拼多多

算法

1. 快速排序的空间复杂度是多少？时间复杂度的最好最坏的情况是多少，有哪些优化方案?
2. 给定 100G 的 URL 磁盘数据，使用最多 1G 内存，统计出现频率最高的 Top K 个 URL
3. 删除链表的倒数第 N 个结点
4. 多数元素 (Leetcode)
5. 二叉树的后序遍历 (Leetcode)
6. LRU 缓存机制
7. 反转链表
8. 翻转二叉树 (Leetcode)
9. 常用的排序方式有哪些，时间复杂度是多少?
10. 二叉树的层序遍历
11. 什么是排序算法中的稳定性?
12. 用队列实现栈 (Leetcode)
13. 字符串解码
14. 红黑树是怎么实现平衡的？它的优点是什么?
15. 不同路径 (Leetcode)
16. 最长和谐子序列 (Leetcode)
17. 二叉树的中序遍历 (Leetcode)
18. 手写无锁队列

Java

1. Java 异常有哪些类型?
2. Java 中垃圾回收机制中如何判断对象需要回收？常见的 GC 回收算法有哪些?
3. 简述 BIO, NIO, AIO 的区别
4. JVM 内存是如何对应到操作系统内存的?
5. 如何设计 Java 的异常体系?
6. 简述 Synchronized，Volatile，可重入锁的不同使用场景及优缺点
7. 如何设计一个线程池
8. Java 类的加载流程是怎样的？什么是双亲委派机制?
9. Spring MVC 的原理和流程
10. 简述常见的工厂模式以及单例模式的使用场景
11. 什么是内存泄漏，怎么确定内存泄漏?
12. 实现单例模式
13. 简述 Java 的反射机制及其应用场景

网络协议

简述 TCP 三次握手以及四次挥手的流程。为什么需要三次握手以及四次挥手? ▵ 26次
TCP 怎么保证可靠传输?
简述 HTTP 1.0，1.1，2.0 的主要区别
TCP 与 UDP 在网络协议中的哪一层，他们之间有什么区别?
HTTP 与 HTTPS 有哪些区别?
如何防止传输内容被篡改?
从输入 URL 到展现页面的全过程
HTTP 是无状态的吗？需要保持状态的场景应该怎么做?
简述 HTTPS 的加密与认证过程
简述 RPC 的调用过程
什么是 ARP 协议？简述其使用场景
简述 TCP 滑动窗口以及重传机制
TCP 如何实现数据有序性?

数据库

数据库的事务隔离级别有哪些？各有哪些优缺点？ 中等 参考1 参考2
MySQL 为什么使用 B+ 树来作索引，对比 B 树它的优点和缺点是什么？ 中等 参考1 参考2
Redis的缓存淘汰策略有哪些？ 简单 参考1 参考2
Redis 如何淘汰过期 Key
为什么 Redis 在单线程下能如此快？ 中等 参考1
简述 Redis 持久化中 RDB 以及 AOF 方案的优缺点 困难 参考1 参考2

- 什么是数据库事务，MySQL 为什么会使用 InnoDB 作为默认选项 中等 参考1 参考2
- 数据库的读写分离的作用是什么？如何实现？ 中等
- 聚簇索引和非聚簇索引有什么区别？ 简单 参考1 参考2
- 数据库索引的实现原理是什么？ 简单
- 如何解决缓存与数据库不一致的问题？
- 简述数据库中的 ACID 分别是什么？ 简单
- 数据库索引的叶子结点为什么是有序链表？

操作系统

- 进程和线程之间有什么区别？ 简单 参考1 参考2
- 进程间有哪些通信方式？ 困难 参考1 参考2管道
- 简述几个常用的 Linux 命令以及他们的功能 简单
- 简述 Linux 进程调度的算法 困难 参考1
- 简述 select, poll, epoll 的使用场景以及区别，epoll 中水平触发以及边缘触发有什么不同？ 困难 参考1 参考2

系统设计

- 项目上有什么技术难点？
- 了解高并发的解决方案吗？例如动静分离，缓存，负载均衡
- 有哪些实现服务发现的方法？
- 如何阅读大型项目的源码？

前端

- readyState 的不同返回值有什么区别？
- 简述什么是 XSS 攻击以及 CSRF 攻击？ 中等 参考1 参考2
- 简述 React 的生命周期
- promise 有哪些状态？简述 promise.all 的实现原理 中等
- HTTP 中 GET 和 POST 区别 简单 参考1
- promise 有哪些状态？简述 promise.all 的实现原理 中等
- CSS 实现三列布局 简单
- 什么是可继承元素和不可继承元素？
- 什么是替换元素与非替换元素
- 简述 ES6 的新特性 简单
- 正则表达式 /w 是什么意思？

C++

- 简述 C++ 中智能指针的特点，简述 new 与 malloc 的区别 中等 参考1 参考2
- 指针和引用的区别是什么？
- C++ 11 有什么新特性 简单 参考1 参考2
- C++ 中虚函数与纯虚函数的区别 简单
- STL 中 vector 与 list 具体是怎么实现的？常见操作的时间复杂度是多少？ 简单 参考1 参考2
- C++ 中智能指针和指针的区别是什么？ 简单 参考1 参考2
- 简述 C++ 中内存对齐的使用场景 中等
- 简述 vector 的实现原理 简单

非技术

- 未来的职业规划是什么？
- 能接受多大强度的加班？
- 你用过拼多多吗？说说对对拼多多的看法，使用上有哪些优缺点
- 对下一家入职的公司主要看重哪些方面？
- 自己有哪些不足之处，准备如何提升？

快手

算法

- 141. 环形链表 简单
- 使用递归及非递归两种方式实现快速排序 中等 参考1 参考2
- 102. 二叉树的层序遍历 中等
- 合并两个有序数组 (Leetcode)
- 3. 无重复字符的最长子串 中等
- 53. 最大子序和 简单
- 19. 删除链表的倒数第 N 个结点 中等
- 对角线遍历 (Leetcode)
- 215. 数组中的第K个最大元素 中等
- 如何从一个数组输出随机数组（洗牌算法）

- 206. 反转链表 简单
- 简述布隆过滤器原理及其使用场景
- 合并两个有序数组 (Leetcode)
- 多数元素 (Leetcode)
- 用队列实现栈 (Leetcode)
- 常用的限流算法有哪些？简述令牌桶算法原理 中等 参考1 参考2
- 有效的括号 (Leetcode)
- 旋转图像 (Leetcode)
- 69. x 的平方根
- 剑指 Offer 10- II. 青蛙跳台阶问题
- 最小覆盖子串 (Leetcode)
- 最大栈 (Leetcode)
- 二叉树的后序遍历 (Leetcode)
- 螺旋矩阵
- 二叉搜索树的第k大节点（剑指 Offer 54）
- 整数反转 (Leetcode)

Java

- HashMap 与 ConcurrentHashMap 的实现原理是怎样的？ConcurrentHashMap 是如何保证线程安全的？ 中等 参考1 参考2 参考3
- 简述 CAS 原理，什么是 ABA 问题，怎么解决？
- Java 中垃圾回收机制中如何判断对象需要回收？常见的 GC 回收算法有哪些？ 中等 参考1 参考2 参考3
- 简述 Synchronized，Volatile，可重入锁的不同使用场景及优缺点 困难 参考1
- Synchronized 关键字底层是如何实现的？它与 Lock 相比优缺点分别是什么？ 中等 参考1 参考2
- 线程池是如何实现的？简述线程池的任务策略 中等
- 简述 ArrayList 与 LinkedList 的底层实现以及常见操作的时间复杂度 简单
- 简述 BIO, NIO, AIO 的区别
- == 和 equals() 的区别？ 简单 参考1
- Java 常见锁有哪些？ReetrantLock 是怎么实现的？ 中等
- JMM 中内存模型是怎样的？什么是指令序列重排序？ 中等 参考1
- 实现单例设计模式（懒汉，饿汉） 中等 参考1 参考2
- volatile 关键字解决了什么问题，它的实现原理是什么？ 中等 参考1
- JVM 内存是如何对应到操作系统内存的？ 中等 参考1 参考2
- 简述 Spring bean 的生命周期 中等 参考1 参考2
- 简述 Java 锁升级的机制
- 简述 Java AQS 的原理以及使用场景
- 简述 Spring AOP 的原理 中等 参考1 参考2
- Java 中接口和抽象类的区别 简单
- 如何解决 Spring 的循环依赖问题？
- 简述 Spring 注解的实现原理
- 什么是公平锁？什么是非公平锁？ 简单
- 在一个静态方法内调用一个非静态成员为什么是非法的？
- 什么是设计模式，描述几个常用的设计模式 中等
- Java 异常有哪些类型？

网络协议

- 简述 TCP 三次握手以及四次挥手的流程。为什么需要三次握手以及四次挥手？ 中等 参考1 参考2
- 从输入 URL 到展现页面的全过程 困难 参考1
- DNS 查询服务器的基本流程是什么？DNS 劫持是什么？ 中等 参考1 参考2
- TCP 怎么保证可靠传输？ 中等 参考1 参考2
- 什么是 TCP 粘包和拆包？ 简单
- 简述 HTTPS 的加密与认证过程 中等 参考1
- TCP 与 UDP 在网络协议中的哪一层，他们之间有什么区别？ 简单 参考1
- 简述对称与非对称加密的概念 简单
- 如何解决 TCP 传输丢包问题？

数据库

- MySQL 为什么使用 B+ 树来作索引，对比 B 树它的优点和缺点是什么？ 中等 参考1 参考2
- 简述一致性哈希算法的实现方式及原理 困难
- 简述什么是最左匹配原则 简单

- 为什么 Redis 在单线程下能如此快？ 中等 参考1
- MySQL 联合索引底层原理是什么？
- 数据库的事务隔离级别有哪些？各有哪些优缺点？ 中等 参考1 参考2
- Redis 如何实现分布式锁？ 困难 参考1 参考2
- 什么情况下会发生死锁，如何解决死锁？ 中等 参考1 参考2
- Redis 有几种数据结构？Zset 是如何实现的？ 中等 参考1 参考2
- 简述常见的负载均衡算法 简单 参考1
- 简述 MySQL 的主从同步机制，如果同步失败会怎么样？ 中等
- MySQL中 InnoDB 和 MyISAM 的区别是什么？ 简单
- 聚簇索引和非聚簇索引有什么区别？ 简单 参考1 参考2
- 简述乐观锁以及悲观锁的区别以及使用场景 简单 参考1 参考2
- 简述 MySQL 常见索引类型，介绍一下覆盖索引
- 如何设计数据库压测方案？
- Redis 的 String 数据类型是如何实现的？

操作系统

- 进程间有哪些通信方式？ 困难 参考1 参考2
- 简述自旋锁与互斥锁的使用场景 中等
- 进程和线程之间有什么区别？ 简单 参考1 参考2

前端

- 手写题库 https://github.com/Mayandev/fe-interview-handwrite 困难
- const, let, var 关键字有什么区别？ 简单
- 简述 Javascript 原型以及原型链 中等 参考1
- 简述 diff 算法的实现机制和使用场景 中等
- 简述 CSS 盒模型 简单 参考1
- CSS 的选择器优先级是怎样？ 简单
- 简述 Dom 节点的不同操作方式 中等
- 简述 Javascript 中 this 的指向有哪些 简单
- 前端如何解决线程安全和进程安全的问题？
- 简述 JavaScript 事件循环机制 中等 参考1
- 简述 Flex 布局的原理和使用场景 中等
- 简述常见异步编程方案 (promise, generator, async) 的原理 中等
- 如何使用 flex 实现两栏布局？
- 简述 ES6 的新特性 简单
- 简述 watch 和 computed 的区别 简单
- 简述浏览器的缓存机制 中等
- 简述 Javascript 中的防抖与节流的原理并尝试实现 中等
- Vuex 有哪些常用属性？
- 简述 CSS 预编译的方式
- 什么是 JavaScript 的变量提升？有什么作用？
- 简述 Vue 的生命周期 中等
- 简述强缓存与协商缓存的区别和使用场景 中等
- 简述 Javascript 的数据类型
- 简述 BFC 的原理及其使用场景 中等
- history 和 hash 的区别是什么？

系统设计

- 电商系统中，如何实现秒杀功能？如何解决商品的超卖问题？ 困难
- 如何解决缓存与数据库不一致的问题？
- 简述中间件削峰和限流的使用场景
- 项目上有什么技术难点？

非技术

- 最近在看什么书吗，有没有接触过什么新技术？
- 你用过快手吗？说说对对快手的看法，使用上有哪些优缺点
- 实习的内容是什么？最大收获是什么？
- 未来的职业规划是什么？
- 你的性格和技能上有什么缺点
- 简单描述一下自己是怎么样的人？
- 为什么想要来快手？
- 学习中遇到的最大的困难是什么？

最有成就感的项目是什么?
能接受多大强度的加班?

拼多多

算法

1. 快速排序的空间复杂度是多少？时间复杂度的最好最坏的情况是多少，有哪些优化方案?

时间复杂度的最好最坏的情况

算法	平均时间	最好时间	最坏时间	空间
快速排序	O(n * log(n))	O(n * log(n))	O(n^2)	O(log(n))

除了快速排序，像插入排序，堆排序，桶排序的时间复杂度也是高频的面试题。那么怎么计算的呢？算法的时间复杂度等于所有子步骤的时间复杂度之和，快速排序的效率取决于数组划分是否平衡，即依赖于主元（pivot）的选择。最好的情况，假设主元每次都恰好是待排序数组的中位数，能够把待排序数组平衡地划分为两个相近长度的子数组。我们可以使用递归树的方法来计算。可以看到在每一次递归中，一方面数组长度变小，另一方面数组数量变多，但是每层的总时间复杂度和不变。（ n 为数组的长度）

递归树	每层总时间
1. T(n)	O(n) （主元划分数组的时间）
2. T(n/2) T(n/2)	O(1/2 n) * 2 = O(n)
3. T(n/4) T(n/4) T(n/4) T(n/4)	O(1/4 n) * 4 = O(n)

我们可以看到每一层划分数组的时间都为 O(n)，**这棵树共有 log(n) 层**，所以总的时间复杂度是 O(nlog(n))

优化方案

1. **随机化**

在快速排序最开始的时候先对数组进行随机化，**攻击者无法通过构造一种特殊的输入来触发快速排序的最坏情况。**

\2. **混合排序**

插入排序的最佳时间复杂度是 O(n)，当数组长度少于一定长度的时候，我们可以使用插入排序。

\3. **更聪明地选择主元**

有非常多选择的方法，例如中位数法，为了不要选到待排序数组的极值，可以选择该数组的首，中间，尾数字，然后取其中位数作为主元，

\4. **双主元排序**

与单主元的本质思想是一样的，不过使用了双主元把待排序数组划分为三部分而不是两部分。

\5. **实际应用**

Java 7 使用了双主元排序，Golang 的快速排序综合了 2，3，4 三种方法，在小数据的时候会使用插入排序以及希尔排序，为了避免大数据的栈溢出所以也使用了堆排序，一般的情况下，Golang 会使用双主元的快速排序。

2. 给定 100G 的 URL 磁盘数据，使用最多 1G 内存，统计出现频率最高的 Top K 个 URL

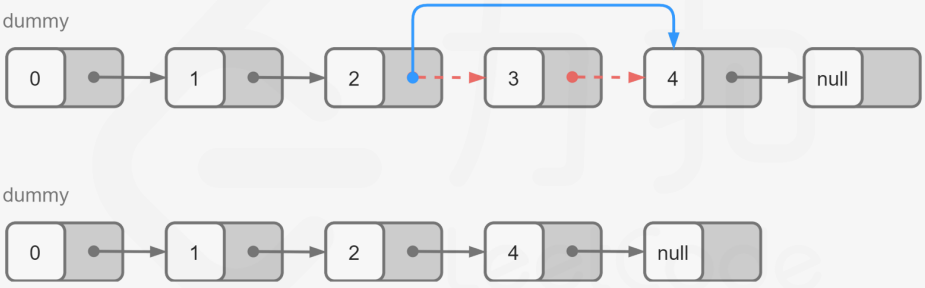
3. 删除链表的倒数第 N 个结点

方法一：计算链表长度
思路与算法

一种容易想到的方法是，我们首先从头节点开始对链表进行一次遍历，得到链表的长度 L。随后我们再从头节点开始对链表进行一次遍历，当遍历到第L−n+1 个节点时，它就是我们需要删除的节点。

> 为了与题目中的 n 保持一致，节点的编号从 1 开始，头节点为编号 1 的节点。

为了方便删除操作，我们可以从哑节点开始遍历 L−n+1 个节点。当遍历到第 L−n+1 个节点时，它的下一个节点就是我们需要删除的节点，这样我们只需要修改一次指针，就能完成删除操作。



代码

```
class Solution {
public:
    int getLength(ListNode* head) {
        int length = 0;
        while (head) {
            ++length;
            head = head->next;
        }
        return length;
    }

    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode* dummy = new ListNode(0, head);
        int length = getLength(head);
        ListNode* cur = dummy;
        for (int i = 1; i < length - n + 1; ++i) {
```

```
            cur = cur->next;
        }
        cur->next = cur->next->next;
        ListNode* ans = dummy->next;
        delete dummy;
        return ans;
    }
};
```

方法二：栈
思路与算法

我们也可以在遍历链表的同时将所有节点依次入栈。根据栈「先进后出」的原则，我们弹出栈的第 nn 个节点就是需要删除的节点，并且目前栈顶的节点就是待删除节点的前驱节点。这样一来，删除操作就变得十分方便了。

代码

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode* dummy = new ListNode(0, head);
        stack<ListNode*> stk;
        ListNode* cur = dummy;
        while (cur) {
            stk.push(cur);
            cur = cur->next;
        }
        for (int i = 0; i < n; ++i) {
            stk.pop();
        }
        ListNode* prev = stk.top();
        prev->next = prev->next->next;
        ListNode* ans = dummy->next;
        delete dummy;
        return ans;
    }
};
```

4. 多数元素 (Leetcode)

5. 二叉树的后序遍历 (Leetcode)

6. LRU 缓存机制

方法一：哈希表 + 双向链表
算法

LRU 缓存机制可以通过哈希表辅以双向链表实现，我们用一个哈希表和一个双向链表维护所有在缓存中的键值对。

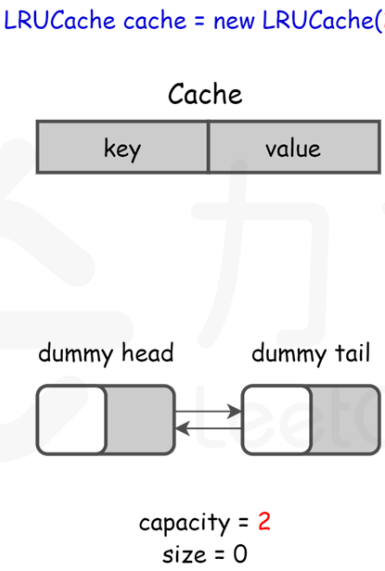
- 双向链表按照被使用的顺序存储了这些键值对，靠近头部的键值对是最近使用的，而靠近尾部的键值对是最久未使用的。
- 哈希表即为普通的哈希映射（HashMap），通过缓存数据的键映射到其在双向链表中的位置。

这样以来，我们首先使用哈希表进行定位，找出缓存项在双向链表中的位置，随后将其移动到双向链表的头部，即可在 O(1) 的时间内完成 get 或者 put 操作。具体的方法如下：

- 对于 get 操作，首先判断 key 是否存在：
  - 如果 key 不存在，则返回 −1；
  - 如果 key 存在，则 key 对应的节点是最近被使用的节点。通过哈希表定位到该节点在双向链表中的位置，并将其移动到双向链表的头部，最后返回该节点的值。
- 对于 put 操作，首先判断 key 是否存在：
  - 如果 key 不存在，使用 key 和 value 创建一个新的节点，在双向链表的头部添加该节点，并将 key 和该节点添加进哈希表中。然后判断双向链表的节点数是否超出容量，如果超出容量，则删除双向链表的尾部节点，并删除哈希表中对应的项；
  - 如果 key 存在，则与 get 操作类似，先通过哈希表定位，再将对应的节点的值更新为 value，并将该节点移到双向链表的头部。

上述各项操作中，访问哈希表的时间复杂度为 O(1)，在双向链表的头部添加节点、在双向链表的尾部删除节点的复杂度也为 O(1)。而将一个节点移到双向链表的头部，可以分成「删除该节点」和「在双向链表的头部添加节点」两步操作，都可以在 O(1) 时间内完成。

> **小贴士**在双向链表的实现中，使用一个伪头部（dummy head）和伪尾部（dummy tail）标记界限，这样在添加节点和删除节点的时候就不需要检查相邻的节点是否存在。



代码

```
class DLinkedNode:
    def __init__(self, key=0, value=0):
        self.key = key
```

```
        self.value = value
        self.prev = None
        self.next = None


class LRUCache:

    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
```

7. 反转链表

- **方法一：迭代**
  假设链表为 1→2→3→∅，我们想要把它改成 ∅←1←2←3。

  在遍历链表时，将当前节点的 next 指针改为指向前一个节点。由于节点没有引用其前一个节点，因此必须事先存储其前一个节点。在更改引用之前，还需要存储后一个节点。最后返回新的头引用。

  代码

```
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode curr = head;
        while (curr != null) {
            ListNode next = curr.next;
            curr.next = prev;
            prev = curr;
            curr = next;
        }
        return prev;
    }
}
```

8. 翻转二叉树 (Leetcode)

9. 常用的排序方式有哪些，时间复杂度是多少?

10. 二叉树的层序遍历

方法一：广度优先搜索
思路和算法

我们可以用广度优先搜索解决这个问题。

我们可以想到最朴素的方法是用一个二元组 (node, level) 来表示状态，它表示某个节点和它所在的层数，每个新进队列的节点的 level 值都是父亲节点的 level 值加一。最后根据每个点的 level 对点进行分类，分类的时候我们可以利用哈希表，维护一个以 level 为键，对应节点值组成的数组为值，广度优先搜索结束以后按键 level 从小到大取出所有值，组成答案返回即可。

考虑如何优化空间开销：如何不用哈希映射，并且只用一个变量 node 表示状态，实现这个功能呢？

我们可以用一种巧妙的方法修改广度优先搜索：

- 首先根元素入队
- 当队列不为空的时候
    - 求当前队列的长度 sis
    - 依次从队列中取 si个元素进行拓展，然后进入下一次迭代

它和普通广度优先搜索的区别在于，普通广度优先搜索每次只取一个元素拓展，而这里每次取s i个元素。在上述过程中的第 ii 次迭代就得到了二叉树的第 ii 层的si个元素。

为什么这么做是对的呢？我们观察这个算法，可以归纳出这样的循环不变式：第 ii 次迭代前，队列中的所有元素就是第 ii 层的所有元素，并且按照从左向右的顺序排列。证明它的三条性质（你也可以把它理解成数学归纳法）：

- 初始化：i = 1的时候，队列里面只有 root，是唯一的层数为 1 的元素，因为只有一个元素，所以也显然满足「从左向右排列」；
- 保持：如果 i = k 时性质成立，即第 kk 轮中出队sk的元素是第 k 层的所有元素，并且顺序从左到右。因为对树进行广度优先搜索的时候由低 k 层的点拓展出的点一定也只能是 k+1 层的点，并且 k+1 层的点只能由第 k 层的点拓展到，所以由这 s k个点能拓展到下一层所有的 sk+1个点。又因为队列的先进先出（FIFO）特性，既然第 kk 层的点的出队顺序是从左向右，那么第 k+1 层也一定是从左向右。至此，我们已经可以通过数学归纳法证明循环不变式的正确性。
- 终止：因为该循环不变式是正确的，所以按照这个方法迭代之后每次迭代得到的也就是当前层的层次遍历结果。至此，我们证明了算法是正确的。

代码

```
   class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector <vector <int>> ret;
        if (!root) {
            return ret;
        }

   queue <TreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        int currentLevelSize = q.size();
        ret.push_back(vector <int> ());
        for (int i = 1; i <= currentLevelSize; ++i) {
            auto node = q.front(); q.pop();
            ret.back().push_back(node->val);
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
    }

    return ret;
```

```
    }
};
```

复杂度分析

记树上所有节点的个数为 n。

时间复杂度：每个点进队出队各一次，故渐进时间复杂度为 O(n)。
空间复杂度：队列中元素的个数不超过 n 个，故渐进空间复杂度为O(n)。

11. 什么是排序算法中的稳定性?

12. 用队列实现栈 (Leetcode)

13. 字符串解码

方法一：栈操作
思路和算法

本题中可能出现括号嵌套的情况，比如 2[a2[bc]]，这种情况下我们可以先转化成 2[abcbc]，在转化成 abcbcabcbc。我们可以把字母、数字和括号看成是独立的 TOKEN，并用栈来维护这些 TOKEN。具体的做法是，遍历这个栈：

- 如果当前的字符为数位，解析出一个数字（连续的多个数位）并进栈
- 如果当前的字符为字母或者左括号，直接进栈
- 如果当前的字符为右括号，开始出栈，一直到左括号出栈，出栈序列反转后拼接成一个字符串，此时取出栈顶的数字（此时栈顶一定是数字，想想为什么？），就是这个字符串应该出现的次数，我们根据这个次数和字符串构造出新的字符串并进栈

重复如上操作，最终将栈中的元素按照从栈底到栈顶的顺序拼接起来，就得到了答案。注意：这里可以用不定长数组来模拟栈操作，方便从栈底向栈顶遍历。

代码

```
class Solution {
public:
    string getDigits(string &s, size_t &ptr) {
        string ret = "";
        while (isdigit(s[ptr])) {
            ret.push_back(s[ptr++]);
        }
        return ret;
    }

    string getString(vector <string> &v) {
        string ret;
        for (const auto &s: v) {
            ret += s;
```

```
        }
        return ret;
    }

    string decodeString(string s) {
        vector <string> stk;
        size_t ptr = 0;

        while (ptr < s.size()) {
            char cur = s[ptr];
            if (isdigit(cur)) {
                // 获取一个数字并进栈
                string digits = getDigits(s, ptr);
                stk.push_back(digits);
            } else if (isalpha(cur) || cur == '[') {
                // 获取一个字母并进栈
                stk.push_back(string(1, s[ptr++]));
            } else {
                ++ptr;
                vector <string> sub;
                while (stk.back() != "[") {
                    sub.push_back(stk.back());
                    stk.pop_back();
                }
                reverse(sub.begin(), sub.end());
                // 左括号出栈
                stk.pop_back();
                // 此时栈顶为当前 sub 对应的字符串应该出现的次数
                int repTime = stoi(stk.back());
                stk.pop_back();
                string t, o = getString(sub);
                // 构造字符串
                while (repTime--) t += o;
                // 将构造好的字符串入栈
                stk.push_back(t);
            }
        }

        return getString(stk);
    }
};
```

14. 红黑树是怎么实现平衡的？它的优点是什么？

15. 不同路径 (Leetcode)

16. 最长和谐子序列 (Leetcode)

17. 二叉树的中序遍历 (Leetcode)

18. 手写无锁队列

Java

1. Java 异常有哪些类型?

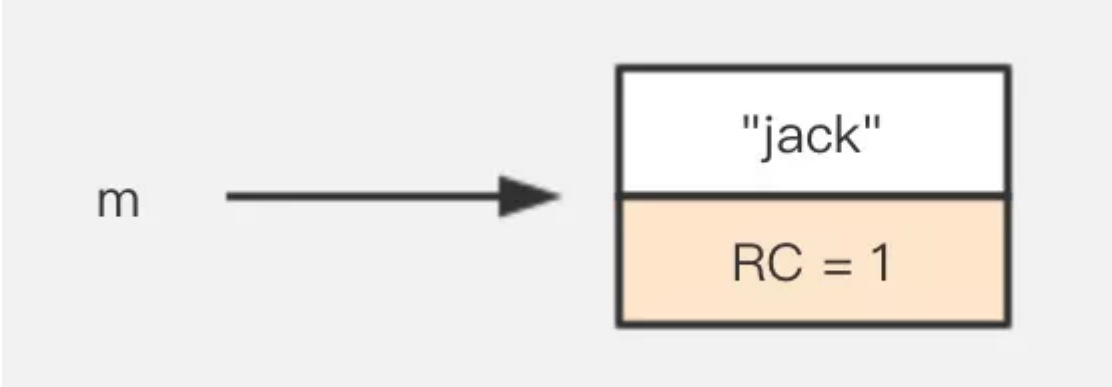
2. Java 中垃圾回收机制中如何判断对象需要回收？常见的 GC 回收算法有哪些?

引用计数算法

引用计数算法（Reachability Counting）是通过在对象头中分配一个空间来保存该对象被引用的次数（Reference Count）。如果该对象被其它对象引用，则它的引用计数加1，如果删除对该对象的引用，那么它的引用计数就减1，当该对象的引用计数为0时，那么该对象就会被回收。

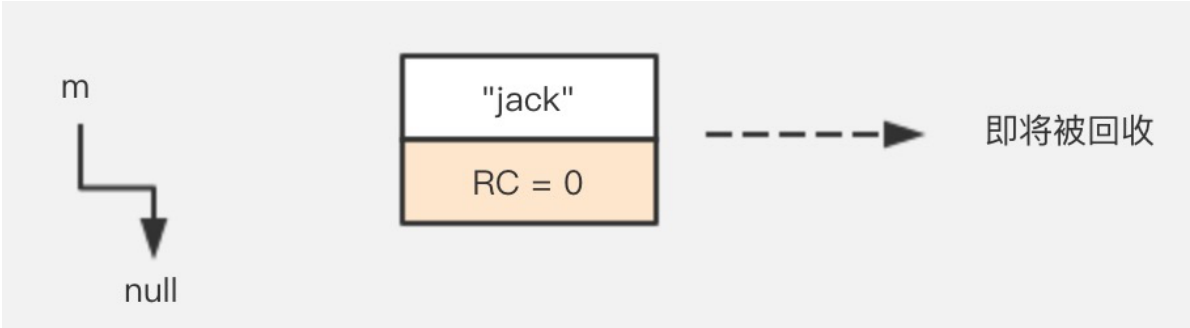
```
String m = new String("jack");
```

先创建一个字符串，这时候"jack"有一个引用，就是 m。



然后将 m 设置为 null，这时候"jack"的引用次数就等于0了，在引用计数算法中，意味着这块内容就需要被回收了。

```
m = null;
```



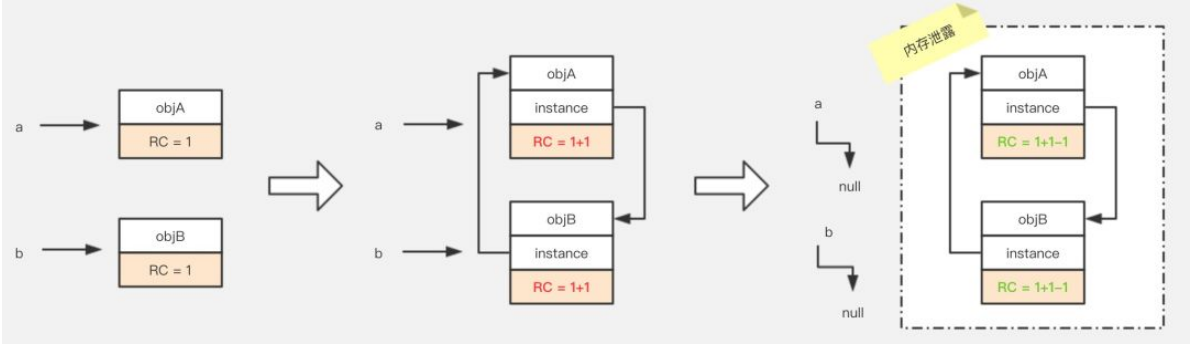
引用计数算法是将垃圾回收分摊到整个应用程序的运行当中了，而不是在进行垃圾收集时，要挂起整个应用的运行，直到对堆中所有对象的处理都结束。因此，采用引用计数的垃圾收集不属于严格意义上的"Stop-The-World"的垃圾收集机制。

看似很美好，但我们知道JVM的垃圾回收就是"Stop-The-World"的，那是什么原因导致我们最终放弃了引用计数算法呢？看下面的例子。

```
public class ReferenceCountingGC {

    public Object instance;

    public ReferenceCountingGC(String name){}
}

public static void testGC(){

    ReferenceCountingGC a = new ReferenceCountingGC("objA");
    ReferenceCountingGC b = new ReferenceCountingGC("objB");

    a.instance = b;
    b.instance = a;

    a = null;
    b = null;
}
```

\1. 定义2个对象

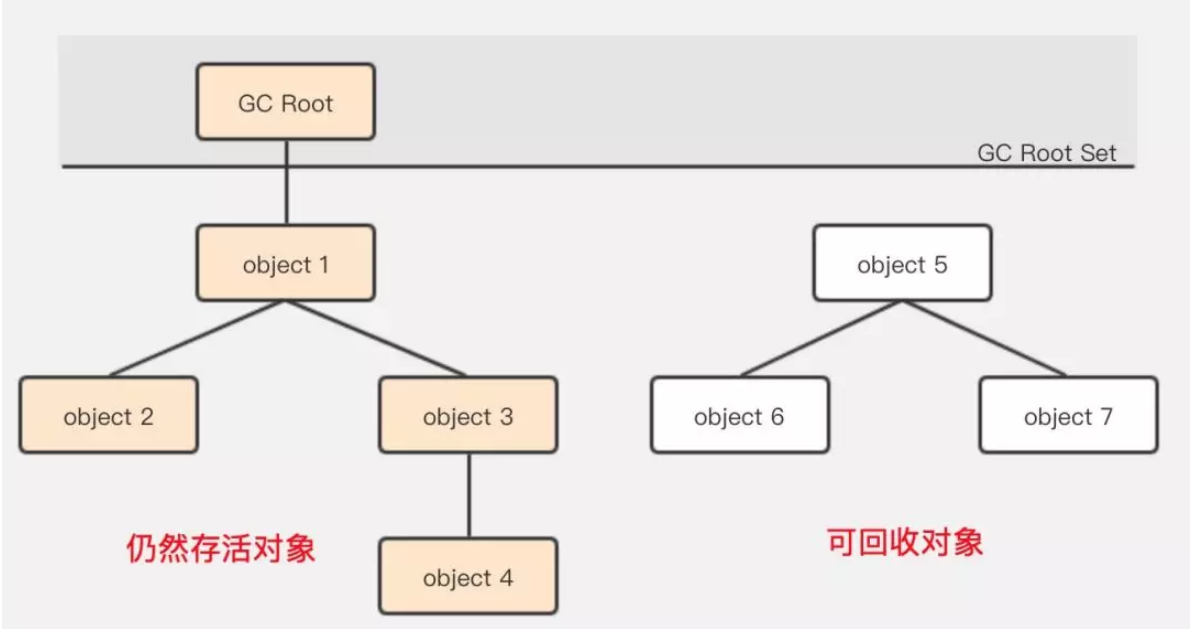
\2. 相互引用
\3. 置空各自的声明引用



我们可以看到，最后这2个对象已经不可能再被访问了，但由于他们相互引用着对方，导致它们的引用计数永远都不会为0，通过引用计数算法，也就永远无法通知GC收集器回收它们。

可达性分析算法

可达性分析算法（Reachability Analysis）的基本思路是，通过一些被称为引用链（GC Roots）的对象作为起点，从这些节点开始向下搜索，搜索走过的路径被称为（Reference Chain），当一个对象到 GC Roots 没有任何引用链相连时（即从 GC Roots 节点到该节点不可达），则证明该对象是不可用的。



通过可达性算法，成功解决了引用计数所无法解决的问题-“循环依赖”，只要你无法与 GC Root 建立直接或间接的连接，系统就会判定你为可回收对象。那这样就引申出了另一个问题，哪些属于 GC Root。

3. 简述 BIO, NIO, AIO 的区别

4. JVM 内存是如何对应到操作系统内存的?

5. 如何设计 Java 的异常体系?

6. 简述 Synchronized，Volatile，可重入锁的不同使用场景及优缺点

Synchronized	Volatile	可重入锁
1.只适用于区块或方法。	1.仅适用于变量。	1.它也只适用于变量。
2.采用同步修饰符实现了一种基于锁的并发算法，即受锁的限制。	2.在这里，易失性提供了实现非阻塞算法的能力，该算法具有更高的可伸缩性。	2.原子算法也给出了实现非阻塞算法的能力。
3.由于锁的获取和释放，性能与易失性和原子关键字相比相对较低。	3.与同步关键字相比，性能相对较高。	3.与易失关键字和同步关键字相比，性能相对较高。
4.由于锁的性质，它也不能幸免于诸如死锁和活锁之类的并发危险。	4.由于它的非锁定特性，它不受死锁和活锁等并发性危害的影响。	4.由于它的非锁定特性，它不受死锁和活锁等并发性危害的影响。

7. 如何设计一个线程池

8. Java 类的加载流程是怎样的？什么是双亲委派机制？

9. Spring MVC 的原理和流程

10. 简述常见的工厂模式以及单例模式的使用场景

11. 什么是内存泄漏，怎么确定内存泄漏？

12. 实现单例模式

13. 简述 Java 的反射机制及其应用场景

反射就是用来让开发者知道这个类中有什么成员，以及别的类中有什么成员。

网络协议

简述 TCP 三次握手以及四次挥手的流程。为什么需要三次握手以及四次挥手？ △ 26次

客户端和服务端通信前要进行连接，“3次握手”的作用就是`双方都能明确自己和对方的收、发能力是正常的`。

`第一次握手`：客户端发送网络包，服务端收到了。这样服务端就能得出结论：客户端的发送能力、服务端的接收能力是正常的。

`第二次握手`：服务端发包，客户端收到了。这样客户端就能得出结论：服务端的接收、发送能力，客户端的接收、发送能力是正常的。 从客户端的视角来看，我接到了服务端发送过来的响应数据包，说明服务端接收到了我在第一次握手时发送的网络包，并且成功发送了响应数据包，这就说明，服务端的接收、发送能力正常。而另一方面，我收到了服务端的响应数据包，说明我第一次发送的网络包成功到达服务端，这样，我自己的发送和接收能力也是正常的。

`第三次握手`：客户端发包，服务端收到了。这样服务端就能得出结论：客户端的接收、发送能力，服务端的发送、接收能力是正常的。 第一、二次握手后，服务端并不知道客户端的接收能力以及自己的发送能力是否正常。而在第三次握手时，服务端收到了客户端对第二次握手作的回应。从服务端的角度，我在第二次握手时的响应数据发送出去了，客户端接收到了。所以，我的发送能力是正常的。而客户端的接收能力也是正常的。

经历了上面的三次握手过程，客户端和服务端都确认了自己的接收、发送能力是正常的。之后就可以正常通信了。

每次都是接收到数据包的一方可以得到一些结论，发送的一方其实没有任何头绪。我虽然有发包的动作，但是我怎么知道我有没有发出去，而对方有没有接收到呢？

而从上面的过程可以看到，最少是需要三次握手过程的。两次达不到让双方都得出自己、对方的接收、发送能力都正常的结论。其实每次收到网络包的一方至少是可以得到：对方的发送、我方的接收是正常的。而每一步都是有关联的，下一次的“响应”是由于第一次的“请求”触发，因此每次握手其实是可以得到额外的结论的。比如第三次握手时，服务端收到数据包，表明看服务端只能得到客户端的发送能力、服务端的接收能力是正常的，但是结合第二次，说明服务端在第二次发送的响应包，客户端接收到了，并且作出了响应，从而得到额外的结论：客户端的接收、服务端的发送是正常的。

用表格总结一下：

视角	客收	客发	服收	服发
客视角	二	一 + 二	一 + 二	二
服视角	二 + 三	一	一	二 + 三

三次握手的过程的示意图如下：

TCP 的连接的拆除需要发送四个包，因此称为四次挥手(Four-way handshake)，也叫做改进的三次握手。客户端或服务器均可主动发起挥手动作，在 socket 编程中，任何一方执行 `close()` 操作即可产生挥手操作。

- 第一次挥手(FIN=1，seq=x)

  假设客户端想要关闭连接，客户端发送一个 FIN 标志位置为1的包，表示自己已经没有数据可以发送了，但是仍然可以接受数据。

  发送完毕后，客户端进入 `FIN_WAIT_1` 状态。

- 第二次挥手(ACK=1，ACKnum=x+1)

  服务器端确认客户端的 FIN 包，发送一个确认包，表明自己接受到了客户端关闭连接的请求，但还没有准备好关闭连接。

发送完毕后，服务器端进入 `CLOSE_WAIT` 状态，客户端接收到这个确认包之后，进入 `FIN_WAIT_2` 状态，等待服务器端关闭连接。

- 第三次挥手(FIN=1，seq=y)

  服务器端准备好关闭连接时，向客户端发送结束连接请求，FIN 置为1。

  发送完毕后，服务器端进入 `LAST_ACK` 状态，等待来自客户端的最后一个ACK。

- 第四次挥手(ACK=1，ACKnum=y+1)

  客户端接收到来自服务器端的关闭请求，发送一个确认包，并进入 `TIME_WAIT`状态，等待可能出现的要求重传的 ACK 包。

  服务器端接收到这个确认包之后，关闭连接，进入 `CLOSED` 状态。

  客户端等待了某个固定时间（两个最大段生命周期，2MSL，2 Maximum Segment Lifetime）之后，没有收到服务器端的 ACK ，认为服务器端已经正常关闭连接，于是自己也关闭连接，进入 `CLOSED` 状态。

TCP 怎么保证可靠传输?

1. 应用数据被分割成 TCP 认为最适合发送的数据块。
2. TCP 给发送的每一个包进行编号，接收方对数据包进行排序，把有序数据传送给应用层。
3. **校验和：** TCP 将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错，TCP 将丢弃这个报文段和不确认收到此报文段。
4. TCP 的接收端会丢弃重复的数据。
5. **流量控制：** TCP 连接的每一方都有固定大小的缓冲空间，TCP的接收端只允许发送端发送接收端缓冲区能接纳的数据。当接收方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。TCP 使用的流量控制协议是可变大小的滑动窗口协议。 （TCP 利用滑动窗口实现流量控制）
6. **拥塞控制：** 当网络拥塞时，减少数据的发送。
7. **ARQ协议：** 也是为了实现可靠传输的，它的基本原理就是每发完一个分组就停止发送，等待对方确认。在收到确认后再发下一个分组。
8. **超时重传：** 当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

简述 HTTP 1.0，1.1，2.0 的主要区别

HTTP1.0最早在网页中使用是在1996年，那个时候只是使用一些较为简单的网页上和网络请求上，而HTTP1.1则在1999年才开始广泛应用于现在的各大浏览器网络请求中，同时HTTP1.1也是当前使用最为广泛的HTTP协议。 主要区别主要体现在：

1. **缓存处理**，在HTTP1.0中主要使用header里的If-Modified-Since,Expires来做为缓存判断的标准，HTTP1.1则引入了更多的缓存控制策略例如Entity tag，If-Unmodified-Since, If-Match, If-None-Match等更多可供选择的缓存头来控制缓存策略。
2. **带宽优化及网络连接的使用**，HTTP1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP1.1则在请求头引入了range头域，它允许只请求资源的某个部分，即返回码是206（Partial Content），这样就方便了开发者自由的选择以便于充分利用带宽和连接。
3. **错误通知的管理**，在HTTP1.1中新增了24个错误状态响应码，如409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。
4. **Host头处理**，在HTTP1.0中认为每台服务器都绑定一个唯一的IP地址，因此，请求消息中的URL并没有传递主机名（hostname）。但随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个IP地址。HTTP1.1的请求消息和响应消息都应支持Host头域，且请求消息中如果没有Host头域会报告一个错误（400 Bad Request）。

5. **长连接**，HTTP 1.1支持长连接（PersistentConnection）和请求的流水线（Pipelining）处理，在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟，在HTTP1.1中默认开启Connection： keep-alive，一定程度上弥补了HTTP1.0每次请求都要创建连接的缺点。

TCP 与 UDP 在网络协议中的哪一层，他们之间有什么区别?

- TCP是面向连接的协议，而UDP是无连接的协议.
- Tcp的速度较慢，而udp的速度更快。
- Tcp使用诸如syn、syn-ack、ack等握手协议，而udp不使用握手协议。
- TCP进行错误检查并进行错误恢复，另一方面，UDP执行错误检查，但它丢弃错误的数据包。
- TCP有确认段，但是UDP没有任何确认段。
- TCP是重量级的，UDP是轻量级的

HTTP 与 HTTPS 有哪些区别?

- HTTP 明文传输，数据都是未加密的，安全性较差，HTTPS（SSL+HTTP） 数据传输过程是加密的，安全性较好。
- 使用 HTTPS 协议需要到 CA（Certificate Authority，数字证书认证机构） 申请证书，一般免费证书较少，因而需要一定费用。证书颁发机构如：Symantec、Comodo、GoDaddy 和 GlobalSign 等。
- HTTP 页面响应速度比 HTTPS 快，主要是因为 HTTP 使用 TCP 三次握手建立连接，客户端和服务器需要交换 3 个包，而 HTTPS除了 TCP 的三个包，还要加上 ssl 握手需要的 9 个包，所以一共是 12 个包。
- http 和 https 使用的是完全不同的连接方式，用的端口也不一样，前者是 80，后者是 443。
- HTTPS 其实就是建构在 SSL/TLS 之上的 HTTP 协议，所以，要比较 HTTPS 比 HTTP 要更耗费服务器资源。

如何防止传输内容被篡改?

从输入 URL 到展现页面的全过程

HTTP 是无状态的吗？需要保持状态的场景应该怎么做?

简述 HTTPS 的加密与认证过程

简述 RPC 的调用过程

什么是 ARP 协议？简述其使用场景

简述 TCP 滑动窗口以及重传机制

TCP 利用滑动窗口实现流量控制。流量控制是为了控制发送方发送速率，保证接收方来得及接收。 接收方发送的确认报文中的窗口字段可以用来控制发送方窗口大小，从而影响发送方的发送速率。将窗口字段设置为 0，则发送方不能发送数据。

TCP 如何实现数据有序性?

数据库

数据库的事务隔离级别有哪些？各有哪些优缺点？ 中等 参考1 参考2

SQL 标准定义了四种隔离级别，MySQL 全都支持。这四种隔离级别分别是：

1. 读未提交（READ UNCOMMITTED）
2. 读提交 （READ COMMITTED）
3. 可重复读 （REPEATABLE READ）
4. 串行化 （SERIALIZABLE）

从上往下，隔离强度逐渐增强，性能逐渐变差。采用哪种隔离级别要根据系统需求权衡决定，其中，**可重复读**是 MySQL 的默认级别。

事务隔离其实就是为了解决上面提到的脏读、不可重复读、幻读这几个问题，下面展示了 4 种隔离级别对这三个问题的解决程度。

隔离级别	脏读	不可重复读	幻读
读未提交	可能	可能	可能
读提交	不可能	可能	可能
可重复读	不可能	不可能	可能
串行化	不可能	不可能	不可能

只有串行化的隔离级别解决了全部这 3 个问题，其他的 3 个隔离级别都有缺陷。

MySQL 为什么使用 B+ 树来作索引，对比 B 树它的优点和缺点是什么？ 中等 参考1 参考2

 image-20210617205931330

Redis的缓存淘汰策略有哪些？ 简单 参考1 参考2

Redis `Expire` 命令设置 `key` 的过期时间（seconds）。 设置的时间过期后，key 会被自动删除。带有超时时间的 key 通常被称为易失的(*volatile*)。

超时时间只能使用删除 key 或者覆盖 key 的命令清除，包括 DEL, SET, GETSET 和所有的 `*STORE` 命令。 对于修改 key 中存储的值，而不是用新值替换旧值的命令，不会修改超时时间。例如，自增 key 中存储的值的 INCR , 向list中新增一个值 LPUSH, 或者修改 hash 域的值 HSET ， 这些都不会修改 key 的过期时间。

通过使用 PERSIST 命令把 key 改回持久的(persistent) key，这样 key 的过期时间也可以被清除。

key使用 RENAME 改名后，过期时间被转移到新 key 上。

已存在的旧 key 使用 RENAME 改名，那么新 key 会继承所有旧 key 的属性。例如，一个名为 Key*A 的 key 使用命令* `RENAME Key_B Key_A` *改名，新的 Key*A 会继承包括超时时间在内的所有 Key_B 的属性。

特别注意，使用负值调用 EXPIRE/PEXPIRE 或使用过去的时间调用 EXPIREAT/PEXPIREAT ，那么 key 会被删除 deleted 而不是过期。 (因为, 触发的key event 将是 `del`, 而不是 `expired`).

语法

redis `Expire` 命令基本语法如下:

```
redis 127.0.0.1:6379> EXPIRE key seconds
```

刷新过期时间

已经设置过期的key，可以调用 EXPIRE 重新设置。在这种情况下 key 的生存时间被更新为新值。因为这个特性出现了很多有用的应用，本文下面的 *Navigation session* pattern 部分就是一个例子。

Expire 在 Redis 2.1.3 之前版本的不同

在 **2.1.3** 之前的版本使用 Expire 修改已经设置过生存时间的 key，将会和删掉 key 有同样的效果。这是被已被修复的复制层的限制所需要的特性。这种情况下 EXPIRE 将会返回 0。

返回值

整数:

- `1` 设置超时成功。
- `0` `key` 不存在。

例子

redis>**SET mykey "Hello"**

```
"OK"
```

redis>**EXPIRE mykey 10**

```
(integer) 1
```

redis>**TTL mykey**

```
(integer) 10
```

redis>**SET mykey "Hello World"**

```
"OK"
```

redis>**TTL mykey**

```
(integer) -1
```

redis>

模式: 导航 session

假设你有个 web 服务并且你关注用户最近最新访问的 N 个页面，每个相邻新页面的访问时间在 60 秒内，概念上我们可把这一系列的页面访问作为一个用户的导航会话。 这里面包含了很多关于用户正在寻找什么样产品的有用信息，你可以据此给用户推荐产品。

在 Redis 中使用下面的策略，我们可以给这种模式建模：用户每访问一个页面，我们都执行下面的指令：

```
MULTI
RPUSH pagewviews.user:<userid> http://.....
EXPIRE pagewviews.user:<userid> 60
EXEC
```

如果用户访问的页面空闲时间超过 60 秒，那么这个 key 将会被删除，只有那些接下来小于 60 秒空闲的页面访问将会被保留。

这个模式可以修改为使用计数指令 INCR 替换列表的 RPUSH。

附录: Redis expires

带有过期时间的 Key

一般来说 Redis 中创建 key 的是不带生存时间，如果你不使用类似 DEL 命令明确删除它，这个key将会一直存在。

EXPIRE 命令族可以给指定的 key 关联过期时间 ， key 会额外多占用一些内存。一旦一个key设置了过期时间，那么指定的时间消耗完后，Redis 需要确保 key 被删除。

key 的过期时间可以被 EXPIRE 命令更新或 PERSIST 命令删掉 。

过期时间的精度

Redis 2.4 过期时间并不精准，一般在 0到 1 秒多。

从Redis 2.6起，过期时间精度提高到 0 到 1 毫秒多 。

过期和持久

key 的过期时间以绝对 Unix 时间戳的方式存储。这意味无论 Redis 是否运行，过期时间都会流逝。

服务器的时间必须稳定准确，这样过期时间才能更准确。如果在两个时间相差较多的机器之间移动 RDB 文件，那么可能会出现所有的 key 在加载的时候都过期了。

运行的 Redis 也会不停的检查服务器的时间，如果你设置一个带有 1000 秒过期时间的key，然后你把服务器的时间向前调了 2000 秒，那么这个 key 会立刻过期，不是等 1000 秒后过期。

Redis 如何淘汰过期 Key

Redis 的 key 有两种过期淘汰的方式：被动方式、主动方式。

被动过期：用户访问某个 key 的时候，key 被发现过期。

当然，被动方式过期对于那些永远也不会再次被访问的 key 并没有效果。不管怎么，这些 key 都应被过期淘汰，所以 Redis 周期性主动随机检查一部分被设置生存时间的 key，那些已经过期的 key 会被从 key 空间中删除。

Redis每秒执行10次下面的操作：

1. 从带有生存时间的 key 的集合中随机选 20 进行检查。
2. 删除所有过期的key。
3. 如20里面有超过25%的key过期，立刻继续执行步骤1。

这是一个狭义概率算法，我们假设我们选出来的样本 key 代表整个 key 空间，我们继续过期检查直到过期 key 的比例降到 25% 以下。

这意味着在任意时刻已经过期但还占用内存的 key 的数量，最多等于每秒最多写操作的四分之一。

为什么 Redis 在单线程下能如此快？ 中等 参考1

简述 Redis 持久化中 RDB 以及 AOF 方案的优缺点 困难 参考1 参考2

RDB优势

- RDB是一个非常紧凑的单文件点对点表示，您的Redis数据.RDB文件非常适合备份。例如，您可能希望在最近的24小时内每小时存档RDB文件，并在30天内每天保存RDB快照。这允许您在发生灾难时轻松地恢复不同版本的数据集。
- RDB非常适合灾难恢复，它是一个单独的压缩文件，可以传输到远程数据中心，也可以传输到AmazonS 3(可能是加密的)。
- RDB最大限度地提高了Redis的性能，因为Redis父进程需要做的惟一工作就是分叉一个将完成所有其余工作的子进程。父实例将永远不会执行磁盘I/O或类似的操作。
- 与AOF相比，RDB允许更快地使用大数据集重新启动。
- 在副本上，rdb支持重新启动和故障转移后的部分重新同步.

RDB缺点

- 如果需要在Redis停止工作(例如停电后)时尽量减少数据丢失的可能性，RDB是不好的。您可以配置不同的*保存点*其中生成了一个RDB(例如，在至少5分钟和100分钟之后对数据集进行写入，但是您可以有多个保存点)。但是，通常每五分钟或更长时间创建一次RDB快照，因此，如果Redis停止工作而不因任何原因而正确关闭，您应该准备好丢失最新的几分钟数据。
- 为了使用子进程在磁盘上持久化，RDB通常需要叉()。如果DataSet很大，FRACK()可能会很费时，如果数据集很大且CPU性能不太好，则可能导致Redis停止为客户端服务几毫秒，甚至一秒。AOF也需要FASK()，但是您可以调优您想要重写日志的频率，而不需要在持久性上进行任何权衡。

AOF优势

- 使用AOFRedis要持久得多：您可以有不同的fsync策略：根本不使用fsync，每秒使用fsync，在每次查询时都使用fsync。对于fsync的默认策略，每秒钟的写性能仍然很好(fsync使用后台线程执行，主线程将在没有fsync进程时努力执行写操作)。但你只能损失一秒的写作。
- AOF日志是一个仅附加的日志，因此如果出现断电，则不存在寻求，也不存在损坏问题。即使日志由于某种原因(磁盘已满或其他原因)以半写命令结尾，redis-Chec-aof工具也能够轻松地修复它。
- Redis能够在AOF变得太大时在后台自动重写它。重写是完全安全的，因为当Redis继续附加到旧文件时，会产生一个全新的文件，其中包含创建当前数据集所需的最小操作集，一旦第二个文件准备就绪，Redis就会切换这两个文件并开始追加到新的数据集。
- AOF以易于理解和解析的格式一个接一个地包含所有操作的日志。您甚至可以轻松地导出AOF文件。例如，即使您意外地使用FLUSHALL命令，只要同时不执行日志重写，您仍然可以通过停止服务器、删除最新命令和重新启动Redis来保存数据集。

AOF缺点

- AOF文件通常大于相同数据集的等效RDB文件。
- AOF可能比RDB慢，具体取决于fsync策略。通常，fsync设置为*每一秒*性能仍然很高，在禁用fsync的情况下，即使在高负载下，它也应该与RDB一样快。尽管如此，RDB仍然能够提供更多关于最大延迟的保证，即使在巨大的写负载情况下也是如此。
- 在过去，我们在特定命令中遇到了罕见的bug(例如，有一个涉及阻塞命令的问题，如BRPOPLPUSH)导致产生的AOF在重新加载时不能完全再现相同的数据集。这些bug是罕见的，我们在测试套件中进行了测试，自动创建随机复杂的数据集，并重新加载它们来检查一切都很好。然而，使用RDB持久性，这些bug几乎是不可能的。为了更清楚地说明这一点：Redis AOF通过增量更新现有状态来工作，就像MySQL或MongoDB所做的那样，而RDB快照一次又一次地创建一切，这在概念上更健壮。但是-1)应该注意的是，每次由Redis重写AOF时，都会从数据集中包含的实际数据开始从头开始重新创建AOF，与始终附加AOF文件(或者重写读取旧AOF而不是读取内存中的数据)相比，对bug的抵抗力更强。2)我们从来没有从用户那里得到过一份关于AOF在现实世界中被发现的腐败的报告。

什么是数据库事务，MySQL 为什么会使用 InnoDB 作为默认选项 中等 参考1 参考2

事务其实就是**并发控制的基本单位**；相信我们都知道，事务是一个序列操作，其中的操作要么都执行，要么都不执行，它是一个不可分割的工作单位；

`InnoDB`表格有以下好处：

- 如果服务器因硬件或软件问题而意外退出，无论当时数据库中发生了什么，在重新启动数据库后不需要做任何特殊的事情。`InnoDB`崩溃恢复会自动完成在崩溃之前提交的更改，并撤消正在进行但未提交的更改，从而使您可以从中断的位置重新启动并继续进行更改。
- 这个`InnoDB`存储引擎维护自己的缓冲池，在访问数据时缓存主内存中的表和索引数据。经常使用的数据直接从内存中处理。此缓存适用于多种类型的信息，并加快处理速度。在专用数据库服务器上，多达80%的物理内存通常分配给缓冲池。
- 如果将相关数据拆分为不同的表，则可以设置增强引用完整性的外键。
- 如果数据在磁盘或内存中损坏，则检查和机制会在您使用伪数据之前通知您。这个[`innodb_checksum_algorithm`]变量定义了`InnoDB`.
- 当为每个表设计具有适当主键列的数据库时，涉及这些列的操作将自动优化。中引用主键列的速度非常快。`WHERE`条款，`ORDER BY`条款，`GROUP BY`子句和联接操作。
- 插入、更新和删除是通过一种称为更改缓冲的自动机制进行优化的。`InnoDB`它不仅允许对同一个表进行并发读写访问，还缓存更改的数据以简化磁盘I/O
- 性能优势并不局限于具有长期运行查询的大型表。当从表中一次又一次地访问相同的行时，AdaptiveHashIndex将接管这些查询，使这些查找速度更快，就好像它们是从哈希表中出来的一样。
- 可以压缩表和相关索引。
- 你可以加密你的数据。
- 您可以创建和删除索引，并执行其他DDL操作，但对性能和可用性的影响要小得多。
- 截断每个表的文件空间非常快，并且可以释放磁盘空间供操作系统重用而不仅仅是`InnoDB`
- 表数据的存储布局对于`BLOB`和较长的文本字段，`DYNAMIC`行格式
- 您可以通过查询监视存储引擎的内部工作。`INFORMATION_SCHEMA`桌子。
- 您可以通过查询PerformanceSchema表来监视存储引擎的性能细节。
- 你可以混在一起`InnoDB`表中包含来自其他MySQL存储引擎的表，即使在同一语句中也是如此。例如，可以使用联接操作将来自`InnoDB`和[`MEMORY`一个查询中的表。
- `InnoDB`是为处理大数据量时的CPU效率和最大性能而设计的。
- `InnoDB`表可以处理大量数据，即使在文件大小限制为2GB的操作系统中也是如此。

为`InnoDB`-特定的调优技术，您可以应用于MySQL服务器和应用程序代码，

数据库的读写分离的作用是什么？如何实现？ 中等

聚簇索引和非聚簇索引有什么区别？ 简单 参考1 参考2

- 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

数据库索引的实现原理是什么？ 简单

如何解决缓存与数据库不一致的问题?

简述数据库中的 ACID 分别是什么? 简单

数据库索引的叶子结点为什么是有序链表?

操作系统

进程和线程之间有什么区别? 简单 参考1 参考2

参数	加工过程	螺纹
定义	进程意味着程序正在执行。	线程是进程的一个片段。
轻量级	这个过程不是轻量级的。	线程很轻。
终止时间	进程需要更多的时间来终止。	线程终止所需的时间较少。
创作时间	创造需要更多的时间。	创造所需的时间较少。
通讯	与线程相比，进程之间的通信需要更多的时间。	与进程相比，线程之间的通信需要更少的时间。
上下文切换时间	它需要更多的时间来切换上下文。	上下文切换所需的时间更短。
资源	进程消耗更多的资源。	线程消耗的资源较少。
OS治疗	操作系统分别处理不同的过程。	所有级别的对等线程都被操作系统视为一个单一任务。
记忆	这一过程大多是孤立的。	线程共享内存。
共享	它不共享数据。	线程彼此共享数据。

进程间有哪些通信方式? 困难 参考1 参考2管道

管道广泛用于两个相关过程之间的通信。这是一个半双工方法，所以第一个进程与第二个进程通信.然而，为了实现全双工，还需要另一根管道.

消息传递:

它是进程通信和同步的一种机制。使用消息传递，流程彼此通信，而不诉诸共享变量。

IPC机制提供两个操作:

- 发送(消息)-消息大小固定或可变

- 收到(电文)

消息队列：

消息队列是存储在内核中的消息的链接列表。它由消息队列标识符标识。该方法提供具有全双工容量的单进程或多进程之间的通信.

直接通讯：

在这种类型的进程间通信过程中，应明确彼此的名称.在该方法中，在一对通信进程之间建立链接，而在每对通信进程之间，只存在一个链接。

间接通信：

间接通信建立，就像只有当进程共享一个公共邮箱时，每对进程共享多个通信链接。一个链接可以与许多进程通信。该链接可以是双向的，也可以是单向的。

共享内存：

共享内存是使用共享内存在所有进程之间建立的两个或多个进程之间共享的内存。这种类型的内存需要通过同步跨所有进程的访问来相互保护。

FIFO：

两个不相关的过程之间的交流。这是一种全双工方法，这意味着第一个进程可以与第二个进程通信，也可能发生相反的情况。

简述几个常用的 Linux 命令以及他们的功能 简单

简述 Linux 进程调度的算法 困难 参考1

简述 select, poll, epoll 的使用场景以及区别，epoll 中水平触发以及边缘触发有什么不同？ 困难 参考1 参考2

区别：

- `select`：调用开销大（需要复制集合）；集合大小有限制；需要遍历整个集合找到就绪的描述符
- `poll`：poll 采用链表的方式存储文件描述符，没有最大存储数量的限制，其他方面和 select 没有区别
- `epoll`：调用开销小（不需要复制）；集合大小无限制；采用回调机制，不需要遍历整个集合

不同

水平触发（LT，Level Trigger）：当文件描述符就绪时，会触发通知，如果用户程序没有一次性把数据读/写完，下次还会发出可读/可写信号进行通知。

边缘触发（ET，Edge Trigger）：仅当描述符从未就绪变为就绪时，通知一次，之后不会再通知。

系统设计

项目上有什么技术难点?

了解高并发的解决方案吗?例如动静分离，缓存，负载均衡

有哪些实现服务发现的方法?

如何阅读大型项目的源码?

前端

readyState 的不同返回值有什么区别?

简述什么是 XSS 攻击以及 CSRF 攻击? 中等 参考1 参考2

简述 React 的生命周期

promise 有哪些状态?简述 promise.all 的实现原理 中等

HTTP 中 GET 和 POST 区别 简单 参考1

	GET	POST
后退按钮/刷新	无害	数据会被重新提交（浏览器应该告知用户数据会被重新提交）。
书签	可收藏为书签	不可收藏为书签
缓存	能被缓存	不能缓存
编码类型	application/x-www-form-urlencoded	application/x-www-form-urlencoded or multipart/form-data。为二进制数据使用多重编码。
历史	参数保留在浏览器历史中。	参数不会保存在浏览器历史中。
对数据长度的限制	是的。当发送数据时，GET 方法向 URL 添加数据；URL 的长度是受限制的（URL 的最大长度是 2048 个字符）。	无限制。
对数据类型的限制	只允许 ASCII 字符。	没有限制。也允许二进制数据。
安全性	与 POST 相比，GET 的安全性较差，因为所发送的数据是 URL 的一部分。 在发送密码或其他敏感信息时绝不要使用 GET ！	POST 比 GET 更安全，因为参数不会被保存在浏览器历史或 web 服务器日志中。
可见性	数据在 URL 中对所有人都是可见的。	数据不会显示在 URL 中。

promise 有哪些状态？简述 promise.all 的实现原理 中等

CSS 实现三列布局 简单

什么是可继承元素和不可继承元素?

什么是替换元素与非替换元素

简述 ES6 的新特性 简单

正则表达式 /w 是什么意思?

C++

简述 C++ 中智能指针的特点，简述 new 与 malloc 的区别 中等 参考1 参考2

指针和引用的区别是什么?

C++ 11 有什么新特性 简单 参考1 参考2

C++11标准库也用新的算法、新的容器类、原子操作、类型特征、正则表达式、新的智能指针进行了改进，`async()`设施，当然还有多线程库。

线程库

毫无疑问，从程序员的角度来看，对C++11最重要的补充是并发性。C++11有一个表示执行线程的线程类，承诺与未来对象，这些对象用于并发环境中的同步，异步()函数模板，用于启动并发任务，以及螺纹局部声明线程唯一数据的存储类型。要快速浏览C++11线程库，请阅读Anthony Williams的C++0x中的简单多线程.

新的智能指针类

C++98只定义了一个智能指针类，`auto_ptr`，现在已经不受欢迎了。C++11包括新的智能指针类：共享PTR最近增加的唯一PTR。这两个组件都与其他标准库组件兼容，因此您可以安全地将这些智能指针存储在标准容器中，并使用标准算法对它们进行操作。

新的C++算法

C++11标准库定义了模仿集合论操作的新算法`all_of()`，`any_of()`和`none_of()`。下面的清单应用谓词`ispositive()`到射程`[first, first+n)`和用途`all_of()`，`any_of()`和`none_of()`若要检查范围的属性，请执行以下操作：

```
#include <algorithm>//C++11 code//are all of the elements positive?all_of(first, first+n, ispositive()); //false//is there at least one positive element? any_of(first, first+n, ispositive());//true// are none of the elements positive? none_of(first, first+n, ispositive()); //false
```

一个新的类别 `copy_n` 算法也是可用的。使用 `copy_n()`，将一个由5个元素组成的数组复制到另一个数组是轻而易举的：

```
#includeint source[5]={0,12,34,50,80};int target[5];//copy 5 elements from source to targetcopy_n(source,5,target);
```

算法 `iota()` 创建一系列顺序递增的值，就像将初始值赋给 `*first`，然后使用前缀++递增该值。在下面的清单中，`iota()` 将连续值{10、11、12、13、14}分配给数组 `arr`，以及字符数组的{'a'、'b'、'c'}。`c`.

```
include <numeric>int a[5]={0};char c[3]={0};iota(a, a+5, 10); //changes a to {10,11,12,13,14}iota(c, c+3, 'a'); //{'a','b','c'}
```

C++ 中虚函数与纯虚函数的区别 简单

STL 中 vector 与 list 具体是怎么实现的？常见操作的时间复杂度是多少？ 简单 参考1 参考2

C++ 中智能指针和指针的区别是什么？ 简单 参考1 参考2

简述 C++ 中内存对齐的使用场景 中等

简述 vector 的实现原理 简单

非技术

未来的职业规划是什么？

能接受多大强度的加班？

你用过拼多多吗？说说对对拼多多的看法，使用上有哪些优缺点

对下一家入职的公司主要看重哪些方面？

自己有哪些不足之处，准备如何提升？

快手

算法

141. 环形链表 简单

方法一：哈希表
思路及算法

最容易想到的方法是遍历所有节点，每次遍历到一个节点时，判断该节点此前是否被访问过。

具体地，我们可以使用哈希表来存储所有已经访问过的节点。每次我们到达一个节点，如果该节点已经存在于哈希表中，则说明该链表是环形链表，否则就将该节点加入哈希表中。重复这一过程，直到我们遍历完整个链表即可。

代码

```
public class Solution {
    public boolean hasCycle(ListNode head) {
        Set<ListNode> seen = new HashSet<ListNode>();
        while (head != null) {
            if (!seen.add(head)) {
                return true;
            }
            head = head.next;
        }
        return false;
    }
}
```

复杂度分析

- 时间复杂度：O(N)，其中 N 是链表中的节点数。最坏情况下我们需要遍历每个节点一次。
- 空间复杂度：O(N)，其中 N 是链表中的节点数。主要为哈希表的开销，最坏情况下我们需要将每个节点插入到哈希表中一次。

方法二：快慢指针
思路及算法

本方法需要读者对「Floyd 判圈算法」（又称龟兔赛跑算法）有所了解。

假想「乌龟」和「兔子」在链表上移动，「兔子」跑得快，「乌龟」跑得慢。当「乌龟」和「兔子」从链表上的同一个节点开始移动时，如果该链表中没有环，那么「兔子」将一直处于「乌龟」的前方；如果该链表中有环，那么「兔子」会先于「乌龟」进入环，并且一直在环内移动。等到「乌龟」进入环时，由于「兔子」的速度快，它一定会在某个时刻与乌龟相遇，即套了「乌龟」若干圈。

我们可以根据上述思路来解决本题。具体地，我们定义两个指针，一快一满。慢指针每次只移动一步，而快指针每次移动两步。初始时，慢指针在位置 head，而快指针在位置 head.next。这样一来，如果在移动的过程中，快指针反过来追上慢指针，就说明该链表为环形链表。否则快指针将到达链表尾部，该链表不为环形链表。

细节

为什么我们要规定初始时慢指针在位置 head，快指针在位置 head.next，而不是两个指针都在位置 head（即与「乌龟」和「兔子」中的叙述相同）？

观察下面的代码，我们使用的是 while 循环，循环条件先于循环体。由于循环条件一定是判断快慢指针是否重合，如果我们将两个指针初始都置于 head，那么 while 循环就不会执行。因此，我们可以假想一个在 head 之前的虚拟节点，慢指针从虚拟节点移动一步到达 head，快指针从虚拟节点移动两步到达 head.next，这样我们就可以使用 while 循环了。

当然，我们也可以使用 do-while 循环。此时，我们就可以把快慢指针的初始值都置为 head。

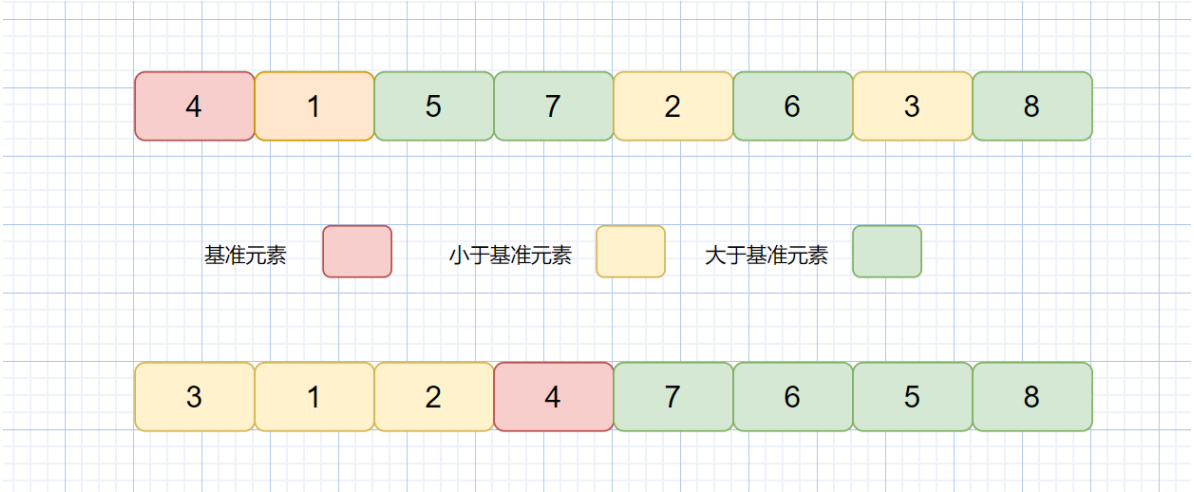
代码

```
public class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null || head.next == null) {
            return false;
        }
        ListNode slow = head;
        ListNode fast = head.next;
        while (slow != fast) {
            if (fast == null || fast.next == null) {
                return false;
            }
            slow = slow.next;
            fast = fast.next.next;
        }
        return true;
    }
}
```

复杂度分析

- 时间复杂度：O(N)，其中 N 是链表中的节点数。
    - 当链表中不存在环时，快指针将先于慢指针到达链表尾部，链表中每个节点至多被访问两次。
    - 当链表中存在环时，每一轮移动后，快慢指针的距离将减小一。而初始距离为环的长度，因此至多移动 N 轮。
- 空间复杂度：O(1)。我们只使用了两个指针的额外空间。

使用递归及非递归两种方式实现快速排序 中等 参考1 参考2

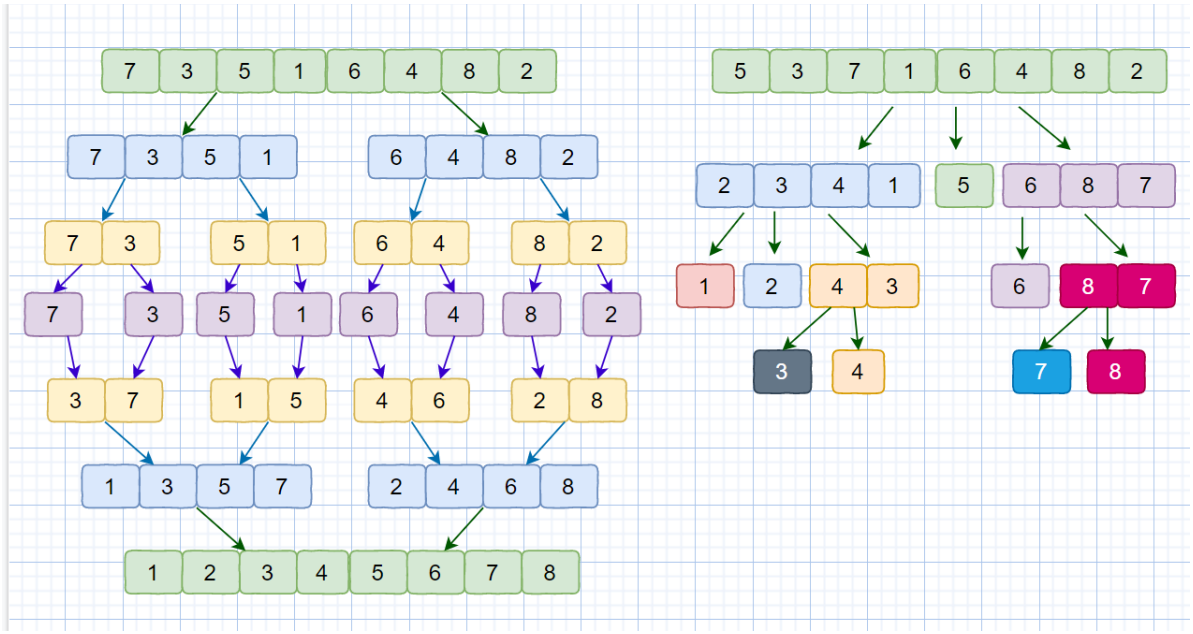


上图则为一次快排示意图，下面我们再利用递归，分别对左半边区间也就是 [3,1,2] 和右半区间 [7,6,5,8] 执行上诉过程，直至区间缩小为 1 也就是第三条，则此时所有的数据都有序。

简单来说就是我们利用基准数通过一趟排序将待排记录分割成独立的两部分，其中一部分记录的关键字均比基准数小，另一部分记录的关键字均比基准数大，然后分别对这两部分记录继续进行排序，进而达到有序。

我们现在应该了解了快速排序的思想，那么大家还记不记得我们之前说过的归并排序，他们两个用到的都是分治思想，那他们两个有什么不同呢？见下图

注：快速排序我们以序列的第一个元素作为基准数



虽然归并排序和快速排序都用到了分治思想，但是归并排序是自下而上的，先处理子问题，然后再合并，将小集合合成大集合，最后实现排序。而快速排序是由上到下的，先分区，然后再处理子问题。

归并排序虽然是稳定的、时间复杂度为 O(nlogn) 的排序算法，但是它是非原地排序算法。我们前面讲过，归并之所以是非原地排序算法，主要原因是合并函数无法在原地执行。快速排序通过设计巧妙的原地分区函数，可以实现原地排序，解决了归并排序占用太多内存的问题

我们根据思想可知，排序算法的核心就是如何利用基准数将记录分区，这里我们主要介绍两种容易理解的方法，一种是挖坑填数，另一种是利用双指针思想进行元素交换。

下面我们先来介绍下挖坑填数的分区方法

基本思想是我们首先以序列的第一个元素为基准数，然后将该位置挖坑，下面判断 nums[hight] 是否大于基准数，如果大于则左移 hight 指针，直至找到一个小于基准数的元素，将其填入之前的坑中，则 hight 位置会出现一个新的坑，此时移动 low 指针，找到大于基准数的元素，填入新的坑中。不断迭代直至完成分区。

代码

```
class Solution {
    public int[] sortArray(int[] nums) {

        quickSort(nums,0,nums.length-1);
        return nums;

    }
    public void quickSort (int[] nums, int low, int high) {

        if (low < high) {
            int index = partition(nums,low,high);
            quickSort(nums,low,index-1);
            quickSort(nums,index+1,high);
        }

    }
    public int partition (int[] nums, int low, int high) {

            int pivot = nums[low];
            while (low < high) {
                //移动high指针
                while (low < high && nums[high] >= pivot) {
                    hight--;
                }
                //填坑
                if (low < high) nums[low] = nums[high];
                while (low < high && nums[low] <= pivot) {
                    low++;
                }
                //填坑
                if (low < high) nums[high] = nums[low];
            }
            //基准数放到合适的位置
            nums[low] = pivot;
            return low;
    }
}
```

102. 二叉树的层序遍历 中等

方法一：广度优先搜索
思路和算法

我们可以用广度优先搜索解决这个问题。

我们可以想到最朴素的方法是用一个二元组 (node, level) 来表示状态，它表示某个节点和它所在的层数，每个新进队列的节点的 level 值都是父亲节点的 level 值加一。最后根据每个点的 level 对点进行分类，分类的时候我们可以利用哈希表，维护一个以 level 为键，对应节点值组成的数组为值，广度优先搜索结束以后按键 level 从小到大取出所有值，组成答案返回即可。

考虑如何优化空间开销：如何不用哈希映射，并且只用一个变量 node 表示状态，实现这个功能呢？

我们可以用一种巧妙的方法修改广度优先搜索：

- 首先根元素入队
- 当队列不为空的时候
    - 求当前队列的长度 si
    - 依次从队列中取 si个元素进行拓展，然后进入下一次迭代

它和普通广度优先搜索的区别在于，普通广度优先搜索每次只取一个元素拓展，而这里每次取s i个元素。在上述过程中的第 ii 次迭代就得到了二叉树的第 ii 层的si个元素。

为什么这么做是对的呢？我们观察这个算法，可以归纳出这样的循环不变式：第 ii 次迭代前，队列中的所有元素就是第 ii 层的所有元素，并且按照从左向右的顺序排列。证明它的三条性质（你也可以把它理解成数学归纳法）：

- 初始化：i = 1的时候，队列里面只有 root，是唯一的层数为 1 的元素，因为只有一个元素，所以也显然满足「从左向右排列」；
- 保持：如果 i = k 时性质成立，即第 kk 轮中出队sk的元素是第 k 层的所有元素，并且顺序从左到右。因为对树进行广度优先搜索的时候由低 k 层的点拓展出的点一定也只能是 k+1 层的点，并且 k+1 层的点只能由第 k 层的点拓展到，所以由这 s k个点能拓展到下一层所有的 sk+1个点。又因为队列的先进先出（FIFO）特性，既然第 kk 层的点的出队顺序是从左向右，那么第 k+1 层也一定是从左向右。至此，我们已经可以通过数学归纳法证明循环不变式的正确性。
- 终止：因为该循环不变式是正确的，所以按照这个方法迭代之后每次迭代得到的也就是当前层的层次遍历结果。至此，我们证明了算法是正确的。

代码

```
   class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector <vector <int>> ret;
        if (!root) {
            return ret;
        }

   queue <TreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        int currentLevelSize = q.size();
        ret.push_back(vector <int> ());
        for (int i = 1; i <= currentLevelSize; ++i) {
            auto node = q.front(); q.pop();
            ret.back().push_back(node->val);
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
    }

    return ret;
```

```
    }
};
```

复杂度分析

记树上所有节点的个数为 n。

时间复杂度：每个点进队出队各一次，故渐进时间复杂度为 O(n)。
空间复杂度：队列中元素的个数不超过 n 个，故渐进空间复杂度为O(n)。

合并两个有序数组 (Leetcode)

3. 无重复字符的最长子串 中等

方法一：滑动窗口
思路和算法

我们先用一个例子考虑如何在较优的时间复杂度内通过本题。

我们不妨以示例一中的字符串 abcabcbb}abcabcbb 为例，找出**从每一个字符开始的，不包含重复字符的最长子串**，那么其中最长的那个字符串即为答案。对于示例一中的字符串，我们列举出这些结果，其中括号中表示选中的字符以及最长的字符串：

- 以(a)bcabcbb 开始的最长字符串为 (abc)abcbb；
- 以 a(b)cabcbb 开始的最长字符串为 a(bca)bcbb；
- 以 ab(c)abcbb 开始的最长字符串为 ab(cab)cbb；
- 以 abc(a)bcbb 开始的最长字符串为 abc(abc)bb；
- 以 abca(b)cbb 开始的最长字符串为 abca(bc)bb；
- 以 abcab(c)bb 开始的最长字符串为 abcab(cb)b；
- 以 abcabc(b)b 开始的最长字符串为 abcabc(b)b；
- 以 abcabcb(b) 开始的最长字符串为 abcabcb(b)。

发现了什么？如果我们依次递增地枚举子串的起始位置，那么子串的结束位置也是递增的！这里的原因在于，假设我们选择字符串中的第 k 个字符作为起始位置，并且得到了不包含重复字符的最长子串的结束位置为 r k 。那么当我们选择第 k+1 个字符作为起始位置时，首先从 k+1 到 r k的字符显然是不重复的，并且由于少了原本的第 kk 个字符，我们可以尝试继续增大 rk，直到右侧出现了重复字符为止。

这样一来，我们就可以使用「滑动窗口」来解决这个问题了：

- 我们使用两个指针表示字符串中的某个子串（或窗口）的左右边界，其中左指针代表着上文中「枚举子串的起始位置」，而右指针即为上文中的 r k；
- 在每一步的操作中，我们会将左指针向右移动一格，表示 我们开始枚举下一个字符作为起始位置，然后我们可以不断地向右移动右指针，但需要保证这两个指针对应的子串中没有重复的字符。在移动结束后，这个子串就对应着 以左指针开始的，不包含重复字符的最长子串。我们记录下这个子串的长度；
- 在枚举结束后，我们找到的最长的子串的长度即为答案。

判断重复字符

在上面的流程中，我们还需要使用一种数据结构来判断 是否有重复的字符，常用的数据结构为哈希集合（即 C++ 中的 std::unordered_set，Java 中的 HashSet，Python 中的 set, JavaScript 中的 Set）。在左指针向右移动的时候，我们从哈希集合中移除一个字符，在右指针向右移动的时候，我们往哈希集合中添加一个字符。

至此，我们就完美解决了本题。

代码

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        // 哈希集合，记录每个字符是否出现过
        unordered_set<char> occ;
        int n = s.size();
        // 右指针，初始值为 -1，相当于我们在字符串的左边界的左侧，还没有开始移动
        int rk = -1, ans = 0;
        // 枚举左指针的位置，初始值隐性地表示为 -1
        for (int i = 0; i < n; ++i) {
            if (i != 0) {
                // 左指针向右移动一格，移除一个字符
                occ.erase(s[i - 1]);
            }
            while (rk + 1 < n && !occ.count(s[rk + 1])) {
                // 不断地移动右指针
                occ.insert(s[rk + 1]);
                ++rk;
            }
            // 第 i 到 rk 个字符是一个极长的无重复字符子串
            ans = max(ans, rk - i + 1);
        }
        return ans;
    }
};
```

53. 最大子序和 简单

方法一：动态规划
思路和算法

假设 nums 数组的长度是 n，下标从 0 到 n−1。

我们用 f(i) 代表以第 i 个数结尾的「连续子数组的最大和」，那么很显然我们要求的答案就是：

 image-20210617220618941

因此我们只需要求出每个位置的 f(i)，然后返回 f 数组中的最大值即可。那么我们如何求 f(i) 呢？我们可以考虑nums[i] 单独成为一段还是加入f(i−1) 对应的那一段，这取决于 nums[i] 和 f(i−1)+nums[i] 的大小，我们希望获得一个比较大的，于是可以写出这样的动态规划转移方程：

 image-20210617220802604

不难给出一个时间复杂度 O(n)、空间复杂度O(n) 的实现，即用一个 f 数组来保存 f(i) 的值，用一个循环求出所有 f(i)。考虑到 f(i) 只和f(i−1) 相关，于是我们可以只用一个变量pre 来维护对于当前 f(i) 的 f(i−1) 的值是多少，从而让空间复杂度降低到 O(1)，这有点类似「滚动数组」的思想。

代码

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int pre = 0, maxAns = nums[0];
        for (const auto &x: nums) {
            pre = max(pre + x, x);
            maxAns = max(maxAns, pre);
        }
        return maxAns;
    }
};
```

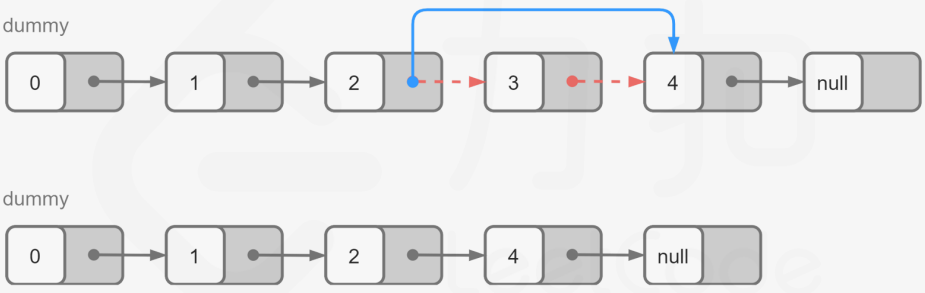
[19. 删除链表的倒数第 N 个结点](#) 中等

方法一：计算链表长度
思路与算法

一种容易想到的方法是，我们首先从头节点开始对链表进行一次遍历，得到链表的长度 L。随后我们再从头节点开始对链表进行一次遍历，当遍历到第L−n+1 个节点时，它就是我们需要删除的节点。

> 为了与题目中的 n 保持一致，节点的编号从 1 开始，头节点为编号 1 的节点。

为了方便删除操作，我们可以从哑节点开始遍历 L−n+1 个节点。当遍历到第 L−n+1 个节点时，它的下一个节点就是我们需要删除的节点，这样我们只需要修改一次指针，就能完成删除操作。



代码

```
class Solution {
public:
    int getLength(ListNode* head) {
        int length = 0;
        while (head) {
            ++length;
            head = head->next;
        }
        return length;
    }

    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode* dummy = new ListNode(0, head);
        int length = getLength(head);
```

```
        ListNode* cur = dummy;
        for (int i = 1; i < length - n + 1; ++i) {
            cur = cur->next;
        }
        cur->next = cur->next->next;
        ListNode* ans = dummy->next;
        delete dummy;
        return ans;
    }
};
```

方法二：栈
思路与算法

我们也可以在遍历链表的同时将所有节点依次入栈。根据栈「先进后出」的原则，我们弹出栈的第 nn 个节点就是需要删除的节点，并且目前栈顶的节点就是待删除节点的前驱节点。这样一来，删除操作就变得十分方便了。

代码

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode* dummy = new ListNode(0, head);
        stack<ListNode*> stk;
        ListNode* cur = dummy;
        while (cur) {
            stk.push(cur);
            cur = cur->next;
        }
        for (int i = 0; i < n; ++i) {
            stk.pop();
        }
        ListNode* prev = stk.top();
        prev->next = prev->next->next;
        ListNode* ans = dummy->next;
        delete dummy;
        return ans;
    }
};
```

对角线遍历 (Leetcode)

215. 数组中的第K个最大元素 中等

方法一：基于堆排序的选择方法
思路和算法

我们也可以使用堆排序来解决这个问题——建立一个大根堆，做 k−1 次删除操作后堆顶元素就是我们要找的答案。在很多语言中，都有优先队列或者堆的的容器可以直接使用，但是在面试中，面试官更倾向于让更面试者自己实现一个堆。所以建议读者掌握这里大根堆的实现方法，在这道题中尤其要搞懂「建堆」、「调整」和「删除」的过程。

友情提醒：「堆排」在很多大公司的面试中都很常见，不了解的同学建议参考《算法导论》或者大家的数据结构教材，一定要学会这个知识点哦！^_^

代码

```
class Solution {
public:
    void maxHeapify(vector<int>& a, int i, int heapSize) {
        int l = i * 2 + 1, r = i * 2 + 2, largest = i;
        if (l < heapSize && a[l] > a[largest]) {
            largest = l;
        }
        if (r < heapSize && a[r] > a[largest]) {
            largest = r;
        }
        if (largest != i) {
            swap(a[i], a[largest]);
            maxHeapify(a, largest, heapSize);
        }
    }

    void buildMaxHeap(vector<int>& a, int heapSize) {
        for (int i = heapSize / 2; i >= 0; --i) {
            maxHeapify(a, i, heapSize);
        }
    }

    int findKthLargest(vector<int>& nums, int k) {
        int heapSize = nums.size();
        buildMaxHeap(nums, heapSize);
        for (int i = nums.size() - 1; i >= nums.size() - k + 1; --i) {
            swap(nums[0], nums[i]);
            --heapSize;
            maxHeapify(nums, 0, heapSize);
        }
        return nums[0];
    }
};
```

如何从一个数组输出随机数组（洗牌算法）

206. 反转链表 简单

方法一：迭代
假设链表为 1→2→3→∅，我们想要把它改成 ∅←1←2←3。

在遍历链表时，将当前节点的 next 指针改为指向前一个节点。由于节点没有引用其前一个节点，因此必须事先存储其前一个节点。在更改引用之前，还需要存储后一个节点。最后返回新的头引用。

代码

```
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode curr = head;
        while (curr != null) {
            ListNode next = curr.next;
            curr.next = prev;
            prev = curr;
            curr = next;
        }
        return prev;
    }
}
```

简述布隆过滤器原理及其使用场景

合并两个有序数组 (Leetcode)

多数元素 (Leetcode)

用队列实现栈 (Leetcode)

常用的限流算法有哪些？简述令牌桶算法原理 中等 参考1 参考2

令牌桶、**固定窗**、**滑动测井**、**滑动窗**

[漏桶](与 [令牌桶]是一种算法，它提供了一种通过队列限制速率的简单、直观的方法，您可以把它看作一个承载请求的桶。注册请求时，系统会将其附加到队列的末尾。队列中第一项的处理是在一定间隔内进行的，或者是先入先出(FIFO)。如果队列已满，则会丢弃其他请求(或泄漏请求)。

有效的括号 (Leetcode)

旋转图像 (Leetcode)

69. x 的平方根

方法：二分查找
由于 x 平方根的整数部分 ans 是满足 k^2 ≤x 的最大 k 值，因此我们可以对 k 进行二分查找，从而得到答案。

二分查找的下界为 0，上界可以粗略地设定为 x。在二分查找的每一步中，我们只需要比较中间元素 mid 的平方与 x 的大小关系，并通过比较的结果调整上下界的范围。由于我们所有的运算都是整数运算，不会存在误差，因此在得到最终的答案 ans 后，也就不需要再去尝试 ans+1 了。

```
class Solution {
public:
    int mySqrt(int x) {
        int l = 0, r = x, ans = -1;
        while (l <= r) {
            int mid = l + (r - l) / 2;
            if ((long long)mid * mid <= x) {
                ans = mid;
                l = mid + 1;
            } else {
                r = mid - 1;
            }
        }
        return ans;
    }
};
```

剑指 Offer 10- II. 青蛙跳台阶问题

最小覆盖子串 (Leetcode)

最大栈 (Leetcode)

二叉树的后序遍历 (Leetcode)

螺旋矩阵

二叉搜索树的第k大节点（剑指 Offer 54）

整数反转 (Leetcode)

Java

HashMap 与 ConcurrentHashMap 的实现原理是怎样的？ ConcurrentHashMap 是如何保证线程安全的？ 中等 参考1 参考2 参考3

简述 CAS 原理，什么是 ABA 问题，怎么解决？

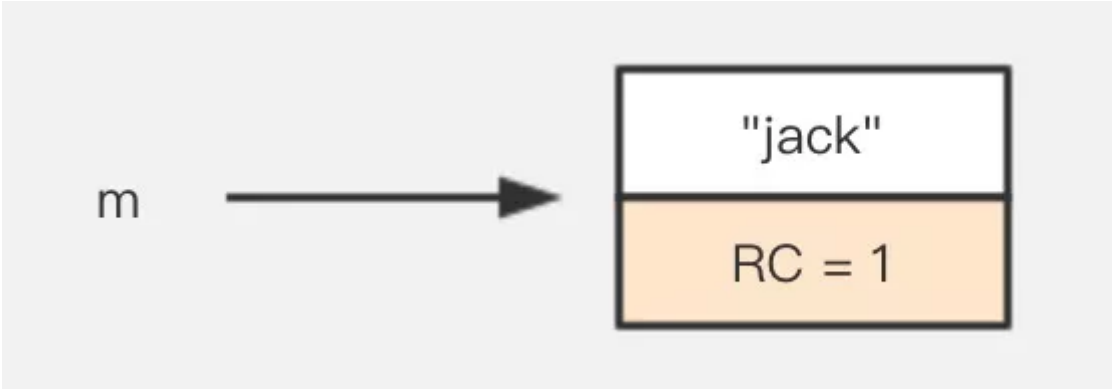
Java 中垃圾回收机制中如何判断对象需要回收？常见的 GC 回收算法有哪些？ 中等 参考1 参考2 参考3

引用计数算法

引用计数算法（Reachability Counting）是通过在对象头中分配一个空间来保存该对象被引用的次数（Reference Count）。如果该对象被其它对象引用，则它的引用计数加1，如果删除对该对象的引用，那么它的引用计数就减1，当该对象的引用计数为0时，那么该对象就会被回收。

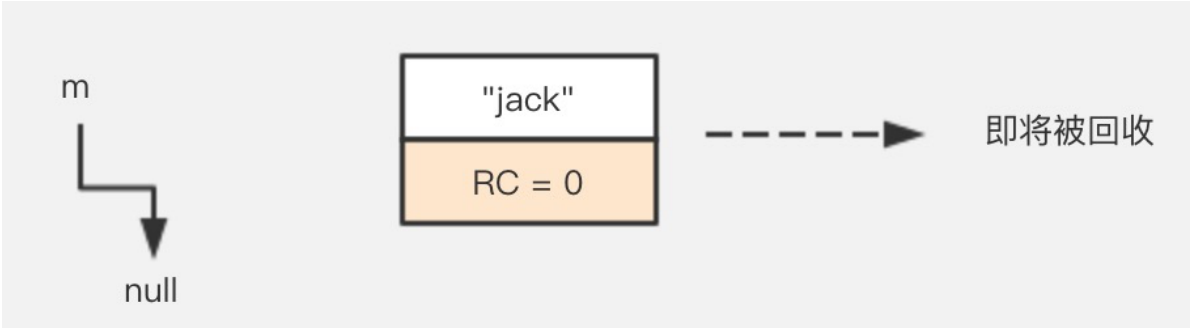
```
String m = new String("jack");
```

先创建一个字符串，这时候"jack"有一个引用，就是 m。



然后将 m 设置为 null，这时候"jack"的引用次数就等于0了，在引用计数算法中，意味着这块内容就需要被回收了。

```
m = null;
```



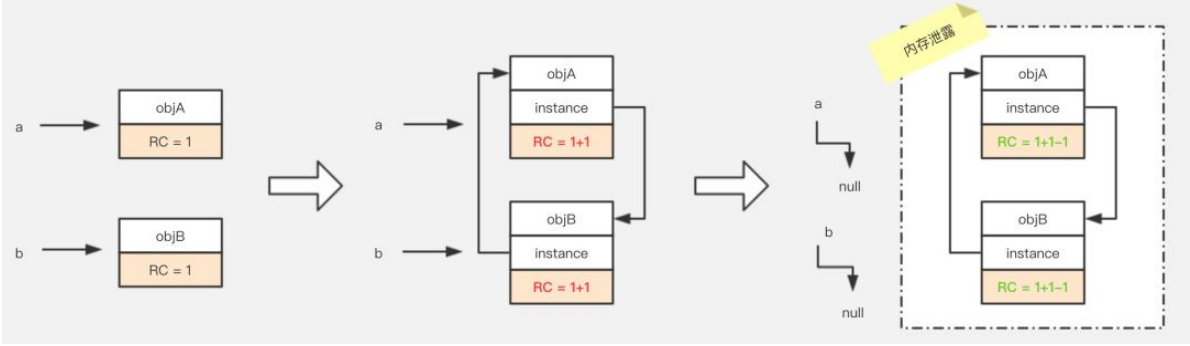
引用计数算法是将垃圾回收分摊到整个应用程序的运行当中了，而不是在进行垃圾收集时，要挂起整个应用的运行，直到对堆中所有对象的处理都结束。因此，采用引用计数的垃圾收集不属于严格意义上的"Stop-The-World"的垃圾收集机制。

看似很美好，但我们知道JVM的垃圾回收就是"Stop-The-World"的，那是什么原因导致我们最终放弃了引用计数算法呢？看下面的例子。

```
public class ReferenceCountingGC {

    public Object instance;

    public ReferenceCountingGC(String name){}
}

public static void testGC(){

    ReferenceCountingGC a = new ReferenceCountingGC("objA");
    ReferenceCountingGC b = new ReferenceCountingGC("objB");

    a.instance = b;
    b.instance = a;

    a = null;
    b = null;
}
```

\1. 定义2个对象

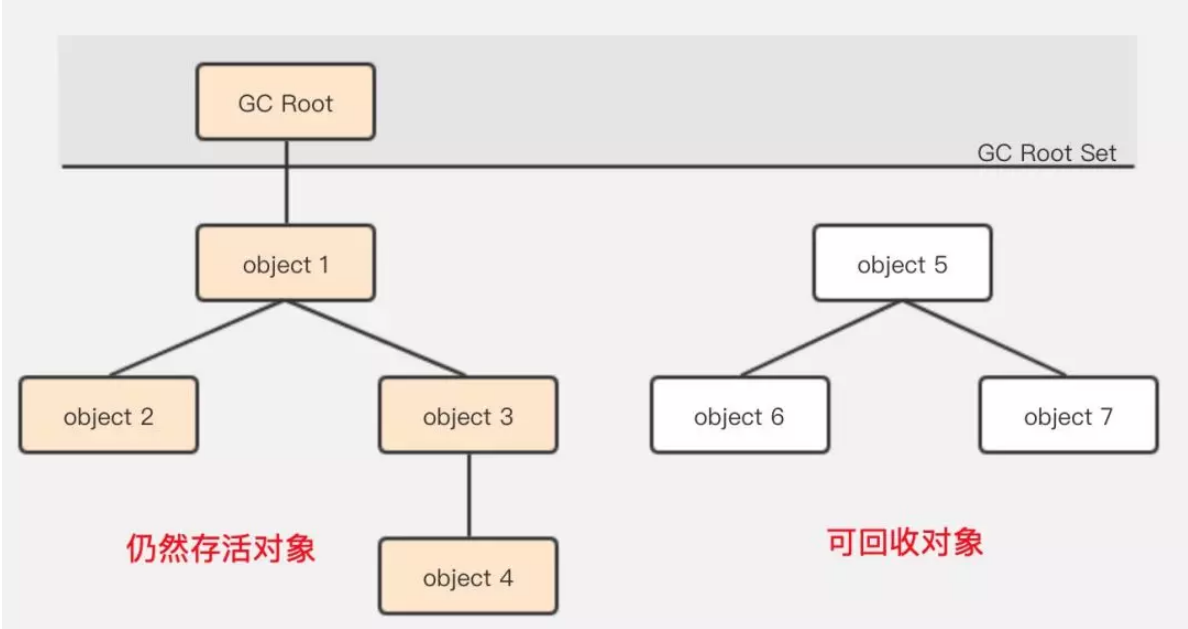
\2. 相互引用
\3. 置空各自的声明引用



我们可以看到，最后这2个对象已经不可能再被访问了，但由于他们相互引用着对方，导致它们的引用计数永远都不会为0，通过引用计数算法，也就永远无法通知GC收集器回收它们。

可达性分析算法

可达性分析算法（Reachability Analysis）的基本思路是，通过一些被称为引用链（GC Roots）的对象作为起点，从这些节点开始向下搜索，搜索走过的路径被称为（Reference Chain），当一个对象到 GC Roots 没有任何引用链相连时（即从 GC Roots 节点到该节点不可达），则证明该对象是不可用的。



通过可达性算法，成功解决了引用计数所无法解决的问题-“循环依赖”，只要你无法与 GC Root 建立直接或间接的连接，系统就会判定你为可回收对象。那这样就引申出了另一个问题，哪些属于 GC Root。

简述 Synchronized，Volatile，可重入锁的不同使用场景及优缺点 困难 参考1

Synchronized	Volatile	可重入锁
1.只适用于区块或方法。	1.仅适用于变量。	1.它也只适用于变量。
2.采用同步修饰符实现了一种基于锁的并发算法，即受锁的限制。	2.在这里，易失性提供了实现非阻塞算法的能力，该算法具有更高的可伸缩性。	2.原子算法也给出了实现非阻塞算法的能力。
3.由于锁的获取和释放，性能与易失性和原子关键字相比相对较低。	3.与同步关键字相比，性能相对较高。	3.与易失关键字和同步关键字相比，性能相对较高。
4.由于锁的性质，它也不能幸免于诸如死锁和活锁之类的并发危险。	4.由于它的非锁定特性，它不受死锁和活锁等并发性危害的影响。	4.由于它的非锁定特性，它不受死锁和活锁等并发性危害的影响。

Synchronized 关键字底层是如何实现的？它与 Lock 相比优缺点分别是什么？ 中等 参考1 参考2

线程池是如何实现的？简述线程池的任务策略 中等

简述 ArrayList 与 LinkedList 的底层实现以及常见操作的时间复杂度 简单

简述 BIO, NIO, AIO 的区别

== 和 equals() 的区别？ 简单 参考1

==用于比较原语。

例如：

```
        String string1 = "Ravi";
        String string2 = "Ravi";
        String string3 = new String("Ravi");
        String string4 = new String("Prakash");

        System.out.println(string1 == string2); // true because same reference in
string pool
        System.out.println(string1 == string3); // false
```

2)equals()用于比较对象。例如：

```
        System.out.println(string1.equals(string2)); // true equals() comparison
of values in the objects
        System.out.println(string1.equals(string3)); // true
        System.out.println(string1.equals(string4)); // false
```

Java 常见锁有哪些？ReetrantLock 是怎么实现的？ 中等

JMM 中内存模型是怎样的？什么是指令序列重排序？ 中等 参考1

我们常说的JVM内存模式指的是JVM的内存分区；而Java内存模式是一种虚拟机规范。

Java虚拟机规范中定义了Java内存模型（Java Memory Model，JMM），用于屏蔽掉各种硬件和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的并发效果，JMM规范了Java虚拟机与计算机内存是如何协同工作的：规定了一个线程如何和何时可以看到由其他线程修改过后的共享变量的值，以及在必须时如何同步的访问共享变量。

原始的Java内存模型存在一些不足，因此Java内存模型在Java1.5时被重新修订。这个版本的Java内存模型在Java8中仍然在使用。

Java内存模型（不仅仅是JVM内存分区）：调用栈和本地变量存放在线程栈上，对象存放在堆上。

指令序列的重排序：

1）编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。

2）指令级并行的重排序。现代处理器采用了指令级并行技术（Instruction-LevelParallelism，ILP）来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

3）内存系统的重排序。由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

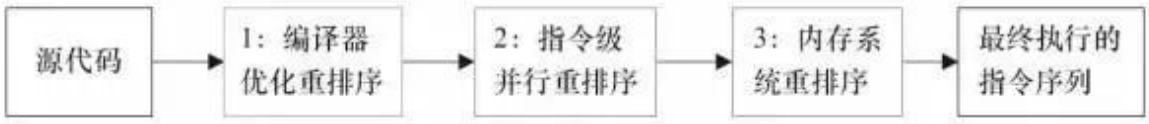


图3-3 从源码到最终执行的指令序列的示意图

每个处理器上的写缓冲区，仅仅对它所在的处理器可见。这会导致处理器执行内存操作的顺序可能会与内存实际的操作执行顺序不一致。由于现代的处理器都会使用写缓冲区，因此现代的处理器都会允许对写-读操作进行重排序：

表3-2 处理器的重排序规则

规则 / 处理器	Load-Load	Load-Store	Store-Store	Store-Load	数据依赖
SPARC-TSO	N	N	N	Y	N
x86	N	N	N	Y	N
IA64	Y	Y	Y	Y	N
PowerPC	Y	Y	Y	Y	N

注意，表3-2单元格中的“N”表示处理器不允许两个操作重排序，“Y”表示允许重排序。

实现单例设计模式（懒汉，饿汉） 中等 参考1 参考2

1、懒汉式，线程不安全

是否 Lazy 初始化：是

是否多线程安全：否

实现难度：易

描述：这种方式是最基本的实现方式，这种实现最大的问题就是不支持多线程。因为没有加锁 synchronized，所以严格意义上它并不算单例模式。
这种方式 lazy loading 很明显，不要求线程安全，在多线程不能正常工作。

实例

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}

    public static Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
    }
}
```

接下来介绍的几种实现方式都支持多线程，但是在性能上有所差异。

2、懒汉式，线程安全

是否 Lazy 初始化：是

是否多线程安全：是

实现难度：易

描述：这种方式具备很好的 lazy loading，能够在多线程中很好的工作，但是，效率很低，99% 情况下不需要同步。
优点：第一次调用才初始化，避免内存浪费。
缺点：必须加锁 synchronized 才能保证单例，但加锁会影响效率。
getInstance() 的性能对应用程序不是很关键（该方法使用不太频繁）。

实例

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}
    public static synchronized Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
    }
}
```

3、饿汉式

是否 Lazy 初始化：否

是否多线程安全：是

实现难度：易

描述：这种方式比较常用，但容易产生垃圾对象。
优点：没有加锁，执行效率会提高。
缺点：类加载时就初始化，浪费内存。
它基于 classloader 机制避免了多线程的同步问题，不过，instance 在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用 getInstance 方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化 instance 显然没有达到 lazy loading 的效果。

实例

```
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton (){}
    public static Singleton getInstance() {
    return instance;
    }
}
```

volatile 关键字解决了什么问题，它的实现原理是什么？ 中等 参考1

JVM 内存是如何对应到操作系统内存的？ 中等 参考1 参考2

简述 Spring bean 的生命周期 中等 参考1 参考2

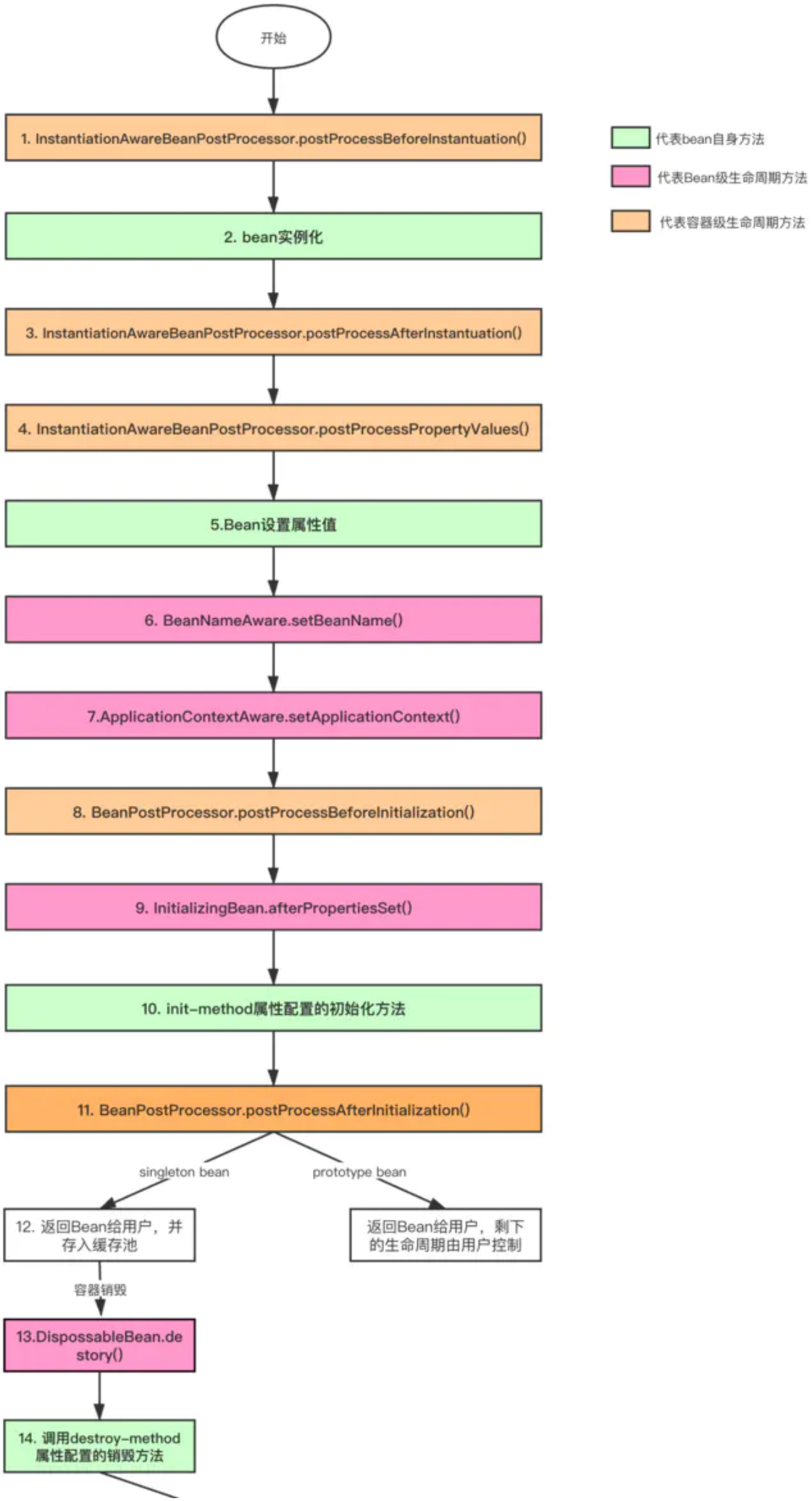
我们需要明确的是，在这里我们的Bean的生命周期主要指的是`singleton bean`，对prototype bean来说，当用户getBean获得prototype bean的实例后，IOC容器就不再对当前实例进行管理，而是把管理权交由用户，此后再getBean生成的是新的实例。对于request/session/application/websocket 这几种scope的bean我们在此不谈。

在不同的容器中，Bean的生命周期开始的时间不同。对于ApplicationContext来说，当容器启动的时候，bean就已经实例化了。而对于BeanFactory来说，直到调用`getBean()`方法的时候才进行实例化。

我们知道对于普通的java对象来说，它们的生命周期就是

- 实例化
- 不再使用的时候通过垃圾回收机制进行回收

但是对于Bean来说却不是这样。Bean的生命周期如下图所示


开始
1. InstantiationAwareBeanPostProcessor.postProcessBeforeInstantuation()
2. bean实例化
3. InstantiationAwareBeanPostProcessor.postProcessAfterInstantuation()
4. InstantiationAwareBeanPostProcessor.postProcessPropertyValues()
5.Bean设置属性值
6. BeanNameAware.setBeanName()
7.ApplicationContextAware.setApplicationContext()
8. BeanPostProcessor.postProcessBeforeInitialization()
9. InitializingBean.afterPropertiesSet()
10. init-method属性配置的初始化方法
11. BeanPostProcessor.postProcessAfterInitialization()
singleton bean
prototype bean
12. 返回Bean给用户，并存入缓存池
返回Bean给用户，剩下的生命周期由用户控制
容器销毁
13.DisposableBean.destory()
14. 调用destroy-method属性配置的销毁方法
代表bean自身方法
代表Bean级生命周期方法
代表容器级生命周期方法

对于如上这些方法，我们可以分成如下几类

1. `Bean自身的方法`：比如构造函数、getter/setter以及init-method和destory-method所指定的方法等
2. `Bean级生命周期方法`：可以理解为Bean类直接实现接口的方法，比如BeanNameAware、BeanFactoryAware、ApplicationContextAware、InitializingBean、DisposableBean等方法，这些方法只对当前Bean生效
3. `容器级的方法(BeanPostProcessor一系列接口)`：主要是后处理器方法，比如上图的InstantiationAwareBeanPostProcessor、BeanPostProcessor接口方法。这些接口的实现类是独立于bean的，并且会注册到Spring容器中。在Spring容器创建任何Bean的时候，这些后处理器都会发生作用。
4. `工厂后处理器方法（BeanFactoryProcessor一系列接口）`：包括AspectJWeavingEnabler、CustomAutowireConfigurer、ConfigurationClassPostProcessor等。这些都是Spring框架中已经实现好的BeanFactoryPostProcessor，用来实现某些特定的功能。

Bean自身的方法和Bean级生命周期方法都只对`当前Bean`起作用，但是容器级生命周期方法和工厂后处理器方法是对`所有的bean`都起作用

对于这几类方法，1 2 4都很好理解，下面我们重点来说一下什么是BeanPostProcessor和BeanFactoryPostProcessor

以我个人理解来说，BeanPostProcessor和BeanFactoryPostProcessor就是`Spring创建的扩展点`，用户可以创建自己的实现类来修改Bean或者BeanFactory

注意对于ApplicatonContext来说，容器可以`自动检测并加载`BeanPostProcessor和BeanFactoryPostProcessor，但是BeanFactory不行，需要自己调用方法`手动注册`。BeanPostProcessor和BeanFactoryPostProcessor都可以有多个。ApplicationContext也可以根据org.springframework.core.PriorityOrdered和org.springframework.core.Ordered来进行`自定义排序`，但是BeanFactory不可以，默认顺序就是`注册顺序`。

这里我需要说明下面两个容易混淆的单词：

- `Instantiation`:实例化，指的是调用`构造函数`进行实例化
- `Initialization`:初始化，在Bean的声明周期中指的是`init-method`所指定的方法或者是`InitializingBean.afterPropertiesSet()`方法

简述 Java 锁升级的机制

简述 Java AQS 的原理以及使用场景

简述 Spring AOP 的原理 中等 参考1 参考2

AOP是一种编程范例，旨在通过允许交叉关注点的分离来增加模块化。它通过在不修改代码本身的情况下向现有代码添加附加行为来做到这一点。

Java 中接口和抽象类的区别 简单

如何解决 Spring 的循环依赖问题?

简述 Spring 注解的实现原理

什么是公平锁？什么是非公平锁？ 简单

在一个静态方法内调用一个非静态成员为什么是非法的?

什么是设计模式，描述几个常用的设计模式 中等

Java 异常有哪些类型?

网络协议

简述 TCP 三次握手以及四次挥手的流程。为什么需要三次握手以及四次挥手？ 中等 参考1 参考2

客户端和服务端通信前要进行连接，“3次握手”的作用就是`双方都能明确自己和对方的收、发能力是正常的`。

`第一次握手`：客户端发送网络包，服务端收到了。这样服务端就能得出结论：客户端的发送能力、服务端的接收能力是正常的。

`第二次握手`：服务端发包，客户端收到了。这样客户端就能得出结论：服务端的接收、发送能力，客户端的接收、发送能力是正常的。 从客户端的视角来看，我接到了服务端发送过来的响应数据包，说明服务端接收到了我在第一次握手时发送的网络包，并且成功发送了响应数据包，这就说明，服务端的接收、发送能力正常。而另一方面，我收到了服务端的响应数据包，说明我第一次发送的网络包成功到达服务端，这样，我自己的发送和接收能力也是正常的。

`第三次握手`：客户端发包，服务端收到了。这样服务端就能得出结论：客户端的接收、发送能力，服务端的发送、接收能力是正常的。 第一、二次握手后，服务端并不知道客户端的接收能力以及自己的发送能力是否正常。而在第三次握手时，服务端收到了客户端对第二次握手作的回应。从服务端的角度，我在第二次握手时的响应数据发送出去了，客户端接收到了。所以，我的发送能力是正常的。而客户端的接收能力也是正常的。

经历了上面的三次握手过程，客户端和服务端都确认了自己的接收、发送能力是正常的。之后就可以正常通信了。

每次都是接收到数据包的一方可以得到一些结论，发送的一方其实没有任何头绪。我虽然有发包的动作，但是我怎么知道我有没有发出去，而对方有没有接收到呢？

而从上面的过程可以看到，最少是需要三次握手过程的。两次达不到让双方都得出自己、对方的接收、发送能力都正常的结论。其实每次收到网络包的一方至少是可以得到：对方的发送、我方的接收是正常的。而每一步都是有关联的，下一次的“响应”是由于第一次的“请求”触发，因此每次握手其实是可以得到额外的结论的。比如第三次握手时，服务端收到数据包，表明看服务端只能得到客户端的发送能力、服务端的接收能力是正常的，但是结合第二次，说明服务端在第二次发送的响应包，客户端接收到了，并且作出了响应，从而得到额外的结论：客户端的接收、服务端的发送是正常的。

用表格总结一下：

视角	客收	客发	服收	服发
客视角	二	一 + 二	一 + 二	二
服视角	二 + 三	一	一	二 + 三

三次握手的过程的示意图如下：

TCP 的连接的拆除需要发送四个包，因此称为四次挥手(Four-way handshake)，也叫做改进的三次握手。客户端或服务器均可主动发起挥手动作，在 socket 编程中，任何一方执行 `close()` 操作即可产生挥手操作。

- 第一次挥手(FIN=1，seq=x)

  假设客户端想要关闭连接，客户端发送一个 FIN 标志位置为1的包，表示自己已经没有数据可以发送了，但是仍然可以接受数据。

  发送完毕后，客户端进入 `FIN_WAIT_1` 状态。

- 第二次挥手(ACK=1，ACKnum=x+1)

  服务器端确认客户端的 FIN 包，发送一个确认包，表明自己接受到了客户端关闭连接的请求，但还没有准备好关闭连接。

发送完毕后，服务器端进入 `CLOSE_WAIT` 状态，客户端接收到这个确认包之后，进入 `FIN_WAIT_2` 状态，等待服务器端关闭连接。

- 第三次挥手(FIN=1，seq=y)

  服务器端准备好关闭连接时，向客户端发送结束连接请求，FIN 置为1。

  发送完毕后，服务器端进入 `LAST_ACK` 状态，等待来自客户端的最后一个ACK。

- 第四次挥手(ACK=1，ACKnum=y+1)

  客户端接收到来自服务器端的关闭请求，发送一个确认包，并进入 `TIME_WAIT` 状态，等待可能出现的要求重传的 ACK 包。

  服务器端接收到这个确认包之后，关闭连接，进入 `CLOSED` 状态。

  客户端等待了某个固定时间（两个最大段生命周期，2MSL，2 Maximum Segment Lifetime）之后，没有收到服务器端的 ACK ，认为服务器端已经正常关闭连接，于是自己也关闭连接，进入 `CLOSED` 状态。

从输入 URL 到展现页面的全过程 困难 参考1

DNS 查询服务器的基本流程是什么？DNS 劫持是什么？ 中等 参考1 参考2

1. 用户在 Web 浏览器中键入 “example.com”，查询传输到 Internet 中，并被 DNS 递归解析器接收。
2. 接着，解析器查询 DNS 根域名服务器（.）。
3. 然后，根服务器使用存储其域信息的顶级域（TLD）DNS 服务器（例如 .com 或 .net）的地址响应该解析器。在搜索 example.com 时，我们的请求指向 .com TLD。
4. 然后，解析器向 .com TLD 发出请求。
5. TLD 服务器随后使用该域的域名服务器 example.com 的 IP 地址进行响应。
6. 最后，递归解析器将查询发送到域的域名服务器。
7. example.com 的 IP 地址而后从域名服务器返回解析器。
8. 然后 DNS 解析器使用最初请求的域的 IP 地址响应 Web 浏览器。
9. DNS 查找的这 8 个步骤返回 example.com 的 IP 地址后，浏览器便能发出对该网页的请求：
10. 浏览器向该 IP 地址发出 HTTP 请求。
11. 位于该 IP 的服务器返回将在浏览器中呈现的网页（第 10 步）。

TCP 怎么保证可靠传输？ 中等 参考1 参考2

1. 应用数据被分割成 TCP 认为最适合发送的数据块。
2. TCP 给发送的每一个包进行编号，接收方对数据包进行排序，把有序数据传送给应用层。
3. **校验和：** TCP 将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错，TCP 将丢弃这个报文段和不确认收到此报文段。
4. TCP 的接收端会丢弃重复的数据。

5. **流量控制：** TCP 连接的每一方都有固定大小的缓冲空间，TCP的接收端只允许发送端发送接收端缓冲区能接纳的数据。当接收方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。TCP 使用的流量控制协议是可变大小的滑动窗口协议。 （TCP 利用滑动窗口实现流量控制）
6. **拥塞控制：** 当网络拥塞时，减少数据的发送。
7. **ARQ协议：** 也是为了实现可靠传输的，它的基本原理就是每发完一个分组就停止发送，等待对方确认。在收到确认后再发下一个分组。
8. **超时重传：** 当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

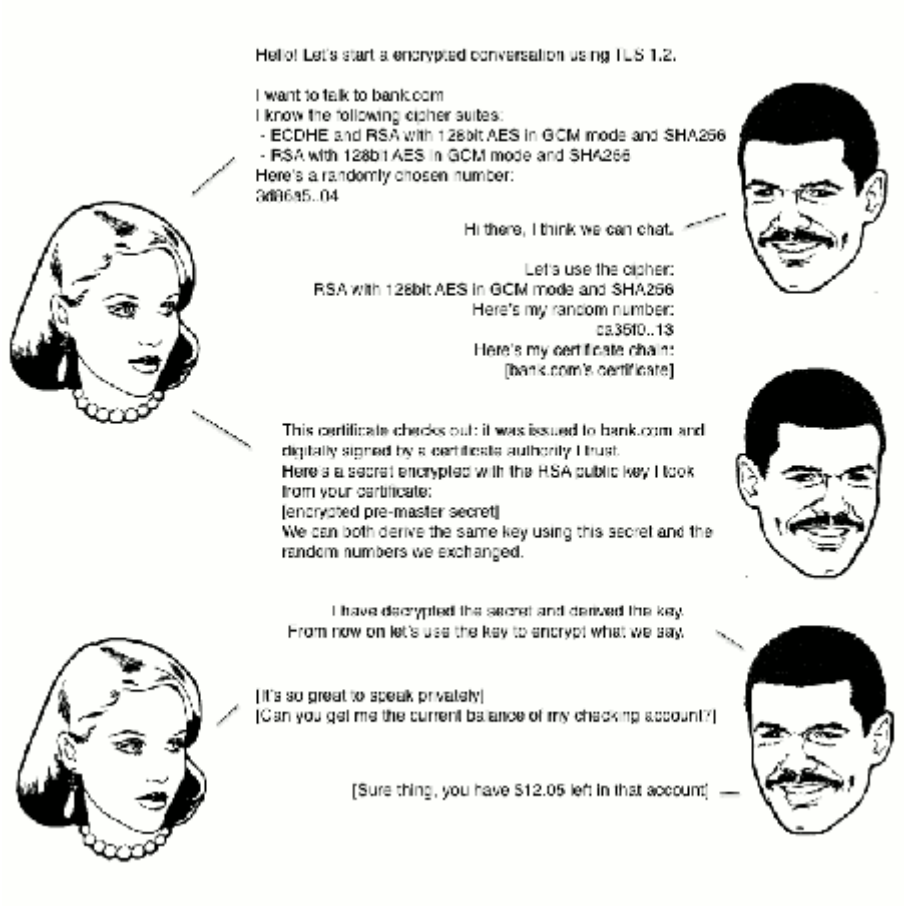
什么是 TCP 粘包和拆包？ 简单

简述 HTTPS 的加密与认证过程 中等 参考1

一、SSL协议的握手过程

开始加密通信之前，客户端和服务器首先必须建立连接和交换参数，这个过程叫做握手（handshake）。

假定客户端叫做爱丽丝，服务器叫做鲍勃，整个握手过程可以用下图说明（点击看大图）。

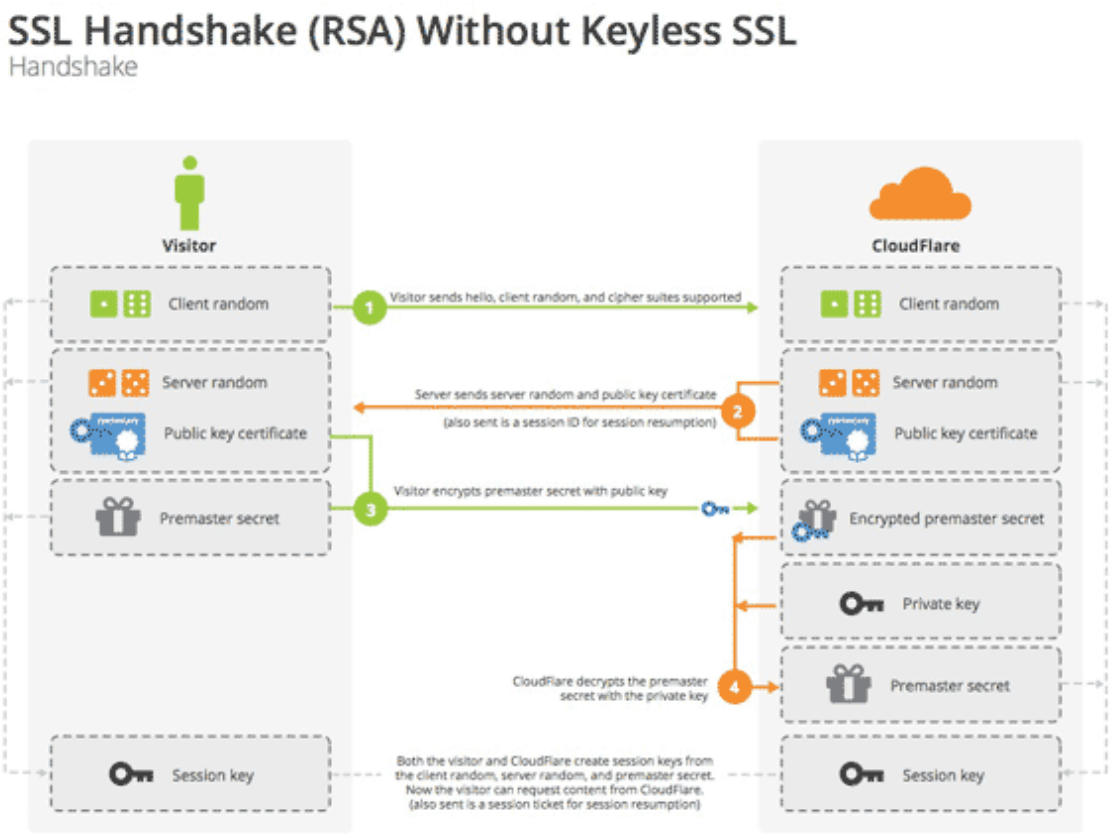


握手阶段分成五步。

> 第一步，爱丽丝给出协议版本号、一个客户端生成的随机数（Client random），以及客户端支持的加密方法。

> 第二步，鲍勃确认双方使用的加密方法，并给出数字证书、以及一个服务器生成的随机数（Server random）。
>
> 第三步，爱丽丝确认数字证书有效，然后生成一个新的随机数（Premaster secret），并使用数字证书中的公钥，加密这个随机数，发给鲍勃。
>
> 第四步，鲍勃使用自己的私钥，获取爱丽丝发来的随机数（即Premaster secret）。
>
> 第五步，爱丽丝和鲍勃根据约定的加密方法，使用前面的三个随机数，生成"对话密钥"（session key），用来加密接下来的整个对话过程。

上面的五步，画成一张图，就是下面这样。



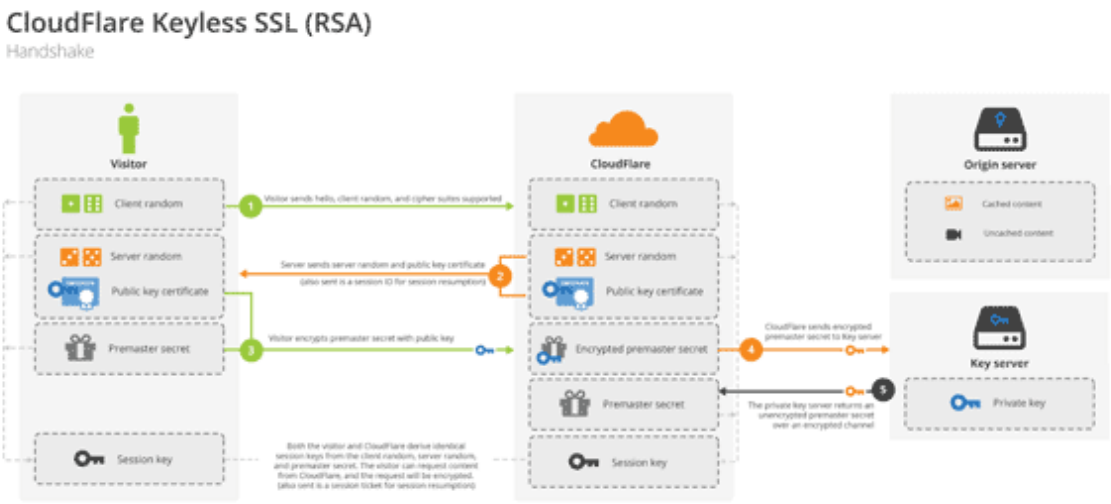
二、私钥的作用

握手阶段有三点需要注意。

> （1）生成对话密钥一共需要三个随机数。
>
> （2）握手之后的对话使用"对话密钥"加密（对称加密），服务器的公钥和私钥只用于加密和解密"对话密钥"（非对称加密），无其他作用。
>
> （3）服务器公钥放在服务器的数字证书之中。

从上面第二点可知，整个对话过程中（握手阶段和其后的对话），服务器的公钥和私钥只需要用到一次。这就是CloudFlare能够提供Keyless服务的根本原因。

某些客户（比如银行）想要使用外部CDN，加快自家网站的访问速度，但是出于安全考虑，不能把私钥交给CDN服务商。这时，完全可以把私钥留在自家服务器，只用来解密对话密钥，其他步骤都让CDN服务商去完成。



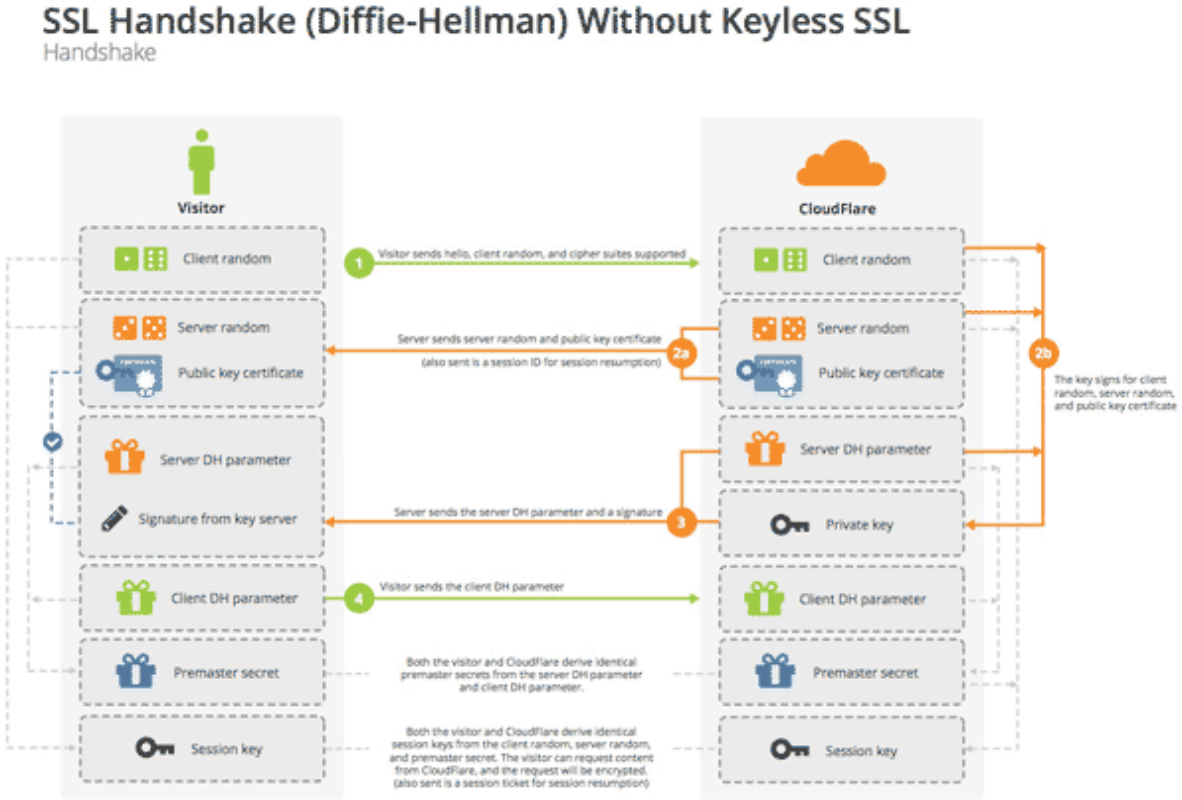
上图中，银行的服务器只参与第四步，后面的对话都不再会用到私钥了。

三、DH算法的握手阶段

整个握手阶段都不加密（也没法加密），都是明文的。因此，如果有人窃听通信，他可以知道双方选择的加密方法，以及三个随机数中的两个。整个通话的安全，只取决于第三个随机数（Premaster secret）能不能被破解。

虽然理论上，只要服务器的公钥足够长（比如2048位），那么Premaster secret可以保证不被破解。但是为了足够安全，我们可以考虑把握手阶段的算法从默认的RSA算法，改为 Diffie-Hellman算法（简称DH算法）。

采用DH算法后，Premaster secret不需要传递，双方只要交换各自的参数，就可以算出这个随机数。



上图中，第三步和第四步由传递Premaster secret变成了传递DH算法所需的参数，然后双方各自算出Premaster secret。这样就提高了安全性。

四、session的恢复

握手阶段用来建立SSL连接。如果出于某种原因，对话中断，就需要重新握手。

这时有两种方法可以恢复原来的session：一种叫做session ID，另一种叫做session ticket。

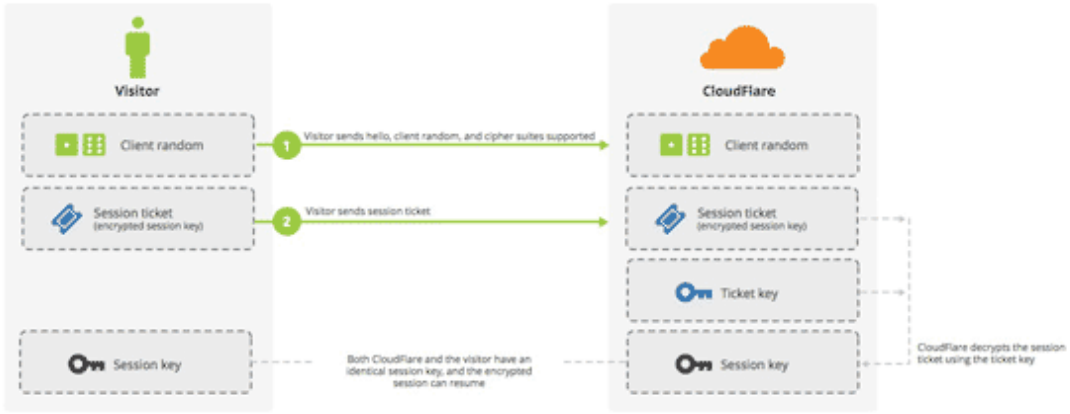
session ID的思想很简单，就是每一次对话都有一个编号（session ID）。如果对话中断，下次重连的时候，只要客户端给出这个编号，且服务器有这个编号的记录，双方就可以重新使用已有的"对话密钥"，而不必重新生成一把。

Session resume with session ID

上图中，客户端给出session ID，服务器确认该编号存在，双方就不再进行握手阶段剩余的步骤，而直接用已有的对话密钥进行加密通信。

session ID是目前所有浏览器都支持的方法，但是它的缺点在于session ID往往只保留在一台服务器上。所以，如果客户端的请求发到另一台服务器，就无法恢复对话。session ticket就是为了解决这个问题而诞生的，目前只有Firefox和Chrome浏览器支持。

Session resume with session ticket



上图中，客户端不再发送session ID，而是发送一个服务器在上一次对话中发送过来的session ticket。这个session ticket是加密的，只有服务器才能解密，其中包括本次对话的主要信息，比如对话密钥和加密方法。当服务器收到session ticket以后，解密后就不必重新生成对话密钥了。

TCP 与 UDP 在网络协议中的哪一层，他们之间有什么区别？ 简单 参考1

tcp	UDP
这是一个面向连接的协议.	这是一个无连接的协议。
TCP以字节流的形式读取数据，并将消息传输到段边界。	UDP消息包含一个接一个发送的数据包。它还检查抵达时的完整性。
TCP消息在互联网上从一台计算机传到另一台计算机。	它不是基于连接的，所以一个程序可以向另一个程序发送大量的数据包.
TCP按特定顺序重新排列数据包。	UDP协议没有固定的顺序，因为所有的数据包都是相互独立的。
TCP的速度较慢。	由于不尝试错误恢复，UDP更快。
头大小为20字节。	标题大小为8字节。
TCP很重.在发送任何用户数据之前，TCP需要三个数据包来设置套接字连接。	UDP是轻量级的。没有跟踪连接，没有消息的排序等。
TCP进行错误检查并进行错误恢复。	UDP执行错误检查，但它丢弃错误的数据包。
确认段	没有确认段
使用像SYN，SYN-ACK，ACK这样的握手协议	不握手(如此无连接协议)
TCP是可靠的，因为它保证将数据传送到目标路由器。	在UDP中无法保证将数据传递到目标。
TCP提供广泛的错误检查机制，因为它提供流控制和数据确认。	UDP只有一个用于校验和的错误检查机制。

简述对称与非对称加密的概念 简单

如何解决 TCP 传输丢包问题?

数据库

MySQL 为什么使用 B+ 树来作索引，对比 B 树它的优点和缺点是什么？ 中等 参考1 参考2

这个**B+树**结构类似于B树结构。最重要的区别是：

- 内部节点只存储值；它们不存储指向实际行的指针。叶节点存储值和行指针。这减少了内部节点的大小，允许在同一内存页上存储更多的节点。反过来，这又增加了分支因子。随着分支因子的增加，树的高度降低，导致磁盘I/O操作减少。
- B+树中的叶节点是链接的，所以我们只需通过一次就可以进行一次完整的扫描。当我们需要找到给定范围内的所有数据时，这是非常有帮助的。因为行指针存储在内部节点和叶节点中，所以这在B

树中是不可能的。

- 在B+树结构中执行删除操作比在B树中执行要容易得多.这是因为我们不需要从内部节点中删除值。在B+树结构中，我们将重复相同的值；在B树结构中，每个值都是唯一的。在B+Tree中，我们将在叶节点中存储一个值和一个数据指针，但该值也可以存储在内部节点中(用于指向子节点)。
- B树的优点是我们可以很快地找到靠近根的值，而在B+Tree中，我们需要一直到叶节点查找任何值。

简述一致性哈希算法的实现方式及原理 困难

简述什么是最左匹配原则 简单

为什么 Redis 在单线程下能如此快？ 中等 参考1

MySQL 联合索引底层原理是什么?

数据库的事务隔离级别有哪些？各有哪些优缺点？ 中等 参考1 参考2

SQL 标准定义了四种隔离级别，MySQL 全都支持。这四种隔离级别分别是：

1. 读未提交（READ UNCOMMITTED）
2. 读提交 （READ COMMITTED）
3. 可重复读 （REPEATABLE READ）
4. 串行化 （SERIALIZABLE）

从上往下，隔离强度逐渐增强，性能逐渐变差。采用哪种隔离级别要根据系统需求权衡决定，其中，**可重复读**是 MySQL 的默认级别。

事务隔离其实就是为了解决上面提到的脏读、不可重复读、幻读这几个问题，下面展示了 4 种隔离级别对这三个问题的解决程度。

隔离级别	脏读	不可重复读	幻读
读未提交	可能	可能	可能
读提交	不可能	可能	可能
可重复读	不可能	不可能	可能
串行化	不可能	不可能	不可能

只有串行化的隔离级别解决了全部这 3 个问题，其他的 3 个隔离级别都有缺陷。

Redis 如何实现分布式锁？ 困难 参考1 参考2

在尝试克服上述单实例设置的限制之前，让我们先讨论一下在这种简单情况下实现分布式锁的正确做法，实际上这是一种可行的方案，尽管存在竞态，结果仍然是可接受的，另外，这里讨论的单实例加锁方法也是分布式加锁算法的基础。

获取锁使用命令:

```
SET resource_name my_random_value NX PX 30000
```

这个命令仅在不存在key的时候才能被执行成功（NX选项），并且这个key有一个30秒的自动失效时间（PX属性）。这个key的值是"my*random*value"(一个随机值），这个值在所有的客户端必须是唯一的，所有同一key的获取者（竞争者）这个值都不能一样。

value的值必须是随机数主要是为了更安全的释放锁，释放锁的时候使用脚本告诉Redis:只有key存在并且存储的值和我指定的值一样才能告诉我删除成功。可以通过以下Lua脚本实现：

```
if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

使用这种方式释放锁可以避免删除别的客户端获取成功的锁。举个例子：客户端A取得资源锁，但是紧接着被一个其他操作阻塞了，当客户端A运行完毕其他操作后要释放锁时，原来的锁早已超时并且被Redis自动释放，并且在这期间资源锁又被客户端B再次获取到。如果仅使用DEL命令将key删除，那么这种情况就会把客户端B的锁给删除掉。使用Lua脚本就不会存在这种情况，因为脚本仅会删除value等于客户端A的value的key（value相当于客户端的一个签名）。

这个随机字符串应该怎么设置？我认为它应该是从/dev/urandom产生的一个20字节随机数，但是我想你可以找到比这种方法代价更小的方法，只要这个数在你的任务中是唯一的就行。例如一种安全可行的方法是使用/dev/urandom作为RC4的种子和源产生一个伪随机流;一种更简单的方法是把以毫秒为单位的unix时间和客户端ID拼接起来，理论上不是完全安全，但是在多数情况下可以满足需求.

key的失效时间，被称作"锁定有效期"。它不仅是key自动失效时间，而且还是一个客户端持有锁多长时间后可以被另外一个客户端重新获得。

截至到目前，我们已经有较好的方法获取锁和释放锁。基于Redis单实例，假设这个单实例总是可用，这种方法已经足够安全。现在让我们扩展一下，假设Redis没有总是可用的保障。

什么情况下会发生死锁，如何解决死锁？ 中等 参考1 参考2

死锁是一种情况，不同的事务无法进行，因为每个事务持有另一个需要的锁。因为这两个事务都在等待资源变得可用，所以这两个事务都没有释放它所持有的锁。

当事务锁定多个表中的行时(通过诸如 `UPDATE` 或 `SELECT ... FOR UPDATE`)，但顺序相反。当这样的语句锁定索引记录和间隙的范围时，也会出现死锁，每个事务都会因为时间问题而获得一些锁，而不是其他锁

当启用死锁检测(默认值)并发生死锁时，`InnoDB`检测条件并回滚其中一个事务(受害者)。如果禁用死锁检测，则使用`innodb_deadlock_detect`变量，`InnoDB`依赖于`innodb_lock_wait_timeout`设置回滚在死锁情况下的事务。因此，即使您的应用程序逻辑是正确的，您仍然必须处理事务必须重新尝试的情况。中查看最后一个死锁的步骤。`InnoDB`用户事务，使用`SHOW ENGINE INNODB STATUS`。如果频繁死锁突出了事务结构或应用程序错误处理的问题，则启用`innodb_print_all_deadlocks`

Redis 有几种数据结构？Zset 是如何实现的？ 中等 参考1 参考2

SDS、链表、字典、跳跃表、整数集合、压缩列表

简述常见的负载均衡算法 简单 参考1

1. **圆罗宾**-请求在服务器上平均分配，与服务器权重考虑到了。默认情况下使用此方法(没有启用该方法的指令)：

```
upstream backend {
   # no load balancing method is specified for Round Robin
   server backend1.example.com;
   server backend2.example.com;
}
```

2. **最小连接**-将请求发送到活动连接最少的服务器，再次使用服务器权重考虑到：

```
upstream backend {
    least_conn;
    server backend1.example.com;
    server backend2.example.com;
}
```

3. **IP散列**-从客户端IP地址确定发送请求的服务器。在这种情况下，使用IPv 4地址的前三个八进制或整个IPv 6地址来计算哈希值。该方法确保来自同一地址的请求到达同一台服务器，除非不可用。

```
upstream backend {
    ip_hash;
    server backend1.example.com;
    server backend2.example.com;
}
```

   如果其中一台服务器需要暂时从负载平衡旋转中移除，则可以将其标记为`down`参数，以保留客户端IP地址的当前散列。将由此服务器处理的请求自动发送到组中的下一个服务器：

```
upstream backend {
    server backend1.example.com;
    server backend2.example.com;
    server backend3.example.com down;
}
```

4. 属属**散列**-发送请求的服务器由用户定义的密钥确定，该密钥可以是文本字符串、变量或组合。例如，密钥可以是配对的源IP地址和端口，也可以是URI，如本例所示：

```
upstream backend {
    hash $request_uri consistent;
    server backend1.example.com;
    server backend2.example.com;
}
```

任选`consistent`参数的`hash`指令启用凯塔马一致-哈希负载平衡。根据用户定义的散列密钥值，请求均匀分布在所有上游服务器上。如果上游服务器被添加到上游组或从上游组中移除，则只有几个密钥被重新映射，在负载平衡缓存服务器或其他累积状态的应用程序中，这将最大限度地减少缓存丢失。

5. **最少时间**(仅适用于Nginx Plus)--对于每个请求，Nginx Plus选择的服务器具有最低的平均延迟和最低的活动连接数，其中最低的平均延迟是根据以下哪一个来计算的参数到`least_time`指令包括：
   - `header`-从服务器接收第一个字节的时间
   - `last_byte`-从服务器接收全部响应的时间
   - `last_byte inflight`-在考虑到不完整请求的情况下，从服务器接收全部响应的时间

```
upstream backend {
    least_time header;
    server backend1.example.com;
    server backend2.example.com;
}
```

6. **随机**-每个请求将传递给随机选择的服务器。如果`two`参数指定，首先，Nginx在考虑服务器权重的情况下随机选择两个服务器，然后使用指定的方法选择其中一个服务器：
   - `least_conn`-有效连接最少
   - `least_time=header`(Nginx Plus)-从服务器接收响应头的最短平均时间(`$upstream_header_time`)
   - `least_time=last_byte`(Nginx Plus)-从服务器接收完整响应的最短平均时间(`$upstream_response_time`)

```
upstream backend {
    random two least_time=last_byte;
    server backend1.example.com;
    server backend2.example.com;
    server backend3.example.com;
    server backend4.example.com;
}
```

这个**随机**在多个负载平衡器将请求传递到同一组后端的分布式环境中，应该使用负载平衡方法。对于负载均衡器具有所有请求的完整视图的环境，请使用其他负载平衡方法，例如循环、最少的连接和最少的时间。

简述 MySQL 的主从同步机制，如果同步失败会怎么样？ 中等

MySQL中 InnoDB 和 MyISAM 的区别是什么？ 简单

聚簇索引和非聚簇索引有什么区别？ 简单 参考1 参考2

- 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

简述乐观锁以及悲观锁的区别以及使用场景 简单 参考1 参考2

区别

乐观锁定是一种策略，在读取记录时，记下版本号(其他方法包括日期、时间戳或校验和/散列)，并在写入记录之前检查版本没有更改。当您将记录写回时，您将过滤版本上的更新，以确保它是原子的。(也就是说，在检查版本和将记录写入磁盘之间没有更新)，并在一次命中中更新版本。

如果记录是脏的(即与您的记录不同的版本)，您将中止事务，用户可以重新启动它。

此策略最适用于大容量系统和三层体系结构，在这些体系结构中，您不一定要为会话维护到数据库的连接。在这种情况下，客户端实际上无法维护数据库锁，因为连接是从池中提取的，而且您可能没有使用从一次访问到下一次访问的相同连接。

悲观锁定当您锁定记录以供独占使用时，直到您完成该记录为止。它比乐观锁定具有更好的完整性，但要求您在应用程序设计时小心避免。死锁。若要使用悲观锁定，您需要直接连接到数据库(如两层客户端服务器或者可以独立于连接使用的外部可用事务ID。

在后一种情况下，您使用TxID打开事务，然后使用该ID重新连接。DBMS维护锁并允许您通过TxID选择会话。这就是使用两阶段提交协议(如沙或COM+事务)工作。

使用场景

乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行retry，这样反倒是降低了性能，所以**一般多写的场景下用悲观锁就比较合适。**

简述 MySQL 常见索引类型，介绍一下覆盖索引

如何设计数据库压测方案?

Redis 的 String 数据类型是如何实现的?

操作系统

进程间有哪些通信方式? 困难 参考1 参考2

管道广泛用于两个相关过程之间的通信。这是一个半双工方法，所以第一个进程与第二个进程通信.然而，为了实现全双工，还需要另一根管道.

消息传递：

它是进程通信和同步的一种机制。使用消息传递，流程彼此通信，而不诉诸共享变量。

IPC机制提供两个操作：

- 发送(消息)-消息大小固定或可变
- 收到(电文)

消息队列：

消息队列是存储在内核中的消息的链接列表。它由消息队列标识符标识。该方法提供具有全双工容量的单进程或多进程之间的通信.

直接通讯：

在这种类型的进程间通信过程中，应明确彼此的名称.在该方法中，在一对通信进程之间建立链接，而在每对通信进程之间，只存在一个链接。

间接通信：

间接通信建立，就像只有当进程共享一个公共邮箱时，每对进程共享多个通信链接。一个链接可以与许多进程通信。该链接可以是双向的，也可以是单向的。

共享内存：

共享内存是使用共享内存在所有进程之间建立的两个或多个进程之间共享的内存。这种类型的内存需要通过同步跨所有进程的访问来相互保护。

FIFO：

两个不相关的过程之间的交流。这是一种全双工方法，这意味着第一个进程可以与第二个进程通信，也可能发生相反的情况。

简述自旋锁与互斥锁的使用场景 中等

进程和线程之间有什么区别？ 简单 参考1 参考2

参数	加工过程	螺纹
定义	进程意味着程序正在执行。	线程是进程的一个片段。
轻量级	这个过程不是轻量级的。	线程很轻。
终止时间	进程需要更多的时间来终止。	线程终止所需的时间较少。
创作时间	创造需要更多的时间。	创造所需的时间较少。
通讯	与线程相比，进程之间的通信需要更多的时间。	与进程相比，线程之间的通信需要更少的时间。
上下文切换时间	它需要更多的时间来切换上下文。	上下文切换所需的时间更短。
资源	进程消耗更多的资源。	线程消耗的资源较少。
OS治疗	操作系统分别处理不同的过程。	所有级别的对等线程都被操作系统视为一个单一任务。
记忆	这一过程大多是孤立的。	线程共享内存。
共享	它不共享数据。	线程彼此共享数据。

前端

手写题库 https://github.com/Mayandev/fe-interview-handwrite 困难

const, let, var 关键字有什么区别？ 简单

简述 Javascript 原型以及原型链 中等 参考1

对于使用过基于类的语言 (如 Java 或 C++) 的开发人员来说，JavaScript 有点令人困惑，因为它是动态的，并且本身不提供一个 `class` 实现。（在 ES2015/ES6 中引入了 `class` 关键字，但那只是语法糖，JavaScript 仍然是基于原型的）。

当谈到继承时，JavaScript 只有一种结构：对象。每个实例对象（ object ）都有一个私有属性（称之为 **proto** ）指向它的构造函数的原型对象（**prototype** ）。该原型对象也有一个自己的原型对象(**proto**) ，层层向上直到一个对象的原型对象为 `null`。根据定义， `null` 没有原型，并作为这个**原型链**中的最后一个环节。

几乎所有 JavaScript 中的对象都是位于原型链顶端的 `Object` 的实例。

尽管这种原型继承通常被认为是 JavaScript 的弱点之一，但是原型继承模型本身实际上比经典模型更强大。例如，在原型模型的基础上构建经典模型相当简单。

简述 diff 算法的实现机制和使用场景 中等

简述 CSS 盒模型 简单 参考1

当对一个文档进行布局（lay out）的时候，浏览器的渲染引擎会根据标准之一的 **CSS 基础框盒模型**（**CSS basic box model**），将所有元素表示为一个个矩形的盒子（box）。CSS 决定这些盒子的大小、位置以及属性（例如颜色、背景、边框尺寸...）。

每个盒子由四个部分（或称*区域*）组成，其效用由它们各自的边界（Edge）所定义（原文：defined by their respective edges，可能意指容纳、包含、限制等）。如图，与盒子的四个组成区域相对应，每个盒子有四个边界：*内容边界 Content edge*、*内边距边界 Padding Edge*、*边框边界 Border Edge*、*外边框边界 Margin Edge*。

CSS 的选择器优先级是怎样？ 简单

简述 Dom 节点的不同操作方式 中等

简述 Javascript 中 this 的指向有哪些 简单

前端如何解决线程安全和进程安全的问题?

简述 JavaScript 事件循环机制 中等 参考1

之所以称之为 **事件循环**，是因为它经常按照类似如下的方式来被实现：

```
while (queue.waitForMessage()) {
  queue.processNextMessage();
}
```

Copy to Clipboard

`queue.waitForMessage()` 会同步地等待消息到达(如果当前没有任何消息等待被处理)。

["执行至完成"]

每一个消息完整地执行后，其它消息才会被执行。这为程序的分析提供了一些优秀的特性，包括：当一个函数执行时，它不会被抢占，只有在它运行完毕之后才会去运行任何其他的代码，才能修改这个函数操作的数据。这与C语言不同，例如，如果函数在线程中运行，它可能在任何位置被终止，然后在另一个线程中运行其他代码。

这个模型的一个缺点在于当一个消息需要太长时间才能处理完毕时，Web应用程序就无法处理与用户的交互，例如点击或滚动。为了缓解这个问题，浏览器一般会弹出一个“这个脚本运行时间过长”的对话框。一个良好的习惯是缩短单个消息处理时间，并在可能的情况下将一个消息裁剪成多个消息。

[添加消息]

在浏览器里，每当一个事件发生并且有一个事件监听器绑定在该事件上时，一个消息就会被添加进消息队列。如果没有事件监听器，这个事件将会丢失。所以当一个带有点击事件处理器的元素被点击时，就会像其他事件一样产生一个类似的消息。

函数 `setTimeout` 接受两个参数：待加入队列的消息和一个时间值（可选，默认为 0）。这个时间值代表了消息被实际加入到队列的最小延迟时间。如果队列中没有其它消息并且栈为空，在这段延迟时间过去之后，消息会被马上处理。但是，如果有其它消息，`setTimeout` 消息必须等待其它消息处理完。因此第二个参数仅仅表示最少延迟时间，而非确切的等待时间。

下面的例子演示了这个概念（`setTimeout` 并不会在计时器到期之后直接执行）：

```
const s = new Date().getSeconds();

setTimeout(function() {
  // 输出 "2"，表示回调函数并没有在 500 毫秒之后立即执行
  console.log("Ran after " + (new Date().getSeconds() - s) + " seconds");
}, 500);

while(true) {
  if(new Date().getSeconds() - s >= 2) {
    console.log("Good, looped for 2 seconds");
    break;
  }
}
```

Copy to Clipboard

[零延迟]

零延迟并不意味着回调会立即执行。以 0 为第二参数调用 `setTimeout` 并不表示在 0 毫秒后就立即调用回调函数。

其等待的时间取决于队列里待处理的消息数量。在下面的例子中，`"这是一条消息"` 将会在回调获得处理之前输出到控制台，这是因为延迟参数是运行时处理请求所需的最小等待时间，但并不保证是准确的等待时间。

基本上，`setTimeout` 需要等待当前队列中所有的消息都处理完毕之后才能执行，即使已经超出了由第二参数所指定的时间。

```
(function() {

  console.log('这是开始');

  setTimeout(function cb() {
    console.log('这是来自第一个回调的消息');
  });

  console.log('这是一条消息');

  setTimeout(function cb1() {
    console.log('这是来自第二个回调的消息');
  }, 0);

  console.log('这是结束');

})();
```

```
// "这是开始"
// "这是一条消息"
// "这是结束"
// "这是来自第一个回调的消息"
// "这是来自第二个回调的消息"
```

Copy to Clipboard

[多个运行时互相通信]

一个 web worker 或者一个跨域的 `iframe` 都有自己的栈、堆和消息队列。两个不同的运行时只能通过 `postMessage` 方法进行通信。如果另一个运行时侦听 `message` 事件，则此方法会向该运行时添加消息。

[永不阻塞]

JavaScript的事件循环模型与许多其他语言不同的一个非常有趣的特性是，它永不阻塞。 处理 I/O 通常通过事件和回调来执行，所以当一个应用正等待一个 IndexedDB 查询返回或者一个 XHR 请求返回时，它仍然可以处理其它事情，比如用户输入。

由于历史原因有一些例外，如 `alert` 或者同步 XHR，但应该尽量避免使用它们。注意，例外的例外也是存在的（但通常是实现错误而非其它原因）。

简述 Flex 布局的原理和使用场景 中等

简述常见异步编程方案 (promise, generator, async) 的原理 中等

如何使用 flex 实现两栏布局?

简述 ES6 的新特性 简单

简述 watch 和 computed 的区别 简单

简述浏览器的缓存机制 中等

简述 Javascript 中的防抖与节流的原理并尝试实现 中等

Vuex 有哪些常用属性?

简述 CSS 预编译的方式

什么是 JavaScript 的变量提升？有什么作用?

简述 Vue 的生命周期 中等

简述强缓存与协商缓存的区别和使用场景 中等

简述 Javascript 的数据类型

简述 BFC 的原理及其使用场景 中等

history 和 hash 的区别是什么?

系统设计

电商系统中，如何实现秒杀功能？如何解决商品的超卖问题？ 困难

如何解决缓存与数据库不一致的问题?

简述中间件削峰和限流的使用场景

项目上有什么技术难点?

非技术

最近在看什么书吗，有没有接触过什么新技术?

你用过快手吗？说说对对快手的看法，使用上有哪些优缺点

实习的内容是什么？最大收获是什么？

未来的职业规划是什么？

你的性格和技能上有什么缺点

简单描述一下自己是怎么样的人？

为什么想要来快手？

学习中遇到的最大的困难是什么？

最有成就感的项目是什么？

能接受多大强度的加班？