户通过注册BeanDefinition到容器，获取bean实例时调用的事用户自定义的getObject方法！

哎，睡吧，明天再解决这些问题！

**7：2021-12-18 AM:微金融面试**

面试心得：两轮面试加起来差不多1个小时吧，主要问业务，可能是因为面试官也在做多租户的产品，所以会问一些业务方面的问题，还是比较专业的，技术的话问的比较少，只问了spring源码，没有写代码，两轮面试速战速决！

1、说一下spring的启动流程

2、你们的身份认证服务能否提取出来作为一个产品使用？

3、你们在多租户共享数据的情况下，是怎么解决修改核心代码后对其他租户的兼容性问题？

这个问题我之前确实是思考过，但是还没想道一种好的解决方案！！

4、在项目中遇到了什么样的问题？

**8：2021-12-15  OPPO面试总结**

心得：唯一一个真开视频面试的，感觉这个OPPO的面试官水平跟我差不多，问的问题都比较大，等我讲完可能就不深入去问了，但我还是有一些没回答上来的，真切觉得自己的数据库水品还是不行，要加强了！！！

1、SQL调优技巧都有哪些？

**2、mysql索引的实现原理或为什么使用B+树？**

数据库索引的实现目的就是为了加快检索速度，因为我们的数据是存储在磁盘中，应用程序处理数据的时候需要先将磁盘的数据读取到内存中，但是不可能一次性读取磁盘全量的数据，而是分块读取，而内存与磁盘数据交互的方式是磁盘页，每次都会读取4K或4K正数倍的数据到到内存，存储引擎在实现过程中是通过k-v格式来存储数据，key就是列名，v就是列值，而支持k-v格式的数据结构有二叉树、AVL树、红黑树等，但它们有一个共同的缺点就是最多支持2个分支，随着数据量的增加，树的高度也会越来约大，导致的结果就是IO次数增多，所以为了解决IO次数可以选择B树，因为在B树中每个节点可以存储更多数据，而且每个节点也可以有多个分支，而且索引和数据都存储在节点中，找到索引也就找到了数据，但是这种方式占用的空间很大，意味着所能存储的数据也就越少，为了支持更大的数据量，对B树进行了优化，就是只有非叶子结点存储索引，叶子阶段只存储数据，这也就是B+树的来源，同时B+树的叶子结点之间有指针连接，可以通过指针找到任何一个数据！

**3、kafka是怎么保证高吞吐量的？**

* 多**分区**
* 磁盘顺序读写：log文件的追加和消费者消费都是顺序读写，一般情况下，磁盘的顺序读写要性能要比磁盘随机读写和内存随机读写都快！
* 页缓存技术：页缓存就是将要读取的磁盘页提前缓存到内存，消息的发送和接受都是优先与页缓存交互，后台会起定时任务在指定时间段内去同步到磁盘，这种同步机制一般是由操作系统完成，所以在Kafka服务挂了也不会影响到数据！
* 零拷贝技术：减少数据在内核态和用户态之间的拷贝

**3、kafka的零拷贝机制实现原理**

零拷贝主要是为了解决数据从一块存储拷贝到另一块存储的问题！

一般情况下，生产者发送一条数据给消费者需要经过以下几个过程：

* 将数据从磁盘复制到内核态的页缓存
* 从内核态的页缓存读取数据到用户态的应用程序
* 再从用户态应用程序写入内核态的socket缓冲区
* 最后从socket缓冲区发送到网卡设备

上面的过程中，数据在用户态除了两次多余的复制，其实没有任何实际的操作，但是这种复制又增加了一次发送过程的处理时长，所以Kafka利用了操作系统的0拷贝技术，

直接让数据从内核态的页缓存发送至内核态的socket缓冲区，然后从缓冲区直接发送到网卡设备！其实零拷贝并不是没有拷贝，只是减少了拷贝的次数。

零拷贝其实是操作系统对数据IO的一种优化，一般有mmap和sendfile两种方式：

Mmap:其实就是让用户态和内核态共享页缓存，当应用程序将数据发送给mmap时，就已经从页缓存发送到socket缓存了（DMA）

Sendfile :数据直接在内核态之间传输，完全不经过用户态，就是直接从页缓存发送到socket缓冲区了，向socket缓冲追加数据在页缓存中的位置和偏移量，在socket缓冲区中直接将数据copy到网卡

Java中的实现：  
传统方式：FIS->FOS(FileInputStream/FileOutputStream)

MMAP方式：FileChannel(des.map())

sendFile方式：FileChannel(srcFc->dexFc)

**4、G1算法的实现原理有了解吗？**

基于三色标记，对内存分区回收

三色标记中的三色其实就是垃圾收集器赋予对象的三种状态：未标记->标记一半->全部标记，分别用白、灰、黑三种颜色表示

白色：所有对象初始状态都为未标记—白色

灰色：如果一个对象自己被标记，但是子对象，即被其引用的对象没有被标记，这种对象会被标记为标记一半，即灰色

黑色：如果一个对象自己以及所有被其引用的对象都被标记完了，这种对象的状态为全部标记，即黑色

如果一个对象已经是黑色，那么垃圾收集线程不会对其进行重复标记！

等一次标记完成之后，白色的就是垃圾，可以被回收，非白色的不是垃圾，不能被回收，但是这种情况会产生浮动垃圾，即一个原本为白色的对象被黑色对象引用，此时进行标记时，黑色不会被重新标记，导致新关联的白色漏标！

CMS对浮动垃圾的解决方案：当一个白色对象被黑色对象引用时，将黑色对象变为灰色，这样下次标记时可以触发重新标记，但是这种解决方案还是无法解决并发标记中的浮动垃圾问题，于是CMS又会为会一个引用消失的记录表，每次标记之前先去记录表中查询是否有消失对象以及消失对象是否被重新建立引用！

G1收集算法是将内存均分成相同大小的区域Region：

Region最小值默认为1M，最大值默认为32M，Region总数默认为2048块，计算实际region的方法是：

min(minHeapSize/2048,minRegionSize)，即最小堆内存除以Region数所得值与最小region大小中取最小值！

**5、创建线程池时如何决定创建多少线程？**

一半情况下，在框架中一般设置的规则是：IO密集型的应用需要设置2n+1，CPU密集型的应用设置为n+1，其中n为CPU核数，具体要怎么设置其实没有一个确定的数，只是分不同的应用和业务场景以及服务器的配置，在我们的应用当中一般都会设置的比较大，可以先设置一个基数，然后在程序运行过程中通过jstack命令查看进程中的线程状态，如果有空闲的，那就在基数上减，反之继续逐步测试，取一个比较合适的值。

**9：2021-12-18 PM:虾皮面试**

面试心得：面试官感觉挺nice，不是那种很严肃的，会在开始之前闲聊一下，一个小时技术面+半小时写代码，代码最后没有跑起来，我还遇到一个尴尬的问题，现在还不知道为什么，我裂了，我先总结下今天的问题，待会儿再去看事出原因！！！

**1、你们的服务之前是怎么做权限控制的？**

我就说外网是通过token，内网之前是通过JWT

**2、说一下一个请求到spring boot的controller层经历的过程**

SpringApplication.run():

* new SpringApplication():推断应用类型—>设置初始化类和监听器(扫描spring.factories文件）
* run()：创建第一步中推断出的应用—>准备上下文【准备并设置环境信息，如环境变量等—>调用初始化类中的初始化方法和监听器中的发布事件方法】—>调用refresh方法初始化spring容器

在调用初始化方法的时候，就会执行servlet中的init方法，设置HandlerMapping

* 发起请求后交给DispatherServlet的doService进行处理，通过HandlerMapping生成处理器链
* HandlerAdapter适配到合适的处理器，将请求交给对应的Controller进行处理
* Controller处理完成返回ModleAndView，然后由视图解析器ViewResolver返回视图数据和页面
* 在对应的view页面渲染数据

**3、spring中的@Autowired的实现原理**

spring在初始化bean的前后会执行AutowiredAnnotationBeanPostProcessor这个后置处理器，该处理器负责对@Autowired注解或@Value两种注解的bean进行自动装配，具体流程大概是：

* 先找到该注解对应类的元信息
* 通过元信息解析出Class
* 找到需要自动装配的bean，封装成InjectionMetadata，注入到BeanDefinition中

**4、hashMap在并发的情况下会有什么问题？**

答案：会造成数据覆盖、数据丢失和死循环等问题！！

分析：

Jdk7中扩容时会出现死循环和数据丢失问题：

Jdk8中进行put时会出现数据覆盖和modCount（计算++modCount）不一致的问题：

**5、volatile的实现原理？**

保证可见性和禁止指令重排序

**6、在高并发场景下为什么不使用volatile，而要加锁？**

* 线程A和线程B同时从内存中获取某个共享变量i
* 线程A即将执行+1操作时被挂起，此时线程B开始执行+1操作，发现i没有被改变，直接加1并写回内存
* 线程A获取到时间片继续执行+1操作，因为已经读取到所以不刷新i，执行+1写回内存

此时内存中的值只加了一次，并没有加两次，与预期值不符

7、什么是mysql的原子性？mysql是如何保证原子性的？

**8、mysql的高并发是如何实现的？**

我就提了MVCC，然后就说了一通，面试官也没有深入问

**9、BigDecimal的实现原理？为什么不用Double？**

这个问题还是比较实用的，为什么我都不知道，我表示很汗颜啊！！！！

为什么不用double？

* 小数由符号位+指数+位数：

Float占四个字节32位，其组成包括【1位符号位+8位指数+23尾数】，小数部分23位能表示的最大十进制数是2^23=8388608即7位

Double占8个字节64为，其组成包括【1位符号位+11位指数+52位尾数】，小数部分52位能表示的最大十进制数是2^52=4503599627370496，总共16位

* 像float和double这种十进制数用二进制表示的时候，小数可能会丢失精度，float最多能表示8位（即精度是8位有效数字），double最多能表示17位（即精度是17位有效数字），超过的部分无法用二进制表示，而是直接被舍弃。
* 算术运算也会丢失精度，比如0.1+0.2，如果用double相加，得到的结果其实并不是0.3，而是无限接近0.3
* 浮点数是用来科学计算的，之所以成为浮点数，而不是小数或实数，是因为浮点数用科学计数法表示时，小数点会浮动，比如100.23456用浮点数表示就是1.0023456e2，即1.0023456\*10^2

可以使用BigDecimal精确表示大数而不会丢失精度

拓展：

十进制转二进制的方法：

* 整数：除2取余，逆序排列
* 小数：整数部分和小数部分分开计算再合并，小数部分使用【乘2取整，顺序排列】

比如2.35用二进制表示时：

2用二进制表示就是10

2/2余0，1/2余1，1/2余0

0.35用二进制表示就是01011001100….（1100无限循环）

0.35\*2=0.7  取0

0.7\*2=1.4    取1

0.4\*2=0.8   取0

0.8\*2=1.6   取1

0.6\*2=1.2   取1

0.2\*2=0.4  取0

0.4\*2=0.8  取0

0.8\*2=1.6   取1

 ……..循环1100

最后2.35用二进制表示就是10.010110011001100….（1100无限循环）

1**0、redis的持久化实现原理？**

RDB和AOF

**11、怎么解决redis与数据库的同步问题？**

* 第一种方案：让缓存失效

一般而言先更新数据库再更新缓存，如果缓存更新失败，可以删除缓存，再从从DB获取最新值存入redis

* 第二种方案：重试补偿机制

当更新缓存失败时，记录失败的信息，并进行重试，但可能存在redis挂了的情况，此时采取第三种补偿方案

* 第三种方案：定时任务补偿机制

将失败日志记录下来，后台开启定时任务重试，或者定时任务定期刷新redis缓存

**12、redis是怎么解决缓存穿透问题的？**

布隆过滤器我都没说上来，哎，那些透过的懒终归是要还的！！

**布隆过滤器**：二进制数组，初始化容量和误差，初始时所有数组元素均为0，当执行put方法时，经过多次hash计算出具体的索引位置，并将该位置的值设置为1，表示元素已经存在

为什么要多次hash?

为了减少误差fpp。因为可能存在多个key的hash值相同的情况，此时不同的key会被指向同一个位置，判断元素是否存在时无法确认到底是哪个key，通过多次hash的方式保证同一个key存在与多个位置，每次判断是否存在时必须确保所有位置的值都为1才算存在，但是这也只能说明key可能存在，而不是一定存在！

**13、分布式锁在释放的时候需要注意什么问题？**

* 获取锁时key尽量保持唯一
* 使用当前请求的唯一标识比如requestId等做为key的value，在释放锁时验证释放的是不是同一个请求的锁，同时也要保证其原子性，通过LUA脚本实现
* 获取锁和设置锁的过期时间必须保证原子性：可以使用setNx(k,v,expire,timeUnit)或LUA脚本保证
* 为了防止在任务执行完成之前提前自动释放锁，需要设置锁续约，可以通过守护线程检查任务是否执行完毕，如果未执行完毕则对锁进行延期

14、使用Redis有没有一些什么自己的心得？

**10：2021-12-20 腾讯二面**

面试心得：面试官问多线程、高并发这块比较多，千算万算没有算到二面也要写代码，本来就心虚的自己加上没有好好刷题，好像从一开始就已经放弃了算法题，时间一长就很心虚紧张，虽然面试官给了我足够的时间和机会，看得出他希望我能做出来，可是我最后还是放弃了，也许我真的不适合进大厂吧！

**1、mysql单表能存储的数据上限是多少？**

 3层结构，每次读取16KB数据，索引的键值占用16byte，那么一次IO读取16KB/16B=1K，三层总共存储1k\*1K\*1K=101⃣️数据，其实限制mysql最优存储大小的是索引，抛开索引那就是自增主键的最大值和磁盘空间了。

**2、redis最多能支持多少并发？**

redis-benchmark -q -n 100000

PING\_INLINE: 169491.53 requests per second, p50=0.135 msec

PING\_MBULK: 185873.61 requests per second, p50=0.135 msec

SET: 189753.31 requests per second, p50=0.135 msec

GET: 129701.68 requests per second, p50=0.247 msec

INCR: 165289.25 requests per second, p50=0.143 msec

LPUSH: 175131.36 requests per second, p50=0.143 msec

RPUSH: 166944.92 requests per second, p50=0.143 msec

LPOP: 138504.16 requests per second, p50=0.199 msec

RPOP: 151285.92 requests per second, p50=0.159 msec

SADD: 142857.14 requests per second, p50=0.167 msec

HSET: 115874.86 requests per second, p50=0.271 msec

SPOP: 131926.12 requests per second, p50=0.231 msec

ZADD: 140252.45 requests per second, p50=0.207 msec

ZPOPMIN: 124533.01 requests per second, p50=0.247 msec

LPUSH (needed to benchmark LRANGE): 105152.48 requests per second, p50=0.271 msec

LRANGE\_100 (first 100 elements): 78926.60 requests per second, p50=0.343 msec

LRANGE\_300 (first 300 elements): 37341.30 requests per second, p50=0.695 msec

LRANGE\_500 (first 500 elements): 25667.35 requests per second, p50=1.007 msec

LRANGE\_600 (first 600 elements): 22512.38 requests per second, p50=1.135 msec

MSET (10 keys): 169779.30 requests per second, p50=0.207 msec

**3、知道怎么生成分布式ID吗？**

* UUID：

优点：简单方便

缺点：不具备自增特性，没有实际的业务意义，无规律，数据频繁变动影响性能

* 自增ID

实现：通过一台单独的实例专门用来生成自增ID，每次需要都向它请求

优点：能够保持单调自增

缺点：单机存在宕机风险，无法应付高并发场景

* ID生成数据库使用双主模式集群

实现：设置两台机节点的ID初始值和增长值，比如DB1，初始值1；每次增2，DB2，初始值2，每次增2，这样DB1生成的为1，3，5，DB2生成的为2，4，6

优点：解决单点问题

缺点：不利于集群扩容，扩容后重新设置节点的初始值和增长步长，可能产生重复值

* 基于数据库的号段模式

实现：每次从数据库批量的获取一个号段的ID，然后由具体业务生成本号段自增的ID加载到内存，使用完后申请新的号段

优点：不强依赖数据库，不会频繁访问数据库，一个号段只访问一次

缺点：需要考虑并发申请号段的问题，需要使用乐观锁

* 基于Redis的方式

实现：利用redis的incr自增命令

优点：基于内存，自增是原子的，响应速度快，不需要依赖数据库

缺点：依赖Redis持久化技术，如果是RDB方式，可能出现重复，基于AOF的方式可以避免重复，但重启恢复时间较长

* 基于雪花算法（Snowflake）模式

实现：生成long类型的64位ID，正数位（占1比特）+ 时间戳（占41比特）+ 机器ID（占5比特）+ 数据中心（占5比特）+ 自增值（占12比特）

优点：封装成工具类直接调用，业务方只知道自己的机器ID即可，无需额外的资源

缺点：比较长

* 百度（uid-generator）

实现：基于雪花算法，支持用户自定义时间戳、机器ID、序列号等，每次服务启动都会向数据库插入一条记录，成功就返回机器ID，主机和IP地址

优点：支持自定义，时间戳使用秒

缺点：依赖数据库，每次重启都会消耗一个ID

* 美团（Leaf）

实现：同时支持号段模式和雪花模式，可以切换使用

* 滴滴（Tinyid）

实现：基于号段模式，类似Leaf

**4、NIO、BIO、AIO的区别？**

BIO:同步阻塞，一个请求一个线程，响应之前线程阻塞，处理多个请求需要使用多线程，输入流和输出流分开

NIO:同步非阻塞，多个请求复用一个线程，线程控制通道，无需创建多个输入输出流，在通道中同时处理输入和输出请求  
AIO:异步非阻塞，类似发布订阅

**5、你是如何创建线程池的，为什么不用现有的固定个数的线程池？**

* newCachedThreadPool

实现：coreSize为0，maxSize为Integer.max，可以无限创建线程，空闲时间为1分钟，创建大量线程容易触发OOM

优点：可以提高线程复用率

缺点：任务执行时间长于1分钟的情况下，会创建大量线程

* newFixedThreadPool

实现：核心线程数和最大线程数固定且相同，无界任务队列， keepAlive=0，线程执行完立即终止

优点：可以控制线程数，根据业务设置合适的线程数，针对请求数固定的场景，比如定时任务分批处理，已知总共有10批，则创建10个即可，且用完即毁

缺点：任务超过线程数时会被阻塞，而且线程不能被复用，只能重复创建-销毁

* newScheduledThreadPool

**11：2021-12-22  虾皮**

面试心得：面试官比较喜欢问底层原理，比如redis各种数据类型的底层实现，zookeeper的底层实现，NIO的底层实现，算法题也相对比较松，会先让我想想怎么实现，如果没有思路会换一道新的题目，必须要吐槽的是HR，刚面试完就打电话过来问怎么样，还劝我转go，我表达了自己不太想转的意愿之后就开始叨叨，说Java市场饱和了怎么滴，我们都30了怎么滴，你要是不转就只能去一个小的创业公司怎么滴，感觉被PUA了，遇到这种傻逼也是不想说什么了，就算我找不到也不会被你忽悠，你算哪根葱？

记录一下没有回答上的问题吧：

1、mybatis的分页实现原理

**2、Redis几种数据类型的底层实现原理？**

Redis中有6种数据结构，几种数据类型基本都是其中1～2种的实现，先来说说6种数据结构吧：

* **SDS：简单动态字符串，维护了字符串长度、未使用空间和字节数组，查询字符串长度的时间复杂度为O(1)**

struct sdshdr{

     //记录buf数组中已使用字节的数量

     //等于 SDS 保存字符串的长度

     int len;

     //记录 buf 数组中未使用字节的数量

     int free;

     //字符数组，用于保存字符串

     char buf[];

}

* **双端链表：按照插入顺序存储的双端链表，无环，head的pred和tail的next为null，且因为维护了节点数量，所以获取链表长度的时间复杂度也为O(1)，主要应用在比如列表键、发布与订阅、慢查询、监视器等。**

typedef struct list{

     //表头节点

     listNode \*head;

     //表尾节点

     listNode \*tail;

     //链表所包含的节点数量

     unsigned **long** len;

     //节点值复制函数

     void (\***free**) (void \*ptr);

     //节点值释放函数

     void (\***free**) (void \*ptr);

     //节点值对比函数

**int** (\*match) (void \*ptr,void \*key);

}list;

* **字典：字典的底层是通过哈希表实现，每个字典都维护两个哈希表，其中的一个是用来在rehash时使用——》每个哈希表都维护一个 dicEntry数组存放数据——》每个Entry都包含键值以及指向下一个节点的指针（解决哈希冲突)**

typedef struct dictht{

     //哈希表数组

     dictEntry \*\*table;

     //哈希表大小

     unsigned **long** size;

     //哈希表大小掩码，用于计算索引值

     //总是等于 size-1

     unsigned **long** sizemask;

     //该哈希表已有节点的数量

     unsigned **long** used;

}dictht

字典rehash的过程实在是太妙了：

首先，每个字典维护了两个哈希表h[0]和h[1]，刚开始使用h[0]，根据负载因子（已存在元素个数/哈希表的容量）对h[1]进行扩容，扩容至第一个大于等于h[0].used\*2^n

然后，将h[0]中已有数据重新计算索引存入h[1]：通过渐进式rehash实现，避免数据量过大时影响性能

最后，当h[0]中所有元素rehash到h[1]时，将h[1]设置为h[0]，释放h[0]，并创建一个空的h[1]为下次rehash使用

渐进式rehash的过程：

h[0]中的数据不是一次性迁移到h[1]中，而是在rehash开始时会维护一个rehashindex，初始值为0，每次进行数据的增删改查时，都会对rehashindex位置的元素进行迁移，每迁移成功一个rehashindex的值就会加1，迁移期间，两个哈希表都能使用，只是h[0]不提供新增服务，所以已经迁移过的rehashindex不存在数据，当rehashindex的值等于h[0]的size时说明h[0]中所有数据迁移完成，此时会将rehashindex设置为-1，这种做法既不影响数据的迁移，又不影响redis继续提供服务，简直太妙了！

redis的数据库底层使用的就是字典这种数据结构！！！

* 整数集合

可以存储16、32和64位的整数，底层通过数组实现，元素有序且不能重复

* 跳跃表

一个有序的链表，包含跳跃表节点和跳跃表链表两种数据结构，一个跳跃表中维护着头节点、尾节点、节点数量和最大层数level，其中头节点中存放1～32的层级

每个跳跃表节点中存放数据对象、分值、层级信息、后退指针等

* 跳跃表链表数据结构：

typedef struct zskiplist{

     //表头节点和表尾节点

     structz skiplistNode \*header, \*tail;

     //表中节点的数量

     unsigned long length;

     //表中层数最大的节点的层数

     int level;

}zskiplist

* 跳跃表链表节点数据结构：

typedef struct zskiplistNode {

     //层

     struct zskiplistLevel{

           //前进指针

           struct zskiplistNode \*forward;

           //跨度

           unsigned **int** span;

     }level[];

     //后退指针

     struct zskiplistNode \*backward;

     //分值

**double** score;

     //成员对象

     robj \*obj;

} zskiplistNode

* 压缩列表

压缩表

**2.1 字符串String**

字符串的底层就是字符串，或字符数组

**2.2 列表List**

* 压缩列表

实现：类似数组，用一片连续的空间存储

前提：每个元素不能超过64字节且列表总元素不能超过512个

特点：节省空间，且可以存储不同类型的数据

* 双向循环链表

实现：单独的List结构体维护head、tail、length，每个节点都是一个链表，包含前驱、后继和当前值

// 以下是C语言代码，因为Redis是用C语言实现的。

typedef struct listnode {

  struct listNode \*prev;

  struct listNode \*next;

  void \*value;

} listNode;

typedef struct list {

  listNode \*head;

  listNode \*tail;

  unsigned long len;

  // ....省略其他定义

} list;

**2.3 字典Hash**

作用：用来存储一组数据对，每个数据对又包含键值两部分

* 压缩列表

实现：类似数组，用一片连续的空间存储一组键值对

前提：每个键和值的大小都不能超过64字节且列表总键值对的个数不能超过512个

特点：节省空间，只有在数据量较小的时候使用

* 散列表

前提：不能满足压缩列表的条件时，使用散列表

实现：支持动态扩容缩容，通过链表解决哈希冲突，装载因子大于1时，扩容2倍；装载因子小于0.1时，缩容两倍

**2.4 集合Set**

作用：用来存储一组不重复的数据

* 基于有序数组

前提：当存储的数据全部都是整型，且元素个数不超过512时

* 基于散列表

当不能满足有序数组条件时，使用散列表

**2.5 有序集合sortedset（ZSet）**

作用：存储一组数据，每个数据都有一个分值，通过分值对数据进行排序

* 基于压缩列表

前提：当存储的数据不超过64字节且数据个数不超过128个时，基于压缩列表实现

* 基于跳表

**3、知道DirectByteBuffer吗？**

NIO中处理堆外内存的

**12：2021-12-23 OPPO二面**

面试心得：先问自己在工作中的沉淀总结，再问底层基础，我为啥没答上来？

**1、说说你总结的技术经验？**

不知道怎么说，主要是考察总结能力

**2、线程池相关**

* 线程池中的线程是怎么回收的？

每个线程执行完之后，都会执行processWorkerExit操作，判断工作线程数是否大于核心线程数：

1、如果工作线程数小于等于核心线程数

1⃣️corePoolKeepAlive=false，则不执行回收

2⃣️corePoolKeepAlive=true，则过期回收

2、如果工作线程数大于核心线程数

1⃣️keepAliveTime=0，立即回收

2⃣️keepAliveTime>0，延迟回收

回收方法：

工作线程数减1；

将任务移出队列

* 核心线程会一直存活吗？

这个问题我居然说会，是我错了！！！

其实是有一个配置参数**allowCoreThreadTimeOut**，如果为false，则核心线程会一直存活；如果为true，则核心线程数也会在等待keepAliveTime后销毁

* 线程执行完毕后是如何保证继续存活的？

如果核心线程数的

**3、ABA问题怎么解决？**

ABA问题就是指在CAS中，检查某个值是否被修改时，被检查值可能从A-B-A，但是只检查是否为A，无法知道曾经成为过B

解决办法：加版本号，1A-2B-2A，修改时先检查地址，再检查值

JUC中提供了AtomicStampedReference和AtomicMarkableReference两种原子类解决ABA问题：

AtomicStampedReference：基于版本号的实现，对象——〉版本号

AtomicMarkableReference：基于修改次数的实现，对象——〉是否被修改

**4、HashMap的容量为什么是2的n次方？**

一方面是提升效率：与运算效率高于取模运算

另一方面是与（2的n次方-1）与运算，能够使数组下标分布在低位

**5、HashMap为什么不用平衡二叉树而使用红黑树？**

平衡二叉树是严格平衡的，层高不超过1，在插入、删除元素时为了平衡需要更多reblance（旋转）操作，而红黑树不是严格平衡，只需要保证局部平衡即可，旋转次数较少

**6、ThreadLocal的底层实现原理？**

我没答上来，事实证明平时看的公众号并没有什么卵用！

每个Thread中都维护了一个ThreadLocal.ThreadLocalMap 类型的threadLocals变量

ThreadLocal中有一个静态内部类ThreadLocalMap，含有一个Entry类型的数组，每个Entry都是一个键值对的方式存储，基于哈希表的实现，通过开放地址法解决哈希冲突，

其中Entry的key为指向ThreadLocal的一个弱引用，value就是线程共享变量的值，因为是弱引用，在GC的时候一定会被回收，这样key的值就会为空，但是value的值

是强引用，所以就会出现内存泄漏的问题，这一点ThreadLocal的实现方式为：每次操作ThreadLocal之前都要调用get()方法，在get方法中会判断某个key是否为空，如果为空就会将其

Value也设置为null，这样这些value也可以被回收掉。

相关知识：

强引用：任何时候都不会被回收

软引用：只有在内存不足的时候才会被回收，在内存足够的情况下，触发GC也不会回收，适合做缓存

弱引用：只要触发了GC，就会被回收

虚引用：GC时会被移到一个队列，通过直接内存访问类 DirectByteBuffer用来管理堆外内存，在回收堆外内存时使用的Cleaner就是一个虚引用

**7、克隆与new()的效率哪个更高？**

**知识点1:类的创建过程**

通过new的方式创建一个对象的过程为：

加载—》验证—》准备—》解析—》初始化—》使用—》卸载

* 加载：将class文件读入内存，并将类变量（静态变量、静态代码块）、常量等放入方法区
* 验证：验证class文件的语法等合法性
* 准备：给类变量（类中的静态变量）分配内存空间，并赋默认值
* 解析：将常量池中的符号引用变成直接引用，即从符号替换为方法区中的地址
* 初始化：给类变量赋真实值，执行静态代码块，如果存在继承，则要先初始化父类，再初始化子类

通过new关键字创建对象的过程，需要经过完整的初始化过程，调用构造方法

通过clone创建对象的过程，只是在内存中复制了已存在的对象放入新的地址，不需要执行初始化过程，不调用构造方法

效率比较：

如果一个类的构造方法中没有太多逻辑，则new会比clone效率高，一旦构造方法中包含复杂逻辑，clone的效率就会比new高，clone只是将内存中已经存在的对象复制一份分配到新的内存空间，不需要进行对象的初始化过程，而new每次都要执行初始化

所以这两者的效率还得分具体的业务场景！

**知识点2:类的初始化顺序**

类的初始化顺序为：

1. 父类静态变量
2. 父类静态代码块
3. 子类静态变量
4. 子类静态代码块
5. 父类非静态变量
6. 父类非静态代码块
7. 父类构造方法
8. 子类非静态变量
9. 子类非静态代码块
10. 子类构造方法

**13：2021-12-28 腾讯电话面试**

心得：这个面试官可能只是想测试我一下，问的都是比较底层，几乎不问Java，我感觉我不是特别适合他们，他们也不是特别适合我，但我还是要把他问的问题记下来！

**1、Redis几种数据结构的底层实现原理**

Redis中有6种数据结构，几种数据类型基本都是其中1～2种的实现，先来说说6种数据结构吧：

* **SDS：简单动态字符串，维护了字符串长度、未使用空间和字节数组，查询字符串长度的时间复杂度为O(1)**

struct sdshdr{

     //记录buf数组中已使用字节的数量

     //等于 SDS 保存字符串的长度

     int len;

     //记录 buf 数组中未使用字节的数量

     int free;

     //字节数组，用于保存字符串

     char buf[];

}

* **双端链表：按照插入顺序存储的双端链表，无环，head的pred和tail的next为null，且因为维护了节点数量，所以获取链表长度的时间复杂度也为O(1)，主要应用在比如列表键、发布与订阅、慢查询、监视器等。**

typedef struct list{

     //表头节点

     listNode \*head;

     //表尾节点

     listNode \*tail;

     //链表所包含的节点数量

     unsigned **long** len;

     //节点值复制函数

     void (\***free**) (void \*ptr);

     //节点值释放函数

     void (\***free**) (void \*ptr);

     //节点值对比函数

**int** (\*match) (void \*ptr,void \*key);

}list;

* **字典：字典的底层是通过哈希表实现，每个字典都维护两个哈希表，其中的一个是用来在rehash时使用——》每个哈希表都维护一个 dicEntry数组存放数据——》每个Entry都包含键值以及指向下一个节点的指针（解决哈希冲突)**

typedef struct dictht{

     //哈希表数组

     dictEntry \*\*table;

     //哈希表大小

     unsigned **long** size;

     //哈希表大小掩码，用于计算索引值

     //总是等于 size-1

     unsigned **long** sizemask;

     //该哈希表已有节点的数量

     unsigned **long** used;

}dictht

字典rehash的过程实在是太妙了：

首先，每个字典维护了两个哈希表h[0]和h[1]，刚开始使用h[0]，根据负载因子（已存在元素个数/哈希表的容量）对h[1]进行扩容，扩容至第一个大于等于h[0].used\*2^n

然后，将h[0]中已有数据重新计算索引存入h[1]：通过渐进式rehash实现，避免数据量过大时影响性能

最后，当h[0]中所有元素rehash到h[1]时，将h[1]设置为h[0]，释放h[0]，并创建一个空的h[1]为下次rehash使用

渐进式rehash的过程：

h[0]中的数据不是一次性迁移到h[1]中，而是在rehash开始时会维护一个rehashindex，初始值为0，每次进行数据的增删改查时，都会对rehashindex位置的元素进行迁移，每迁移成功一个rehashindex的值就会加1，迁移期间，两个哈希表都能使用，只是h[0]不提供新增服务，所以已经迁移过的rehashindex不存在数据，当rehashindex的值等于h[0]的size时说明h[0]中所有数据迁移完成，此时会将rehashindex设置为-1，这种做法既不影响数据的迁移，又不影响redis继续提供服务，简直太妙了！

redis的数据库底层使用的就是字典这种数据结构！！！

* 整数集合

可以存储16、32和63位的整数，底层通过数组实现，元素有序且不能重复

* 跳跃表

一个有序的链表，包含跳跃表节点和跳跃表链表两种数据结构，一个跳跃表中维护着头节点、尾节点、节点数量和最大层数level，其中头节点中存放1～32的层级

每个跳跃表节点中存放数据对象、分值、层级信息、后退指针等

* 跳跃表链表数据结构：

typedef struct zskiplist{

     //表头节点和表尾节点

     structz skiplistNode \*header, \*tail;

     //表中节点的数量

     unsigned long length;

     //表中层数最大的节点的层数

     int level;

}zskiplist

* 跳跃表链表节点数据结构：

typedef struct zskiplistNode {

     //层

     struct zskiplistLevel{

           //前进指针

           struct zskiplistNode \*forward;

           //跨度

           unsigned **int** span;

     }level[];

     //后退指针

     struct zskiplistNode \*backward;

     //分值

**double** score;

     //成员对象

     robj \*obj;

} zskiplistNode

* 压缩列表

压缩表

**2.1 字符串String**

字符串的底层就是字符串，或字符数组

**2.2 列表List**

* 压缩列表

实现：类似数组，用一片连续的空间存储

前提：每个元素不能超过64字节且列表总元素不能超过512个

特点：节省空间，且可以存储不同类型的数据

* 双向循环链表

实现：单独的List结构体维护head、tail、length，每个节点都是一个链表，包含前驱、后继和当前值

// 以下是C语言代码，因为Redis是用C语言实现的。

typedef struct listnode {

  struct listNode \*prev;

  struct listNode \*next;

  void \*value;

} listNode;

typedef struct list {

  listNode \*head;

  listNode \*tail;

  unsigned long len;

  // ....省略其他定义

} list;

**2.3 字典Hash**

作用：用来存储一组数据对，每个数据对又包含键值两部分

* 压缩列表

实现：类似数组，用一片连续的空间存储一组键值对

前提：每个键和值的大小都不能超过64字节且列表总键值对的个数不能超过512个

特点：节省空间，只有在数据量较小的时候使用

* 散列表

前提：不能满足压缩列表的条件时，使用散列表

实现：支持动态扩容缩容，通过链表解决哈希冲突，装载因子大于1时，扩容2倍；装载因子小于0.1时，缩容两倍

**2.4 集合Set**

作用：用来存储一组不重复的数据

* 基于有序数组

前提：当存储的数据全部都是整型，且元素个数不超过512时

* 基于散列表

当不能满足有序数组条件时，使用散列表

**2.5 有序集合sortedset（ZSet）**

作用：存储一组数据，每个数据都有一个分值，通过分值对数据进行排序

* 基于压缩列表

前提：当存储的数据不超过64字节且数据个数不超过128个时，基于压缩列表实现

* 基于跳表

**2、TCP是怎样保证可靠传输的？**

我就说了序列号、去重、超时重发、重新排序、滑动窗口，他就追问我是怎样检测超时的？

* **校验和**

由发送端对伪首部、TCP报文、TCP数据计算校验和之后放入头部，在接收端进行验证，保证数据在传输过程中没有被篡改

* **序列号和ACK应答机制**

发送方和接受方发送数据的过程中，通过序列号和ACK应该来保证要发送的数据包和是否成功接受，比如A和B的一次通信中：

**第一步：3次握手建立连接**

A：SYN seq=x

B：ACK ack=x+1,seq=y

A：SYN seq=x+1,ack=y+1

针对发送方的每一个seq，接受方都会发送一个ack！

**第二步：发送数据**

A：seq=x

B：ack=x+1,seq=y

A：ack=y+1,seq=x+1

B：ack=x+2,seq=y+2

**第三步：4次挥手断开连接**

断开连接的过程中，B在接受到被动断开时可能还没有完成数据传输，所以不会直接回复FIN+ACK，而是先回复一个表示接受到关闭请求的ACK，等完成数据传输后再继续回复SYN+ACK，比起三次挥手，多出的一次就是完成数据传输的过程，在建立连接时不需要关系目前的状态，所以接收端在接受到建立连接请求后直接回复SYN+ACK！

A：FIN，seq=u

B：ACK，ack=u+1，seq=v

B：FIN,ACK，ack=u+1,seq=w

A：ACK，ack=w+1,seq=u+1

* **重传机制（超时重发和快速重传）**

**1、超时重传**

发送端设置一个定时器，在RTO（RTT+抖动时间）时间内没有收到接收端的ACK时，说明数据包可能在发送过程或回传过程中丢失了，因此需要重新传送数据包，保证其被真实接受！

**2、快速重传**

每次发送后不等待接受方的ACK继续发送，如果接受方多次传回同一个 ack，则说明ack的数据包已丢失，发送方进行重传，比如：

A：seq=1,seq=2,seq=3,seq=4,seq=5连续发送5个数据包

B：seq=2,seq=2,seq=2,seq=2

对于数据包1,3,4,5，接收方B的ack都回复2，说明seq=2的数据包已丢失，发送方接受到这种相同的ack时，对当前ack进行重传，即快速重传，无需定时器及过期时间的检查！

* **流量控制（滑动窗口协议）**

为了提高网络吞吐率并解决发送方和接收方速率不匹配的问题，引入了发送缓冲区和滑动窗口协议。

缓冲区要求发送方在接受到ack应答之前需要保留缓冲区的数据包，直到接受到应该才能从缓冲区删除；

滑动窗口就是指无需确认应该能发送数据包的最大值，即缓冲区的大小。

在数据传输过程中，数据的状态有以下四种：

1、已发送且已接受到应该

2、已发送但未接受到应答

3、未发送但在窗口范围之内

4、未发送但在窗口范围之外

针对以上四种状态，只有第2、3在滑动窗口内，第一种已完成传输，第二种变成第一种后，窗口向右滑动，可以接受第四种状态的数据，这就是滑动窗口的含义，其中窗口大小是按照接受方处理能力规定的，在每次接受到的ack中，接受方都会回传当前能接受的最大范围，从而让发送方实时控制发送的数据量。

* **拥塞控制**

如果出现重传，说明发生了网络拥塞，发送方需要通过慢启动的方式探测，避免发送过快导致拥塞

3、讲一个挑战性最大的项目

**4、你有自己写过哈希函数吗？**

static final int hash(Object key) {

    int h;

    // 将key的hashCode与自身无符号右移16位后取得的结果进行异或运算，这样就能保证有效位数分布到1～32位上，减少碰撞，为什么会减少呢，且看下面！

    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

计算索引的方法是：（tab.size()-1)&hash()，一般桶的size为2的n次幂，减1之后相当于n之后的位置都为1，n之前的位置都变成了0，这样再与hash()函数得到的结果进行与运算，就能保证索引的值总是小于2^n，因为哈希函数得到的哈希值与自身无符号右移16位的值进行了异或运算，同时也就保证了低位的值就是hash值，而且位正数

拓展知识：

 1与任何数与运算都为任何数

0与任何数或运算都为任何数

0与任何数异或运算都为任何数，两个相同的数异或结果位0

**14：2021-12-29  OPPO商城面试**

面试体验：面试官还是很有礼貌，提问都是：“我想跟您探讨一下”这种口吻，但是问题问题更偏向于具体的操作，比如某个SQL怎么实现？

**1、数据库方面**

两个一摸一样的表怎样查出A中存在B中不存在的，这题为什么我没答上来？？？

Select \* from A a left join B b on [a.id](http://a.id) = [b.id](http://b.id) where [a.id](http://a.id)!=[b.id](http://b.id);

Select \* from a where id not in (select id from b);

**2、线上OOM怎样排查？**

哎，本来能回答的，我发现我这个只要一个问题回答不好，就会对自己失去信心，导致后面本来知道的问题也回答不好了，这一点一定要在下次的面试中改正

1、jps：查看java进程. ps -ef |grep java.  ps aux |grep java

2、top：命令查看最占用CPU的进程

3、Top -Hp 进程ID查看进程中吃CPU的线程

4、printf ‘0x%x\n’ 线程pid ：将线程ID转化成16进制

5、jstack pid：打印指定进程中的堆栈信息，找到上面的线程ID

6、jmap -heap pid：查看当前jvm堆内存的使用情况

7、jmap -histo pid |head 20 查看对象内存中存活对象的数量和大小

8、jstat -gc pid ：查看当前虚拟机内存使用情况和gc垃圾回收统计情况

9、jstat -gcutil pid 10000 10

3、知道Netty吗，用过Rocket MQ吗？

**4、hashMap中的hash函数是怎样计算的？**

static final int hash(Object key) {

    int h;

    // 将key的hashCode与自身无符号右移16位后取得的结果进行异或运算，这样就能保证有效位数分布到1～32位上，减少碰撞，为什么会减少呢，且看下面！

    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

计算索引的方法是：（tab.size()-1)&hash()，一般桶的size为2的n次幂，减1之后相当于n之后的位置都为1，n之前的位置都变成了0，这样再与hash()函数得到的结果进行与运算，就能保证索引的值总是小于2^n，因为哈希函数得到的哈希值与自身无符号右移16位的值进行了异或运算，同时也就保证了低位的值就是hash值，而且位正数

拓展知识：

 1与任何数与运算都为任何数

0与任何数或运算都为任何数

0与任何数异或运算都为任何数，两个相同的数异或结果位0