链滴

【java 并发系列】非阻塞算法

作者：gitzzzf

原文链接：https://ld246.com/article/1585416392348

来源网站：链滴

引言

我们知道在java中，在不止一个线程访问一个互斥变量的时候，所有的线程必须要使用同步，不然的，可能会发生很多我们意想不到的情况，而我们常用的可能是互斥锁synchronized，他强制实现线程的互斥动作。synchronized保证了程序的安全性，但我们也知道在并发的情况下，它也带来了频繁的下文切换、线程反复挂起调用的资源消耗问题，那么有什么比较好的解决办法吗？既能保证安全性，能提升性能和解决资源消耗等问题。当然有啦，接下来我们就介绍下本文的猪脚，啊呸！是主角 --- 阻塞算法。

一、计数器问题

我们说到并发，很多人可能都会想到计数器问题，也会拿计数这个事情来做测试。那么，显然，毫不外的，本文也是！

我们先看第一段实例如下：

```
public class BlockingCounter {
    /**
     * 计数器
     */
    private int count;

    public synchronized int getCount(){
        return count;
    }

    public synchronized int increment(){
        return ++count;
    }
}
```

我们知道上面这段简单的程序中通过synchronized在并发访问这个计数器的时候，保证了它的安全性但是正如引言中所述，我们知道synchronized在并发的情况下是阻塞式的，也因此性能不够优秀，那有不阻塞的办法吗？当然有，比如下面这个计数器：

```
public class NonBlockingCounter {
    /**
     * 计数器
     */
    private AtomicInteger count;

    public int getValue(){
        return count.get();
    }

    public int increment(){
        int v;
        do {
            v = count.get();
        }while (count.compareAndSet(v, v+1));
        return v+1;
    }
}
```

这其实就是一种简单的非阻塞算法，在并发访问时，使用具有原子特性的AtomicInteger，在更新计器时候使用CAS保证了数据的正确性。在访问这个计数器的过程中，我们发现没有任何阻塞的地方，不需要竞争锁。但是它也是具有一定的的冒险性的。它假设我们先获取到现在的计数值v，并希望我更新计数器（CAS）的时候，这期间计数器的值是没有变化的，否则，我就需要再走一遍这个逻辑。

二、非阻塞的堆栈

在上面我们讲到了非阻塞的计数器，计数器是一种非常简单的场景，那么有复杂一点的数据结构吗？然有，比如非阻塞的堆栈。

```
public class NonBlockingStack<E> {
    /**
     * 头节点
     */
```

```
    private AtomicReference<Node<E>> head = new AtomicReference<>();

    /**
     * 进栈
     */
    public void push(E item) {
        Node<E> newHead = new Node<>(item);
        Node<E> oldHead;
        do {
            oldHead = head.get();
            newHead.next = oldHead;
        } while (!head.compareAndSet(oldHead, newHead));
    }

    /**
     * 出栈
     */
    public E pop() {
        Node<E> oldHead;
        Node<E> newHead;
        do {
            oldHead = head.get();
            if (oldHead == null){
                return null;
            }
            newHead = oldHead.next;
        } while (!head.compareAndSet(oldHead, newHead));
        return oldHead.item;
    }

    /**
     * 节点
     */
    static class Node<E> {
        final E item;
        Node<E> next;

        public Node(E item) {
            this.item = item;
        }
    }
}
```

其实非阻塞的堆栈的操作逻辑跟NonBlockingCounter类似，都是先基于一种假设，假设我们开始获值到最后提交任务的时候，这期间我需要更新的数据是没有发生变化的。如果发生了变化，那么我就来一遍。

三、非阻塞的链表

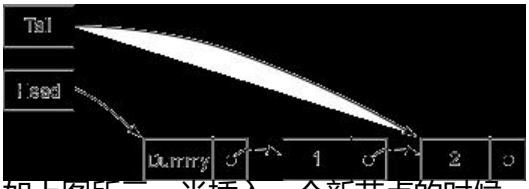
到目前为止我们讲了非阻塞的计数器和栈结构，都是比较简单的非阻塞算法。一旦我们掌握了CAS，可以写出类似的逻辑。但是对于更加复杂的结构，对应的非阻塞算法就要复杂得多。比如修改链表，会同时涉及到多个指针的更新。而CAS支持的是单一指针的原子性修改。所以，如果我们想构建一个阻塞的链表，就需要找到一个可以用CAS的形式同时更新多个指针，且能保证数据的一致性。

首先我们思考下，在链表的尾部插入一个元素的时候，通常会涉及到两个指针的更新。尾指针总是指链表中的最后一个元素，而链表中的最后一个元素也需要更新为新插入的元素。因此，更新两个指针也就是两个CAS的操作。那么在这两个独立的CAS的操作下如何保证一致性呢？如果第一个 CAS 成，而第二个 CAS 失败，会发生什么？如果其他线程在第一个和第二个 CAS 之间企图访问链表，又会生什么？

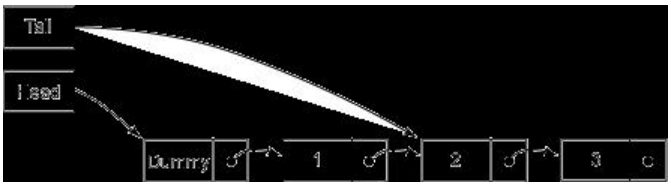
下面就是Michael-Scott提出的非阻塞链表的插入部分。

```
public class LinkedQueue<E> {
    /**
     * 哑节点
     */
    private final Node<E> empty = new Node<>(null, null);
    /**
     * 头节点，初始化指向"哑节点"
     */
    private final AtomicReference<Node<E>> head = new AtomicReference<>(empty);
    /**
     * 尾节点，初始化指向"哑节点"
     */
    private final AtomicReference<Node<E>> tail = new AtomicReference<>(empty);

    /**
     * 插入元素
     */
    public boolean put(E item) {
        // 通过元素构建节点
        Node node = new Node(item, null);
        while (true) {
            // 拿到当前的尾节点和尾节点的next节点，稳定态下尾节点的next节点应该都是空节点
            Node<E> curTail = tail.get();
            Node<E> tailNext = curTail.next.get();

            if (curTail == tail.get()) {
                // tailNext不等于null，说明队列处于中间状态，推进尾节点
                if (tailNext != null) {          /* A */
                    tail.compareAndSet(curTail, tailNext);       /* B */
                } else {
                    // 队列处于稳定状态，正常插入节点
                    if (curTail.next.compareAndSet(null, node)) {     /* C */
                        /*
                         插入成功的情况下，推进尾节点，
                         不管我自己推进尾节点成不成功，都return true，
                         因为我推进没成功，说明别人已经帮我推进了
                         */
                        tail.compareAndSet(curTail, node);        /* D */
                        return true;
                    }
                }
            }
        }
    }

    /**
     * 节点
```

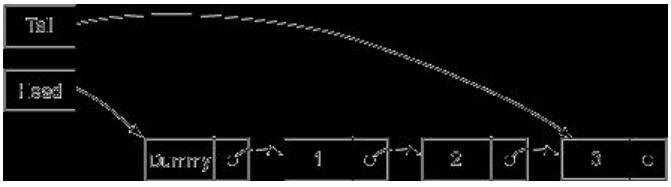
```
     */
    static class Node<E> {
        /**
         * 元素
         */
        private final E item;
        /**
         * next节点
         */
        private final AtomicReference<Node<E>> next;

        public Node(E item, Node<E> nextNode) {
            this.item = item;
            this.next = new AtomicReference<>(nextNode);
        }
    }
}
```



如上图所示，当插入一个新节点的时候，首先更新当前最后一个节点的next指针，将新的节点插入到表的队尾，然后再更新尾节点tail，将其指向这个新节点。当这两个操作都完成时候，则链表处于稳状态，在这两个操作之间时候，链表处于中间状态。我们知道，稳定状态下，tail.next为null，如果于中间状态，也就是tail.next为非空（其实也就是第一个CAS操作完成阶段，这个时候新的节点已经入到队尾，但是tail节点还没有指向这个新元素，还是指向老元素，而老元素的next节点，此时已经是null了，是新插入的节点）。因此，每个线程都可以通过检查tail.next与null的关系来判断当前链表于的状态。如果处于中间态（如下图一所示），那么主动帮助去推进尾节点即可，使链表恢复稳定态如下图二所示）；如果处于稳定态，则执行自己的插入新节点操作即可。



图一 链表中间态



图二 链表稳定态

LinkedQueue的put方法在插入新元素时，首先检查链表是否处于中间态（步骤A）。如果是，则说有另一个线程正在插入元素（步骤C和D之间）。此时线程不会等待其他线程执行完成，而是”帮助它完成操作，也就是推进尾节点（步骤B）。然后，再重新检查，直到它发现队列已经处于稳定状态，这个时候它才会执行自己的插入新元素到队尾的操作，也就是步骤C。步骤C的操作是将新元素插入队尾，在执行期间步骤C期间，如果执行失败了，那么说明有另一个线程同时在插入新元素，那么这个

AS操作失败后，再重新检查重试即可。如果执行成功，说明新元素成功插入到链表的队尾，接下来执步骤D，也就是这个第二个CAS操作，这第二个CAS操作不管执行成功与否都return true，也就是表插入新元素完成。为什么这样呢？因为如果步骤D执行成功，也就是tail节点顺利指向了新元素，链表次恢复到稳定态，结束；如果步骤D没有执行成功，那说明有其他的线程在步骤B中已经”帮忙“推了尾节点tail，链表也已经恢复稳定态了，结束。

以上，就是链表的非阻塞算法，而我们非常熟悉的ConcurrentLinkedQueue 使用的就是该算法。

结语

在多线程访问下，非阻塞算法的实现比基于互斥锁的实现要复杂的多，但是却给我们带来了很大的性提升。这样的算法啊，值得我们好好的品，细品！