**java集合框架之ConcurrentLinkedQueue 深度解析（一）**

**摘要：**

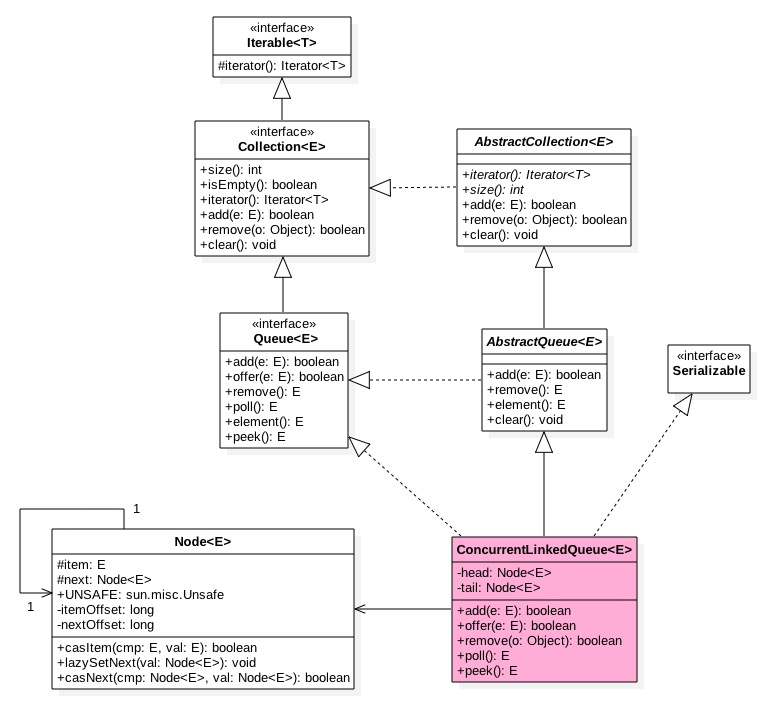
在并发编程中我们有时候需要使用线程安全的队列。如果我们要实现一个线程安全的队列有两种实现方式一种是使用阻塞算法，另一种是使用非阻塞算法。使用阻塞算法的队列可以用一个锁（入队和出队用同一把锁）或两个锁（入队和出队用不同的锁）等方式来实现，而非阻塞的实现方式则可以使用循环CAS的方式来实现，本文让我们一起来研究下Doug Lea是如何使用非阻塞的方式来实现线程安全队列ConcurrentLinkedQueue的，相信从大师身上我们能学到不少并发编程的技巧。

**ConcurrentLinkedQueue的介绍**

ConcurrentLinkedQueue是一个基于链接节点的无界线程安全队列，它采用先进先出的规则对节点进行排序，当我们添加一个元素的时候，它会添加到队列的尾部，当我们获取一个元素时，它会返回队列头部的元素。它采用了“wait－free”算法来实现，该算法在Michael & Scott算法上进行了一些修改, Michael & Scott算法的详细信息可以参见[参考资料一](http://www.cs.rochester.edu/u/michael/PODC96.html)。

**ConcurrentLinkedQueue的结构**

我们通过ConcurrentLinkedQueue的类图来分析一下它的结构。



如图ConcurrentLinkedQueue中有两个volatile类型的head节点和tair节点分别用来存在列表的首尾节点，其中head节点存放链表第一个item为null的节点，tail则并不是总指向最后一个节点。Node节点内部则维护一个变量item用来存放节点的值，next用来存放下一个节点，从而链接为一个单向无界列表。

*/\*\**

*\* Creates a {@code ConcurrentLinkedQueue} that is initially empty.*

*\*/*

public ConcurrentLinkedQueue() {

head = tail = new Node&lt;E>(null);

}

**方法摘要**



**boolean offer(E e)**

*/\*\**

*\* Inserts the specified element at the tail of this queue.*

*\* As the queue is unbounded, this method will never return {@code false}.*

*\**

*\* @return {@code true} (as specified by {@link Queue#offer})*

*\* @throws NullPointerException if the specified element is null*

*\*/*

public boolean offer(E e) {

*//e为null则抛出空指针异常*

checkNotNull(e);

*//构造Node节点构造函数内部调用sun.misc.Unsafe.putObject，后面统一讲*

final Node&lt;E> newNode = new Node&lt;E>(e);

*//从尾节点插入*

for (Node&lt;E> t = tail, p = t;;) {

Node&lt;E> q = p.next;

*//如果q==null说明p是尾节点则插入*

if (q == null) {

*// p is last node*

*//cas插入*

if (p.casNext(null, newNode)) {

*//cas成功说明新增节点已经被放入链表，然后设置当前尾节点*

*// Successful CAS is the linearization point*

*// for e to become an element of this queue,*

*// and for newNode to become "live".*

if (p != t) *// hop two nodes at a time*

casTail(t, newNode); *// Failure is OK.*

return true;

}

*// Lost CAS race to another thread; re-read next*

}

else if (p == q)

*//多线程操作时候，由于poll时候会把老的head变为自引用，然后head的next变为新head，所以这里需要*

*//重新找新的head，因为新的head后面的节点才是激活的节点*

*// We have fallen off list. If tail is unchanged, it*

*// will also be off-list, in which case we need to*

*// jump to head, from which all live nodes are always*

*// reachable. Else the new tail is a better bet.*

p = (t != (t = tail)) ? t : head;

else

*// Check for tail updates after two hops.*

p = (p != t && t != (t = tail)) ? t : q;

}

}

public E poll() {

restartFromHead:

*//死循环*

for (;;) {

*//死循环*

for (Node<E> h = head, p = h, q;;) {

*//保存当前节点值*

E item = p.item;

*//当前节点有值则cas变为null*

if (item != null && p.casItem(item, null)) {

*//cas成功标志当前节点以及从链表中移除*

*// Successful CAS is the linearization point*

*// for item to be removed from this queue.*

if (p != h) *// hop two nodes at a time/(类似tail间隔2设置一次头节点)*

updateHead(h, ((q = p.next) != null) ? q : p);

return item;

}

*//当前队列为空则返回null*

else if ((q = p.next) == null) {

updateHead(h, p);

return null;

}

*//自引用了，则重新找新的队列头节点*

else if (p == q)

continue restartFromHead;

else*//(5)*

p = q;

}

}

}

final void updateHead(Node&lt;E> h, Node&ltE> p) {

if (h != p && casHead(h, p))

h.lazySetNext(h);

}

**总结：**

ConcurrentLinkedQueue使用CAS非阻塞算法实现使用CAS解决了当前节点与next节点之间的安全链接和对当前节点值的赋值。由于使用CAS没有使用锁，所以获取size的时候有可能进行offer，poll或者remove操作，导致获取的元素个数不精确，所以在并发情况下size函数不是很有用。另外第一次peek或者first时候会把head指向第一个真正的队列元素。

下面总结下如何实现线程安全的，可知入队出队函数都是操作volatile变量：head，tail。所以要保证队列线程安全只需要保证对这两个Node操作的可见性和原子性，由于volatile本身保证可见性，所以只需要看下多线程下如果保证对着两个变量操作的原子性。

对于offer操作是在tail后面添加元素，也就是调用tail.casNext方法，而这个方法是使用的CAS操作，只有一个线程会成功，然后失败的线程会循环一下，重新获取tail，然后执行casNext方法。对于poll也是这样的。