JUC工具类: Exchanger详解

Exchanger是用于线程协作的工具类, 主要用于两个线程之间的数据交换。

面试问题去理解Exchanger

- Exchanger主要解决什么问题?
- 对比SynchronousQueue，为什么说Exchanger可被视为 SynchronousQueue 的双向形式?
- Exchanger在不同的JDK版本中实现有什么差别?
- Exchanger实现机制?
- Exchanger已经有了slot单节点，为什么会加入arena node数组? 什么时候会用到数组?
- arena可以确保不同的slot在arena中是不会相冲突的，那么是怎么保证的呢?
- 什么是伪共享，Exchanger中如何体现的?
- Exchanger实现举例

Exchanger简介

Exchanger用于进行两个线程之间的数据交换。它提供一个同步点，在这个同步点，两个线程可以交换彼此的数据。这两个线程通过exchange()方法交换数据，当一个线程先执行exchange()方法后，它会一直等待第二个线程也执行exchange()方法，当这两个线程到达同步点时，这两个线程就可以交换数据了。

Exchanger实现机制

```
for (;;) {
    if (slot is empty) { // offer
        // slot为空时，将item 设置到Node 中
        place item in a Node;
        if (can CAS slot from empty to node) {
            // 当将node通过CAS交换到slot中时，挂起线程等待被唤醒
            wait for release;
            // 被唤醒后返回node中匹配到的item
            return matching item in node;
        }
    } else if (can CAS slot from node to empty) { // release
         // 将slot设置为空
        // 获取node中的item，将需要交换的数据设置到匹配的item
        get the item in node;
        set matching item in node;
        // 唤醒等待的线程
        release waiting thread;
    }
    // else retry on CAS failure
}
```

比如有2条线程A和B，A线程交换数据时，发现slot为空，则将需要交换的数据放在slot中等待其它线程进来交换数据，等线程B进来，读取A设置的数据，然后设置线程B需要交换的数据，然后唤醒A线程，原理就是这么简单。但是当多个线程之间进行交换数据时就会出现问题，所以Exchanger加入了slot数组。

Exchanger源码解析

内部类 - *Participant*

```
static final class Participant extends ThreadLocal<Node> {
    public Node initialValue() { return new Node(); }
}
```

Participant的作用是为每个线程保留唯一的一个Node节点, 它继承ThreadLocal，说明每个线程具有不同的状态。

内部类 - *Node*

```
@sun.misc.Contended static final class Node {
     // arena的下标，多个槽位的时候利用
    int index;
    // 上一次记录的Exchanger.bound
    int bound;
    // 在当前bound下CAS失败的次数；
    int collides;
    // 用于自旋；
    int hash;
    // 这个线程的当前项，也就是需要交换的数据；
    Object item;
    //做releasing操作的线程传递的项；
    volatile Object match;
    //挂起时设置线程值，其他情况下为null；
    volatile Thread parked;
}
```

在Node定义中有两个变量值得思考：bound以及collides。前面提到了数组area是为了避免竞争而产生的，如果系统不存在竞争问题，那么完全没有必要开辟一个高效的arena来徒增系统的复杂性。首先通过单个slot的exchanger来交换数据，当探测到竞争时将安排不同的位置的slot来保存线程Node，并且可以确保没有slot会在同一个缓存行上。如何来判断会有竞争呢? CAS替换slot失败，如果失败，则通过记录冲突次数来扩展arena的尺寸，我们在记录冲突的过程中会跟踪“bound”的值，以及会重新计算冲突次数在bound的值被改变时。

核心属性

```
private final Participant participant;
private volatile Node[] arena;
private volatile Node slot;
```

- **为什么会有 arena数组槽?**

slot为单个槽，arena为数组槽, 他们都是Node类型。在这里可能会感觉到疑惑，slot作为Exchanger交换数据的场景，应该只需要一个就可以了啊? 为何还多了一个Participant 和数组类型的arena呢? 一个slot交换场所原则上来说应该是可以的，但实际情况却不是如此，多个参与者使用同一个交换场所时，会存在严重伸缩性问题。既然单个交换场所存在问题，那么我们就安排多个，也就是数组arena。通过数组arena来安排不同的线程使用不同的slot来降低竞争问题，并且可以保证最终一定会成对交换数据。但是**Exchanger不是一来就会生成arena数组来降低竞争，只有当产生竞争是才会生成arena数组。**

- **那么怎么将Node与当前线程绑定呢?**

Participant，Participant 的作用就是为每个线程保留唯一的一个Node节点，它继承ThreadLocal，同时在Node节点中记录在arena中的下标index。

构造函数

```
/**
* Creates a new Exchanger.
*/
public Exchanger() {
    participant = new Participant();
}
```

初始化participant对象。

核心方法 - *exchange(V x)*

等待另一个线程到达此交换点(除非当前线程被中断)，然后将给定的对象传送给该线程，并接收该线程的对象。

```
public V exchange(V x) throws InterruptedException {
    Object v;
    // 当参数为null时需要将item设置为空的对象
    Object item = (x == null) ? NULL_ITEM : x; // translate null args
    // 注意到这里的这个表达式是整个方法的核心
    if ((arena != null ||
            (v = slotExchange(item, false, 0 L)) == null) &&
        ((Thread.interrupted() || // disambiguates null return
            (v = arenaExchange(item, false, 0 L)) == null)))
        throw new InterruptedException();
    return (v == NULL_ITEM) ? null : (V) v;
}
```

这个方法比较好理解：arena为数组槽，如果为null，则执行slotExchange()方法，否则判断线程是否中断，如果中断值抛出InterruptedException异常，没有中断则执行arenaExchange()方法。整套逻辑就是：如果slotExchange(Object item, boolean timed, long ns)方法执行失败了就执行arenaExchange(Object item, boolean timed, long ns)方法，最后返回结果V。

NULL_ITEM 为一个空节点，其实就是一个Object对象而已，slotExchange()为单个slot交换。

slotExchange(Object item, boolean timed, long ns)

```
private final Object slotExchange(Object item, boolean timed, long ns) {
    // 获取当前线程node对象
    Node p = participant.get();
    // 当前线程
    Thread t = Thread.currentThread();
    // 若果线程被中断，就直接返回null
    if (t.isInterrupted()) // preserve interrupt status so caller can recheck
        return null;
    // 自旋
    for (Node q;;) {
        // 将slot值赋给q
        if ((q = slot) != null) {
             // slot 不为null，即表示已有线程已经把需要交换的数据设置在slot中了
            // 通过CAS将slot设置成null
            if (U.compareAndSwapObject(this, SLOT, q, null)) {
                // CAS操作成功后，将slot中的item赋值给对象v，以便返回。
                // 这里也是就读取之前线程要交换的数据
                Object v = q.item;
                // 将当前线程需要交给的数据设置在q中的match
                q.match = item;
                 // 获取被挂起的线程
                Thread w = q.parked;
                if (w != null)
                    // 如果线程不为null，唤醒它
                    U.unpark(w);
                // 返回其他线程给的V
                return v;
            }
            // create arena on contention, but continue until slot null
            // CAS 操作失败，表示有其它线程竞争，在此线程之前将数据已取走
            // NCPU:CPU的核数
            // bound == 0 表示arena数组未初始化过，CAS操作bound将其增加SEQ
            if (NCPU > 1 && bound == 0 &&
                U.compareAndSwapInt(this, BOUND, 0, SEQ))
                // 初始化arena数组
                arena = new Node[(FULL + 2) << ASHIFT];
        }
        // 上面分析过，只有当arena不为空才会执行slotExchange方法的
        // 所以表示刚好已有其它线程加入进来将arena初始化
        else if (arena != null)
            // 这里就需要去执行arenaExchange
            return null; // caller must reroute to arenaExchange
        else {
            // 这里表示当前线程是以第一个线程进来交换数据
            // 或者表示之前的数据交换已进行完毕，这里可以看作是第一个线程
            // 将需要交换的数据先存放在当前线程变量p中
            p.item = item;
            // 将需要交换的数据通过CAS设置到交换区slot
            if (U.compareAndSwapObject(this, SLOT, null, p))
                // 交换成功后跳出自旋
                break;
            // CAS操作失败，表示有其它线程刚好先于当前线程将数据设置到交换区slot
            // 将当前线程变量中的item设置为null，然后自旋获取其它线程存放在交换区slot的数据
            p.item = null;
        }
    }

    // await release
```

```
        // 执行到这里表示当前线程已将需要的交换的数据放置于交换区slot中了，
        // 等待其它线程交换数据然后唤醒当前线程
        int h = p.hash;
        long end = timed ? System.nanoTime() + ns : 0 L;
        // 自旋次数
        int spins = (NCPU > 1) ? SPINS : 1;
        Object v;
        // 自旋等待直到p.match不为null，也就是说等待其它线程将需要交换的数据放置于交换区slot
        while ((v = p.match) == null) {
            // 下面的逻辑主要是自旋等待，直到spins递减到0为止
            if (spins > 0) {
                h ^= h << 1;
                h ^= h >>> 3;
                h ^= h << 10;
                if (h == 0)
                    h = SPINS | (int) t.getId();
                else if (h < 0 && (--spins & ((SPINS >>> 1) - 1)) == 0)
                    Thread.yield();
            } else if (slot != p)
                spins = SPINS;
            // 此处表示未设置超时或者时间未超时
            else if (!t.isInterrupted() && arena == null &&
                (!timed || (ns = end - System.nanoTime()) > 0 L)) {
                // 设置线程t被当前对象阻塞
                U.putObject(t, BLOCKER, this);
                // 给p挂机线程的值赋值
                p.parked = t;
                if (slot == p)
                    // 如果slot还没有被置为null，也就表示暂未有线程过来交换数据，需要将当前线程挂起
                    U.park(false, ns);
                // 线程被唤醒，将被挂起的线程设置为null
                p.parked = null;
                // 设置线程t未被任何对象阻塞
                U.putObject(t, BLOCKER, null);
            // 不是以上条件时(可能是arena已不为null或者超时)
            } else if (U.compareAndSwapObject(this, SLOT, p, null)) {
                 // arena不为null则v为null,其它为超时则v为超市对象TIMED_OUT，并且跳出循环
                v = timed && ns <= 0 L && !t.isInterrupted() ? TIMED_OUT : null;
                break;
            }
        }
        // 取走match值，并将p中的match置为null
        U.putOrderedObject(p, MATCH, null);
        // 设置item为null
        p.item = null;
        p.hash = h;
        // 返回交换值
        return v;
    }
```

程序首先通过participant获取当前线程节点Node。检测是否中断，如果中断return null，等待后续抛出InterruptedException异常。

- 如果slot不为null，则进行slot消除，成功直接返回数据V，否则失败，则创建arena消除数组。
- 如果slot为null，但arena不为null，则返回null，进入arenaExchange逻辑。
- 如果slot为null，且arena也为null，则尝试占领该slot，失败重试，成功则跳出循环进入spin+block(自旋+阻塞)模式。

在自旋+阻塞模式中，首先取得结束时间和自旋次数。如果match(做releasing操作的线程传递的项)为null，其首先尝试spins+随机次自旋(改自旋使用当前节点中的hash，并改变之)和退让。当自旋数为0后，假如slot发生了改变(slot != p)则重置自旋数并重试。否则假如：当前未中断&arena为null&(当前不是限时版本或者限时版本+当前时间未结束)：阻塞或者限时阻塞。假如：当前中断或者arena不为null或者当前为限时版本+时间已经结束：不限时版本：置v为null；限时版本：如果时间结束以及未中断则TIMED_OUT；否则给出null(原因是探测到arena非空或者当前线程中断)。

match不为空时跳出循环。

arenaExchange(Object item, boolean timed, long ns)

此方法被执行时表示多个线程进入交换区交换数据，arena数组已被初始化，此方法中的一些处理方式和slotExchange比较类似，它是通过遍历arena数组找到需要交换的数据。

```
// timed 为true表示设置了超时时间，ns为>0的值，反之没有设置超时时间
private final Object arenaExchange(Object item, boolean timed, long ns) {
    Node[] a = arena;
    // 获取当前线程中的存放的node
    Node p = participant.get();
    //index初始值0
    for (int i = p.index;;) { // access slot at i
        // 遍历，如果在数组中找到数据则直接交换并唤醒线程，如未找到则将需要交换给其它线程的数据放置于数组中
        int b, m, c;
        long j; // j is raw array offset
        // 其实这里就是向右遍历数组，只是用到了元素在内存偏移的偏移量
        // q实际为arena数组偏移(i + 1) *  128个地址位上的node
        Node q = (Node) U.getObjectVolatile(a, j = (i << ASHIFT) + ABASE);
        // 如果q不为null，并且CAS操作成功，将下标j的元素置为null
        if (q != null && U.compareAndSwapObject(a, j, q, null)) {
            // 表示当前线程已发现有交换的数据，然后获取数据，唤醒等待的线程
            Object v = q.item; // release
            q.match = item;
            Thread w = q.parked;
            if (w != null)
                U.unpark(w);
            return v;
        // q 为null 并且 i 未超过数组边界
        } else if (i <= (m = (b = bound) & MMASK) && q == null) {
             // 将需要给其它线程的item赋予给p中的item
            p.item = item; // offer
            if (U.compareAndSwapObject(a, j, null, p)) {
                // 交换成功
                long end = (timed && m == 0) ? System.nanoTime() + ns : 0 L;
                Thread t = Thread.currentThread(); // wait
                // 自旋直到有其它线程进入，遍历到该元素并与其交换，同时当前线程被唤醒
                for (int h = p.hash, spins = SPINS;;) {
                    Object v = p.match;
                    if (v != null) {
                        // 其它线程设置的需要交换的数据match不为null
                        // 将match设置null,item设置为null
                        U.putOrderedObject(p, MATCH, null);
                        p.item = null; // clear for next use
                        p.hash = h;
                        return v;
                    } else if (spins > 0) {
                        h ^= h << 1;
```

```
                    h ^= h >>> 3;
                    h ^= h << 10; // xorshift
                    if (h == 0) // initialize hash
                        h = SPINS | (int) t.getId();
                    else if (h < 0 && // approx 50% true
                        (--spins & ((SPINS >>> 1) - 1)) == 0)
                        Thread.yield(); // two yields per wait
                } else if (U.getObjectVolatile(a, j) != p)
                    // 和slotExchange方法中的类似，arena数组中的数据已被CAS设置
                   // match值还未设置，让其再自旋等待match被设置
                    spins = SPINS; // releaser hasn't set match yet
                else if (!t.isInterrupted() && m == 0 &&
                    (!timed ||
                        (ns = end - System.nanoTime()) > 0 L)) {
                    // 设置线程t被当前对象阻塞
                    U.putObject(t, BLOCKER, this); // emulate LockSupport
                     // 线程t赋值
                    p.parked = t; // minimize window
                    if (U.getObjectVolatile(a, j) == p)
                        // 数组中对象还相等，表示线程还未被唤醒，唤醒线程
                        U.park(false, ns);
                    p.parked = null;
                     // 设置线程t未被任何对象阻塞
                    U.putObject(t, BLOCKER, null);
                } else if (U.getObjectVolatile(a, j) == p &&
                    U.compareAndSwapObject(a, j, p, null)) {
                    // 这里给bound增加加一个SEQ
                    if (m != 0) // try to shrink
                        U.compareAndSwapInt(this, BOUND, b, b + SEQ - 1);
                    p.item = null;
                    p.hash = h;
                    i = p.index >>>= 1; // descend
                    if (Thread.interrupted())
                        return null;
                    if (timed && m == 0 && ns <= 0 L)
                        return TIMED_OUT;
                    break; // expired; restart
                }
            }
        } else
            // 交换失败，表示有其它线程更改了arena数组中下标i的元素
            p.item = null; // clear offer
    } else {
        // 此时表示下标不在bound & MMASK或q不为null但CAS操作失败
       // 需要更新bound变化后的值
        if (p.bound != b) { // stale; reset
            p.bound = b;
            p.collides = 0;
            // 反向遍历
            i = (i != m || m == 0) ? m : m - 1;
        } else if ((c = p.collides) < m || m == FULL ||
            !U.compareAndSwapInt(this, BOUND, b, b + SEQ + 1)) {
             // 记录CAS失败的次数
            p.collides = c + 1;
            // 循环遍历
            i = (i == 0) ? m : i - 1; // cyclically traverse
        } else
            // 此时表示bound值增加了SEQ+1
            i = m + 1; // grow
        // 设置下标
        p.index = i;
```

```
        }
    }
}
```

首先通过participant取得当前节点Node，然后根据当前节点Node的index去取arena中相对应的节点node。

- **前面提到过arena可以确保不同的slot在arena中是不会相冲突的，那么是怎么保证的呢?**

```
arena = new Node[(FULL + 2) << ASHIFT];
// 这个arena到底有多大呢? 我们先看FULL 和ASHIFT的定义:
static final int FULL = (NCPU >= (MMASK << 1)) ? MMASK : NCPU >>> 1;
private static final int ASHIFT = 7;

private static final int NCPU = Runtime.getRuntime().availableProcessors();
private static final int MMASK = 0xff;        // 255
// 假如我的机器NCPU = 8 ，则得到的是768大小的arena数组。然后通过以下代码取得在arena中的节点:

Node q = (Node)U.getObjectVolatile(a, j = (i << ASHIFT) + ABASE);
// 它仍然是通过右移ASHIFT位来取得Node的，ABASE定义如下:

Class<?> ak = Node[].class;
ABASE = U.arrayBaseOffset(ak) + (1 << ASHIFT);
// U.arrayBaseOffset获取对象头长度，数组元素的大小可以通过unsafe.arrayIndexScale(T[].class) 方法获取到。这也就是说要访问类型为T的第N个元素的话，你的偏移量offset应该是arrayOffset+N*arrayScale。也就是说BASE = arrayOffset+ 128 。
```

- **用@sun.misc.Contended来规避伪共享?**

伪共享说明：假设一个类的两个相互独立的属性a和b在内存地址上是连续的(比如FIFO队列的头尾指针)，那么它们通常会被加载到相同的cpu cache line里面。并发情况下，如果一个线程修改了a，会导致整个cache line失效(包括b)，这时另一个线程来读b，就需要从内存里再次加载了，这种多线程频繁修改ab的情况下，虽然a和b看似独立，但它们会互相干扰，非常影响性能。

我们再看Node节点的定义, 在Java 8 中我们是可以利用sun.misc.Contended来规避伪共享的。所以说通过 << ASHIFT 方式加上sun.misc.Contended，所以使得任意两个可用Node不会再同一个缓存行中。

```
@sun.misc.Contended static final class Node{
....
}
```

我们再次回到arenaExchange()。取得arena中的node节点后，如果定位的节点q 不为空，且CAS操作成功，则交换数据，返回交换的数据，唤醒等待的线程。

- 如果q等于null且下标在bound & MMASK范围之内，则尝试占领该位置，如果成功，则采用自旋 + 阻塞的方式进行等待交换数据。
- 如果下标不在bound & MMASK范围之内获取由于q不为null但是竞争失败的时候：消除p。加入bound 不等于当前节点的bond(b != p.bound)，则更新p.bound = b，collides = 0 ， i = m或者m - 1。如果冲突的次数不到m 获取m 已经为最大值或者修改当前bound的值失败，则通过增加一次collides以及循环递减下标i的值；否则更新当前bound的值成功：我们令i为m+1即为此时最大的下标。最后更新当前index的值。

更深入理解

- **SynchronousQueue对比?**

Exchanger是一种线程间安全交换数据的机制。可以和之前分析过的SynchronousQueue对比一下：线程A通过SynchronousQueue将数据a交给线程B；线程A通过Exchanger和线程B交换数据，线程A把数据a交给线程B，同时线程B把数据b交给线程A。可见，SynchronousQueue是交给一个数据，Exchanger是交换两个数据。

- **不同JDK实现有何差别?**
  - 在JDK5中Exchanger被设计成一个容量为1的容器，存放一个等待线程，直到有另外线程到来就会发生数据交换，然后清空容器，等到下一个到来的线程。
  - 从JDK6开始，Exchanger用了类似ConcurrentMap的分段思想，提供了多个slot，增加了并发执行时的吞吐量。

*Exchanger*示例

来一个非常经典的并发问题：你有相同的数据buffer，一个或多个数据生产者，和一个或多个数据消费者。只是Exchange类只能同步2个线程，所以你只能在你的生产者和消费者问题中只有一个生产者和一个消费者时使用这个类。

```
public class Test {
    static class Producer extends Thread {
        private Exchanger<Integer> exchanger;
        private static int data = 0;
        Producer(String name, Exchanger<Integer> exchanger) {
            super("Producer-" + name);
            this.exchanger = exchanger;
        }

        @Override
        public void run() {
            for (int i=1; i<5; i++) {
                try {
                    TimeUnit.SECONDS.sleep(1);
                    data = i;
                    System.out.println(getName()+" 交换前:" + data);
                    data = exchanger.exchange(data);
                    System.out.println(getName()+" 交换后:" + data);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    static class Consumer extends Thread {
        private Exchanger<Integer> exchanger;
        private static int data = 0;
        Consumer(String name, Exchanger<Integer> exchanger) {
            super("Consumer-" + name);
            this.exchanger = exchanger;
        }

        @Override
        public void run() {
            while (true) {
                data = 0;
```

```
                System.out.println(getName()+" 交换前:" + data);
                try {
                    TimeUnit.SECONDS.sleep(1);
                    data = exchanger.exchange(data);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(getName()+" 交换后:" + data);
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Exchanger<Integer> exchanger = new Exchanger<Integer>();
        new Producer("", exchanger).start();
        new Consumer("", exchanger).start();
        TimeUnit.SECONDS.sleep(7);
        System.exit(-1);
    }
}
```

可以看到，其结果可能如下：

```
Consumer- 交换前:0
Producer- 交换前:1
Consumer- 交换后:1
Consumer- 交换前:0
Producer- 交换后:0
Producer- 交换前:2
Producer- 交换后:0
Consumer- 交换后:2
Consumer- 交换前:0
Producer- 交换前:3
Producer- 交换后:0
Consumer- 交换后:3
Consumer- 交换前:0
Producer- 交换前:4
Producer- 交换后:0
Consumer- 交换后:4
Consumer- 交换前:0
```