链滴

同步容器和并发容器总结

作者：valarchie

原文链接：https://ld246.com/article/1569027973614

来源网站：链滴

<h4 id="什么是同步容器-">什么是同步容器？</h4>
同步容器通过 synchronized 关键字修饰容器保证同一时刻内只有一个线程在使用容器，从而使容器线程安全。synchronized 的意思是同步，它体现在将多线程变为串行等待执行。（但注意一点复合操作不能保证线程安全。举例：A 线程第一步获取尾节点，第二步将尾结点的值加 1，但在 A 线执行完第一步的时候，B 线程删除了尾节点，在 A 线程执行第二步的时候就会报空指针）
<h4 id="什么是并发容器-">什么是并发容器？</h4>
并发容器指的是允许多线程同时使用容器，并且保证线程安全。而为了达到尽可能提高并发，Jav 并发工具包中采用了多种优化方式来提高并发容器的执行效率，核心的就是：锁、CAS（无锁）、C W（读写分离）、分段锁。
<h2 id="同步容器">同步容器</h2>
<h3 id="1-Vector">1.Vector</h3>
Vector 和 ArrayList 一样实现了 List 接口，其对于数组的各种操作和 ArrayList 一样，区别在于ertor 在可能出现线程不安全的所有方法都用 synchronized 进行了修饰。
<h3 id="2-Stack">2.Stack</h3>
Stack 是 Vertor 的子类，Stack 实现的是先进后出的栈。在出栈入栈等操作都进行了 synchroni ed 修饰。
<h3 id="3-HashTable">3.HashTable</h3>
HashTable 实现了 Map 接口，它实现的功能 HashMap 基本一致（HashTable 不可存 null,而 ashMap 的键和值都可以存 null）。区别在于 HashTable 使用了 synchronized 修饰了方法。
<h3 id="4-Collections提供的同步集合类">4.Collections 提供的同步集合类</h3>

List list = Collections.synchronizedList(new ArrayList());
Set set = Collections.synchronizedSet(new HashSet());
Map map = Collections.synchronizedMap(new HashMap());

其实以上三个容器都是 Collections 通过代理模式对原本的操作加上了 synchronized 同步。而 s nchronized 的同步粒度太大，导致在多线程处理的效率很低。所以在 JDK1.5 的时候推出了并发包的并发容器，来应对多线程下容器处理效率低的问题。
<h2 id="并发容器">并发容器</h2>
<h3 id="1-CopyOnWriteArrayList">1.CopyOnWriteArrayList</h3>
CopyOnWriteArrayList 相当于实现了线程安全的 ArrayList，它的机制是在对容器有写入操作，copy 出一份副本数组，完成操作后将副本数组引用赋值给容器。底层是通过 ReentrantLock 来保同步。但它通过牺牲容器的一致性来换取容器的高并发效率（在 copy 期间读到的是旧数据）。所以能在需要强一致性的场景下使用。
<h3 id="2-CopyOnWriteArraySet">2.CopyOnWriteArraySet</h3>
CopyOnWriteArraySet 和 CopyOnWriteArrayList 原理一样，它是实现了 CopyOnWrite 机制 Set 集合。
<h3 id="3-ConcurrentHashMap">3.ConcurrentHashMap</h3>
ConcurrentHashMap 相当于实现了线程安全的 HashMap。其中的 key 是无序的，并且 key 和 value 都不能为 null。在 JDK8 之前，ConcurrentHashMap 采用了分段锁机制来提高并发效率，只在操作同一分段的键值对时才需要加锁。到了 JDK8 之后，摒弃了锁分段机制，改为利用 CAS 算法。/p>
<h3 id="4-ConcurrentSkipListMap">4.ConcurrentSkipListMap</h3>
ConcurrentSkipListMap 相当于实现了线程安全的 TreeMap。其中的 key 是有序的，并且 key 和 value 都不能为 null。它采用的跳跃表的机制来替代红黑树。为什么不继续使用红黑树呢？因为红树在插入或删除节点的时候需要旋转调整，导致需要控制的粒度较大。而跳跃表使用的是链表，利用锁 CAS 机制实现高并发线程安全。
<h3 id="5-ConcurrentSkipListSet">5.ConcurrentSkipListSet</h3>
ConcurrentSkipListSet 和 ConcurrentSkipListMap 原理一样，它是实现了高并发线程安全的 T eeSet。
<h2 id="Queue类型">Queue 类型</h2>
<h3 id="阻塞型">阻塞型</h3>
<h4 id="1-ArrayBlockingQueue">1.ArrayBlockingQueue</h4>
ArrayBlockingQueue 是采用数组实现的有界阻塞线程安全队列。如果向已满的队列继续塞入元

，将导致当前的线程阻塞。如果向空队列获取元素，那么将导致当前线程阻塞。采用 ReentrantLock 来保证在并发情况下的线程安全。
<h4 id="2-LinkedBlockingQueue">2.LinkedBlockingQueue</h4>
LinkedBlockingQueue 是一个基于单向链表的、范围任意的（其实是有界的）、FIFO 阻塞队列访问与移除操作是在队头进行，添加操作是在队尾进行，并分别使用不同的锁进行保护，只有在可能及多个节点的操作才同时对两个锁进行加锁。
<h4 id="3-PriorityBlockingQueue">3.PriorityBlockingQueue</h4>
PriorityBlockingQueue 是一个支持优先级的无界阻塞队列。默认情况下元素采用自然顺序升序列。也可以自定义类实现 compareTo()方法来指定元素排序规则，
<h4 id="4-DelayQueue">4.DelayQueue</h4>
DelayQueue 是一个内部使用优先级队列实现的无界阻塞队列。同时元素节点数据需要等待多久后才可被访问。取数据队列为空时等待，有数据但延迟时间未到时超时等待。
<h4 id="5-SynchronousQueue">5.SynchronousQueue</h4>
SynchronousQueue 没有容量，是一个不存储元素的阻塞队列，会直接将元素交给消费者，必等队列中的添加元素被消费后才能继续添加新的元素。相当于一条容量为 1 的传送带。
<h4 id="6-LinkedTransferQueue">6.LinkedTransferQueue</h4>
LinkedTransferQueue 是一个有链表组成的无界传输阻塞队列。它集合了 ConcurrentLinkedQeue、SynchronousQueue、LinkedBlockingQueue 等优点。具体机制较为复杂。
<h4 id="7-LinkedBlockingDeque">7.LinkedBlockingDeque</h4>
LinkedBlockingDeque 是一个由链表结构组成的双向阻塞队列。所谓双向队列指的是可以从队的两端插入和移出元素。
<h3 id="非阻塞型">非阻塞型</h3>
<h3 id="1-ConcurrentLinkedQueue">1.ConcurrentLinkedQueue</h3>
ConcurrentLinkedQueue 是线程安全的无界非阻塞队列,其底层数据结构是使用单向链表实现,队和出队操作是使用我们经常提到的 CAS 来保证线程安全。