20 | 并发容器：都有哪些“坑”需要我们填？

2019-04-13 王宝令

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。

```
SafeArrayList<T>{
  //封装ArrayList
  List<T> c = new ArrayList<>();
  //控制访问路径
  synchronized
  T get(int idx){
    return c.get(idx);
  }

  synchronized
  void add(int idx, T t) {
    c.add(idx, t);
  }

  synchronized
  boolean addIfNotExist(T t){
    if(!c.contains(t)) {
      c.add(t);
      return true;
    }
    return false;
  }
}
```

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。

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

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。

```
List list = Collections.
  synchronizedList(new ArrayList());
Iterator i = list.iterator();
while (i.hasNext())
  foo(i.next());
```

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。

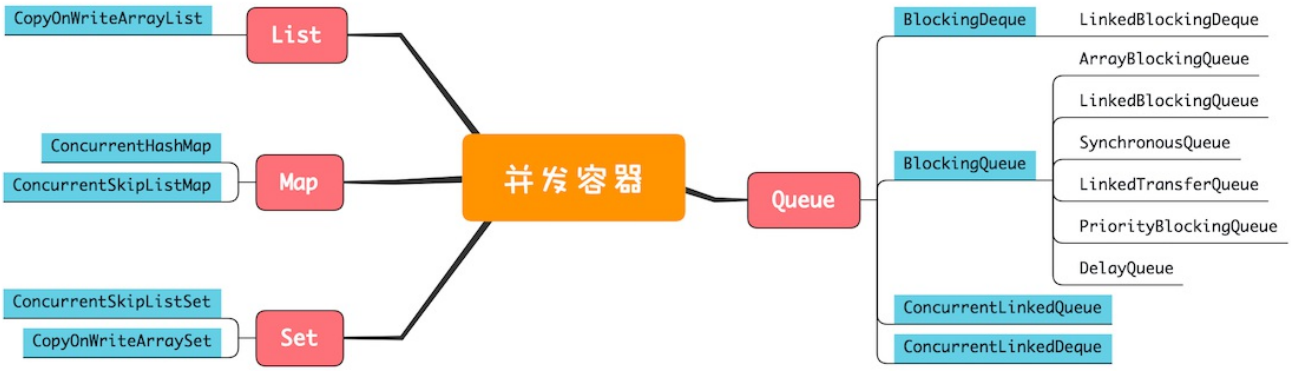
```
List list = Collections.
  synchronizedList(new ArrayList());
synchronized (list) {
  Iterator i = list.iterator();
  while (i.hasNext())
    foo(i.next());
}
```

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

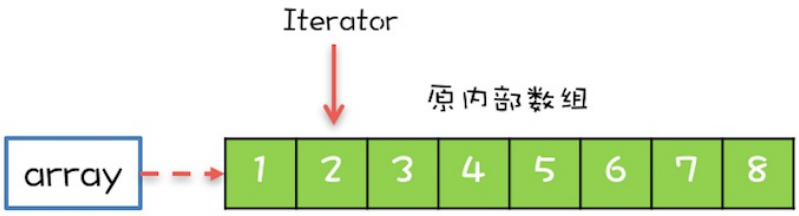
鉴于并发容器的数量太多，再加上篇幅限制，所以我并不会一一详细介绍它们的用法，只是把关键点介绍一下。

（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将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。

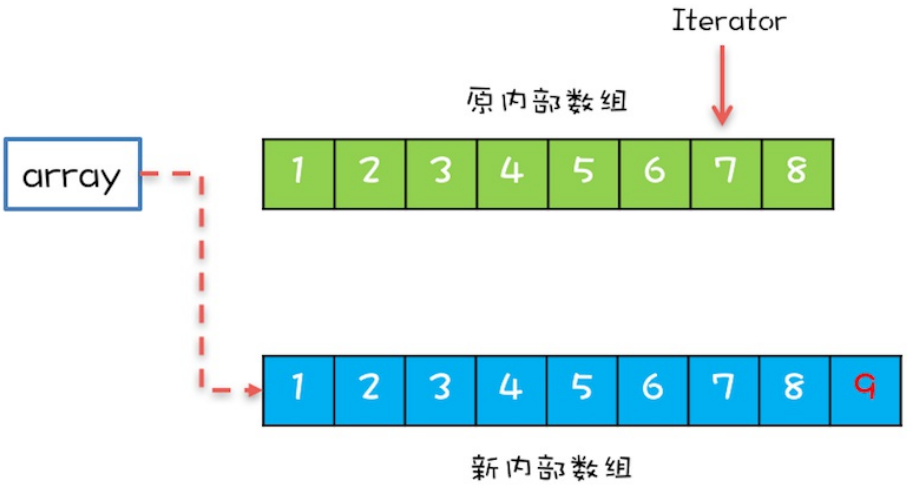
那CopyOnWriteArrayList的实现原理是怎样的呢？下面我们就来简单介绍一下

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**。

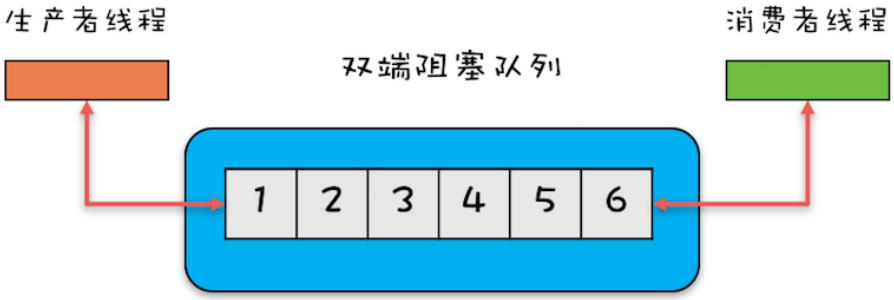
这两个维度组合后，可以将Queue细分为四大类，分别是：

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）；甚至还可以不持有队列（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。

单端阻塞队列示意图

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，你不单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。

精选留言

黑白尤文 25

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%。

2019-04-13

作者回复

2019-04-13

Grubby 21

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

2019-04-13

张天屹 10

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

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

2019-04-13

WolvesLeader 3

个人认为您第二篇内存模型讲的非常棒，，，，，，，，，，

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。

2019-04-13

CCC 3

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

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）

2019-04-13

木木匠 3

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。

2019-04-13

作者回复

2019-04-13

ykkk88 3

没有理解为什么concurrentskiplistmap比concurrenthashmap性能好

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。

2019-04-14

Liam

3

LinkedTransferQueue有什么应用场景吗？

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。

2019-04-13

我劝你善良

2

老师，针对CopyOnWriteArrayList

1.如果正在遍历的时候，同时有两个写操作执行，是会顺序在一个新数组上执行写操作，还是有两个写操作分别进行？如果是两个新数组的话，那么array又将指向哪一个新数组？

2.如果在遍历的过程中，写操作已经完成了，但是遍历尚未结束，那么是array是直接指向新数组，并继续在新数组上执行未完成的遍历，还是等待遍历完成了，再修改array的指向呢？如果在遍历完之前就修改指向，那么就会存在问题了啊！

2019-04-22

作者回复

CopyOnWriteArrayList写操作是互斥的。

2019-04-22

曾轼麟

2

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

2019-04-13

陈华应

2

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

傲

1

老师，我有个问题：
文章里面说，使用CopyOnWriteArrayList时，需要能够容忍读写的短暂不一致，但是我理解CopyOnWriteArrayList应该不会出现不一致问题吧。因为底层的array是用volatile修饰的，根据happens-before原则，对volatile变量的写happens-before于对变量的读，也就是说如果存在并发读写的情况，写线程的setArray()一定是对读线程的getArray()可见的，所以我认为读到的始终都是最新的数据。
不知道我的理解有没有问题？

2019-05-24

作者回复

复制的时候允许读，可能读到数组里旧的元素。数组的引用是一致的，一旦设置就能读到，但是里面的元素会有不一致的情况

2019-05-24

Geek_49a9e9

1

老师，你好，最近两天，我线上跑的计费进程假死了(从1月11日开始跑的，4月10日第一次出现假死）， ExecutorService services = Executors.newFixedThreadPool(taskThreads); CountDownLatch cdt = new CountDownLatch(size);
//一个个的处理数据
for (int j = 0; j < size; j++) {
CFTask task = new CFTask(table, channelIds.get(j), batchId, cdt);
services.submit(task);
}
cdt.await(); 这个有什么错误吗？让多个线程处理步调一致

线上jstack pid 查看 部分日志，如下：好像线程池所有线程都在等待执行，感觉一个数据库查询操作跑死了，很奇怪

2019-04-20

作者回复

看这几行看不出来，一般问题都能通过线程栈发现问题。我遇到过生产者和消费者共用一个线程池，生产者把线程池里的线程用光了，导致消费不了。这种情况下通过线程池不太容易看，需要去计数。不知道你的问题是不是这个。所有线程都等待，还没有死锁，就查查为什么会等待吧。

2019-04-20

月月月月

1

老师，我想问下，文章里提到容器在遍历时要注意加锁保证线程安全，对于非线程安全的容器，我们可以通过包装让它变成线程安全的容器，然后在遍历的时候锁住集合对象。但是对于并发容器来说，在遍历的时候要怎么保证线程安全呢？如果还是锁住容器对象，但是对于不是使用synchronized去实现的并发容器，锁对象不就不一样了吗？那这样该怎么保证线程安全呢？

2019-04-19

作者回复

并发容器的遍历是线程安全的

2019-04-19

龙猫 1

java8之前的版本hashmap执行put方法时会有环形链表的风险，java8以后改成了红黑树

2019-04-18

作者回复

[]

2019-04-18

周治慧 1

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

2019-04-13

wang 0

跳表不是可以做到查找时间复杂度为log(n)么

2019-05-23

xiaoming 0

List list = Collections.synchronizedList(new ArrayList());
Iterator i = list.iterator();
while (i.hasNext())
foo(i.next());
老师, 这段代码是否可以这样理解？ while (i.hasNext()) foo(i.next()); 这段代码没有加锁, 多个线程可以同时操作。
A线程 list[0]= i.next(), 与B线程list[0] = i.next()获取了相同的数据data = 0,
假设foo就是进行+1的操作
然后A,B线程都对data进行 操作 foo(data) ， 两个线程运行完后, list[0] =1。
如果加锁list了, 这段就是就是互斥的, A线程执行完后释放了锁list, B线程获取锁才能执行while (i.hasNext()) foo(i.next()); 两个线程都执行完成后 list[0] =2;

2019-05-12

刘鹏 0

cpu 100%的猜测--》可能用jstack看看？？？？？？

2019-05-10

Geek_c991e0 0

老师好，copyonwrite如果并发写入很多，复制多个array写入，那重新指向的时候怎么把多个合并那，还是说写的时候只允许一个一个的写

2019-05-06

作者回复

只允许一个一个写，写操作是有锁的

2019-05-20