目录

20 SynchronousQueue 源码解析

更新时间：2019-10-15 11:19:50

> 只有在那崎岖的小路上不畏艰险奋勇攀登的人,才有希望达到光辉的顶点。
>
> ——马克思

引导语

SynchronousQueue 是比较独特的队列，其本身是没有容量大小，比如我放一个数据到队列中，我是不能够立马返回的，我必须等待别人把我放进去的数据消费掉了，才能够返回。SynchronousQueue 在消息队列技术中间件中被大量使用，本文就来从底层实现来看下 SynchronousQueue 到底是如何做到的。

1 整体架构

SynchronousQueue 的整体设计比较抽象，在内部抽象出了两种算法实现，一种是先入先出的队列，一种是后入先出的堆栈，两种算法被两个内部类实现，而直接对外的 put，take 方法的实现就非常简单，都是直接调用两个内部类的 transfer 方法进行实现，整体的调用关系如下图所示：

目录

1.1 类注释

源码的类注释往往能给我带来很多疑问和有用的信息，我们来看下类注释都说了什么：

1. 队列不存储数据，所以没有大小，也无法迭代；
2. 插入操作的返回必须等待另一个线程完成对应数据的删除操作，反之亦然；
3. 队列由两种数据结构组成，分别是后入先出的堆栈和先入先出的队列，堆栈是非公平的，队列是公平的。

看到类注释，大家是不是有一些疑问，比如第二点是如何做到的？堆栈又是如何实现的呢？接下来我们一点一点揭晓。

1.2 类图

SynchronousQueue 整体类图和 LinkedBlockingQueue 相似，都是实现了 BlockingQueue 接口，但因为其不储存数据结构，有一些方法是没有实现的，比如说 isEmpty、size、contains、remove 和迭代等方法，这些方法都是默认实现，如下截图：

目录

```
006         *
007         * @param o the element
008         * @return {@code false}
009         */
010        public boolean contains(Object o) { return false; }
013
014        /**
015         * Always returns {@code false}.
016         * A {@code SynchronousQueue} has no internal capacity.
017         *
018         * @param o the element to remove
019         * @return {@code false}
020         */
021        public boolean remove(Object o) { return false; }
024
025        /**
026         * Returns {@code false} unless the given collection is empty.
027         * A {@code SynchronousQueue} has no internal capacity.
028         *
029         * @param c the collection
030         * @return {@code false} unless given collection is empty
031         */
032        public boolean containsAll(Collection<?> c) { return c.isEmpty(); }
035
036        /**
037         * Always returns {@code false}.
038         * A {@code SynchronousQueue} has no internal capacity.
039         *
040         * @param c the collection
041         * @return {@code false}
042         */
043        public boolean removeAll(Collection<?> c) { return false; }
```

和容量相关的方法都是默认实现

1.3 结构细节

SynchronousQueue 底层结构和其它队列完全不同，有着独特的两种数据结构：队列和堆栈。

我们一起来看下数据结构：

```
// 堆栈和队列共同的接口
// 负责执行 put or take
abstract static class Transferer<E> {
    // e 为空的，会直接返回特殊值，不为空会传递给消费者
    // timed 为 true，说明会有超时时间
    abstract E transfer(E e, boolean timed, long nanos);
}

// 堆栈 后入先出 非公平
// Scherer-Scott 算法
static final class TransferStack<E> extends Transferer<E> {
}

// 队列 先入先出 公平
static final class TransferQueue<E> extends Transferer<E> {
}

private transient volatile Transferer<E> transferer;

// 无参构造器默认为非公平的
public SynchronousQueue(boolean fair) {
    transferer = fair ? new TransferQueue<E>() : new TransferStack<E>();
}
```

从源码中我们可以得到几点：

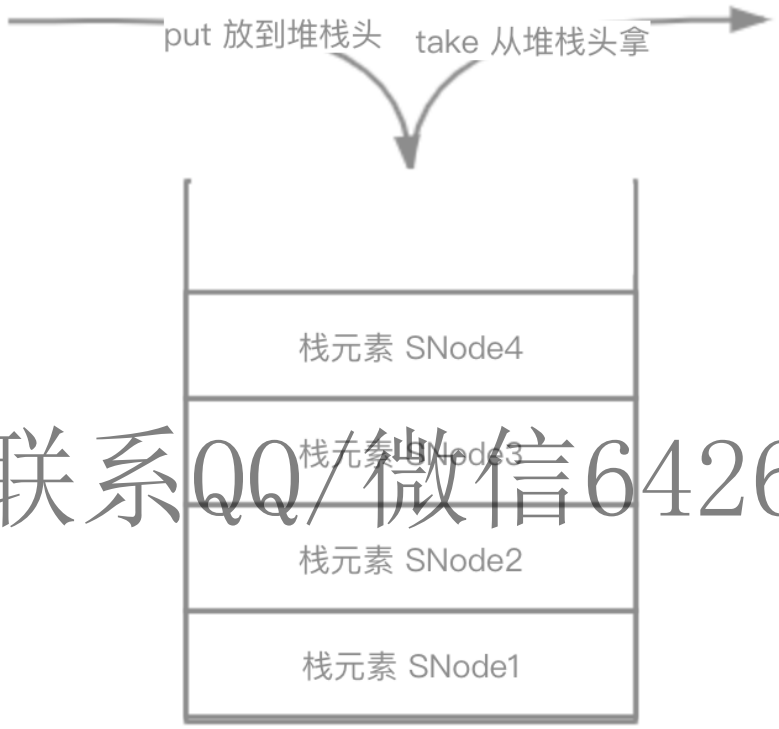
2. 在我们初始化的时候，是可以选择是使用堆栈还是队列的，如果你不选择，默认的就是堆栈，类注释中也说明了这一点，堆栈的效率比队列更高。

接下来我们来看下堆栈和队列的具体实现。

2 非公平的堆栈

2.1 堆栈的结构

首先我们来介绍下堆栈的整体结构，如下：



从上图中我们可以看到，我们有一个大的堆栈池，池的开口叫做堆栈头，put 的时候，就往堆栈池中放数据。take 的时候，就从堆栈池中拿数据，两者操作都是在堆栈头上操作数据，从图中可以看到，越靠近堆栈头，数据越新，所以每次 take 的时候，都会拿到堆栈头的最新数据，这就是我们说的后入先出，也就是非公平的。

图中 SNode 就是源码中栈元素的表示，我们看下源码：

```
static final class SNode {
    // 栈的下一个，就是被当前栈压在下面的栈元素
    volatile SNode next;
    // 节点匹配，用来判断阻塞栈元素能被唤醒的时机
    // 比如我们先执行 take，此时队列中没有数据，take 被阻塞了，栈元素为 SNode1
    // 当有 put 操作时，会把当前 put 的栈元素赋值给 SNode1 的 match 属性，并唤醒 take 操作
    // 当 take 被唤醒，发现 SNode1 的 match 属性有值时，就能拿到 put 进来的数据，从而返回
    volatile SNode match;
    // 栈元素的阻塞是通过线程阻塞来实现的，waiter 为阻塞的线程
    volatile Thread waiter;
    // 未投递的消息，或者未消费的消息
```

目录

2.2 入栈和出栈

入栈指的是使用 put 等方法，把数据放到堆栈池中，出栈指的使用 take 等方法，把数据从堆栈池中拿出来，操作的对象都是堆栈头，虽然两者的一个是从堆栈头拿数据，一个是放数据，但底层实现的方法却是同一个，源码如下：

```
// transfer 方法思路比较复杂，因为 take 和 put 两个方法都揉在了一起
@SuppressWarnings("unchecked")
E transfer(E e, boolean timed, long nanos) {
    SNode s = null; // constructed/reused as needed
    // e 为空，说明是 take 方法，不为空是 put 方法
    int mode = (e == null) ? REQUEST : DATA;
    // 自旋
    for (;;) {
        // 拿出头节点，有几种情况
        // 1：头节点为空，说明队列中还没有数据
        // 2：头节点不为空，并且是 take 类型的，说明头节点线程正等着拿数据。
        // 3：头节点不为空，并且是 put 类型的，说明头节点线程正等着放数据。
        SNode h = head;
        // 栈头为空，说明队列中还没有数据。
        // 栈头不为空，并且栈头的类型和本次操作一致，比如都是 put，那么就把
        // 本次 put 操作放到该栈头的前面即可，让本次 put 能够先执行
        if (h == null || h.mode == mode) {  // empty or same-mode
            // 设置了超时时间，并且 e 进栈或者出栈要超时了，
            // 就会丢弃本次操作，返回 null 值。
            // 如果栈头此时被取消了，丢弃栈头，取下一个节点继续消费
            if (timed && nanos <= 0) {      // can't wait
                // 栈头操作被取消
                if (h != null && h.isCancelled())
                    // 丢弃栈头，把栈头后一个元素作为栈头
                    casHead(h, h.next);     // pop cancelled node
                //栈头是空的，直接返回 null
                else
                    return null;
            // 没有超时，直接把 e 作为新的栈头
            } else if (casHead(h, s = snode(s, e, h, mode))) {
                // e 等待出栈，一种是空队列 take，一种是 put
                SNode m = awaitFulfill(s, timed, nanos);
                if (m == s) {               // wait was cancelled
                    clean(s);
                    return null;
                }
                // 本来 s 是栈头的，现在 s 不是栈头了，s 后面又来了一个数，把新的数据作为栈头
                if ((h = head) != null && h.next == s)
                    casHead(h, s.next);     // help s's fulfiller
                return (E) ((mode == REQUEST) ? m.item : s.item);
            }
        // 栈头正在等待其他线程 put 或 take
        // 比如栈头正在阻塞，并且是 put 类型，而此次操作正好是 take 类型，走此处
        } else if (!isFulfilling(h.mode)) { // try to fulfill
            // 栈头已经被取消，把下一个元素作为栈头
            if (h.isCancelled())            // already cancelled
                casHead(h, h.next);         // pop and retry
            // snode 方法第三个参数 h 代表栈头，赋值给 s 的 next 属性
            else if (casHead(h, s=snode(s, e, h, FULFILLING|mode))) {
                for (;;) { // loop until matched or waiters disappear
                    // m 就是栈头，通过上面 snode 方法刚刚赋值
                    SNode m = s.next;       // m is s's match
                    if (m == null) {        // all waiters are gone
                        casHead(s, null);   // pop fulfill node
```

目录

```
                SNode mn = m.next;
                // tryMatch 非常重要的方法，两个作用：
                // 1 唤醒被阻塞的栈头 m，2 把当前节点 s 赋值给 m 的 match 属性
                // 这样栈头 m 被唤醒时，就能从 m.match 中得到本次操作 s
                // 其中 s.item 记录着本次的操作节点，也就是记录本次操作的数据
                if (m.tryMatch(s)) {
                    casHead(s, mn);     // pop both s and m
                    return (E) ((mode == REQUEST) ? m.item : s.item);
                } else                  // lost match
                    s.casNext(m, mn);   // help unlink
            }
        }
    } else {                            // help a fulfiller
        SNode m = h.next;               // m is h's match
        if (m == null)                  // waiter is gone
            casHead(h, null);           // pop fulfilling node
        else {
            SNode mn = m.next;
            if (m.tryMatch(h))          // help match
                casHead(h, mn);         // pop both h and m
            else                        // lost match
                h.casNext(m, mn);       // help unlink
        }
    }
  }
}
```

从源码中密密麻麻的注释，我们就可以看出来此方法比较复杂，我们总结一下大概的操作思路：

1. 判断是 put 方法还是 take 方法；
2. 判断栈头数据是否为空，如果为空或者栈头的操作和本次操作一致，是的话走 3，否则走 5；
3. 判断操作有无设置超时时间，如果设置了超时时间并且已经超时，返回 null，否则走 4；
4. 如果栈头为空，把当前操作设置成栈头，或者栈头不为空，但栈头的操作和本次操作相同，也把当前操作设置成栈头，并看看其它线程能否满足自己，不能满足则阻塞自己。比如当前操作是 take，但队列中没有数据，则阻塞自己；
5. 如果栈头已经是阻塞住的，需要别人唤醒的，判断当前操作能否唤醒栈头，可以唤醒走 6，否则走 4；
6. 把自己当作一个节点，赋值到栈头的 match 属性上，并唤醒栈头节点；
7. 栈头被唤醒后，拿到 match 属性，就是把自己唤醒的节点的信息，返回。

在整个过程中，有一个节点阻塞的方法，实现原理如下：

```
SNode awaitFulfill(SNode s, boolean timed, long nanos) {
    // deadline 死亡时间，如果设置了超时时间的话，死亡时间等于当前时间 + 超时时间，否则就是
    final long deadline = timed ? System.nanoTime() + nanos : 0L;
    Thread w = Thread.currentThread();
    // 自旋的次数，如果设置了超时时间，会自旋 32 次，否则自旋 512 次。
    // 比如本次操作是 take 操作，自选次数后，仍没有其他线程 put 数据进来
    // 就会阻塞，有超时时间的，会阻塞固定的时间，否则一致阻塞下去
    int spins = (shouldSpin(s) ?
                 (timed ? maxTimedSpins : maxUntimedSpins) : 0);
    for (;;) {
        // 当前线程有无被打断，如果过了超时时间，当前线程就会被打断
        if (w.isInterrupted())
            s.tryCancel();
```

```
            if (timed) {
                nanos = deadline - System.nanoTime();
                // 超时了，取消当前线程的等待操作
                if (nanos <= 0L) {
                    s.tryCancel();
                    continue;
                }
            }
            // 自选次数减少 1
            if (spins > 0)
                spins = shouldSpin(s) ? (spins-1) : 0;
            // 把当前线程设置成 waiter，主要是通过线程来完成阻塞和唤醒
            else if (s.waiter == null)
                s.waiter = w; // establish waiter so can park next iter
            else if (!timed)
                // 通过 park 进行阻塞，这个我们在锁章节中会说明
                LockSupport.park(this);
            else if (nanos > spinForTimeoutThreshold)
                LockSupport.parkNanos(this, nanos);
        }
    }
```

从节点阻塞代码中，我们可以发现，其阻塞的策略，并不是一上来就阻塞住，而是在自旋一定次数后，仍然没有其它线程来满足自己的要求时，才会真正的阻塞住。

3 公平的队列

首先我们来看一下队列中的每个元素的组成：

```
/** 队列头 */
transient volatile QNode head;
/** 队列尾 */
transient volatile QNode tail;

// 队列的元素
static final class QNode {
    // 当前元素的下一个元素
    volatile QNode next;
    // 当前元素的值，如果当前元素被阻塞住了，等其他线程来唤醒自己时，其他线程
    // 会把自己 set 到 item 里面
    volatile Object item;        // CAS'ed to or from null
    // 可以阻塞住的当前线程
    volatile Thread waiter;      // to control park/unpark
    // true 是 put，false 是 take
    final boolean isData;
}
```

公平的队列主要使用的是 TransferQueue 内部类的 transfer 方法，我们一起来看下源码：

```
E transfer(E e, boolean timed, long nanos) {

    QNode s = null; // constructed/reused as needed
    // true 是 put，false 是 get
    boolean isData = (e != null);

    for (;;) {
        // 队列头和尾的临时变量,队列是空的时候，t=h
```

目录

```
// 虽然这种 continue 非常耗cpu，但感觉不会碰到这种情况
// 因为 tail 和 head 在 TransferQueue 初始化的时候，就已经被赋值空节点了
if (t == null || h == null)
    continue;
// 首尾节点相同，说明是空队列
// 或者尾节点的操作和当前节点操作一致
if (h == t || t.isData == isData) {
    QNode tn = t.next;
    // 当 t 不是 tail 时，说明 tail 已经被修改过了
    // 因为 tail 没有被修改的情况下，t 和 tail 必然相等
    // 因为前面刚刚执行赋值操作：  t = tail
    if (t != tail)
        continue;
    // 队尾后面的值还不为空，t 还不是队尾，直接把 tn 赋值给 t，这是一步加强校验。
    if (tn != null) {
        advanceTail(t, tn);
        continue;
    }
    //超时直接返回 null
    if (timed && nanos <= 0)        // can't wait
        return null;
    //构造node节点
    if (s == null)
        s = new QNode(e, isData);
    //如果把 e 放到队尾失败，继续递归放进去
    if (!t.casNext(null, s))        // failed to link in
        continue;

    advanceTail(t, s);              // swing tail and wait
    // 阻塞住自己
    Object x = awaitFulfill(s, e, timed, nanos);
    if (x == s) {                   // wait was cancelled
        clean(t, s);
        return null;
    }

    if (!s.isOffList()) {           // not already unlinked
        advanceHead(t, s);          // unlink if head
        if (x != null)              // and forget fields
            s.item = s;
        s.waiter = null;
    }
    return (x != null) ? (E)x : e;
// 队列不为空，并且当前操作和队尾不一致
// 也就是说当前操作是队尾是对应的操作
// 比如说队尾是因为 take 被阻塞的，那么当前操作必然是 put
} else {                            // complementary-mode
    // 如果是第一次执行，此处的 m 代表就是 tail
    // 也就是这行代码体现出队列的公平，每次操作时，从头开始按照顺序进行操作
    QNode m = h.next;               // node to fulfill
    if (t != tail || m == null || h != head)
        continue;                   // inconsistent read

    Object x = m.item;
    if (isData == (x != null) ||    // m already fulfilled
        x == m ||                   // m cancelled
        // m 代表栈头
        // 这里把当前的操作值赋值给阻塞住的 m 的 item 属性
        // 这样 m 被释放时，就可得到此次操作的值
        !m.casItem(x, e)) {         // lost CAS
        advanceHead(h, m);          // dequeue and retry
        continue;
```

```
            // 释放队头阻塞节点
            LockSupport.unpark(m.waiter);
            return (x != null) ? (E)x : e;
        }
    }
}
```

源码比较复杂，我们需要搞清楚的是，线程被阻塞住后，当前线程是如何把自己的数据传给阻塞线程的。为了方便说明，我们假设线程 1 往队列中 take 数据，被阻塞住了，变成阻塞线程 A，然后线程 2 开始往队列中 put 数据 B，大致的流程是这样的：

1. 线程 1 从队列中拿数据，发现队列中没有数据，于是被阻塞，成为 A；
2. 线程 2 往队尾 put 数据，会从队尾往前找到第一个被阻塞的节点，假设此时能找到的就是节点 A，然后线程 B 把将 put 的数据放到节点 A 的 item 属性里面，并唤醒线程 1；
3. 线程 1 被唤醒后，就能从 A.item 里面拿到线程 2 put 的数据了，线程 1 成功返回。

从这个过程中，我们能看出公平主要体现在，每次 put 数据的时候，都 put 到队尾上，而每次拿数据时，并不是直接从堆头拿数据，而是从队尾往前寻找第一个被阻塞的线程，这样就会按照顺序释放被阻塞的线程。

4 总结

SynchronousQueue 源码比较复杂，建议大家进行源码的 debug 来学习源码，为大家准备了调试类：SynchronousQueueDemo，大家可以下载源码自己调试一下，这样学起来应该会更加轻松一点。

← 19 LinkedBlockingQueue 源码解析

21 DelayQueue 源码解析 →

精选留言 6

欢迎在这里发表留言，作者筛选后可公开显示

所相虚妄

老师能不能说一下，这个数据都应用场景是啥？

0 回复 6天前

慕粉1150563265

是有容量大小的吧，只不过他的同步机制，促使线程，需要同步等待。等待的线程会保存到队列或者堆栈中

0 回复 2019-11-29

文贺 回复 **慕粉1150563265**

没有容量大小，size() 方法返回的永远是 0

回复 2019-11-30 13:07:09

目录

为了angular耻辱上线

怎么感觉总结transfer方法的那一段写错了…

0 回复 2019-11-29

大LOVE辉

老师,好像和源码不太一样,源码比较麻烦...

0 回复 2019-11-27

慕斯卡0137221

总结之前的话得重新梳理下吧，尤其是线程和节点两个之间的关系，感觉有点乱啊

1 回复 2019-11-22

licly

老师，transfer方法中，else if (!isFulfilling(h.mode)) --> else if (casHead(h, s=snode(s, e, h, FULFILLING|mode))) --> for死循环中，if (m == null)这种情况什么时候会发生呀，感觉走到这一步，应该不是空的，有点懵，SynchronousQueue这个类太难理解了

0 回复 2019-10-18