目录

19 LinkedBlockingQueue 源码解析

更新时间：2019-10-11 17:02:21

> 从不浪费时间的人，没有工夫抱怨时间不够。
>
> ——杰弗逊

引导语

说到队列，大家的反应可能是我从来都没有用过，应该是不重要的 API 吧。如果这么想，那就大错特错了，我们平时使用到的线程池、读写锁、消息队列等等技术和框架，底层原理都是队列，所以我们万万不可轻视队列，队列是很多高级 API 的基础，学好队列，对自己深入 Java 学习非常重要。

本文主要以 LinkedBlockingQueue 队列为例，详细描述一下底层具体的实现。

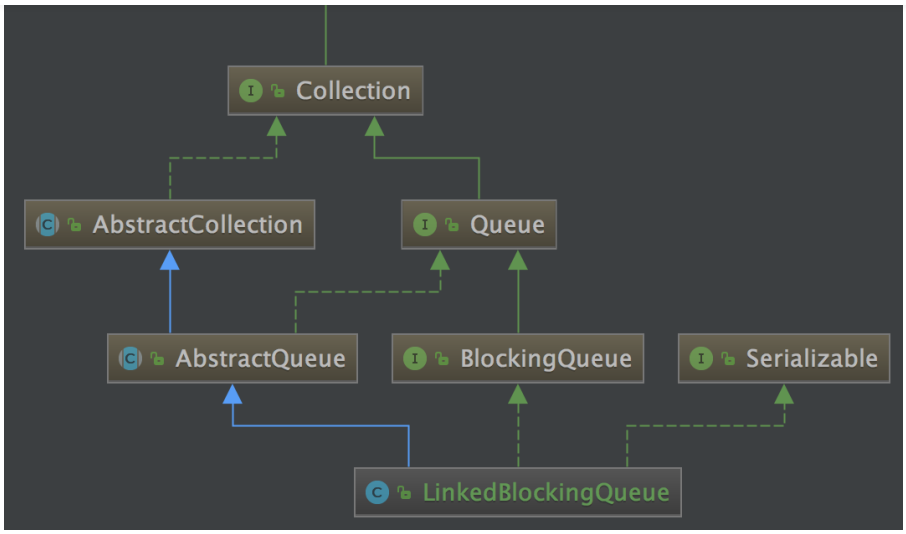
1 整体架构

LinkedBlockingQueue 中文叫做链表阻塞队列，这个命名很好，从命名上就知道其底层数据结构是链表，并且队列是可阻塞的。接下来，我们就从整体结构上看看 LinkedBlockingQueue。

1.1 类图

首先我们来看下 LinkedBlockingQueue 类图，如下：

目录



从类图中，我们大概可以看出两条路径：

1. AbstractQueue -> AbstractCollection -> Collection ->Iterable 这条路径依赖，主要是想复用 Collection 和 迭代器的一些操作，这些我们在说集合的时候，都知道这些类是干什么，能干什么，就不细说了；
2. BlockingQueue -> Queue -> Collection，BlockingQueue 和 Queue 是新出来的两个接口，我们重点说一下。

Queue 是最基础的接口，几乎所有的队列实现类都会实现这个接口，该接口定义出了队列的三大类操作：

新增操作：

1. add 队列满的时候抛出异常；
2. offer 队列满的时候返回 false。

查看并删除操作：

1. remove 队列空的时候抛异常；
2. poll 队列空的时候返回 null。

只查看不删除操作：

1. element 队列空的时候抛异常；
2. peek 队列空的时候返回 null。

一共 6 种方法，除了以上分类方法，也可以分成两类：

1. 遇到队列满或空的时候，抛异常，如 add、remove、element；
2. 遇到队列满或空的时候，返回特殊值，如 offer、poll、peek。

实际上，这些都比较难记忆。每次需要使用的时候，我都会看会源码，才能想起这个方法是抛异常还是返回特殊值。

BlockingQueue 在 Queue 的基础上加上了阻塞的概念，比如一直阻塞，还是阻塞一段时间。为了方便记忆，我们画一个表格，如下：

新增操作–队列满	add	offer 返回 false	put	offer 过超时时间返回 false
查看并删除操作–队列空	remove	poll 返回 null	take	poll 过超时时间返回 null
只查看不删除操作–队列空	element	peek 返回 null	暂无	暂无

PS: remove 方法，BlockingQueue 类注释中定义的是抛异常，但 LinkedBlockingQueue 中 remove 方法实际是返回 false。
从表格中可以看到，在新增和查看并删除两大类操作上，BlockingQueue 增加了阻塞的功能，而且可以选择一直阻塞，或者阻塞一段时间后，返回特殊值。

1.2 类注释

我们看看从 LinkedBlockingQueue 的类注释中能得到那些信息：

1. 基于链表的阻塞队列，其底层的数据结构是链表；
2. 链表维护先入先出队列，新元素被放在队尾，获取元素从队头部拿；
3. 链表大小在初始化的时候可以设置，默认是 Integer 的最大值；
4. 可以使用 Collection 和 Iterator 两个接口的所有操作，因为实现了两者的接口。

1.3 内部构成

LinkedBlockingQueue 内部构成简单来说，分成三个部分：链表存储 + 锁 + 迭代器，我们来看下源码。

```
// 链表结构 begin
//链表的元素
static class Node<E> {
    E item;

    //当前元素的下一个，为空表示当前节点是最后一个
    Node<E> next;

    Node(E x) { item = x; }
}

//链表的容量，默认 Integer.MAX_VALUE
private final int capacity;

//链表已有元素大小，使用 AtomicInteger，所以是线程安全的
private final AtomicInteger count = new AtomicInteger();

//链表头
transient Node<E> head;

//链表尾
private transient Node<E> last;
// 链表结构 end

// 锁 begin
//take 时的锁
private final ReentrantLock takeLock = new ReentrantLock();

// take 的条件队列，condition 可以简单理解为基于 ASQ 同步机制建立的条件队列
```

```
private final ReentrantLock putLock = new ReentrantLock();

// put 的条件队列
private final Condition notFull = putLock.newCondition();
// 锁 end

// 迭代器
// 实现了自己的迭代器
private class Itr implements Iterator<E> {
.....................
}
```

从代码上来看，结构是非常清晰的，三种结构各司其职：

1. 链表的作用是为了保存当前节点，节点中的数据可以是任意东西，是一个泛型，比如说队列被应用到线程池时，节点就是线程，比如队列被应用到消息队列中，节点就是消息，节点的含义主要看队列被使用的场景；
2. 锁有 take 锁和 put 锁，是为了保证队列操作时的线程安全，设计两种锁，是为了 take 和 put 两种操作可以同时进行，互不影响。

1.4 初始化

初始化有三种方式：

1. 指定链表容量大小；
2. 不指定链表容量大小，默认是 Integer 的最大值；
3. 对已有集合数据进行初始化。

源码如下：

```
// 不指定容量，默认 Integer 的最大值
public LinkedBlockingQueue() {
    this(Integer.MAX_VALUE);
}
// 指定链表容量大小，链表头尾相等，节点值（item）都是 null
public LinkedBlockingQueue(int capacity) {
    if (capacity <= 0) throw new IllegalArgumentException();
    this.capacity = capacity;
    last = head = new Node<E>(null);
}

// 已有集合数据进行初始化
public LinkedBlockingQueue(Collection<? extends E> c) {
    this(Integer.MAX_VALUE);
    final ReentrantLock putLock = this.putLock;
    putLock.lock(); // Never contended, but necessary for visibility
    try {
        int n = 0;
        for (E e : c) {
            // 集合内的元素不能为空
            if (e == null)
                throw new NullPointerException();
            // capacity 代表链表的大小，在这里是 Integer 的最大值
            // 如果集合类的大小大于 Integer 的最大值，就会报错
            // 其实这个判断完全可以放在 for 循环外面，这样可以减少 Integer 的最大值次循环(最坏情
            if (n == capacity)
                throw new IllegalStateException("Queue full");
```

```
            count.set(n);
        } finally {
            putLock.unlock();
        }
    }
```

对于初始化源码，我们说明两点：

1. 初始化时，容量大小是不会影响性能的，只影响在后面的使用，因为初始化队列太小，容易导致没有放多少就会报队列已满的错误；
2. 在对给定集合数据进行初始化时，源码给了一个不优雅的示范，我们不反对在每次 for 循环的时候，都去检查当前链表的大小是否超过容量，但我们希望在 for 循环开始之前就做一步这样的工作。举个列子，给定集合大小是 1 w，链表大小是 9k，按照现在代码实现，只能在 for 循环 9k 次时才能发现，原来给定集合的大小已经大于链表大小了，导致 9k 次循环都是在浪费资源，还不如在 for 循环之前就 check 一次，如果 1w > 9k，直接报错即可。

2 阻塞新增

新增有多种方法，如：add、put、offer，三者的区别上文有说。我们拿 put 方法为例，put 方法在碰到队列满的时候，会一直阻塞下去，直到队列不满时，并且自己被唤醒时，才会继续去执行，源码如下：

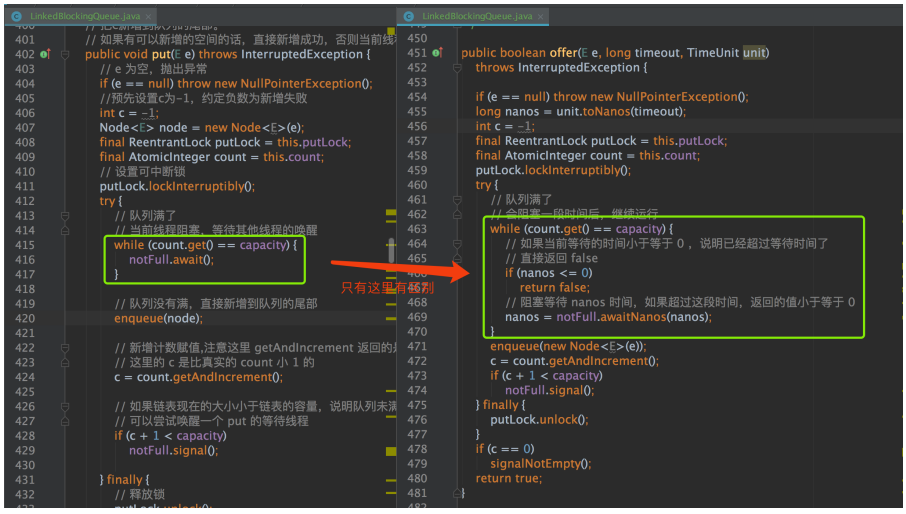
```
// 把e新增到队列的尾部。
// 如果有可以新增的空间的话，直接新增成功，否则当前线程陷入等待
public void put(E e) throws InterruptedException {
    // e 为空，抛出异常
    if (e == null) throw new NullPointerException();
    // 预先设置 c 为 -1，约定负数为新增失败
    int c = -1;
    Node<E> node = new Node<E>(e);
    final ReentrantLock putLock = this.putLock;
    final AtomicInteger count = this.count;
    // 设置可中断锁
    putLock.lockInterruptibly();
    try {
        // 队列满了
        // 当前线程阻塞，等待其他线程的唤醒(其他线程 take 成功后就会唤醒此处被阻塞的线程)
        while (count.get() == capacity) {
            // await 无限等待
            notFull.await();
        }

        // 队列没有满，直接新增到队列的尾部
        enqueue(node);

        // 新增计数赋值,注意这里 getAndIncrement 返回的是旧值
        // 这里的 c 是比真实的 count 小 1 的
        c = count.getAndIncrement();

        // 如果链表现在的大小 小于链表的容量，说明队列未满
        // 可以尝试唤醒一个 put 的等待线程
        if (c + 1 < capacity)
            notFull.signal();
```

```
    }
    // c==0，代表队列里面有一个元素
    // 会尝试唤醒一个take的等待线程
    if (c == 0)
        signalNotEmpty();
}
// 入队，把新元素放到队尾
private void enqueue(Node<E> node) {
    last = last.next = node;
}
```

从源码中我们可以总结以下几点：

1. 往队列新增数据，第一步是上锁，所以新增数据是线程安全的；
2. 队列新增数据，简单的追加到链表的尾部即可；
3. 新增时，如果队列满了，当前线程是会被阻塞的，阻塞的底层使用是锁的能力，底层实现其它也和队列相关，原理我们在锁章节会说到；
4. 新增数据成功后，在适当时机，会唤起 put 的等待线程（队列不满时），或者 take 的等待线程（队列不为空时），这样保证队列一旦满足 put 或者 take 条件时，立马就能唤起阻塞线程，继续运行，保证了唤起的时机不被浪费。

以上就是 put 方法的原理，至于 offer 方法阻塞超过一端时间后，仍未成功，就会直接返回默认值的实现，和 put 方法相比只修改了几行代码，如下截图：



3 阻塞删除

删除的方法也很多，我们主要看两个关键问题：

1. 删除的原理是怎样的；
2. 查看并删除和只查看不删除两种的区别是如何实现的。

首先我们来看第一个问题，我们以 take 方法为例，说明一下查看并删除的底层源码：

```
// 阻塞拿数据
public E take() throws InterruptedException {
    E x;
    // 默认负数，代表失败
    int c = -1;
```

```
    takeLock.lockInterruptibly();
    try {
        // 空队列时，阻塞，等待其他线程唤醒
        while (count.get() == 0) {
            notEmpty.await();
        }
        // 非空队列，从队列的头部拿一个出来
        x = dequeue();
        // 减一计算，注意 getAndDecrement 返回的值是旧值
        // c 比真实的 count 大1
        c = count.getAndDecrement();

        // 如果队列里面有值，从 take 的等待线程里面唤醒一个。
        // 意思是队列里面有值啦,唤醒之前被阻塞的线程
        if (c > 1)
            notEmpty.signal();
    } finally {
        // 释放锁
        takeLock.unlock();
    }
    // 如果队列空闲还剩下一个，尝试从 put 的等待线程中唤醒一个
    if (c == capacity)
        signalNotFull();
    return x;
}
// 队头中取数据
private E dequeue() {
    Node<E> h = head;
    Node<E> first = h.next;
    h.next = h; // help GC
    head = first;
    E x = first.item;
    first.item = null;// 头节点指向 null，删除
    return x;
}
```

整体流程和 put 很相似，都是先上锁，然后从队列的头部拿出数据，如果队列为空，会一直阻塞到队列有值为止。

而查看不删除元素更加简单，直接把队列头的数据拿出来即可，我们以 peek 为例，源码如下：

```
// 查看并不删除元素，如果队列为空，返回 null
public E peek() {
    // count 代表队列实际大小，队列为空，直接返回 null
    if (count.get() == 0)
        return null;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lock();
    try {
        // 拿到队列头
        Node<E> first = head.next;
        // 判断队列头是否为空，并返回
        if (first == null)
            return null;
        else
            return first.item;
    } finally {
        takeLock.unlock();
    }
}
```

除，一个不会删除队头数据。

4 总结

本文通过 LinkedBlockingQueue 的源码，来介绍了下链表队列，当队列满和空的场景下，新增和删除数据时，队列有啥变化。

队列本身就是一个阻塞工具，我们可以把这个工具应用到各种阻塞场景中，比如说队列应用到线程池，当线程池跑满时，我们把新的请求都放到阻塞队列中等待；队列应用到消息队列，当消费者处理能力有限时，我们可以把消息放到队列中等待，让消费者慢慢消费；每应用到一个新的场景中，都是一个新的技术工具，所以学好队列，用处很大。

← 18 场景集合：并发 List、Map的应用场景

20 SynchronousQueue 源码解析 →

精选留言 10

欢迎在这里发表留言，作者筛选后可公开显示

慕粉2311555628

// c==0，代表队列里面有一个元素 // 会尝试唤醒一个take的等待线程 if (c == 0) signalNotEmpty(); } 老师，你好，对这段代码不是很理解，为什么要到只剩下一个才去唤醒take线程，不应该是if(c>=0) 么，还有下面的 删除逻辑 要到c==capacity 才去唤醒put线程 ，这是为什么啊

0 回复 2020-01-20

记住没有

老师h.next = h; 这行代码是让h节点脱离链表，如果h.next=null的话，是不是就相当把first设置为null,后面就代码就会报错。

0 回复 2020-01-10

慕村6418685

老师开一门视频课吧，专门讲java源码

3 回复 2019-12-26

慕尼黑7546459

老师，关于出队方法dequeue()的 h.next = h; // help GC 这行代码，为什么不能h.next = null，自己查了下，网上解释说是为了区分是被task了还是队列已结束，但是还是不太理解，麻烦老师帮解释一下，谢谢🙏

0 回复 2019-12-06

文贺 回复 **慕尼黑7546459**

回复 2019-12-08 13:09:20

文贺 回复 **慕尼黑7546459**

你可以在 h.next = h 这一行 debug 一下，然后再 watch 里面使 h.next = null。

回复 2019-12-08 13:09:57

---

qq_Ezio_1

老师 之前那个微信交流群我没进

0 回复 2019-11-08

初一 回复 **qq_Ezio_1**

你先加我微信吧 any97501，我邀请你进微信群。

回复 2019-11-08 11:08:20

---

student19

队列在多节点项目不适合吧，只能是单节点项目吧，实际项目用的不多吧

0 回复 2019-11-05

文贺 回复 **student19**

多节点项目指的是？队列实际工作中还是有一些场景可以用的，我们在 25 中有说，平时简单业务开发中的确用的不多，因为需要用的地方，JDK 底层线程池，锁，公司的中间件都帮我包装好了。

回复 2019-11-07 10:50:11

煮沧海

老师，有个小疑惑没理解透，希望得到您的提示下。 删除头结点并返回调用dequeue()方法，删除first节点的方法难道不是将first.next = head.next吗，为什么将first.item = null就表示删除了？

0 回复 2019-10-31

文贺 回复 **煮沧海**

// 取出头节点 Node h = head; // 头节点的下一个节点为 first Node first = h.next; // 使 h 的下一个节点指向自己 h.next = h; // help GC // 给链表头赋值 head = first; // 取出链表头值 E x = first.item; // 旧头节点指向 null，帮助 GC first.item = null; // 返回旧头节点值 return x; 整个方法是这样的，我加了注释，你在看看哈

回复 2019-10-31 19:46:24

煮沧海 回复 **文贺**

老师你看，问题就出现了。 一开始链表： head.. first.. last。 然后head. next =head，变成了head.. head.. first.. last。head从指向first变成指向head，但是后面那个head的next还是指向first啊。我理解的这里是不是欠缺了什么？ 然后head =first，变成了first.. head.. first.. last。 最后取出first头，最后置为Null。 老师求指教，感激不尽。

回复 2019-10-31 20:35:58

文贺 回复 **煮沧海**

同学你好，变成head..head..first..tail这里就不对了，是没有两个head的，的确能看出你比较纠结，我也经历过和你一样的阶段，有两种办法1：拿出纸笔画一画，2：尝试 debug 一下，如果你只是看的话，估计会很晕，帮你写好了 debug的代码，你跟 take 方法进去看看： LinkedBlo

目录

玩名堂s

put方法中: while (count.get() == capacity) { // await 无限等待 notFull.await(); } 这里的检查是否 “满” 的操作为什么是while不是if呢?

1 回复 2019-10-19

文贺 回复 **玩名堂s**

同学你好，在队列满的情况下，代码会在 notFull.await() 这一行阻塞住，如果有其他线程从队列中拿数据了，此时队列被唤醒，是从 notFull.await() 这行代码处被唤醒，接着继续执行，这时会再次走 while 循环，再次校验队列是不是满的，如果不是满的，就会继续往下走。 写 while 不写 if 的原因，就在于再次校验队列是不是满的，因为 Java 线程在很小几率下会虚假唤醒，假如此时 Java 线程被虚假唤醒，再次走 while 循环就能被校验住，个人理解哈。

回复 2019-10-21 22:18:13

慕码人6169125

Java8里面remove方法返回的是Boolean类型，源码里面写的。最上面的表格这个地方写错了

0 回复 2019-10-11

文贺 回复 **慕码人6169125**

同学你好，感谢提醒，已加了一个 PS，看 BlockingQueue 类注释画的表格，没注意 LinkedBlockingQueue 的实现不同，已在表格下加了备注。

回复 2019-10-11 17:09:09

慕设计5408150

老师我们源码的github是什么

0 回复 2019-10-10

初一 回复 **慕设计5408150**

源码解析：https://github.com/luanqiu/java8 文章 demo：https://github.com/luanqiu/java8_demo

回复 2019-10-11 09:56:04

千学不如一看，千看不如一练