<https://mp.weixin.qq.com/s?__biz=MzAwOTE3NDY5OA==&mid=2647911761&idx=3&sn=e9ef198de5a1de23394caf3080279b2c&chksm=8344e694b4336f82799d76437a3c803572459378e9a7a80e589204c6aa18c0874e1a6563aa92&scene=21#wechat_redirect>

# Java 集合高频要点问题

## ****List****

### ****1. 为什么 arraylist 不安全？****

我们查看源码发现 arraylist 的 CRUD 操作，并没有涉及到锁之类的东西。底层是数组，初始大小为 10。插入时会判断数组容量是否足够，不够的话会进行扩容。所谓扩容就是新建一个新的数组，然后将老的数据里面的元素复制到新的数组里面(所以增加较慢)。

### ****2. CopyOnWriteArrayList 有什么特点？****

它是 List 接口的一个实现类，在 java.util.concurrent（简称 JUC，后面我全部改成 juc，大家注意下）。

内部持有一个 ReentrantLock lock = new ReentrantLock(); 对于增删改操作都是先加锁再释放锁，线程安全。并且锁只有一把，而读操作不需要获得锁，支持并发。

读写分离，写时复制出一个新的数组，完成插入、修改或者移除操作后将新数组赋值给 array。

### ****3. CopyOnWriteArrayList 与 Vector 的选择？****

Vector 是增删改查方法都加了 synchronized，保证同步，但是每个方法执行的时候都要去获得锁，性能就会大大下降，而 CopyOnWriteArrayList 只是在增删改上加锁，但是读不加锁，在读方面的性能就好于 Vector，CopyOnWriteArrayList 支持读多写少的并发情况。

Vector 和 CopyOnWriteArrayList 都是 List 接口的一个实现类。

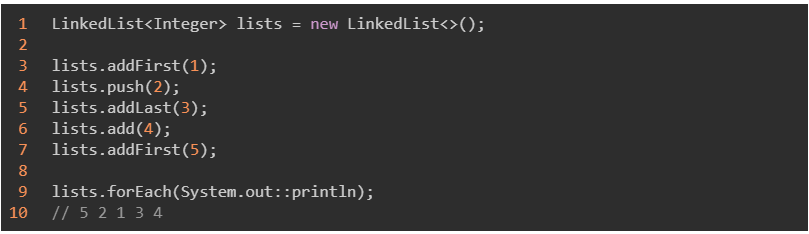
### ****4. CopyOnWriteArrayList 适用于什么情况？****

我们看源码不难发现他每次增加一个元素都要进行一次拷贝，此时严重影响了增删改的性能，其中和 arraylist 差了好几百倍。

所以对于读多写少的操作 CopyOnWriteArrayList 更加适合，而且线程安全。

DriverManager 这个类就使用到了CopyOnWriteArrayList。

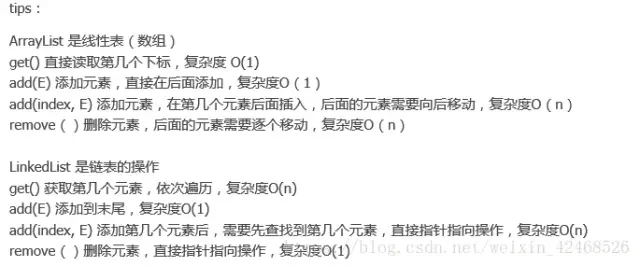
### ****5. LinkedList  和 ArrayList  对比？****



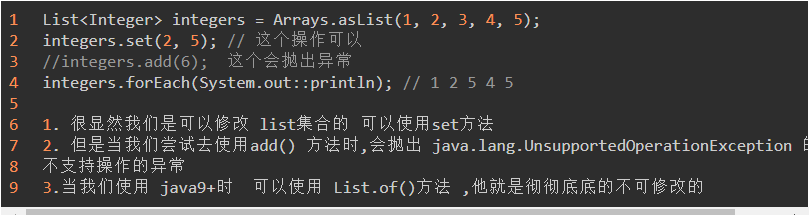
addFirst 和 addLast 方法很清楚。push 方法默认是 addFirst 实现。add 方法默认是 addLast 实现。所以总结一下就是 add 和 last，push 和 first。

其实我们要明白一下，链表相对于数组来说，链表的添加和删除速度很快，是顺序添加删除很快，因为一个 linkedList 会保存第一个节点和最后一个节点，时间复杂度为O(1)，但是你要指定位置添加 add(int index, E element) ，那么此时他会先遍历，然后找到改位置的节点，将你的节点添加到他前面，此时时间复杂度最大值为 O(n)。

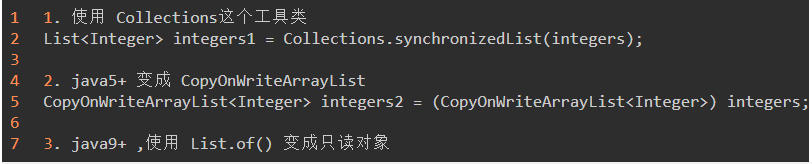
数组呢？我们知道 ArrayList 底层实现就是数组，数组优点就是由于内存地址是顺序的，属于一块整的，此时遍历起来很快，添加删除的话，他会复制数组，当数组长度特别大时所消耗的时间会很长。这是一张图，大家可以看一下：



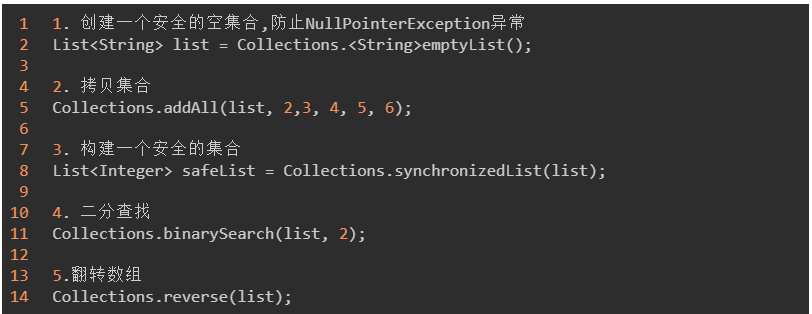
### ****6. Arrays.asList() 方法返回的数组是不可变得吗？****



### ****7. 怎么将一个不安全数组换成安全数组？****



### ****8. Collections 工具类？****



## ****Set****

### ****1. HashSet、TreeSet 和 LinkedHashSet 三种类型什么时候使用它们？****

如你的需求是要一个能快速访问的 Set，那么就要用 HashSet，HashSet 底层是 HashMap 实现的，其中的元素没有按顺序排列。

如果你要一个可排序 Set，那么你应该用 TreeSet，TreeSet 的底层实现是 TreeMap。

如果你要记录下插入时的顺序时，你应该使用 LinkedHashSet。

Set 集合中不能包含重复的元素，每个元素必须是唯一的，你只要将元素加入 set 中，重复的元素会自动移除。所以可以去重，很多情况下都需要使用（但是去重方式不同）。

LinkedHashSet 正好介于 HashSet 和 TreeSet 之间，它也是一个基于 HashMap 和双向链表的集合，但它同时维护了一个双链表来记录插入的顺序，基本方法的复杂度为 O(1)。

三者都是线程不安全的，需要使用 Collections.synchronizedSet(new HashSet(…));。

### ****2. HashSet 和 LinkedHashSet 判定元素重复的原则是相同的？****

会先去执行 hashCode() 方法，判断是否重复。如果 hashCode() 返回值相同，就会去判断 equals 方法。如果 equals() 方法还是相同，那么就认为重复。

### ****3. TreeSet 判断元素重复原则？****

TreeSet 的元素必须是实现了 java.lang.Comparable<T> 接口，所以他是根据此个接口的方法 compareTo 方法进行判断重复的，当返回值一样的时认定重复。

### ****4. 怎么实现一个线程安全的 hashset？****

我们看源码会发现他里面有一个 HashMap（用 transient 关键字标记的成员变量不参与序列化过程，因为 HashMap 已经实现 Serializable）。

**5. CopyOnWriteArraySet 的实现？**



很显然翻源码我们发现他实现了 CopyOnWriteArrayList()。

## ****Map****

### ****1. Hashtable 特点？****

Hashtable 和 ConcurrentHashMap 以及 ConcurrentSkipListMap 以及 TreeMap 不允许 key 和 value 值为空，但是 HashMap 可以 key 和 value 值都可以为空。

Hashtable 的方法都加了 Synchronized 关键字修饰，所以线程安全。

它是数组+链表的实现。

### ****2. ConcurrentHashMap 问题？****

取消 segments 字段，直接采用 transient volatile HashEntry<K,V>[] table 保存数据。

采用 table 数组元素作为锁，从而实现了对每一行数据进行加锁，进一步减少并发冲突的概率。

把 Table 数组＋单向链表的数据结构变成为 Table 数组 ＋ 单向链表 ＋ 红黑树的结构。

当链表长度超过 8 以后，单向链表变成了红黑数；在哈希表扩容时，如果发现链表长度小于 6，则会由红黑树重新退化为链表。

对于其他详细我不吹，看懂的么几个，他比 HashMap 还要难。

对于线程安全环境下建议使用 ConcurrentHashMap 而不去使用 Hashtable。

### ****3. 为什么不去使用 Hashtable 而去使用 ConcurrentHashMap？****

HashTable 容器使用 synchronized 来保证线程安全，但在线程竞争激烈的情况下 HashTable 的效率非常低下。因为当一个线程访问 HashTable 的同步方法时，其他线程访问 HashTable 的同步方法时，可能会进入阻塞或轮询状态。如线程 1 使用 put 进行添加元素，线程 2 不但不能使用 put 方法添加元素，并且也不能使用 get 方法来获取元素，所以竞争越激烈效率越低。

### ****4. ConcurrentSkipListMap 与 TreeMap 的选择？****

ConcurrentSkipListMap 提供了一种线程安全的并发访问的排序映射表。内部是 SkipList（跳表）结构实现，利用底层的插入、删除的 CAS 原子性操作，通过死循环不断获取最新的结点指针来保证不会出现竞态条件。在理论上能够在 O(log(n)) 时间内完成查找、插入、删除操作。调用 ConcurrentSkipListMap 的 size 时，由于多个线程可以同时对映射表进行操作，所以映射表需要遍历整个链表才能返回元素个数，这个操作是个 O(log(n)) 的操作。

在 JDK1.8 中，ConcurrentHashMap 的性能和存储空间要优于 ConcurrentSkipListMap，但是 ConcurrentSkipListMap 有一个功能：它会按照键的自然顺序进行排序。

故需要对键值排序，则我们可以使用 TreeMap，在并发场景下可以使用 ConcurrentSkipListMap。

所以我们并不会去纠结 ConcurrentSkipListMap 和 ConcurrentHashMap 两者的选择。

### ****5. LinkedHashMap 的使用？****

主要是为了解决读取的有序性。基于 HashMap 实现的。

## ****Queue****

### ****1. 队列是什么？****

我们都知道队列 (Queue) 是一种先进先出 (FIFO) 的数据结构，Java 中定义了 java.util.Queue 接口用来表示队列。Java 中的 Queue 与 List、Set 属于同一个级别接口，它们都是实现了 Collection 接口。

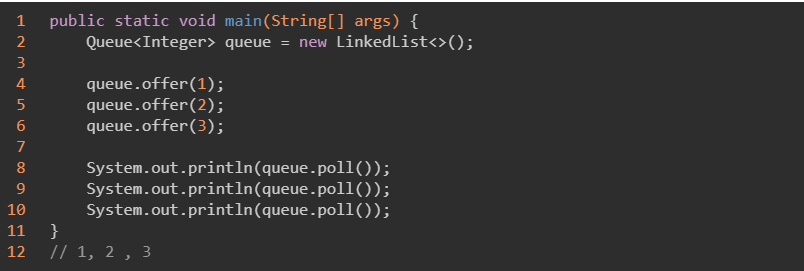
注意：HashMap 没有实现 Collection 接口。

### ****2. Deque 是什么？****

它是一个双端队列。我们用到的 linkedlist 就是实现了 deque 的接口。支持在两端插入和移除元素。

### ****3. 常见的几种队列实现？****

#### LinkedList 是链表结构，队列呢也是一个列表结构，继承关系上 LinkedList 实现了 Queue，所以对于 Queue 来说，添加是 offer(obj)，删除是 poll()，获取队头(不删除)是 peek() 。



#### PriorityQueue 维护了一个有序列表，插入或者移除对象会进行 Heapfy 操作，默认情况下可以称之为小顶堆。当然，我们也可以给它指定一个实现了 java.util.Comparator 接口的排序类来指定元素排列的顺序。PriorityQueue 是一个无界队列，当你设置初始化大小还是不设置都不影响他继续添加元素。

#### ConcurrentLinkedQueue 是基于链接节点的并且线程安全的队列。因为它在队列的尾部添加元素并从头部删除它们，所以只要不需要知道队列的大小 ConcurrentLinkedQueue 对公共集合的共享访问就可以工作得很好。收集关于队列大小的信息会很慢，需要遍历队列。

### ****4. ArrayBlockingQueue 与 LinkedBlockingQueue 的区别，哪个性能好呢？****

ArrayBlockingQueue 是有界队列。LinkedBlockingQueue 看构造方法区分，默认构造方法最大值是 2^31-1。但是当 take 和 put 操作时，ArrayBlockingQueue 速度要快于 LinkedBlockingQueue。

ArrayBlockingQueue 中的锁是没有分离的，即生产和消费用的是同一个锁。LinkedBlockingQueue 中的锁是分离的，即生产用的是 putLock，消费是 takeLock；ArrayBlockingQueue 基于数组，在生产和消费的时候，是直接将枚举对象插入或移除的，不会产生或销毁任何额外的对象实例；LinkedBlockingQueue 基于链表，在生产和消费的时候，需要把枚举对象转换为 Node 进行插入或移除，会生成一个额外的 Node 对象，这在长时间内需要高效并发地处理大批量数据的系统中，其对于 GC 的影响还是存在一定的区别。

LinkedBlockingQueue 的消耗是 ArrayBlockingQueue 消耗的 10 倍左右，即 LinkedBlockingQueue 消耗在 1500 毫秒左右，而 ArrayBlockingQueue 只需 150 毫秒左右。

按照实现原理来分析，ArrayBlockingQueue 完全可以采用分离锁，从而实现生产者和消费者操作的完全并行运行。Doug Lea 之所以没这样去做，也许是因为 ArrayBlockingQueue 的数据写入和获取操作已经足够轻巧，以至于引入独立的锁机制，除了给代码带来额外的复杂性外，其在性能上完全占不到任何便宜。

在使用 LinkedBlockingQueue 时，若用默认大小且当生产速度大于消费速度时候，有可能会内存溢出。

在使用 ArrayBlockingQueue 和 LinkedBlockingQueue 分别对 1000000 个简单字符做入队操作时，我们测试的是 ArrayBlockingQueue 会比 LinkedBlockingQueue 性能好 , 好差不多 50% 起步。

### ****5. BlockingQueue 的问题以及 ConcurrentLinkedQueue 的问题？****

BlockingQueue 可以是限定容量的。

BlockingQueue 实现主要用于生产者-使用者队列，但它另外还支持 collection 接口。

BlockingQueue 实现是线程安全的。

BlockingQueue 是阻塞队列（看你使用的方法），ConcurrentLinkedQueue 是非阻塞队列。

LinkedBlockingQueue 是一个线程安全的阻塞队列，基于链表实现，一般用于生产者与消费者模型的开发中。采用锁机制来实现多线程同步，提供了一个构造方法用来指定队列的大小，如果不指定大小，队列采用默认大小（Integer.MAX\_VALUE，即整型最大值）。

ConcurrentLinkedQueue 是一个线程安全的非阻塞队列，基于链表实现。java 并没有提供构造方法来指定队列的大小，因此它是无界的。为了提高并发量，它通过使用更细的锁机制，使得在多线程环境中只对部分数据进行锁定，从而提高运行效率。他并没有阻塞方法，take 和 put 方法，注意这一点。

### ****6. 简要概述 BlockingQueue 常用的七个实现类？****

ArrayBlockingQueue 构造函数必须传入指定大小，所以他是一个有界队列。

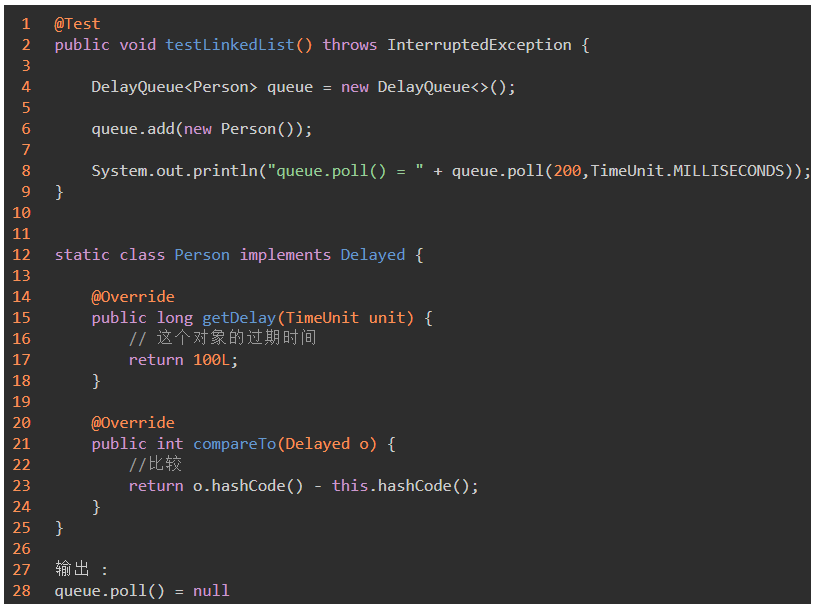
LinkedBlockingQueue 分为两种情况，第一种构造函数指定大小，他是一个有界队列，第二种情况不指定大小他可以称之为无界队列，队列最大值为 Integer.MAX\_VALUE。

PriorityBlockingQueue（还有一个双向的 LinkedBlockingDeque）他是一个无界队列，不管你使用什么构造函数。一个内部由优先级堆支持的、基于时间的调度队列。队列中存放 Delayed 元素，只有在延迟期满后才能从队列中提取元素。当一个元素的 getDelay() 方法返回值小于等于 0 时才能从队列中 poll 中元素，否则 poll() 方法会返回 null。

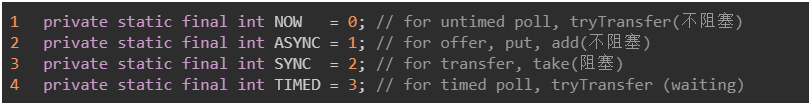
SynchronousQueue 这个队列类似于 Golang的channel，也就是 chan，跟无缓冲区的 chan 很相似。比如 take 和 put 操作就跟 chan 一模一样。但是区别在于他的 poll 和 offer 操作可以设置等待时间。

DelayQueue 延迟队列提供了在指定时间才能获取队列元素的功能，队列头元素是最接近过期的元素。没有过期元素的话，使用 poll() 方法会返回 null 值，超时判定是通过 getDelay(TimeUnit.NANOSECONDS) 方法的返回值小于等于 0 来判断。延时队列不能存放空元素。

添加的元素必须实现 java.util.concurrent.Delayed 接口：

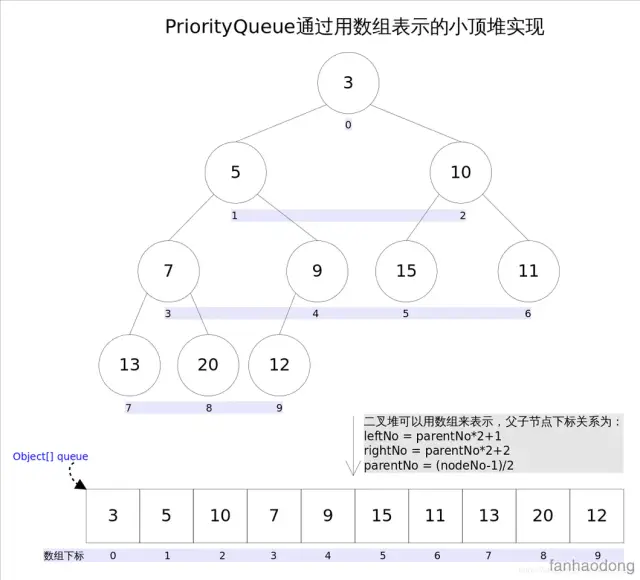


LinkedTransferQueue 是 JDK1.7 加入的无界队列，亮点就是无锁实现的，性能高。Doug Lea 说这个是最有用的 BlockingQueue 了，性能最好的一个。Doug Lea 说从功能角度来讲，LinkedTransferQueue 实际上是 ConcurrentLinkedQueue、SynchronousQueue（公平模式）和 LinkedBlockingQueue 的超集。他的 transfer 方法表示生产必须等到消费者消费才会停止阻塞。生产者会一直阻塞直到所添加到队列的元素被某一个消费者所消费（不仅仅是添加到队列里就完事）。同时我们知道上面那些 BlockingQueue 使用了大量的 condition 和 lock，这样子效率很低，而 LinkedTransferQueue 则是无锁队列。他的核心方法其实就是 xfer() 方法，基本所有方法都是围绕着这个进行的，一般就是 SYNC、ASYNC、NOW 来区分状态量。像 put、offer、add 都是 ASYNC，所以不会阻塞。下面几个状态对应的变量。



### ****7. (小顶堆) 优先队列 PriorityQueue 的实现？****

小顶堆是什么：任意一个非叶子节点的权值都不大于其左右子节点的权值。



PriorityQueue 是非线程安全的，PriorityBlockingQueue 是线程安全的。

两者都使用了堆，算法原理相同。

PriorityQueue 的逻辑结构是一棵完全二叉树，就是因为完全二叉树的特点，他实际存储确实可以为一个数组的，所以他的存储结构其实是一个数组。

首先 java 中的 PriorityQueue 是优先队列，使用的是小顶堆实现，因此结果不一定是完全升序。

## ****Stack****

栈结构属于一种先进者后出，类似于一个瓶子，先进去的会压到栈低（push 操作），出去的时候只有一个出口就是栈顶，返回栈顶元素，这个操作称为 pop。

Stack 类继承自 Vector，所有方法都加入了 sync 修饰，使得效率很低，线程安全。

