不考虑多线程并发的情况下，容器类一般使用 ArrayList、HashMap 等线程不安全的类，效率更高。在并发场景下，常会用到 ConcurrentHashMap、ArrayBlockingQueue 等线程安全的容器类，虽然牺牲了一些效率，但却得到了安全。

上面提到的线程安全容器都在 java.util.concurrent 包下，这个包下并发容器不少，今天全部翻出来鼓捣一下。

仅做简单介绍，后续再分别深入探索。

1. ConcurrentHashMap：并发版 HashMap
2. CopyOnWriteArrayList：并发版 ArrayList
3. CopyOnWriteArraySet：并发 Set
4. ConcurrentLinkedQueue：并发队列 (基于链表)
5. ConcurrentLinkedDeque：并发队列 (基于双向链表)
6. ConcurrentSkipListMap：基于跳表的并发 Map
7. ConcurrentSkipListSet：基于跳表的并发 Set
8. ArrayBlockingQueue：阻塞队列 (基于数组)
9. LinkedBlockingQueue：阻塞队列 (基于链表)
10. LinkedBlockingDeque：阻塞队列 (基于双向链表)
11. PriorityBlockingQueue：线程安全的优先队列
12. SynchronousQueue：读写成对的队列
13. LinkedTransferQueue：基于链表的数据交换队列
14. DelayQueue：延时队列

**1.ConcurrentHashMap 并发版 HashMap**

最常见的并发容器之一，可以用作并发场景下的缓存。底层依然是哈希表，但在 JAVA 8 中有了不小的改变，而 JAVA 7 和 JAVA 8 都是用的比较多的版本，因此经常会将这两个版本的实现方式做一些比较（比如面试中）。

一个比较大的差异就是，JAVA 7 中采用分段锁来减少锁的竞争，JAVA 8 中放弃了分段锁，采用 CAS（一种乐观锁），同时为了防止哈希冲突严重时退化成链表（冲突时会在该位置生成一个链表，哈希值相同的对象就链在一起），会在链表长度达到阈值（8）后转换成红黑树（比起链表，树的查询效率更稳定）。

**2.CopyOnWriteArrayList 并发版 ArrayList**

并发版 ArrayList，底层结构也是数组，和 ArrayList 不同之处在于：当新增和删除元素时会创建一个新的数组，在新的数组中增加或者排除指定对象，最后用新增数组替换原来的数组。

适用场景：由于读操作不加锁，写（增、删、改）操作加锁，因此适用于读多写少的场景。

局限：由于读的时候不会加锁（读的效率高，就和普通 ArrayList 一样），读取的当前副本，因此可能读取到脏数据。如果介意，建议不用。

看看源码感受下：



**3.CopyOnWriteArraySet 并发 Set**

基于 CopyOnWriteArrayList 实现（内含一个 CopyOnWriteArrayList 成员变量），也就是说底层是一个数组，意味着每次 add 都要遍历整个集合才能知道是否存在，不存在时需要插入（加锁）。

适用场景：在 CopyOnWriteArrayList 适用场景下加一个，集合别太大（全部遍历伤不起）。

**4.ConcurrentLinkedQueue 并发队列 (基于链表)**

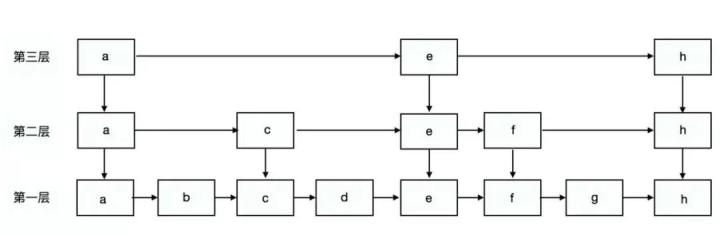
基于链表实现的并发队列，使用乐观锁 (CAS) 保证线程安全。因为数据结构是链表，所以理论上是没有队列大小限制的，也就是说添加数据一定能成功。

**5.ConcurrentLinkedDeque 并发队列 (基于双向链表)**

基于双向链表实现的并发队列，可以分别对头尾进行操作，因此除了先进先出 (FIFO)，也可以先进后出（FILO），当然先进后出的话应该叫它栈了。

**6.ConcurrentSkipListMap 基于跳表的并发 Map**

SkipList 即跳表，跳表是一种空间换时间的数据结构，通过冗余数据，将链表一层一层索引，达到类似二分查找的效果



**7.ConcurrentSkipListSet 基于跳表的并发 Set**

类似 HashSet 和 HashMap 的关系，ConcurrentSkipListSet 里面就是一个 ConcurrentSkipListMap，就不细说了。

**8.ArrayBlockingQueue 阻塞队列 (基于数组)**

基于数组实现的可阻塞队列，构造时必须制定数组大小，往里面放东西时如果数组满了便会阻塞直到有位置（也支持直接返回和超时等待），通过一个锁 ReentrantLock 保证线程安全。



乍一看会有点疑惑，读和写都是同一个锁，那要是空的时候正好一个读线程来了不会一直阻塞吗？

答案就在 notEmpty、notFull 里，这两个出自 lock 的小东西让锁有了类似 synchronized + wait + notify 的功能。传送门 → 终于搞懂了 sleep/wait/notify/notifyAll

**9.LinkedBlockingQueue 阻塞队列 (基于链表)**

基于链表实现的阻塞队列，想比与不阻塞的 ConcurrentLinkedQueue，它多了一个容量限制，如果不设置默认为 int 最大值。

**10.LinkedBlockingDeque 阻塞队列 (基于双向链表)**

类似 LinkedBlockingQueue，但提供了双向链表特有的操作。

**11.PriorityBlockingQueue 线程安全的优先队列**

构造时可以传入一个比较器，可以看做放进去的元素会被排序，然后读取的时候按顺序消费。某些低优先级的元素可能长期无法被消费，因为不断有更高优先级的元素进来。

**12.SynchronousQueue 数据同步交换的队列**

一个虚假的队列，因为它实际上没有真正用于存储元素的空间，每个插入操作都必须有对应的取出操作，没取出时无法继续放入。

import java.util.concurrent.SynchronousQueue;

public class Main {

public static void main(String[] args) {

SynchronousQueue<Integer> queue = new SynchronousQueue<>();

new Thread(()->{

try{

for(int i=0;;i++){

System.out.println("放入：" + i);

queue.put(i);

}

}catch (InterruptedException e){

e.printStackTrace();

}

}).start();

new Thread(()->{

try{

while(true){

System.out.println("取出：" + queue.take());

Thread.sleep((long)(Math.random()\*2000));

}

}catch (InterruptedException e){

e.printStackTrace();

}

}).start();

}

}

运行结果：

取出：0

放入：0

取出：1

放入：1

放入：2

取出：2

取出：3

放入：3

取出：4

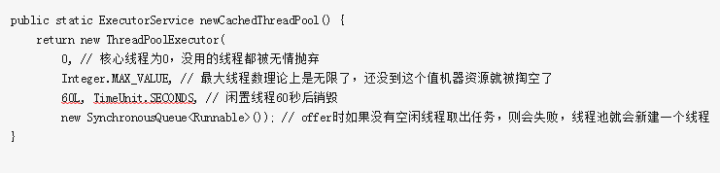
放入：4

...

...

可以看到，写入的线程没有任何 sleep，可以说是全力往队列放东西，而读取的线程又很不积极，读一个又 sleep 一会。输出的结果却是读写操作成对出现。

JAVA 中一个使用场景就是 Executors.newCachedThreadPool()，创建一个缓存线程池。



**13.LinkedTransferQueue 基于链表的数据交换队列**

实现了接口 TransferQueue，通过 transfer 方法放入元素时，如果发现有线程在阻塞在取元素，会直接把这个元素给等待线程。如果没有人等着消费，那么会把这个元素放到队列尾部，并且此方法阻塞直到有人读取这个元素。和 SynchronousQueue 有点像，但比它更强大。

**14.DelayQueue 延时队列**

可以使放入队列的元素在指定的延时后才被消费者取出，元素需要实现 Delayed 接口。

**总结**

上面简单介绍了 JAVA 并发包下的一些容器类，知道有这些东西，遇到合适的场景时就能想起有个现成的东西可以用了。想要知其所以然，后续还得再深入探索一番。