当数组遇上队列: Java线程安全实现详解(一)

> 当你养成一种分析问题、琢磨文章的习惯之后，日积月累；你便会感到复杂的东西也是由少数几个大的部分组成的。这些部分出现的原因和它们之间的相互关系也是可以理解的。与此同时，由于读的东西多了，运算的技巧也高了，你会发现，一些复杂的推演过程大部分是由某些必然的步骤所组成，就比较容易抓住新的关键性的部分 --- 越民义

> 这是一个系列的文章，我们会逐步完善。用到什么讲什么。

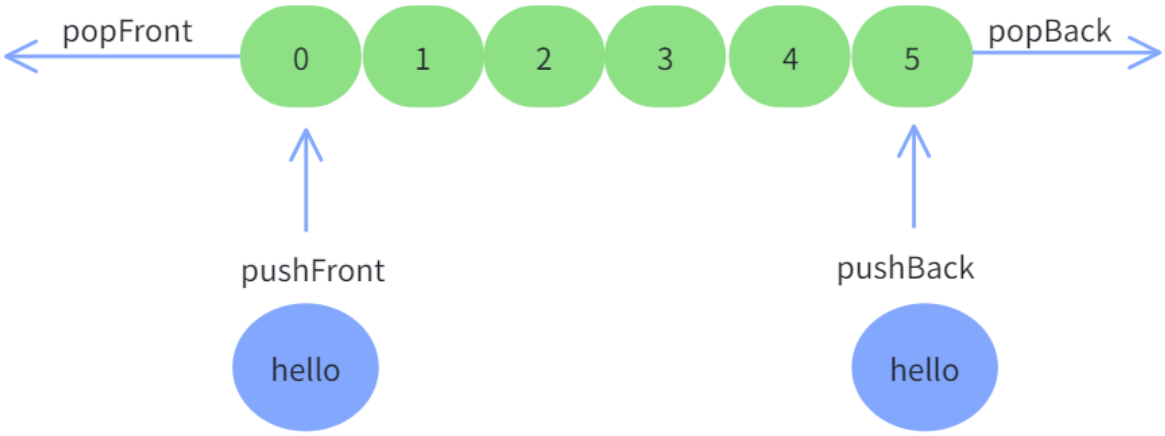
前言

最近碰上个有趣的题目，用数组实现一个队列，要求线程安全，高性能。这个问题刚好能糅合一些对基础知识的运用，于是打算做一下，同时展开自己的思考过程。要求是递进式的，首先是用数组实现一个队列，队列的特点是先进先出，先进先出意味着我们要提供一个方法入队和出队方法:

- pushBack: 将元素放到队尾，如果队列满了，返回添加失败。
- popFront: 从队头开始出队，如果队列为空返回失败。

也可以是从队头入队，然后从队尾出队:

- pushFront: 队头入队 如果队列满了，返回添加失败
- popBack: 队尾出队



有时候我们也希望知道这个队列实际存储了几个元素:

- size方法: 返回了队列实际存储了多少个元素

首先用数组实现队列

有了设计目标就可以着手编写代码了，在Java里面首先我们需要一个类，类里面存放数据和行为, 为了让我们的队列更通用一些，我们将为我们的队列引入泛型支持，但是在Java中无法实现泛型数组的语法，也就是下面这样:

```
T[] array = new T[5];
```

在C#中我们就可以这么声明:

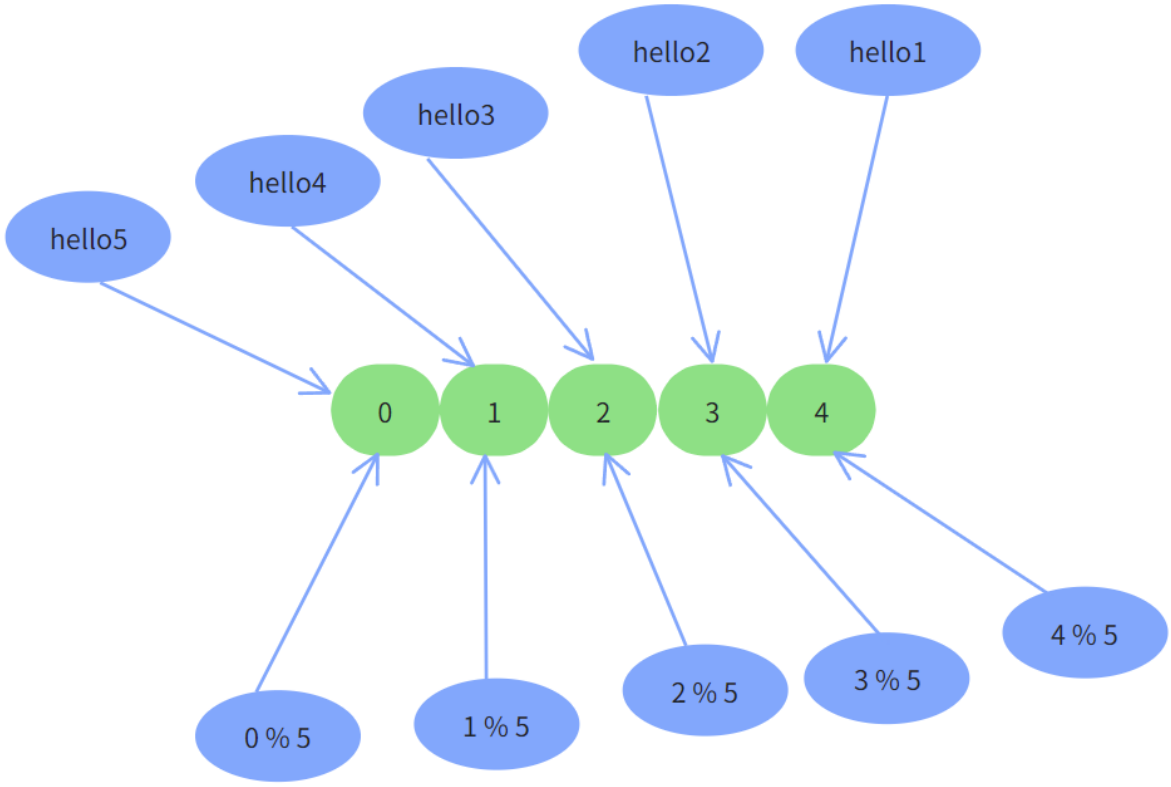
```
using System;
class Program
{
    public static void Main(string[] args)
    {
          // 声明并初始化一个整型数组
        int[] intArray = new int[5];
        intArray[0] = 1;
        Console.WriteLine(intArray[0]);

        // 声明并初始化一个字符串数组
        string[] stringArray = new string[5];
        stringArray[0] = "Hello";
        Console.WriteLine(stringArray[0]);

        // 使用泛型方法创建和操作泛型数组
        CreateAndOperateArray<int>(10, 5);
        CreateAndOperateArray<string>("World", 5);
    }
    static void CreateAndOperateArray<T>(T initialValue, int size)
    {
        T[] array = new T[size];
        array[0] = initialValue;
        Console.WriteLine(array[0]);
    }
}
```

回顾ArrayList的实现你会发现，ArrayList实现泛型化，底层仍然是一个Object数组，然后在get的时候，做了强制类型转换

```
public E get(int index) {
    rangeCheck(index);
    return elementData(index);
}
```

```
@SuppressWarnings("unchecked")
E elementData(int index) {
    return (E) elementData[index];
}
```

所以我们也是用这样的思路来解决类型转换问题。解决完泛型问题，我们来设计pushBack方法，这个方法的设计目标是将请求入队的元素放在队尾，如果我们基于引用实现，在类里面我们可以放一个尾节点(指针)指向最后一个节点。用数组实现的时候我们存储的就是尾节点的下标了，在一开始队列初始化的时候尾结点和头节点可以都指向0下标，于是我们就需要有一个方法计算出来我们的元素应该放在哪个位置，我们自然就想到了取余，pushBack方法将元素放在队尾，设定我们有hello5、hello4、hello3、hello2、hello1 这五个元素依次通过进入队列，hello5对应0位置，hello4对应1位置，hello3对应2位置，hello2对应3位置，hello1对应4位置:



在这个入队过程中，队尾在朝前移动的： 0 => 1 => 2 => 3 => 4 所以我们可以声明一个整型的变量记录队尾的位置:

```
// capacity 是数组容量
int tail = 0;
```

计算下一个位置的公式就可以为: next = (tail++) % capacity，现在让我们考虑入队是指针迁移，出队就是指针后移，所以popBack方法首先获取当前头节点的位置，然后指针后移即可。现在我们可以写出下面的代码:

```
public class FixMemoryQueue<T> {
```

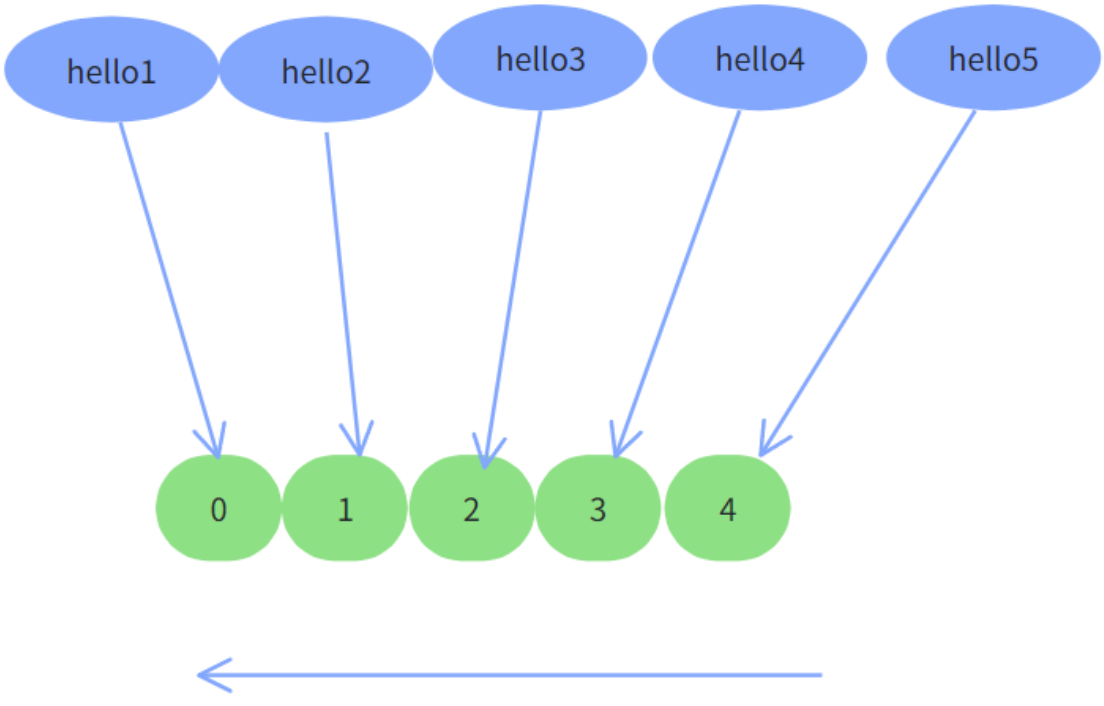
```
    /**
     * 存储数据的泛型数组
     */
    private Object[] dataArray;

    /**
     * 记录头结点的位置
     */
    private int head;

    /**
     * 记录尾结点的位置
     */
    private int tail;
    /**
     * 记录队列存储了多少元素
     */
    private int size;

    private int capacity;

    public FixMemoryQueue(int capacity) {
        if (capacity <= 0) {
            throw new IllegalArgumentException();
        }
        this.capacity = capacity;
        this.head = 0;
        this.tail = 0;
        this.size = 0;
        dataArray = new Object[capacity];
    }


    public boolean pushBack(T t) {
        if (size == capacity) {
            return false;
        }
        dataArray[tail++ % capacity] = t;
        size++;
        return true;
    }

     public T popFront() {
        if (size == 0) {
            return null;
        }
        size--;
        T result = (T) dataArray[head];
        dataArray[head] = null;
        head++;
        return result;
    }

    public int size() {
        return size;
    }
}
```

这里我还想实现方法返回值给出出队成功还是失败，那怎么接出队的元素呢？其实再有一个类就好了，我们可以将方法变形为：

```
public boolean popFront(DataHolder<T> dataHolder) {
    if (Objects.isNull(dataHolder)) {
        dataHolder = new DataHolder<>();
    }
    if (size == 0) {
        return false;
    }
    size--;
    T result = (T) dataArray[head];
    dataHolder.setT(result);
    dataArray[head] = null;
    head++;
    return true;
}
```

```
public static class DataHolder<T> {
    T t;

    public void setT(T t) {
        this.t = t;
    }

    public T getT() {
        return t;
    }
}
```

现在让我们从队头入队pushFront，队尾出popBack，也就是说入队顺序是hello5、hello4、hello3、hello2、hello1，出队顺序是hello1、hello2、hello3、hello4、hello5。也就是先进后出。



入队的过程中头指针前移，这次是尾指针固定不动。 因此我们可以给出下面这两个方法：

```
public  boolean pushFront(T t){
    if (size == capacity) {
        return false;
    }
    int next = (size ) % capacity;
    this.head = next;
    dataArray[next] = t;
    size++;
    return true;
}
```

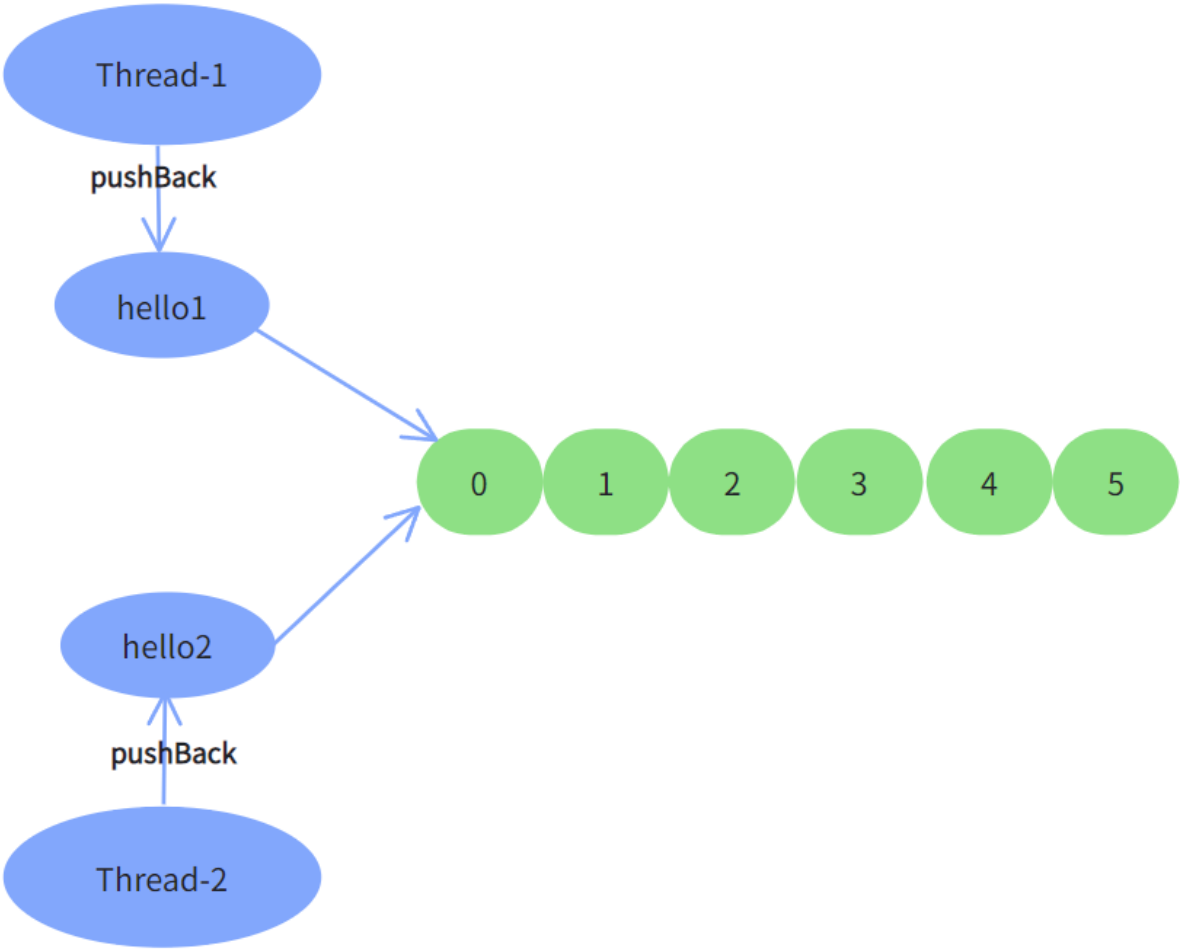
```
public T popBack(){
    if (size == 0) {
        return null;
    }
    T result = (T)dataArray[head];
    size--;
    head--;
    return  result;
}
```

注意到现在为止我们都保持了良好的操作对称性，即先进(pushBack)先出(popFront)，先进(pushFront)后出(popBack)。现在让我们为这个队列加上一个限制词，我们希望他线程安全。

然后是线程安全

在《我们来聊聊线程安全吧》这一篇我们已经对线程安全进行了详细论述，虽然我们没有为线程安全找到一个形式化的定义，但是我们做到了工程化的回答，也就是说对于线程安全的集合来说， 我们要求以下三点:

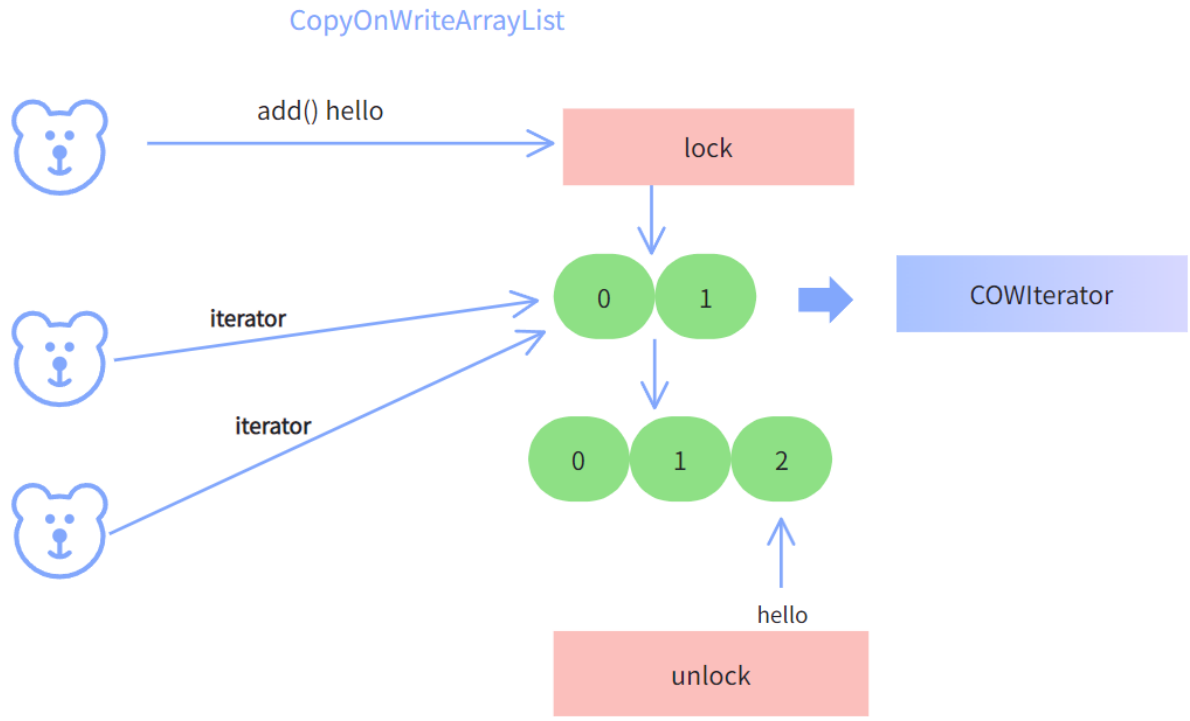
1. 不能丢值：比如我们上面的队列，两个线程都向集合放值，集合可以容纳两个元素的时候，不应该出现丢值，在上面我们实现的队列中假设两个线程都添加一个，后操作的可能把先操作的覆盖掉。



2. 出现死循环，导致调用方无法接着往下执行。
3. 多个线程操作的时候应当像线程单独排队操作的效果，也就是隔离性，这也被称为一致性，在《我们来聊聊线程安全吧》这一篇里面我们已经讨论过这个弱一致性了，当时我对这个弱一致性的理解为，多个线程并发读写，多个读线程在什么时候看到的是一致的，由于CopyOnWriteArrayList的特性，每次操作的时候都会产生新的数组，所以如果多个读线程在写之前获得了迭代器，那么这些迭代器遍历出来的元素就是一样的。

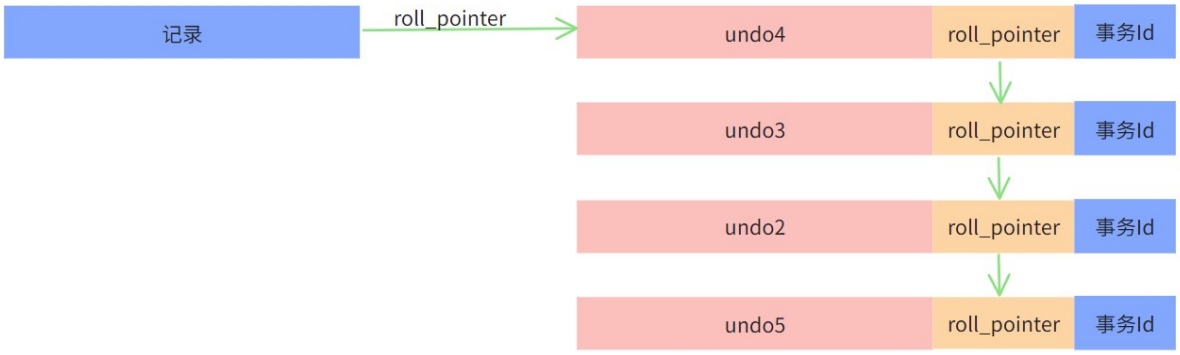
   对于线程安全的集合我们不能靠集合的大小去遍历的原因在于，并发读写的时候，集合实际的大小是在变化的。但是线程安全的集合也需要迭代，因此线程安全的集合在遍历的时候，可以通过迭代器来遍历。下面是CopyOnWriteArrayList的源码:

```
// CopyOnWriteArrayList的源码
public Iterator<E> iterator() {
    return new COWIterator<E>(getArray(), 0);
}
// 省略部分代码
static final class COWIterator<E> implements ListIterator<E> {
 /** Snapshot of the array */
 private final Object[] snapshot;
 /** Index of element to be returned by subsequent call to next.  */
 private int cursor;

 private COWIterator(Object[] elements, int initialCursor) {
      cursor = initialCursor;
      snapshot = elements;
  }
}
```



看CopyonWriteArrayList的设计思想的时候，我想到了MySQL的MVCC(Multi-Version Concurrency Control) 多版本并发控制，感觉隐隐约约有些类似。MySQL根据事务的隔离级别来决定是否给予查询者查看最新的数据，在对数据进行更改的时候，MySQL并不会直接操纵数据，而是会先将改变记录到undo log里面，同时在MySQL的行记录里面有多个隐藏列，涉及事务的有两个值得我们注意:

- trx_id： 每次一个事务对某条聚簇索引记录进行改动时，都会把该事务得到事务id赋值给trx_id隐藏列。
- roll_pointer: 每次对某条聚簇索引记录进行改动时，都会把旧的版本写入到undo 日志中，然后这个隐藏列就相当于一个指针，可以通过它来记录修改前的信息。



每次对记录进行改动，都会记录一条undo日志，每条undo日志也都会有roll_pointer属性，这些日志可以串起来一条链表。版本链的头结点记录的是当前记录最新的值。事务在执行过程中，只有在第一次真正修改记录时(INSERT DELETE UPDATE)，才会被分配一个单独的事务id, 这个事务id是递增的。对于隔离级别是READ UNCOMMITED来说，由于可以读取未提交事务修改过的数据，直接读取最新节点即可。对于READ COMMITED 和 REPEATABLE READ隔离级别的事务来说，都必须保证读到已经提交过的事务，那意味着不能读取最新的Undo Log。那现在的问题就演变成了该读取链表中的哪条记录，这也就引出了READ VIEW这个概念。

回想了一下，感觉又不太想，我以前对CopyOnWriteArrayList这个一致性的理解是，多个线程在产生CopyOnWriteArrayList的迭代器的时候，在产生完成之前没有其他线程对CopyOnWriteArrayList做修改添加操作，这样遍历出来的元素就是一致的。于是我认为ConcurrentHashMap应该也是这样，也生成一个快照，现在让我们来看ConcurrentHashMap 生成迭代器的源码:

```
//  ConcurrentHashMap 源码
public Set<Map.Entry<K,V>> entrySet() {
    EntrySetView<K,V> es;
    return (es = entrySet) != null ? es : (entrySet = new EntrySetView<K,V>
(this));
}
static final class EntrySetView<K,V> extends CollectionView<K,V,Map.Entry<K,V>>
 implements Set<Map.Entry<K,V>>, java.io.Serializable {
  EntrySetView(ConcurrentHashMap<K,V> map) { super(map); }
}
abstract static class CollectionView<K,V,E>  implements Collection<E>,
java.io.Serializable {
        final ConcurrentHashMap<K,V> map;
    CollectionView(ConcurrentHashMap<K,V> map)  { this.map = map; }
}
```

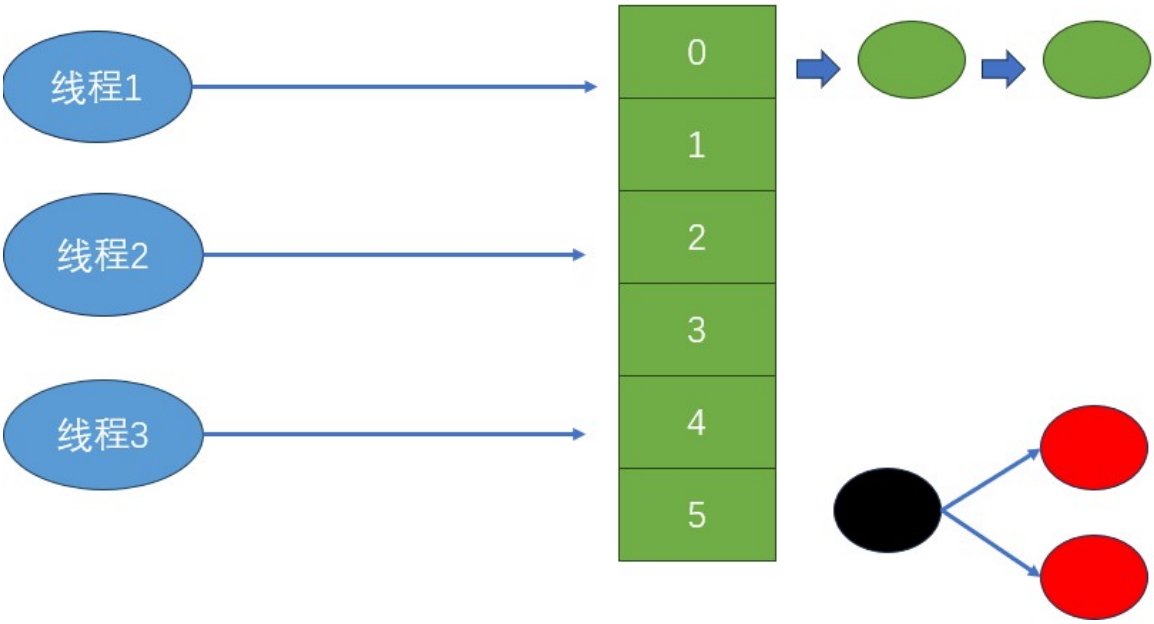
这里看源码可以发现，我们在调用entrySet的时候，生成了一个EntrySetView对象，这个对象持有当前ConcurrentHashMap对象的引用。我们接着看迭代器方法:

```
public Iterator<Map.Entry<K,V>> iterator() {
    ConcurrentHashMap<K,V> m = map;
    Node<K,V>[] t;
    int f = (t = m.table) == null ? 0 : t.length;
    return new EntryIterator<K,V>(t, f, 0, f, m);
}
```

这里获取了ConcurrentHashMap的数组节点，然后构造了一个EntryIterator对象:

```
// 省略无关代码
static final class EntryIterator<K,V> extends BaseIterator<K,V>
        implements Iterator<Map.Entry<K,V>> {
        EntryIterator(Node<K,V>[] tab, int index, int size, int limit,
                      ConcurrentHashMap<K,V> map) {
            super(tab, index, size, limit, map);
        }
}
```

最终产生的迭代器对象持有创建entrySet的ConcurrentHashMap对象的引用和ConcurrentHashMap内部数组的引用。则A、B、C三个读线程去遍历的时候，产生的迭代器对象不一样，保证了线程安全，如果在产生迭代器对象的过程中不扩容，则A、B、C三个读线程持有相同的数组引用:



这在某种程度上也维持了一致性，只不过不像CopyOnWriteArrayList那么强，因为CopyOnWriteArrayList每次操作都是在新的上面进行操作。但是ConcurrentHashMap在扩容之前都是在原来的基础做操作，如果ConcurrentHashMap不扩容，只是在原来的基础上做修改，可以反应变化?带着这个疑问我写出来了下面的例子:

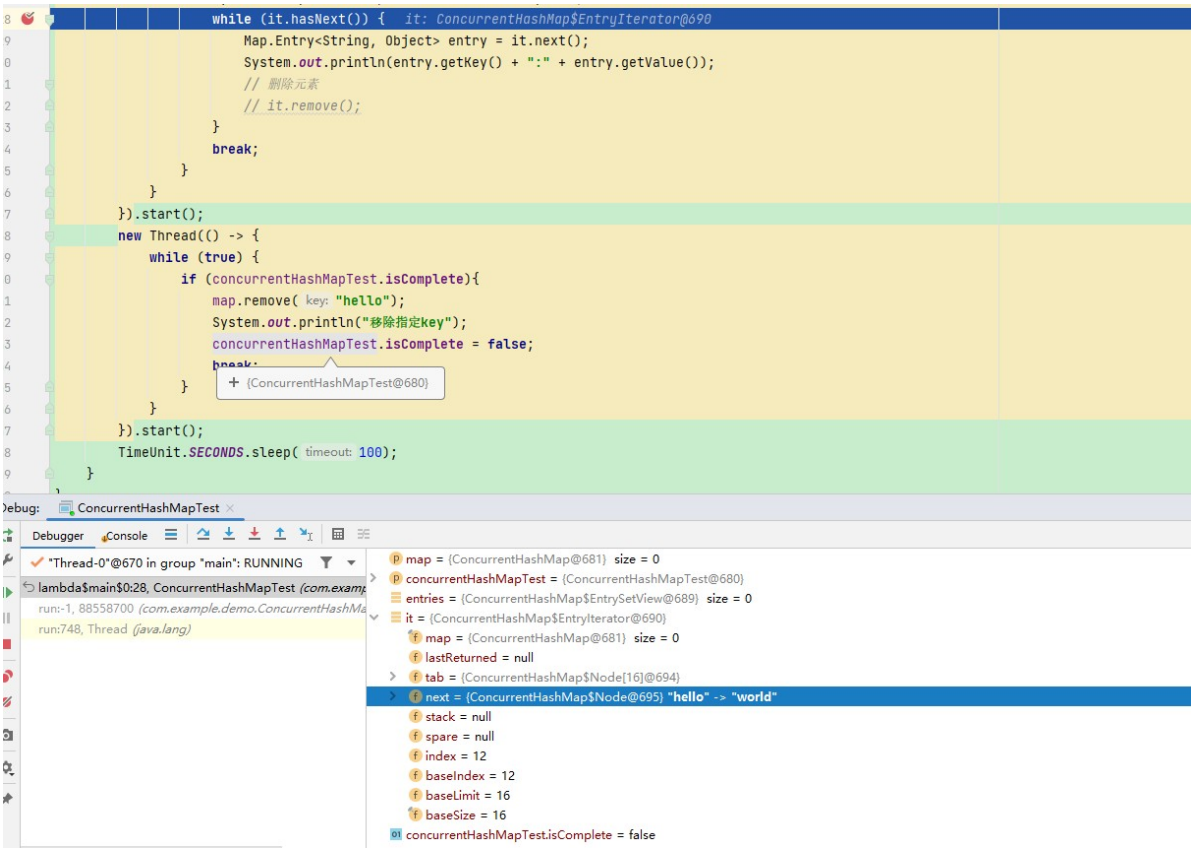
```
public class ConcurrentHashMapTest {
    private volatile boolean isComplete = false;

    public static void main(String[] args) throws InterruptedException {
        ConcurrentHashMap<String, Object> map = new ConcurrentHashMap<>();
        map.put("hello", "world");
        ConcurrentHashMapTest concurrentHashMapTest = new
ConcurrentHashMapTest();
        new Thread(() -> {
            Set<Map.Entry<String, Object>> entries = map.entrySet();
            Iterator<Map.Entry<String, Object>> it = entries.iterator();
            concurrentHashMapTest.isComplete = true;
            while (true) {
                if (!concurrentHashMapTest.isComplete){
                    System.out.println(map.contains("hello"));
                    while (it.hasNext()) {
                        Map.Entry<String, Object> entry = it.next();
                        System.out.println(entry.getKey() + ":" +
entry.getValue());
                    }
                    break;
                }
            }
        }).start();
        new Thread(() -> {
            while (true) {
                if (concurrentHashMapTest.isComplete){
                    map.remove("hello");
                    System.out.println("移除指定key");
                    concurrentHashMapTest.isComplete = false;
                    break;
                }
            }
        }).start();
```

```
            TimeUnit.SECONDS.sleep(100);
        }
    }
```

输出结果为:

```
移除指定key
false
hello:world
```

输出结果跟我的预期不相同，这是为什么呢，明明已经不包含这个key了，带着这个问题，我开始了debug:



迭代器中的map确实为空，但是有个next成员变量还有值，是不是我们忽略了哪里，我们再回顾一下源码:

```
static class BaseIterator<K,V> extends Traverser<K,V> {
    final ConcurrentHashMap<K,V> map;
    Node<K,V> lastReturned;
    BaseIterator(Node<K,V>[] tab, int size, int index, int limit,
                 ConcurrentHashMap<K,V> map) {
        super(tab, size, index, limit);
        this.map = map;
        advance();
    }
}
```

答案就在advance这个方法上:

```
final Node<K,V> advance() {
    Node<K,V> e;
    // advance来自Traverser方法
```

```
        // next 是Traverser的成员变量
        if ((e = next) != null)
            e = e.next;
        for (;;) {
            Node<K,V>[] t; int i, n;  // must use locals in checks
            if (e != null)
                // 因此next这个节点就挂上了值
                return next = e;
            if (baseIndex >= baseLimit || (t = tab) == null ||
                (n = t.length) <= (i = index) || i < 0)
                return next = null;
            // tabAt 确定数组的指定位置是否有值，且判断是否是链表或者树
            // 如果是树就去获取链表或者树的第一个节点
            if ((e = tabAt(t, i)) != null && e.hash < 0) {
                if (e instanceof ForwardingNode) {
                    tab = ((ForwardingNode<K,V>)e).nextTable;
                    e = null;
                    pushState(t, i, n);
                    continue;
                }
                else if (e instanceof TreeBin)
                    e = ((TreeBin<K,V>)e).first;
                else
                    e = null;
            }
            if (stack != null)
                recoverState(n);
            else if ((index = i + baseSize) >= n)
                index = ++baseIndex; // visit upper slots if present
        }
    }
```

所以结果也是符合我们的预期的，只不过是我们对迭代器的理解出了问题，迭代器像是链表，我们总需要一个头结点来不断的往下走，所以在构造迭代器的时候，会构造出第一个头节点。

回顾弱一致性

我们现在再来回忆一下，Java文档中对弱一致性(见参考文档[3])的定义, 这个定义来自JUC包下面的package-summary.html， 关于弱一致性是这么论述的:

Most concurrent Collection implementations (including most Queues) also differ from the usual java.util conventions in that their Iterators and Spliterators provide weakly consistent rather than fast-fail traversal:

> 大部分并发集合实现(包括大多数队列)也与通常的java.util的约定不同的，他们的迭代器(Iterators)和 分裂器(Spliterators) 提供的是弱一致的遍历，而不是快速失败遍历:

1. they may proceed concurrently with other operations: 可与其他操作并行
2. they will never throw ConcurrentModificationException：不会抛出ConcurrentModificationException
3. they are guaranteed to traverse elements as they existed upon construction exactly once, and may (but are not guaranteed to) reflect any modifications subsequent to construction：迭代器可以确保遍历一次在迭代器产生的时候就存在的元素，但是对于之后的修改，迭代器可能接收到，也可能接收不到。

我们对弱一致性的第一点要求取否定，就能获得强一致性集合的定义：也就是不能和其他操作并行。强一致集合的我们很好构造，我们只需要在入队和出队方法上加锁就可以了。

悲观锁实现

```
public class PessimisticFixMemoryQueue<T> {

    /**
     * 存储数据的泛型数组
     */
    private Object[] dataArray;

    /**
     * 记录头结点的位置
     */
    private int head;

    /**
     * 记录尾结点的位置
     */
    private int tail;
    /**
     * 记录队列存储了多少元素
     */
    private int size;

    private int capacity;

    final ReentrantLock lock = new ReentrantLock();

    public PessimisticFixMemoryQueue(int capacity) {
        if (capacity <= 0) {
            throw new IllegalArgumentException();
        }
        this.capacity = capacity;
        this.head = 0;
        this.tail = 0;
        this.size = 0;
        dataArray = new Object[capacity];
    }

    public boolean pushBack(T t) {
        final ReentrantLock lock = this.lock;
        try {
            if (size == capacity) {
                return false;
            }
            dataArray[tail++ % capacity] = t;
            size++;
            return true;
        } finally {
            lock.unlock();
        }

    }

    public T popFront() {
        final ReentrantLock lock = this.lock;
        try {
            if (size == 0) {
```

```
                return null;
            }
            size--;
            T result = (T) dataArray[head];
            dataArray[head] = null;
            head++;
            return result;
        } finally {
            lock.unlock();
        }
    }

    public boolean pushFront(T t) {
        final ReentrantLock lock = this.lock;
        try {
            if (size == capacity) {
                return false;
            }
            int next = (size) % capacity;
            this.head = next;
            dataArray[next] = t;
            size++;
            return true;
        } finally {
            lock.unlock();
        }

    }

    public T popBack() {
        final ReentrantLock lock = this.lock;
        try {
            if (size == 0) {
                return null;
            }
            size--;
            T result = (T) dataArray[head];
            dataArray[head] = null;
            head--;
            return result;
        } finally {
            lock.unlock();
        }
    }

    public int size() {
        final ReentrantLock lock = this.lock;
        try {
            return size;
        } finally {
            lock.unlock();
        }
    }
}
```

这种的确实现了我们对线程安全集合的要求，一致性也是最强的，但是性能也比较弱，原因在于多线程在操作队列的时候都是排队执行，思想和Vector集合一样。

读写锁实现

但有时候我们可能不那么需要强一致，我们希望能够放松对一致性的要求来换性能提升，现在让我们再放松一下对一致性的要求，读写互斥，但是读读可以并行，同时我们也选择移除size上面的锁，改为在size字段上加上volatile来确保能获得相对新的值。但注意在这个队列里面我们的读其实也是写，伴随着下标移动，所以入队方法和出队方法上都需要加写锁，然后在获取容量的方法上加读锁:

```
public class RWFixMemoryQueue<T> {
    /**
     * 存储数据的泛型数组
     */
    private Object[] dataArray;

    /**
     * 记录头结点的位置
     */
    private int head;

    /**
     * 记录尾结点的位置
     */
    private int tail;
    /**
     * 记录队列存储了多少元素
     */
    private int size;

    private int capacity;

    final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();


    public RWFixMemoryQueue(int capacity) {
        if (capacity <= 0) {
            throw new IllegalArgumentException();
        }
        this.capacity = capacity;
        this.head = 0;
        this.tail = 0;
        this.size = 0;
        dataArray = new Object[capacity];
    }


    public boolean pushBack(T t) {
        final ReentrantReadWriteLock.WriteLock writeLock =
this.lock.writeLock();
        try {
            if (size == capacity) {
                return false;
            }
            dataArray[tail++ % capacity] = t;
            size++;
            return true;
        } finally {
            writeLock.unlock();
        }
```

```
    }

    public T popFront() {
        final ReentrantReadWriteLock.ReadLock writeLock = this.lock.readLock();
        ;
        try {
            if (size == 0) {
                return null;
            }
            if (size == 0) {
                return null;
            }
            size--;
            T result = (T) dataArray[head];
            dataArray[head] = null;
            head++;
            return result;
        } finally {
            writeLock.unlock();
        }
    }

    public boolean pushFront(T t) {
        final ReentrantReadWriteLock.WriteLock writeLock =
this.lock.writeLock();
        ;
        try {
            if (size == capacity) {
                return false;
            }
            int next = (size) % capacity;
            this.head = next;
            dataArray[next] = t;
            size++;
            return true;
        } finally {
            writeLock.unlock();
        }

    }

    public T popBack() {
        final ReentrantReadWriteLock.WriteLock writeLock =
this.lock.writeLock();
        try {
            if (size == 0) {
                return null;
            }
            size--;
            T result = (T) dataArray[head];
            dataArray[head] = null;
            head--;
            return result;
        } finally {
            writeLock.unlock();
        }
    }
```

```
    public int size() {
        final ReentrantReadWriteLock.ReadLock readLock = this.lock.readLock();
        try {
            return size;
        } finally {
            readLock.unlock();
        }
    }
}
```

其实我们还可以再放松一点对一致性的要求，在size上加上volatile，移除读锁来，这样size可能不那么准确，但是不用加锁相对性能更强。

乐观锁实现

入队方法改造

那有没有无锁的方案，这也就是我们的原子类了，首先改造的是pushBack方法:

```
public boolean pushBack(T t) {
    if (size == capacity) {
        return false;
    }
    dataArray[tail++ % capacity] = t;
    size++;
    return true;
}
```

对于入队方法我们要解决三个问题需要我们解决:

1. 第一队列满了不能再入队，避免数组下标越界
2. 不能丢值，假设队列容量为5，那么两个线程入队，那么队列的实际大小应当是2，不能是1.
3. 队列的实际大小和放入的元素数量相等，这里强调的是不能越界，跟第一条是重复的。

基于上面三点要求，我们写出的原子类pushBack的版本如下所示:

```
public class AtomicFixMemoryQueue<T> {
    /**
     * 存储数据的泛型数组
     */
    private AtomicReferenceArray<T> dataArray;

    /**
     * 记录头结点的位置
     */
    private AtomicInteger head;

    /**
     * 记录尾结点的位置
     */
    private AtomicInteger tail;
    /**
     * 记录队列存储了多少元素
     */
    private  AtomicInteger size;
```

```
    private int capacity;

    public AtomicFixMemoryQueue(int capacity) {
        if (capacity <= 0) {
            throw new IllegalArgumentException();
        }
        this.capacity = capacity;
        this.head = new AtomicInteger(0);
        this.tail = new AtomicInteger(0);
        this.size = new AtomicInteger(0);
        dataArray = new AtomicReferenceArray<>(capacity);
    }


    public boolean pushBack(T t) {
        if (Objects.isNull(t)){
            throw new NullPointerException();
        }
        while(size.get() < capacity){
           int currentTail = tail.get();
           int nextTail = tail.get() + 1;
           // 交换成功
           if (tail.compareAndSet(currentTail,nextTail)){
               if (dataArray.compareAndSet(nextTail,null,t)){ // 语句一
                    size.incrementAndGet();
               }
           }
           return true; // 语句二
        }
        return false;
    }
}
```

然后我们随手给一个用例:

```
AtomicFixMemoryQueue<String>  atomicFixMemoryQueue = new AtomicFixMemoryQueue<>
(5);
for (int i = 0; i < 5 ; i++) {
    // RandomUtil 来自hutool,也可以随手扔点数字扔进我们构造的队列里面
    new Thread(()->{
        atomicFixMemoryQueue.pushBack("aaa"+ RandomUtil.randomNumbers(5));
    }).start();
}
TimeUnit.SECONDS.sleep(2);
System.out.println(atomicFixMemoryQueue.dataArray);
```

然后输出结果为：[null, aaa35845, aaa06412, aaa49904, null]。很明显我们的pushBack没有满足我们的要求，我们入队了五个元素，但是实际上只放入了三个，除此之外，我们的队列也不允许放入空值，那究竟错在哪里呢，错在语句一和语句二上，在入队的时候我们没有执行取余操作，然后再成功入队之后没有直接返回。明确了这两点之后，我们改造之后的样例如下图所示:

```
public boolean pushBack(T t) {
    if (Objects.isNull(t)){
        throw new NullPointerException();
    }
    while(size.get() < capacity){
```

```
        int currentTail = tail.get();
        int nextTail = tail.get() + 1;
        // 交换成功
        if (tail.compareAndSet(currentTail,nextTail)){
            if (dataArray.compareAndSet( currentTail % capacity,null,t)){
                 size.incrementAndGet();
            }
            return true;
        }
    }
    return false;
}
```

改造成这样之后，感觉好像没有多大问题了，但我还是担心在自旋失败比较多的情况下，导致CPU使用率飙升，于是在这里引入退避策略，也就是说让失败执行超过一定次数的线程先休息一会，那应该怎么休息呢，大致有三种方式:

1. Thread.sleep(); 时间不好估计
2. 放个空循环
3. 也是最推荐的JDK 9 新加入的方法，Thread.onSpinWait, 这个方法请求将更多CPU资源分配给其他线程，而无需实际调用操作系统调度器和将就绪线程出队。
4. Thread.yield 让出CPU

于是我们就在入队方法里面定义一个让步策略:

```
private static final int MAX_ATTEMPTS = 100;
public boolean pushBack(T t) {
    if (Objects.isNull(t)) {
        throw new NullPointerException();
    }
    while (size.get() < capacity) {
        int attempts = 0;
        int currentTail = tail.get();
        int nextTail = tail.get() + 1;
        // 交换成功
        if (tail.compareAndSet(currentTail, nextTail)) {
            if (dataArray.compareAndSet(currentTail % capacity, null, t)) {
                size.incrementAndGet();
            }
            return true;
        } else {
            backOff(++attempts);
        }
    }
    return false;
}
```

```
private void backOff(int attempts) {
    if (attempts == MAX_ATTEMPTS) {
        for (int i = 0; i < MAX_ATTEMPTS; i++) {
            Thread.onSpinWait();
        }
    }
}
```

出队方法改造

有了入队方法的改造，我们同样对出队方法做如下改造:

```
public T popFront() {
    while (size.get() > 0) {
        int headIndex = head.get();
        int tailIndex = tail.get();
        int attempts = 0;
        if (head.compareAndSet(headIndex, headIndex + 1)) {
            size.decrementAndGet();
            T result = dataArray.get(headIndex);
            dataArray.compareAndSet(headIndex,result,null);
            return result;
        } else {
            backOff(++attempts);
        }
    }
    return null;
}
```

给出测试样例

我们上面给了几个测试样例，但我还是有些疑心，虽然在我的电脑上跑出来了，但是放别人的电脑上可能跑不出来，我们可以预测并发的结果集，但是我们不能准确预测并发执行会跑到结果集的哪一个结果上。于是我们就想到JCStress，如果不熟悉JCStress，参看《JMM测试利器-JCStress学习笔记》这篇文章，见参考链接[4]。如果觉得命令行不太方便，IDEA里面可以选择安装一个JCStress插件, 这样跑JCStress速度更快。那测试用例应该怎么写，首先我们应当保证并发写值的时候应当不出现数组下标越界异常:

```
@JCStressTest

// These are the test outcomes.
@Outcome(id = "1, 1", expect = Expect.ACCEPTABLE, desc = "Both actors threw
exception")
@Outcome(id = "1, 0", expect = Expect.ACCEPTABLE, desc = "Only actor1 threw
exception")
@Outcome(id = "0, 1", expect = Expect.ACCEPTABLE, desc = "Only actor2 threw
exception")
@Outcome(id = "0, 0", expect = Expect.ACCEPTABLE, desc = "No exceptions")
// This is a state object
@State
public class API_01_Simple {

    AtomicFixMemoryQueue<String> atomicFixMemoryQueue = new
AtomicFixMemoryQueue<>(1);

    @Actor
    public void actor1(II_Result r) {
        try{
            for (int i = 0; i < 1000 ; i++) {
                atomicFixMemoryQueue.pushBack("11");
            }
        }catch (Exception  e){
            r.r1 = 1;
        }
```

```
    }

    @Actor
    public void actor2(II_Result r) {
        try{
            for (int i = 0; i < 100; i++) {
                atomicFixMemoryQueue.pushBack("11");
            }
        }catch (Exception  e){
            r.r2 = 1;
        }
    }
}
```

然后这个测试只跑出来了结果为0， 0的， 说明没有下标越界，照我的想法应该有下标越界的。我们再把pushBack这个方法拉出来看一下:

```
public boolean pushBack(T t) {
    if (Objects.isNull(t)) {
        throw new NullPointerException();
    }
    while (size.get() < capacity) {
        int attempts = 0;
        int currentTail = tail.get(); // 语句二
        int nextTail = currentTail + 1;
        // 交换成功
        if (tail.compareAndSet(currentTail, nextTail)) {
            if (dataArray.compareAndSet(currentTail % capacity, null, t)) {
                size.incrementAndGet(); // 语句一
            }
            return true;
        } else {
            backOff(++attempts);
        }
    }
    return false;
}
```

照我的想法，线程A还没调用语句一完成自增，这个时候线程B进来的，调用语句二拿到尾指针，如果数组大小为1，则tail自增到2，然后dataArray的compareAndSet方法访问数组下标2，应当出现越界，但是跑了几轮测试发现没有，难道里面做了边界检查? 但是在源码里面又没看到，于是我想给我们的队列增加一个访问尾指针的方法，再跑一下测试看看是否会跑出来尾指针大于数组大小的情况:

```
@JCStressTest

// These are the test outcomes.
@Outcome(id = "1, 1", expect = Expect.ACCEPTABLE, desc = "Both actors threw
exception")
@Outcome(id = "1, 0", expect = Expect.ACCEPTABLE, desc = "Only actor1 threw
exception")
@Outcome(id = "0, 1", expect = Expect.ACCEPTABLE, desc = "Only actor2 threw
exception")
@Outcome(id = "0, 0", expect = Expect.ACCEPTABLE, desc = "No exceptions")
// This is a state object
@State
public class API_01_Simple {
```

```
    AtomicFixMemoryQueue<String> atomicFixMemoryQueue = new
AtomicFixMemoryQueue<>(1);

    @Actor
    public void actor1(II_Result r) {
        atomicFixMemoryQueue.pushBack("11");
        if (atomicFixMemoryQueue.getTail().get() > atomicFixMemoryQueue.size())
{
            r.r1 = 1;
        }
    }

    @Actor
    public void actor2(II_Result r) {
        atomicFixMemoryQueue.pushBack("11");
        if (atomicFixMemoryQueue.getTail().get() > atomicFixMemoryQueue.size())
{
            r.r2 = 1;
        }
    }
}
```

于是心满意足的跑出来我想要的结果:

```
Results across all configurations:

RESULT     SAMPLES     FREQ      EXPECT  DESCRIPTION
  0, 0  18,036,855   23.60%  Acceptable  No exceptions
  0, 1   5,387,399    7.05%  Acceptable  Only actor2 threw exception
  1, 0   5,915,478    7.74%  Acceptable  Only actor1 threw exception
  1, 1  47,077,292   61.61%  Acceptable  Both actors threw exception
```

那怎么避免这种尾指针越界呢，于是再度想到了取余操作，如果尾指针再前进的过程中回到了头位置，那么说明已经到达数组的边界。于是我们就将源码改写如下:

```
public boolean pushBack(T t) {
    if (Objects.isNull(t)) {
        throw new NullPointerException();
    }
    while (size.get() < capacity) {
        int attempts = 0;
        int currentTail = tail.get(); // 语句二
        int nextTail = (currentTail + 1) % capacity;
        if (tail.compareAndSet(currentTail, nextTail)) { // 语句三
            if (dataArray.compareAndSet(currentTail % capacity, null, t)) {
                size.incrementAndGet(); // 语句一
            }
            return true;
        } else {
            backOff(++attempts);
        }
    }
    return false;
}
```

我们接着跑上面的测试用例，看看这次会不会有尾指针的大小超过队列容量大小的问题。然后跑JCStress心满意足的没有跑出来尾指针大于队列容量的现象:

```
  RESULT    SAMPLES     FREQ      EXPECT  DESCRIPTION
    0, 0  2,029,568  100.00%  Acceptable  No exceptions
    0, 1          0    0.00%  Acceptable  Only actor2 threw exception
    1, 0          0    0.00%  Acceptable  Only actor1 threw exception
    1, 1          0    0.00%  Acceptable  Both actors threw exception
```

但是这个问题解决之后出现了新的问题，设A、B、C 三个线程同时进入while循环里面，此时队列容量为3，A线程将tail变为4，B读到tail为4，然后将tail来到0，此时队列的元素还没有线程消费，C读到语句tail = 0，然后tail = 1。这会导致出队问题，这是我的预测现在我们构造一个并发测试，来验证我们的猜想:

```
@JCStressTest

// These are the test outcomes.
@Outcome(id = "1", expect = Expect.ACCEPTABLE, desc = "accept")
@Outcome(id = "0", expect = Expect.ACCEPTABLE, desc = "accept")
@State
public class API_01_Simple {

    AtomicFixMemoryQueue<String> atomicFixMemoryQueue = new
AtomicFixMemoryQueue<>(5);

    @Actor
    public void actor1(I_Result r) {
        atomicFixMemoryQueue.pushBack("1");
        atomicFixMemoryQueue.pushBack("2");
        atomicFixMemoryQueue.pushBack("3");
        atomicFixMemoryQueue.pushBack("4");
        atomicFixMemoryQueue.pushBack("5");
        if (atomicFixMemoryQueue.getTail().get() > 0) {
            r.r1 = 1;
        }
    }

    @Actor
    public void actor2(I_Result r) {
        atomicFixMemoryQueue.pushBack("6");
        atomicFixMemoryQueue.pushBack("7");
        atomicFixMemoryQueue.pushBack("8");
        atomicFixMemoryQueue.pushBack("9");
        atomicFixMemoryQueue.pushBack("10");
        if (atomicFixMemoryQueue.getTail().get() > 0) {
            r.r1 = 1;
        }
    }
}
```

所以这里应当是先在对应的位置上写值，在移动尾指针。

```
public boolean pushBack(T t) {
    if (Objects.isNull(t)) {
        throw new NullPointerException();
    }
    while (size.get() < capacity) {
        int attempts = 0;
        int currentTail = tail.get();
```

```
        int nextTail = (currentTail + 1) % capacity;
        if (dataArray.compareAndSet(currentTail, null, t)) {
             tail.compareAndSet(currentTail, nextTail)
             size.incrementAndGet();
             return true;
        } else {
            backOff(++attempts);
        }
    }
    return false;
}
```

我们现在再跑一下测试看看会不会出现尾指针异常前移的情况:

```
Results across all configurations:

RESULT      SAMPLES     FREQ       EXPECT  DESCRIPTION
     0  45,389,824  100.00%  Acceptable  accept
     1           0    0.00%  Acceptable  accept
```

到现在为止我们都是单独的跑入队方法，现在考虑入队和出队并发，仅考虑pushBack和pushFront方法，pushBack方法在移动尾指针，popFront在移动头指针，一个我们需要预防的场景就是出队方法超过了入队方法，也就是出队速度超过了入队速度，得到了空值。测试代码如下:

```
@JCStressTest
@Outcome(id = "1", expect = Expect.ACCEPTABLE, desc = "accept")
@Outcome(id = "0", expect = Expect.ACCEPTABLE, desc = "accept")
@State
public class API_01_Simple02 {

    AtomicFixMemoryQueue<String> atomicFixMemoryQueue = new
AtomicFixMemoryQueue<>(5);

    @Actor
    public void actor1() {
        atomicFixMemoryQueue.pushBack("1");
    }

    @Actor
    public void actor3() {
        atomicFixMemoryQueue.pushBack("1");
    }

    @Actor
    public void actor2() {
        atomicFixMemoryQueue.popFront();
    }

    @Actor
    public void actor5() {
        atomicFixMemoryQueue.popFront();
    }

    @Actor
    public void actor4() {
        atomicFixMemoryQueue.popFront();
    }
```

```
    @Arbiter
    public void arbiter(I_Result r) {
        if (atomicFixMemoryQueue.getHead().get() >
atomicFixMemoryQueue.getTail().get()) {
            r.r1 = 1;
        }
    }
}
```

```
  RESULT      SAMPLES      FREQ      EXPECT  DESCRIPTION
       0  27,518,624    97.43%  Acceptable  accept
       1     725,344     2.57%  Acceptable  accept
```

果真跑出来了越界，于是我们就需要浅浅的对出队方法进行改造，如果头指针和尾指针在一个位置，这就有点像一个坏了。

```
RESULT      SAMPLES      FREQ      EXPECT  DESCRIPTION
     0  262,385,664  100.00%  Acceptable  accept
     1            0    0.00%  Acceptable  accept
```

到现在好像一切都很好，但我还是有些不放心，于是还是想系统的跑一下测试。

测试样例总结

不能丢值

设队列大小为n，x个线程入队操作，x < n, 则队列大小应当为x， 不能小于x。测试用例如下:

```
@JCStressTest
@Outcome(id = "1", expect = Expect.ACCEPTABLE, desc = "accept")
@Outcome(id = "0", expect = Expect.ACCEPTABLE, desc = "accept")
@State
public class API01SimpleO3 {

    AtomicFixMemoryQueue<String> atomicFixMemoryQueue = new
AtomicFixMemoryQueue<>(5);

    @Actor
    public void actor1() {
        atomicFixMemoryQueue.pushBack("1");
    }

    @Actor
    public void actor2() {
        atomicFixMemoryQueue.pushBack("2");
    }

    @Actor
    public void actor3() {
        atomicFixMemoryQueue.pushBack("3");
    }
```

```
    @Actor
    public void actor4() {
        atomicFixMemoryQueue.pushBack("4");
    }

    @Arbiter
    public void arbiter(I_Result r) {
        if (atomicFixMemoryQueue.size() < 4) {
            r.r1 = 1;
        }
        List<String> list = new ArrayList<>();
        for (int i = 0; i < 4; i++) {
            list.add(atomicFixMemoryQueue.popFront());
        }
        Collections.sort(list, String::compareTo);
        if (!Joiner.on(",").join(list).equals("1,2,3,4")) {
            r.r1 = 1;
        }
    }
}
```

```
      0  102,057,984  100.00%  Acceptable  accept
      1            0    0.00%  Acceptable  accept
```

不能让调用方出现异常打断执行

```
@JCStressTest
@Outcome(id = "1", expect = Expect.ACCEPTABLE, desc = "accept")
@Outcome(id = "0", expect = Expect.ACCEPTABLE, desc = "accept")
@State
public class API01SimpleO4 {

    AtomicFixMemoryQueue<String> atomicFixMemoryQueue = new
AtomicFixMemoryQueue<>(5);

    @Actor
    public void actor1(I_Result i_result) {
        try {
            for (int i = 0; i < 10; i++) {
                atomicFixMemoryQueue.pushBack("1");
            }
        } catch (Exception e) {
            e.printStackTrace();
            i_result.r1 = 1;
        }
    }

    @Actor
    public void actor2(I_Result i_result) {
        try {
            for (int i = 0; i < 10; i++) {
                atomicFixMemoryQueue.popFront();
            }
```

```
        } catch (Exception e) {
            e.printStackTrace();
            i_result.r1 = 1;
        }
    }
}
```

然后跑出来了异常：

```
        at java.base/java.lang.invoke.VarHandleReferences$Array.getVolatile(VarHandleReferences.java:602) <1 intern
        at com.example.demo.async.AtomicFixMemoryQueue.popFront(AtomicFixMemoryQueue.java:85)
        at com.example.demo.API01Simple04.actor2(API01Simple04.java:40)
        at com.example.demo.API01Simple04_jcstress.run_actor2(API01Simple04_jcstress.java:243)
        at com.example.demo.API01Simple04_jcstress.task_actor2(API01Simple04_jcstress.java:224)
        at com.example.demo.API01Simple04_jcstress.access$100(API01Simple04_jcstress.java:19)
        at com.example.demo.API01Simple04_jcstress$5.internalRun(API01Simple04_jcstress.java:134)
        at org.openjdk.jcstress.infra.runners.CounterThread.run(CounterThread.java:38)
    java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
        at java.base/java.lang.invoke.VarHandle$1.apply(VarHandle.java:2187)
        at java.base/java.lang.invoke.VarHandle$1.apply(VarHandle.java:2184) <5 internal lines>
        at java.base/java.lang.invoke.VarHandleReferences$Array.getVolatile(VarHandleReferences.java:602) <1 intern
        at com.example.demo.async.AtomicFixMemoryQueue.popFront(AtomicFixMemoryQueue.java:85)
        at com.example.demo.API01Simple04.actor2(API01Simple04.java:40)
        at com.example.demo.API01Simple04_jcstress.run_actor2(API01Simple04_jcstress.java:243)
        at com.example.demo.API01Simple04_jcstress.task_actor2(API01Simple04_jcstress.java:224)
```

现在代码肯定是有问题的，我们接着来看popFront方法：

```
public T popFront() {
    while (size.get() > 0) {
        int headIndex = head.get();
        int attempts = 0;
        if (head.compareAndSet(headIndex, headIndex + 1)) { // 语句一
            size.decrementAndGet();
            T result = dataArray.get(headIndex);
            dataArray.compareAndSet(headIndex,result,null);
            return result;
        } else {
            backOff(++attempts);
        }
    }
    return null;
}
```

我觉得问题应该出现在语句一上面，设队列只剩最后一个元素，也就是headIndex = 3， A线程进来完成自增，头指针为4，B线程进来将头指针变为5，于是就出现了数组下标越界，现在让我们对popFront进行改造：

```
 public T popFront() {
        while (size.get() > 0) {
            int headIndex = head.get();
            int next = headIndex++ % capacity;
            int attempts = 0;
            T result = dataArray.getAcquire(next);
            if (Objects.isNull(result)) {
                return null;
            }
            if (dataArray.compareAndSet(next, result, null)) {
                if (head.compareAndSet(next, headIndex)) {
                    size.decrementAndGet();
```

```
                    return result;
                }

            } else {
                backOff(++attempts);
            }
        }
        return null;
    }
```

然后结果正常。

队列里面不能存储空值

```
@JCStressTest
@Outcome(id = "1", expect = Expect.ACCEPTABLE, desc = "accept")
@Outcome(id = "0", expect = Expect.ACCEPTABLE, desc = "accept")
@State
public class API01SimpleO5 {

    AtomicFixMemoryQueue<String> atomicFixMemoryQueue = new
AtomicFixMemoryQueue<>(5);

    @Actor
    public void actor1(I_Result i_result) {
        for (int i = 0; i < 10; i++) {
            atomicFixMemoryQueue.pushBack("1");
        }
    }

    @Actor
    public void actor2(I_Result i_result) {
        for (int i = 0; i < 10; i++) {
            atomicFixMemoryQueue.popFront();
        }
    }

    @Arbiter
    public void arbiter(I_Result r) {
        int size = atomicFixMemoryQueue.size();
        if (size > 0) {
            for (int i = 0; i < size; i++) {
                if (atomicFixMemoryQueue.popFront() == null) {
                    r.r1 = 1;
                }
            }
        }
    }
}
```

跑出来结果如下:

```
  RESULT      SAMPLES     FREQ      EXPECT  DESCRIPTION
       0      215,268    0.20%  Acceptable  accept
       1  108,928,796   99.80%  Acceptable  accept
```

那也就是说size的大小是不正确的，跑出这个结果的时候，我的脑袋中开始不断构造场景，去模拟怎样的场景，这是一种思维方式，但是怎么都构造不出来，我在吃饭的时候，换到了另一种思维方式，由结果翻过来推原因，如果size大于0，但是出队元素为空，只说明一点，就是size的大小没有被正常增减。我们重新写一下测试用例，然后看下结果:

```
@JCStressTest
@Outcome(id = "1, 0, 1, 1, 1, 1", expect = Expect.ACCEPTABLE, desc = "accept")
@Outcome(id = "0, 1, 1, 1, 1, 1", expect = Expect.ACCEPTABLE, desc = "accept")
@State
public class API01SimpleO5 {

    AtomicFixMemoryQueue<String> atomicFixMemoryQueue = new
AtomicFixMemoryQueue<>(5);

    @Actor
    public void actor1() {
        for (int i = 0; i < 10; i++) {
            atomicFixMemoryQueue.pushBack("1");
        }
    }

    @Actor
    public void actor2( ) {
        for (int i = 0; i < 10; i++) {
            atomicFixMemoryQueue.popFront();
        }
    }

    @Arbiter
    public void arbiter(IIIIII_Result r) {
        int size = atomicFixMemoryQueue.size();
        r.r1 = atomicFixMemoryQueue.getTail().get();
        r.r2 = atomicFixMemoryQueue.getHead().get();
        r.r3 = size;
        int notNullTotal = 0;
        int nullTotal = 0;
        if (size > 0) {
            for (int i = 0; i < size; i++) {
                if (atomicFixMemoryQueue.popFront() == null) {
                    nullTotal++;
                }else{
                    notNullTotal++;
                }
            }
        }
        // 说明有null,也有非空
        if (notNullTotal < size){
            r.r4 = 1;
        }else {
            r.r6 = 1;
```

```
        }
        // 说明全出队了，size大小没有正常递减
        if (nullTotal == size){
            r.r5 = 1;
        }
    }
}
```

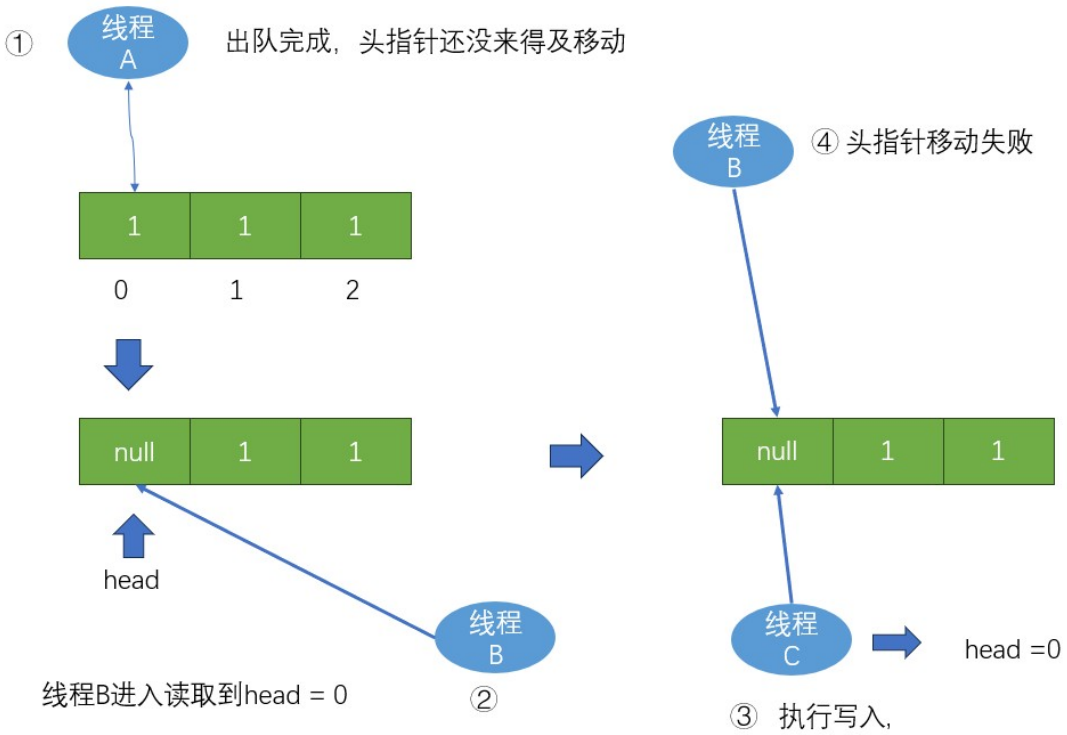
观测结果为:

```
  0, 0, 0, 0, 1, 1     435,104    0.64%   Forbidden  No default case provided,
assume Forbidden // size = 0 正常结果
  0, 0, 5, 1, 1, 0  66,980,001   98.31%   Forbidden  No default case provided,
assume Forbidden // size = 5,  出队为Null

  0, 1, 4, 0, 0, 1      44,223    0.06%   Forbidden  No default case provided,
assume Forbidden // size = 4 , 不为空的元素和size 相
  1, 1, 0, 0, 1, 1      67,035    0.10%   Forbidden  No default case provided,
assume Forbidden  // size =0 正常结果
  1, 1, 5, 1, 1, 0      34,724    0.05%   Forbidden  No default case provided,
assume Forbidden // size = 5 出队全为null
  1, 2, 4, 0, 0, 1         451   <0.01%   Forbidden  No default case provided,
assume Forbidden //  size = 4  不为空的元素和size 相等
  2, 2, 0, 0, 1, 1      81,064    0.12%   Forbidden  No default case provided,
assume Forbidden // size = 0  正常结果
  4, 4, 5, 1, 1, 0      49,935    0.07%   Forbidden  No default case provided,
assume Forbidden  //  size = 4  不为空的元素和size 相等
```

现在来看都属于size大于0，但是出队的时候全为空，我们接着把popFront方法拿出来看一下:

```
public T popFront() {
    while (size.get() > 0) {
        int headIndex = head.get();
        int next = (headIndex + 1) % capacity;
        int attempts = 0;
        T result = dataArray.getAcquire(headIndex);
        if (Objects.isNull(result)) {
            return null;
        }
        if (dataArray.compareAndSet(headIndex, result, null)) { //语句二
            if (head.compareAndSet(headIndex, next)) { // 语句三
                size.decrementAndGet(); // 语句四
            }
            return result;
        } else {
            backOff(++attempts);
        }
    }
    return null;
}
```

这意味着语句一在正常执行之后，也就是语句二在某些情况下失败，导致语句三没有正常执行，那是怎样的情况呢？ 注意观察我们的测试用例，入队的都是相同元素，设队列大小为3，存储的都为1,1,1。设A线程进入获取头指针为0，然后将0位置设置为null，在执行语句二的时候，B线程进入获取head = 0，然后走到语句二的时候，线程C执行写入，将0位置写入元素1，于是语句二执行成功，但是语句三执行失败，这就导致少递减了一次。如下图所示:



于是我将代码修改为:

```
public T popFront() {
    while (true) {
        if (size.get() == 0) {
            return null;
        }
        int headIndex = head.get();
        int next = (headIndex + 1) % capacity;
        int attempts = 0;
        T result = dataArray.getAcquire(headIndex);
        if (Objects.isNull(result)) {
            return null;
        }
        if (dataArray.compareAndSet(headIndex, result, null)) {
            size.decrementAndGet();
            head.compareAndSet(headIndex, next);
            return result;
        } else {
            backOff(++attempts);
        }
    }
}
```

然后对JCStress代码进行修正:

```
@JCStressTest
@Outcome(id = "1, 0, 1, 1, 1, 1", expect = Expect.ACCEPTABLE, desc = "accept")
@Outcome(id = "0, 1, 1, 1, 1, 1", expect = Expect.ACCEPTABLE, desc = "accept")
@Outcome(id = "0, 0, 0, 0, 1, 1 ", expect = Expect.ACCEPTABLE, desc = "accept")
@State
public class API01SimpleO5 {
```

```
    AtomicFixMemoryQueue<String> atomicFixMemoryQueue = new
AtomicFixMemoryQueue<>(5);

    @Actor
    public void actor1() {
        for (int i = 0; i < 10; i++) {
            atomicFixMemoryQueue.pushBack(i + "");
        }
    }

    @Actor
    public void actor2( ) {
        for (int i = 0; i < 10; i++) {
            atomicFixMemoryQueue.popFront();
        }
    }

    @Arbiter
    public void arbiter(IIIIII_Result r) {
        int size = atomicFixMemoryQueue.size();
        r.r1 = atomicFixMemoryQueue.getTail().get();
        r.r2 = atomicFixMemoryQueue.getHead().get();
        r.r3 = size;
        int notNullTotal = 0;
        int nullTotal = 0;
        if (size > 0) {
            for (int i = 0; i < size; i++) {
                if (atomicFixMemoryQueue.popFront() == null) {
                    nullTotal++;
                }else{
                    notNullTotal++;
                }
            }
        }
        // 说明有null,也有非空
        if (0 < notNullTotal && notNullTotal < size){
            r.r4 = 1;
        }
        if (notNullTotal == size){
            r.r6 = 1;
        }
        if (0 < nullTotal && nullTotal < size){
            r.r5 = 1;
        }
    }
}
```

```
0, 0, 0, 0, 0, 1   1,152,338    2.87%   Forbidden  No default case provided,
assume Forbidden
0, 0, 5, 0, 0, 1  38,788,921   96.74%   Forbidden  No default case provided,
assume Forbidden
```

看这个结果就是我们所需要的，即在不为空的情况下，出队的元素数量和队列的实际容量相等。

小插曲

这里将我思考错误的结果也放出来，刚开始考虑读写并发的时候，出队方法的代码如下:

```
public T popFront() {
    while (true) {
        if (size.get() == 0) {
            return null;
        }
        int headIndex = head.get();
        int tailIndex = tail.get();
        if (headIndex == tailIndex){ // 语句一
           return null;
        }
        int next = (headIndex + 1) % capacity;
        int attempts = 0;
        T result = dataArray.getAcquire(headIndex);
        if (Objects.isNull(result)) {
            return null;
        }
        if (dataArray.compareAndSet(headIndex, result, null)) {
            size.decrementAndGet();
            head.compareAndSet(headIndex, next);
            return result;
        } else {
            backOff(++attempts);
        }
    }
}
```

但其实结合入队方法来看，入队方法之后头指针就会走到数组的0位置，然后这个时候再出队，触发语句一，就会造成队列里面有值，但是出队元素为null的状况。这个时候就突出了JCStress的重要性，我迫切的希望我根据理论进行预测的时候，结果是可观测的，可验证的。如果用普通的线程池去跑这些例子，跑出来的结果是不稳定的。尽管并发的行为的确是不可预测的，就像扔一枚均匀的硬币一样，在扔出去之前，我们无法知道硬币会朝上还是朝下，但是我们知道实验足够的多，正面出现的次数和背面出现的次数会向1:1靠近，JCStress就满足了我的要求替我跑足够的次数，跑出所有并发的结果。

总结一下

在Java里面基于数组线程安全的队列有ArrayBlockingQueue、Disruptor、MpscQueue(来自JCTools)，ArrayBlockingQueue通过加锁来实现线程安全。Disruptor和MpscQueue通过原子操作来实现线程安全，MpscQueue适用于多生产者单消费者，而Disruptor相对更灵活一些，可以指定等待策略:

- BlockingWaitStrategy: 锁和条件变量实现的等待
- SleepingWaitStrategy: 自旋 + yield + sleep
- YieldingWaitStrategy: 自旋 + yield +自旋
- BusySpinWaitStrategy: 自旋

Disruptor有更加灵活的消费者模式，支持单生产者、多生产者，单消费者、多消费者。Apache Storm、Camel、Log4j 2在内的很多知名项目都应用了Disruptor以获取高性能。写本篇文章的时候，我没有选择一开始告诉正确的答案，我选择的方案是演进式的，原因在于在构造过程中出错是正常的事情，有了错误才明白为什么对。有时候是这样，你必须先有一个错误的答案，才知道正确的答案在哪里。

在写本篇的时候，我再次想对线程安全问题进行作答，我希望得到的定义直观一点，至少能对工程的指导性强一点，一般来说线程安全的定义如下:

> Thread-safe code is code that will work even if many Threads are executing it simultaneously.
>
> 线程安全代码是指即使有多个线程同时执行也能正常运行的代码。

我认为这句话的问题在于什么叫正常运行她没有回答，这句话事实上没有回答什么问题，如果让你编写一个线程安全的类，你能从这里拿到多少指导意见呢？这个答案来自于StackOverFlow，顺着这个答案我找到了参考链接[5]，在连接[5]里面引用了维基百科:

> In particular, it must satisfy the need for multiple threads to access the same shared data,*...and the need for a shared piece of data to be accessed by only one thread at any given time
>
> 特别是，它必须满足多个线程访问相同共享数据的需求，以及在任何给定时间只有一个线程可以访问共享数据的需求。

这看起来像是创建线程安全的技术，但并不像是线程安全的定义，在现在的维基百科中对线程安全的定义如下:

> In multi-threaded computer programming, a function is thread-safe when it can be invoked or accessed concurrently by multiple threads without causing unexpected behavior, race conditions, or data corruption.
>
> 在多线程计算机编程中，如果一个函数能够被多个线程同时调用或访问并且能够正确处理竞争条件，避免因此产生意外行为或数据损坏，那么这个函数就被认为是线程安全的。

现在来看这个定义进步了一点，给出了当我们声明这个函数是安全的时候，这个函数不会出现什么，那什么是竞态条件:

> 竞态条件发生在两个或更多线程可以访问共享数据，并且它们试图同时改变这些数据时。由于线程调度算法可以在任何时候在线程之间进行切换，你无法知道线程尝试访问共享数据的确切顺序。因此，数据变化的结果取决于线程调度算法，即线程之间在"竞争"访问/改变数据。

好像清晰了那么一点点，但在Java里面是以类为单位对外提供功能调用的，那对于一个类来说，当它声明她是线程安全意味着什么呢? 在《Java多线程编程实战指南》中是这么说的:

> 一般而言，如果一个类在单线程环境下能够运作正常，并且在多线程下面不必做任何改变的情况下也能运作正常，那么相应的我们称这个类具备线程安全性。

所谓的正常我想至少不应该导致多个线程在调用这个类的时候，出现无限循环、死锁、内存泄漏、导致JVM进程崩溃等状况。那这个定义好了一点？ 在参考文档[5] 有一个评论：

> 在C#中的工作窃取队列中，只在一个方法中实现了线程安全，即即TrySteal()方法，其他方法，如Push()和TryPop()，根据设计始终由同一线程调用。从不同线程调用它们将违反约定。

这意味着对于一个类来说是线程安全的时候，你还需要按照它要求的方式去调用，才不会出现异常状况。所以当我们说一个方法是线程安全的时候，这意味着这个方法良好的处理了竞态条件，不会在多线程调用下出现异常(死锁、内存泄漏)。当我们将安全上升到一个类，这意味着调用要按照约定的方式调用，可能某些方法允许多线程调用，某些方法不允许多线程并发调用，这就像是说明书一样。在按约定的使用情况下，我们接着提出行为的正确性，这意味着和具体的实例耦合了起来，比如对于一个线程安全的集合来说，在多线程向集合里面放入元素，不应当丢值。对于集合来说，还有一个特性是迭代，如果迭代器上不一定能反应出写线程的修改，那么我们称这种集合是弱一致的，如果一定能反应出来，这意味着是强一致的。

我也尝试跟别人讨论，但是有些人会讲从理论的角度证明，但是计算机是讲究实践的学科，不像数学一样，在19世纪中期以前，几乎所有的数学家都愿意相信，除去某些例外的孤立点，连续函数总是可微的，那时的教科书甚至证明了这样的定理，能得出这样结论的原因在于当时的函数表示手段有限，但是随着级数理论的发展，数学家可以通过函数项级数来表示更广泛的函数类，最终Weierstrass构造出来了一个处处连续处处不可导的函数，来为大家愿意相信的命题做出了否定。0

参考资料

[1] LMAX Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads https://lmax-exchange.github.io/disruptor/disruptor.html

[2] MySQL事务学习笔记(二) 相识篇 https://juejin.cn/post/7074171192508153892#heading-4

[3] onSpinWait() method of Thread class - Java 9 https://stackoverflow.com/questions/44622384/onspinwait-method-of-thread-class-java-9

[4] JMM测试利器-JCStress学习笔记 https://juejin.cn/post/7233359844974952485?searchId=20240720142027F610C3F02973D1D480A1

[5] What is this thing you call "thread safe"? https://learn.microsoft.com/en-us/archive/blogs/ericlippert/what-is-this-thing-you-call-thread-safe

[6] What is a race condition? https://stackoverflow.com/questions/34510/what-is-a-race-condition