

讲堂 > Java核心技术36讲 > 文章详情

## 第20讲 | 并发包中的ConcurrentLinkedQueue和LinkedBlockingQueue有什么区别？

2018-06-21 杨晓峰



第20讲 | 并发包中的ConcurrentLinkedQueue和LinkedBlockingQueue有什么区...

朗读人：黄洲君 08'36" | 3.94M

在上一讲中，我分析了 Java 并发包中的部分内容，今天我来介绍一下线程安全队列。Java 标准库提供了非常多的线程安全队列，很容易混淆。

今天我要问你的问题是，**并发包中的 ConcurrentLinkedQueue 和 LinkedBlockingQueue 有什么区别？**

### 典型回答

有时候我们把并发包下面的所有容器都习惯叫作并发容器，但是严格来讲，类似 ConcurrentLinkedQueue 这种 “Concurrent\*” 容器，才是真正代表并发。

关于问题中它们的区别：

- Concurrent 类型基于 lock-free，在常见的多线程访问场景，一般可以提供较高吞吐量。
- 而 LinkedBlockingQueue 内部则是基于锁，并提供了 BlockingQueue 的等待性方法。

不知道你有没有注意到，`java.util.concurrent` 包提供的容器（`Queue`、`List`、`Set`）、`Map`，从命名上可以大概区分为 `Concurrent*`、`CopyOnWrite` 和 `Blocking` 等三类，同样是线程安全容器，可以简单认为：

- `Concurrent` 类型没有类似 `CopyOnWrite` 之类容器相对较重的修改开销。
- 但是，凡事都是有代价的，`Concurrent` 往往提供了较低的遍历一致性。你可以这样理解所谓的弱一致性，例如，当利用迭代器遍历时，如果容器发生修改，迭代器仍然可以继续遍历。
- 与弱一致性对应的，就是我介绍过的同步容器常见的行为“fail-fast”，也就是检测到容器在遍历过程中发生了修改，则抛出 `ConcurrentModificationException`，不再继续遍历。
- 弱一致性的另外一个体现是，`size` 等操作准确性是有限的，未必是 100% 准确。
- 与此同时，读取的性能具有一定的不确定性。

## 考点分析

今天的问题是又是一个引子，考察你是否了解并发包内部不同容器实现的设计目的和实现区别。

队列是非常重要的数据结构，我们日常开发中很多线程间数据传递都要依赖于它，`Executor` 框架提供的各种线程池，同样无法离开队列。面试官可以从不同角度考察，比如：

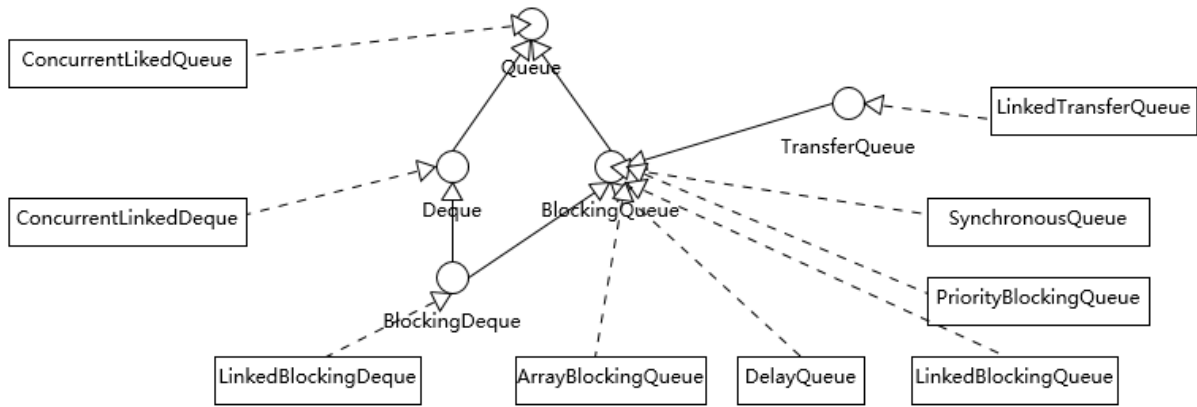
- 哪些队列是有界的，哪些是无界的？（很多同学反馈了这个问题）
- 针对特定场景需求，如何选择合适的队列实现？
- 从源码的角度，常见的线程安全队列是如何实现的，并进行了哪些改进以提高性能表现？

为了能更好地理解这一讲，需要你掌握一些基本的队列本身和数据结构方面知识，如果这方面知识比较薄弱，《数据结构与算法分析》是一本比较全面的参考书，专栏还是尽量专注于 Java 领域的特性。

## 知识扩展

### 线程安全队列一览

我在[专栏第 8 讲](#)中介绍过，常见的集合中如 `LinkedList` 是个 `Deque`，只不过不是线程安全的。下面这张图是 Java 并发类库提供的各种各样的线程安全队列实现，注意，图中并未将非线程安全部分包含进来。



我们可以从不同的角度进行分类，从基本的数据结构的角度分析，有两个特别的Deque实现，ConcurrentLinkedDeque 和 LinkedBlockingDeque。Deque 的侧重点是支持对队列头尾都进行插入和删除，所以提供了特定的方法，如：

- 尾部插入时需要的[addLast\(e\)](#)、[offerLast\(e\)](#)。
- 尾部删除所需要的[removeLast\(\)](#)、[pollLast\(\)](#)。

从上面这些角度，能够理解 ConcurrentLinkedDeque 和 LinkedBlockingQueue 的主要功能区别，也就足够日常开发的需要了。但是如果深入一些，通常会更加关注下面这些方面。

从行为特征来看，绝大部分 Queue 都是实现了 BlockingQueue 接口。在常规队列操作基础上，Blocking 意味着其提供了特定的等待性操作，获取时（take）等待元素进队，或者插入时（put）等待队列出现空位。

复制代码

```

1  /**
2   * 获取并移除队列头结点，如果必要，其会等待直到队列出现元素
3   * ...
4   */
5  E take() throws InterruptedException;
6
7  /**
8   * 插入元素，如果队列已满，则等待直到队列出现空闲空间
9   * ...
10  */
11 void put(E e) throws InterruptedException;
  
```

另一个 BlockingQueue 经常被考察的点，就是是否有界（Bounded、Unbounded），这一点也往往会影响我们在应用开发中的选择，我这里简单总结一下。

- `ArrayBlockingQueue` 是最典型的有界队列，其内部以 `final` 的数组保存数据，数组的大小就决定了队列的边界，所以我们在创建 `ArrayBlockingQueue` 时，都要指定容量，如

```
1 public ArrayBlockingQueue(int capacity, boolean fair)
```

[复制代码](#)

- `LinkedBlockingQueue`，容易被误解为无边界，但其实其行为和内部代码都是基于有界的逻辑实现的，只不过如果我们没有在创建队列时就指定容量，那么其容量限制就自动被设置为 `Integer.MAX_VALUE`，成为了无界队列。
- `SynchronousQueue`，这是一个非常奇葩的队列实现，每个删除操作都要等待插入操作，反之每个插入操作也都要等待删除动作。那么这个队列的容量是多少呢？是 1 吗？其实不是的，其内部容量是 0。
- `PriorityBlockingQueue` 是无边界的优先队列，虽然严格意义上来讲，其大小总归是要受系统资源影响。
- `DelayedQueue` 和 `LinkedTransferQueue` 同样是无边界的队列。对于无边界的队列，有一个自然的结果，就是 `put` 操作永远也不会发生其他 `BlockingQueue` 的那种等待情况。


如果我们分析不同队列的底层实现，`BlockingQueue` 基本都是基于锁实现，一起来看看典型的 `LinkedBlockingQueue`。

```
1 /** Lock held by take, poll, etc */
2 private final ReentrantLock takeLock = new ReentrantLock();
3
4 /** Wait queue for waiting takes */
5 private final Condition notEmpty = takeLock.newCondition();
6
7 /** Lock held by put, offer, etc */
8 private final ReentrantLock putLock = new ReentrantLock();
9
10 /** Wait queue for waiting puts */
11 private final Condition notFull = putLock.newCondition();
```

[复制代码](#)

我在介绍 `ReentrantLock` 的条件变量用法的时候分析过 `ArrayBlockingQueue`，不知道你有没有注意到，其条件变量与 `LinkedBlockingQueue` 版本的实现是有区别的。`notEmpty`、`notFull` 都是同一个再入锁的条件变量，而 `LinkedBlockingQueue` 则改进了锁操作的粒度，头、尾操作使用不同的锁，所以在通用场景下，它的吞吐量相对要更好一些。

下面的 `take` 方法与 `ArrayBlockingQueue` 中的实现，也是有不同的，由于其内部结构是链表，需要自己维护元素数量值，请参考下面的代码。

 复制代码


```
1 public E take() throws InterruptedException {
2     final E x;
3     final int c;
4     final AtomicInteger count = this.count;
5     final ReentrantLock takeLock = this.takeLock;
6     takeLock.lockInterruptibly();
7     try {
8         while (count.get() == 0) {
9             notEmpty.await();
10        }
11        x = dequeue();
12        c = count.getAndDecrement();
13        if (c > 1)
14            notEmpty.signal();
15    } finally {
16        takeLock.unlock();
17    }
18    if (c == capacity)
19        signalNotFull();
20    return x;
21 }
```

类似 `ConcurrentLinkedQueue` 等，则是基于 CAS 的无锁技术，不需要在每个操作时使用锁，所以扩展性表现要更加优异。

相对比较另类的 `SynchronousQueue`，在 Java 6 中，其实现发生了非常大的变化，利用 CAS 替换掉了原本基于锁的逻辑，同步开销比较小。它是 `Executors.newCachedThreadPool()` 的默认队列。

## 队列使用场景与典型用例

在实际开发中，我提到过 `Queue` 被广泛使用在生产者 - 消费者场景，比如利用 `BlockingQueue` 来实现，由于其提供的等待机制，我们可以少操心很多协调工作，你可以参考下面样例代码：

 复制代码

```
1 import java.util.concurrent.ArrayBlockingQueue;
2 import java.util.concurrent.BlockingQueue;
3
4 public class ConsumerProducer {
5     public static final String EXIT_MSG = "Good bye!";
6     public static void main(String[] args) {
7         // 使用较小的队列，以更好地在输出中展示其影响
8         BlockingQueue<String> queue = new ArrayBlockingQueue<>(3);
9         Producer producer = new Producer(queue);
10        Consumer consumer = new Consumer(queue);
11        new Thread(producer).start();
12        new Thread(consumer).start();
13    }
```

```
14
15
16 static class Producer implements Runnable {
17     private BlockingQueue<String> queue;
18     public Producer(BlockingQueue<String> q) {
19         this.queue = q;
20     }
21
22     @Override
23     public void run() {
24         for (int i = 0; i < 20; i++) {
25             try{
26                 Thread.sleep(5L);
27                 String msg = "Message" + i;
28                 System.out.println("Produced new item: " + msg);
29                 queue.put(msg);
30             } catch (InterruptedException e) {
31                 e.printStackTrace();
32             }
33         }
34
35         try {
36             System.out.println("Time to say good bye!");
37             queue.put(EXIT_MSG);
38         } catch (InterruptedException e) {
39             e.printStackTrace();
40         }
41     }
42 }
43
44 static class Consumer implements Runnable{
45     private BlockingQueue<String> queue;
46     public Consumer(BlockingQueue<String> q){
47         this.queue=q;
48     }
49
50     @Override
51     public void run() {
52         try{
53             String msg;
54             while(!EXIT_MSG.equalsIgnoreCase( (msg = queue.take()))){
55                 System.out.println("Consumed item: " + msg);
56                 Thread.sleep(10L);
57             }
58             System.out.println("Got exit message, bye!");
59         }catch(InterruptedException e) {
60             e.printStackTrace();
61         }
62     }
63 }
64 }
```



上面是一个典型的生产者 - 消费者样例，如果使用非 Blocking 的队列，那么我们就需要自己去实现轮询、条件判断（如检查 poll 返回值是否 null）等逻辑，如果没有特别的场景要求，Blocking 实现起来代码更加简单、直观。

前面介绍了各种队列实现，在日常的应用开发中，如何进行选择呢？

以 `LinkedBlockingQueue`、`ArrayBlockingQueue` 和 `SynchronousQueue` 为例，我们一起来分析一下，根据需求可以从很多方面考量：

- 考虑应用场景中对队列边界的要求。`ArrayBlockingQueue` 是有明确的容量限制的，而 `LinkedBlockingQueue` 则取决于我们是否在创建时指定，`SynchronousQueue` 则干脆不能缓存任何元素。
- 从空间利用角度，数组结构的 `ArrayBlockingQueue` 要比 `LinkedBlockingQueue` 紧凑，因为其不需要创建所谓节点，但是其初始分配阶段就需要一段连续的空间，所以初始内存需求更大。
- 通用场景中，`LinkedBlockingQueue` 的吞吐量一般优于 `ArrayBlockingQueue`，因为它实现了更加细粒度的锁操作。
- `ArrayBlockingQueue` 实现比较简单，性能更好预测，属于表现稳定的“选手”。
- 如果我们需要实现的是两个线程之间接力性（handoff）的场景，按照[专栏上一讲](#)的例子，你可能会选择 `CountDownLatch`，但是[SynchronousQueue](#)也是完美符合这种场景的，而且线程间协调和数据传输统一起来，代码更加规范。
- 可能令人意外的是，很多时候 `SynchronousQueue` 的性能表现，往往大大超过其他实现，尤其是在队列元素较小的场景。

今天我分析了 Java 中让人眼花缭乱的各种线程安全队列，试图从几个角度，让每个队列的特点更加明确，进而希望减少你在日常工作中使用时的困扰。

## 一课一练

关于今天我们讨论的题目你做到心中有数了吗？今天的内容侧重于 Java 自身的角度，面试官也可能从算法的角度来考察，所以今天留给你的思考题是，指定某种结构，比如栈，用它实现一个 `BlockingQueue`，实现思路是怎样的呢？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



# Java核心技术36讲

—— 前 Oracle 首席工程师  
带你修炼 Java 内功 ——

杨晓峰 前 Oracle 首席工程师



版权归极客邦科技所有，未经许可不得转载

写留言

## 精选留言



sunlight001

这个看着很吃力啊，都没接触过☺

2018-06-21

👍 17



石头狮子

实现课后题过程中把握以下几个维度，

- 1，数据操作的锁粒度。
- 2，计数，遍历方式。
- 3，数据结构空，满时线程的等待方式，有锁或无锁方式。
- 4，使用离散还是连续的存储结构。

2018-06-21

👍 4



crazyone

从上面这些角度，能够理解 ConcurrentLinkedDeque 和 LinkedBlockingQueue 的主要功能区别。这段应该是 "ConcurrentLinkedDeque 和 LinkedBlockingDeque 的主要功能区别"

2018-06-22

👍 3



无呢可称

@Jerry银银。用两个栈可以实现fifo的队列

2018-06-27

👍 2







丘壑

👍 1

栈来实现blockqueue，个人感觉比较好的有

方案一：总共3个栈，其中2个写入栈（A、B），1个消费栈C（消费数据），但是有1个写入栈是空闲的栈（B），随时等待写入，当消费栈(C)中数据为空的时候，消费线程（await），触发数据转移，原写入栈(A)停止写入，由空闲栈（B）接受写入的工作，原写入栈(A)中的数据转移到消费栈（C）中，转移完成后继续（sign）继续消费，2个写入栈，1个消费栈优点是：不会堵塞写入，但是消费会有暂停

方案二：总共4个栈，其中2个写入栈（A、B），2个消费栈（C、D），其中B为空闲的写入栈，D为空闲的消费栈，当消费栈（C）中的数据下降到一定的数量，则触发数据转移，这时候A栈停止写入，由B栈接受写入数据，然后将A栈中的数据转入空闲的消费栈D，当C中的数据消费完了后，则C栈转为空闲，D栈转为激活消费状态，当D栈中的数据消费到一定比例后，重复上面过程，该方案优点即不堵塞写入，也不会造成消费线程暂停

2018-09-13



Jerry银银

👍 1

用栈来实现BlockingQueue，换句话说，用先进后出的数据结构来实现先进先出的数据结构，怎么感觉听起来不那么对劲呢？请指点

2018-06-22



猕猴桃 □盛哥

👍 1

```
{
  "test": [
    [
      89,
      90,
      [
        [
          1093,
          709
        ],
        [
          1056,
          709
        ]
      ]
    ]
  ]
}
```

测试题：这个json用java对象怎么表示？

2018-06-22



灰飞灰猪不会灰飞.烟灭

1

老师 线程池中如果线程已经运行结束则删除该线程。如何判断线程已经运行结束了呢？源码中我看见按照线程的状态，我不清楚这些状态值哪来的。java代码有判断线程状态的方法吗？谢谢老师

2018-06-21

### 作者回复

所谓结束是指terminated？正常的线程池移除工作线程，要么线程意外退出，比如任务抛异常，要么线程闲置，又规定了闲置时间；线程池中线程是把额外封装的，本来下章写了，内容篇幅超标移到后面了，慢慢来；有，建议学会看文档，自己找答案

2018-06-22



石头狮子

1

实现课后题过程中把握以下几个维度，

- 1，数据操作的锁粒度。
- 2，计数，遍历方式。
- 3，数据结构空，满时线程的等待方式，有锁或无锁方式。
- 4，使用离散还是连续的存储结构。

2018-06-21



小飞哥 超級會員

0

向我们这些写业务代码的应该如何使用并发类和框架？

2018-10-11



null

0

SynchronousQueue，删除操作依赖插入操作，而插入操作又依赖删除操作，死锁了么？SynchronousQueue 一般应用在啥场景呢？

2018-09-11



酱了个油

0

队列的一个问题是不能持久化、不能做到分布式，有时候考虑到系统可靠性，使用的机会不多。杨老师可以给一些使用队列的例子吗？

2018-08-05



纯爷们

0

Java并发包里的东西平时基本没怎么接触，这块知识太缺乏了，看这篇文章痛苦！

2018-07-19



汉彬

0

用栈实现BlockingQueue，我的理解是：栈是LIFO，BlockingQueue是FIFO，因此需要两个栈。take时先把栈A全部入栈到栈B，然后栈B出栈得到目标元素；put时把栈B全部入栈到栈A，然后栈A再入栈目标元素。相当于倒序一下。

不知道理解对不对，请老师指出。

2018-07-17



爱新觉罗老流氓

0

杨老师，“与弱一致性对应的，就是我介绍过的同步容器常见的行为“fast-fail”，也就是检测到容器在遍历过程中发生了修改，则抛出 `ConcurrentModificationException`，不再继续遍历。”

这一段落里，快速失败的英文在doc上是“fail-fast”，在ArrayList源码中文档可以搜到。还有，同步容器不应该是“fail-safe”吗？

2018-07-03

作者回复

谢谢指出，我查查是不是我记反了

2018-07-04



Invoker.C

0

求老师解答一个困扰已久的问题，就是初始化`arrayblockingqueue`的时候，`capacity`的大小如何评估和设置？望解答

2018-06-27

作者回复

不清楚你的硬件、业务特点，一个大概原则是尽量让进和出的速率一致，不然出慢，进就block，反过来也不好；实际操作上，你试试找时机检查`remaincapacity`，就可以判断进出速率的对比

2018-06-28



夏洛克的救赎

0

老师你好，问个问题外问题，在jdk10源码 `string`类中，成员变量`coder`起到什么作用？如何理解？

2018-06-21

作者回复

编码，区分拉丁和非拉丁语系

2018-06-22