**Java并发学习(二十三)-LinkedBlockingQueue和LinkedBlockingDeque分析**

有两个比较相似的并发阻塞队列，LinkedBlockingQueue和LinkedBlockingDeque，两个都是队列，只不过前者只能一端出一端入，后者则可以两端同时出入，并且都是结构改变线程安全的队列。其实两个队列从实现思想上比较容易理解，有以下特点：

链表结构（动态数组）

通过ReentrantLock实现锁

利用Condition实现队列的阻塞等待，唤醒

以下将分开讲述LinkedBlockingQueue和LinkedBlockingDeque的基本特点及操作。

LinkedBlockingQueue

这是一个只能一端出一端如的单向队列结构，是有FIFO特性的，并且是通过两个ReentrantLock和两个Condition来实现的。先看它的结构基本字段：

/\*\*

\* 基于链表。

\* FIFO

\* 单向

\*最大容量是Integer.MAX\_VALUE.

\*/

public class LinkedBlockingQueueAnalysis<E> extends AbstractQueue<E>

implements BlockingQueue<E>, java.io.Serializable {

/\*

\* 两个方向。

\* putLock

\* takeLock

\* 有些操作会需要同时获取两把锁。

\* 例如remove操作，也需要获取两把锁

\*/

//主要的node节点

static class Node<E> {

E item;

Node<E> next;

Node(E x) { item = x; }

}

//容量，一开始就固定了的。

private final int capacity;

//用AtomicInteger 来记录数量。

private final AtomicInteger count = new AtomicInteger();

//head节点 head.item == null

transient Node<E> head;

//last节点，last.next == null

private transient Node<E> last;

//take锁

private final ReentrantLock takeLock = new ReentrantLock();

//等待take的节点序列。

private final Condition notEmpty = takeLock.newCondition();

//put的lock。

private final ReentrantLock putLock = new ReentrantLock();

//等待puts的队列。

private final Condition notFull = putLock.newCondition();

... }

和LinkedBlockingDeque的区别之一就是，LinkedBlockingQueue采用了两把锁来对队列进行操作，也就是队尾添加的时候，   
队头仍然可以删除等操作。接下来看典型的操作。

### put操作

首先看put操作：

public void put(E e) throws InterruptedException {

if (e == null) throw new NullPointerException(); //e不能为null

int c = -1;

Node<E> node = new Node<E>(e);

final ReentrantLock putLock = this.putLock; //获取put锁

final AtomicInteger count = this.count; //获取count

putLock.lockInterruptibly();

try {

while (count.get() == capacity) { //如果满了，那么就需要使用notFull阻塞

notFull.await();

}

enqueue(node);

c = count.getAndIncrement();

if (c + 1 < capacity) //如果此时又有空间了，那么notFull唤醒

notFull.signal();

} finally {

putLock.unlock(); //释放锁

}

if (c == 0) //当c为0时候，也要根take锁说一下，并发下

signalNotEmpty(); //调用notEmpty

}

主要的思想还是比较容易理解的，现在看看enqueue 方法：

private void enqueue(Node<E> node) { //入对操作。

last = last.next = node; //队尾进

}

再看看signalNotEmpty方法：

private void signalNotEmpty() {

final ReentrantLock takeLock = this.takeLock;

takeLock.lock(); //加锁

try {

notEmpty.signal(); //用于signal，notEmpty

} finally {

takeLock.unlock();

}

}

take操作

take操作，就是从队列里面弹出一个元素，下面看它的详细代码：

public E take() throws InterruptedException {

E x;

int c = -1; //设定一个记录变量

final AtomicInteger count = this.count; //获得count

final ReentrantLock takeLock = this.takeLock;

takeLock.lockInterruptibly(); //加锁

try {

while (count.get() == 0) { //如果没有元素，那么就阻塞性等待

notEmpty.await();

}

x = dequeue(); //一定可以拿到。

c = count.getAndDecrement();

if (c > 1)

notEmpty.signal(); //报告还有元素，唤醒队列

} finally {

takeLock.unlock();

}

if (c == capacity)

signalNotFull(); //解锁

return x;

}

接下来看dequeue方法：

private E dequeue() {

Node<E> h = head;

Node<E> first = h.next;

h.next = h; // help GC 指向自己，帮助gc回收

head = first;

E x = first.item; //从队头出。

first.item = null; //将head.item设为null。

return x;

}

对于LinkedBlockingQueue来说，有两个ReentrantLock分别控制队头和队尾，这样就可以使得添加操作分开来做，一般的操作是获取一把锁就可以，但有些操作例如remove操作，则需要同时获取两把锁：

public boolean remove(Object o) {

if (o == null) return false;

fullyLock(); //获取锁

try {

for (Node<E> trail = head, p = trail.next;

p != null;

trail = p, p = p.next) { //依次循环遍历

if (o.equals(p.item)) { //找到了

unlink(p, trail); //解除链接

return true;

}

}

return false; //没找到，或者解除失败

} finally {

fullyUnlock();

}

}

当然，除了上述的remove方法外，在Iterator的next方法，remove方法以及LBQSpliterator分割迭代器中也是需要加全锁进行操作的。

LinkedBlockingDeque

名字很相近，LinkedBlockingDeque就是一个双端队列，任何一端都可以进行元素的出入，接下来看它的主要字段：

/\*\*

\* 双端队列。

\* 最大值是Integer.MAX\_VALUE

\* 所谓弱一致性有利于删除，有点理解了，

\* 或许是比如clear方法，不知直接把引用置为null，而是一个个解除连接。

\* 利用lock锁去控制并发访问，利用condition去控制阻塞

\* weakly consistent的iterators。

\* 我们需要保持所有的node都要是gc可达的。

\*/

public class LinkedBlockingDeque<E>

extends AbstractQueue<E>

implements BlockingDeque<E>, java.io.Serializable {

//双向联结的节点。

static final class Node<E> {

E item; //泛型的item变量

// 前一个节点

Node<E> prev;

//next后一个节点

Node<E> next;

Node(E x) {

item = x;

}

}

//头节点

transient Node<E> first;

//尾节点。

transient Node<E> last;

//count，表示数值。

private transient int count;

//容量

private final int capacity;

//实现控制访问的锁

final ReentrantLock lock = new ReentrantLock();

//take的Condition

private final Condition notEmpty = lock.newCondition();

//put的Condition

private final Condition notFull = lock.newCondition();

...

}

从上面的结果来看，其实LinkedBlockingDeque的结构上来说，有点像ArrayBlockingQueue的构造，也是一个ReentrantLock和两个Condition，下面分别对其中重要方法进行分析。

public void addFirst(E e)

public void addLast(E e)

public boolean offerFirst(E e)

public boolean offerLast(E e)

…

对于LinkedBlockingDeque，和ArrayBlockingQueue结构还是很类似的，也是一个ReentrantLock和两个Condition使用，但是仅仅是在这二者使用上，其实内部运转还是很大不同的。

offerFirst操作

offerFirst就是在队头添加一个元素：

public boolean offerFirst(E e) {

if (e == null) throw new NullPointerException();

Node<E> node = new Node<E>(e);

final ReentrantLock lock = this.lock; //加锁

lock.lock();

try {

return linkFirst(node);

} finally {

lock.unlock();

}

}

接下来看linkFirst方法：

private boolean linkFirst(Node<E> node) {

if (count >= capacity) //容量满了

return false;

Node<E> f = first; //在队头添加

node.next = f;

first = node;

if (last == null) //第一个节点

last = node;

else

f.prev = node;

++count; //count自增

notEmpty.signal(); //说明不为null。唤醒等待队列

return true;

}

其他的方法类似，都是加锁后对链表的操作，这里就不赘述了。

clear操作

其实我一开始看clear操作时候，总以为它是直接把first和last分别置为null就行了，非常简单，但实际上，它的实现方法却是遍历以便，分别把所有node指针都指向null从而方便gc。

public void clear() {

final ReentrantLock lock = this.lock;

lock.lock(); //加锁后清空所有。

try {

for (Node<E> f = first; f != null; ) { //遍历一遍

f.item = null; //置空操作

Node<E> n = f.next;

f.prev = null;

f.next = null;

f = n; //f后移动一个

}

first = last = null;

count = 0;

notFull.signalAll(); //通知等待put线程

} finally {

lock.unlock();

}

}

这样的思路很值得学习借鉴。

总结

总的来说，这两个阻塞队列实现上还是比较容易理解的，具体细节方面还是很值得阅读的。

---------------------

作者：6点A君

来源：CSDN

原文：https://blog.csdn.net/anla\_/article/details/79027867

版权声明：本文为博主原创文章，转载请附上博文链接！