# 字节面试题

## 一、算法篇

### 1. 链表排序

**原题地址**

1. 对链表做插入排序 https://leetcode-cn.com/problems/insertion-sort-list/
2. 排序链表 https://leetcode-cn.com/problems/sort-list/

**解答**

对链表做插入排序，模仿数组插入排序的思路，复杂度O(n^2)

class Solution:
 def insertionSortList(self, head: ListNode) -> ListNode:
 if not head:
 return head

 dummyHead = ListNode(0)
 dummyHead.next = head
 lastSorted = head
 curr = head.next

 while curr:
 if lastSorted.val <= curr.val:
 lastSorted = lastSorted.next
 else:
 prev = dummyHead
 while prev.next.val <= curr.val:
 prev = prev.next
 lastSorted.next = curr.next
 curr.next = prev.next
 prev.next = curr
 curr = lastSorted.next

 return dummyHead.next

利用归并排序的思路，时间复杂度：O(nlogn)，其中 n 是链表的长度。空间复杂度：O(1)。

class Solution:
 def sortList(self, head: ListNode) -> ListNode:
 def merge(head1: ListNode, head2: ListNode) -> ListNode:
 dummyHead = ListNode(0)
 temp, temp1, temp2 = dummyHead, head1, head2
 while temp1 and temp2:
 if temp1.val <= temp2.val:
 temp.next = temp1
 temp1 = temp1.next
 else:
 temp.next = temp2
 temp2 = temp2.next
 temp = temp.next
 if temp1:
 temp.next = temp1
 elif temp2:
 temp.next = temp2
 return dummyHead.next

 if not head:
 return head

 length = 0
 node = head
 while node:
 length += 1
 node = node.next

 dummyHead = ListNode(0, head)
 subLength = 1
 while subLength < length:
 prev, curr = dummyHead, dummyHead.next
 while curr:
 head1 = curr
 for i in range(1, subLength):
 if curr.next:
 curr = curr.next
 else:
 break
 head2 = curr.next
 curr.next = None
 curr = head2
 for i in range(1, subLength):
 if curr and curr.next:
 curr = curr.next
 else:
 break

 succ = None
 if curr:
 succ = curr.next
 curr.next = None

 merged = merge(head1, head2)
 prev.next = merged
 while prev.next:
 prev = prev.next
 curr = succ
 subLength <<= 1

 return dummyHead.next

### 2. 手写LRU

**原题地址**

1. LRU缓存机制

https://leetcode-cn.com/problems/lru-cache

**解答**

使用哈希表+双向链表

class DLinkedNode:
 def \_\_init\_\_(self, key=0, value=0):
 self.key = key
 self.value = value
 self.prev = None
 self.next = None

class LRUCache1:

 def \_\_init\_\_(self, capacity: int):
 self.cache = dict()
 # 使用伪头部和伪尾部节点
 self.head = DLinkedNode()
 self.tail = DLinkedNode()
 self.head.next = self.tail
 self.tail.prev = self.head
 self.capacity = capacity
 self.size = 0

 def get(self, key: int) -> int:
 if key not in self.cache:
 return -1
 # 如果 key 存在，先通过哈希表定位，再移到头部
 node = self.cache[key]
 self.moveToHead(node)
 return node.value

 def put(self, key: int, value: int) -> None:
 if key not in self.cache:
 # 如果 key 不存在，创建一个新的节点
 node = DLinkedNode(key, value)
 # 添加进哈希表
 self.cache[key] = node
 # 添加至双向链表的头部
 self.addToHead(node)
 self.size += 1
 if self.size > self.capacity:
 # 如果超出容量，删除双向链表的尾部节点
 removed = self.removeTail()
 # 删除哈希表中对应的项
 self.cache.pop(removed.key)
 self.size -= 1
 else:
 # 如果 key 存在，先通过哈希表定位，再修改 value，并移到头部
 node = self.cache[key]
 node.value = value
 self.moveToHead(node)

 def addToHead(self, node):
 node.prev = self.head
 node.next = self.head.next
 self.head.next.prev = node
 self.head.next = node

 def removeNode(self, node):
 node.prev.next = node.next
 node.next.prev = node.prev

 def moveToHead(self, node):
 self.removeNode(node)
 self.addToHead(node)

 def removeTail(self):
 node = self.tail.prev
 self.removeNode(node)
 return node

### 3. 对称的二叉树

**原题地址**

1. 对称二叉树

https://leetcode-cn.com/problems/symmetric-tree/

**解答**

使用哈希表+双向链表

# Definition for a binary tree node.
class TreeNode:
 def \_\_init\_\_(self, x):
 self.val = x
 self.left = None
 self.right = None

# 递归
class Solution1:
 def isSymmetric(self, root: TreeNode) -> bool:
 def check(p, q):
 if not p and not q: return True
 if not p or not q: return False
 return p.val == q.val and check(p.left, q.right) and check(p.right, q.left)
 return check(root, root)

# 迭代
class Solution3:
 def isSymmetric(self, root: TreeNode) -> bool:
 queue = [root, root]
 while queue:
 u = queue.pop(0)
 v = queue.pop(0)
 if not u and not v: continue
 if (not u or not v) or (u.val != v.val): return False
 queue.append(u.left)
 queue.append(v.right)
 queue.append(u.right)
 queue.append(v.left)
 return True

### 4. 给定一个字符串，删除其中的“ab”

### 5. 二叉树的镜像

### 6. 二叉树层次遍历

### 7. 二叉树对称判断

### 8. 10G 的文件里面是Int32，内存1G，如何找到重复出现的数字

### 9. M 边形N 分

### 10. 求二叉树两点最长距离

### 11. 两数的和（说出来后会继续问3 个数的和，4 个数的和）

### 12. 二叉树的两个节点之间的最短路径

### 13. 现有5 个抽屉分别装有1，2，4，1，3 个糖果，小明有三次取糖果的机会，而且每次只能从相邻的抽屉取，问小明要从哪个抽屉开始取糖果能获取最多糖果？

### 14. 某公司有n 个工区连成一个环形，每个工区一定量的汽油gas[i]，现有一辆油车装油量不受限制，油车从i 工区到i+1 工区需要消耗汽油量是consume[i]，gas = [1，3，5，2，3，4]，consume[1，4，2，3，2，3]，问油车从什么工区开始能走完全程？

### 15. 知道哪些排序并且有什么区别

### 16. N 皇后问题

### 17. 小明要在街上一排互相相邻的糖果屋拿糖，但不能在两个相邻的糖果屋拿糖。给定一个数组列表，每个元素代表每间房子中的糖的数目，小明能看到所有屋的数据，走一遍所有糖果屋，最多能拿多少糖？

### 18. 两个栈实现一个队列

## Java篇

### 1. java 如何制造OOM

// 制造堆的oom
public class MakeOomHeap {
 static class OOMObject {
 }

 public static void main(String[] args) throws InterruptedException {
 List<OOMObject> list = new ArrayList<>();

 while(true) {
// TimeUnit.MILLISECONDS.sleep(1);
 list.add(new OOMObject());
 }
 }
}

// 制造栈的oom
public class MakeOomStack {
 public static void infiniteRun() {
 while(true) {
 Thread thread = new Thread(() -> {

 while (true) {
 try {
 TimeUnit.HOURS.sleep(1);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }

 });
 thread.start();
 }
 }

 public static void main(String[] args) {
 infiniteRun();
 }
}

参考资料：https://www.cnblogs.com/huangqingshi/p/13336648.html

### 2. synchronized 的底层实现

参考资料：https://www.cnblogs.com/paddix/p/5367116.html

### 3. java AQS j.u.c

todo

### 4. java 如何判断是个对象是垃圾

分代回收 复制算法，标记整理算法

### 5. 你们公司为什么还用java8,你对高版本的什么特性觉得好

### 6. 微服务化及springcloud 相关组件，服务注册发现，配置中心，熔断降级

### 7. 重排序细节

### 8. java 泛型的内部实现

### 9. volatile 关键字java 是怎么实现的

是什么？ - volatile是一种同步机制，比synchronized或者Lock相关类更轻量，因为使用volatile并不会发生上下文切换等开销很大的行为。 - 如果一个变量被修饰成volatile，那么JVM就知道了这个变量可能会被并发修改 - 但是开销小，相应的能力也小，虽然说volatile是用来同步的保证线程安全的，但是volatile做不到synchronized那样的原子保护，volatile仅在很有限的场景下才能发挥作用。

不适用场景：a++ 适用场合： - boolean flag，如果一个共享变量自始至终只被各个线程赋值，而没有其他的操作，那么就可以用volatile来代替synchronized或者代替原子变量，因为赋值自身是有原子性的，而volatile又保证了可见性，所以就足以保证线程安全。（不一定是布尔变量） - 作为刷新之前变量的触发器（FieldVisibility）

作用： - 可见性：读一个volatile变量之前，需要先使相应的本地缓存失效，这样就必须到主内存读取最新值，写一个volatile属性会立即刷入到主内存。 - 禁止指令重排序优化：解决单例双重锁乱序问题

volatile和Synchronized的关系 - volatile在这方面可以看做是轻量版的synchronized：如果一个共享变量自始至终只被各个线程赋值，而没有其他的操作，那么就可以用volatile来代替synchronized或者代替原子变量，因为赋值自身是有原子性的，而volatile又保证了可见性，所以就足以保证线程安全

### 10. 锁的内部实现（Synchronize关键字内部实现）

对象锁 - 包括方法锁（默认锁对象为this当前实例对象）和同步代码块锁（自己指定锁对象）

类锁 - 指synchronized修饰静态的方法或指定锁为Class对象

第一个用法：对象锁

代码块形式：手动指定锁对象 方法锁形式：Synchronized修饰普通方法，锁对象默认为this

第二个用法：类锁

概念（重要）：Java类可能有很多个对象，但只有1个Class对象 本质：所以所谓的类锁，不过是Class对象的锁而已 用法和效果：类锁只能在同一时刻被一个对象拥有

每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权。

Synchronized的语义底层是通过一个monitor的对象来完成，其实wait/notify等方法也依赖于monitor对象，这就是为什么只有在同步的块或者方法中才能调用wait/notify等方法，否则会抛出java.lang.IllegalMonitorStateException的异常的原因。

JVM会自动通过使用monitor来加锁和解锁，保证了同时只有一个线程可以执行指定代码，从而保证了线程安全，同时具有可重入和不可中断的性质。

### 11. dubbo 基本原理

### 12. RPC 调用原理

### 13. 描述java 虚拟机的内存模型,有哪些垃圾回收算法，垃圾回收器G1，G2 的区别

### 14. 强引用软引用弱引用虚引用

### 15. volatile 关键字作用和实现原理

### 16. 闭包

### 17. JVM GC 时间长如何发现、后来改成问成接口慢如何排查原因

### 18. Java 对象结构

### 19. 对象如何创建，创建对象还有那些方式

### 20. 内存结构，那些是线程私有的，那些是共有的

### 21. 内存模型、CPU 有这套架构，为什么JVM 实现一遍

### 22. 线程切换的上下文要装载什么

### 23. 有没有JVM 调优的经验、如何JVM 调优，Dump 日志如何分析

### 24. 为什么要避免FullGC

### 25. 新生代垃圾收集算法，会不会STW

### 26. 垃圾回收算法有那些

### 27. CMS 垃圾回收针对哪些部分

### 28. 老年代GC 和FullGC 的关系

### 29. Java 读写锁如何实现

### 30. ThreadPoolExecutor 有哪些参数

### 31. 介绍一下ThreadLocal

### 32. 使用ThreadLocal 存在什么问题，有什么风险（向面试官要提示的时候，问了Java 中的引用类型）

### 33. Java 中有哪些垃圾回收器，CMS 的垃圾回收过程

## 数据库篇

### 1. mysql 中的乐观锁和悲观锁实现

### 2. mysql explain 的作用

### 3. mysql 的主从复制

### 4. mysql 的引擎区别

### 5. 一个表user\_id,order\_date,要查用户订单,某一天订单,某个用户某天订单,如何建索引

### 6. 复合索引的结构

### 7. mysql 索引的数据结构

### 8. 事务的基本特征，事务的隔离级别

### 9. 怎么分析优化慢查询

### 10. mysql 索引结构，B+树，IN 是否参与索引，最左查询

### 11. 乐观锁和悲观锁

### 12. 数据库参数调优

### 13. MySQL 如何避免幻读

### 14. MySQL 锁机制是怎样的

### 15. 如果避免、减少锁等待、团队中如何监控MySQL 的锁等待的情况

### 16. MySQL 为什么采用B+树而不用B 树

### 17. MySQL 有哪些锁

### 18. 索引什么时候不生效

## 中间件篇

### 1. redis 数据类型

### 2. redis 的aof 太大如何优化

### 3. nginx 如何处理连接

### 4. nginx 如何做性能优化

### 5. apache 和nginx 区别

### 6. zset 的底层数据结构,跳跃表如何实现

### 7. redis 如何持久化RDB

### 8. memcache 与redis 区别

### 9. redis对长字符串和短字符串处理有什么不同

### 10. redis cluster 如何搭建，master-slave 模式，如何选主

### 11. redis 过期策略

### 12. Reids 如何实现分布式锁

### 13. Reids 如何持久化方式

### 14. 如何避免Redis 缓存穿透

### 15. 分布式锁如何实现，Redis 如何实现分布式锁

### 16. 介绍下Redis 中的哨兵机制

## 网络篇

### 1. https 的建立链接过程

### 2. https 的传输数据是否是对称加密

### 3. tcp 拥塞控制

### 4. socket 建立过程

### 5. 如何高效处理socket

### 6. NIO 的原理和,连接切换方式

### 7. 网页加载时如何重定向

### 8. 长连接短连接区别，应用场景

### 9. TCP 如何保证可靠性，解释一下滑动窗口

## 操作系统篇

### 1. epoll 和select

### 2. 查看系统负载。使用什么命令查看系统负载,第一行是什么

### 3. 进程通信方法

### 4. 管道如何使用

### 5. 进程和线程

进程是资源分配的最小单位，线程是CPU调度的最小单位

* 所有与进程相关的资源，都被记录在PCB中
* 进程是抢占处理机的调度单位；线程属于某个进程，共享其资源
* 线程只由堆栈寄存器、程序计数器和TCB组成

总结： - 线程不能看做独立应用，而进程可看做独立应用 - 进程有独立的地址空间，相互不影响，线程只是进程的不同执行路径 - 线程没有独立的地址空间，多进程的程序比多线程程序健壮 - 进程的切换比线程的切换开销大

## 系统设计篇

### 1. 服务器降级方案，如何指定降级优先级

todo

### 2. redis 集群架构方案

### 3. RPC 服务的底层架构，线程管理

### 4. 如何找项目性能瓶颈

### 5. 有多少种类型的缓存, 从客户端到服务端

### 6. 是否遇到分布式事务，有哪些解决方案，分布式锁怎么做，用redis 实现分布式锁有哪些步骤

MySQL查询缓存

### 7. 如何查看系统性能，性能指标

### 8. im 里面群聊消息体设计，如何设计xx 人已读xx 人未读

### 9. 微服务分布式事务如何实现

### 10. 如何设计秒杀系统，应该关注什么：防恶意提交、NG 限流等

### 11. 微服务的某个节点挂了怎么办

## 其他篇

### 1. 两支形状不规则的香，都能烧1小时，如何用这两支香测出15分钟？

### 2. 项目中的难点是什么？

### 3. 日常工作各项工作内容时间比例是怎样的

### 4. 团队的规模是怎样，在团队里承担什么角色

### 5. 描述项目的技术架构，都采用哪些技术

### 6. 怎么做微服务拆分

### 7. 遇到哪些大的挑战和工作中的亮点

### 8. 以后职业规划如何

## HR 面

### 1. 如何你是owner 你是怎么设计这个项目

### 2. 讲一下项目中沟通协作的过程