美团字节

美团

算法部分

146. LRU 缓存机制

912. 排序数组

1114. 按序打印

二叉树的前序遍历 (Leetcode)

旋转图像 (Leetcode)

剑指 Offer 10- II. 青蛙跳台阶问题

19. 删除链表的倒数第 N 个结点

155. 最小栈

215. 数组中的第K个最大元素

按奇偶排序数组 II (Leetcode)

多数元素 (Leetcode)

232. 用栈实现队列

合并两个有序数组 (Leetcode)

300. 最长递增子序列

33. 搜索旋转排序数组

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?

1143. 最长公共子序列

两数相加 (Leetcode)

验证IP地址 (Leetcode)

组件之间通信方式有哪些?

5. 最长回文子串

112. 路径总和

二叉树的中序遍历 (Leetcode)

19. 删除链表的倒数第 N 个结点

移除元素 (Leetcode)

Java部分

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

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?

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?

Java 的线程有哪些状态，转换关系是怎么样的?

JVM 内存是如何对应到操作系统内存的?

Java 怎么防止内存溢出

简述 BIO, NIO, AIO 的区别

JMM 中内存模型是怎样的？什么是指令序列重排序?

Java 类的加载流程是怎样的？什么是双亲委派机制?

实现单例模式

volatile 关键字解决了什么问题，它的实现原理是什么?

简述常见的工厂模式以及单例模式的使用场景

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

hashcode 和 equals 方法的联系

什么是重写和重载?

简述 CAS 原理，什么是 ABA 问题，怎么解决?

线程池是如何实现的？简述线程池的任务策略

HashMap 1.7 / 1.8 的实现区别

什么情况下会发生死锁，如何解决死锁?

简述装饰者模式以及适配器模式

简述 Java 中 final 关键字的作用

== 和 equals() 的区别?

简述 Netty 线程模型，Netty 为什么如此高效?

ThreadLocal 实现原理是什么?

简述 HashSet 实现原理

简述 GC 引用链，G1收集器原理

简述动态代理与静态代理

网络协议部分
- HTTP 与 HTTPS 有哪些区别?
- TCP 怎么保证可靠传输?
- TCP 与 UDP 在网络协议中的哪一层，他们之间有什么区别?
- 从输入 URL 到展现页面的全过程
- 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?
- TCP长连接和短连接有那么不同的使用场景?
- HTTP 中 GET 和 POST 区别
- 简述对称与非对称加密的概念
- 简述常见的 HTTP 状态码的含义（301，304，401，403）
- Cookie 和 Session 的关系和区别是什么?
- 简述 DDOS 攻击原理，如何防范它?
- 简述 HTTP 1.0，1.1，2.0 的主要区别
- 什么是 ARP 协议？简述其使用场景
- 简述在四层和七层网络协议中负载均衡的原理
- 什么是 TCP 粘包和拆包?
- RestFul 是什么？RestFul 请求的 URL 有什么特点?
- 简述 HTTP 报文头部的组成结构

数据库部分
- 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?
- 简述 MySQL 常见索引类型，介绍一下覆盖索引
- 简述乐观锁以及悲观锁的区别以及使用场景
- Redis 如何实现分布式锁?
- 简述 Redis 持久化中 RDB 以及 AOF 方案的优缺点
- 简述事务的四大特性
- 简述 Redis 中常见类型的底层数据结构
- 数据库的事务隔离级别有哪些？各有哪些优缺点?
- MySQL 有哪些常见的存储引擎？它们的区别是什么?
- 为什么 Redis 在单线程下能如此快?
- 简述脏读和幻读的发生场景，InnoDB 是如何解决幻读的?
- 简述 Redis 的线程模型以及底层架构设计
- 聚簇索引和非聚簇索引有什么区别?
- 简述 Redis 如何处理热点 key 访问
- 简述 Redis 的过期机制和内存淘汰策略
- 什么是数据库事务，MySQL 为什么会使用 InnoDB 作为默认选项
- 简述 Redis 的哨兵机制
- 简述 Redis 中如何防止缓存雪崩和缓存击穿
- 简述 Redis 中跳表的应用以及优缺点
- 简述什么是最左匹配原则
- 简述数据库中什么情况下进行分库，什么情况下进行分表?
- 简述 MySQL MVCC 的实现原理
- B+ 树叶子节点存储的是什么数据
- B+ 树中叶子节点存储的是什么数据
- MySQL中 InnoDB 和 MyISAM 的区别是什么?
- Redis 有几种数据结构？Zset 是如何实现的?
- 简述 Redis 集群配置以及基础原理
- 数据库如何设计索引，如何优化查询?
- 什么是 SQL 注入攻击？如何防止这类攻击?
- 简述数据库中的 ACID 分别是什么?

操作系统模块
- 进程和线程之间有什么区别?
- 进程间有哪些通信方式?
- 多线程和多进程的区别是什么?
- 为什么进程切换慢，线程切换快?
- 简述创建进程的流程
- 简述 Linux 虚拟内存的页面置换算法

前端部分
- 简述 diff 算法的实现机制和使用场景
- 什么是闭包，什么是立即执行函数，它的作用是什么？简单说一下闭包的使用场景

- Javascript 中 == 与 === 的区别是什么?
- 简述什么是 XSS 攻击以及 CSRF 攻击?
- 简述 Vue 的生命周期
- 手写题库 https://github.com/Mayandev/fe-interview-handwrite
- 简述项目打包和发布的流程
- 简述 React 的生命周期以及通信方式
- sessionStorage 和 localStorage 有什么区别?
- 简述浏览器的垃圾回收机制
- 简述浏览器的渲染过程，重绘和重排在渲染过程中的哪一部分?
- 简述 Javascript 中的防抖与节流的原理并尝试实现
- 移动端适配有哪些方案?
- MVC 模型和 MVVM 模型的区别
- 简述 React 中的 Effect Hook 机制
- 简述 React setstate 原理
- 简述 CSS 盒模型
- 简述 weakMap 与 Map 的区别
- 简述强缓存与协商缓存的区别和使用场景
- 简述 typeof 和 instanceof 的原理
- 简述 JavaScript 事件循环机制
- promise 有哪些状态？简述 promise.all 的实现原理

系统设计部分

- 电商系统中，如何实现秒杀功能？如何解决商品的超卖问题?
- 简述 CAP 理论
- 如何实现唯一的分布式 ID
- 简述中间件削峰和限流的使用场景
- 简述 Kafka 的基本架构，如何用 Kafka 保证消息的有序性?
- 简述 Dubbo 服务注册与发现的过程
- 简述一致性哈希算法的实现方式及原理

非技术模块

- 下一份工作希望学习到什么?
- 项目中最难的地方是哪里？你学习到了什么?
- 最近在看什么书吗，有没有接触过什么新技术?
- 团队合作沟通中遇到过什么问题?
- 未来的职业规划是什么?
- 成长过程中影响你最深的事件和人
- 你用过美团吗？说说对对美团的看法，使用上有哪些优缺点
- 团队中对技术选型有冲突的话怎么解决?

字节

算法部分

- 10亿个数中如何高效地找到最大的一个数以及最大的第 K 个数
- 25. K 个一组翻转链表
- 103. 二叉树的锯齿形层序遍历
- 912. 排序数组
- 445. 两数相加 II
- 33. 搜索旋转排序数组
- 142. 环形链表 II
- 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
- 153. 寻找旋转排序数组中的最小值
- 方法一：二分查找
- 81. 搜索旋转排序数组 II
- 236. 二叉树的最近公共祖先
- 128. 最长连续序列
- 53. 最大子序和
- 给定一个 foo 函数，60%的概率返回0，40%的概率返回1，如何利用 foo 函数实现一个 50% 返回 0 的函数?
- 64 匹马，8 个赛道，找出前 4 匹马最少需要比几次
- 112. 路径总和
- 300. 最长递增子序列
- 300. 最长递增子序列

141. 环形链表
300. 最长递增子序列

美团字节

美团

算法部分

146. LRU 缓存机制

相信如果有认真看过 ***LinkedHashMap*** 源码的小伙伴，一定会很快的跟官方题解写的一模一样！

简单介绍LinkedHashMap（跟题目有关的知识点）

HashMap 大家都清楚，底层是 数组 + 红黑树 + 链表 （不清楚也没有关系），同时其是无序的，而 LinkedHashMap 刚好就比 HashMap 多这一个功能，就是其提供 **有序**，并且，LinkedHashMap的有序可以按两种顺序排列，一种是按照插入的顺序，一种是按照**读取**的顺序（这个题目的示例就是告诉我们要按照读取的顺序进行排序），而其内部是靠 **建立一个双向链表** 来维护这个顺序的，在每次插入、删除后，都会调用一个函数来进行 **双向链表的维护** ，准确的来说，是有三个函数来做这件事，这三个函数都统称为 **回调函数** ，这三个函数分别是：

- **void afterNodeAccess(Node<K,V> p) { }**

  其作用就是在访问元素之后，将该元素放到双向链表的尾巴处(所以这个函数只有在按照读取的顺序的时候才会执行)，之所以提这个，是建议大家去看看，如何优美的实现在双向链表中将指定元素放入链尾！
- **void afterNodeRemoval(Node<K,V> p) { }**

  其作用就是在删除元素之后，将元素从双向链表中删除，还是非常建议大家去看看这个函数的，很优美的方式在双向链表中删除节点！
- **void afterNodeInsertion(boolean evict) { }**

  这个才是我们题目中会用到的，在插入新元素之后，需要回调函数判断是否需要移除一直不用的某些元素！

其次，我再介绍一下 LinkedHashMap 的构造函数！

其主要是两个构造方法，一个是继承 HashMap ，一个是可以选择 accessOrder 的值(默认 false，代表按照插入顺序排序)来确定是按插入顺序还是读取顺序排序。

- Java

```
/**
 * //调用父类HashMap的构造方法。
 * Constructs an empty insertion-ordered <tt>LinkedHashMap</tt> instance
 * with the default initial capacity (16) and load factor (0.75).
 */
public LinkedHashMap() {
    super();
    accessOrder = false;
}
// 这里的 accessOrder 默认是为false，如果要按读取顺序排序需要将其设为 true
// initialCapacity 代表 map 的 容量，loadFactor 代表加载因子 (默认即可)
public LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder) {
    super(initialCapacity, loadFactor);
    this.accessOrder = accessOrder;
}
```

思路 & 代码

下面是我自己在分析 LinkedHashMap 源码时做的一些笔记，应该是比较清楚的，主体意思就是我们要继承 LinkedHashMap，然后复写 removeEldestEntry()函数，就能拥有我们自己的缓存策略！

- Java

```
// 在插入一个新元素之后，如果是按插入顺序排序，即调用newNode()中的linkNodeLast()完成
// 如果是按照读取顺序排序，即调用afterNodeAccess()完成
// 那么这个方法是干嘛的呢，这个就是著名的 LRU 算法啦
// 在插入完成之后，需要回调函数判断是否需要移除某些元素！
// LinkedHashMap 函数部分源码
```

/**

- 插入新节点才会触发该方法，因为只有插入新节点才需要内存
- 根据 HashMap 的 putVal 方法, evict 一直是 true
- removeEldestEntry 方法表示移除规则, 在 LinkedHashMap 里一直返回 false
- 所以在 LinkedHashMap 里这个方法相当于什么都不做
  */
  void afterNodeInsertion(boolean evict) { // possibly remove eldest
  LinkedHashMap.Entry<K,V> first;
  // 根据条件判断是否移除最近最少被访问的节点
  if (evict && (first = head) != null && removeEldestEntry(first)) {
  K key = first.key;
  removeNode(hash(key), key, null, false, true);
  }
  }

```
    // 移除最近最少被访问条件之一，通过覆盖此方法可实现不同策略的缓存
    // LinkedHashMap是默认返回false的，我们可以继承LinkedHashMap然后复写该方法即可
    // 例如 LeetCode 第 146 题就是采用该种方法，直接 return size() > capacity;
    protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
       return false;
    }
```

通过上述代码，我们就已经知道了只要复写 removeEldestEntry() 即可，而条件就是 map 的大小不超过 给定的容量，超过了就得使用 LRU 了！然后根据题目给定的语句构造和调用：

- Java

```
/**
 * LRUCache 对象会以如下语句构造和调用:
 * LRUCache obj = new LRUCache(capacity);
 * int param_1 = obj.get(key);
 * obj.put(key,value);
 */
```

很明显我们只需要直接继承父类的put函数即可，因为题目没有特殊要求，故可以不写！至于 get() 函数，题目是有要求的！

> 获取数据 get(key) - 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1。

所以我们可以调用 LinkedHashMap 中的 getOrDefault()，完美符合这个要求，即当key不存在时会返回默认值 -1。

至此，我们就基本完成了本题的要求，只要写一个构造函数即可，答案的 `super(capacity, 0.75F, true);`，没看过源码的小伙伴可能不太清楚这个构造函数，这就是我上文讲的 LinkedHashMap 中的常用的第二个构造方法，具体大家可以看我上面代码的注释！

至此，大功告成！

- Java

```
class LRUCache extends LinkedHashMap<Integer, Integer>{
    private int capacity;
```

```
<span class="hljs-function"><span class="hljs-keyword">public</span> <span class="hljs-
title">LRUCache</span><span class="hljs-params">(<span class="hljs-keyword">int</span>
capacity)</span> </span>{
    <span class="hljs-keyword">super</span>(capacity, <span class="hljs-
number">0.75F</span>, <span class="hljs-keyword">true</span>);
```

```
    <span class="hljs-keyword">this</span>.capacity = capacity;
}

<span class="hljs-function"><span class="hljs-keyword">public</span> <span class="hljs-
keyword">int</span> <span class="hljs-title">get</span><span class="hljs-params">(<span
class="hljs-keyword">int</span> key)</span> </span>{
    <span class="hljs-keyword">return</span> <span class="hljs-
keyword">super</span>.getOrDefault(key, -<span class="hljs-number">1</span>);
}

<span class="hljs-comment">// 这个可不写</span>
<span class="hljs-function"><span class="hljs-keyword">public</span> <span class="hljs-
keyword">void</span> <span class="hljs-title">put</span><span class="hljs-params">(<span
class="hljs-keyword">int</span> key, <span class="hljs-keyword">int</span> value)</span>
</span>{
    <span class="hljs-keyword">super</span>.put(key, value);
}

<span class="hljs-meta">@Override</span>
<span class="hljs-function"><span class="hljs-keyword">protected</span> <span class="hljs-
keyword">boolean</span> <span class="hljs-title">removeEldestEntry</span><span class="hljs-
params">(Map.Entry&amp;lt;Integer, Integer&amp;gt; eldest)</span> </span>{
    <span class="hljs-keyword">return</span> size() &amp;gt; capacity;
}
```

}

最后，附上我最开始讲的那两个函数的源码以及部分自己的解析

- Java
- Java

```
//标准的如何在双向链表中将指定元素放入队尾
// LinkedHashMap 中覆写
//访问元素之后的回调方法
```

/**

- 1. 使用 get 方法会访问到节点, 从而触发调用这个方法
- 2. 使用 put 方法插入节点, 如果 key 存在, 也算要访问节点, 从而触发该方法
- 3. 只有 accessOrder 是 true 才会调用该方法
- 4. 这个方法会把访问到的最后节点重新插入到双向链表结尾
  */

```
void afterNodeAccess(Node<K,V> e) { // move node to last
  // 用 last 表示插入 e 前的尾节点
  // 插入 e 后 e 是尾节点, 所以也是表示 e 的前一个节点
  LinkedHashMap.Entry<K,V> last;
```

```
//如果是访问序，且当前节点并不是尾节点
//将该节点置为双向链表的尾部
if (accessOrder && (last = tail) != e) {
// p: 当前节点
// b: 前一个节点
// a: 后一个节点
// 结构为: b <=> p <=> a
LinkedHashMap.Entry<K,V> p =
   (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;
// 结构变成: b <=> p <- a
p.after = null;
```

```
          // 如果当前节点 p 本身是头节点，那么头结点要改成a
          if (b == null)
              head = a;
          // 如果 p 不是头尾节点，把前后节点连接，变成: b -&gt;a
          else
              b.after = a;

          // a 非空，和 b 连接，变成: b &lt;- a
          if (a != null)
              a.before = b;
          // 如果 a 为空，说明 p 是尾节点，b 就是它的前一个节点，符合 last 的定义
          // 这个 else 没有意义，因为最开头if已经确保了p不是尾结点了，自然after不会是null
          else
              last = b;

          // 如果这是空链表，p 改成头结点
          if (last == null)
              head = p;
          // 否则把 p 插入到链表尾部
          else {
              p.before = last;
              last.after = p;
          }
          tail = p;
          ++modCount;
          }

        }
        </code></pre>
        <input type="radio" style="display: none;" class="css-5pejp0" id="group-5-6" name="group-5"><pre><button class="md-btn-copy" title="复制代码"><i></i></button><code><span class="hljs-function"><span class="hljs-keyword">void</span> <span class="hljs-title">afterNodeRemoval</span><span class="hljs-params">(Node&lt;K,V&gt; e)</span> </span>{ <span class="hljs-comment">// 优美的一笔，学习一波如何在双向链表中删除节点</span>
          LinkedHashMap.Entry&lt;K,V&gt; p =
          (LinkedHashMap.Entry&lt;K,V&gt;)e, b = p.before, a = p.after;
          <span class="hljs-comment">// 将 p 节点的前驱后后继引用置空</span>
          p.before = p.after = <span class="hljs-keyword">null</span>;
```

```
            <span class="hljs-comment">// b 为 null，表明 p 是头节点</span>
            <span class="hljs-keyword">if</span> (b == <span class="hljs-
keyword">null</span>)
            head = a;
            <span class="hljs-keyword">else</span>
            b.after = a;
            <span class="hljs-comment">// a 为 null，表明 p 是尾节点</span>
            <span class="hljs-keyword">if</span> (a == <span class="hljs-
keyword">null</span>)
            tail = b;
            <span class="hljs-keyword">else</span>
            a.before = b;
        }
        </code></pre>
        </div><p>希望对大家有所帮助~ 也希望武汉疫情赶紧结束，大家 2020 新年快乐，刷题愉快！！！
</p>
        </div>
```

912. 排序数组

1114. 按序打印

并发问题

> 并发问题来自并发计算的场景，该场景下，程序在多线程（或多进程）中 *同时* 执行。

同时进行并不是完全指进程或线程在不同的物理 CPU 上独立运行，更多情况下，是在一个物理 CPU 上交替执行多个线程或进程。*并发既可在线程中，也可在进程中。*

并发主要为多任务情况设计。但如果应用不当，可能会引发一些漏洞。按照情况不同，可以分为三种：

- **竞态条件**：由于多进程之间的竞争执行，导致程序未按照期望的顺序输出。
- **死锁**：并发程序等待一些必要资源，导致没有程序可以执行。
- **资源不足**：进程被永久剥夺了运行所需的资源。

此题中存在竞态条件。下面展示一个竞态条件的例子。

假设有一个方法 `withdraw(amount)`，如果请求量小于当前余额，则从当前余额中减去请求量，然后返回余额。方法定义如下：

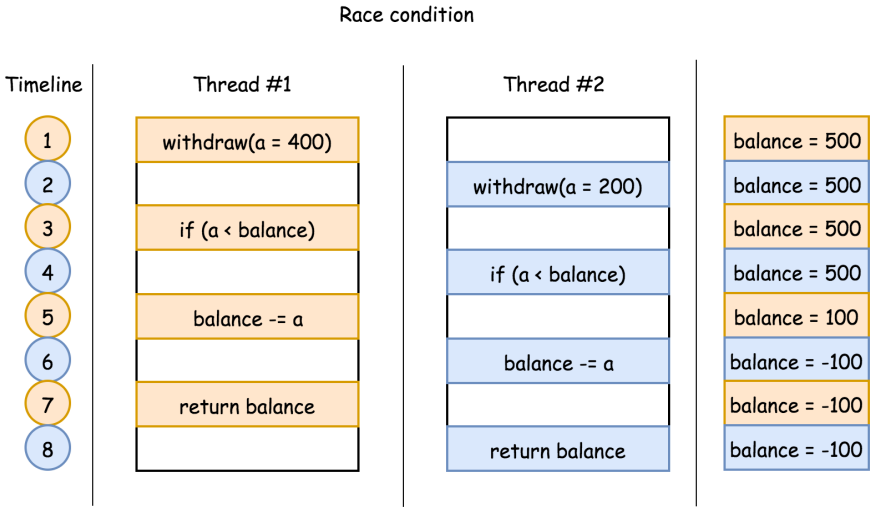
- Python
- Java

```
balance = 500
def withdraw(amount):
    if (amount < balance):
        balance -= amount
    return balance
```

```
int balance = 500;
int withdraw(int amount) {
  if (amount < balance) {
    balance -= amount;
  }
  return balance;
}
```

我们 *期望* 该方法执行后余额永远不会为负。

但是有可能出现竞态条件，使得余额变为负数。假设两个线程同时使用不同的参数执行该方法。例如：线程 1 执行 `withdraw(amount=400)`，线程 2 执行 `withdraw(amount=200)`。这两个线程的执行顺序如下图所示。在每个时刻只执行一条语句。



上述流程执行结束后，余额变成负数，这并不是期望的输出。

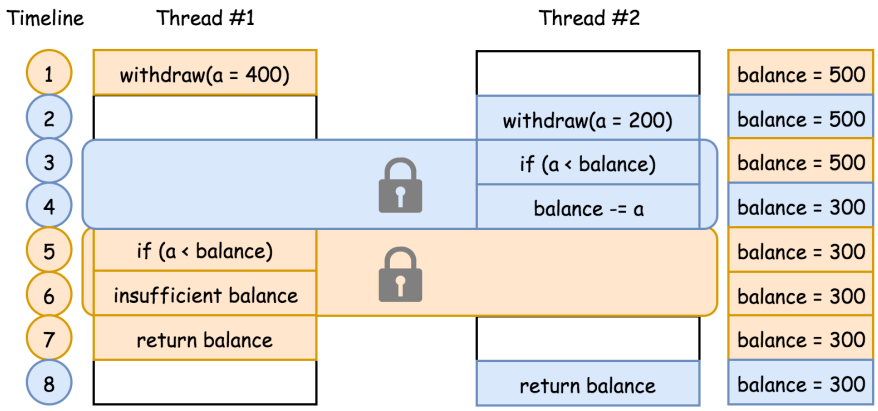
无竞争并发

并发问题有一个共同特征：多个线程/进程之间共享一些资源（例如：余额）。由于无法消除资源共享的约束，防止并发问题就变成了 *资源共享的协调* 问题。

根据这个思路，如果可以确保程序中 **关键部分代码的独占性**（例如：检查和减少余额），就可以防止程序进入不一致的状态。

> 竞争条件的解决方案为：需要某些关键部分代码具有排他性，即在给定的时间内，只有一个线程可以进入关键部分代码。

可以将这种机制看做限制关键部分代码访问的锁。在前面示例的关键部分代码加锁，即检查余额和减少余额的语句。然后重新运行两个线程，会有下图的执行顺序：



在该机制下，一旦一个线程进入关键部分，它就可以阻止其他线程进入该关键部分。例如，在时间点 3，`线程 2` 进入关键部分，那么在时间点 4，如果没有锁保护，`线程 1` 就可能进入关键部分。最后两个线程同时运行，保证系统的一致性，并确保余额正确。

如果该线程未被授权进入关键代码，可以认为该线程被阻塞或进入睡眠状态。例如，`线程 1` 在时间点 4 被阻塞，之后关键部分被释放，可以通知其他等待线程。`线程 2` 在时间点 5 释放了关键部分，就可以通知 `线程 1` 进入。

> 这种机制还具有唤醒其他等待线程的功能。

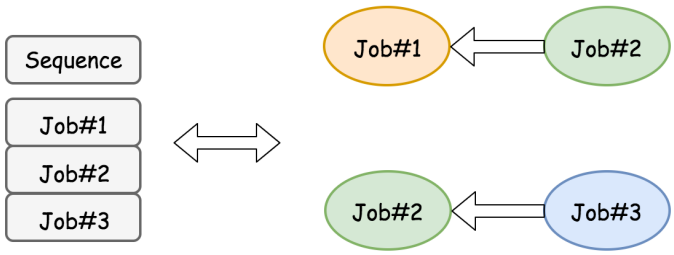
总之，为了防止出现并发竞争状态，需要一种具有两种功能的机制：1）关键部分的访问控制；2）通知阻塞线程。

方法一：使用 synchronization

思路

题目要求按顺序依次执行三个方法，且每个方法都在单独的线程中运行。为了保证线程的执行顺序，可以在方法之间创建一些依赖关系，即第二个方法必须在第一个方法之后执行，第三个方法必须在第二个方法之后执行。

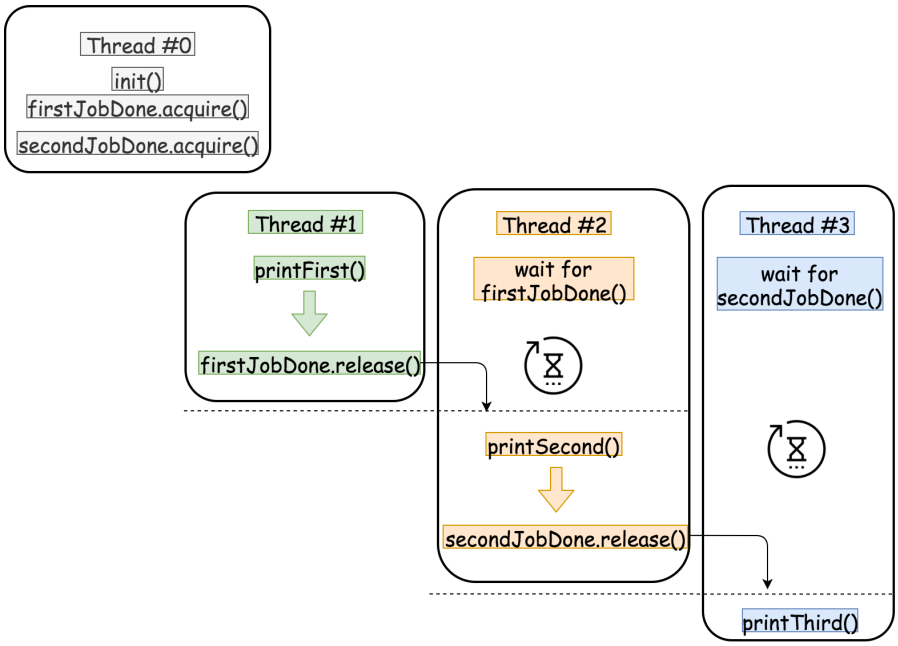
> 方法对之间的依赖关系形成了所有方法的特定的执行顺序。例如 `A < B`，`B < C`，则所有方法的执行顺序为 `A < B < C`。



依赖关系可以通过并发机制实现。使用一个共享变量 `firstJobDone` 协调第一个方法与第二个方法的执行顺序，使用另一个共享变量 `secondJobDone` 协调第二个方法与第三个方法的执行顺序。

算法

- 首先初始化共享变量 `firstJobDone` 和 `secondJobDone`，初始值表示所有方法未执行。
- 方法 `first()` 没有依赖关系，可以直接执行。在方法最后更新变量 `firstJobDone` 表示该方法执行完成。
- 方法 `second()` 中，检查 `firstJobDone` 的状态。如果未更新则进入等待状态，否则执行方法 `second()`。在方法末尾，更新变量 `secondJobDone` 表示方法 `second()` 执行完成。
- 方法 `third()` 中，检查 `secondJobDone` 的状态。与方法 `second()` 类似，执行 `third()` 之前，需要先等待 `secondJobDone` 的状态。



实现

上述算法的实现在很大程度上取决于选择的编程语言。尽管在 Java，C++ 和 Python 中都存在互斥与信号量，但不同语言对并发机制有不同实现。

- Python
- Cpp
- Java

`from threading import Lock`

class Foo:
    def **init**(self):
        self.firstJobDone = Lock()
        self.secondJobDone = Lock()
        self.firstJobDone.acquire()
        self.secondJobDone.acquire()

```
<span class="hljs-function"><span class="hljs-keyword">def</span> <span class="hljs-title">first</span><span class="hljs-params">(self, printFirst: <span class="hljs-string">'Callable[[], None]'</span>)</span> -&gt; <span class="hljs-keyword">None</span>:</span>
    <span class="hljs-comment"># printFirst() outputs "first".</span>
    printFirst()
    <span class="hljs-comment"># Notify the thread that is waiting for the first job to be done.</span>
    self.firstJobDone.release()
```

```
<span class="hljs-function"><span class="hljs-keyword">def</span> <span class="hljs-
title">second</span><span class="hljs-params">(self, printSecond: <span class="hljs-
string">'Callable[[], None]'</span>)</span> -&gt; <span class="hljs-keyword">None</span>:
</span>
    <span class="hljs-comment"># Wait for the first job to be done</span>
    <span class="hljs-keyword">with</span> self.firstJobDone:
        <span class="hljs-comment"># printSecond() outputs "second".</span>
        printSecond()
        <span class="hljs-comment"># Notify the thread that is waiting for the second job to
be done.</span>
        self.secondJobDone.release()

<span class="hljs-function"><span class="hljs-keyword">def</span> <span class="hljs-
title">third</span><span class="hljs-params">(self, printThird: <span class="hljs-
string">'Callable[[], None]'</span>)</span> -&gt; <span class="hljs-keyword">None</span>:
</span>

    <span class="hljs-comment"># Wait for the second job to be done.</span>
    <span class="hljs-keyword">with</span> self.secondJobDone:
        <span class="hljs-comment"># printThird() outputs "third".</span>
        printThird()
```

○

```
#include <semaphore.h>

class Foo {

protected:
    sem_t firstJobDone;
    sem_t secondJobDone;

public:

Foo() {
    sem_init(&amp;firstJobDone, <span class="hljs-number">0</span>, <span class="hljs-
number">0</span>);
    sem_init(&amp;secondJobDone, <span class="hljs-number">0</span>, <span class="hljs-
number">0</span>);
}

<span class="hljs-function"><span class="hljs-keyword">void</span> <span class="hljs-
title">first</span><span class="hljs-params">(function&lt;<span class="hljs-keyword">void</span>()&gt;
printFirst)</span> </span>{
    <span class="hljs-comment">// printFirst() outputs "first".</span>
    printFirst();
    sem_post(&amp;firstJobDone);
}

<span class="hljs-function"><span class="hljs-keyword">void</span> <span class="hljs-
title">second</span><span class="hljs-params">(function&lt;<span class="hljs-keyword">void</span>()&gt;
printSecond)</span> </span>{
    sem_wait(&amp;firstJobDone);
    <span class="hljs-comment">// printSecond() outputs "second".</span>
    printSecond();
    sem_post(&amp;secondJobDone);

}

<span class="hljs-function"><span class="hljs-keyword">void</span> <span class="hljs-
title">third</span><span class="hljs-params">(function&lt;<span class="hljs-keyword">void</span>()&gt;
printThird)</span> </span>{
    sem_wait(&amp;secondJobDone);
    <span class="hljs-comment">// printThird() outputs "third".</span>
    printThird();
}


};
```

◯

```
class Foo {

  private AtomicInteger firstJobDone = new AtomicInteger(0);
  private AtomicInteger secondJobDone = new AtomicInteger(0);

  public Foo() {}

  public void first(Runnable printFirst) throws InterruptedException {
    // printFirst.run() outputs "first".
    printFirst.run();
    // mark the first job as done, by increasing its count.
    firstJobDone.incrementAndGet();
  }

  public void second(Runnable printSecond) throws InterruptedException {
    while (firstJobDone.get() != 1) {
      // waiting for the first job to be done.
    }
    // printSecond.run() outputs "second".
    printSecond.run();
    // mark the second as done, by increasing its count.
    secondJobDone.incrementAndGet();
  }

  public void third(Runnable printThird) throws InterruptedException {
    while (secondJobDone.get() != 1) {
      // waiting for the second job to be done.
    }
    // printThird.run() outputs "third".
    printThird.run();
  }
}
```

二叉树的前序遍历 (Leetcode)

旋转图像 (Leetcode)

剑指 Offer 10- II. 青蛙跳台阶问题

19. 删除链表的倒数第 N 个结点

解题思路
标签：链表
整体思路是让前面的指针先移动n步，之后前后指针共同移动直到前面的指针到尾部为止
首先设立预先指针 pre，预先指针是一个小技巧，在第2题中进行了讲解
设预先指针 pre 的下一个节点指向 head，设前指针为 start，后指针为 end，二者都等于 pre
start 先向前移动n步
之后 start 和 end 共同向前移动，此时二者的距离为 n，当 start 到尾部时，end 的位置恰好为倒数第 n 个节点
因为要删除该节点，所以要移动到该节点的前一个才能删除，所以循环结束条件为 start.next != null
删除后返回 pre.next，为什么不直接返回 head 呢，因为 head 有可能是被删掉的点
时间复杂度：O(n)O(n)
代码
Java

```
/**
```

- Definition for singly-linked list.
- public class ListNode {

- int val;
- ListNode next;
- ListNode(int x) { val = x; }
- }

```
*/
class Solution {
public ListNode removeNthFromEnd(ListNode head, int n) {
    ListNode pre = new ListNode(0);
    pre.next = head;
    ListNode start = pre, end = pre;
    while(n != 0) {
        start = start.next;
        n--;
    }
    while(start.next != null) {
        start = start.next;
        end = end.next;
    }
    end.next = end.next.next;
    return pre.next;
}
}
```

155. 最小栈

解题思路

题目要求在常数时间内获得栈中的最小值，因此不能在 `getMin()` 的时候再去计算最小值，最好应该在 `push` 或者 `pop` 的时候就已经计算好了当前栈中的最小值。

前排的众多题解中，基本都讲了「辅助栈」的概念，这是一种常见的思路，但是有没有更容易懂的方法呢？

可以用一个栈，这个栈同时保存的是每个数字 `x` 进栈的时候的**值 与 插入该值后的栈内最小值**。即每次新元素 `x` 入栈的时候保存一个元组：**（当前值 x，栈内最小值）**。

这个元组是一个整体，同时进栈和出栈。即栈顶同时有值和栈内最小值，`top()` 函数是获取栈顶的**当前值**，即栈顶元组的第一个值；`getMin()` 函数是获取**栈内最小值**，即栈顶元组的第二个值；`pop()` 函数时删除栈顶的元组。

每次新元素入栈时，要求新的栈内最小值：比较当前新插入元素 x 和 当前栈内最小值（即栈顶元组的第二个值）的大小。

1. 新元素入栈：当栈为空，保存元组 `(x, x)`；当栈不空，保存元组 `(x, min(此前栈内最小值, x))`
2. 出栈：删除栈顶的元组。

代码

- Python
- C++

```
class MinStack(object):
```

```
    def __init__(self):
        """
        initialize your data structure here.
        """
        self.stack = []

    def push(self, x):
        """
        :type x: int
        :rtype: void
        """
        if not self.stack:
            self.stack.append((x, x))
        else:
            self.stack.append((x, min(x, self.stack[-1][1])))

    def pop(self):
        """
        :rtype: void
        """
        self.stack.pop()

    def top(self):
        """
        :rtype: int
        """
        return self.stack[-1][0]

    def getMin(self):
        """
        :rtype: int
        """
        return self.stack[-1][1]

# Your MinStack object will be instantiated and called as such:
# obj = MinStack()
# obj.push(x)
# obj.pop()
# param_3 = obj.top()
# param_4 = obj.getMin()
```

○

```
class MinStack {
public:
    /** initialize your data structure here. */
    MinStack() {
    }
```

```
<span class="hljs-function"><span class="hljs-keyword">void</span> <span class="hljs-title">push</span>
<span class="hljs-params">(<span class="hljs-keyword">int</span> x)</span> </span>{
    <span class="hljs-keyword">if</span> (st.size() == <span class="hljs-number">0</span>) {
        st.push({x, x});
    } <span class="hljs-keyword">else</span> {
        st.push({x, min(x, st.top().second)});
    }
}

<span class="hljs-function"><span class="hljs-keyword">void</span> <span class="hljs-title">pop</span>
<span class="hljs-params">()</span> </span>{
    st.pop();
}

<span class="hljs-function"><span class="hljs-keyword">int</span> <span class="hljs-title">top</span>
<span class="hljs-params">()</span> </span>{
    <span class="hljs-keyword">return</span> st.top().first;
}

<span class="hljs-function"><span class="hljs-keyword">int</span> <span class="hljs-
title">getMin</span><span class="hljs-params">()</span> </span>{
    <span class="hljs-keyword">return</span> st.top().second;
}
```

```
private:
    stack<pair<int, int>> st;
};

/**
```

- Your MinStack object will be instantiated and called as such:
- MinStack* obj = new MinStack();
- obj->push(x);
- obj->pop();
- int param_3 = obj->top();
- int param_4 = obj->getMin();

```
 */
```

215. 数组中的第K个最大元素

Partition函数

首先，先写partition模板```

```
def partition(nums, left, right):
    pivot = nums[left]#初始化一个待比较数据
    i,j = left, right
    while(i < j):
        while(i<j and nums[j]>=pivot): #从后往前查找，直到找到一个比pivot更小的数
            j-=1
        nums[i] = nums[j] #将更小的数放入左边
        while(i<j and nums[i]<=pivot): #从前往后找，直到找到一个比pivot更大的数
            i+=1
        nums[j] = nums[i] #将更大的数放入右边
    #循环结束，i与j相等
    nums[i] = pivot #待比较数据放入最终位置
    return i #返回待比较数据最终位置
```

快速排序

复习一下快速排序：

```
#快速排序
def quicksort(nums, left, right):
    if left < right:
        index = partition(nums, left, right)
        quicksort(nums, left, index-1)
        quicksort(nums, index+1, right)
```

arr = [1,3,2,2,0]
quicksort(arr, 0, len(arr)-1)
print(arr)

topk切分

将快速排序改成快速选择，即我们希望寻找到一个位置，这个位置左边是k个比这个位置上的数更小的数，右边是n-k个比该位置上的数大的数，我将它命名为topk_split，找到这个位置后停止迭代，完成了一次划分。

```
def topk_split(nums, k, left, right):
    #寻找到第k个数停止递归，使得nums数组中index左边是前k个小的数，index右边是后面n-k个大的数
    if (left<right):
        index = partition(nums, left, right)
        if index==k:
            return
        elif index < k:
            topk_split(nums, k, index+1, right)
        else:
            topk_split(nums, k, left, index-1)
```

接下来就依赖于上面这两个函数解决所有的topk问题

获得前k小的数

```
#获得前k小的数
def topk_smalls(nums, k):
    topk_split(nums, k, 0, len(nums)-1)
    return nums[:k]
```

```
arr = [1,3,2,3,0,-19]
k = 2
print(topk_smalls(arr, k))
print(arr)
```

获取第k小的数

```
#获得第k小的数
def topk_small(nums, k):
    topk_split(nums, k, 0, len(nums)-1)
    return nums[k-1] #右边是开区间，需要-1
```

```
arr = [1,3,2,3,0,-19]
k = 3
print(topk_small(arr, k))
print(arr)
```

获得前k大的数

```
#获得前k大的数
def topk_larges(nums, k):
    #parttion是按从小到大划分的，如果让index左边为前n-k个小的数，则index右边为前k个大的数
    topk_split(nums, len(nums)-k, 0, len(nums)-1) #把k换成len(nums)-k
    return nums[len(nums)-k:]
```

```
arr = [1,3,-2,3,0,-19]
k = 3
print(topk_larges(arr, k))
print(arr)
```

获得第k大的数

```
#获得第k大的数
def topk_large(nums, k):
    #parttion是按从小到大划分的，如果让index左边为前n-k个小的数，则index右边为前k个大的数
    topk_split(nums, len(nums)-k, 0, len(nums)-1) #把k换成len(nums)-k
    return nums[len(nums)-k]
```

```
arr = [1,3,-2,3,0,-19]
k = 2
print(topk_large(arr, k))
print(arr)
```

只排序前k个小的数

```
#只排序前k个小的数
#获得前k小的数O(n)，进行快排O(klogk)
def topk_sort_left(nums, k):
    topk_split(nums, k, 0, len(nums)-1)
    topk = nums[:k]
    quicksort(topk, 0, len(topk)-1)
    return topk+nums[k:] #只排序前k个数字
```

```
arr = [0,0,1,3,4,5,0,7,6,7]
k = 4
topk_sort_left(arr, k)
```

只排序后k个大的数

```
#只排序后k个大的数
#获得前n-k小的数O(n)，进行快排O(klogk)
def topk_sort_right(nums, k):
    topk_split(nums, len(nums)-k, 0, len(nums)-1)
    topk = nums[len(nums)-k:]
    quicksort(topk, 0, len(topk)-1)
    return nums[:len(nums)-k]+topk #只排序后k个数字
```

arr = [0,0,1,3,4,5,0,-7,6,7]
k = 4
print(topk_sort_right(arr, k))

按奇偶排序数组 II (Leetcode)

多数元素 (Leetcode)

232. 用栈实现队列

基本思路
无论「用栈实现队列」还是「用队列实现栈」，思路都是类似的。

都可以通过使用两个栈/队列来解决。

我们创建两个栈，分别为 out 和 in，用作处理「输出」和「输入」操作。

其实就是两个栈来回「倒腾」。

而对于「何时倒腾」决定了是 O(n) 解法 还是 均摊 O(1) 解法。

O(n) 解法
我们创建两个栈，分别为 out 和 in：

in 用作处理输入操作 push()，使用 in 时需确保 out 为空
out 用作处理输出操作 pop() 和 peek()，使用 out 时需确保 in 为空

```
class MyQueue {
   Deque out, in;
   public MyQueue() {
      in = new ArrayDeque<>();
      out = new ArrayDeque<>();
   }
```

```
public void push(int x) {
    while (!out.isEmpty()) in.addLast(out.pollLast());
    in.addLast(x);
}

public int pop() {
    while (!in.isEmpty()) out.addLast(in.pollLast());
    return out.pollLast();
}
```

```
public int peek() {
    while (!in.isEmpty()) out.addLast(in.pollLast());
    return out.peekLast();
}

public boolean empty() {
    return out.isEmpty() && in.isEmpty();
}
```

}
时间复杂度：O(n)O(n)
空间复杂度：O(n)O(n)
均摊 O(1) 解法
事实上，我们不需要在每次的「入栈」和「出栈」操作中都进行「倒腾」。

我们只需要保证，输入的元素总是跟在前面的输入元素的后面，而输出元素总是最早输入的那个元素即可。

可以通过调整「倒腾」的时机来确保满足上述要求，但又不需要发生在每一次操作中：

只有在「输出栈」为空的时候，才发生一次性的「倒腾」

class MyQueue {
  Deque out, in;
  public MyQueue() {
    in = new ArrayDeque<>();
    out = new ArrayDeque<>();
  }

```
public void push(int x) {
    in.addLast(x);
}

public int pop() {
    if (out.isEmpty()) {
        while (!in.isEmpty()) out.addLast(in.pollLast());
    }
    return out.pollLast();
}

public int peek() {
    if (out.isEmpty()) {
        while (!in.isEmpty()) out.addLast(in.pollLast());
    }
    return out.peekLast();
}

public boolean empty() {
    return out.isEmpty() && in.isEmpty();
}
```

}
时间复杂度：pop() 和 peek() 操作都是均摊 O(1)O(1)
空间复杂度：O(n)O(n)
关于「均摊复杂度」的说明
我们先用另外一个例子来理解「均摊复杂度」，大家都知道「哈希表」底层是通过数组实现的。

正常情况下，计算元素在哈希桶的位置，然后放入哈希桶，复杂度为 O(1)O(1)，假定是通过简单的“拉链法”搭配「头插法」方式来解决哈希冲突。

但当某次元素插入后，「哈希表」达到扩容阈值，则需要对底层所使用的数组进行扩容，这个复杂度是 O(n)O(n)

显然「扩容」操作不会发生在每一次的元素插入中，因此扩容的 O(n)O(n) 都会伴随着 n 次的 O(1)O(1)，也就是 O(n)O(n) 的复杂度会被均摊到每一次插入当中，因此哈希表插入仍然是 O(1)O(1) 的。

同理，我们的「倒腾」不是发生在每一次的「输出操作」中，而是集中发生在一次「输出栈为空」的时候，因此 pop 和 peek 都是均摊复杂度为 O(1)O(1) 的操作。

由于本题的调用次数只有 100 次，所以铁定是一个人均 100% 的算法（0 ms）

我们需要对操作进行复杂度分析进行判断，而不是看时间来判断自己是不是均摊 O(1) 哦 ~

合并两个有序数组 (Leetcode)

300. 最长递增子序列

动态规划
子序列的问题->动态规划。

使用数组 cell 保存每步子问题的最优解。
cell[i] 代表含第 i 个元素的最长上升子序列的长度。
求解 cell[i] 时，向前遍历找出比 i 元素小的元素 j，令 cell[i] 为 max（cell[i],cell[j]+1)。
详情可以参照官方题解。

Python

```
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        if nums == []:
            return 0
        cell = [1]
        for i in range(1,len(nums)):
            cell.append(1)
            for j in range(i):
                if(nums[j] < nums[i]):
                    cell[i] = max(cell[i], cell[j]+1)
        return max(cell)
```

复杂度分析：

时间复杂度：O(n2)O(n2)，双层遍历

空间复杂度：O(n)O(n)

动态规划+二分查找
很具小巧思。新建数组 cell，用于保存最长上升子序列。

对原序列进行遍历，将每位元素二分插入 cell 中。

如果 cell 中元素都比它小，将它插到最后
否则，用它覆盖掉比它大的元素中最小的那个。
总之，思想就是让 cell 中存储比较小的元素。这样，cell 未必是真实的最长上升子序列，但长度是对的。

Python

```
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        size = len(nums)
        if size<2:
```

```
        return size
```

```
    cell = [nums[0]]
    for num in nums[1:]:
        if num>cell[-1]:
            cell.append(num)
            continue

        l,r = 0,len(cell)-1
        while l<r:
            mid = l + (r - l) // 2
            if cell[mid]<num:
                l = mid + 1
            else:
                r = mid
        cell[l] = num
    return len(cell)
```

33. 搜索旋转排序数组

```
class Solution {
public:
  int search(vector& nums, int target) {
    int lo = 0, hi = nums.size() - 1;
    while (lo < hi) {
      int mid = (lo + hi) / 2;
      if ((nums[0] > target) ^ (nums[0] > nums[mid]) ^ (target > nums[mid]))
        lo = mid + 1;
      else
        hi = mid;
    }
    return lo == hi && nums[lo] == target ? lo : -1;
  }
};
```

以二分搜索为基本思路

简要来说：

nums[0] <= nums[mid]（0 - mid不包含旋转）且nums[0] <= target <= nums[mid] 时 high 向前规约；

nums[mid] < nums[0]（0 - mid包含旋转），target <= nums[mid] < nums[0] 时向前规约（target 在旋转位置到 mid 之间）

nums[mid] < nums[0]，nums[mid] < nums[0] <= target 时向前规约（target 在 0 到旋转位置之间）

其他情况向后规约

也就是说nums[mid] < nums[0]，nums[0] > target，target > nums[mid] 三项均为真或者只有一项为真时向后规约。

原文的分析是：

注意到原数组为有限制的有序数组（除了在某个点会突然下降外均为升序数组）

if nums[0] <= nums[l] 那么 nums[0] 到 nums[i] 为有序数组,那么当 nums[0] <= target <= nums[i] 时我们应该在 0-i0−i 范围内查找；

if nums[i] < nums[0] 那么在 0-i0−i 区间的某个点处发生了下降（旋转），那么 l+1l+1 到最后一个数字的区间为有序数组，并且所有的数字都是小于 nums[0] 且大于 nums[i]，当target不属于 nums[0] 到 nums[i] 时（target <= nums[i] < nums[0] or nums[i] < nums[0] <= target），我们应该在 0-i0−i 区间内查找。
上述三种情况可以总结如下：

```
nums[0] <= target <= nums[i]
           target <= nums[i] < nums[0]
                     nums[i] < nums[0] <= target
```

所以我们进行三项判断：

(nums[0] <= target)，(target <= nums[i])，(nums[i] < nums[0])，现在我们想知道这三项中有哪两项为真（明显这三项不可能均为真或均为假（因为这三项可能已经包含了所有情况））

所以我们现在只需要区别出这三项中有两项为真还是只有一项为真。

使用 “异或” 操作可以轻松的得到上述结果（两项为真时异或结果为假，一项为真时异或结果为真，可以画真值表进行验证）

之后我们通过二分查找不断做小 target 可能位于的区间直到 low==high，此时如果 nums[low]==target 则找到了，如果不等则说明该数组里没有此项。

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？

1143. 最长公共子序列

方法一：动态规划
最长公共子序列问题是典型的二维动态规划问题。

假设字符串 \textit{text}_1text
1

 和 \textit{text}_2text
2

 的长度分别为 mm 和 nn，创建 m+1m+1 行 n+1n+1 列的二维数组 \textit{dp}dp，其中 \textit{dp}[i][j]dp[i][j] 表示 \textit{text}_1[0:i]text
1

 [0:i] 和 \textit{text}_2[0:j]text
2

 [0:j] 的最长公共子序列的长度。

!--上述表示中，\textit{text}_1[0:i]text--

2

 [0:j] 表示 \textit{text}_2text
2

 的长度为 jj 的前缀。

考虑动态规划的边界情况：

当 i=0i=0 时，\textit{text}_1[0:i]text
1

[0:i] 为空，空字符串和任何字符串的最长公共子序列的长度都是 00，因此对任意 0 \le j \le n0≤j≤n，有 \textit{dp}[0][j]=0dp[0][j]=0；

当 j=0j=0 时，\textit{text}_2[0:j]text
2

[0:j] 为空，同理可得，对任意 0 \le i \le m0≤i≤m，有 \textit{dp}[i][0]=0dp[i][0]=0。

因此动态规划的边界情况是：当 i=0i=0 或 j=0j=0 时，\textit{dp}[i][j]=0dp[i][j]=0。

当 i>0i>0 且 j>0j>0 时，考虑 \textit{dp}[i][j]dp[i][j] 的计算：

当 \textit{text}*1[i-1]*=*textit{text}*2[j-1]text
1

[i−1]=text
2

[j−1] 时，将这两个相同的字符称为公共字符，考虑 \textit{text}_1[0:i-1]text
1

[0:i−1] 和 \textit{text}_2[0:j-1]text
2

[0:j−1] 的最长公共子序列，再增加一个字符（即公共字符）即可得到 \textit{text}_1[0:i]text
1

[0:i] 和 \textit{text}_2[0:j]text
2

[0:j] 的最长公共子序列，因此 \textit{dp}[i][j]=\textit{dp}[i-1][j-1]+1dp[i][j]=dp[i−1][j−1]+1。

当 \textit{text}*1[i-1] \ne *\textit{text}*2[j-1]text
1

[i−1]
\

=text
2

[j−1] 时，考虑以下两项：

\textit{text}_1[0:i-1]text
1

[0:i−1] 和 \textit{text}_2[0:j]text
2

[0:j] 的最长公共子序列；

\textit{text}_1[0:i]text
1

[0:i] 和 \textit{text}_2[0:j-1]text
2

[0:j−1] 的最长公共子序列。

要得到 \textit{text}_1[0:i]text
1

[0:i] 和 \textit{text}_2[0:j]text
2

[0:j] 的最长公共子序列，应取两项中的长度较大的一项，因此 \textit{dp}[i][j]=\max(\textit{dp}[i-1][j],\textit{dp}[i][j-1])dp[i][j]=max(dp[i−1][j],dp[i][j−1])。

由此可以得到如下状态转移方程：

\textit{dp}[i][j] = \begin{cases} \textit{dp}[i-1][j-1]+1, & \textit{text}*1[i-1]*=*textit{text}*2[j-1] \ \max(\textit{dp}[i-1][j],\textit{dp}[i][j-1]), & \textit{text}*1[i-1] \ne *textit{text}*2[j-1] \end{cases}
dp[i][j]={
dp[i−1][j−1]+1,
max(dp[i−1][j],dp[i][j−1]),

text
1

[i−1]=text
2

[j−1]
text
1

[i−1]
\

=text
2

[j−1]

最终计算得到 \textit{dp}[m][n]dp[m][n] 即为 \textit{text}_1text
1

和 \textit{text}_2text
2

的最长公共子序列的长度。

JavaJavaScriptGolangPython3C++C

```
class Solution {
    public int longestCommonSubsequence(String text1, String text2) {
        int m = text1.length(), n = text2.length();
        int[][] dp = new int[m + 1][n + 1];
        for (int i = 1; i <= m; i++) {
            char c1 = text1.charAt(i - 1);
            for (int j = 1; j <= n; j++) {
                char c2 = text2.charAt(j - 1);
                if (c1 == c2) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }
        return dp[m][n];
    }
```

}
复杂度分析

时间复杂度：O(mn)O(mn)，其中 mm 和 nn 分别是字符串 \textit{text}_1text
1

 和 \textit{text}_2text
2

 的长度。二维数组 \textit{dp}dp 有 m+1m+1 行和 n+1n+1 列，需要对 \textit{dp}dp 中的每个元素进行计算。

空间复杂度：O(mn)O(mn)，其中 mm 和 nn 分别是字符串 \textit{text}_1text
1

 和 \textit{text}_2text
2

 的长度。创建了 m+1m+1 行 n+1n+1 列的二维数组 \textit{dp}dp。

两数相加 (Leetcode)

验证IP地址 (Leetcode)

组件之间通信方式有哪些?

5. 最长回文子串

112. 路径总和

这个题要背下来！！

DFS

首先是 DFS 解法，该解法的想法是一直向下找到**叶子节点**，如果到**叶子节点**时`sum == 0`，说明找到了一条符合要求的路径。

我自己第一遍做的时候犯了一个错误，把递归函数写成了下面的解法：

```
def hasPathSum(self, root: TreeNode, sum: int) -> bool:
    if not root:
        return sum == 0
    return self.hasPathSum(root.left, sum - root.val) or self.hasPathSum(root.right, sum - root.val)
```

这种代码的错误在，**没有判断 root 是否为叶子节点。**比如 root 为空的话，题目的意思是要返回 False 的，而上面的代码会返回`sum == 0`。又比如，对于测试用例 树为`[1,2]`，`sum = 0`时，上面的结果也会返回为 True，因为对于上述代码，只要左右任意一个孩子的为空时`sum == 0`就返回 True 。

当题目中提到了**叶子节点**时，正确的做法一定要同时判断节点的**左右子树同时为空**才是叶子节点。

Python 代码如下：

- Python
- Java

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
```

```
class Solution(object):
    def hasPathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: bool
        """
        if not root: return False
        if not root.left and not root.right:
            return sum == root.val
        return self.hasPathSum(root.left, sum - root.val) or self.hasPathSum(root.right, sum - root.val)
```

○

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
        if(root == null){
            return false;
        }
        if(root.left == null && root.right == null){
            return root.val == sum;
        }
        return hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, sum - root.val);

    }
}
```

回溯

这里的回溯指 利用 DFS 找出从根节点到叶子节点的所有路径，只要有任意一条路径的 和 等于 sum，就返回 True。

下面的代码并非是严格意义上的回溯法，因为没有重复利用 path 变量。

Python 代码如下：

- Python

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
```

```
class Solution(object):
    def hasPathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: bool
        """
        if not root: return False
        res = []
        return self.dfs(root, sum, res, [root.val])
```

```
<span class="hljs-function"><span class="hljs-keyword">def</span> <span class="hljs-title">dfs</span><span class="hljs-params">(self, root, target, res, path)</span>:</span>
    <span class="hljs-keyword">if</span> <span class="hljs-keyword">not</span> root: <span class="hljs-keyword">return</span> <span class="hljs-literal">False</span>
    <span class="hljs-keyword">if</span> sum(path) == target <span class="hljs-keyword">and</span> <span class="hljs-keyword">not</span> root.left <span class="hljs-keyword">and</span> <span class="hljs-keyword">not</span> root.right:
        <span class="hljs-keyword">return</span> <span class="hljs-literal">True</span>
    left_flag, right_flag = <span class="hljs-literal">False</span>, <span class="hljs-literal">False</span>
    <span class="hljs-keyword">if</span> root.left:
        left_flag = self.dfs(root.left, target, res, path + [root.left.val])
    <span class="hljs-keyword">if</span> root.right:
        right_flag = self.dfs(root.right, target, res, path + [root.right.val])
    <span class="hljs-keyword">return</span> left_flag <span class="hljs-keyword">or</span> right_flag
```

BFS

BFS 使用 **队列** 保存遍历到每个节点时的**路径和**，如果该节点恰好是叶子节点，并且 路径和 正好等于 sum，说明找到了解。

Python 代码如下：

- Python

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

```
class Solution:
    def hasPathSum(self, root: TreeNode, sum: int) -> bool:
        if not root:
            return False
        que = collections.deque()
        que.append((root, root.val))
        while que:
            node, path = que.popleft()
            if not node.left and not node.right and path == sum:
                return True
            if node.left:
                que.append((node.left, path + node.left.val))
            if node.right:
                que.append((node.right, path + node.right.val))
        return False
```

栈

除了上面的 队列 解法以外，也可以使用 **栈**，同时保存节点和到这个节点的路径和。但是这个解法已经不是 BFS。因为会优先访问 后进来 的节点，导致会把根节点的右子树访问结束之后，才访问左子树。

可能会有朋友好奇很少见到这种写法，为什么代码可行？答案是：栈中同时保存了`(节点，路径和)`，也就是说只要能把所有的节点访问一遍，那么就一定能找到正确的结果。无论是用 队列 还是 栈，都是一种 树的遍历 方式，只不过访问顺序有所有不同罢了。

Python 代码如下：

- Python

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
class Solution(object):
    def hasPathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: bool
        """
        if not root:
            return False
        stack = []
        stack.append((root, root.val))
        while stack:
            node, path = stack.pop()
            if not node.left and not node.right and path == sum:
                return True
            if node.left:
                stack.append((node.left, path + node.left.val))
            if node.right:
                stack.append((node.right, path + node.right.val))
        return False
```

二叉树的中序遍历 (Leetcode)

[19. 删除链表的倒数第 N 个结点]

解题思路

此处撰写解题思路

代码

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
```

```
struct ListNode* removeNthFromEnd(struct ListNode* head, int n)
{
    struct ListNode *cur=head;
    int len=0;
    while(cur)
    {
        len++;
        cur=cur->next;
    }
    if(n==len)
    {
        return head->next;
    }
    cur=head;
    int count=len-n-1;
    while(count--)
    {
        cur=cur->next;
    }
    cur->next=cur->next->next;
    return head;
}
```

移除元素 (Leetcode)

Java部分

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

HashMap是工作中使用频度非常高的一个K-V存储容器。在多线程环境下，使用HashMap是不安全的，可能产生各种非期望的结果。

关于HashMap线程安全问题，可参考笔者的另一篇文章： 深入解读HashMap线程安全性问题

针对HashMap在多线程环境下不安全这个问题，HashMap的作者认为这并不是bug，而是应该使用线程安全的HashMap。

目前有如下一些方式可以获得线程安全的HashMap：

- Collections.synchronizedMap
- HashTable
- ConcurrentHashMap

其中，前两种方式由于全局锁的问题，存在很严重的性能问题。所以，著名的并发编程大师Doug Lea在JDK1.5的java.util.concurrent包下面添加了一大堆并发工具。其中就包含ConcurrentHashMap这个线程安全的HashMap。

本文就来简单介绍一下ConcurrentHashMap的实现原理。

PS：基于JDK8

0 ConcurrentHashMap在JDK7中的回顾

ConcurrentHashMap在JDK7和JDK8中的实现方式上有较大的不同。首先我们先来大概回顾一下ConcurrentHashMap在JDK7中的原理是怎样的。

0.1 分段锁技术

针对HashTable会锁整个hash表的问题，ConcurrentHashMap提出了分段锁的解决方案。

分段锁的思想就是：锁的时候不锁整个hash表，而是只锁一部分。

如何实现呢？这就用到了ConcurrentHashMap中最关键的Segment。

ConcurrentHashMap中维护着一个Segment数组，每个Segment可以看做是一个HashMap。

而Segment本身继承了ReentrantLock，它本身就是一个锁。

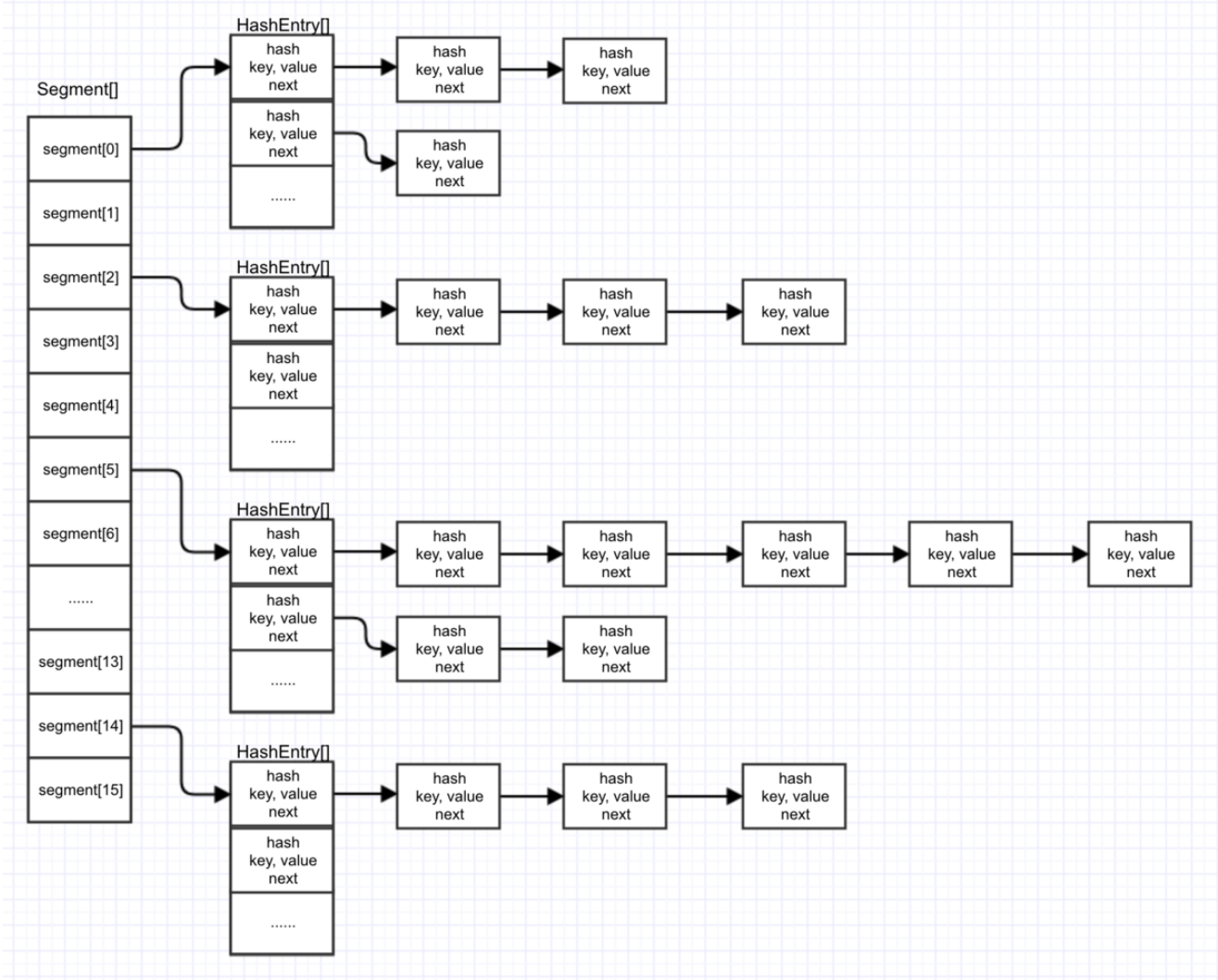
在Segment中通过HashEntry数组来维护其内部的hash表。

每个HashEntry就代表了map中的一个K-V，用HashEntry可以组成一个链表结构，通过next字段引用到其下一个元素。

上述内容在源码中的表示如下：

```
public class ConcurrentHashMap<K, V> extends AbstractMap<K, V>
        implements ConcurrentMap<K, V>, Serializable {

    // ... 省略 ...
    /**
     * The segments, each of which is a specialized hash table.
     */
    final Segment<K,V>[] segments;

    // ... 省略 ...

    /**
     * Segment是ConcurrentHashMap的静态内部类
     *
     * Segments are specialized versions of hash tables.  This
     * subclasses from ReentrantLock opportunistically, just to
     * simplify some locking and avoid separate construction.
     */
    static final class Segment<K,V> extends ReentrantLock implements Serializable {
        // ... 省略 ...
        /**
         * The per-segment table. Elements are accessed via
         * entryAt/setEntryAt providing volatile semantics.
         */
        transient volatile HashEntry<K,V>[] table;
        // ... 省略 ...
    }
    // ... 省略 ...

    /**
     * ConcurrentHashMap list entry. Note that this is never exported
     * out as a user-visible Map.Entry.
     */
    static final class HashEntry<K,V> {
        final int hash;
        final K key;
        volatile V value;
        volatile HashEntry<K,V> next;
        // ... 省略 ...
    }
}
```

所以，JDK7中，ConcurrentHashMap的整体结构可以描述为下图这样子。



由上图可见，只要我们的hash值足够分散，那么每次put的时候就会put到不同的segment中去。 而segment自己本身就是一个锁，put的时候，当前segment会将自己锁住，此时其他线程无法操作这个segment， 但不会影响到其他segment的操作。这个就是锁分段带来的好处。

0.2 线程安全的put

ConcurrentHashMap的put方法源码如下：

```
public V put(K key, V value) {
    Segment<K,V> s;
    if (value == null)
        throw new NullPointerException();
    int hash = hash(key);
    int j = (hash >>> segmentShift) & segmentMask;

    // 根据key的hash定位出一个segment，如果指定index的segment还没初始化，则调用ensureSegment方法初始化
    if ((s = (Segment<K,V>)UNSAFE.getObject          // nonvolatile; recheck
         (segments, (j << SSHIFT) + SBASE)) == null) //  in ensureSegment
        s = ensureSegment(j);
    // 调用segment的put方法
    return s.put(key, hash, value, false);
}
```

最终会调用segment的put方法，将元素put到HashEntry数组中，这里的注释中只给出锁相关的说明

```
final V put(K key, int hash, V value, boolean onlyIfAbsent) {
    // 因为segment本身就是一个锁
    // 这里调用tryLock尝试获取锁
```

```
    // 如果获取成功，那么其他线程都无法再修改这个segment
    // 如果获取失败，会调用scanAndLockForPut方法根据key和hash尝试找到这个node，如果不存在，则创建一个node并返回，如果存在则返回null
    // 查看scanAndLockForPut源码会发现他在查找的过程中会尝试获取锁，在多核CPU环境下，会尝试64次tryLock()，如果64次还没获取到，会直接调用lock()
    // 也就是说这一步一定会获取到锁
    HashEntry<K,V> node = tryLock() ? null :
        scanAndLockForPut(key, hash, value);
    V oldValue;
    try {
        HashEntry<K,V>[] tab = table;
        int index = (tab.length - 1) & hash;
        HashEntry<K,V> first = entryAt(tab, index);
        for (HashEntry<K,V> e = first;;) {
            if (e != null) {
                K k;
                if ((k = e.key) == key ||
                    (e.hash == hash && key.equals(k))) {
                    oldValue = e.value;
                    if (!onlyIfAbsent) {
                        e.value = value;
                        ++modCount;
                    }
                    break;
                }
                e = e.next;
            }
            else {
                if (node != null)
                    node.setNext(first);
                else
                    node = new HashEntry<K,V>(hash, key, value, first);
                int c = count + 1;
                if (c > threshold && tab.length < MAXIMUM_CAPACITY)
                    // 扩容
                    rehash(node);
                else
                    setEntryAt(tab, index, node);
                ++modCount;
                count = c;
                oldValue = null;
                break;
            }
        }
    } finally {
        // 释放锁
        unlock();
    }
    return oldValue;
}
```

0.3 线程安全的扩容(Rehash)

HashMap的线程安全问题大部分出在扩容(rehash)的过程中。

ConcurrentHashMap的扩容只针对每个segment中的HashEntry数组进行扩容。

由上述put的源码可知，ConcurrentHashMap在rehash的时候是有锁的，所以在rehash的过程中，其他线程无法对segment的hash表做操作，这就保证了线程安全。

1 JDK8中ConcurrentHashMap的初始化

以无参数构造函数为例，来看一下ConcurrentHashMap类初始化的时候会做些什么。

```
ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>();
```

首先会执行静态代码块和初始化类变量。 主要会初始化以下这些类变量：

```
// Unsafe mechanics
private static final sun.misc.Unsafe U;
private static final long SIZECTL;
private static final long TRANSFERINDEX;
private static final long BASECOUNT;
private static final long CELLSBUSY;
private static final long CELLVALUE;
private static final long ABASE;
private static final int ASHIFT;

static {
    try {
        U = sun.misc.Unsafe.getUnsafe();
        Class<?> k = ConcurrentHashMap.class;
        SIZECTL = U.objectFieldOffset
            (k.getDeclaredField("sizeCtl"));
        TRANSFERINDEX = U.objectFieldOffset
            (k.getDeclaredField("transferIndex"));
        BASECOUNT = U.objectFieldOffset
            (k.getDeclaredField("baseCount"));
        CELLSBUSY = U.objectFieldOffset
            (k.getDeclaredField("cellsBusy"));
        Class<?> ck = CounterCell.class;
        CELLVALUE = U.objectFieldOffset
            (ck.getDeclaredField("value"));
        Class<?> ak = Node[].class;
        ABASE = U.arrayBaseOffset(ak);
        int scale = U.arrayIndexScale(ak);
        if ((scale & (scale - 1)) != 0)
            throw new Error("data type scale not a power of two");
        ASHIFT = 31 - Integer.numberOfLeadingZeros(scale);
    } catch (Exception e) {
        throw new Error(e);
    }
}
```

这里用到了Unsafe类，其中objectFieldOffset方法用于获取指定Field(例如sizeCtl)在内存中的偏移量。

获取的这个偏移量主要用于干啥呢？不着急，在下文的分析中，遇到的时候再研究就好。

> PS：关于Unsafe的介绍和使用，可以查看笔者的另一篇文章 Unsafe类的介绍和使用

2 内部数据结构

先来从源码角度看一下JDK8中是怎么定义的存储结构。

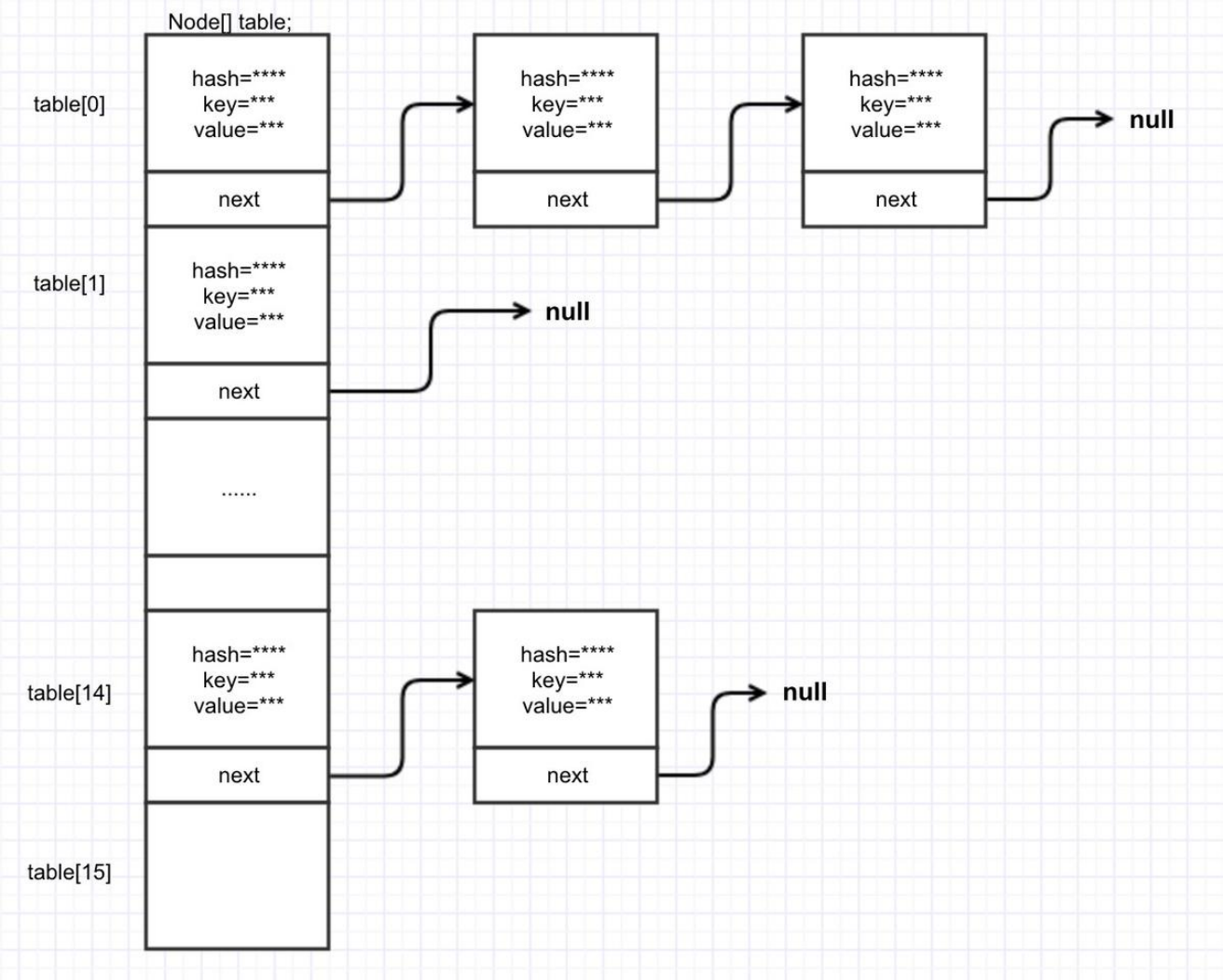
```
/**
 * The array of bins. Lazily initialized upon first insertion.
 * Size is always a power of two. Accessed directly by iterators.
 *
 * hash表，在第一次put数据的时候才初始化，他的大小总是2的倍数。
```

```
 */
transient volatile Node<K,V>[] table;

/**
 * 用来存储一个键值对
 *
 * Key-value entry.  This class is never exported out as a
 * user-mutable Map.Entry (i.e., one supporting setValue; see
 * MapEntry below), but can be used for read-only traversals used
 * in bulk tasks.  Subclasses of Node with a negative hash field
 * are special, and contain null keys and values (but are never
 * exported).  Otherwise, keys and vals are never null.
 */
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    volatile V val;
    volatile Node<K,V> next;
}
```

可以发现，JDK8与JDK7的实现由较大的不同，JDK8中不在使用Segment的概念，他更像HashMap的实现方式。

> PS：关于HashMap的原理，可以参考笔者的另一篇文章 HashMap原理及内部存储结构

这个结构可以通过下图描述出来



3 线程安全的hash表初始化

由上文可知ConcurrentHashMap是用table这个成员变量来持有hash表的。

table的初始化采用了延迟初始化策略，他会在第一次执行put的时候初始化table。

put方法源码如下（省略了暂时不相关的代码）：

```
/**
 * Maps the specified key to the specified value in this table.
 * Neither the key nor the value can be null.
 *
 * <p>The value can be retrieved by calling the {@code get} method
 * with a key that is equal to the original key.
 *
 * @param key key with which the specified value is to be associated
 * @param value value to be associated with the specified key
 * @return the previous value associated with {@code key}, or
 *         {@code null} if there was no mapping for {@code key}
 * @throws NullPointerException if the specified key or value is null
 */
public V put(K key, V value) {
    return putVal(key, value, false);
}

/** Implementation for put and putIfAbsent */
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    // 计算key的hash值
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        // 如果table是空，初始化之
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        // 省略...
    }
    // 省略...
}
```

initTable源码如下

```
/**
 * Initializes table, using the size recorded in sizeCtl.
 */
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    // #1
    while ((tab = table) == null || tab.length == 0) {
        // sizeCtl的默认值是0，所以最先走到这的线程会进入到下面的else if判断中
        // #2
        if ((sc = sizeCtl) < 0)
            Thread.yield(); // lost initialization race; just spin
        // 尝试原子性的将指定对象(this)的内存偏移量为SIZECTL的int变量值从sc更新为-1
        // 也就是将成员变量sizeCtl的值改为-1
        // #3
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
                // 双重检查，原因会在下文分析
                // #4
                if ((tab = table) == null || tab.length == 0) {
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY; // 默认初始容量为16
```

```
                    @SuppressWarnings("unchecked")
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    // #5
                    table = tab = nt; // 创建hash表，并赋值给成员变量table
                    sc = n - (n >>> 2);
                }
            } finally {
                // #6
                sizeCtl = sc;
            }
            break;
        }
    }
    return tab;
}
```

成员变量sizeCtl在ConcurrentHashMap中的其中一个作用相当于HashMap中的threshold，当hash表中元素个数超过sizeCtl时，触发扩容； 他的另一个作用类似于一个标识，例如，当他等于-1的时候，说明已经有某一线程在执行hash表的初始化了，一个小于-1的值表示某一线程正在对hash表执行resize。

这个方法首先判断sizeCtl是否小于0，如果小于0，直接将当前线程变为就绪状态的线程。

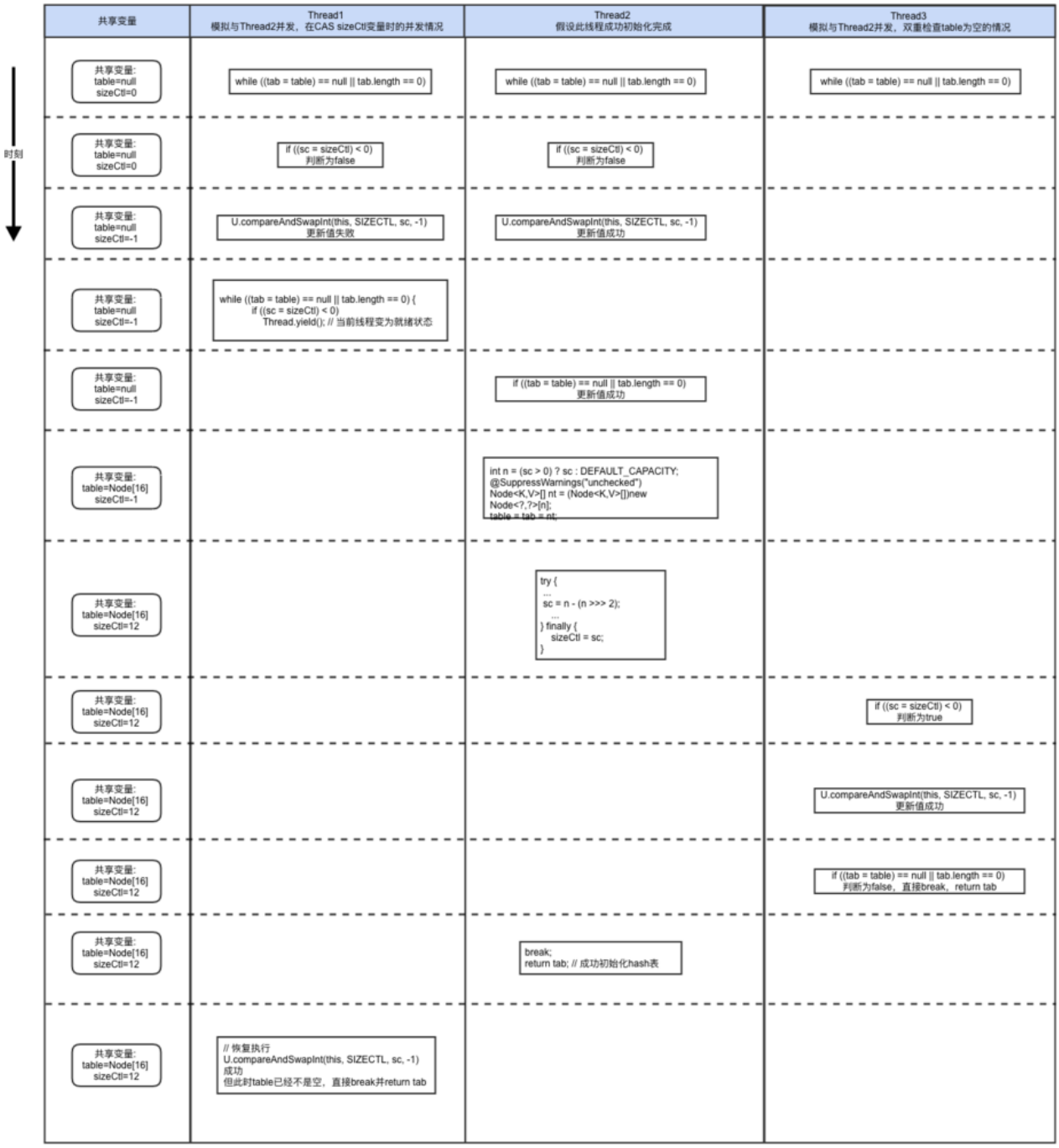
当sizeCtl大于等于0时，当前线程会尝试通过CAS的方式将sizeCtl的值修改为-1。修改失败的线程会进入下一轮循环，判断sizeCtl<0了，被yield住；修改成功的线程会继续执行下面的初始化代码。

在new Node[]之前，要再检查一遍table是否为空，这里做双重检查的原因在于，如果另一个线程执行完#1代码后挂起，此时另一个初始化的线程执行完了#6的代码，此时sizeCtl是一个大于0的值，那么再切回这个线程执行的时候，是有可能重复初始化的。关于这个问题会在下图的并发场景中说明。

然后初始化hash表，并重新计算sizeCtl的值，最终返回初始化好的hash表。

下图详细说明了几种可能导致重复初始化hash表的并发场景，我们假设Thread2最终成功初始化hash表。

- Thread1模拟的是CAS更新sizeCtl变量的并发场景
- Thread2模拟的是table的双重检查的必要性



由上图可以看出，在Thread1中如果不对sizeCtl的值更新做并发控制，Thread1是有可能走到new Node[]这一步的。在Thread3中，如果不做双重判断，Thread3也会走到new Node[]这一步。

4 线程安全的put

put操作可分为以下两类

- 当前hash表对应当前key的index上没有元素时
- 当前hash表对应当前key的index上已经存在元素时(hash碰撞)

4.1 hash表上没有元素时

对应源码如下

```
else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
    if (casTabAt(tab, i, null,
                 new Node<K,V>(hash, key, value, null)))
        break;                   // no lock when adding to empty bin
}
```

```
static final <K,V> Node<K,V> tabAt(Node<K,V>[] tab, int i) {
    return (Node<K,V>)U.getObjectVolatile(tab, ((long)i << ASHIFT) + ABASE);
}

static final <K,V> boolean casTabAt(Node<K,V>[] tab, int i,
                                    Node<K,V> c, Node<K,V> v) {
    return U.compareAndSwapObject(tab, ((long)i << ASHIFT) + ABASE, c, v);
}
```

tabAt方法通过Unsafe.getObjectVolatile()的方式获取数组对应index上的元素，getObjectVolatile作用于对应的内存偏移量上，是具备volatile内存语义的。

如果获取的是空，尝试用cas的方式在数组的指定index上创建一个新的Node。

4.2 hash碰撞时

对应源码如下

```
else {
    V oldVal = null;
    // 锁f是在4.1中通过tabAt方法获取的
    // 也就是说，当发生hash碰撞时，会以链表的头结点作为锁
    synchronized (f) {
        // 这个检查的原因在于：
        // tab引用的是成员变量table，table在发生了rehash之后，原来index上的Node可能会变
        // 这里就是为了确保在put的过程中，没有收到rehash的影响，指定index上的Node仍然是f
        // 如果不是f，那这个锁就没有意义了
        if (tabAt(tab, i) == f) {
            // 确保put没有发生在扩容的过程中，fh=-1时表示正在扩容
            if (fh >= 0) {
                binCount = 1;
                for (Node<K,V> e = f;; ++binCount) {
                    K ek;
                    if (e.hash == hash &&
                        ((ek = e.key) == key ||
                         (ek != null && key.equals(ek)))) {
                        oldVal = e.val;
                        if (!onlyIfAbsent)
                            e.val = value;
                        break;
                    }
                    Node<K,V> pred = e;
                    if ((e = e.next) == null) {
                        // 在链表后面追加元素
                        pred.next = new Node<K,V>(hash, key,
                                                  value, null);
                        break;
                    }
                }
            }
            else if (f instanceof TreeBin) {
                Node<K,V> p;
                binCount = 2;
                if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                      value)) != null) {
                    oldVal = p.val;
                    if (!onlyIfAbsent)
                        p.val = value;
                }
            }
```

```
            }
        }
        if (binCount != 0) {
            // 如果链表长度超过8个，将链表转换为红黑树，与HashMap相同，相对于JDK7来说，优化了查找效率
            if (binCount >= TREEIFY_THRESHOLD)
                treeifyBin(tab, i);
            if (oldVal != null)
                return oldVal;
            break;
        }
    }
```

不同于JDK7中segment的概念，JDK8中直接用链表的头节点做为锁。 JDK7中，HashMap在多线程并发put的情况下可能会形成环形链表，ConcurrentHashMap通过这个锁的方式，使同一时间只有有一个线程对某一链表执行put，解决了并发问题。

5 线程安全的扩容

put方法的最后一步是统计hash表中元素的个数，如果超过sizeCtl的值，触发扩容。

扩容的代码略长，可大致看一下里面的中文注释，再参考下面的分析。 其实我们主要的目的是弄明白ConcurrentHashMap是如何解决HashMap的并发问题的。 带着这个问题来看源码就好。关于HashMap存在的问题，参考本文一开始说的笔者的另一篇文章即可。

其实HashMap的并发问题多半是由于put和扩容并发导致的。

这里我们就来看一下ConcurrentHashMap是如何解决的。

扩容涉及的代码如下：

```
/**
 * The array of bins. Lazily initialized upon first insertion.
 * Size is always a power of two. Accessed directly by iterators.
 * 业务中使用的hash表
 */
transient volatile Node<K,V>[] table;

/**
 * The next table to use; non-null only while resizing.
 * 扩容时才使用的hash表，扩容完成后赋值给table，并将nextTable重置为null。
 */
private transient volatile Node<K,V>[] nextTable;

/**
 * Adds to count, and if table is too small and not already
 * resizing, initiates transfer. If already resizing, helps
 * perform transfer if work is available.  Rechecks occupancy
 * after a transfer to see if another resize is already needed
 * because resizings are lagging additions.
 *
 * @param x the count to add
 * @param check if <0, don't check resize, if <= 1 only check if uncontended
 */
private final void addCount(long x, int check) {
    // ----- 计算键值对的个数 start -----
    CounterCell[] as; long b, s;
    if ((as = counterCells) != null ||
        !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
        CounterCell a; long v; int m;
        boolean uncontended = true;
```

```
        if (as == null || (m = as.length - 1) < 0 ||
            (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
            !(uncontended =
              U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
            fullAddCount(x, uncontended);
            return;
        }
        if (check <= 1)
            return;
        s = sumCount();
    }
    // ----- 计算键值对的个数 end -----
    // ----- 判断是否需要扩容 start -----
    if (check >= 0) {
        Node<K,V>[] tab, nt; int n, sc;
        // 当上面计算出来的键值对个数超出sizeCtl时，触发扩容，调用核心方法transfer
        while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
               (n = tab.length) < MAXIMUM_CAPACITY) {
            int rs = resizeStamp(n);
            if (sc < 0) {
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transferIndex <= 0)
                    break;
                // 如果有已经在执行的扩容操作，nextTable是正在扩容中的新的hash表
                // 如果并发扩容，transfer直接使用正在扩容的新hash表，保证了不会出现hash表覆盖的情况
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                    transfer(tab, nt);
            }
            // 更新sizeCtl的值，更新成功后为负数，扩容开始
            // 此时没有并发扩容的情况，transfer中会new一个新的hash表来扩容
            else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                         (rs << RESIZE_STAMP_SHIFT) + 2))
                transfer(tab, null);
            s = sumCount();
        }
    }
    // ----- 判断是否需要扩容 end -----
}

/**
 * Moves and/or copies the nodes in each bin to new table. See
 * above for explanation.
 */
private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
    int n = tab.length, stride;
    if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_STRIDE)
        stride = MIN_TRANSFER_STRIDE; // subdivide range
    if (nextTab == null) {            // initiating
        try {
            @SuppressWarnings("unchecked")
            // 初始化新的hash表，大小为之前的2倍，并赋值给成员变量nextTable
            Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];
            nextTab = nt;
        } catch (Throwable ex) {      // try to cope with OOME
            sizeCtl = Integer.MAX_VALUE;
            return;
        }
        nextTable = nextTab;
```

```
        transferIndex = n;
    }
    int nextn = nextTab.length;
    ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);
    boolean advance = true;
    boolean finishing = false; // to ensure sweep before committing nextTab
    for (int i = 0, bound = 0;;) {
        Node<K,V> f; int fh;
        while (advance) {
            int nextIndex, nextBound;
            if (--i >= bound || finishing)
                advance = false;
            else if ((nextIndex = transferIndex) <= 0) {
                i = -1;
                advance = false;
            }
            else if (U.compareAndSwapInt
                     (this, TRANSFERINDEX, nextIndex,
                      nextBound = (nextIndex > stride ?
                                   nextIndex - stride : 0))) {
                bound = nextBound;
                i = nextIndex - 1;
                advance = false;
            }
        }
        if (i < 0 || i >= n || i + n >= nextn) {
            int sc;
            // 扩容完成时，将成员变量nextTable置为null，并将table替换为rehash后的nextTable
            if (finishing) {
                nextTable = null;
                table = nextTab;
                sizeCtl = (n << 1) - (n >>> 1);
                return;
            }
            if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
                if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
                    return;
                finishing = advance = true;
                i = n; // recheck before commit
            }
        }
        else if ((f = tabAt(tab, i)) == null)
            advance = casTabAt(tab, i, null, fwd);
        else if ((fh = f.hash) == MOVED)
            advance = true; // already processed
        else {
            // 接下来是遍历每个链表，对每个链表的元素进行rehash
            // 仍然用头结点作为锁，所以在扩容的时候，无法对这个链表执行put操作
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    Node<K,V> ln, hn;
                    if (fh >= 0) {
                        int runBit = fh & n;
                        Node<K,V> lastRun = f;
                        for (Node<K,V> p = f.next; p != null; p = p.next) {
                            int b = p.hash & n;
                            if (b != runBit) {
                                runBit = b;
                                lastRun = p;
```

```
                            }
                        }
                        if (runBit == 0) {
                            ln = lastRun;
                            hn = null;
                        }
                        else {
                            hn = lastRun;
                            ln = null;
                        }
                        for (Node<K,V> p = f; p != lastRun; p = p.next) {
                            int ph = p.hash; K pk = p.key; V pv = p.val;
                            if ((ph & n) == 0)
                                ln = new Node<K,V>(ph, pk, pv, ln);
                            else
                                hn = new Node<K,V>(ph, pk, pv, hn);
                        }
                        // setTabAt方法调用了Unsafe.putObjectVolatile来完成hash表元素的替换，具备
volatile内存语义
                        setTabAt(nextTab, i, ln);
                        setTabAt(nextTab, i + n, hn);
                        setTabAt(tab, i, fwd);
                        advance = true;
                    }
                    else if (f instanceof TreeBin) {
                        TreeBin<K,V> t = (TreeBin<K,V>)f;
                        TreeNode<K,V> lo = null, loTail = null;
                        TreeNode<K,V> hi = null, hiTail = null;
                        int lc = 0, hc = 0;
                        for (Node<K,V> e = t.first; e != null; e = e.next) {
                            int h = e.hash;
                            TreeNode<K,V> p = new TreeNode<K,V>
                                (h, e.key, e.val, null, null);
                            if ((h & n) == 0) {
                                if ((p.prev = loTail) == null)
                                    lo = p;
                                else
                                    loTail.next = p;
                                loTail = p;
                                ++lc;
                            }
                            else {
                                if ((p.prev = hiTail) == null)
                                    hi = p;
                                else
                                    hiTail.next = p;
                                hiTail = p;
                                ++hc;
                            }
                        }
                        ln = (lc <= UNTREEIFY_THRESHOLD) ? untreeify(lo) :
                            (hc != 0) ? new TreeBin<K,V>(lo) : t;
                        hn = (hc <= UNTREEIFY_THRESHOLD) ? untreeify(hi) :
                            (lc != 0) ? new TreeBin<K,V>(hi) : t;
                        setTabAt(nextTab, i, ln);
                        setTabAt(nextTab, i + n, hn);
                        setTabAt(tab, i, fwd);
                        advance = true;
                    }
```

```
                }
            }
        }
    }
}
```

根据上述代码，对ConcurrentHashMap是如何解决HashMap并发问题这一疑问进行简要说明。

- 首先new一个新的hash表(nextTable)出来，大小是原来的2倍。后面的rehash都是针对这个新的hash表操作，不涉及原hash表(table)。
- 然后会对原hash表(table)中的每个链表进行rehash，此时会尝试获取头节点的锁。这一步就保证了在rehash的过程中不能对这个链表执行put操作。
- 通过sizeCtl控制，使扩容过程中不会new出多个新hash表来。
- 最后，将所有键值对重新rehash到新表(nextTable)中后，用nextTable将table替换。这就避免了HashMap中get和扩容并发时，可能get到null的问题。
- 在整个过程中，共享变量的存储和读取全部通过volatile或CAS的方式，保证了线程安全。

6 总结

多线程环境下，对共享变量的操作一定要小心。要充分从Java内存模型的角度考虑问题。

ConcurrentHashMap中大量的用到了Unsafe类的方法，我们自己虽然也能拿到Unsafe的实例，但在生产中不建议这么做。 多数情况下，我们可以通过并发包中提供的工具来实现，例如Atomic包下面的可以用来实现CAS操作，lock包下可以用来实现锁相关的操作。

善用线程安全的容器工具，例如ConcurrentHashMap、CopyOnWriteArrayList、ConcurrentLinkedQueue等，因为我们在工作中无法像ConcurrentHashMap这样通过Unsafe的getObjectVolatile和setObjectVolatile原子性的更新数组中的元素，所以这些并发工具是很重要的。

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?

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?

Java 的线程有哪些状态，转换关系是怎么样的?

JVM 内存是如何对应到操作系统内存的?

Java 怎么防止内存溢出

简述 BIO, NIO, AIO 的区别

JMM 中内存模型是怎样的？什么是指令序列重排序?

Java 类的加载流程是怎样的？什么是双亲委派机制?

实现单例模式

volatile 关键字解决了什么问题，它的实现原理是什么?

简述常见的工厂模式以及单例模式的使用场景

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

hashcode 和 equals 方法的联系

什么是重写和重载?

简述 CAS 原理，什么是 ABA 问题，怎么解决?

线程池是如何实现的？简述线程池的任务策略

HashMap 1.7 / 1.8 的实现区别

什么情况下会发生死锁，如何解决死锁?

简述装饰者模式以及适配器模式

简述 Java 中 final 关键字的作用

== 和 equals() 的区别?

简述 Netty 线程模型，Netty 为什么如此高效?

ThreadLocal 实现原理是什么?

简述 HashSet 实现原理

简述 GC 引用链，G1收集器原理

简述动态代理与静态代理

##

网络协议部分

HTTP 与 HTTPS 有哪些区别?

TCP 怎么保证可靠传输?

TCP 与 UDP 在网络协议中的哪一层，他们之间有什么区别?

从输入 URL 到展现页面的全过程

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?

TCP长连接和短连接有那么不同的使用场景?

HTTP 中 GET 和 POST 区别

简述对称与非对称加密的概念

简述常见的 HTTP 状态码的含义（301，304，401，403)

Cookie 和 Session 的关系和区别是什么?

简述 DDOS 攻击原理，如何防范它?

简述 HTTP 1.0，1.1，2.0 的主要区别

什么是 ARP 协议？简述其使用场景

简述在四层和七层网络协议中负载均衡的原理

什么是 TCP 粘包和拆包?

RestFul 是什么？RestFul 请求的 URL 有什么特点?

简述 HTTP 报文头部的组成结构

数据库部分

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?

简述 MySQL 常见索引类型，介绍一下覆盖索引

简述乐观锁以及悲观锁的区别以及使用场景

Redis 如何实现分布式锁?

简述 Redis 持久化中 RDB 以及 AOF 方案的优缺点

简述事务的四大特性

简述 Redis 中常见类型的底层数据结构

数据库的事务隔离级别有哪些？各有哪些优缺点?

MySQL 有哪些常见的存储引擎？它们的区别是什么?

为什么 Redis 在单线程下能如此快?

简述脏读和幻读的发生场景，InnoDB 是如何解决幻读的?

简述 Redis 的线程模型以及底层架构设计

聚簇索引和非聚簇索引有什么区别?

简述 Redis 如何处理热点 key 访问

简述 Redis 的过期机制和内存淘汰策略

什么是数据库事务，MySQL 为什么会使用 InnoDB 作为默认选项

简述 Redis 的哨兵机制

简述 Redis 中如何防止缓存雪崩和缓存击穿

简述 Redis 中跳表的应用以及优缺点

简述什么是最左匹配原则

简述数据库中什么情况下进行分库，什么情况下进行分表?

简述 MySQL MVCC 的实现原理

B+ 树叶子节点存储的是什么数据

B+ 树中叶子节点存储的是什么数据

MySQL中 InnoDB 和 MyISAM 的区别是什么?

Redis 有几种数据结构？Zset 是如何实现的?

简述 Redis 集群配置以及基础原理

数据库如何设计索引，如何优化查询?

什么是 SQL 注入攻击？如何防止这类攻击?

简述数据库中的 ACID 分别是什么?

操作系统模块

进程和线程之间有什么区别?

进程间有哪些通信方式?

What is Inter Process Communication?

Inter process communication (IPC) is used for exchanging data between multiple threads in one or more processes or programs. The Processes may be running on single or multiple computers connected by a network. The full form of IPC is Inter-process communication.

It is a set of programming interface which allow a programmer to coordinate activities among various program processes which can run concurrently in an operating system. This allows a specific program to handle many user requests at the same time.

Since every single user request may result in multiple processes running in the operating system, the process may require to communicate with each other. Each IPC protocol approach has its own advantage and limitation, so it is not unusual for a single program to use all of the IPC methods.

In this Operating System Tutorial, you will learn:

Approaches for Inter-Process Communication

Here, are few important methods for interprocess communication:

What is Linux Linux Beginner Tutorial

Pipes

Pipe is widely used for communication between two related processes. This is a half-duplex method, so the first process communicates with the second process. However, in order to achieve a full-duplex, another pipe is needed.

Message Passing:

It is a mechanism for a process to communicate and synchronize. Using message passing, the process communicates with each other without resorting to shared variables.

IPC mechanism provides two operations:

- Send (message)- message size fixed or variable
- Received (message)

Message Queues:

A message queue is a linked list of messages stored within the kernel. It is identified by a message queue identifier. This method offers communication between single or multiple processes with full-duplex capacity.

Direct Communication:

In this type of inter-process communication process, should name each other explicitly. In this method, a link is established between one pair of communicating processes, and between each pair, only one link exists.

Indirect Communication:

Indirect communication establishes like only when processes share a common mailbox each pair of processes sharing several communication links. A link can communicate with many processes. The link may be bi-directional or unidirectional.

Shared Memory:

Shared memory is a memory shared between two or more processes that are established using shared memory between all the processes. This type of memory requires to protected from each other by synchronizing access across all the processes.

FIFO:

Communication between two unrelated processes. It is a full-duplex method, which means that the first process can communicate with the second process, and the opposite can also happen.

Why IPC?

Here, are the reasons for using the interprocess communication protocol for information sharing:

- It helps to speedup modularity
- Computational
- Privilege separation
- Convenience
- Helps operating system to communicate with each other and synchronize their actions.

Terms Used in IPC

The following are a few important terms used in IPC:

Semaphores: A semaphore is a signaling mechanism technique. This OS method either allows or disallows access to the resource, which depends on how it is set up.

Signals: It is a method to communicate between multiple processes by way of signaling. The source process will send a signal which is recognized by number, and the destination process will handle it.

What is Like FIFOS and Unlike FIFOS

Like FIFOS	Unlike FIFOS
It follows FIFO method	Method to pull specific urgent messages before they reach the front
FIFO exists independently of both sending and receiving processes.	Always ready, so don't need to open or close.
Allows data transfer among unrelated processes.	Not have any synchronization problems between open & close.

Summary:

- Definition: Inter-process communication is used for exchanging data between multiple threads in one or more processes or programs.
- Pipe is widely used for communication between two related processes.
- Message passing is a mechanism for a process to communicate and synchronize.
- A message queue is a linked list of messages stored within the kernel
- Direct process is a type of inter-process communication process, should name each other explicitly.
- Indirect communication establishes like only when processes share a common mailbox each pair of processes sharing several communication links.
- Shared memory is a memory shared between two or more processes that are established using shared memory between all the processes.

- Inter Process Communication method helps to speedup modularity.
- A semaphore is a signaling mechanism technique.
- Signaling is a method to communicate between multiple processes by way of signaling.
- Like FIFO follows FIFO method whereas Unlike FIFO use method to pull specific urgent messages before they reach the front.

多线程和多进程的区别是什么?

为什么进程切换慢，线程切换快?

简述创建进程的流程

简述 Linux 虚拟内存的页面置换算法

前端部分

简述 diff 算法的实现机制和使用场景

什么是闭包，什么是立即执行函数，它的作用是什么？简单说一下闭包的使用场景

Javascript 中 == 与 === 的区别是什么?

简述什么是 XSS 攻击以及 CSRF 攻击?

简述 Vue 的生命周期

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

简述项目打包和发布的流程

简述 React 的生命周期以及通信方式

sessionStorage 和 localStorage 有什么区别?

简述浏览器的垃圾回收机制

简述浏览器的渲染过程，重绘和重排在渲染过程中的哪一部分?

简述 Javascript 中的防抖与节流的原理并尝试实现

移动端适配有哪些方案?

MVC 模型和 MVVM 模型的区别

简述 React 中的 Effect Hook 机制

简述 React setstate 原理

简述 CSS 盒模型

简述 weakMap 与 Map 的区别

简述强缓存与协商缓存的区别和使用场景

简述 typeof 和 instanceof 的原理

简述 JavaScript 事件循环机制

promise 有哪些状态？简述 promise.all 的实现原理

系统设计部分

电商系统中，如何实现秒杀功能？如何解决商品的超卖问题？

简述 CAP 理论

如何实现唯一的分布式 ID

简述中间件削峰和限流的使用场景

简述 Kafka 的基本架构，如何用 Kafka 保证消息的有序性？

简述 Dubbo 服务注册与发现的过程

简述一致性哈希算法的实现方式及原理

非技术模块

下一份工作希望学习到什么？

项目中最难的地方是哪里？你学习到了什么？

最近在看什么书吗，有没有接触过什么新技术？

团队合作沟通中遇到过什么问题？

未来的职业规划是什么？

成长过程中影响你最深的事件和人

你用过美团吗？说说对对美团的看法，使用上有哪些优缺点

团队中对技术选型有冲突的话怎么解决？

字节

算法部分

21. 合并两个有序链表

```
# `Definition for singly-linked list.`

# `class ListNode:`

#     `def __init__(self, x):`

#         `self.val = x`

#         `self.next = None`

`class Solution:`
    `def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:`
        `if l1 and l2:`
```

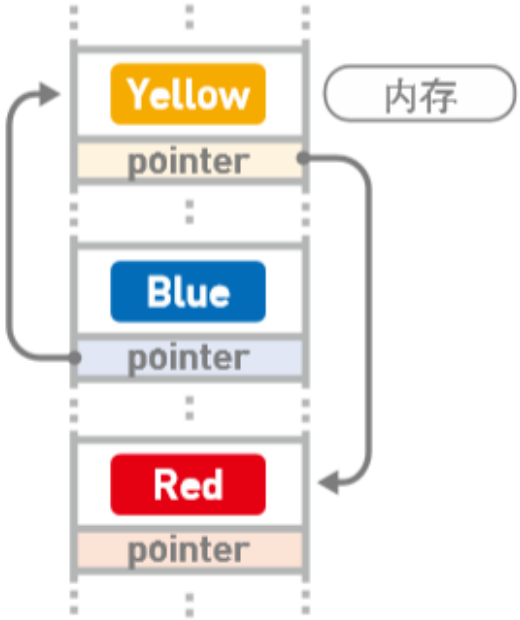
```
            `if l1.val > l2.val: l1, l2 = l2, l1`
            `l1.next = self.mergeTwoLists(l1.next, l2)`
        `return l1 or l2`
```

备注： 在 Python 中，and 和 or 都有提前截至运算的功能。

and：如果 and 前面的表达式已经为 False，那么 and 之后的表达式将被 跳过，返回左表达式结果
or：如果 or 前面的表达式已经为 True，那么 or 之后的表达式将被跳过，直接返回左表达式的结果
例子：[] and 7 等于 []
代码流程： （按行数）

判断 l1 或 l2 中是否有一个节点为空，如果存在，那么我们只需要把不为空的节点接到链表后面即可
对 l1 和 l2 重新赋值，使得 l1 指向比较小的那个节点对象
修改 l1 的 next 属性为递归函数返回值
返回 l1，注意：如果 l1 和 l2 同时为 None，此时递归停止返回 None
效率

时间复杂度：O(n)O(n)
空间复杂度：【考虑递归开栈】O(n)O(n)【不考虑】O(1)O(1)
拓展
关于链表不熟的童鞋看过里



【知识卡片】链表是数据结构之一，其中的数据呈线性排列。在链表中，数据的添加和删除都较为方便， 就是访问比较耗费时间。

关键点：
实际上，相比较数组来说，并不存在链表这样一个对象，链表是由多个节点组成的，因此，我们能接触到的数据对象只有节点。我们可以根据节点来寻找周围节点，许多节点之间的关系抽象地构成了一个链表。

10亿个数中如何高效地找到最大的一个数以及最大的第 K 个数

25. K 个一组翻转链表

方法一：模拟
思路与算法

本题的目标非常清晰易懂，不涉及复杂的算法，但是实现过程中需要考虑的细节比较多，容易写出冗长的代码。主要考查面试者设计的能力。

我们需要把链表节点按照 k 个一组分组，所以可以使用一个指针 head 依次指向每组的头节点。这个指针每次向前移动 k 步，直至链表结尾。对于每个分组，我们先判断它的长度是否大于等于 k。若是，我们就翻转这部分链表，否则不需要翻转。

接下来的问题就是如何翻转一个分组内的子链表。翻转一个链表并不难，过程可以参考「206. 反转链表」。但是对于一个子链表，除了翻转其本身之外，还需要将子链表的头部与上一个子链表连接，以及子链表的尾部与下一个子链表连接。如下图所示：

因此，在翻转子链表的时候，我们不仅需要子链表头节点 head，还需要有 head 的上一个节点 pre，以便翻转完后把子链表再接回 pre。

但是对于第一个子链表，它的头节点 head 前面是没有节点 pre 的。太麻烦了！难道只能特判了吗？答案是否定的。没有条件，我们就创造条件；没有节点，我们就创建一个节点。我们新建一个节点，把它接到链表的头部，让它作为 pre 的初始值，这样 head 前面就有了一个节点，我们就可以避开链表头部的边界条件。这么做还有一个好处，下面我们会看到。

反复移动指针 head 与 pre，对 head 所指向的子链表进行翻转，直到结尾，我们就得到了答案。下面我们该返回函数值了。

有的同学可能发现这又是一件麻烦事：链表翻转之后，链表的头节点发生了变化，那么应该返回哪个节点呢？照理来说，前 k 个节点翻转之后，链表的头节点应该是第 k 个节点。那么要在遍历过程中记录第 k 个节点吗？但是如果链表里面没有 k 个节点，答案又还是原来的头节点。我们又多了一大堆循环和判断要写，太崩溃了！

等等！还记得我们创建了节点 pre 吗？这个节点一开始被连接到了头节点的前面，而无论之后链表有没有翻转，它的 next 指针都会指向正确的头节点。那么我们只要返回它的下一个节点就好了。至此，问题解决。

```
class Solution:
  # 翻转一个子链表，并且返回新的头与尾
  def reverse(self, head: ListNode, tail: ListNode):
    prev = tail.next
    p = head
    while prev != tail:
      nex = p.next
      p.next = prev
      prev = p
      p = nex
    return tail, head
```

```
def reverseKGroup(self, head: ListNode, k: int) -> ListNode:
    hair = ListNode(0)
    hair.next = head
    pre = hair

    while head:
        tail = pre
        # 查看剩余部分长度是否大于等于 k
        for i in range(k):
            tail = tail.next
            if not tail:
                return hair.next
        nex = tail.next
        head, tail = self.reverse(head, tail)
        # 把子链表重新接回原链表
        pre.next = head
        tail.next = nex
        pre = tail
        head = tail.next
```

```
    return hair.next
```

复杂度分析

时间复杂度：O(n)O(n)，其中 nn 为链表的长度。head 指针会在 O(\lfloor \frac{n}{k} \rfloor)O(⌊
k
n
⌋) 个节点上停留，每次停留需要进行一次 O(k)O(k) 的翻转操作。

空间复杂度：O(1)O(1)，我们只需要建立常数个变量。

103. 二叉树的锯齿形层序遍历

解题思路

这题是之前的二叉树的层次遍历的变形题，我们可以发现，我们只要每次在偶数层反转我们那层的答案就能够得到我们的锯齿形层序遍历的结果

代码

```
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> ans = new ArrayList<>();
        Deque<TreeNode> deque = new LinkedList<>();
        if(root != null)
            deque.add(root);
        boolean k = false;  //判断是否在偶数层，初始为第一层，默认为false
        while(!deque.isEmpty()) {
            List<Integer> t = new ArrayList<>();
            for(int i = deque.size(); i > 0; i--) {
                TreeNode node = deque.poll();
                t.add(node.val);
                if(node.left != null)
                    deque.add(node.left);
                if(node.right != null)
                    deque.add(node.right);
            }
            if(k)   //如果是偶数层则反转
                Collections.reverse(t);
            k = !k; //奇数变偶数，偶数变奇数
            ans.add(t);
        }
        return ans;
    }
}
```

复杂度分析

- 时间复杂度：O(n)
- 空间复杂度：O(n)

912. 排序数组

前言
本题你可以选择直接调用库函数来对序列进行排序，但意义不大。由于排序算法有很多，本文只介绍三种常见的基于比较的复杂度较低的排序。

方法一：快速排序
思路和算法

快速排序的主要思想是通过划分将待排序的序列分成前后两部分，其中前一部分的数据都比后一部分的数据要小，然后再递归调用函数对两部分的序列分别进行快速排序，以此使整个序列达到有序。

我们定义函数 randomized_quicksort(nums, l, r) 为对 nums 数组里 [l,r][l,r] 的部分进行排序，每次先调用 randomized_partition 函数对 nums 数组里 [l,r][l,r] 的部分进行划分，并返回分界值的下标 pos，然后按上述将的递归调用 randomized_quicksort(nums, l, pos - 1) 和 randomized_quicksort(nums, pos + 1, r) 即可。

那么核心就是划分函数的实现了，划分函数一开始需要确定一个分界值（我们称之为主元 pivot），然后再进行划分。而主元的选取有很多种方式，这里我们采用随机的方式，对当前划分区间 [l,r][l,r] 里的数等概率随机一个作为我们的主元，再将主元放到区间末尾，进行划分。

整个划分函数 partition 主要涉及两个指针 ii 和 jj，一开始 i = l - 1，j = l。我们需要实时维护两个指针使得任意时候，对于任意数组下标 kk，我们有如下条件成立：

l\leq k\leq il≤k≤i 时，\textit{nums}[k]\leq \textit{pivot}nums[k]≤pivot。

i+1\leq k\leq j-1i+1≤k≤j−1 时，\textit{nums}[k]> \textit{pivot}nums[k]>pivot。

k==rk==r 时，\textit{nums}[k]=\textit{pivot}nums[k]=pivot。

我们每次移动指针 jj，如果 \textit{nums}[j]> pivotnums[j]>pivot，我们只需要继续移动指针 jj，即能使上述三个条件成立，否则我们需要将指针 ii 加一，然后交换 \textit{nums}[i]nums[i] 和 \textit{nums}[j]nums[j]，再移动指针 jj 才能使得三个条件成立。

当 jj 移动到 r-1r−1 时结束循环，此时我们可以由上述三个条件知道 [l,i][l,i] 的数都小于等于主元 pivot，[i+1,r-1][i+1,r−1] 的数都大于主元 pivot，那么我们只要交换 \textit{nums}[i+1]nums[i+1] 和 \textit{nums}[r]nums[r]，即能使得 [l,i+1][l,i+1] 区间的数都小于 [i+2,r][i+2,r] 区间的数，完成一次划分，且分界值下标为 i+1i+1，返回即可。

如下的动图展示了一次划分的过程，刚开始随机选了 44 作为主元，与末尾元素交换后开始划分：


```
class Solution {
    int partition(vector<int>& nums, int l, int r) {
        int pivot = nums[r];
        int i = l - 1;
        for (int j = l; j <= r - 1; ++j) {
```

```
            if (nums[j] <= pivot) {
                i = i + 1;
                swap(nums[i], nums[j]);
            }
        }
        swap(nums[i + 1], nums[r]);
        return i + 1;
    }
    int randomized_partition(vector<int>& nums, int l, int r) {
        int i = rand() % (r - l + 1) + l; // 随机选一个作为我们的主元
        swap(nums[r], nums[i]);
        return partition(nums, l, r);
    }
    void randomized_quicksort(vector<int>& nums, int l, int r) {
        if (l < r) {
            int pos = randomized_partition(nums, l, r);
            randomized_quicksort(nums, l, pos - 1);
            randomized_quicksort(nums, pos + 1, r);
        }
    }
public:
    vector<int> sortArray(vector<int>& nums) {
        srand((unsigned)time(NULL));
        randomized_quicksort(nums, 0, (int)nums.size() - 1);
        return nums;
    }
};
```

复杂度分析

时间复杂度：基于随机选取主元的快速排序时间复杂度为期望 O(n\log n)O(nlogn)，其中 nn 为数组的长度。详细证明过程可以见《算法导论》第七章，这里不再大篇幅赘述。

空间复杂度：O(h)O(h)，其中 hh 为快速排序递归调用的层数。我们需要额外的 O(h)O(h) 的递归调用的栈空间，由于划分的结果不同导致了快速排序递归调用的层数也会不同，最坏情况下需 O(n)O(n) 的空间，最优情况下每次都平衡，此时整个递归树高度为 \log nlogn，空间复杂度为 O(\log n)O(logn)。

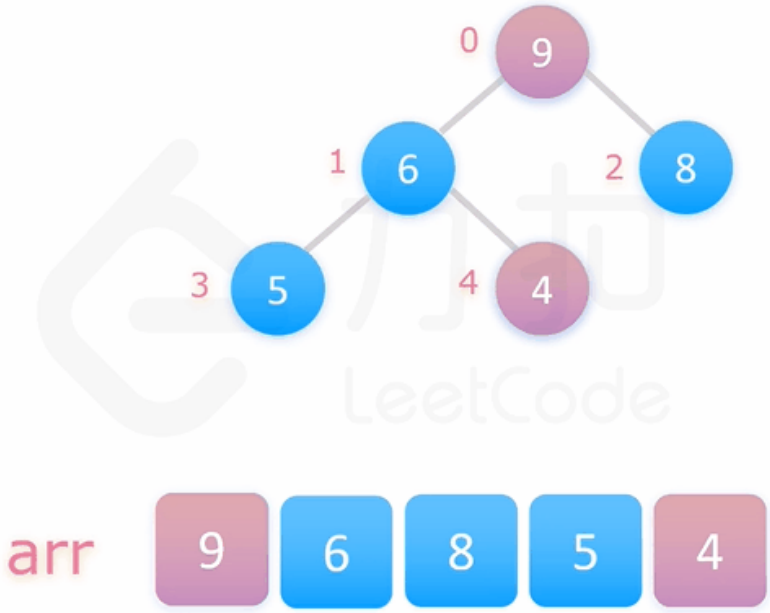
方法二：堆排序
预备知识

堆
思路和算法

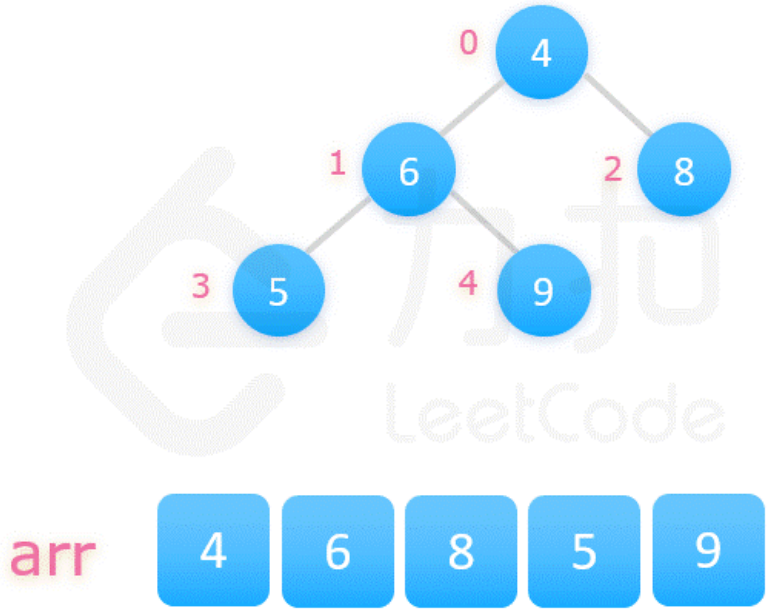
堆排序的思想就是先将待排序的序列建成大根堆，使得每个父节点的元素大于等于它的子节点。此时整个序列最大值即为堆顶元素，我们将其与末尾元素交换，使末尾元素为最大值，然后再调整堆顶元素使得剩下的 n-1n−1 个元素仍为大根堆，再重复执行以上操作我们即能得到一个有序的序列。

如下两个动图展示了对 [4, 6, 8, 5, 9] 这个数组堆排序的过程：

交换元素



初始化堆




```
class Solution {
    void maxHeapify(vector<int>& nums, int i, int len) {
        for (; (i << 1) + 1 <= len;) {
            int lson = (i << 1) + 1;
            int rson = (i << 1) + 2;
            int large;
            if (lson <= len && nums[lson] > nums[i]) {
                large = lson;
            } else {
                large = i;
            }
            if (rson <= len && nums[rson] > nums[large]) {
                large = rson;
            }
            if (large != i) {
                swap(nums[i], nums[large]);
```

```
                i = large;
            } else {
                break;
            }
        }
    }
    void buildMaxHeap(vector<int>& nums, int len) {
        for (int i = len / 2; i >= 0; --i) {
            maxHeapify(nums, i, len);
        }
    }
    void heapSort(vector<int>& nums) {
        int len = (int)nums.size() - 1;
        buildMaxHeap(nums, len);
        for (int i = len; i >= 1; --i) {
            swap(nums[i], nums[0]);
            len -= 1;
            maxHeapify(nums, 0, len);
        }
    }
public:
    vector<int> sortArray(vector<int>& nums) {
        heapSort(nums);
        return nums;
    }
};
```

复杂度分析

时间复杂度：O(n\log n)O(nlogn)。初始化建堆的时间复杂度为 O(n)O(n)，建完堆以后需要进行 n-1n−1 次调整，一次调整（即 maxHeapify） 的时间复杂度为 O(\log n)O(logn)，那么 n-1n−1 次调整即需要 O(n\log n)O(nlogn) 的时间复杂度。因此，总时间复杂度为 O(n+n\log n)=O(n\log n)O(n+nlogn)=O(nlogn)。

空间复杂度：O(1)O(1)。只需要常数的空间存放若干变量。

方法三：归并排序
思路

归并排序利用了分治的思想来对序列进行排序。对一个长为 nn 的待排序的序列，我们将其分解成两个长度为 \frac{n}{2}
2
n

的子序列。每次先递归调用函数使两个子序列有序，然后我们再线性合并两个有序的子序列使整个序列有序。

算法

定义 mergeSort(nums, l, r) 函数表示对 nums 数组里 [l,r][l,r] 的部分进行排序，整个函数流程如下：

递归调用函数 mergeSort(nums, l, mid) 对 nums 数组里 [l,\textit{mid}][l,mid] 部分进行排序。

递归调用函数 mergeSort(nums, mid + 1, r) 对 nums 数组里 [\textit{mid}+1,r][mid+1,r] 部分进行排序。

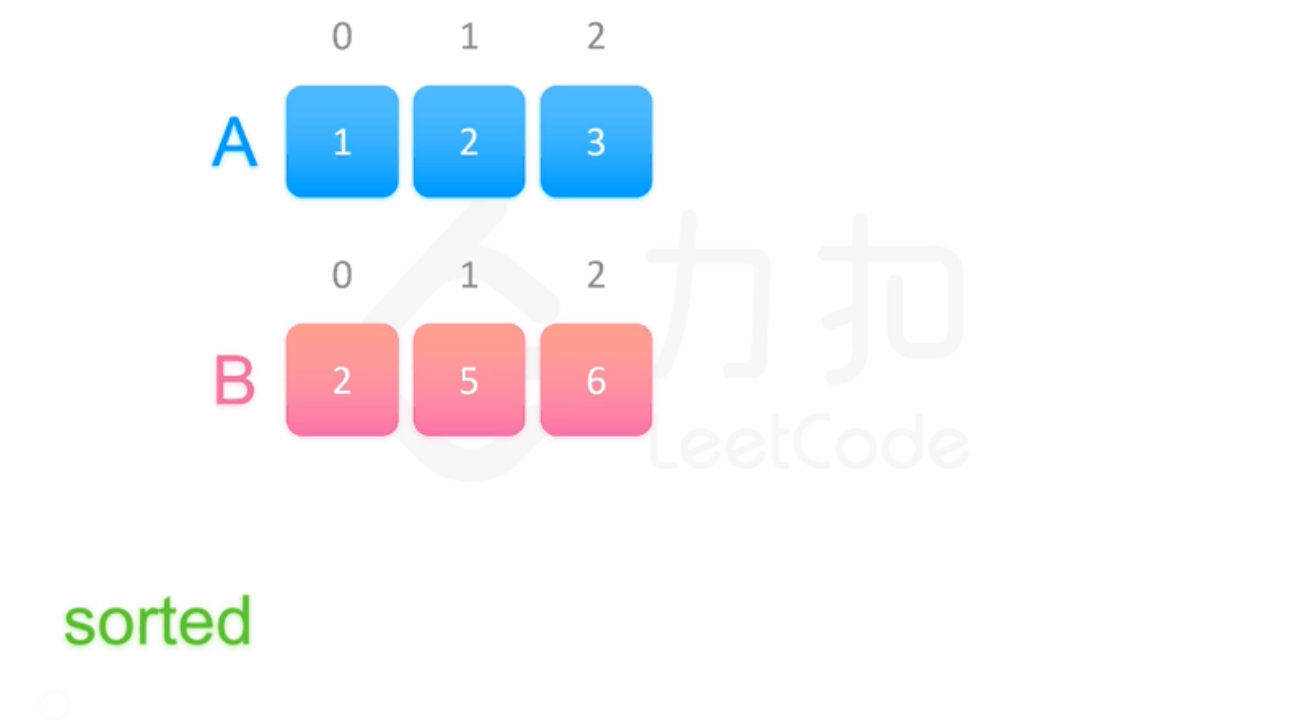
此时 nums 数组里 [l,\textit{mid}][l,mid] 和 [\textit{mid}+1,r][mid+1,r] 两个区间已经有序，我们对两个有序区间线性归并即可使 nums 数组里 [l,r][l,r] 的部分有序。

线性归并的过程并不难理解，由于两个区间均有序，所以我们维护两个指针 ii 和 jj 表示当前考虑到 [l,\textit{mid}][l,mid] 里的第 ii 个位置和 [\textit{mid}+1,r][mid+1,r] 的第 jj 个位置。

如果 nums[i] <= nums[j]，那么我们就将 \textit{nums}[i]nums[i] 放入临时数组 tmp 中并让 i += 1，即指针往后移。否则我们就将 \textit{nums}[j]nums[j] 放入临时数组 tmp 中并让 j += 1 。如果有一个指针已经移到了区间的末尾，那么就把另一个区间里的数按顺序加入 tmp 数组中即可。

这样能保证我们每次都是让两个区间中较小的数加入临时数组里，那么整个归并过程结束后 [l,r][l,r] 即为有序的。

如下的动图展示了两个有序数组线性归并的过程：



函数递归调用的入口为 `mergeSort(nums, 0, nums.length - 1)`，递归结束当且仅当 `l >= r`。

```
class Solution {
    vector<int> tmp;
    void mergeSort(vector<int>& nums, int l, int r) {
        if (l >= r) return;
        int mid = (l + r) >> 1;
        mergeSort(nums, l, mid);
        mergeSort(nums, mid + 1, r);
        int i = l, j = mid + 1;
        int cnt = 0;
        while (i <= mid && j <= r) {
            if (nums[i] <= nums[j]) {
                tmp[cnt++] = nums[i++];
            }
            else {
                tmp[cnt++] = nums[j++];
            }
        }
        while (i <= mid) {
            tmp[cnt++] = nums[i++];
        }
        while (j <= r) {
            tmp[cnt++] = nums[j++];
        }
        for (int i = 0; i < r - l + 1; ++i) {
            nums[i + l] = tmp[i];
        }
    }
public:
    vector<int> sortArray(vector<int>& nums) {
        tmp.resize((int)nums.size(), 0);
        mergeSort(nums, 0, (int)nums.size() - 1);
        return nums;
    }
```

```
};
```

复杂度分析

时间复杂度：O(n\log n)O(nlogn)。由于归并排序每次都将当前待排序的序列折半成两个子序列递归调用，然后再合并两个有序的子序列，而每次合并两个有序的子序列需要 O(n)O(n) 的时间复杂度，所以我们可以列出归并排序运行时间 T(n)T(n) 的递归表达式：
T(n)=2T(\frac{n}{2})+O(n)
T(n)=2T(
2
n

)+O(n)

根据主定理我们可以得出归并排序的时间复杂度为 O(n\log n)O(nlogn)。

空间复杂度：O(n)O(n)。我们需要额外 O(n)O(n) 空间的 \textit{tmp}tmp 数组，且归并排序递归调用的层数最深为 \log_2 nlog
2

n，所以我们还需要额外的 O(\log n)O(logn) 的栈空间，所需的空间复杂度即为 O(n+\log n) = O(n)O(n+logn)=O(n)。

445. 两数相加 II

解题思路

1. 首先先得到l1比l2多几位数字，即得到div变量。
2. div<0的情况同于div>0的情况，这里以div>=0来说明。
3. 由于链表是从高位开始，故可设置函数返回值为进位信号，即可递归得到低位数的进位情况。
4. div>0时，两链表位数不对，要继续递归更高位数链表的下一个节点。div==0时，两链表位数匹配，就可更新值，并且返回新节点值。

代码

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
    int div=0;      //l1和l2节点数量之差的绝对值
    bool add(ListNode* l1, ListNode* l2)    //l1节点数量>=l2节点数量
    {
        if(!l1||!l2)    return 0;       //都是空指针，无进位
        else if(div==0)                 //如果位数匹配
        {
            l1->val+=l2->val+add(l1->next,l2->next);    //更新l1值
            bool carry=l1->val>9;                       //得到进位信号
            l1->val%=10;                                //更新l1值
            return carry;                               //返回进位信号
        }
        else
        {
            --div;                          //更新div值
            if(add(l1->next,l2))            //如果低位进位
            {
                l1->val=(l1->val+1)%10;     //更新l1值
                return l1->val==0;          //返回进位信号
            }
            return 0;                       //返回进位信号
        }
    }
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode *h1=l1,*h2=l2;
        while(l1||l2)
        {
            if(l1)
            {
                l1=l1->next;
                ++div;
            }
            if(l2)
            {
                l2=l2->next;
                --div;
            }
        }
        if(div<0)
        {
            div=-div;
            if(add(h2,h1))  h2=new ListNode(1,h2);
```

```
            return h2;
        }
        else
        {
            if(add(h1,h2))  h1=new ListNode(1,h1);
            return h1;
        }
    }
};
```

同类题目的类似解法

面试题02.05.链表求和

类似题目题解链接：链表求和题解

2.两数相加

类似题目题解链接：两数相加题解

33. 搜索旋转排序数组

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int lo = 0, hi = nums.size() - 1;
        while (lo < hi) {
            int mid = (lo + hi) / 2;
            if ((nums[0] > target) ^ (nums[0] > nums[mid]) ^ (target > nums[mid]))
                lo = mid + 1;
            else
                hi = mid;
        }
        return lo == hi && nums[lo] == target ? lo : -1;
    }
};
```

以二分搜索为基本思路

简要来说：

nums[0] <= nums[mid]（0 - mid不包含旋转）且nums[0] <= target <= nums[mid] 时 high 向前规约；

nums[mid] < nums[0]（0 - mid包含旋转），target <= nums[mid] < nums[0] 时向前规约（target 在旋转位置到 mid 之间）

nums[mid] < nums[0]，nums[mid] < nums[0] <= target 时向前规约（target 在 0 到旋转位置之间）

其他情况向后规约

也就是说nums[mid] < nums[0]，nums[0] > target，target > nums[mid] 三项均为真或者只有一项为真时向后规约。

原文的分析是：

注意到原数组为有限制的有序数组（除了在某个点会突然下降外均为升序数组）

if nums[0] <= nums[I] 那么 nums[0] 到 nums[i] 为有序数组,那么当 nums[0] <= target <= nums[i] 时我们应该在 0-i0−i 范围内查找；

if nums[i] < nums[0] 那么在 0-i0−i 区间的某个点处发生了下降（旋转），那么 I+1I+1 到最后一个数字的区间为有序数组，并且所有的数字都是小于 nums[0] 且大于 nums[i]，当target不属于 nums[0] 到 nums[i] 时（target <= nums[i] < nums[0] or nums[i] < nums[0] <= target），我们应该在 0-i0−i 区间内查找。
上述三种情况可以总结如下：

```
nums[0] <= target <= nums[i]
           target <= nums[i] < nums[0]
                     nums[i] < nums[0] <= target
```

所以我们进行三项判断：

(nums[0] <= target)，(target <= nums[i])，(nums[i] < nums[0])，现在我们想知道这三项中有哪两项为真（明显这三项不可能均为真或均为假（因为这三项可能已经包含了所有情况））

所以我们现在只需要区别出这三项中有两项为真还是只有一项为真。

使用 “异或” 操作可以轻松的得到上述结果（两项为真时异或结果为假，一项为真时异或结果为真，可以画真值表进行验证）

之后我们通过二分查找不断做小 target 可能位于的区间直到 low==high，此时如果 nums[low]==target 则找到了，如果不等则说明该数组里没有此项。

142. 环形链表 II

方法一：哈希表
思路与算法

一个非常直观的思路是：我们遍历链表中的每个节点，并将它记录下来；一旦遇到了此前遍历过的节点，就可以判定链表中存在环。借助哈希表可以很方便地实现。

代码

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        unordered_set<ListNode *> visited;
        while (head != nullptr) {
            if (visited.count(head)) {
                return head;
            }
            visited.insert(head);
            head = head->next;
        }
        return nullptr;
    }
```

```
};
```

复杂度分析

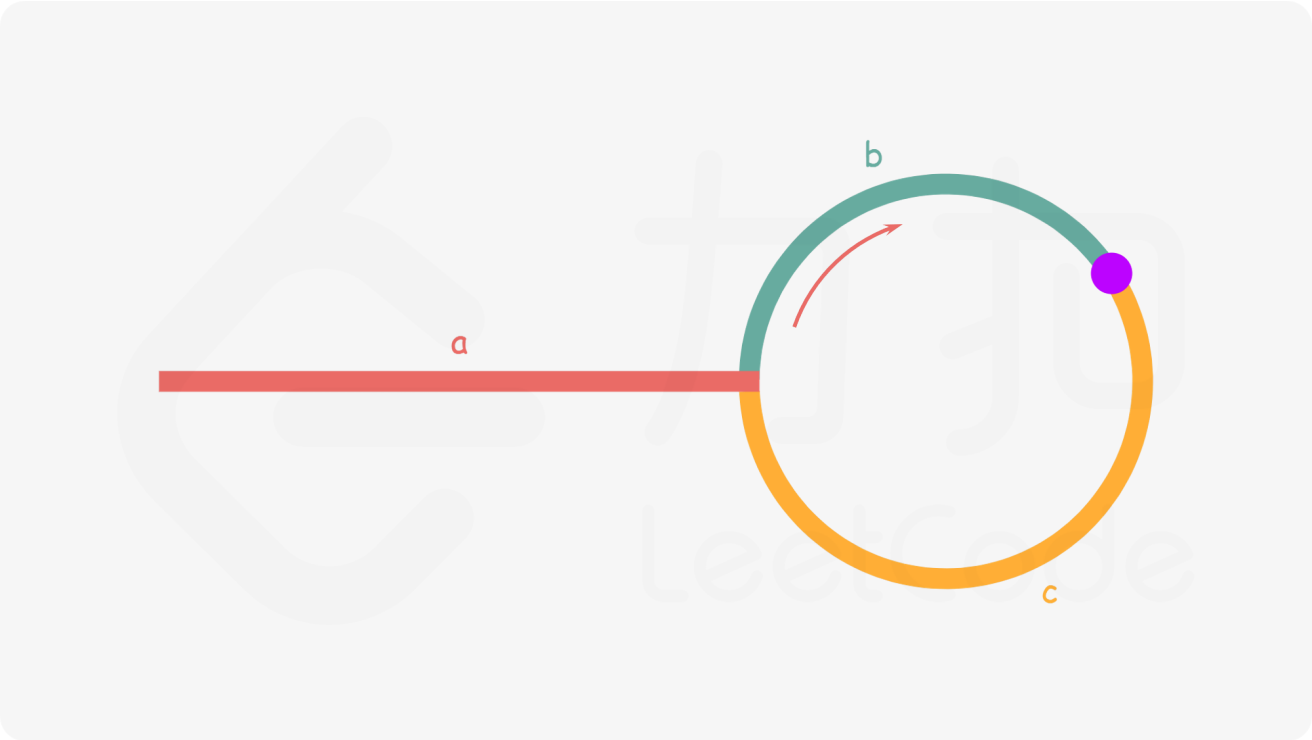
时间复杂度：O(N)O(N)，其中 NN 为链表中节点的数目。我们恰好需要访问链表中的每一个节点。

空间复杂度：O(N)O(N)，其中 NN 为链表中节点的数目。我们需要将链表中的每个节点都保存在哈希表当中。

方法二：快慢指针
思路与算法

我们使用两个指针，\textit{fast}fast 与 \textit{slow}slow。它们起始都位于链表的头部。随后，\textit{slow}slow 指针每次向后移动一个位置，而 \textit{fast}fast 指针向后移动两个位置。如果链表中存在环，则 \textit{fast}fast 指针最终将再次与 \textit{slow}slow 指针在环中相遇。

如下图所示，设链表中环外部分的长度为 aa。\textit{slow}slow 指针进入环后，又走了 bb 的距离与 \textit{fast}fast 相遇。此时，\textit{fast}fast 指针已经走完了环的 nn 圈，因此它走过的总距离为 a+n(b+c)+b=a+(n+1)b+nca+n(b+c)+b=a+(n+1)b+nc。



根据题意，任意时刻，\textit{fast}fast 指针走过的距离都为 \textit{slow}slow 指针的 22 倍。因此，我们有

a+(n+1)b+nc=2(a+b) \implies a=c+(n-1)(b+c)
a+(n+1)b+nc=2(a+b)⟹a=c+(n−1)(b+c)

有了 a=c+(n-1)(b+c)a=c+(n−1)(b+c) 的等量关系，我们会发现：从相遇点到入环点的距离加上 n-1n−1 圈的环长，恰好等于从链表头部到入环点的距离。

因此，当发现 \textit{slow}slow 与 \textit{fast}fast 相遇时，我们再额外使用一个指针 \textit{ptr}ptr。起始，它指向链表头部；随后，它和 \textit{slow}slow 每次向后移动一个位置。最终，它们会在入环点相遇。

代码

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode *slow = head, *fast = head;
        while (fast != nullptr) {
            slow = slow->next;
            if (fast->next == nullptr) {
                return nullptr;
```

```
            }
            fast = fast->next->next;
            if (fast == slow) {
                ListNode *ptr = head;
                while (ptr != slow) {
                    ptr = ptr->next;
                    slow = slow->next;
                }
                return ptr;
            }
        }
        return nullptr;
    }
};
```

复杂度分析

时间复杂度：O(N)O(N)，其中 NN 为链表中节点的数目。在最初判断快慢指针是否相遇时，\textit{slow}slow 指针走过的距离不会超过链表的总长度；随后寻找入环点时，走过的距离也不会超过链表的总长度。因此，总的执行时间为 O(N)+O(N)=O(N)O(N)+O(N)=O(N)O(N)。

空间复杂度：O(1)O(1)。我们只使用了 \textit{slow}, \textit{fast}, \textit{ptr}slow,fast,ptr 三个指针。

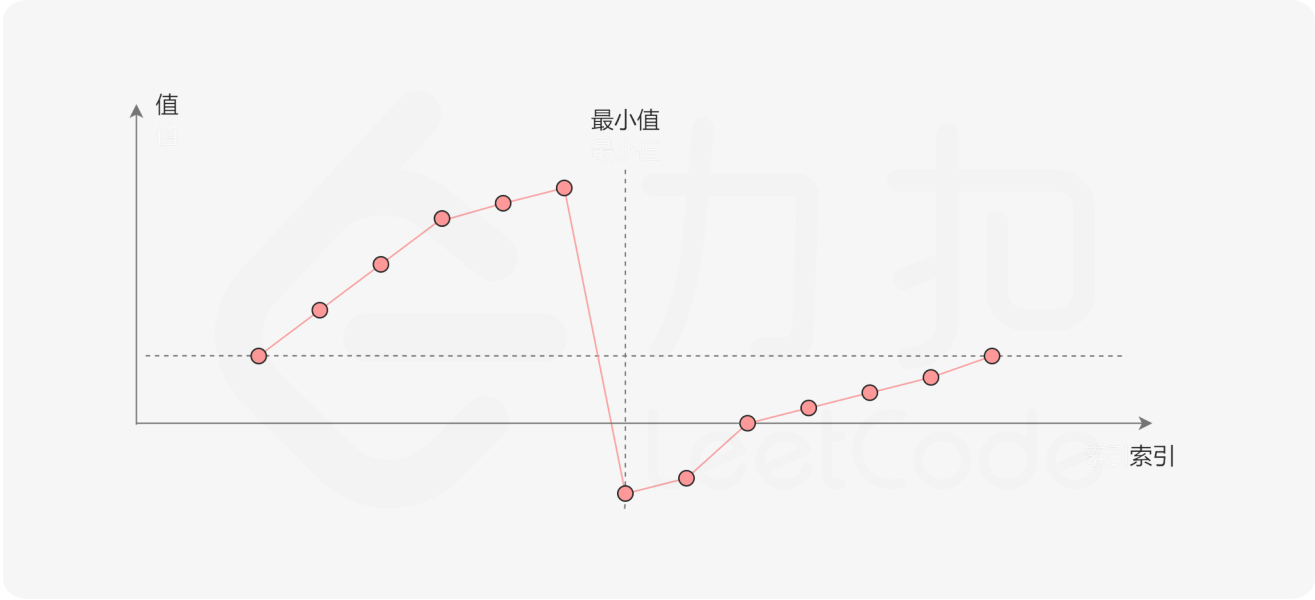
给定 100G 的 URL 磁盘数据，使用最多 1G 内存，统计出现频率最高的 Top K 个 URL

153. 寻找旋转排序数组中的最小值

方法一：二分查找

思路与算法

一个不包含重复元素的升序数组在经过旋转之后，可以得到下面可视化的折线图：

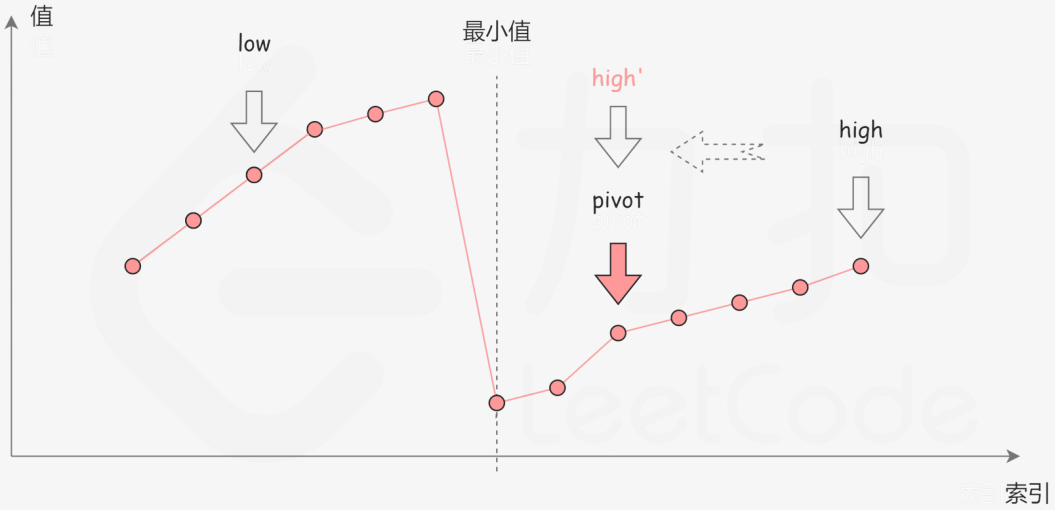


其中横轴表示数组元素的下标，纵轴表示数组元素的值。图中标出了最小值的位置，是我们需要查找的目标。

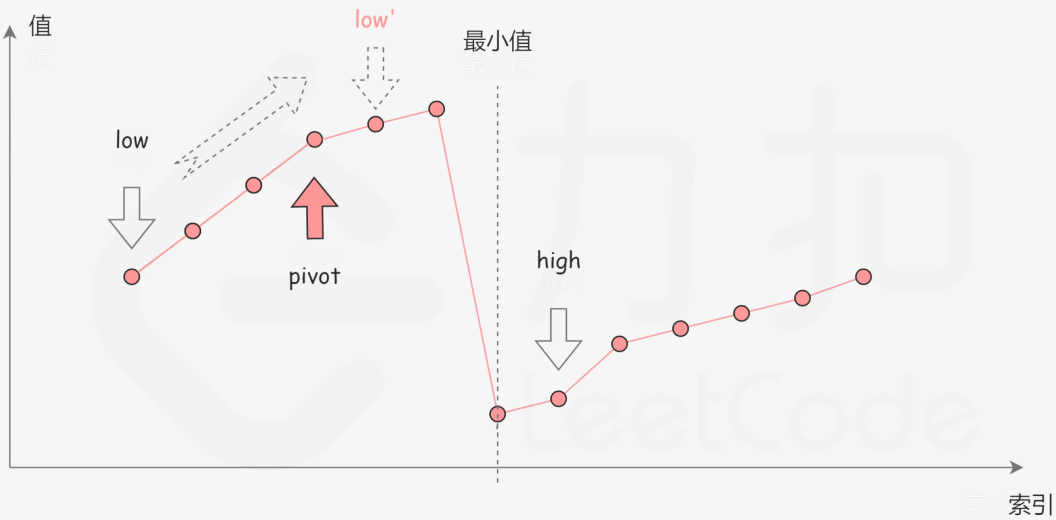
我们考虑数组中的最后一个元素 xx：在最小值右侧的元素（不包括最后一个元素本身），它们的值一定都严格小于 xx；而在最小值左侧的元素，它们的值一定都严格大于 xx。因此，我们可以根据这一条性质，通过二分查找的方法找出最小值。

在二分查找的每一步中，左边界为 \it lowlow，右边界为 \it highhigh，区间的中点为 \it pivotpivot，最小值就在该区间内。我们将中轴元素 \textit{nums}[\textit{pivot}]nums[pivot] 与右边界元素 \textit{nums}[\textit{high}]nums[high] 进行比较，可能会有以下的三种情况：

第一种情况是 \textit{nums}[\textit{pivot}] < \textit{nums}[\textit{high}]nums[pivot]<nums[high]。如下图所示，这说明 \textit{nums}[\textit{pivot}]nums[pivot] 是最小值右侧的元素，因此我们可以忽略二分查找区间的右半部分。



第二种情况是 \textit{nums}[\textit{pivot}] > \textit{nums}[\textit{high}]nums[pivot]>nums[high]。如下图所示，这说明 \textit{nums}[\textit{pivot}]nums[pivot] 是最小值左侧的元素，因此我们可以忽略二分查找区间的左半部分。



由于数组不包含重复元素，并且只要当前的区间长度不为 11，\it pivotpivot 就不会与 \it highhigh 重合；而如果当前的区间长度为 11，这说明我们已经可以结束二分查找了。因此不会存在 \textit{nums}[\textit{pivot}] = \textit{nums}[\textit{high}]nums[pivot]=nums[high] 的情况。

当二分查找结束时，我们就得到了最小值所在的位置。

```
class Solution {
public:
    int findMin(vector<int>& nums) {
        int low = 0;
        int high = nums.size() - 1;
        while (low < high) {
            int pivot = low + (high - low) / 2;
            if (nums[pivot] < nums[high]) {
                high = pivot;
            }
            else {
                low = pivot + 1;
```

```
            }
        }
        return nums[low];
    }
};
```

复杂度分析

时间复杂度：时间复杂度为 O(\log n)O(logn)，其中 nn 是数组 \textit{nums}nums 的长度。在二分查找的过程中，每一步会忽略一半的区间，因此时间复杂度为 O(\log n)O(logn)。

空间复杂度：O(1)O(1)。

81. 搜索旋转排序数组 II

前言
本篇题解基于「33. 搜索旋转排序数组的官方题解」，请读者在阅读完该题解后再继续阅读本篇题解。

方法一：二分查找
思路

对于数组中有重复元素的情况，二分查找时可能会有 a[l]=a[\textit{mid}]=a[r]a[l]=a[mid]=a[r]，此时无法判断区间 [l,\textit{mid}][l,mid] 和区间 [\textit{mid}+1,r][mid+1,r] 哪个是有序的。

例如 \textit{nums}=[3,1,2,3,3,3,3]nums=[3,1,2,3,3,3,3]，\textit{target}=2target=2，首次二分时无法判断区间 [0,3][0,3] 和区间 [4,6][4,6] 哪个是有序的。

对于这种情况，我们只能将当前二分区间的左边界加一，右边界减一，然后在新区间上继续二分查找。

代码

```
class Solution {
public:
    bool search(vector<int> &nums, int target) {
        int n = nums.size();
        if (n == 0) {
            return false;
        }
        if (n == 1) {
            return nums[0] == target;
        }
        int l = 0, r = n - 1;
        while (l <= r) {
            int mid = (l + r) / 2;
            if (nums[mid] == target) {
                return true;
            }
            if (nums[l] == nums[mid] && nums[mid] == nums[r]) {
                ++l;
                --r;
            } else if (nums[l] <= nums[mid]) {
                if (nums[l] <= target && target < nums[mid]) {
                    r = mid - 1;
                } else {
                    l = mid + 1;
                }
            } else {
                if (nums[mid] < target && target <= nums[n - 1]) {
                    l = mid + 1;
                } else {
                    r = mid - 1;
```

```
                }
            }
        }
        return false;
    }
};
```

复杂度分析

时间复杂度：O(n)O(n)，其中 nn 是数组 \textit{nums}nums 的长度。最坏情况下数组元素均相等且不为 \textit{target}target，我们需要访问所有位置才能得出结果。

空间复杂度：O(1)O(1)。

[236. 二叉树的最近公共祖先]

128. 最长连续序列

方法一：哈希表
思路和算法

我们考虑枚举数组中的每个数 xx，考虑以其为起点，不断尝试匹配 x+1, x+2, \cdotsx+1,x+2,⋯ 是否存在，假设最长匹配到了 x+yx+y，那么以 xx 为起点的最长连续序列即为 x, x+1, x+2, \cdots, x+yx,x+1,x+2,⋯,x+y，其长度为 y+1y+1，我们不断枚举并更新答案即可。

对于匹配的过程，暴力的方法是 O(n)O(n) 遍历数组去看是否存在这个数，但其实更高效的方法是用一个哈希表存储数组中的数，这样查看一个数是否存在即能优化至 O(1)O(1) 的时间复杂度。

仅仅是这样我们的算法时间复杂度最坏情况下还是会达到 O(n^2)O(n
2
)（即外层需要枚举 O(n)O(n) 个数，内层需要暴力匹配 O(n)O(n) 次），无法满足题目的要求。但仔细分析这个过程，我们会发现其中执行了很多不必要的枚举，如果已知有一个 x, x+1, x+2, \cdots, x+yx,x+1,x+2,⋯,x+y 的连续序列，而我们却重新从 x+1x+1，x+2x+2 或者是 x+yx+y 处开始尝试匹配，那么得到的结果肯定不会优于枚举 xx 为起点的答案，因此我们在外层循环的时候碰到这种情况跳过即可。

那么怎么判断是否跳过呢？由于我们要枚举的数 xx 一定是在数组中不存在前驱数 x-1x−1 的，不然按照上面的分析我们会从 x-1x−1 开始尝试匹配，因此我们每次在哈希表中检查是否存在 x-1x−1 即能判断是否需要跳过了。

增加了判断跳过的逻辑之后，时间复杂度是多少呢？外层循环需要 O(n)O(n) 的时间复杂度，只有当一个数是连续序列的第一个数的情况下才会进入内层循环，然后在内层循环中匹配连续序列中的数，因此数组中的每个数只会进入内层循环一次。根据上述分析可知，总时间复杂度为 O(n)O(n)，符合题目要求。

```
class Solution {
    public int longestConsecutive(int[] nums) {
        Set<Integer> num_set = new HashSet<Integer>();
        for (int num : nums) {
            num_set.add(num);
        }

        int longestStreak = 0;

        for (int num : num_set) {
            if (!num_set.contains(num - 1)) {
                int currentNum = num;
                int currentStreak = 1;

                while (num_set.contains(currentNum + 1)) {
                    currentNum += 1;
```

```
                    currentStreak += 1;
                }

                longestStreak = Math.max(longestStreak, currentStreak);
            }
        }

        return longestStreak;
    }

}
```

复杂度分析

时间复杂度：O(n)O(n)，其中 nn 为数组的长度。具体分析已在上面正文中给出。

空间复杂度：O(n)O(n)。哈希表存储数组中所有的数需要 O(n)O(n) 的空间。

53. 最大子序和

思路
这道题用动态规划的思路并不难解决，比较难的是后文提出的用分治法求解，但由于其不是最优解法，所以先不列出来
动态规划的是首先对数组进行遍历，当前最大连续子序列和为 sum，结果为 ans
如果 sum > 0，则说明 sum 对结果有增益效果，则 sum 保留并加上当前遍历数字
如果 sum <= 0，则说明 sum 对结果无增益效果，需要舍弃，则 sum 直接更新为当前遍历数字
每次比较 sum 和 ans的大小，将最大值置为ans，遍历结束返回结果
时间复杂度：O(n)O(n)

代码

```
class Solution {
    public int maxSubArray(int[] nums) {
        int ans = nums[0];
        int sum = 0;
        for(int num: nums) {
            if(sum > 0) {
                sum += num;
            } else {
                sum = num;
            }
            ans = Math.max(ans, sum);
        }
        return ans;
    }
}
```

想看大鹏画解更多高频面试题，欢迎阅读大鹏的 LeetBook：《画解剑指 Offer 》，O(∩_∩)O

给定一个 foo 函数，60%的概率返回0，40%的概率返回1，如何利用 foo 函数实现一个 50% 返回 0 的函数?

64 匹马，8 个赛道，找出前 4 匹马最少需要比几次

112. 路径总和

这个题要背下来！！

DFS

首先是 DFS 解法，该解法的想法是一直向下找到**叶子节点**，如果到**叶子节点**时`sum == 0`，说明找到了一条符合要求的路径。

我自己第一遍做的时候犯了一个错误，把递归函数写成了下面的解法：

```
def hasPathSum(self, root: TreeNode, sum: int) -> bool:
    if not root:
        return sum == 0
    return self.hasPathSum(root.left, sum - root.val) or self.hasPathSum(root.right, sum - root.val)
```

这种代码的错误在，**没有判断 root 是否为叶子节点。**比如 root 为空的话，题目的意思是要返回 False 的，而上面的代码会返回`sum == 0`。又比如，对于测试用例 树为`[1,2]`，`sum = 0`时，上面的结果也会返回为 True，因为对于上述代码，只要左右任意一个孩子的为空时`sum == 0`就返回 True 。

当题目中提到了**叶子节点**时，正确的做法一定要同时判断节点的**左右子树同时为空**才是叶子节点。

Python 代码如下：

- Python
- Java

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def hasPathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: bool
        """
        if not root: return False
        if not root.left and not root.right:
            return sum == root.val
        return self.hasPathSum(root.left, sum - root.val) or self.hasPathSum(root.right, sum - root.val)
```

○

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
        if(root == null){
            return false;
        }
        if(root.left == null && root.right == null){
            return root.val == sum;
        }
        return hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, sum - root.val);

    }
}
```

回溯

这里的回溯指 利用 DFS 找出从根节点到叶子节点的所有路径，只要有任意一条路径的 和 等于 sum，就返回 True。

下面的代码并非是严格意义上的回溯法，因为没有重复利用 path 变量。

Python 代码如下：

- Python

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def hasPathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: bool
        """
        if not root: return False
        res = []
        return self.dfs(root, sum, res, [root.val])
```

```
<span class="hljs-function"><span class="hljs-keyword">def</span> <span class="hljs-title">dfs</span><span class="hljs-params">(self, root, target, res, path)</span>:</span>
    <span class="hljs-keyword">if</span> <span class="hljs-keyword">not</span> root: <span class="hljs-keyword">return</span> <span class="hljs-literal">False</span>
    <span class="hljs-keyword">if</span> sum(path) == target <span class="hljs-keyword">and</span> <span class="hljs-keyword">not</span> root.left <span class="hljs-keyword">and</span> <span class="hljs-keyword">not</span> root.right:
        <span class="hljs-keyword">return</span> <span class="hljs-literal">True</span>
    left_flag, right_flag = <span class="hljs-literal">False</span>, <span class="hljs-literal">False</span>
    <span class="hljs-keyword">if</span> root.left:
        left_flag = self.dfs(root.left, target, res, path + [root.left.val])
    <span class="hljs-keyword">if</span> root.right:
        right_flag = self.dfs(root.right, target, res, path + [root.right.val])
    <span class="hljs-keyword">return</span> left_flag <span class="hljs-keyword">or</span> right_flag
```

BFS

BFS 使用 **队列** 保存遍历到每个节点时的**路径和**，如果该节点恰好是叶子节点，并且 路径和 正好等于 sum，说明找到了解。

Python 代码如下：

- Python

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

class Solution:
    def hasPathSum(self, root: TreeNode, sum: int) -> bool:
        if not root:
            return False
        que = collections.deque()
        que.append((root, root.val))
        while que:
            node, path = que.popleft()
            if not node.left and not node.right and path == sum:
                return True
            if node.left:
                que.append((node.left, path + node.left.val))
            if node.right:
                que.append((node.right, path + node.right.val))
        return False
```

栈

除了上面的 队列 解法以外，也可以使用 **栈**，同时保存节点和到这个节点的路径和。但是这个解法已经不是 BFS。因为会优先访问 后进来 的节点，导致会把根节点的右子树访问结束之后，才访问左子树。

可能会有朋友好奇很少见到这种写法，为什么代码可行？答案是：栈中同时保存了 `(节点，路径和)`，也就是说只要能把所有的节点访问一遍，那么就一定能找到正确的结果。无论是用 队列 还是 栈，都是一种 树的遍历 方式，只不过访问顺序有所有不同罢了。

Python 代码如下：

- Python

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
class Solution(object):
    def hasPathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: bool
        """
        if not root:
            return False
        stack = []
        stack.append((root, root.val))
        while stack:
            node, path = stack.pop()
            if not node.left and not node.right and path == sum:
                return True
            if node.left:
                stack.append((node.left, path + node.left.val))
            if node.right:
                stack.append((node.right, path + node.right.val))
        return False
```

300. 最长递增子序列

方法一：动态规划
思路与算法

定义 \textit{dp}[i]dp[i] 为考虑前 ii 个元素，以第 ii 个数字结尾的最长上升子序列的长度，注意 \textit{nums}[i]nums[i] 必须被选取。

我们从小到大计算 \textit{dp}dp 数组的值，在计算 \textit{dp}[i]dp[i] 之前，我们已经计算出 \textit{dp}[0 \ldots i-1]dp[0…i−1] 的值，则状态转移方程为：

\textit{dp}[i] = \max(\textit{dp}[j]) + 1, \text{其中} \, 0 \leq j < i \, \text{且} \, \textit{num}[j]<\textit{num}[i]
dp[i]=max(dp[j])+1,其中0≤j<i且num[j]<num[i]

即考虑往 \textit{dp}[0 \ldots i-1]dp[0…i−1] 中最长的上升子序列后面再加一个 \textit{nums}[i]nums[i]。由于 \textit{dp}[j]dp[j] 代表 \textit{nums}[0 \ldots j]nums[0…j] 中以 \textit{nums}[j]nums[j] 结尾的最长上升子序列，所以如果能从 \textit{dp}[j]dp[j] 这个状态转移过来，那么 \textit{nums}[i]nums[i] 必然要大于 \textit{nums}[j]nums[j]，才能将 \textit{nums}[i]nums[i] 放在 \textit{nums}[j]nums[j] 后面以形成更长的上升子序列。

最后，整个数组的最长上升子序列即所有 \textit{dp}[i]dp[i] 中的最大值。

\text{LIS}_{\textit{length}}= \max(\textit{dp}[i]), \text{其中} \, 0\leq i < n
LIS
length

=max(dp[i]),其中0≤i<n

以下动画演示了该方法：

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        if (nums.length == 0) {
            return 0;
        }
        int[] dp = new int[nums.length];
        dp[0] = 1;
        int maxans = 1;
        for (int i = 1; i < nums.length; i++) {
            dp[i] = 1;
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
            maxans = Math.max(maxans, dp[i]);
        }
        return maxans;
    }
}
```

复杂度分析

时间复杂度：O(n^2)O(n
2
)，其中 nn 为数组 \textit{nums}nums 的长度。动态规划的状态数为 nn，计算状态 dp[i]dp[i] 时，需要 O(n)O(n) 的时间遍历 dp[0 \ldots i-1]dp[0…i−1] 的所有状态，所以总时间复杂度为 O(n^2)O(n
2
)。

空间复杂度：O(n)O(n)，需要额外使用长度为 nn 的 dpdp 数组。

方法二：贪心 + 二分查找
思路与算法

考虑一个简单的贪心，如果我们要使上升子序列尽可能的长，则我们需要让序列上升得尽可能慢，因此我们希望每次在上升子序列最后加上的那个数尽可能的小。

基于上面的贪心思路，我们维护一个数组 d[i]d[i] ，表示长度为 ii 的最长上升子序列的末尾元素的最小值，用 \textit{len}len 记录目前最长上升子序列的长度，起始时 lenlen 为 11，d[1] = \textit{nums}[0]d[1]=nums[0]。

同时我们可以注意到 d[i]d[i] 是关于 ii 单调递增的。因为如果 d[j] \geq d[i]d[j]≥d[i] 且 j < ij<i，我们考虑从长度为 ii 的最长上升子序列的末尾删除 i-ji−j 个元素，那么这个序列长度变为 jj ，且第 jj 个元素 xx（末尾元素）必然小于 d[i]d[i]，也就小于 d[j]d[j]。那么我们就找到了一个长度为 jj 的最长上升子序列，并且末尾元素比 d[j]d[j] 小，从而产生了矛盾。因此数组 dd 的单调性得证。

我们依次遍历数组 \textit{nums}nums 中的每个元素，并更新数组 dd 和 lenlen 的值。如果 \textit{nums}[i] > d[\textit{len}]nums[i]>d[len] 则更新 len = len + 1len=len+1，否则在 d[1 \ldots len]d[1…len]中找满足 d[i - 1] < \textit{nums}[j] < d[i]d[i−1]<nums[j]<d[i] 的下标 ii，并更新 d[i] = \textit{nums}[j]d[i]=nums[j]。

根据 dd 数组的单调性，我们可以使用二分查找寻找下标 ii，优化时间复杂度。

最后整个算法流程为：

设当前已求出的最长上升子序列的长度为 \textit{len}len（初始时为 11），从前往后遍历数组 \textit{nums}nums，在遍历到 \textit{nums}[i]nums[i] 时：

如果 \textit{nums}[i] > d[\textit{len}]nums[i]>d[len]，则直接加入到 dd 数组末尾，并更新 \textit{len} = \textit{len} + 1len=len+1；

否则，在 dd 数组中二分查找，找到第一个比 \textit{nums}[i]nums[i] 小的数 d[k]d[k]，并更新 d[k + 1] = \textit{nums}[i]d[k+1]=nums[i]。

以输入序列 [0, 8, 4, 12, 2][0,8,4,12,2] 为例：

第一步插入 00，d = [0]d=[0]；

第二步插入 88，d = [0, 8]d=[0,8]；

第三步插入 44，d = [0, 4]d=[0,4]；

第四步插入 1212，d = [0, 4, 12]d=[0,4,12]；

第五步插入 22，d = [0, 2, 12]d=[0,2,12]。

最终得到最大递增子序列长度为 33。

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        int len = 1, n = nums.length;
        if (n == 0) {
            return 0;
        }
        int[] d = new int[n + 1];
        d[len] = nums[0];
        for (int i = 1; i < n; ++i) {
            if (nums[i] > d[len]) {
                d[++len] = nums[i];
            } else {
                int l = 1, r = len, pos = 0; // 如果找不到说明所有的数都比 nums[i] 大，此时要更新 d[1]，所以这里将 pos 设为 0
                while (l <= r) {
                    int mid = (l + r) >> 1;
                    if (d[mid] < nums[i]) {
                        pos = mid;
                        l = mid + 1;
                    } else {
                        r = mid - 1;
                    }
                }
                d[pos + 1] = nums[i];
            }
        }
        return len;
    }
}
```

复杂度分析

时间复杂度：O(n\log n)O(nlogn)。数组 \textit{nums}nums 的长度为 nn，我们依次用数组中的元素去更新 dd 数组，而更新 dd 数组时需要进行 O(\log n)O(logn) 的二分搜索，所以总时间复杂度为 O(n\log n)O(nlogn)。

空间复杂度：O(n)O(n)，需要额外使用长度为 nn 的 dd 数组。

300. 最长递增子序列

方法一：动态规划
思路与算法

定义 \textit{dp}[i]dp[i] 为考虑前 ii 个元素，以第 ii 个数字结尾的最长上升子序列的长度，注意 \textit{nums}[i]nums[i] 必须被选取。

我们从小到大计算 \textit{dp}dp 数组的值，在计算 \textit{dp}[i]dp[i] 之前，我们已经计算出 \textit{dp}[0 \ldots i-1]dp[0…i−1] 的值，则状态转移方程为：

\textit{dp}[i] = \max(\textit{dp}[j]) + 1, \text{其中} \, 0 \leq j < i \, \text{且} \, \textit{num}[j]<\textit{num}[i]
dp[i]=max(dp[j])+1,其中0≤j<i且num[j]<num[i]

即考虑往 \textit{dp}[0 \ldots i-1]dp[0…i−1] 中最长的上升子序列后面再加一个 \textit{nums}[i]nums[i]。由于 \textit{dp}[j]dp[j] 代表 \textit{nums}[0 \ldots j]nums[0…j] 中以 \textit{nums}[j]nums[j] 结尾的最长上升子序列，所以如果能从 \textit{dp}[j]dp[j] 这个状态转移过来，那么 \textit{nums}[i]nums[i] 必然要大于 \textit{nums}[j]nums[j]，才能将 \textit{nums}[i]nums[i] 放在 \textit{nums}[j]nums[j] 后面以形成更长的上升子序列。

最后，整个数组的最长上升子序列即所有 \textit{dp}[i]dp[i] 中的最大值。

\text{LIS}_{\textit{length}}= \max(\textit{dp}[i]), \text{其中} \, 0\leq i < n
LIS
length

=max(dp[i]),其中0≤i<n

以下动画演示了该方法：

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        if (nums.length == 0) {
            return 0;
        }
        int[] dp = new int[nums.length];
        dp[0] = 1;
        int maxans = 1;
        for (int i = 1; i < nums.length; i++) {
            dp[i] = 1;
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
            maxans = Math.max(maxans, dp[i]);
        }
        return maxans;
    }
}
```

复杂度分析

时间复杂度：O(n^2)O(n
2
)，其中 nn 为数组 \textit{nums}nums 的长度。动态规划的状态数为 nn，计算状态 dp[i]dp[i] 时，需要 O(n)O(n) 的时间遍历 dp[0 \ldots i-1]dp[0…i−1] 的所有状态，所以总时间复杂度为 O(n^2)O(n

2
)。

空间复杂度：O(n)O(n)，需要额外使用长度为 nn 的 dpdp 数组。

方法二：贪心 + 二分查找
思路与算法

考虑一个简单的贪心，如果我们要使上升子序列尽可能的长，则我们需要让序列上升得尽可能慢，因此我们希望每次在上升子序列最后加上的那个数尽可能的小。

基于上面的贪心思路，我们维护一个数组 d[i]d[i] ，表示长度为 ii 的最长上升子序列的末尾元素的最小值，用 \textit{len}len 记录目前最长上升子序列的长度，起始时 lenlen 为 11，d[1] = \textit{nums}[0]d[1]=nums[0]。

同时我们可以注意到 d[i]d[i] 是关于 ii 单调递增的。因为如果 d[j] \geq d[i]d[j]≥d[i] 且 j < ij<i，我们考虑从长度为 ii 的最长上升子序列的末尾删除 i-ji−j 个元素，那么这个序列长度变为 jj ，且第 jj 个元素 xx（末尾元素）必然小于 d[i]d[i]，也就小于 d[j]d[j]。那么我们就找到了一个长度为 jj 的最长上升子序列，并且末尾元素比 d[j]d[j] 小，从而产生了矛盾。因此数组 dd 的单调性得证。

我们依次遍历数组 \textit{nums}nums 中的每个元素，并更新数组 dd 和 lenlen 的值。如果 \textit{nums}[i] > d[\textit{len}]nums[i]>d[len] 则更新 len = len + 1len=len+1，否则在 d[1 \ldots len]d[1…len]中找满足 d[i - 1] < \textit{nums}[j] < d[i]d[i−1]<nums[j]<d[i] 的下标 ii，并更新 d[i] = \textit{nums}[j]d[i]=nums[j]。

根据 dd 数组的单调性，我们可以使用二分查找寻找下标 ii，优化时间复杂度。

最后整个算法流程为：

设当前已求出的最长上升子序列的长度为 \textit{len}len（初始时为 11），从前往后遍历数组 \textit{nums}nums，在遍历到 \textit{nums}[i]nums[i] 时：

如果 \textit{nums}[i] > d[\textit{len}]nums[i]>d[len] ，则直接加入到 dd 数组末尾，并更新 \textit{len} = \textit{len} + 1len=len+1；

否则，在 dd 数组中二分查找，找到第一个比 \textit{nums}[i]nums[i] 小的数 d[k]d[k] ，并更新 d[k + 1] = \textit{nums}[i]d[k+1]=nums[i]。

以输入序列 [0, 8, 4, 12, 2][0,8,4,12,2] 为例：

第一步插入 00，d = [0]d=[0]；

第二步插入 88，d = [0, 8]d=[0,8]；

第三步插入 44，d = [0, 4]d=[0,4]；

第四步插入 1212，d = [0, 4, 12]d=[0,4,12]；

第五步插入 22，d = [0, 2, 12]d=[0,2,12]。

最终得到最大递增子序列长度为 33。

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        int len = 1, n = nums.length;
        if (n == 0) {
            return 0;
        }
        int[] d = new int[n + 1];
        d[len] = nums[0];
        for (int i = 1; i < n; ++i) {
            if (nums[i] > d[len]) {
                d[++len] = nums[i];
            } else {
```

```
                int l = 1, r = len, pos = 0; // 如果找不到说明所有的数都比 nums[i] 大，此时要更新
d[1]，所以这里将 pos 设为 0
                while (l <= r) {
                    int mid = (l + r) >> 1;
                    if (d[mid] < nums[i]) {
                        pos = mid;
                        l = mid + 1;
                    } else {
                        r = mid - 1;
                    }
                }
                d[pos + 1] = nums[i];
            }
        }
        return len;
    }
}
```

复杂度分析

时间复杂度：O(n\log n)O(nlogn)。数组 \textit{nums}nums 的长度为 nn，我们依次用数组中的元素去更新 dd 数组，而更新 dd 数组时需要进行 O(\log n)O(logn) 的二分搜索，所以总时间复杂度为 O(n\log n)O(nlogn)。

空间复杂度：O(n)O(n)，需要额外使用长度为 nn 的 dd 数组。

141. 环形链表

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)O(N)，其中 NN 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)O(N)，其中 NN 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。

[300. 最长递增子序列]

方法一：动态规划
思路与算法

定义 \textit{dp}[i]dp[i] 为考虑前 ii 个元素，以第 ii 个数字结尾的最长上升子序列的长度，注意 \textit{nums}[i]nums[i] 必须被选取。

我们从小到大计算 \textit{dp}dp 数组的值，在计算 \textit{dp}[i]dp[i] 之前，我们已经计算出 \textit{dp}[0 \ldots i-1]dp[0…i−1] 的值，则状态转移方程为：

\textit{dp}[i] = \max(\textit{dp}[j]) + 1, \text{其中} \, 0 \leq j < i \, \text{且} \, \textit{num}[j]<\textit{num}[i]
dp[i]=max(dp[j])+1,其中0≤j<i且num[j]<num[i]

即考虑往 \textit{dp}[0 \ldots i-1]dp[0…i−1] 中最长的上升子序列后面再加一个 \textit{nums}[i]nums[i]。由于 \textit{dp}[j]dp[j] 代表 \textit{nums}[0 \ldots j]nums[0…j] 中以 \textit{nums}[j]nums[j] 结尾的最长上升子序列，所以如果能从 \textit{dp}[j]dp[j] 这个状态转移过来，那么 \textit{nums}[i]nums[i] 必然要大于 \textit{nums}[j]nums[j]，才能将 \textit{nums}[i]nums[i] 放在 \textit{nums}[j]nums[j] 后面以形成更长的上升子序列。

最后，整个数组的最长上升子序列即所有 \textit{dp}[i]dp[i] 中的最大值。

\text{LIS}_{\textit{length}}= \max(\textit{dp}[i]), \text{其中} \, 0\leq i < n
LIS
length

=max(dp[i]),其中0≤i<n

以下动画演示了该方法：

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        if (nums.length == 0) {
            return 0;
        }
        int[] dp = new int[nums.length];
        dp[0] = 1;
        int maxans = 1;
        for (int i = 1; i < nums.length; i++) {
            dp[i] = 1;
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
            maxans = Math.max(maxans, dp[i]);
        }
        return maxans;
    }
}
```

复杂度分析

时间复杂度：O(n^2)O(n
2
)，其中 nn 为数组 \textit{nums}nums 的长度。动态规划的状态数为 nn，计算状态 dp[i]dp[i] 时，需要 O(n)O(n) 的时间遍历 dp[0 \ldots i-1]dp[0…i−1] 的所有状态，所以总时间复杂度为 O(n^2)O(n
2
)。

空间复杂度：O(n)O(n)，需要额外使用长度为 nn 的 dpdp 数组。

方法二：贪心 + 二分查找
思路与算法

考虑一个简单的贪心，如果我们要使上升子序列尽可能的长，则我们需要让序列上升得尽可能慢，因此我们希望每次在上升子序列最后加上的那个数尽可能的小。

基于上面的贪心思路，我们维护一个数组 d[i]d[i] ，表示长度为 ii 的最长上升子序列的末尾元素的最小值，用 \textit{len}len 记录目前最长上升子序列的长度，起始时 lenlen 为 11，d[1] = \textit{nums}[0]d[1]=nums[0]。

同时我们可以注意到 d[i]d[i] 是关于 ii 单调递增的。因为如果 d[j] \geq d[i]d[j]≥d[i] 且 j < ij<i，我们考虑从长度为 ii 的最长上升子序列的末尾删除 i-ji−j 个元素，那么这个序列长度变为 jj ，且第 jj 个元素 xx（末尾元素）必然小于 d[i]d[i]，也就小于 d[j]d[j]。那么我们就找到了一个长度为 jj 的最长上升子序列，并且末尾元素比 d[j]d[j] 小，从而产生了矛盾。因此数组 dd 的单调性得证。

我们依次遍历数组 \textit{nums}nums 中的每个元素，并更新数组 dd 和 lenlen 的值。如果 \textit{nums}[i] > d[\textit{len}]nums[i]>d[len] 则更新 len = len + 1len=len+1，否则在 d[1 \ldots len]d[1…len]中找满足 d[i - 1] < \textit{nums}[j] < d[i]d[i−1]<nums[j]<d[i] 的下标 ii，并更新 d[i] = \textit{nums}[j]d[i]=nums[j]。

根据 dd 数组的单调性，我们可以使用二分查找寻找下标 ii，优化时间复杂度。

最后整个算法流程为：

设当前已求出的最长上升子序列的长度为 \textit{len}len（初始时为 11），从前往后遍历数组 \textit{nums}nums，在遍历到 \textit{nums}[i]nums[i] 时：

如果 \textit{nums}[i] > d[\textit{len}]nums[i]>d[len] ，则直接加入到 dd 数组末尾，并更新 \textit{len} = \textit{len} + 1len=len+1；

否则，在 dd 数组中二分查找，找到第一个比 \textit{nums}[i]nums[i] 小的数 d[k]d[k] ，并更新 d[k + 1] = \textit{nums}[i]d[k+1]=nums[i]。

以输入序列 [0, 8, 4, 12, 2][0,8,4,12,2] 为例：

第一步插入 00，d = [0]d=[0]；

第二步插入 88，d = [0, 8]d=[0,8]；

第三步插入 44，d = [0, 4]d=[0,4]；

第四步插入 1212，d = [0, 4, 12]d=[0,4,12]；

第五步插入 22，d = [0, 2, 12]d=[0,2,12]。

最终得到最大递增子序列长度为 33。

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        int len = 1, n = nums.length;
        if (n == 0) {
            return 0;
        }
        int[] d = new int[n + 1];
        d[len] = nums[0];
        for (int i = 1; i < n; ++i) {
            if (nums[i] > d[len]) {
                d[++len] = nums[i];
            } else {
                int l = 1, r = len, pos = 0; // 如果找不到说明所有的数都比 nums[i] 大，此时要更新 d[1]，所以这里将 pos 设为 0
                while (l <= r) {
                    int mid = (l + r) >> 1;
                    if (d[mid] < nums[i]) {
                        pos = mid;
```

```
                    l = mid + 1;
                } else {
                    r = mid - 1;
                }
            }
            d[pos + 1] = nums[i];
        }
    }
    return len;
  }
}
```

复杂度分析

- 时间复杂度：O(n\log n)O(nlogn)。数组 \textit{nums}nums 的长度为 nn，我们依次用数组中的元素去更新 dd 数组，而更新 dd 数组时需要进行 O(\log n)O(logn) 的二分搜索，所以总时间复杂度为 O(n\log n)O(nlogn)。
- 空间复杂度：O(n)O(n)，需要额外使用长度为 nn 的 dd 数组。