

20 | 并发容器：都有哪些“坑”需要我们填？

王宝令 2019-04-13



00:00

讲述：王宝令 大小：9.15M

09:59

Java 并发包有很大一部分内容都是关于**并发容器**的，因此学习和搞懂这部分的内容很有必要。

Java 1.5 之前提供的**同步容器**虽然也能保证线程安全，但是性能很差，而 Java 1.5 版本之后提供的并发容器在性能方面则做了很多优化，并且容器的类型也更加丰富了。下面我们就对比二者来学习这部分的内容。

同步容器及其注意事项

Java 中的容器主要可以分为四个大类，分别是 List、Map、Set 和 Queue，但并不是所有的 Java 容器都是线程安全的。例如，我们常用的 ArrayList、HashMap 就不是线程安全的。在介绍线程安全的容器之前，我们先思考这样一个问题：如何将非线程安全的容器变成线程安全的容器？

在前面《12 | 如何用面向对象思想写好并发程序？》我们讲过实现思路其实很简单，只要把非线程安全的容器封装在对象内部，然后控制好访问路径就可以了。

下面我们就以 ArrayList 为例，看看如何将它变成线程安全的。在下面的代码中，SafeArrayList 内部持有一个 ArrayList 的实例 c，所有访问 c 的方法我们都增加了 synchronized 关键字，需要注意的是我们还增加了一个 addIfNotExist() 方法，这个方法也是用 synchronized 来保证原子性的。

复制代码

```
1 SafeArrayList<T>{
2     // 封装 ArrayList
3     List<T> c = new ArrayList<>();
4     // 控制访问路径
5     synchronized
```

```

6   T get(int idx){
7       return c.get(idx);
8   }
9
10  synchronized
11  void add(int idx, T t) {
12      c.add(idx, t);
13  }
14
15  synchronized
16  boolean addIfNotExist(T t){
17      if(!c.contains(t)) {
18          c.add(t);
19          return true;
20      }
21      return false;
22  }
23 }
24


```

看到这里，你可能会举一反三，然后想到：所有非线程安全的类是不是都可以用这种包装的方式来实现线程安全呢？其实这一点不止你想到了，Java SDK 的开发人员也想到了，所以他们在 Collections 这个类中还提供了一套完备的包装类，比如下面的示例代码中，分别把 ArrayList、HashSet 和 HashMap 包装成了线程安全的 List、Set 和 Map。

```

1 List list = Collections.
2   synchronizedList(new ArrayList());
3 Set set = Collections.
4   synchronizedSet(new HashSet());
5 Map map = Collections.
6   synchronizedMap(new HashMap());
7

```

 复制代码

我们曾经多次强调，**组合操作需要注意竞态条件问题**，例如上面提到的 addIfNotExist() 方法就包含组合操作。组合操作往往隐藏着竞态条件问题，即便每个操作都能保证原子性，也并不能保证组合操作的原子性，这个一定要注意。

在容器领域一个容易被忽视的“坑”是用迭代器遍历容器，例如在下面的代码中，通过迭代器遍历容器 list，对每个元素调用 foo() 方法，这就存在并发问题，这些组合的操作不具备原子性。

```

1 List list = Collections.
2   synchronizedList(new ArrayList());
3 Iterator i = list.iterator();
4 while (i.hasNext())
5     foo(i.next());
6

```

 复制代码

而正确做法是下面这样，锁住 list 之后再执行遍历操作。如果你查看 Collections 内部的包装类源码，你会发现包装类的公共方法锁的是对象的 this，其实就是我们这里的 list，所以锁住 list 绝对是线程安全的。

 复制代码

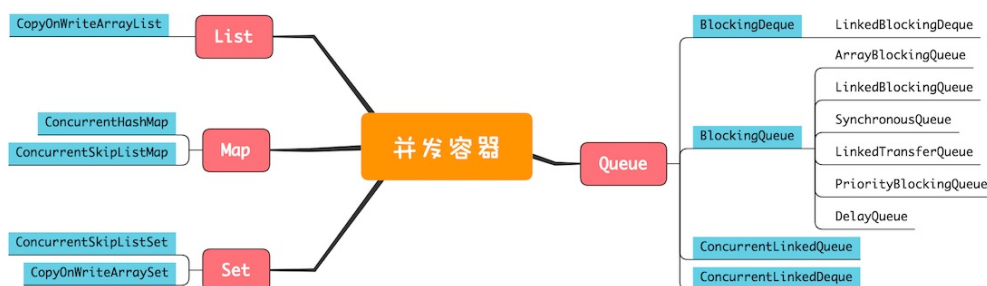
```
1 List list = Collections.  
2   synchronizedList(new ArrayList());  
3   synchronized (list) {  
4     Iterator i = list.iterator();  
5     while (i.hasNext())  
6       foo(i.next());  
7   }  
8
```

上面我们提到的这些经过包装后线程安全容器，都是基于 `synchronized` 这个同步关键字实现的，所以也被称为**同步容器**。Java 提供的同步容器还有 `Vector`、`Stack` 和 `Hashtable`，这三个容器不是基于包装类实现的，但同样是基于 `synchronized` 实现的，对这三个容器的遍历，同样要加锁保证互斥。

并发容器及其注意事项

Java 在 1.5 版本之前所谓的线程安全的容器，主要指的就是**同步容器**。不过同步容器有个最大的问题，那就是性能差，所有方法都用 `synchronized` 来保证互斥，串行度太高了。因此 Java 在 1.5 及之后版本提供了性能更高的容器，我们一般称为**并发容器**。

并发容器虽然数量非常多，但依然是前面我们提到的四大类：`List`、`Map`、`Set` 和 `Queue`，下面的并发容器关系图，基本上把我们经常用的容器都覆盖到了。



并发容器关系图

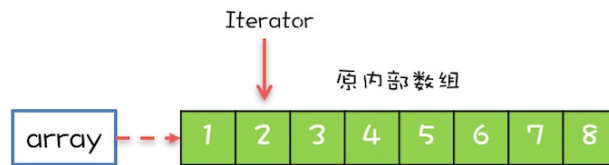
鉴于并发容器的数量太多，再加上篇幅限制，所以我并不会一一详细介绍它们的用法，只是把关键点介绍一下。

（一）List

`List` 里面只有一个实现类就是 `CopyOnWriteArrayList`。`CopyOnWrite`，顾名思义就是写的时候会共享变量新复制一份出来，这样做的好处是读操作完全无锁。

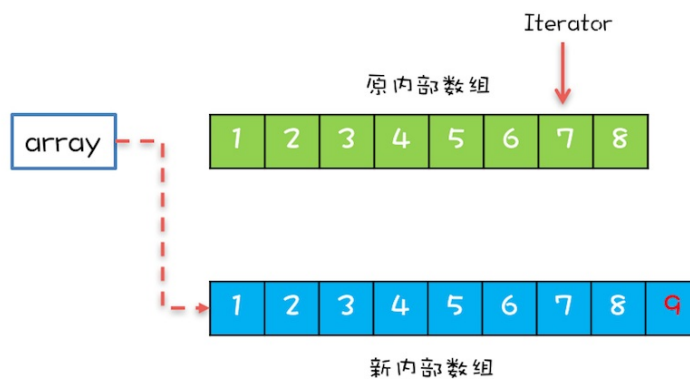
那 `CopyOnWriteArrayList` 的实现原理是怎样的呢？下面我们就来简单介绍一下

`CopyOnWriteArrayList` 内部维护了一个数组，成员变量 `array` 就指向这个内部数组，所有的读操作都是基于 `array` 进行的，如下图所示，迭代器 `Iterator` 遍历的就是 `array` 数组。



执行迭代的内部结构图

如果在遍历 array 的同时，还有一个写操作，例如增加元素，CopyOnWriteArrayList 是如何处理的呢？CopyOnWriteArrayList 会将 array 复制一份，然后在新复制处理的数组上执行增加元素的操作，执行完之后再将 array 指向这个新的数组。通过下图你可以看到，读写是可以并行的，遍历操作一直都是基于原 array 执行，而写操作则是基于新 array 进行。



执行增加元素的内部结构图

使用 CopyOnWriteArrayList 需要注意的“坑”主要有两个方面。一个是应用场景，CopyOnWriteArrayList 仅适用于写操作非常少的场景，而且能够容忍读写的短暂不一致。例如上面的例子中，写入的新元素并不能立刻被遍历到。另一个需要注意的是，CopyOnWriteArrayList 迭代器是只读的，不支持增删改。因为迭代器遍历的仅仅是一个快照，而对快照进行增删改是没有意义的。

（二）Map

Map 接口的两个实现是 ConcurrentHashMap 和 ConcurrentSkipListMap，它们从应用的角度来看，主要区别在于ConcurrentHashMap 的 key 是无序的，而 ConcurrentSkipListMap 的 key 是有序的。所以如果你需要保证 key 的顺序，就只能使用 ConcurrentSkipListMap。

使用 ConcurrentHashMap 和 ConcurrentSkipListMap 需要注意的地方是，它们的 key 和 value 都不能为空，否则会抛出NullPointerException

这个运行时异常。下面这个表格总结了 Map 相关的实现类对于 key 和 value 的要求，你可以对比学习。

集合类	Key	Value	是否线程安全
HashMap	允许为null	允许为null	否
TreeMap	不允许为null	允许为null	否
Hashtable	不允许为null	不允许为null	是
ConcurrentHashMap	不允许为null	不允许为null	是
ConcurrentSkipListMap	不允许为null	不允许为null	是

ConcurrentSkipListMap 里面的 SkipList 本身就是一种数据结构，中文一般都翻译为“跳表”。跳表插入、删除、查询操作平均的时间复杂度是 $O(\log n)$ ，理论上和并发线程数没有关系，所以在并发度非常高的情况下，若你对 ConcurrentHashMap 的性能还不满意，可以尝试一下 ConcurrentSkipListMap。

（三）Set

Set 接口的两个实现是 CopyOnWriteArraySet 和 ConcurrentSkipListSet，使用场景可以参考前面讲述的 CopyOnWriteArrayList 和 ConcurrentSkipListMap，它们的原理都是一样的，这里就不再赘述了。

（四）Queue

Java 并发包里面 Queue 这类并发容器是最复杂的，你可以从以下两个维度来分类。一个维度是阻塞与非阻塞，所谓阻塞指的是当队列已满时，入队操作阻塞；当队列已空时，出队操作阻塞。另一个维度是单端与双端，单端指的是只能队尾入队，队首出队；而双端指的是队首队尾皆可入队出队。Java 并发包里阻塞队列都用 Blocking 关键字标识，单端队列使用 Queue 标识，双端队列使用 Deque 标识。

这两个维度组合后，可以将 Queue 细分为四大类，分别是：

1.单端阻塞队列：其实现有 ArrayBlockingQueue、LinkedBlockingQueue、SynchronousQueue、LinkedTransferQueue、PriorityBlockingQueue 和 DelayQueue。内部一般会持有一个队列，这个队列可以是数组（其实现是 ArrayBlockingQueue）也可以是链表（其实现是 LinkedBlockingQueue）；甚至可以不持有队列（其实现是 SynchronousQueue），此时生产者线程的入队操作必须等待消费者线程的出队操作。而 LinkedTransferQueue 融合 LinkedBlockingQueue 和 SynchronousQueue 的功能，性能比 LinkedBlockingQueue 更好；PriorityBlockingQueue 支持按照优先级出队；DelayQueue 支持延时出队。



2. **双端阻塞队列**：其实现是 `LinkedBlockingDeque`。



双端阻塞队列示意图

3. **单端非阻塞队列**：其实现是 `ConcurrentLinkedQueue`。

4. **双端非阻塞队列**：其实现是 `ConcurrentLinkedDeque`。

另外，使用队列时，需要格外注意队列是否支持有界（所谓有界指的是内部的队列是否有容量限制）。实际工作中，一般都不建议使用无界的队列，因为数据量大了之后很容易导致 OOM。上面我们提到的这些 Queue 中，只有 `ArrayBlockingQueue` 和 `LinkedBlockingQueue` 是支持有界的，所以在使用其他无界队列时，一定要充分考虑是否存在导致 OOM 的隐患。

总结

Java 并发容器的内容很多，但鉴于篇幅有限，我们只是对一些关键点进行了梳理和介绍。

而在实际工作中，你不仅要清楚每种容器的特性，还要能选对容器，这才是关键，至于每种容器的用法，用的时候看一下 API 说明就可以了，这些容器的使用都不难。在文中，我们甚至都没有介绍 Java 容器的快速失败机制（Fail-Fast），原因就在于当你选对容器的时候，根本不会触发它。

课后思考

线上系统 CPU 突然飙升，你怀疑有同学在并发场景里使用了 `HashMap`，因为在 1.8 之前的版本里并发执行 `HashMap.put()` 可能会导致 CPU 飙升到 100%，你觉得该如何验证你的猜测呢？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。



由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

Ctrl + Enter 发表

0/2000字

提交留言

精选留言(16)



黑白尤文

java7中的HashMap在执行put操作时会涉及到扩容，由于扩容时链表并发操作会造成链表成环，所以可能导致cpu飙升100%。

👍 11

2019-04-13

作者回复: 👍



Grubby

这篇太简单了，但其实这些容器平时用的挺多的，希望老师后面能出一篇更加详细的介绍

👍 10

2019-04-13



CCC

老师，用跳表实现的ConcurrentSkipListMap为什么可以做到无锁并发呢

👍 3

2019-04-13

作者回复: 那个跳表就跟字典的索引一样，通过这个索引既能快速定位数据，也能隔离并发（可以并发查看不同页上的字）



Liam

LinkedTransferQueue有什么应用场景吗？

👍 3

2019-04-13

作者回复: 实际工作中，为了防止OOM，基本上都使用有界队列，我工作中也没用过LinkedTransferQueue。



张天屹

我理解的hashMap比其它线性容器更容易出问题是因为有扩容操作，存在更多竞态条件，所以如果条件满足时切换可能导致新生成很多数组，甚至可能出现链表闭环，这种情况可以查看堆栈，比如jstack查看会发现方法调用栈一直卡在HashMap的方法。另外上文迭代器遍历不安全是因为hasNext(size)和next()存在的竞态条件吗

👍 3

2019-04-13

作者回复: 👍，不止是存在竞态条件，如果在遍历的时候出现修改操作，直接抛快速失败异常



WolvesLeader

个人认为您第二篇内存模型讲的非常棒，，，，，，，，，，

👍 2

2019-04-13

作者回复: 我觉得自己理解起来困难而且对实际工作还有用的就会讲的深入一些, 反之我觉得概念或者工具跟正常思维没有冲突, 就会讲的简单, 甚至略过。毕竟我们只是工具的使用者, 首要问题是利用这些工具解决问题。感谢你的认可, 我甚至觉得写完第二篇和管程之后就可以收工了, 其他所有章节不过就是帮助大家进一步理解, 从不同角度理解。



ykkk88

没有理解为什么concurrentskiplistmap比concurrenthashmap性能好

👍 2 2019-04-13

作者回复: 如果key冲突比较大, hashmap还是要靠链表或者tree来解决冲突的, 所以O(1)是理想值。同时增删改操作很多也影响hashmap性能。这个也是要看冲突情况。也就是说hashmap的稳定性差, 如果很不幸正好偶遇它的稳定性问题, 同时又接受不了, 就可以尝试skiplistmap, 它能保证稳定性, 无论你的并发量是多大, 也没有key冲突的问题。



独立自主

老师组合操作为什么不能保证原子性。能举个例子吗这里没了解清楚。谢谢

👍 1 2019-04-13

作者回复: 两个原子操作直接, 可能发生很多事情。你可以参考《07 | 安全性、活跃性以及性能问题》里面的set(get()+1)问题



木木匠

jdk1.8以前的HashMap并发扩容的时候会导致陷入死循环, 所以会导致cpu飙升, 那么验证猜想我觉得有2种方法:

1. 线上查故障, 用dump分析线程。
2. 用1.8以前的jdk在本地模拟。

👍 1 2019-04-13

作者回复: 👍



Zach_

老师, hashmap 扩容时, 会遍历链表我能想来, 但是为啥cpu会飙升到100%, 是因为扩容时遍历的是无界链表的原因嘛? 🤔🤔🤔🤔

👍 2019-04-15

作者回复: 链表死循环了



undefined

老师 CopyOnWriteArrayList 如果用于读多写少的场景, 而且短暂不一致, 其实真正可用的场景比较少, 如果增加元素不会引起扩容, 那加锁的开销是不是比复制数组小, 这样看 CopyOnWriteArrayList 其实不是很有意义

👍 2019-04-14

作者回复: cow的读是无锁的, 这个是关键

**QQ怪**

除了jdk8之前因为并发导致的链表成环的问题还有一种可能是因为jdk8之前hash冲突还是使用的是链表，而jdk8之后使用了红黑树，开始还是生成链表，当链表长度为8时就会转变为红黑树，时间复杂度为 $O(\log n)$ ，比链表效果好的多。



2019-04-13

作者回复: 是的，底层实现变了，我同事在1.8版本费了好大劲都没重现出来

**曾轼麟**

帮老师补充HashMap：当数据的HashCode 分布状态良好，并且冲突较少的时候对ConcurrentHashMap（查询，value修改，不包括插入），性能上基本上是和HashMap一致的，主要取决于分段锁的插思想。但是由于插入使用的是CAS的方式，所以如果对数据追加不多（插入）的情况下，建议可以考虑多使用ConcurrentHashMap避免由于修改数据产生一些意想不到的并发问题，当然内部也有保护机制通过抛出ConcurrentModificationException（快速失败机制）来让我们及时发现出现并发数据异常的情况，不知道我补充的是否正确。



2019-04-13

作者回复: 1.8版本之后ConcurrentHashMap的实现改了

**陈华应**

选对容器的前提还是要对原理，特性，使用场景，优缺点，坑，甚至底层实现都了如指掌才能说选对容器，要不然更多的也是蒙对容器



2019-04-13

**crazypokerk**

满满干货，条理立马清晰了！



2019-04-13

**周治慧**

hashmap在put的时候扩容导致链表的死环导致，可以通过遍历去entries中entry的next一直不为空来判断



2019-04-13

作者回复: 原因是对的，cpu飙升不降的问题都可以用dump线程栈来分析