蚂蚁 Java 一面

1. 二叉搜索树和平衡二叉树有什么关系，强平衡二叉树（AVL 树）和弱平衡二叉树（红黑树）有什么区别

二叉搜索树：也称二叉查找树，或二叉排序树。定义也比较简单，要么是一颗空树，要么就是具有如下性质的二叉树：

（1）若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

（2）若任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

（3）任意节点的左、右子树也分别为二叉查找树；

（4）没有键值相等的节点。

平衡二叉树：在二叉搜索树的基础上多了两个重要的特点

（1）左右两子树的高度差的绝对值不能超过 1；

（2）左右两子树也是一颗平衡二叉树。

红黑书：红黑树是在普通二叉树上，对每个节点添加一个颜色属性形成的，需要同时满足一下五条性质

（1）节点是红色或者是黑色；

（2）根节点是黑色；

（3）每个叶节点（NIL 或空节点）是黑色；

（4）每个红色节点的两个子节点都是黑色的（也就是说不存在两个连续的红色节点）；

（5）从任一节点到其没个叶节点的所有路径都包含相同数目的黑色节点。

区别：AVL 树需要保持平衡，但它的旋转太耗时，而红黑树就是一个没有 AVL 树那样平衡，因此插入、删除效率会高于 AVL 树，而 AVL 树的查找效率显然高于红黑树。

参考文章 1：https://blog.csdn.net/qq_25940921/article/details/82183093

参考文章 2：https://blog.csdn.net/yang_yulei/article/details/26066409

2. B 树和 B+树的区别，为什么 MySQL 要使用 B+树

B 树：

（1）关键字集合分布在整颗树中；

（2）任何一个关键字出现且只出现在一个结点中；

（3）搜索有可能在非叶子结点结束；

（4）其搜索性能等价于在关键字全集内做一次二分查找；

B+树：

（1）有 n 棵子树的非叶子结点中含有 n 个关键字（b 树是 n-1 个），这些关键字不保存数据，只用来索引，所有数据都保存在叶子节点（b 树是每个关键字都保存数据）；

（2）所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接；

（3）所有的非叶子结点可以看成是索引部分，结点中仅含其子树中的最大（或最小）关键字；

（4）通常在 b+树上有两个头指针，一个指向根结点，一个指向关键字最小的叶子结点；

（5）同一个数字会在不同节点中重复出现，根节点的最大元素就是 b+树的最大元素。

B+树相比于 B 树的查询优势：

（1）B+树的中间节点不保存数据，所以磁盘页能容纳更多节点元素，更“矮胖”；

（2）B+树查询必须查找到叶子节点，B 树只要匹配到即可不用管元素位置，因此 B+树查找更稳定（并不慢）；

（3）对于范围查找来说，B+树只需遍历叶子节点链表即可，B 树却需要重复地中序遍历

参考文章：https://www.cnblogs.com/xueqiuqiu/articles/8779029.html

3. HashMap 如何解决 Hash 冲突

通过引入单向链表来解决 Hash 冲突。当出现 Hash 冲突时，比较新老 key 值是否相等，如果相等，新值覆盖旧值。如果不相等，新值会存入新的 Node 结点，指向老节点，形成链式结构，即链表。

当 Hash 冲突发生频繁的时候，会导致链表长度过长，以致检索效率低，所以 JDK1.8 之后引入了红黑树，当链表长度大于 8 时，链表会转换成红黑书，以此提高查询性能。

参考文章：https://blog.csdn.net/qedgbmwyz/article/details/79908333

4. epoll 和 poll 的区别，及其应用场景

select 和 epoll 都是 I/O 多路复用的方式，但是 select 是通过不断轮询监听 socket 实现，epoll 是当 socket 有变化时通过回掉的方式主动告知用户进程实现

参考文章:https://www.cnblogs.com/hsmwlyl/p/10652503.html

5. 简述线程池原理，FixedThreadPool 用的阻塞队列是什么？

Java 线程池的实现原理其实就是一个线程集合 workerSet 和一个阻塞队列 workQueue。当用户向线程池提交一个任务(也就是线程)时，线程池会先将任务放入 workQueue 中。

workerSet 中的线程会不断的从 workQueue 中获取线程然后执行。当 workQueue 中没有任务的时候，worker 就会阻塞，直到队列中有任务了就取出来继续执行。

FixedThreadPool 使用的是“无界队列”LinkedBlockingQueue

参考文章：https://blog.csdn.net/wanghao112956/article/details/99938893

6. sychronized 和 ReentrantLock 的区别

（1）ReentrantLock 显示获得、释放锁，synchronized 隐式获得释放锁

（2）ReentrantLock 可响应中断、可轮回，synchronized 是不可以响应中断的，为处理锁的不可用性提供了更高的灵活性

（3）ReentrantLock 是 API 级别的，synchronized 是 JVM 级别的

（4）ReentrantLock 可以实现公平锁

（5）ReentrantLock 通过 Condition 可以绑定多个条件

参考文章：https://blog.csdn.net/zxd8080666/article/details/83214089

7. sychronized 的自旋锁、偏向锁、轻量级锁、重量级锁，分别介绍和联系

自旋锁：果持有锁的线程能在很短时间内释放锁资源，那么那些等待竞争锁的线程就不需要做内核态和用户态之间的切换进入阻塞挂起状态，它们只需要等一等（自旋），

等持有锁的线程释放锁后即可立即获取锁，这样就避免用户线程和内核的切换的消耗。

偏向锁：顾名思义，它会偏向于第一个访问锁的线程，如果在运行过程中，同步锁只有一个线程访问，不存在多线程争用的情况，则线程是不需要触发同步的，减少加锁 / 解锁

的一些 CAS 操作（比如等待队列的一些 CAS 操作），这种情况下，就会给线程加一个偏向锁。 如果在运行过程中，遇到了其他线程抢占锁，则持有偏向锁的线程会被挂起，JVM 会

消除它身上的偏向锁，将锁恢复到标准的轻量级锁。

轻量级锁：轻量级锁是由偏向所升级来的，偏向锁运行在一个线程进入同步块的情况下，当第二个线程加入锁争用的时候，偏向锁就会升级为轻量级锁；

重量级锁：我们知道，我们要进入一个同步、线程安全的方法时，是需要先获得这个方法的锁的，退出这个方法时，则会释放锁。如果获取不到这个锁的话，意味着有别的线程在

执行这个方法，这时我们就会马上进入阻塞的状态，等待那个持有锁的线程释放锁，然后再把我们从阻塞的状态唤醒，我们再去获取这个方法的锁。这种获取不到锁就马上进入阻

塞状态的锁，我们称之为重量级锁。

参考文章：https://blog.csdn.net/zqz_zqz/article/details/70233767

参考文章：https://www.cnblogs.com/myseries/p/10773078.html

8. HTTP 有哪些问题，加密算法有哪些，针对不同加密方式可能产生的问题，及其 HTTPS 是如何保证安全传输的

HTTP 的不足：

通信使用明文，内容可能会被窃听；

不验证通信方的身份，因此有可能遭遇伪装；

无法证明报文的完整性，有可能已遭篡改；

常用加密算法：MD5 算法、DES 算法、AES 算法、RSA 算法

参考文章：https://blog.csdn.net/baidu_22254181/article/details/82594072

蚂蚁 Java 二面

1. 设计模式有哪些大类，及熟悉其中哪些设计模式

创建型模式、结构型模式、行为型模式

参考文章：http://c.biancheng.net/design_pattern/

2. volatile 关键字，他是如何保证可见性，有序性

volatile 可以保证线程可见性且提供了一定的有序性，但是无法保证原子性。在 JVM 底层 volatile 是采用“内存屏障”来实现的。

观察加入 volatile 关键字和没有加入 volatile 关键字时所生成的汇编代码发现，加入 volatile 关键字时，会多出一个 lock 前缀指令，

lock 前缀指令实际上相当于一个内存屏障（也成内存栅栏），内存屏障会提供 3 个功能：

I. 它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内

存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；

II. 它会强制将对缓存的修改操作立即写入主存；

III. 如果是写操作，它会导致其他 CPU 中对应的缓存行无效。

参考文章：https://blog.csdn.net/summerZBH123/article/details/80547516

3. Java 的内存结构，堆分为哪几部分，默认年龄多大进入老年代

Java 的内存结构：程序计数器、虚拟机栈、本地方法栈、堆、方法区。

Java 虚拟机根据对象存活的周期不同，把堆内存划分为几块，一般分为新生代、老年代和永久代。

默认的设置下，当对象的年龄达到 15 岁的时候，也就是躲过 15 次 Gc 的时候，他就会转移到老年代中去躲过 15 次 GC 之后进入老年代。

4. ConcurrentHashMap 如何保证线程安全，jdk1.8 有什么变化

JDK1.7：使用了分段锁机制实现 ConcurrentHashMap，ConcurrentHashMap 在对象中保

存了一个 Segment 数组，即将整个 Hash 表划分为多个分段；

而每个 Segment 元素，即每个分段则类似于一个 Hashtable；这样，在执行 put 操作时首先根据 hash 算法定位到元素属于哪个 Segment，然后对该

Segment 加锁即可。因此，ConcurrentHashMap 在多线程并发编程中可是实现多线程 put 操作，不过其最大并发度受 Segment 的个数限制。

JDK1.8： 底层采用数组+链表+红黑树的方式实现，而加锁则采用 CAS 和 synchronized 实现

参考文章：https://blog.csdn.net/weixin_44460333/article/details/86770169

5. 为什么 ConcurrentHashMap 底层为什么要红黑树

因为发生 hash 冲突的时候，会在链表上新增节点，但是链表过长的话会影响检索效率，引入红黑书可以提高插入和查询的效率。

6. 如何做的 MySQL 优化

MySQL 的优化有多种方式，我们可以从以下几个方面入手：

存储引擎的选择、字段类型的选择、索引的选择、分区分表、主从复制、读写分离、SQL 优化。详细优化请查看参考文章

参考文章：https://blog.csdn.net/zls986992484/article/details/52860496

7. 讲一下 oom 以及遇到这种情况怎么处理的，是否使用过日志分析工具

OOM，全称“Out Of Memory”，翻译成中文就是“内存用完了”，当 JVM 因为没有足够的内存来为对象分配空间并且垃圾回收器也已经没有空间可回收时，就会抛出这个 error。

处理过程：首先通过内存映射分析工具 如 Eclipse Memory Analyzer 堆 dump 出的异常堆转储进行快照解析确认内存中的对象是否是必要的，

也就是先分清楚是 内存泄漏 Memory Leak 还是 Memory Overflow 如果是内存泄漏 可通过工具进一步查看泄露的对象到 GC Roots 的引用链，

就能找到泄露对象是怎么通过路径与 GC Roots 相关联导致垃圾收集器无法回收他们如果不存在泄露 就检查堆参数 -Xmx 与 -Xms 与机器物理

内存对比是否还可以调大 从代码上检测 是否是某些对象的生命周期过长持有状态时间过长 尝试减少代码运行期间的内存消耗。

参考文章：https://www.cnblogs.com/ThinkVenus/p/6805495.html

蚂蚁 Java 三面

1. 项目介绍
2. 你们怎么保证 Redis 缓存和数据库的数据一致性？

可以通过双删延时策略来保证他们的一致性。

参考文章：https://blog.kido.site/2018/12/07/db-and-cache-02/

3. Redis 缓存雪崩？击穿？穿透？

缓存雪崩：缓存同一时间大面积的失效，所以，后面的请求都会落到数据库上，造成数据库短时间内承受大量请求而崩掉。

缓存击穿：key 对应的数据存在，但在 redis 中过期，此时若有大量并发请求过来，这些请求发现缓存过期一般都会从后端 DB 加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端 DB 压垮。

缓存穿透：key 对应的数据在数据源并不存在，每次针对此 key 的请求从缓存获取不到，请求都会到数据源，从而可能压垮数据源。比如用一个不存在的用户 id 获取用户信息，不论缓存还是数据库都没有，若黑客利用此漏洞进行攻击可能压垮数据库。

4. 你熟悉哪些消息中间件,有做过性能比较?

RocketMQ、RabbitMQ、ActiveMQ、Kafka

参考文章：https://blog.csdn.net/wqc19920906/article/details/82193316