## JUC篇(jdk1.8)

### 原子类框架

1. **基本类型**: AtomicInteger, AtomicLong, AtomicBoolean ;  
2. **数组类型**: AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray ;  
3. **引用类型**: AtomicReference, AtomicStampedRerence, AtomicMarkableReference ;  
4. **对象的属性修改类型**: AtomicIntegerFieldUpdater, AtomicLongFieldUpdater, AtomicReferenceFieldUpdater 。

**5.累加器:** DoubleAccumulator、DoubleAdder、LongAccumulator、LongAdder、Striped64

### CAS

1. **介绍**

CAS,compare and swap比较并替换, **CAS有三个操作数：内存值V、旧的预期值A、要修改的值B，当且仅当预期值A和内存值V相同时，将内存值修改为B并返回true，否则什么都不做并返回false**。

1. **源码解析(AtomicInteger)**

public class AtomicInteger extends Number implements java.io.Serializable {  
 private static final long *serialVersionUID* = 6214790243416807050L;  
  
 // setup to use Unsafe.compareAndSwapInt for updates  
 private static final Unsafe *unsafe* = Unsafe.*getUnsafe*();  
 //表示变量值在内存中的偏移地址  
 private static final long *valueOffset*;  
  
 static {  
 try {  
 *valueOffset* = *unsafe*.objectFieldOffset  
 (java.util.concurrent.atomic.AtomicInteger.class.getDeclaredField("value"));  
 } catch (Exception ex) {  
 throw new Error(ex);  
 }  
 }  
  
 private volatile int value;  
  
 */\*\*  
 \* Creates a new AtomicInteger with the given initial value.  
 \*  
 \** ***@param*** *initialValue the initial value  
 \*/* public AtomicInteger(int initialValue) {  
 value = initialValue;  
 }  
  
 */\*\*  
 \* Creates a new AtomicInteger with initial value {****@code*** *0}.  
 \*/* public AtomicInteger() {  
 }  
 public final int incrementAndGet() {  
 return *unsafe*.getAndAddInt(this, *valueOffset*, 1) + 1;  
 }  
  
}

 Unsafe是CAS的核心类，Java无法直接访问底层操作系统，而是通过本地（native）方法来访问。不过尽管如此，JVM还是开了一个后门，JDK中有一个类Unsafe，它提供了硬件级别的**原子操作**。

 valueOffset表示的是变量值在内存中的偏移地址，因为Unsafe就是根据内存偏移地址获取数据的原值的, 偏移量可以简单理解为指针指向该变量的内存地址。

 value是用volatile修饰的，保证了多线程之间看到的value值是同一份。

**Unsafe源码(jdk1.8)**:

*/\*\*  
 \* Atomically adds the given value to the current value of a field  
 \* or array element within the given object <code>o</code>  
 \* at the given <code>offset</code>.  
 \*  
 \** ***@param*** *o object/array to update the field/element in  
 \** ***@param*** *offset field/element offset  
 \** ***@param*** *delta the value to add  
 \** ***@return*** *the previous value  
 \** ***@since*** *1.8  
 \*/*public final int getAndAddInt(Object o, long offset, int delta) {  
 int v;  
 do {  
 v = getIntVolatile(o, offset);  
 } while (!compareAndSwapInt(o, offset, v, v + delta));  
 return v;  
}

假设现在线程A和线程B同时执行getAndAdd操作：

1. AtomicInteger里面的value原始值为3，即主内存中AtomicInteger的value为3，根据Java内存模型，线程A和线程B各自持有一份value的副本，值为3。
2. 线程A通过getIntVolatile(var1, var2)方法获取到value值3，线程切换，线程A挂起。
3. 线程B通过getIntVolatile(var1, var2)方法获取到value值3，并利用compareAndSwapInt方法比较内存值也为3，比较成功，修改内存值为2，线程切换，线程B挂起。
4. 线程A恢复，利用compareAndSwapInt方法比较，发现手里的值3和内存值2不一致，**此时value正在被另外一个线程修改，线程A不能修改value值**。
5. 线程的compareAndSwapInt实现，循环判断，重新获取value值，因为value是volatile变量，所以线程对它的修改，线程A总是能够看到。线程A继续利用compareAndSwapInt进行比较并替换，直到compareAndSwapInt修改成功返回true。
6. **CAS缺点**

CAS存在一个很明显的问题，即ABA问题。  
如果变量V初次读取的时候是A，并且在准备赋值的时候检查到它仍然是A，那能说明它的值没有被其他线程修改过了吗？如果在这段期间它的值曾经被改成了B，然后又改回A，那CAS操作就会误认为它从来没有被修改过。针对这种情况，java并发包中提供了一个带有标记的原子引用类"AtomicStampedReference"，它可以通过控制变量值的版本来保证CAS的正确性。

### JUC锁

#### 框架



#### AQS ( AbstractQueuedSynchronizer )

##### 3.2.1 概述

抽象的队列式的同步器，AQS定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的ReentrantLock/Semaphore/CountDownLatch...。

说明：AbstractQueuedSynchronizer类底层的数据结构是使用双向链表，是队列的一种实现，故也可看成是队列，其中Sync queue，即同步队列，是双向链表，包括head结点和tail结点，head结点主要用作后续的调度。而Condition queue不是必须的，其是一个单向链表，只有当使用Condition时，才会存在此单向链表。并且可能会有多个Condition queue

Node源码:

static final class Node {  
 */\*\*  
 \* Marker to indicate a node is waiting in shared mode  
 \*/* static final Node *SHARED* = new Node();  
 */\*\*  
 \* Marker to indicate a node is waiting in exclusive mode  
 \*/* static final Node *EXCLUSIVE* = null;  
 //取消  
 static final int *CANCELLED* = 1;  
 //等待触发  
 static final int *SIGNAL* = -1;  
 //等待条件  
 static final int *CONDITION* = -2;  
 //状态需要向后传播  
 static final int *PROPAGATE* = -3;  
  
 volatile int waitStatus;  
 volatile Node prev;  
 volatile Node next;  
 volatile Thread thread;  
 Node nextWaiter;  
}

###### 3.2.1 等待状态

① CANCELLED，值为1，表示当前的线程被取消。

② SIGNAL，值为-1，表示当前节点的后继节点包含的线程需要运行，需要进行unpark操作。

一般发生情况是：当前线程的后继线程处于阻塞状态，而当前线程被release或cancel掉，因此需要唤醒当前线程的后继线程。

　　③ CONDITION，值为-2，表示当前节点在等待condition，也就是在condition queue中。

　　④ PROPAGATE，值为-3，(共享锁) 表示当前场景下后续的acquireShared能够得以执行。

　　⑤ 值为0，表示当前节点在sync queue中，等待着获取锁。



###### 3.2.2 数据结构





说明：AbstractQueuedSynchronizer类底层的数据结构是使用双向链表，是队列的一种实现，故也可看成是队列，其中Sync queue，即同步队列，是双向链表，包括head结点和tail结点，head结点主要用作后续的调度。而Condition queue不是必须的，其是一个单向链表，只有当使用Condition时，才会存在此单向链表。并且可能会有多个Condition queue。

AQS定义两种资源共享方式：Exclusive（独占，只有一个线程能执行，如ReentrantLock）和Share（共享，多个线程可同时执行，如Semaphore/CountDownLatch）。

不同的自定义同步器争用共享资源的方式也不同。**自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可**，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

* isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。
* tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。
* tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。
* tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
* tryReleaseShared(int)：共享方式。尝试释放资源，成功则返回true，失败则返回false。

##### 3.2.2源码详解

###### 3.2.2.1 acquire(int)

此方法是独占模式下线程获取共享资源的顶层入口。如果获取到资源，线程直接返回，否则进入等待队列，直到获取到资源为止，且整个过程忽略中断的影响,即线程在acquire过程中,中断此线程是无效的。这也正是lock()的语义，当然不仅仅只限于lock()。获取到资源后，线程就可以去执行其临界区代码了。源码：

public final void acquire(int arg) {  
 if (!tryAcquire(arg) &&  
 acquireQueued(addWaiter(Node.*EXCLUSIVE*), arg))  
 *selfInterrupt*();  
}

函数流程如下：

tryAcquire()尝试直接去获取资源，如果成功则直接返回；

addWaiter()将该线程加入等待队列的尾部，并标记为独占模式；

acquireQueued()使线程在等待队列中获取资源，一直获取到资源后才返回。如果在整个等待过程中被中断过，则返回true，否则返回false。

如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断selfInterrupt()，将中断补上。



tryAcquire()

尝试直接去获取资源，如果成功则直接返回；

protected boolean tryAcquire(int arg) {  
 throw new UnsupportedOperationException();  
}

具体资源获取/释放方式交由自定义同步器实现

ReentrantLock中公平锁和非公平锁的实现源码:

//公平锁  
protected final boolean tryAcquire(int acquires) {  
 final Thread current = Thread.*currentThread*();  
 int c = getState();  
 if (c == 0) {  
 if (!hasQueuedPredecessors() &&  
 compareAndSetState(0, acquires)) {  
 setExclusiveOwnerThread(current);  
 return true;  
 }  
 }  
 else if (current == getExclusiveOwnerThread()) {  
 int nextc = c + acquires;  
 if (nextc < 0)  
 throw new Error("Maximum lock count exceeded");  
 setState(nextc);  
 return true;  
 }  
 return false;  
}  
//非公平锁  
final boolean nonfairTryAcquire(int acquires) {  
 final Thread current = Thread.*currentThread*();  
 int c = getState();  
 if (c == 0) {  
 if (compareAndSetState(0, acquires)) {  
 setExclusiveOwnerThread(current);  
 return true;  
 }  
 }  
 else if (current == getExclusiveOwnerThread()) {  
 int nextc = c + acquires;  
 if (nextc < 0) // overflow  
 throw new Error("Maximum lock count exceeded");  
 setState(nextc);  
 return true;  
 }  
 return false;  
}

addWaiter()

将该线程加入等待队列的尾部，并标记为独占模式；

private Node addWaiter(Node mode) {  
 //以给定模式构造节点.mode有两种：EXCLUSIVE（独占）和SHARED（共享）  
 Node node = new Node(Thread.*currentThread*(), mode);  
 //尝试快速方式直接放到队尾。  
 Node pred = tail;  
 if (pred != null) {  
 node.prev = pred;  
 if (compareAndSetTail(pred, node)) {  
 pred.next = node;  
 return node;  
 }  
 }  
 //上一步失败则通过enq入队。  
 enq(node);  
 return node;  
}  
private Node enq(final Node node) {  
 //CAS"自旋"，直到成功加入队尾  
 for (;;) {  
 Node t = tail;  
 if (t == null) { // 队列为空，创建一个空的标志结点作为head结点，并将tail也指向它。  
 if (compareAndSetHead(new Node()))  
 tail = head;  
 } else {//正常流程，放入队尾  
 node.prev = t;  
 if (compareAndSetTail(t, node)) {  
 t.next = node;  
 return t;  
 }  
 }  
 }  
}

acquireQueued()

使线程在等待队列中获取资源，一直获取到资源后才返回。如果在整个等待过程中被中断过，则返回true，否则返回false。具体流程:

1.结点进入队尾后，检查状态，找到安全休息点；

2.调用park()进入waiting状态，等待unpark()或interrupt()唤醒自己；

3.被唤醒后，看自己是不是有资格能拿到号。如果拿到，head指向当前结点，并返回从入队到拿到号的整个过程中是否被中断过；如果没拿到，继续流程1。

源码:

final boolean acquireQueued(final Node node, int arg) {  
 boolean failed = true;//标记是否成功拿到资源  
 try {  
 boolean interrupted = false;//标记等待过程中是否被中断过  
 //自旋  
 for (; ; ) {  
 final Node p = node.predecessor();//拿到前节点  
 //如果前节点是head，即该结点已成next，那么便有资格去尝试获取资源（可能是head释放完资源唤醒自己的，当然也可能被interrupt了）。  
 if (p == head && tryAcquire(arg)) {  
 setHead(node);//拿到资源后，将head指向该结点。所以head所指的标杆结点，就是当前获取到资源的那个结点或null。  
 p.next = null; // help GC  
 failed = false;  
 return interrupted;//返回等待过程中是否被中断过  
 }  
 //如果自己可以休息了，就进入waiting状态，直到被unpark()  
 if (*shouldParkAfterFailedAcquire*(p, node) &&  
 parkAndCheckInterrupt())  
 interrupted = true;//如果等待过程中被中断过，就将interrupted标记为true  
 }  
 } finally {  
 if (failed)  
 cancelAcquire(node);  
 }  
}  
  
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {  
 int ws = pred.waitStatus;//拿到前节点的状态  
 if (ws == Node.*SIGNAL*)  
 //如果已经告诉前节点拿完号后通知自己一下，那就可以安心休息了  
 return true;  
 if (ws > 0) {  
 */\*\*  
 \* 如果前节点放弃了，那就一直往前找，直到找到最近一个正常等待的状态，并排在它的后边。  
 \* 注意：那些放弃的结点，由于被自己“加塞”到它们前边，它们相当于形成一个无引用链，稍后就会被GC回收  
 \*/* do {  
 node.prev = pred = pred.prev;  
 } while (pred.waitStatus > 0);  
 pred.next = node;  
 } else {  
 //如果前节点正常，那就把前节点的状态设置成SIGNAL，告诉它拿完号后通知自己一下。有可能失败，人家说不定刚刚释放完呢  
 compareAndSetWaitStatus(pred, ws, Node.*SIGNAL*);  
 }  
 return false;  
}  
*/\*\*如果线程找好安全休息点后，那就可以安心去休息了。此方法就是让线程去休息，真正进入等待状态。\*/*private final boolean parkAndCheckInterrupt() {  
 LockSupport.*park*(this);//使线程进入waiting状态  
 return Thread.*interrupted*();//如果被唤醒，查看自己是不是被中断的。  
}

park()会让当前线程进入waiting状态。在此状态下，有两种途径可以唤醒该线程：

1）被unpark()；“前继节点对应的线程”使用完锁之后，通过unpark()方式唤醒当前线程。

2）被interrupt()。需要注意的是，Thread.interrupted()会清除当前线程的中断标记位。

如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断selfInterrupt()，将中断补上。

###### 3.2.2.2 release(int)

此方法是独占模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了（即state=0）,它会唤醒等待队列里的其他线程来获取资源。这也正是unlock()的语义，当然不仅仅只限于unlock()。下面是release()的源码：

public final boolean release(int arg) {  
 if (tryRelease(arg)) {  
 Node h = head;//找到头节点  
 if (h != null && h.waitStatus != 0)  
 unparkSuccessor(h);//唤醒等待队列的下一个线程  
 return true;  
 }  
 return false;  
}

tryRelease(int)

此方法尝试去释放指定量的资源。下面是tryRelease()的源码：

protected boolean tryRelease(int arg) {  
 throw new UnsupportedOperationException();  
}

跟tryAcquire()一样，这个方法是需要独占模式的自定义同步器去实现的。正常来说，tryRelease()都会成功的，因为这是独占模式，该线程来释放资源，那么它肯定已经拿到独占资源了，直接减掉相应量的资源即可(state-=arg)，也不需要考虑线程安全的问题。但要注意它的返回值，**release()是根据tryRelease()的返回值来判断该线程是否已经完成释放掉资源了！**所以自义定同步器在实现时，如果已经彻底释放资源(state=0)，要返回true，否则返回false

unparkSuccessor(Node)

说明: 唤醒等待队列里的下一个线程

private void unparkSuccessor(Node node) {  
 //这里，node一般为当前线程所在的结点。  
 int ws = node.waitStatus;  
 if (ws < 0)//置零当前线程所在的结点状态，允许失败。  
 compareAndSetWaitStatus(node, ws, 0);  
  
 Node s = node.next;//找到下一个需要唤醒的结点  
 if (s == null || s.waitStatus > 0) {  
 s = null;  
 for (Node t = tail; t != null && t != node; t = t.prev)  
 if (t.waitStatus <= 0)  
 s = t;  
 }  
 if (s != null)  
 LockSupport.*unpark*(s.thread);  
}

小结

release()是独占模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了（即state=0）,它会唤醒等待队列里的其他线程来获取资源。

###### 3.2.2.3 acquireShared(int)

此方法是共享模式下线程获取共享资源的顶层入口。它会获取指定量的资源，获取成功则直接返回，获取失败则进入等待队列，直到获取到资源为止，整个过程忽略中断。源码：

public final void acquireShared(int arg) {  
 if (tryAcquireShared(arg) < 0)  
 doAcquireShared(arg);  
}

这里tryAcquireShared()依然需要自定义同步器去实现。但是AQS已经把其返回值的语义定义好了：负值代表获取失败；0代表获取成功，但没有剩余资源；正数表示获取成功，还有剩余资源，其他线程还可以去获取。所以这里acquireShared()的流程就是：

* + 1. tryAcquireShared()尝试获取资源，成功则直接返回；
    2. 失败则通过doAcquireShared()进入等待队列park()，直到被unpark()/interrupt并成功获取到资源为止才返回。整个等待过程也是忽略中断的

tryAcquiredShared(int)

tryAcquireShared()在ReentrantReadWriteLock中实现源码(详细解析见[ReadWriteLock篇 tryAcquireShared(int)](#_tryAcquireShared(int))):

protected final int tryAcquireShared(int unused) {  
   
 Thread current = Thread.*currentThread*();  
 int c = getState();  
 if (exclusiveCount(c) != 0 &&  
 getExclusiveOwnerThread() != current)  
 return -1;  
 int r = sharedCount(c);  
 if (!readerShouldBlock() &&  
 r < MAX\_COUNT &&  
 compareAndSetState(c, c + SHARED\_UNIT)) {  
 if (r == 0) {  
 firstReader = current;  
 firstReaderHoldCount = 1;  
 } else if (firstReader == current) {  
 firstReaderHoldCount++;  
 } else {  
 HoldCounter rh = cachedHoldCounter;  
 if (rh == null || rh.tid != getThreadId(current))  
 cachedHoldCounter = rh = readHolds.get();  
 else if (rh.count == 0)  
 readHolds.set(rh);  
 rh.count++;  
 }  
 return 1;  
 }  
 return fullTryAcquireShared(current);  
}

doAcquireShared(int)

此方法用于将当前线程加入等待队列尾部休息，直到其他线程释放资源唤醒自己，自己成功拿到相应量的资源后才返回。源码：

private void doAcquireShared(int arg) {  
 final Node node = addWaiter(Node.*SHARED*);//加入队列尾部  
 boolean failed = true;//是否成功标记  
 try {  
 boolean interrupted = false;//等待过程中是否被中断过标记  
 for (;;) {  
 final Node p = node.predecessor();//前节点  
 if (p == head) {//如果前节点为head,此时唤醒node  
 int r = tryAcquireShared(arg);//尝试获取资源  
 if (r >= 0) {  
 setHeadAndPropagate(node, r);//将head指向自己,如果还有剩余资源可以再唤醒之后的线程  
 p.next = null; // help GC  
 if (interrupted)//如果等待过程中被中断,此时将中断标记补上  
 selfInterrupt();  
 failed = false;  
 return;  
 }  
 }  
 //判断状态,寻找安全点,进入waiting状态,等待被unpark()或interrupt()  
 if (*shouldParkAfterFailedAcquire*(p, node) &&  
 parkAndCheckInterrupt())  
 interrupted = true;  
 }  
 } finally {  
 if (failed)  
 cancelAcquire(node);  
 }  
}

setHeadAndPropagate(Node,int)

成功获取资源后,如果还有剩余资源,继续唤醒下一个线程

private void setHeadAndPropagate(Node node, int propagate) {  
 Node h = head; // Record old head for check below  
 setHead(node);//head指向自己  
 //如果还有剩余资源,继续唤醒下一个线程  
 if (propagate > 0 || h == null || h.waitStatus < 0 ||  
 (h = head) == null || h.waitStatus < 0) {  
 Node s = node.next;  
 if (s == null || s.isShared())  
 doReleaseShared();  
 }  
}

###### 3.2.2.4 releaseShared(int)

此方法是共享模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了（即state=0）,它会唤醒等待队列里的其他线程来获取资源。源码:

public final boolean releaseShared(int arg) {  
 if (tryReleaseShared(arg)) {  
 doReleaseShared();//唤醒后续节点  
 return true;  
 }  
 return false;  
}

释放掉资源后，唤醒后继线程 .跟独占模式下的release()相似，但有一点稍微需要注意：独占模式下的tryRelease()在完全释放掉资源（state=0）后，才会返回true去唤醒其他线程，这主要是基于可重入的考量；而共享模式下的releaseShared()则没有这种要求，一是共享的实质--多线程可并发执行；二是共享模式基本也不会重入，所以自定义同步器可以根据需要决定返回值。

tryReleaseShared(int)

tryReleaseShared()在ReentrantReadWriteLock中的实现[见ReadWriteLock篇 tryReleaseShared(int)](#_tryReleaseShared(int)):

protected final boolean tryReleaseShared(int unused) {  
 Thread current = Thread.*currentThread*();  
 if (firstReader == current) {  
 // assert firstReaderHoldCount > 0;  
 if (firstReaderHoldCount == 1)  
 firstReader = null;  
 else  
 firstReaderHoldCount--;  
 } else {  
 HoldCounter rh = cachedHoldCounter;  
 if (rh == null || rh.tid != *getThreadId*(current))  
 rh = readHolds.get();  
 int count = rh.count;  
 if (count <= 1) {  
 readHolds.remove();  
 if (count <= 0)  
 throw unmatchedUnlockException();  
 }  
 --rh.count;  
 }  
 for (;;) {  
 int c = getState();  
 int nextc = c - *SHARED\_UNIT*;  
 if (compareAndSetState(c, nextc))  
 // Releasing the read lock has no effect on readers,  
 // but it may allow waiting writers to proceed if  
 // both read and write locks are now free.  
 return nextc == 0;  
 }  
}

doReleaseShared(int)

主要用于唤醒下一个线程,源码:

private void doReleaseShared() {  
 //自旋  
 for (;;) {  
 Node h = head;  
 if (h != null && h != tail) {  
 int ws = h.waitStatus;  
 if (ws == Node.*SIGNAL*) {  
 if (!compareAndSetWaitStatus(h, Node.*SIGNAL*, 0))  
 continue; // loop to recheck cases  
 unparkSuccessor(h);//唤醒后续线程  
 }  
 else if (ws == 0 &&  
 !compareAndSetWaitStatus(h, 0, Node.*PROPAGATE*))  
 continue; // loop on failed CAS  
 }  
 if (h == head) // loop if head changed  
 break;  
 }  
}

###### 3.2.2.5 其他方法

//独占式超时获取锁  
public final boolean tryAcquireNanos(int arg, long nanosTimeout)  
//共享式超时获取锁   
public final boolean tryAcquireSharedNanos(int arg, long nanosTimeout)

如果未在指定时间内获取锁,返回false

##### 3.2.3 小结

acquire()和acquireSahred()两种方法下，线程在等待队列中都是忽略中断的。AQS也支持响应中断的，acquireInterruptibly()/acquireSharedInterruptibly()即是，这里相应的源码跟acquire()和acquireSahred()差不多，这里就不再详解了。

##### 3.2.4 应用

不同的自定义同步器争用共享资源的方式也不同。**自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可**，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

* isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。
* tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。
* tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。
* tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
* tryReleaseShared(int)：共享方式。尝试释放资源，成功则返回true，失败则返回false。
* **在AQS中维护着一个FIFO的同步队列，当线程获取同步状态失败后，则会加入到这个CLH同步队列的对尾并一直保持着自旋。在CLH同步队列中的线程在自旋时会判断其前驱节点是否为首节点，如果为首节点则不断尝试获取同步状态，获取成功则退出CLH同步队列。当线程执行完逻辑后，会释放同步状态，释放后会唤醒其后继节点。**

#### ReetrantLock

##### 概述

ReentrantLock是一个可重入的互斥锁，又被称为“独占锁”。

顾名思义，ReentrantLock锁在同一个时间点只能被一个线程锁持有；而可重入的意思是，ReentrantLock锁，可以被单个线程多次获取。  
ReentrantLock分为“**公平锁**”和“**非公平锁**”。它们的区别体现在获取锁的机制上是否公平。“锁”是为了保护竞争资源，防止多个线程同时操作线程而出错，ReentrantLock在同一个时间点只能被一个线程获取(当某线程获取到“锁”时，其它线程就必须等待)；ReentraantLock是通过一个FIFO的等待队列来管理获取该锁所有线程的。在“公平锁”的机制下，线程依次排队获取锁；而“非公平锁”在锁是可获取状态时，不管自己是不是在队列的开头都会获取锁。

##### 继承关系及内部类

ReentrantLock实现了Lock接口，Lock接口中定义了lock与unlock相关操作，并且还存在newCondition方法，表示生成一个条件。源码:

public class ReentrantLock extends AbstractQueuedSynchronizer implements Lock, java.io.Serializable



##### Lock.lock()

###### 3.3.3.1 公平锁和非公平锁

公平锁 -- 公平锁的lock()函数，会直接调用acquire(1)。

非公平锁 -- 非公平锁会先判断当前锁的状态是不是空闲，是的话，就不排队，而是直接获取锁。

源码:

//非公平锁获取  
final void lock() {  
 if (compareAndSetState(0, 1))  
 setExclusiveOwnerThread(Thread.*currentThread*());  
 else  
 acquire(1);  
}  
//公平锁获取  
final void lock() {  
 acquire(1);  
}

acquire()源码:

public final void acquire(int arg) {  
 if (!tryAcquire(arg) &&  
 acquireQueued(addWaiter(Node.*EXCLUSIVE*), arg))  
 selfInterrupt();  
}

tryAcquire()源码对比:

//公平锁  
protected final boolean tryAcquire(int acquires) {  
 final Thread current = Thread.*currentThread*();  
 int c = getState();  
 if (c == 0) {  
 if (!hasQueuedPredecessors() &&  
 compareAndSetState(0, acquires)) {  
 setExclusiveOwnerThread(current);  
 return true;  
 }  
 }  
 else if (current == getExclusiveOwnerThread()) {  
 int nextc = c + acquires;  
 if (nextc < 0)  
 throw new Error("Maximum lock count exceeded");  
 setState(nextc);  
 return true;  
 }  
 return false;  
}  
//非公平锁  
final boolean nonfairTryAcquire(int acquires) {  
 final Thread current = Thread.*currentThread*();  
 int c = getState();  
 if (c == 0) {  
 if (compareAndSetState(0, acquires)) {  
 setExclusiveOwnerThread(current);  
 return true;  
 }  
 }  
 else if (current == getExclusiveOwnerThread()) {  
 int nextc = c + acquires;  
 if (nextc < 0) // overflow  
 throw new Error("Maximum lock count exceeded");  
 setState(nextc);  
 return true;  
 }  
 return false;  
}

###### 3.3.3.2“公平锁”和“非公平锁”关于acquire()的对比

公平锁和非公平锁，它们尝试获取锁的方式不同。

公平锁在尝试获取锁时，即使“锁”没有被任何线程锁持有，它也会判断自己是不是CLH等待队列的表头；是的话，才获取锁。

而非公平锁在尝试获取锁时，如果“锁”没有被任何线程持有，则不管它在CLH队列的何处，它都直接获取锁。



##### Lock.unlock()

public void unlock() {  
 sync.release(1);  
}

protected final boolean tryRelease(int releases) {  
 int c = getState() - releases;  
 if (Thread.*currentThread*() != getExclusiveOwnerThread())  
 throw new IllegalMonitorStateException();  
 boolean free = false;  
 if (c == 0) {  
 free = true;  
 setExclusiveOwnerThread(null);  
 }  
 setState(c);  
 return free;  
}

[见3.2.2.2 release(int)](#_3.2.2.2_release(int))

##### Condition条件

###### 概述

Condition的作用是对锁进行更精确的控制。Condition中的await()方法相当于Object的wait()方法，Condition中的signal()方法相当于Object的notify()方法，Condition中的signalAll()相当于Object的notifyAll()方法。不同的是，Object中的wait(),notify(),notifyAll()方法是和"同步锁"(synchronized关键字)捆绑使用的；而Condition是需要与"互斥锁"/"共享锁"捆绑使用的。

###### 函数列表

// 造成当前线程在接到信号或被中断之前一直处于等待状态。  
void await()  
// 造成当前线程在接到信号、被中断或到达指定等待时间之前一直处于等待状态。  
boolean await(long time, TimeUnit unit)  
// 造成当前线程在接到信号、被中断或到达指定等待时间之前一直处于等待状态。  
long awaitNanos(long nanosTimeout)  
// 造成当前线程在接到信号之前一直处于等待状态。  
void awaitUninterruptibly()  
// 造成当前线程在接到信号、被中断或到达指定最后期限之前一直处于等待状态。  
boolean awaitUntil(Date deadline)  
// 唤醒一个等待线程。  
void signal()  
// 唤醒所有等待线程。  
void signalAll()

#### LockSupport

##### 概述

* + - 1. LockSupport是用来创建锁和其他同步类的基本线程阻塞原语。
      2. LockSupport中的park() 和 unpark() 的作用分别是阻塞线程和解除阻塞线程，而且park()和unpark()不会遇到“Thread.suspend 和 Thread.resume所可能引发的死锁”问题。  
         因为park() 和 unpark()有许可的存在；调用 park() 的线程和另一个试图将其 unpark() 的线程之间的竞争将保持活性。
      3. park和wait的区别。wait让线程阻塞前，必须通过synchronized获取同步锁。

##### 源码解析

// 返回提供给最近一次尚未解除阻塞的 park 方法调用的 blocker 对象，如果该调用不受阻塞，则返回 null。  
static Object getBlocker(Thread t)  
// 为了线程调度，禁用当前线程，除非许可可用。  
static void park()  
// 为了线程调度，在许可可用之前禁用当前线程。  
static void park(Object blocker)  
// 为了线程调度禁用当前线程，最多等待指定的等待时间，除非许可可用。  
static void parkNanos(long nanos)  
// 为了线程调度，在许可可用前禁用当前线程，并最多等待指定的等待时间。  
static void parkNanos(Object blocker, long nanos)  
// 为了线程调度，在指定的时限前禁用当前线程，除非许可可用。  
static void parkUntil(long deadline)  
// 为了线程调度，在指定的时限前禁用当前线程，除非许可可用。  
static void parkUntil(Object blocker, long deadline)  
// 如果给定线程的许可尚不可用，则使其可用。  
static void unpark(Thread thread)

说明:所有方法均是调用UNSAFE类的native接口实现

#### ReadWriteLock

##### 概述

Java的JUC(java.util.concurrent)包中的锁包括"独占锁(ReentrantLock)"和"共享锁"。JUC中的共享锁有CountDownLatch, CyclicBarrier, Semaphore, ReentrantReadWriteLock,JDK1.8新增的StampedLock等

ReadWriteLock，顾名思义，是读写锁。它维护了一对相关的锁 — — “读取锁”和“写入锁”，一个用于读取操作，另一个用于写入操作。  
“读取锁”用于只读操作，它是“共享锁”，能同时被多个线程获取。  
“写入锁”用于写入操作，它是“独占锁”，写入锁只能被一个线程锁获取。  
注意：不能同时存在读取锁和写入锁,持有写入锁的线程可以继续获取读取锁,称为锁降级！  
ReadWriteLock是一个接口。ReentrantReadWriteLock是它的实现类，ReentrantReadWriteLock包括子类ReadLock和WriteLock。

源码:

public interface ReadWriteLock {  
 */\*\*  
 \* 读锁,共享锁  
 \*/* Lock readLock();  
  
 */\*\*  
 \* 写锁,独占锁  
 \*/* Lock writeLock();  
}

###### 数据结构



从中可以看出：

1. ReentrantReadWriteLock实现了ReadWriteLock接口。ReadWriteLock是一个读写锁的接口，提供了"获取读锁的readLock()函数" 和 "获取写锁的writeLock()函数"。
2. ReentrantReadWriteLock中包含：sync对象，读锁readerLock和写锁writerLock。读锁ReadLock和写锁WriteLock都实现了Lock接口。读锁ReadLock和写锁WriteLock中也都分别包含了"Sync对象"，它们的Sync对象和ReentrantReadWriteLock的Sync对象 是一样的，就是通过sync，读锁和写锁实现了对同一个对象的访问。
3. 和"ReentrantLock"一样，sync是Sync类型；而且，Sync也是一个继承于AQS的抽象类。Sync也包括"公平锁"FairSync和"非公平锁"NonfairSync。sync对象是"FairSync"和"NonfairSync"中的一个，默认是"NonfairSync"。

Sync

abstract static class Sync extends AbstractQueuedSynchronizer {  
 private static final long *serialVersionUID* = 6317671515068378041L;  
 // 最多支持65535个写锁和65535个读锁；低16位表示写锁计数，高16位表示持有读锁的线程数  
 static final int *SHARED\_SHIFT* = 16;  
 // 由于读锁用高位部分，读锁个数加1，其实是状态值加 2^16  
 static final int *SHARED\_UNIT* = (1 << *SHARED\_SHIFT*);  
 // 读锁最大数量  
 static final int *MAX\_COUNT* = (1 << *SHARED\_SHIFT*) - 1;  
 // 写锁的掩码，用于状态的低16位有效值  
 static final int *EXCLUSIVE\_MASK* = (1 << *SHARED\_SHIFT*) - 1;  
 */\*\* 读锁计数，当前持有读锁的线程数，c的高16位 \*/* static int sharedCount(int c) { return c >>> *SHARED\_SHIFT*; }  
 */\*\* 写锁的计数，也就是它的重入次数,c的低16位\*/* static int exclusiveCount(int c) { return c & *EXCLUSIVE\_MASK*; }  
  
 //表示占有读锁的线程数量  
 static int sharedCount(int c) { return c >>> *SHARED\_SHIFT*; }  
 //表示占有写锁的线程数量  
 static int exclusiveCount(int c) { return c & *EXCLUSIVE\_MASK*; }  
 */\*\*  
 \* 当前线程持有的可重入读锁的数量，仅在构造方法和readObject(反序列化)  
 \* 时被初始化，当持有锁的数量为0时，移除此对象。  
 \*/* private transient ThreadLocalHoldCounter readHolds;  
 */\*\*  
 \* 最近一个成功获取读锁的线程的计数。这省却了ThreadLocal查找，  
 \* 通常情况下，下一个释放线程是最后一个获取线程。这不是 volatile 的，  
 \* 因为它仅用于试探的，线程进行缓存也是可以的  
 \* （因为判断是否是当前线程是通过线程id来比较的）。  
 \*/* private transient HoldCounter cachedHoldCounter;  
 // 第一个读线程  
 private transient Thread firstReader = null;  
 // 第一个读线程的计数  
 private transient int firstReaderHoldCount;  
}  
// 构造函数  
Sync() {  
 // 本地线程计数器  
 readHolds = new ThreadLocalHoldCounter();  
 // 设置AQS的状态  
 setState(getState()); // ensures visibility of readHolds  
}

Sync类内部存在两个内部类，分别为HoldCounter和ThreadLocalHoldCounter，其中HoldCounter主要与读锁配套使用

HoldCounter

// 计数器  
static final class HoldCounter {  
 // 计数  
 int count = 0;  
 // Use id, not reference, to avoid garbage retention  
 // 获取当前线程的TID属性的值  
 final long tid = getThreadId(Thread.*currentThread*());  
}

说明：HoldCounter主要有两个属性，count和tid，其中count表示某个读线程重入的次数，tid表示该线程的tid字段的值，该字段可以用来唯一标识一个线程。

ThreadLocalHoldCounter

// 本地线程计数器  
static final class ThreadLocalHoldCounter  
 extends ThreadLocal<HoldCounter> {  
 // 重写初始化方法，在没有进行set的情况下，获取的都是该HoldCounter值  
 public HoldCounter initialValue() {  
 return new HoldCounter();  
 }  
}

说明：ThreadLocalHoldCounter重写了ThreadLocal的initialValue方法，ThreadLocal类可以将线程与对象相关联。在没有进行set的情况下，get到的均是initialValue方法里面生成的那个HolderCounter对象。

###### 函数列表

// 创建一个新的 ReentrantReadWriteLock，默认是采用“非公平策略”。  
ReentrantReadWriteLock()  
// 创建一个新的 ReentrantReadWriteLock，fair是“公平策略”。fair为true，意味着公平策略；否则，意味着非公平策略。  
ReentrantReadWriteLock(boolean fair)  
  
// 返回当前拥有写入锁的线程，如果没有这样的线程，则返回 null。  
protected Thread getOwner()  
// 返回一个 collection，它包含可能正在等待获取读取锁的线程。  
protected Collection<Thread> getQueuedReaderThreads()  
// 返回一个 collection，它包含可能正在等待获取读取或写入锁的线程。  
protected Collection<Thread> getQueuedThreads()  
// 返回一个 collection，它包含可能正在等待获取写入锁的线程。  
protected Collection<Thread> getQueuedWriterThreads()  
// 返回等待获取读取或写入锁的线程估计数目。  
int getQueueLength()  
// 查询当前线程在此锁上保持的重入读取锁数量。  
int getReadHoldCount()  
// 查询为此锁保持的读取锁数量。  
int getReadLockCount()  
// 返回一个 collection，它包含可能正在等待与写入锁相关的给定条件的那些线程。  
protected Collection<Thread> getWaitingThreads(Condition condition)  
// 返回正等待与写入锁相关的给定条件的线程估计数目。  
int getWaitQueueLength(Condition condition)  
// 查询当前线程在此锁上保持的重入写入锁数量。  
int getWriteHoldCount()  
// 查询是否给定线程正在等待获取读取或写入锁。  
boolean hasQueuedThread(Thread thread)  
// 查询是否所有的线程正在等待获取读取或写入锁。  
boolean hasQueuedThreads()  
// 查询是否有些线程正在等待与写入锁有关的给定条件。  
boolean hasWaiters(Condition condition)  
// 如果此锁将公平性设置为 ture，则返回 true。  
boolean isFair()  
// 查询是否某个线程保持了写入锁。  
boolean isWriteLocked()  
// 查询当前线程是否保持了写入锁。  
boolean isWriteLockedByCurrentThread()  
// 返回用于读取操作的锁。  
ReentrantReadWriteLock.ReadLock readLock()  
// 返回用于写入操作的锁。  
ReentrantReadWriteLock.WriteLock writeLock()

##### 3.5.2 ReadLock

###### 3.5.2.1 源码

public static class ReadLock implements Lock, java.io.Serializable {  
 private static final long *serialVersionUID* = -5992448646407690164L;  
 //持有的AQS对象  
 private final Sync sync;  
  
 protected ReadLock(ReentrantReadWriteLock lock) {  
 sync = lock.sync;  
 }  
  
 //获取共享锁  
 public void lock() {  
 sync.acquireShared(1);  
 }  
  
 //获取共享锁(响应中断)  
 public void lockInterruptibly() throws InterruptedException {  
 sync.acquireSharedInterruptibly(1);  
 }  
  
 //尝试获取共享锁  
 public boolean tryLock(long timeout, TimeUnit unit)  
 throws InterruptedException {  
 return sync.tryAcquireSharedNanos(1, unit.toNanos(timeout));  
 }  
  
 //释放锁  
 public void unlock() {  
 sync.releaseShared(1);  
 }  
  
 //新建条件  
 public Condition newCondition() {  
 throw new UnsupportedOperationException();  
 }  
  
 public String toString() {  
 int r = sync.getReadLockCount();  
 return super.toString() +  
 "[Read locks = " + r + "]";  
 }  
}

###### 3.5.2.2 lock()

public void lock() {  
 sync.acquireShared(1);  
}

acquireShared(int)

[见AQS篇 acquireShared(int)](#_3.2.2.3_acquireShared(int))

tryAcquireShared(int)

protected final int tryAcquireShared(int unused) {  
 //获取当前线程  
 Thread current = Thread.*currentThread*();  
 int c = getState();  
 //持有写锁的线程可以获取读锁，如果获取锁的线程不是current线程；则返回-1。  
 if (exclusiveCount(c) != 0 &&  
 getExclusiveOwnerThread() != current)  
 return -1;  
 //获取共享计数  
 int r = sharedCount(c);  
 // 如果“不需要阻塞”，并且“读取锁”的共享计数小于MAX\_COUNT  
 // 则通过CAS函数更新“锁的状态”，将“读取锁”的共享计数+1  
 if (!readerShouldBlock() &&  
 r < MAX\_COUNT &&  
 compareAndSetState(c, c + SHARED\_UNIT)) {  
 // 第1次获取“共享锁”  
 if (r == 0) {  
 firstReader = current;  
 firstReaderHoldCount = 1;  
 //如果想要获取锁的线程(current)是第1个获取锁(firstReader)的线程  
 } else if (firstReader == current) {  
 firstReaderHoldCount++;  
 } else {  
 // HoldCounter用来统计该线程获取“读取锁”的次数。  
 HoldCounter rh = cachedHoldCounter;  
 if (rh == null || rh.tid != getThreadId(current))  
 cachedHoldCounter = rh = readHolds.get();  
 else if (rh.count == 0)  
 readHolds.set(rh);  
 // 将该线程获取“读取锁”的次数+1。  
 rh.count++;  
 }  
 return 1;  
 }  
 return fullTryAcquireShared(current);  
}

**说明**：tryAcquireShared()的作用是尝试获取“共享锁”。  
如果在尝试获取锁时，“不需要阻塞等待”并且“读取锁的共享计数小于MAX\_COUNT”，则直接通过CAS函数更新“读取锁的共享计数”，以及将“当前线程获取读取锁的次数+1”。  
否则，通过fullTryAcquireShared()获取读取锁。

fullTryAcquireShared(Thread current)

final int fullTryAcquireShared(Thread current) {  
 HoldCounter rh = null;  
 for (;;) {  
 // 获取“锁”的状态  
 int c = getState();  
 // 如果“锁”是“互斥锁”，并且获取锁的线程不是current线程；则返回-1。  
 if (exclusiveCount(c) != 0) {  
 if (getExclusiveOwnerThread() != current)  
 return -1;  
 // 如果“需要阻塞等待”。  
 // (01) 当“需要阻塞等待”的线程是第1个获取锁的线程的话，则继续往下执行。  
 // (02) 当“需要阻塞等待”的线程获取锁的次数=0时，则返回-1。  
 } else if (readerShouldBlock()) {  
 // 如果想要获取锁的线程(current)是第1个获取锁(firstReader)的线程  
 if (firstReader == current) {  
 } else {  
 if (rh == null) {  
 rh = cachedHoldCounter;  
 if (rh == null || rh.tid != current.getId()) {  
 rh = readHolds.get();  
 if (rh.count == 0)  
 readHolds.remove();  
 }  
 }  
 // 如果当前线程获取锁的计数=0,则返回-1。  
 if (rh.count == 0)  
 return -1;  
 }  
 }  
 // 如果“不需要阻塞等待”，则获取“读取锁”的共享统计数；  
 // 如果共享统计数超过MAX\_COUNT，则抛出异常。  
 if (sharedCount(c) == MAX\_COUNT)  
 throw new Error("Maximum lock count exceeded");  
 // 将线程获取“读取锁”的次数+1。  
 if (compareAndSetState(c, c + SHARED\_UNIT)) {  
 // 如果是第1次获取“读取锁”，则更新firstReader和firstReaderHoldCount。  
 if (sharedCount(c) == 0) {  
 firstReader = current;  
 firstReaderHoldCount = 1;  
 // 如果想要获取锁的线程(current)是第1个获取锁(firstReader)的线程，  
 // 则将firstReaderHoldCount+1。  
 } else if (firstReader == current) {  
 firstReaderHoldCount++;  
 } else {  
 if (rh == null)  
 rh = cachedHoldCounter;  
 if (rh == null || rh.tid != current.getId())  
 rh = readHolds.get();  
 else if (rh.count == 0)  
 readHolds.set(rh);  
 // 更新线程的获取“读取锁”的共享计数  
 rh.count++;  
 cachedHoldCounter = rh; // cache for release  
 }  
 return 1;  
 }  
 }  
}

**说明**：fullTryAcquireShared()会根据“是否需要阻塞等待”，“读取锁的共享计数是否超过限制”等等进行处理。如果不需要阻塞等待，并且锁的共享计数没有超过限制，则通过CAS尝试获取锁，并返回1。

doAcquireShared(int)

[见AQS篇 doAcquireShared(int)](#_doAcquireShared(int))

###### 3.5.2.3 unlock()

public void unlock() {  
 sync.releaseShared(1);  
}

releaseShared(int)

public final boolean releaseShared(int arg) {  
 if (tryReleaseShared(arg)) {  
 doReleaseShared();//唤醒后续节点  
 return true;  
 }  
 return false;  
}

[见AQS篇 releaseShared(int)](#_3.2.2.4_releaseShared(int))

tryReleaseShared(int)

protected final boolean tryReleaseShared(int unused) {  
 // 获取当前线程，即释放共享锁的线程。  
 Thread current = Thread.*currentThread*();  
 // 如果想要释放锁的线程(current)是第1个获取锁(firstReader)的线程，  
 // 并且“第1个获取锁的线程获取锁的次数”=1，则设置firstReader为null；  
 // 否则，将“第1个获取锁的线程的获取次数”-1。  
 if (firstReader == current) {  
 // assert firstReaderHoldCount > 0;  
 if (firstReaderHoldCount == 1)  
 firstReader = null;  
 else  
 firstReaderHoldCount--;  
 // 获取rh对象，并更新“当前线程获取锁的信息”。  
 } else {  
  
 HoldCounter rh = cachedHoldCounter;  
 if (rh == null || rh.tid != current.getId())  
 rh = readHolds.get();  
 int count = rh.count;  
 if (count <= 1) {  
 readHolds.remove();  
 if (count <= 0)  
 throw unmatchedUnlockException();  
 }  
 --rh.count;  
 }  
 for (;;) {  
 // 获取锁的状态  
 int c = getState();  
 // 将锁的获取次数-1。  
 int nextc = c - SHARED\_UNIT;  
 // 通过CAS更新锁的状态。  
 if (compareAndSetState(c, nextc))  
 return nextc == 0;  
 }  
}

**说明**：tryReleaseShared()的作用是尝试释放共享锁。

doReleaseShared()

[见AQS篇 doReleaseShared(int)](#_doReleaseShared(int))

#### CountDownLatch

##### 概述

CountDownLatch是一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。

**CountDownLatch和CyclicBarrier的区别**  
(01) CountDownLatch的作用是允许1或N个线程等待其他线程完成执行；而CyclicBarrier则是允许N个线程相互等待。  
(02) CountDownLatch的计数器无法被重置；CyclicBarrier的计数器可以被重置后使用，因此它被称为是循环的barrier。

###### 函数列表

CountDownLatch(int count)  
构造一个用给定计数初始化的 CountDownLatch。  
  
// 使当前线程在锁存器倒计数至零之前一直等待，除非线程被中断。  
void await()  
// 使当前线程在锁存器倒计数至零之前一直等待，除非线程被中断或超出了指定的等待时间。  
boolean await(long timeout, TimeUnit unit)  
// 递减锁存器的计数，如果计数到达零，则释放所有等待的线程。  
void countDown()  
// 返回当前计数。  
long getCount()

###### 数据结构



##### 源码详解

###### CountDownLatch(int)

public CountDownLatch(int count) {  
 if (count < 0) throw new IllegalArgumentException("count < 0");  
 this.sync = new Sync(count);  
}

**说明**：该函数是创建一个Sync对象，而Sync是继承于AQS类。Sync构造函数如下：

Sync(int count) {  
 setState(count);  
}

protected final void setState(int newState) {  
 state = newState;  
}

**说明**：在AQS中，state是一个private volatile long类型的对象。对于CountDownLatch而言，state表示的”锁计数器“。CountDownLatch中的getCount()最终是调用AQS中的getState()，返回的state对象，即”锁计数器“。

###### await()

使当前线程在锁存器倒计数至零之前一直等待，除非线程被中断或超出了指定的等待时间

public void await() throws InterruptedException {  
 sync.acquireSharedInterruptibly(1);  
}

**说明**：该函数实际上是调用的AQS的acquireSharedInterruptibly(1);

acquireSharedInterruptibly(int)

public final void acquireSharedInterruptibly(int arg)  
 throws InterruptedException {  
 if (Thread.*interrupted*())  
 throw new InterruptedException();  
 if (tryAcquireShared(arg) < 0)  
 doAcquireSharedInterruptibly(arg);  
}

**说明**：acquireSharedInterruptibly()的作用是获取共享锁。  
如果当前线程是中断状态，则抛出异常InterruptedException。否则，调用tryAcquireShared(arg)尝试获取共享锁；尝试成功则返回，否则就调用doAcquireSharedInterruptibly()。doAcquireSharedInterruptibly()会使当前线程一直等待，直到当前线程获取到共享锁(或被中断)才返回。

tryAcquireShared()

tryAcquireShared()在CountDownLatch.java中被重写，它的源码如下：

protected int tryAcquireShared(int acquires) {  
 return (getState() == 0) ? 1 : -1;  
}

**说明**：tryAcquireShared()的作用是尝试获取共享锁。  
如果"锁计数器=0"，即锁是可获取状态，则返回1；否则，锁是不可获取状态，则返回-1。

doAcquireSharedInterruptibly(int)

private void doAcquireSharedInterruptibly(long arg)  
 throws InterruptedException {  
 // 创建"当前线程"的Node节点，且Node中记录的锁是"共享锁"类型；并将该节点添加到CLH队列末尾。  
 final Node node = addWaiter(Node.*SHARED*);  
 boolean failed = true;  
 try {  
 for (;;) {  
 // 获取上一个节点。  
 // 如果上一节点是CLH队列的表头，则"尝试获取共享锁"。  
 final Node p = node.predecessor();  
 if (p == head) {  
 long r = tryAcquireShared(arg);  
 if (r >= 0) {  
 setHeadAndPropagate(node, r);  
 p.next = null; // help GC  
 failed = false;  
 return;  
 }  
 }  
 // (上一节点不是CLH队列的表头) 当前线程一直等待，直到获取到共享锁。  
 // 如果线程在等待过程中被中断过，则再次中断该线程(还原之前的中断状态)。  
 if (shouldParkAfterFailedAcquire(p, node) &&  
 parkAndCheckInterrupt())  
 throw new InterruptedException();  
 }  
 } finally {  
 if (failed)  
 cancelAcquire(node);  
 }  
}

**说明**：  
(01) addWaiter(Node.SHARED)的作用是，创建”当前线程“的Node节点，且Node中记录的锁的类型是”共享锁“(Node.SHARED)；并将该节点添加到CLH队列末尾。  
(02) node.predecessor()的作用是，获取上一个节点。如果上一节点是CLH队列的表头，则”尝试获取共享锁“。  
(03) shouldParkAfterFailedAcquire()的作用和它的名称一样，如果在尝试获取锁失败之后，线程应该等待，则返回true；否则，返回false。  
(04) 当shouldParkAfterFailedAcquire()返回ture时，则调用parkAndCheckInterrupt()，当前线程会进入等待状态，直到获取到共享锁才继续运行。

响应中断的doAcquireShared(int),[见AQS篇doAcquireShared(int)](#_doAcquireShared(int))

###### countDown()

public void countDown() {  
 sync.releaseShared(1);  
}

**说明**：该函数实际上调用releaseShared(1)释放共享锁。

releaseShared(int)

public final boolean releaseShared(int arg) {  
 if (tryReleaseShared(arg)) {  
 doReleaseShared();//唤醒后续节点  
 return true;  
 }  
 return false;  
}

[见AQS篇 3.2.2.4 releaseShared(int)](#_3.2.2.4_releaseShared(int))

##### 小结

CountDownLatch是通过“共享锁”实现的。在创建CountDownLatch中时，会传递一个int类型参数count，该参数是“锁计数器”的初始状态，表示该“共享锁”最多能被count给线程同时获取。当某线程调用该CountDownLatch对象的await()方法时，该线程会等待“共享锁”可用时，才能获取“共享锁”进而继续运行。而“共享锁”可用的条件，就是“锁计数器”的值为0！而“锁计数器”的初始值为count，每当一个线程调用该CountDownLatch对象的countDown()方法时，才将“锁计数器”-1；通过这种方式，必须有count个线程调用countDown()之后，“锁计数器”才为0，而前面提到的等待线程才能继续运行！

#### CyclicBarrier

##### 概述

CyclicBarrier是一个同步辅助类，允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)。因为该 barrier 在释放等待线程后可以重用，所以称它为循环 的 barrier。

注意比较CountDownLatch和CyclicBarrier：  
(01) CountDownLatch的作用是允许1或N个线程等待其他线程完成执行；而CyclicBarrier则是允许N个线程相互等待。  
(02) CountDownLatch的计数器无法被重置；CyclicBarrier的计数器可以被重置后使用，因此它被称为是循环的barrier。

###### 函数列表

// 每次对barrier的使用可以表现为一个 generation 实例。当条件 trip 改变或者重置 generation 也会  
// 随之改变。可以有多个 generation 和使用barrier的线程关联，但是只有一个可以获得锁。  
private static class Generation {  
 boolean broken = false;  
}  
*/\*\* 守护barrier入口的锁 \*/*private final ReentrantLock lock = new ReentrantLock();  
*/\*\* 等待条件，直到所有线程到达barrier \*/*private final Condition trip = lock.newCondition();  
*/\*\* 要屏障的线程数 \*/*private final int parties;  
/\* 当线程都到达barrier，运行的 Runnable \*/  
private final Runnable barrierCommand;  
*/\*\* The current generation \*/*private Generation generation = new Generation();  
  
//还要等待多少个线程到达。线程到达屏障点就减去 1。  
//每次新建 generation 的时候或者屏障 broken，count重新设置为 parties 参数值  
private int count;  
  
//创建一个新的 CyclicBarrier，它将在给定数量的参与者（线程）处于等待状态时启动，但它不会在启动 barrier 时执行预定义的操作。  
CyclicBarrier(int parties)  
//创建一个新的 CyclicBarrier，它将在给定数量的参与者（线程）处于等待状态时启动，并在启动 barrier 时执行给定的屏障操作，该操作由最后一个进入 barrier 的线程执行。  
CyclicBarrier(int parties, Runnable barrierAction);  
//在 一个 barrier 完成后, 重新初始化值  
private void nextGeneration();  
//用于等待的线程当被中断, 或等待超时执行  
private void breakBarrier();  
private int dowait(boolean timed, long nanos);  
//在所有参与者都已经在此 barrier 上调用 await 方法之前，将一直等待。  
public int await();  
//在所有参与者都已经在此屏障上调用 await 方法之前将一直等待,或者超出了指定的等待时间。  
public int await(long timeout, TimeUnit unit);  
//返回要求启动此 barrier 的参与者数目。  
public int getParties();  
//查询此屏障是否处于损坏状态。  
public boolean isBroken();  
//将屏障重置为其初始状态。  
public void reset();  
//返回当前在屏障处等待的参与者数目。  
public int getNumberWaiting();

##### 源码详解

###### await()

public int await() throws InterruptedException, BrokenBarrierException {  
 try {  
 return dowait(false, 0L);  
 } catch (TimeoutException toe) {  
 throw new Error(toe); // cannot happen  
 }  
}

public int await(long timeout, TimeUnit unit)  
 throws InterruptedException,  
 BrokenBarrierException,  
 TimeoutException {  
 return dowait(true, unit.toNanos(timeout));  
}

doawait(boolean timed, long nanos)

private int dowait(boolean timed, long nanos)  
 throws InterruptedException, BrokenBarrierException,  
 TimeoutException {  
 final ReentrantLock lock = this.lock;  
 // 获取“独占锁(lock)”  
 lock.lock();  
 try {  
 // 保存“当前的generation”  
 final Generation g = generation;  
  
 // 若“当前generation已损坏”，则抛出异常。  
 if (g.broken)  
 throw new BrokenBarrierException();  
  
 // 如果当前线程被中断，则通过breakBarrier()终止CyclicBarrier，唤醒CyclicBarrier中所有等待线程。  
 if (Thread.*interrupted*()) {  
 breakBarrier();  
 throw new InterruptedException();  
 }  
  
 // 将“count计数器”-1  
 int index = --count;  
 // 如果index=0，则意味着“有parties个线程到达barrier”。  
 if (index == 0) { // tripped  
 boolean ranAction = false;  
 try {  
 // 如果barrierCommand不为null，则执行该动作。  
 final Runnable command = barrierCommand;  
 if (command != null)  
 command.run();  
 ranAction = true;  
 // 唤醒所有等待线程，并更新generation。  
 nextGeneration();  
 return 0;  
 } finally {  
 if (!ranAction)  
 breakBarrier();  
 }  
 }  
  
 // 当前线程一直阻塞，直到“有parties个线程到达barrier” 或 “当前线程被中断” 或 “超时”这3者之一发生，  
 // 当前线程才继续执行。  
 for (;;) {  
 try {  
 // 如果不是“超时等待”，则调用awati()进行等待；否则，调用awaitNanos()进行等待。  
 if (!timed)  
 trip.await();  
 else if (nanos > 0L)  
 nanos = trip.awaitNanos(nanos);  
 } catch (InterruptedException ie) {  
 // 如果等待过程中，线程被中断，则执行下面的函数。  
 if (g == generation && ! g.broken) {  
 breakBarrier();  
 throw ie;  
 } else {  
 Thread.*currentThread*().interrupt();  
 }  
 }  
  
 // 如果“当前generation已经损坏”，则抛出异常。  
 if (g.broken)  
 throw new BrokenBarrierException();  
  
 // 如果“generation已经换代”，则返回index。  
 if (g != generation)  
 return index;  
  
 // 如果是“超时等待”，并且时间已到，则通过breakBarrier()终止CyclicBarrier，唤醒CyclicBarrier中所有等待线程。  
 if (timed && nanos <= 0L) {  
 breakBarrier();  
 throw new TimeoutException();  
 }  
 }  
 } finally {  
 // 释放“独占锁(lock)”  
 lock.unlock();  
 }  
}

**说明**：dowait()的作用就是让当前线程阻塞，直到“有parties个线程到达barrier” 或 “当前线程被中断” 或 “超时”这3者之一发生，当前线程才继续执行。

(01) generation是CyclicBarrier的一个成员遍历，它的定义如下：

private static class Generation {  
 boolean broken = false;  
}

在CyclicBarrier中，同一批的线程属于同一代，即同一个Generation；CyclicBarrier中通过generation对象，记录属于哪一代。  
当有parties个线程到达barrier，generation就会被更新换代。

(02) 如果当前线程被中断，即Thread.interrupted()为true；则通过breakBarrier()终止CyclicBarrier。breakBarrier()的源码如下：

private void breakBarrier() {  
 generation.broken = true;  
 count = parties;  
 trip.signalAll();  
}

breakBarrier()会设置当前中断标记broken为true，意味着“将该Generation中断”；同时，设置count=parties，即重新初始化count；最后，通过signalAll()唤醒CyclicBarrier上所有的等待线程。

(03) 将“count计数器”-1，即--count；然后判断是不是“有parties个线程到达barrier”，即index是不是为0。当index=0时，如果barrierCommand不为null，则执行该barrierCommand，barrierCommand就是我们创建CyclicBarrier时，传入的Runnable对象。然后，调用nextGeneration()进行换代工作，nextGeneration()的源码如下：

private void nextGeneration() {  
 // signal completion of last generation  
 trip.signalAll();  
 // set up next generation  
 count = parties;  
 generation = new Generation();  
}

首先，它会调用signalAll()唤醒CyclicBarrier上所有的等待线程；接着，重新初始化count；最后，更新generation的值。

1. 在for(;;)循环中。timed是用来表示当前是不是“超时等待”线程。如果不是，则通过trip.await()进行等待；否则，调用awaitNanos()进行超时等待

#### Semaphore

##### 概述

Semaphore是一个计数信号量，它的本质是一个"**共享锁**"。

信号量维护了一个信号量许可集。线程可以通过调用acquire()来获取信号量的许可；当信号量中有可用的许可时，线程能获取该许可；否则线程必须等待，直到有可用的许可为止。 线程可以通过release()来释放它所持有的信号量许可。

###### 函数列表

// 创建具有给定的许可数和非公平的公平设置的 Semaphore。  
Semaphore(int permits)  
// 创建具有给定的许可数和给定的公平设置的 Semaphore。  
Semaphore(int permits, boolean fair)  
  
// 从此信号量获取一个许可，在提供一个许可前一直将线程阻塞，否则线程被中断。  
void acquire()  
// 从此信号量获取给定数目的许可，在提供这些许可前一直将线程阻塞，或者线程已被中断。  
void acquire(int permits)  
// 从此信号量中获取许可，在有可用的许可前将其阻塞。  
void acquireUninterruptibly()  
// 从此信号量获取给定数目的许可，在提供这些许可前一直将线程阻塞。  
void acquireUninterruptibly(int permits)  
// 返回此信号量中当前可用的许可数。  
int availablePermits()  
// 获取并返回立即可用的所有许可。  
int drainPermits()  
// 返回一个 collection，包含可能等待获取的线程。  
protected Collection<Thread> getQueuedThreads()  
// 返回正在等待获取的线程的估计数目。  
int getQueueLength()  
// 查询是否有线程正在等待获取。  
boolean hasQueuedThreads()  
// 如果此信号量的公平设置为 true，则返回 true。  
boolean isFair()  
// 根据指定的缩减量减小可用许可的数目。  
protected void reducePermits(int reduction)  
// 释放一个许可，将其返回给信号量。  
void release()  
// 释放给定数目的许可，将其返回到信号量。  
void release(int permits)  
// 返回标识此信号量的字符串，以及信号量的状态。  
String toString()  
// 仅在调用时此信号量存在一个可用许可，才从信号量获取许可。  
boolean tryAcquire()  
// 仅在调用时此信号量中有给定数目的许可时，才从此信号量中获取这些许可。  
boolean tryAcquire(int permits)  
// 如果在给定的等待时间内此信号量有可用的所有许可，并且当前线程未被中断，则从此信号量获取给定数目的许可。  
boolean tryAcquire(int permits, long timeout, TimeUnit unit)  
// 如果在给定的等待时间内，此信号量有可用的许可并且当前线程未被中断，则从此信号量获取一个许可。  
boolean tryAcquire(long timeout, TimeUnit unit)

###### 数据结构



(01) 和"ReentrantLock"一样，Semaphore也包含了sync对象，sync是Sync类型；而且，Sync是一个继承于AQS的抽象类。(02) Sync包括两个子类："公平信号量"FairSync 和 "非公平信号量"NonfairSync。sync是"FairSync的实例"，或者"NonfairSync的实例"；默认情况下，sync是NonfairSync(即，默认是非公平信号量)。

Sync源码:

abstract static class Sync extends AbstractQueuedSynchronizer {  
 private static final long *serialVersionUID* = 1192457210091910933L;  
  
 Sync(int permits) {  
 setState(permits);  
 }  
 //获取许可  
 final int getPermits() {  
 return getState();  
 }  
 //非公平获取  
 final int nonfairTryAcquireShared(int acquires) {  
 for (;;) {  
 int available = getState();  
 int remaining = available - acquires;  
 if (remaining < 0 ||  
 compareAndSetState(available, remaining))  
 return remaining;  
 }  
 }  
 //释放  
 protected final boolean tryReleaseShared(int releases) {  
 for (;;) {  
 int current = getState();  
 int next = current + releases;  
 if (next < current) // overflow  
 throw new Error("Maximum permit count exceeded");  
 if (compareAndSetState(current, next))  
 return true;  
 }  
 }  
 //减少指定许可数  
 final void reducePermits(int reductions) {  
 for (;;) {  
 int current = getState();  
 int next = current - reductions;  
 if (next > current) // underflow  
 throw new Error("Permit count underflow");  
 if (compareAndSetState(current, next))  
 return;  
 }  
 }  
 //获取并返回立即可用的所有许可  
 final int drainPermits() {  
 for (;;) {  
 int current = getState();  
 if (current == 0 || compareAndSetState(current, 0))  
 return current;  
 }  
 }  
}

*/\*\*非公平Sync\*/*static final class NonfairSync extends Sync {  
 private static final long *serialVersionUID* = -2694183684443567898L;  
  
 NonfairSync(int permits) {  
 super(permits);  
 }  
  
 protected int tryAcquireShared(int acquires) {  
 return nonfairTryAcquireShared(acquires);  
 }  
}  
*/\*\*公平Sync\*/*static final class FairSync extends Sync {  
 private static final long *serialVersionUID* = 2014338818796000944L;  
  
 FairSync(int permits) {  
 super(permits);  
 }  
  
 protected int tryAcquireShared(int acquires) {  
 for (;;) {  
 if (hasQueuedPredecessors())  
 return -1;  
 int available = getState();  
 int remaining = available - acquires;  
 if (remaining < 0 ||  
 compareAndSetState(available, remaining))  
 return remaining;  
 }  
 }  
}

**"公平信号量"和"非公平信号量"的区别**

"公平信号量"和"非公平信号量"的释放信号量的机制是一样的！不同的是它们获取信号量的机制：线程在尝试获取信号量许可时，对于公平信号量而言，如果当前线程不在CLH队列的头部，则排队等候；而对于非公平信号量而言，无论当前线程是不是在CLH队列的头部，它都会直接获取信号量。该差异具体的体现在，它们的tryAcquireShared()函数的实现不同。

##### 源码解析

###### acquire()

//获取信号量  
public void acquire() throws InterruptedException {  
 sync.acquireSharedInterruptibly(1);  
}  
//获取指定permits数的信号量  
public void acquire(int permits) throws InterruptedException {  
 if (permits < 0) throw new IllegalArgumentException();  
 sync.acquireSharedInterruptibly(permits);  
}  
//获取指定permits数的信号量,不响应中断  
public void acquireUninterruptibly(int permits) {  
 if (permits < 0) throw new IllegalArgumentException();  
 sync.acquireShared(permits);  
}

**说明:** 可以看出,Semaphore的获取锁是响应中断的,实际调用的是AQS的acquireSharedInterruptibly(int),

acquireSharedInterruptibly(int)

public final void acquireSharedInterruptibly(int arg)  
 throws InterruptedException {  
 if (Thread.*interrupted*())  
 throw new InterruptedException();  
 if (tryAcquireShared(arg) < 0)  
 doAcquireSharedInterruptibly(arg);  
}

[见CountDownLatch篇acquireSharedInterruptibly(int)](#_acquireSharedInterruptibly(int))

tryAcquireShared(int)

tryAcquireShared(int)在Semaphore中被重写,具体实现如下:

//非公平信号量获取  
protected int tryAcquireShared(int acquires) {  
 return nonfairTryAcquireShared(acquires);  
}  
//公平信号量获取  
protected int tryAcquireShared(int acquires) {  
 for (;;) {  
 // 判断当前线程是否还有前任线程  
 if (hasQueuedPredecessors())  
 return -1;  
 //可获得的信号数  
 int available = getState();  
 //获取信号数之后剩余的信号数  
 int remaining = available - acquires;  
 if (remaining < 0 ||  
 compareAndSetState(available, remaining))  
 return remaining;  
 }  
}

**说明:**

(1)非公平获取中,调用Sync的nonfairTryAcquireShared (int),源码如下:

final int nonfairTryAcquireShared(int acquires) {  
 for (;;) {  
 int available = getState();  
 int remaining = available - acquires;  
 if (remaining < 0 ||  
 compareAndSetState(available, remaining))  
 return remaining;  
 }  
}

(2)公平获取信号量中,首先根据hasQueuePredecessors()判断是否还有前任线程,有的话则当前线程需要通过doAcquireSharedInterruptibly(int)进入等待队列自旋等待.

doAcquireSharedInterruptibly(int)

**说明:** [见CountDownLatch篇doAcquireSharedInterruptibly(int)](#_doAcquireSharedInterruptibly(int))

###### release()

"公平信号量"和"非公平信号量"的释放信号量的机制是相同的

//释放信号量  
public void release() {  
 sync.releaseShared(1);  
}  
//释放指定permits数的信号量  
public void release(int permits) {  
 if (permits < 0) throw new IllegalArgumentException();  
 sync.releaseShared(permits);  
}

**说明:** 实际调用的是AQS的releaseShared(int).

releaseShared(int)

public final boolean releaseShared(int arg) {  
 if (tryReleaseShared(arg)) {  
 doReleaseShared();  
 return true;  
 }  
 return false;  
}

**说明:** tryReleaseShared(int)在Semaphore中的Sync重写.

tryReleaseShared(int)

protected final boolean tryReleaseShared(int releases) {  
 for (;;) {  
 //可获得的信号数  
 int current = getState();  
 //释放releases个信号后,剩余可获得的信号数  
 int next = current + releases;  
 if (next < current) // overflow  
 throw new Error("Maximum permit count exceeded");  
 //设置可获得的信号数为next  
 if (compareAndSetState(current, next))  
 return true;  
 }  
}

说明: 如果在tryReleaseShared()中尝试释放共享锁失败,则会调用doReleaseShared()进行释放.

doReleaseShared()

[见AQS篇doReleaseShared(int)](#_doReleaseShared(int))

#### StampedLock

##### 概述

StampedLock是JDK1.8新增的锁,与之前的ReentrantLock，ReentrantReadWriteLock使用队列同步列AQS实现有所不同，StampedLock的state改为了一个long型的变量，同时状态的设计也有所不同。同时由于没有使用AQSStampedLock直接在内部实现了同步等待队列，并且节点属性中有一个叫做cowait的分支用于标识另一个等待获取读状态的链。

StampedLock采用了一个long型作为state，把这个64位的state的前7位作为读状态，第8位标识写状态，这也是为什么不支持重入的原因吧。

##### 相关属性定义

//获取CPU的可用线程数量，用于确定自旋的时候循环次数  
private static final int *NCPU* = Runtime.*getRuntime*().availableProcessors();  
//根据NCPU确定自旋的次数限制(并不是一定这么多次，因为实际代码中是随机的)  
private static final int *SPINS* = (*NCPU* > 1) ? 1 << 6 : 0;  
//头节点上的自旋次数  
private static final int *HEAD\_SPINS* = (*NCPU* > 1) ? 1 << 10 : 0;  
//头节点上的最大自旋次数  
private static final int *MAX\_HEAD\_SPINS* = (*NCPU* > 1) ? 1 << 16 : 0;  
private static final int *LG\_READERS* = 7;  
//一个读状态单位  
private static final long *RUNIT* = 1L;  
//写状态标识  
private static final long *WBIT* = 1L << *LG\_READERS*;  
//读状态标识(前7位)  
private static final long *RBITS* = *WBIT* - 1L;  
//最大的读状态  
private static final long *RFULL* = *RBITS* - 1L;  
//用于获取读写状态  
private static final long *ABITS* = *RBITS* | *WBIT*;  
private static final long *SBITS* = ~*RBITS*; // note overlap with ABITS  
//初始化状态  
private static final long *ORIGIN* = *WBIT* << 1;  
//中断标识  
private static final long *INTERRUPTED* = 1L;  
// 等待/取消  
private static final int *WAITING* = -1;  
private static final int *CANCELLED* = 1;  
//读/写状态  
private static final int *RMODE* = 0;  
private static final int *WMODE* = 1;  
//因为读状态只有7位很小，所以当超过了128之后将使用一个int变量来记录  
private transient int readerOverflow;

状态标识:

写线程获取到了写状态只需判断：state < WBIT

读状态是否超出：(state & ABITS) < RFULL

获取读状态： state + RUNIT(或者readerOverflow + 1)

获取写状态: state + WBIT

释放读状态： state - RUNIT(或者readerOverflow - 1)

释放写状态： (s += WBIT) == 0L ? ORIGIN : s

是否为写锁： (state & WBIT) != 0L

是否为读锁： (state & RBITS) != 0L

##### 小结

一 StampedLock是不可重入的，

二StampedLock对于带有中断的线程的处理可能导致CPU暴涨。

*/\*\*  
 \* 先开启一个线程获取写锁并保持10秒，再开启10个带着中断状态的线程去获取读锁（readLock方法），结果是3个核心被占据了近10秒。  
 \* 原因在于没有使用保存/复原中断状态的机制  
 \* Created by zhaoshengqi on 2017/8/1.  
 \*/*public class TestBugStampedLock {  
  
 public static void main(String[] args) throws InterruptedException{  
 final StampedLock lock = new StampedLock();  
 new Thread(){  
 public void run(){  
 long readLong = lock.writeLock();  
 LockSupport.*parkNanos*(10100000000L);  
 lock.unlockWrite(readLong);  
 }  
 }.start();  
 Thread.*sleep*(100);  
 for( int i = 0; i < 10; ++i)  
 new Thread(new OccupiedCPUReadThread(lock)).start();  
 }  
 private static class OccupiedCPUReadThread implements Runnable{  
 private StampedLock lock;  
 public OccupiedCPUReadThread(StampedLock lock){  
 this.lock = lock;  
 }  
 public void run(){  
 Thread.*currentThread*().interrupt();  
 long lockr = lock.readLock();  
 System.*out*.println(Thread.*currentThread*().getName() + " get read lock");  
 lock.unlockRead(lockr);  
 }  
 }  
}

三StampedLock的所有的try开头的获取都将直接尝试获取锁。

四StampedLock适用于读多写少的场景。

五 源码暂时看不懂

### JUC集合篇

#### 框架

##### List和Set



(01) CopyOnWriteArrayList相当于线程安全的ArrayList，它实现了List接口。CopyOnWriteArrayList是支持高并发的。  
(02) CopyOnWriteArraySet相当于线程安全的HashSet，它继承于AbstractSet类。CopyOnWriteArraySet内部包含一个CopyOnWriteArrayList对象，它是通过CopyOnWriteArrayList实现的。

(03) ConcurrentSkipListSet在Map中说明。

##### Map



(01) ConcurrentHashMap是线程安全的哈希表(相当于线程安全的HashMap)；它继承于AbstractMap类，并且实现ConcurrentMap接口。ConcurrentHashMap是通过“锁分段”来实现的，它支持并发。

(02) ConcurrentSkipListMap是线程安全的有序的哈希表(相当于线程安全的TreeMap); 它继承于AbstractMap类，并且实现ConcurrentNavigableMap接口。ConcurrentSkipListMap是通过“跳表”来实现的，它支持并发。

(03) ConcurrentSkipListSet是线程安全的有序的集合(相当于线程安全的TreeSet)；它继承于AbstractSet，并实现了NavigableSet接口。ConcurrentSkipListSet是通过ConcurrentSkipListMap实现的，它也支持并发。

##### Queue



(01) ArrayBlockingQueue是数组实现的线程安全的有界的阻塞队列。  
(02) LinkedBlockingQueue是单向链表实现的(指定大小)阻塞队列,使用锁机制实现，该队列按 FIFO（先进先出）排序元素。  
(03) LinkedBlockingDeque是双向链表实现的(指定大小)双向并发阻塞队列，使用锁机制实现,该阻塞队列同时支持FIFO和FILO两种操作方式。  
(04) ConcurrentLinkedQueue是单向链表实现的无界并发队列，通过cas实现并发,该队列按 FIFO（先进先出）排序元素。  
(05) ConcurrentLinkedDeque是双向链表实现的无界并发队列，通过cas实现并发,该队列同时支持FIFO和FILO两种操作方式。  
(06) DelayQueue延时无界阻塞队列，队列里只允许放可以“延期”的元素，队列中的head是最先“到期”的元素。如果队里中没有元素到“到期”，那么就算队列中有元素也不能获取到。  
(07) PriorityBlockingQueue 无界优先级阻塞队列，priorityQueue的线程安全版，不允许存放null值，依赖于comparable的排序，不允许存放不可比较的对象类型。  
(08) SynchronousQueue没有容量的同步队列，支持FIFO和FILO  
(09) LinkedTransferQueue 1.7新增,阻塞队列，可以说是ConcurrentLinkedQueue、SynchronousQueue（公平模式）和LinkedBlockingQueue的超集, 它不仅仅综合了这几个类的功能，同时也提供了更高效的实现。

#### CopyOnwriteArrayList

##### 概述

它相当于线程安全的[ArrayList](http://www.cnblogs.com/skywang12345/p/3308556.html)。和ArrayList一样，它是个可变数组；但是和ArrayList不同的时，它具有以下特性：  
1. 它最适合于具有以下特征的应用程序：List 大小通常保持很小，只读操作远多于可变操作，需要在遍历期间防止线程间的冲突。  
2. 它是线程安全的,使用ReentrantLock和volatile保证线程安全。  
3. 因为通常需要复制整个基础数组，所以可变操作（add()、set() 和 remove() 等等）的开销很大。  
4. 迭代器支持hasNext(), next()等不可变操作，但不支持可变 remove()等操作。  
5. 使用迭代器进行遍历的速度很快，并且不会与其他线程发生冲突。在构造迭代器时，迭代器依赖于不变的数组快照。

###### 数据结构



**说明**：  
1. CopyOnWriteArrayList实现了List接口，因此它是一个队列。  
2. CopyOnWriteArrayList包含了成员lock。每一个CopyOnWriteArrayList都和一个互斥锁lock绑定，通过lock，实现了对CopyOnWriteArrayList的互斥访问。  
3. CopyOnWriteArrayList包含了用volatile修饰的成员array数组，这说明CopyOnWriteArrayList本质上通过数组实现的。

下面从“动态数组”和“线程安全”两个方面进一步对CopyOnWriteArrayList的原理进行说明。  
1.**CopyOnWriteArrayList的“动态数组”机制** -- 它内部有个“volatile数组”(array)来保持数据。在“添加/修改/删除”数据时，都会新建一个数组，并将更新后的数据拷贝到新建的数组中，最后再将该数组赋值给“volatile数组”。这就是它叫做CopyOnWriteArrayList的原因！CopyOnWriteArrayList就是通过这种方式实现的动态数组；不过正由于它在“添加/修改/删除”数据时，都会新建数组，所以涉及到修改数据的操作，CopyOnWriteArrayList效率很低；但是单单只是进行遍历查找的话，效率比较高。  
2.**CopyOnWriteArrayList的“线程安全”机制** -- 是通过volatile和互斥锁来实现的。  
(01) CopyOnWriteArrayList是通过“volatile数组”来保存数据的。一个线程读取volatile数组时，总能看到其它线程对该volatile变量最后的写入；就这样，通过volatile提供了“读取到的数据总是最新的”这个机制的保证。  
(02) CopyOnWriteArrayList通过互斥锁来保护数据。在“添加/修改/删除”数据时，会先“获取互斥锁”，再修改完毕之后，先将数据更新到“volatile数组”中，然后再“释放互斥锁”；这样，就达到了保护数据的目的。

###### 函数列表

// 创建一个空列表。  
CopyOnWriteArrayList();  
// 创建一个按 collection 的迭代器返回元素的顺序包含指定 collection 元素的列表。  
CopyOnWriteArrayList(Collection<? extends E> c)  
//创建一个保存给定数组的副本的列表。  
CopyOnWriteArrayList(E[] toCopyIn)  
  
// 将指定元素添加到此列表的尾部。  
boolean add(E e)  
// 在此列表的指定位置上插入指定元素。  
void add(int index, E element)  
// 按照指定 collection 的迭代器返回元素的顺序，将指定 collection 中的所有元素添加此列表的尾部。  
boolean addAll(Collection<? extends E> c)  
// 从指定位置开始，将指定 collection 的所有元素插入此列表。  
boolean addAll(int index, Collection<? extends E> c)  
// 按照指定 collection 的迭代器返回元素的顺序，将指定 collection 中尚未包含在此列表中的所有元素添加列表的尾部。  
int addAllAbsent(Collection<? extends E> c)  
// 添加元素（如果不存在）。  
boolean addIfAbsent(E e)  
// 从此列表移除所有元素。  
void clear()  
// 返回此列表的浅表副本。  
Object clone()  
// 如果此列表包含指定的元素，则返回 true。  
boolean contains(Object o)  
// 如果此列表包含指定 collection 的所有元素，则返回 true。  
boolean containsAll(Collection<?> c)  
// 比较指定对象与此列表的相等性。  
boolean equals(Object o)  
// 返回列表中指定位置的元素。  
E get(int index)  
// 返回此列表的哈希码值。  
int hashCode()  
// 返回第一次出现的指定元素在此列表中的索引，从 index 开始向前搜索，如果没有找到该元素，则返回 -1。  
int indexOf(E e, int index)  
// 返回此列表中第一次出现的指定元素的索引；如果此列表不包含该元素，则返回 -1。  
int indexOf(Object o)  
// 如果此列表不包含任何元素，则返回 true。  
boolean isEmpty()  
// 返回以恰当顺序在此列表元素上进行迭代的迭代器。  
Iterator<E> iterator()  
// 返回最后一次出现的指定元素在此列表中的索引，从 index 开始向后搜索，如果没有找到该元素，则返回 -1。  
int lastIndexOf(E e, int index)  
// 返回此列表中最后出现的指定元素的索引；如果列表不包含此元素，则返回 -1。  
int lastIndexOf(Object o)  
// 返回此列表元素的列表迭代器（按适当顺序）。  
ListIterator<E> listIterator()  
// 返回列表中元素的列表迭代器（按适当顺序），从列表的指定位置开始。  
ListIterator<E> listIterator(int index)  
// 移除此列表指定位置上的元素。  
E remove(int index)  
// 从此列表移除第一次出现的指定元素（如果存在）。  
boolean remove(Object o)  
// 从此列表移除所有包含在指定 collection 中的元素。  
boolean removeAll(Collection<?> c)  
// 只保留此列表中包含在指定 collection 中的元素。  
boolean retainAll(Collection<?> c)  
// 用指定的元素替代此列表指定位置上的元素。  
E set(int index, E element)  
// 返回此列表中的元素数。  
int size()  
// 返回此列表中 fromIndex（包括）和 toIndex（不包括）之间部分的视图。  
List<E> subList(int fromIndex, int toIndex)  
// 返回一个按恰当顺序（从第一个元素到最后一个元素）包含此列表中所有元素的数组。  
Object[] toArray()  
// 返回以恰当顺序（从第一个元素到最后一个元素）包含列表所有元素的数组；返回数组的运行时类型是指定数组的运行时类型。  
<T> T[] toArray(T[] a)  
// 返回此列表的字符串表示形式。  
String toString();

##### 源码解析

###### 创建

private transient volatile Object[] array;  
final Object[] getArray() {  
 return array;  
}  
final void setArray(Object[] a) {  
 array = a;  
}  
  
public CopyOnWriteArrayList() {  
 setArray(new Object[0]);  
}  
  
public CopyOnWriteArrayList(Collection<? extends E> c) {  
 Object[] elements;  
 if (c.getClass() == CopyOnWriteArrayList.class)  
 elements = ((CopyOnWriteArrayList<?>)c).getArray();  
 else {  
 elements = c.toArray();  
 // c.toArray might (incorrectly) not return Object[] (see 6260652)  
 if (elements.getClass() != Object[].class)  
 elements = Arrays.*copyOf*(elements, elements.length, Object[].class);  
 }  
 setArray(elements);  
}  
  
public CopyOnWriteArrayList(E[] toCopyIn) {  
 setArray(Arrays.*copyOf*(toCopyIn, toCopyIn.length, Object[].class));  
}

**说明**：setArray()的作用是给array赋值；其中，array是volatile transient Object[]类型，即array是“volatile数组”。  
关于volatile关键字，我们知道“volatile能让变量变得可见”，即对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入。正在由于这种特性，每次更新了“volatile数组”之后，其它线程都能看到对它所做的更新。  
关于transient关键字，它是在序列化中才起作用，transient变量不会被自动序列化。

###### add(index, E)

public boolean add(E e) {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 Object[] elements = getArray();  
 int len = elements.length;  
 Object[] newElements = Arrays.*copyOf*(elements, len + 1);  
 newElements[len] = e;  
 setArray(newElements);  
 return true;  
 } finally {  
 lock.unlock();  
 }  
}  
public void add(int index, E element) {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 Object[] elements = getArray();  
 int len = elements.length;  
 if (index > len || index < 0)  
 throw new IndexOutOfBoundsException("Index: "+index+  
 ", Size: "+len);  
 Object[] newElements;  
 //计算偏移量  
 int numMoved = len - index;  
 if (numMoved == 0)  
 //作为add(E)处理  
 newElements = Arrays.*copyOf*(elements, len + 1);  
 else {  
 newElements = new Object[len + 1];  
 //调用native方法根据index拷贝原数组的前半段  
 System.*arraycopy*(elements, 0, newElements, 0, index);  
 //拷贝后半段  
 System.*arraycopy*(elements, index, newElements, index + 1,  
 numMoved);  
 }  
 newElements[index] = element;  
 setArray(newElements);  
 } finally {  
 lock.unlock();  
 }  
}

//System中arrayCopy的实现  
public static native void arraycopy(Object src, int srcPos,  
 Object dest, int destPos,  
 int length);

**说明**：add(E e)的作用就是将数据e添加到”volatile数组“中。它的实现方式是，新建一个数组，接着将原始的”volatile数组“的数据拷贝到新数组中，然后将新增数据也添加到新数组中；最后，将新数组赋值给”volatile数组“。add(int index, E e)中,调用了native方法,根据index把原E分为两段拷贝

###### addIfAbsent(E)

public boolean addIfAbsent(E e) {  
 Object[] snapshot = getArray();  
 return indexOf(e, snapshot, 0, snapshot.length) >= 0 ? false :  
 addIfAbsent(e, snapshot);  
}  
  
private boolean addIfAbsent(E e, Object[] snapshot) {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 Object[] current = getArray();  
 int len = current.length;  
 if (snapshot != current) {  
 // Optimize for lost race to another addXXX operation  
 //操作中有别的线程对array做了修改,取较小的那个length  
 int common = Math.*min*(snapshot.length, len);  
 for (int i = 0; i < common; i++)  
 if (current[i] != snapshot[i] && eq(e, current[i]))  
 return false;  
 if (indexOf(e, current, common, len) >= 0)  
 return false;  
 }  
 Object[] newElements = Arrays.*copyOf*(current, len + 1);  
 newElements[len] = e;  
 setArray(newElements);  
 return true;  
 } finally {  
 lock.unlock();  
 }  
}

**说明**：addIfAbsent(E e)与add不同的是,如果当前元素存在就不添加,并返回false“  
注意:addIfAbsent(E e)首先判断indexOf,这个操作是没有lock的,所以addIfAbsent(E, O[])会判断参数数组是否跟加了lock之后获取到的数组是否相同

###### set(index, e)

public E set(int index, E element) {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 Object[] elements = getArray();  
 E oldValue = get(elements, index);  
  
 if (oldValue != element) {  
 int len = elements.length;  
 Object[] newElements = Arrays.*copyOf*(elements, len);  
 newElements[index] = element;  
 setArray(newElements);  
 } else {  
 // Not quite a no-op; ensures volatile write semantics  
 setArray(elements);  
 }  
 return oldValue;  
 } finally {  
 lock.unlock();  
 }  
}

**说明**：和add()相比,add返回boolean,set返回修改之前的值oldValue

###### remove()

public E remove(int index) {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 Object[] elements = getArray();  
 int len = elements.length;  
 E oldValue = get(elements, index);  
 int numMoved = len - index - 1;  
 if (numMoved == 0)  
 setArray(Arrays.*copyOf*(elements, len - 1));  
 else {  
 Object[] newElements = new Object[len - 1];  
 System.*arraycopy*(elements, 0, newElements, 0, index);  
 System.*arraycopy*(elements, index + 1, newElements, index,  
 numMoved);  
 setArray(newElements);  
 }  
 return oldValue;  
 } finally {  
 lock.unlock();  
 }  
}  
//移除某个元素  
public boolean remove(Object o) {  
 Object[] snapshot = getArray();  
 int index = indexOf(o, snapshot, 0, snapshot.length);  
 return (index < 0) ? false : remove(o, snapshot, index);  
}  
  
private boolean remove(Object o, Object[] snapshot, int index) {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 Object[] current = getArray();  
 int len = current.length;  
 //判断是否有别的线程对array做了修改  
 if (snapshot != current) findIndex: {  
 int prefix = Math.*min*(index, len);  
 for (int i = 0; i < prefix; i++) {  
 //找到其他线程修改后的不同的元素 比较是否为此线程需要操作的元素  
 if (current[i] != snapshot[i] && eq(o, current[i])) {  
 index = i;  
 break findIndex;  
 }  
 }  
 if (index >= len)  
 return false;  
 if (current[index] == o)  
 break findIndex;  
 index = indexOf(o, current, index, len);  
 if (index < 0)  
 return false;  
 }  
 Object[] newElements = new Object[len - 1];  
 System.*arraycopy*(current, 0, newElements, 0, index);  
 System.*arraycopy*(current, index + 1,  
 newElements, index,  
 len - index - 1);  
 setArray(newElements);  
 return true;  
 } finally {  
 lock.unlock();  
 }  
}  
  
void removeRange(int fromIndex, int toIndex) {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 Object[] elements = getArray();  
 int len = elements.length;  
  
 if (fromIndex < 0 || toIndex > len || toIndex < fromIndex)  
 throw new IndexOutOfBoundsException();  
 int newlen = len - (toIndex - fromIndex);  
 int numMoved = len - toIndex;  
 if (numMoved == 0)  
 setArray(Arrays.*copyOf*(elements, newlen));  
 else {  
 Object[] newElements = new Object[newlen];  
 System.*arraycopy*(elements, 0, newElements, 0, fromIndex);  
 System.*arraycopy*(elements, toIndex, newElements,  
 fromIndex, numMoved);  
 setArray(newElements);  
 }  
 } finally {  
 lock.unlock();  
 }  
}

**说明**：(1) remove(int index)的作用就是将”volatile数组“中第index个元素删除。它的实现方式是，如果被删除的是最后一个元素，则直接通过Arrays.copyOf()进行处理，而不需要新建数组。否则，新建数组，然后将”volatile数组中被删除元素之外的其它元素“拷贝到新数组中；最后，将新数组赋值给”volatile数组“。

(2)remove(O)是把数组中的o元素移除(如果存在),成功返回true.调用了remove(O,snapshot,index),跟addIfAbsent一样,也会判断是否在操作中有其他线程对数组做了修改.

###### iterator()

public Iterator<E> iterator() {  
 return new COWIterator<E>(getArray(), 0);  
}

**说明**：iterator()会返回COWIterator对象。

COWIterator实现额ListIterator接口，它的源码如下：

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;  
 }  
  
 public boolean hasNext() {  
 return cursor < snapshot.length;  
 }  
  
 public boolean hasPrevious() {  
 return cursor > 0;  
 }  
  
 @SuppressWarnings("unchecked")  
 public E next() {  
 if (! hasNext())  
 throw new NoSuchElementException();  
 return (E) snapshot[cursor++];  
 }  
  
 @SuppressWarnings("unchecked")  
 public E previous() {  
 if (! hasPrevious())  
 throw new NoSuchElementException();  
 return (E) snapshot[--cursor];  
 }  
  
 public int nextIndex() {  
 return cursor;  
 }  
  
 public int previousIndex() {  
 return cursor-1;  
 }  
  
 */\*\*  
 \* Not supported. Always throws UnsupportedOperationException.  
 \** ***@throws*** *UnsupportedOperationException always; {****@code*** *remove}  
 \* is not supported by this iterator.  
 \*/* public void remove() {  
 throw new UnsupportedOperationException();  
 }  
  
 */\*\*  
 \* Not supported. Always throws UnsupportedOperationException.  
 \** ***@throws*** *UnsupportedOperationException always; {****@code*** *set}  
 \* is not supported by this iterator.  
 \*/* public void set(E e) {  
 throw new UnsupportedOperationException();  
 }  
  
 */\*\*  
 \* Not supported. Always throws UnsupportedOperationException.  
 \** ***@throws*** *UnsupportedOperationException always; {****@code*** *add}  
 \* is not supported by this iterator.  
 \*/* public void add(E e) {  
 throw new UnsupportedOperationException();  
 }  
  
 @Override  
 public void forEachRemaining(Consumer<? super E> action) {  
 Objects.*requireNonNull*(action);  
 Object[] elements = snapshot;  
 final int size = elements.length;  
 for (int i = cursor; i < size; i++) {  
 @SuppressWarnings("unchecked") E e = (E) elements[i];  
 action.accept(e);  
 }  
 cursor = size;  
 }  
}

**说明**：COWIterator不支持修改元素的操作。例如，对于remove(),set(),add()等操作，COWIterator都会抛出异常！  
另外，需要提到的一点是，CopyOnWriteArrayList返回迭代器不会抛出ConcurrentModificationException异常，即它不是fail-fast机制的

#### CopyOnwriteArraySet

##### 概述

它是线程安全的无序的集合，可以将它理解成线程安全的HashSet。CopyOnWriteArraySet和HashSet虽然都继承于共同的父类AbstractSet；但是，HashSet是通过“散列表(HashMap)”实现的，而CopyOnWriteArraySet则是通过“动态数组(CopyOnWriteArrayList)”实现的，并不是散列表。  
和CopyOnWriteArrayList类似，CopyOnWriteArraySet具有以下特性：  
1. 它最适合于具有以下特征的应用程序：Set 大小通常保持很小，只读操作远多于可变操作，需要在遍历期间防止线程间的冲突。  
2. 它是线程安全的。  
3. 因为通常需要复制整个基础数组，所以可变操作（add()、set() 和 remove() 等等）的开销很大。  
4. 迭代器支持hasNext(), next()等不可变操作，但不支持可变 remove()等 操作。  
5. 使用迭代器进行遍历的速度很快，并且不会与其他线程发生冲突。在构造迭代器时，迭代器依赖于不变的数组快照。

###### 数据结构



**说明**：  
  1. CopyOnWriteArraySet继承于AbstractSet，这就意味着它是一个集合。  
  2. CopyOnWriteArraySet包含CopyOnWriteArrayList对象，它是通过CopyOnWriteArrayList实现的。而CopyOnWriteArrayList本质是个动态数组队列，所以CopyOnWriteArraySet相当于通过通过动态数组实现的“集合”. CopyOnWriteArrayList中允许有重复的元素；但是，CopyOnWriteArraySet是一个集合，所以它不能有重复集合。因此，CopyOnWriteArrayList额外提供了addIfAbsent()和addAllAbsent()这两个添加元素的API，通过这些API来添加元素时，只有当元素不存在时才执行添加操作！  
   至于CopyOnWriteArraySet的“线程安全”机制，和CopyOnWriteArrayList一样，是通过volatile和互斥锁来实现的。

###### 函数列表

// 创建一个空 set。  
CopyOnWriteArraySet()  
// 创建一个包含指定 collection 所有元素的 set。  
CopyOnWriteArraySet(Collection<? extends E> c)  
  
// 如果指定元素并不存在于此 set 中，则添加它。  
boolean add(E e)  
// 如果此 set 中没有指定 collection 中的所有元素，则将它们都添加到此 set 中。  
boolean addAll(Collection<? extends E> c)  
// 移除此 set 中的所有元素。  
void clear()  
// 如果此 set 包含指定元素，则返回 true。  
boolean contains(Object o)  
// 如果此 set 包含指定 collection 的所有元素，则返回 true。  
boolean containsAll(Collection<?> c)  
// 比较指定对象与此 set 的相等性。  
boolean equals(Object o)  
// 如果此 set 不包含任何元素，则返回 true。  
boolean isEmpty()  
// 返回按照元素添加顺序在此 set 中包含的元素上进行迭代的迭代器。  
Iterator<E> iterator()  
// 如果指定元素存在于此 set 中，则将其移除。  
boolean remove(Object o)  
// 移除此 set 中包含在指定 collection 中的所有元素。  
boolean removeAll(Collection<?> c)  
// 仅保留此 set 中那些包含在指定 collection 中的元素。  
boolean retainAll(Collection<?> c)  
// 返回此 set 中的元素数目。  
int size()  
// 返回一个包含此 set 所有元素的数组。  
Object[] toArray()  
// 返回一个包含此 set 所有元素的数组；返回数组的运行时类型是指定数组的类型。  
<T> T[] toArray(T[] a);

##### 源码解析

public boolean add(E e) {  
 return al.addIfAbsent(e);  
}  
public boolean remove(Object o) {  
 return al.remove(o);  
}

CopyOnWriteArraySet是通过CopyOwriteArrayList实现的,具体源码解析见[CopyOwriteArrayList篇源码解析](#_源码解析)

#### ConcurrentHashMap

##### 概述

ConcurrentHashMap是线程安全的哈希表。HashMap, Hashtable, ConcurrentHashMap之间的关联如下：

**HashMap**是非线程安全的哈希表，常用于单线程程序中。

**Hashtable**是线程安全的哈希表，它是通过synchronized来保证线程安全的；即，多线程通过同一个“对象的同步锁”来实现并发控制。Hashtable在线程竞争激烈时，效率比较低(此时建议使用ConcurrentHashMap)！因为当一个线程访问Hashtable的同步方法时，其它线程就访问Hashtable的同步方法时，可能会进入阻塞状态。

**ConcurrentHashMap**是线程安全的哈希表，1.7之前是通过“锁分段”来保证线程安全的,ConcurrentHashMap将哈希表分成许多片段(Segment)，每一个片段除了保存哈希表之外，本质上也是一个“可重入的互斥锁”(ReentrantLock)。多线程对同一个片段的访问，是互斥的；但是，对于不同片段的访问，却是可以同步进行的。1.8之后放弃了这种臃肿的设计,取而代之的是Node+CAS+synchronized来保证并发安全.



图4.4.1 jdk1.7实现方式



图4.4.2 jdk1.8实现方式

##### 函数列表

// 创建一个带有默认初始容量 (16)、加载因子 (0.75) 和 concurrencyLevel (16) 的新的空映射。  
ConcurrentHashMap()  
// 创建一个带有指定初始容量、默认加载因子 (0.75) 和 concurrencyLevel (16) 的新的空映射。  
ConcurrentHashMap(int initialCapacity)  
// 创建一个带有指定初始容量、加载因子和默认 concurrencyLevel (16) 的新的空映射。  
ConcurrentHashMap(int initialCapacity, float loadFactor)  
// 创建一个带有指定初始容量、加载因子和并发级别的新的空映射。  
ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel)  
// 构造一个与给定映射具有相同映射关系的新映射。  
ConcurrentHashMap(Map<? extends K,? extends V> m)  
  
// 从该映射中移除所有映射关系  
void clear()  
// 一种遗留方法，测试此表中是否有一些与指定值存在映射关系的键。  
boolean contains(Object value)  
// 测试指定对象是否为此表中的键。  
boolean containsKey(Object key)  
// 如果此映射将一个或多个键映射到指定值，则返回 true。  
boolean containsValue(Object value)  
// 返回此表中值的枚举。  
Enumeration<V> elements()  
// 返回此映射所包含的映射关系的 Set 视图。  
Set<Map.Entry<K,V>> entrySet()  
// 返回指定键所映射到的值，如果此映射不包含该键的映射关系，则返回 null。  
V get(Object key)  
// 如果此映射不包含键-值映射关系，则返回 true。  
boolean isEmpty()  
// 返回此表中键的枚举。  
Enumeration<K> keys()  
// 返回此映射中包含的键的 Set 视图。  
Set<K> keySet()  
// 将指定键映射到此表中的指定值。  
V put(K key, V value)  
// 将指定映射中所有映射关系复制到此映射中。  
void putAll(Map<? extends K,? extends V> m)  
// 如果指定键已经不再与某个值相关联，则将它与给定值关联。  
V putIfAbsent(K key, V value)  
// 从此映射中移除键（及其相应的值）。  
V remove(Object key)  
// 只有目前将键的条目映射到给定值时，才移除该键的条目。  
boolean remove(Object key, Object value)  
// 只有目前将键的条目映射到某一值时，才替换该键的条目。  
V replace(K key, V value)  
// 只有目前将键的条目映射到给定值时，才替换该键的条目。  
boolean replace(K key, V oldValue, V newValue)  
// 返回此映射中的键-值映射关系数。  
int size()  
// 返回此映射中包含的值的 Collection 视图。  
Collection<V> values();

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*以下为1.8新增方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
  
/\*\*对于指定key做remappingFunction函数调用，remappingFunction函数返回值即为新的value，  
如果返回值为null，则从map中删除对应的key。compute返回key更新后的值（remappingFunction函数返回值）\*/*public V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction);  
  
*/\*\*如果指定的key不存在，对该key做mappingFunction函数操作，mappingFunction函数返回值不为null，则将对应的k-v放到map中，否则不操作。  
如果key存在返回key对应的value（此时mappingFunction不会调用）\*/*public V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction);  
*/\*\*类似与computeIfAbsent，仅对已经存在的key才计算新value。同样，如果remappingFunction返回值为null，会删除对应的k-v。\*/*public V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction);  
*/\*\*当key不存在，直接插入对应value，remappingFunction不会被调用；否则，对oldValue与value做remappingFunction函数，结果作为新的newValue插入到map中。  
 同样null结果会删除对应的k-v。\*/*public V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction);

##### 数据结构



###### 重要属性

*/\*\*存放Node元素的数组,在第一次插入数据时初始化\*/*transient volatile Node<K,V>[] table;  
  
*/\*\*一个过渡的table表 只有在扩容的时候才会使用 \*/*private transient volatile Node<K,V>[] nextTable;  
  
*/\*\*  
 \* Base counter value, used mainly when there is no contention,  
 \* but also as a fallback during table initialization  
 \* races. Updated via CAS.  
 \*/*private transient volatile long baseCount;  
  
*/\*\*  
 \* Table initialization and resizing control. When negative, the  
 \* table is being initialized or resized: -1 for initialization,  
 \* else -(1 + the number of active resizing threads). Otherwise,  
 \* when table is null, holds the initial table size to use upon  
 \* creation, or 0 for default. After initialization, holds the  
 \* next element count value upon which to resize the table.  
 \* 用来标识table的初始化和扩容操作  
 负数代表正在进行初始化或扩容操作  
 -1代表正在初始化  
 -N 表示有N-1个线程正在进行扩容操作  
 正数或0代表table还没有被初始化，这个数值表示初始化或下一次进行扩容的大小  
 如果获得了初始化权限，就用CAS方法将sizeCtl置为-1，防止其他线程进入。  
 初始化数组后，将sizeCtl的值改为0.75\*n（n - (n >>> 2)）  
 \*/*private transient volatile int sizeCtl;  
  
*/\*\*  
 \* The next table index (plus one) to split while resizing.  
 \*  
 \* 初值为最后一个桶，表示从transferIndex开始到后面所有的桶的迁移任务已经被分配出去了。  
 \* 所以每次线程领取扩容任务，则需要对该属性进行CAS的减操作，即一般是transferIndex-stride。  
 \*/*private transient volatile int transferIndex;

*/\*\*  
 \* Spinlock (locked via CAS) used when resizing and/or creating CounterCells.  
 \* 元素变化时用与锁控制  
 \*/*private transient volatile int cellsBusy;  
  
*/\*\*  
 \* Table of counter cells. When non-null, size is a power of 2.  
 \* 保存部分元素的变化个数  
 \* size = baseCount + CounterCell[i].value  
 \*/*private transient volatile CounterCell[] counterCells;

**说明:**

**table**：默认为null，初始化发生在第一次插入操作，默认大小为16的数组，用来存储Node节点数据，扩容时大小总是2的幂次方。

**nextTable**：默认为null，扩容时新生成的数组，其大小为原数组的两倍。

**sizeCtl** ：默认为0，用来控制table的初始化和扩容操作

* **-1** 代表table正在初始化
* **-N** 表示有N-1个线程正在进行扩容操作
* 正数或0代表table还没有被初始化,表示初始化或下一次扩容的大小
* 初始化数组后,将saizeCtl的值改为0.75\*n (n-(n >>> 2))

**transferIndex**：扩容时用到,初始时为table.length，表示剩余还未分配出去的扩容任务, 所以每次线程领取扩容任务后,需要对该属性进行CAS减操作(transferIndex-stride)

###### Node<K, V>

*/\*\*  
 \* Key-value entry. This class is never exported out as a  
 \* user-mutable Map.Entry (i.e., one supporting setValue; see  
 \* MapEntry below), but can be used for read-only traversals used  
 \* in bulk tasks. Subclasses of Node with a negative hash field  
 \* are special, and contain null keys and values (but are never  
 \* exported). Otherwise, keys and vals are never null.  
 \* 这个Node内部类与HashMap中定义的Node类很相似，但是有一些差别  
 它对value和next属性设置了volatile同步锁  
 \*/*static class Node<K,V> implements Map.Entry<K,V> {  
 final int hash;  
 final K key;  
 volatile V val;  
 volatile Node<K,V> next;  
  
 Node(int hash, K key, V val, Node<K,V> next) {  
 this.hash = hash;  
 this.key = key;  
 this.val = val;  
 this.next = next;  
 }  
  
 public final K getKey() { return key; }  
 public final V getValue() { return val; }  
 public final int hashCode() { return key.hashCode() ^ val.hashCode(); }  
 public final String toString(){ return key + "=" + val; }  
 //不允许直接改变value的值  
 public final V setValue(V value) {  
 throw new UnsupportedOperationException();  
 }  
  
 public final boolean equals(Object o) {  
 Object k, v, u; Map.Entry<?,?> e;  
 return ((o instanceof Map.Entry) &&  
 (k = (e = (Map.Entry<?,?>)o).getKey()) != null &&  
 (v = e.getValue()) != null &&  
 (k == key || k.equals(key)) &&  
 (v == (u = val) || v.equals(u)));  
 }  
  
 */\*\*  
 \* Virtualized support for map.get(); overridden in subclasses.  
 \* 提供find方法辅助map.get()  
 \*/* Node<K,V> find(int h, Object k) {  
 Node<K,V> e = this;  
 if (k != null) {  
 do {  
 K ek;  
 if (e.hash == h &&  
 ((ek = e.key) == k || (ek != null && k.equals(ek))))  
 return e;  
 } while ((e = e.next) != null);  
 }  
 return null;  
 }  
}

**说明:** 保存k-v,k的hash值和链表的下一个节点next,其中V和next用volatile修饰,保证多线程环境下的可见性.

###### TreeNode<K, V>

*/\*\*  
 \* Nodes for use in TreeBins  
 \* 树节点类，另外一个核心的数据结构。当链表长度过长的时候，会转换为TreeNode。  
 \* 但是与HashMap不相同的是，它并不是直接转换为红黑树，而是把这些结点包装成TreeNode放在TreeBin对象中，  
 \* 由TreeBin完成对红黑树的包装。而且TreeNode在ConcurrentHashMap集成自Node类，  
 \* 而并非HashMap中的集成自LinkedHashMap.Entry<K,V>类，也就是说TreeNode带有next指针，  
 \* 这样做的目的是方便基于TreeBin的访问。  
 \*/*static final class TreeNode<K,V> extends Node<K,V> {  
 TreeNode<K,V> parent; // red-black tree links  
 TreeNode<K,V> left;  
 TreeNode<K,V> right;  
 TreeNode<K,V> prev; // needed to unlink next upon deletion  
 boolean red;  
  
 TreeNode(int hash, K key, V val, Node<K,V> next,  
 TreeNode<K,V> parent) {  
 super(hash, key, val, next);  
 this.parent = parent;  
 }  
  
 Node<K,V> find(int h, Object k) {  
 return findTreeNode(h, k, null);  
 }  
  
 */\*\*  
 \* Returns the TreeNode (or null if not found) for the given key  
 \* starting at given root.  
 \*/* final TreeNode<K,V> findTreeNode(int h, Object k, Class<?> kc) {  
 if (k != null) {  
 TreeNode<K,V> p = this;  
 do {  
 int ph, dir; K pk; TreeNode<K,V> q;  
 TreeNode<K,V> pl = p.left, pr = p.right;  
 if ((ph = p.hash) > h)  
 p = pl;  
 else if (ph < h)  
 p = pr;  
 else if ((pk = p.key) == k || (pk != null && k.equals(pk)))  
 return p;  
 else if (pl == null)  
 p = pr;  
 else if (pr == null)  
 p = pl;  
 else if ((kc != null ||  
 (kc = *comparableClassFor*(k)) != null) &&  
 (dir = *compareComparables*(kc, k, pk)) != 0)  
 p = (dir < 0) ? pl : pr;  
 else if ((q = pr.findTreeNode(h, k, kc)) != null)  
 return q;  
 else  
 p = pl;  
 } while (p != null);  
 }  
 return null;  
 }  
}

**说明:** 红黑树节点类,当链表长度>=8且数组长度>=64时,node会转为treeNode,但它不是直接转为红黑树,而是把这些treeNode节点放入treeBin对象中,由treeBin完成红黑树的包装

###### TreeBin<K, V>

*/\*\*  
 \* TreeNodes used at the heads of bins. TreeBins do not hold user  
 \* keys or values, but instead point to list of TreeNodes and  
 \* their root. They also maintain a parasitic read-write lock  
 \* forcing writers (who hold bin lock) to wait for readers (who do  
 \* not) to complete before tree restructuring operations.  
 \*  
 \* 这个类并不负责包装用户的key、value信息，而是包装的很多TreeNode节点。  
 \* 它代替了TreeNode的根节点，也就是说在实际的ConcurrentHashMap“数组”中，  
 \* 存放的是TreeBin对象，而不是TreeNode对象，这是与HashMap的区别。另外这个类还带有了读写锁。  
 \*/*static final class TreeBin<K,V> extends Node<K,V> {  
 TreeNode<K,V> root;  
 volatile TreeNode<K,V> first;  
 volatile Thread waiter;  
 volatile int lockState;  
 // values for lockState  
 static final int *WRITER* = 1; // set while holding write lock  
 static final int *WAITER* = 2; // set when waiting for write lock  
 static final int *READER* = 4; // increment value for setting read lock

*//省略部分代码*

说明: 封装了TreeNode节点,代替了TreeNode的根节点,也就是说在concurrentHashMap中红黑树存储的是TreeBin对象,这也是与HashMap的区别之一

###### ForwardingNode<K, V>

*/\*\*  
 \* A node inserted at head of bins during transfer operations.  
 \*  
 \* 一个用于连接两个table的节点类。它包含一个nextTable指针，用于指向下一张表。  
 \* 而且这个节点的key value next指针全部为null，它的hash值为-1.  
 \* 这里面定义的find的方法是从nextTable里进行查询节点，而不是以自身为头节点进行查找  
 \* 只有在扩容时发挥作用,作为一个占位符放在table中表示当前节点为null或者已经被移动  
 \*/*static final class ForwardingNode<K,V> extends Node<K,V> {  
 final Node<K,V>[] nextTable;  
 ForwardingNode(Node<K,V>[] tab) {  
 super(*MOVED*, null, null, null);  
 this.nextTable = tab;  
 }  
  
 Node<K,V> find(int h, Object k) {  
 // loop to avoid arbitrarily deep recursion on forwarding nodes  
 outer: for (Node<K,V>[] tab = nextTable;;) {  
 Node<K,V> e; int n;  
 if (k == null || tab == null || (n = tab.length) == 0 ||  
 (e = *tabAt*(tab, (n - 1) & h)) == null)  
 return null;  
 for (;;) {  
 int eh; K ek;  
 if ((eh = e.hash) == h &&  
 ((ek = e.key) == k || (ek != null && k.equals(ek))))  
 return e;  
 if (eh < 0) {  
 if (e instanceof ForwardingNode) {  
 tab = ((ForwardingNode<K,V>)e).nextTable;  
 continue outer;  
 }  
 else  
 return e.find(h, k);  
 }  
 if ((e = e.next) == null)  
 return null;  
 }  
 }  
 }  
}

**说明:** 扩容时用到,作为一个占位符放到table中表示当前节点已经被处理

##### 源码解析

###### 构造方法

*/\*\*  
 \* Creates a new, empty map with the default initial table size (16).  
 \*/*public ConcurrentHashMap() {  
}  
  
*/\*\*  
 \* Creates a new, empty map with an initial table size  
 \* accommodating the specified number of elements without the need  
 \* to dynamically resize.  
 \*  
 \** ***@param*** *initialCapacity The implementation performs internal  
 \* sizing to accommodate this many elements.  
 \** ***@throws*** *IllegalArgumentException if the initial capacity of  
 \* elements is negative  
 \*/*public ConcurrentHashMap(int initialCapacity) {  
 if (initialCapacity < 0)  
 throw new IllegalArgumentException();  
 //初始化容量,通过tableSizeFor计算,确保为2的幂次方  
 int cap = ((initialCapacity >= (*MAXIMUM\_CAPACITY* >>> 1)) ?  
 *MAXIMUM\_CAPACITY* :  
 *tableSizeFor*(initialCapacity + (initialCapacity >>> 1) + 1));  
 this.sizeCtl = cap;  
}  
  
*/\*\*  
 \* Creates a new map with the same mappings as the given map.  
 \*  
 \** ***@param*** *m the map  
 \*/*public ConcurrentHashMap(Map<? extends K, ? extends V> m) {  
 this.sizeCtl = *DEFAULT\_CAPACITY*;  
 putAll(m);  
}  
  
*/\*\*  
 \* Creates a new, empty map with an initial table size based on  
 \* the given number of elements ({****@code*** *initialCapacity}) and  
 \* initial table density ({****@code*** *loadFactor}).  
 \*  
 \** ***@param*** *initialCapacity the initial capacity. The implementation  
 \* performs internal sizing to accommodate this many elements,  
 \* given the specified load factor.  
 \** ***@param*** *loadFactor the load factor (table density) for  
 \* establishing the initial table size  
 \** ***@throws*** *IllegalArgumentException if the initial capacity of  
 \* elements is negative or the load factor is nonpositive  
 \*  
 \** ***@since*** *1.6  
 \*/*public ConcurrentHashMap(int initialCapacity, float loadFactor) {  
 this(initialCapacity, loadFactor, 1);  
}  
  
*/\*\*  
 \* Creates a new, empty map with an initial table size based on  
 \* the given number of elements ({****@code*** *initialCapacity}), table  
 \* density ({****@code*** *loadFactor}), and number of concurrently  
 \* updating threads ({****@code*** *concurrencyLevel}).  
 \*  
 \** ***@param*** *initialCapacity the initial capacity. The implementation  
 \* performs internal sizing to accommodate this many elements,  
 \* given the specified load factor.  
 \** ***@param*** *loadFactor the load factor (table density) for  
 \* establishing the initial table size  
 \** ***@param*** *concurrencyLevel the estimated number of concurrently  
 \* updating threads. The implementation may use this value as  
 \* a sizing hint.  
 \** ***@throws*** *IllegalArgumentException if the initial capacity is  
 \* negative or the load factor or concurrencyLevel are  
 \* nonpositive  
 \*/*public ConcurrentHashMap(int initialCapacity,  
 float loadFactor, int concurrencyLevel) {  
 if (!(loadFactor > 0.0f) || initialCapacity < 0 || concurrencyLevel <= 0)  
 throw new IllegalArgumentException();  
 if (initialCapacity < concurrencyLevel) // Use at least as many bins  
 initialCapacity = concurrencyLevel; // as estimated threads  
 long size = (long)(1.0 + (long)initialCapacity / loadFactor);  
 int cap = (size >= (long)*MAXIMUM\_CAPACITY*) ?  
 *MAXIMUM\_CAPACITY* : *tableSizeFor*((int)size);  
 this.sizeCtl = cap;  
}

**说明:** 可以看到简单构造中并没有初始化table,table的初始化放到了第一次put

实例化ConcurrentHashMap时带参数时，会根据参数调整table的大小，假设参数为100，最终会调整成256，确保table的大小总是2的幂次方,算法源码:

private static final int tableSizeFor(int c) {  
 int n = c - 1;  
 n |= n >>> 1;  
 n |= n >>> 2;  
 n |= n >>> 4;  
 n |= n >>> 8;  
 n |= n >>> 16;  
 return (n < 0) ? 1 : (n >= *MAXIMUM\_CAPACITY*) ? *MAXIMUM\_CAPACITY* : n + 1;  
}

###### put(K, V)

public V put(K key, V value) {  
 return putVal(key, value, false);  
}  
  
*/\*\* Implementation for put and putIfAbsent \*/*final V putVal(K key, V value, boolean onlyIfAbsent) {  
 if (key == null || value == null) throw new NullPointerException();  
 //计算hash值  
 int hash = *spread*(key.hashCode());  
 int binCount = 0;//  
 for (Node<K,V>[] tab = table;;) {//保证拿到最新的table,然后自旋  
 //f:索引元素; n:tab.length; i:(n - 1) & hash; fh:f.hash  
 Node<K,V> f; int n, i, fh;  
 if (tab == null || (n = tab.length) == 0)  
 //初始化Node  
 tab = initTable();  
 //通过hash找到在node中的位置(tabAt),f=索引元素  
 //如果还未初始化则通过cas操作(casTabAt)插入相应数据  
 else if ((f = *tabAt*(tab, i = (n - 1) & hash)) == null) {  
 //cas插入元素,成功则跳出循环  
 if (*casTabAt*(tab, i, null,  
 new Node<K,V>(hash, key, value, null)))  
 break; // no lock when adding to empty bin  
 }  
 //当前节点处于移动状态-其他线程正在进行扩容操作  
 else if ((fh = f.hash) == *MOVED*)  
 //一起进行tab扩容  
 tab = helpTransfer(tab, f);  
 else {  
 V oldVal = null;  
 //使用内置锁synchronized控制并发  
 synchronized (f) {  
 //在节点f上进行同步，节点插入之前，再次利用tabAt(tab, i) == f判断，防止被其它线程修改。  
 if (*tabAt*(tab, i) == f) {  
 //f.hash(fh)>=0,说明f是链表的头结点  
 if (fh >= 0) {  
 binCount = 1;  
 for (Node<K,V> e = f;; ++binCount) {  
 K ek;  
 //修改  
 if (e.hash == hash &&  
 ((ek = e.key) == key ||  
 (ek != null && key.equals(ek)))) {  
 oldVal = e.val;  
 if (!onlyIfAbsent)  
 e.val = value;  
 break;  
 }  
 Node<K,V> pred = e;  
 //如果遍历到了最后一个结点，那么就证明新的节点需要插入 就把它插入在链表尾部  
 if ((e = e.next) == null) {  
 pred.next = new Node<K,V>(hash, key,  
 value, null);  
 break;  
 }  
 }  
 }  
 //红黑树节点操作  
 else if (f instanceof TreeBin) {  
 Node<K,V> p;  
 binCount = 2;  
 if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,  
 value)) != null) {  
 oldVal = p.val;  
 if (!onlyIfAbsent)  
 p.val = value;  
 }  
 }  
 }  
 }  
 if (binCount != 0) {  
 //如果链表中节点数binCount >= TREEIFY\_THRESHOLD(默认是8)，则把链表转化为红黑树结构  
 if (binCount >= *TREEIFY\_THRESHOLD*)  
 treeifyBin(tab, i);  
 if (oldVal != null)  
 return oldVal;  
 break;  
 }  
 }  
 }  
 //尝试更新新元素个数  
 addCount(1L, binCount);  
 return null;  
}

**说明:** put方法依然沿用HashMap的put方法的思想，根据hash值计算这个新插入的点在table中的位置i，如果i位置是空的，直接放进去，否则进行判断，如果i位置是树节点，按照树的方式插入新的节点，否则把i插入到链表的末尾。

ConcurrentHashMap中依然沿用这个思想，有一个最重要的不同点就是ConcurrentHashMap不允许key或value为null值。另外由于涉及到多线程，put方法就要复杂一点。在多线程中可能有以下两个情况:

4.4.4.2.1 initTable()

* 1. 初始化操作,ConcurrentHashMap会在第一次插入的时候(判断table是否为null)进行table的初始化,源码如下:

*/\*\*  
 \* Initializes table, using the size recorded in sizeCtl.  
 \*/*private final Node<K,V>[] initTable() {  
 Node<K,V>[] tab; int sc;  
 while ((tab = table) == null || tab.length == 0) {  
 //如果一个线程发现sizeCtl<0，意味着另外的线程执行CAS操作成功，当前线程只需要让出cpu时间片  
 if ((sc = sizeCtl) < 0)  
 Thread.*yield*(); // lost initialization race; just spin  
 else if (*U*.compareAndSwapInt(this, *SIZECTL*, sc, -1)) {//利用CAS方法把sizectl的值置为-1 表示本线程正在进行初始化  
 try {  
 if ((tab = table) == null || tab.length == 0) {  
 int n = (sc > 0) ? sc : *DEFAULT\_CAPACITY*;  
 @SuppressWarnings("unchecked")  
 Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];  
 table = tab = nt;  
 sc = n - (n >>> 2);//相当于0.75\*n 设置一个扩容的阈值  
 }  
 } finally {  
 sizeCtl = sc;//扩容完成后,将sizeCtl设为0.75\*n  
 }  
 break;  
 }  
 }  
 return tab;  
}

4.4.4.2.2 helpTransfer(Node<K,V>[], Node<K,V>)

* 1. 如果一个或多个线程正在对ConcurrentHashMap进行扩容操作，当前线程也要进入扩容的操作中。这个扩容的操作之所以能被检测到，是因为transfer方法中在空结点上插入forward节点，如果检测到需要插入的位置被forward节点占有，就调用helpTransfer()方法帮助进行扩容,源码如下:

final Node<K,V>[] helpTransfer(Node<K,V>[] tab, Node<K,V> f) {  
 Node<K,V>[] nextTab; int sc;  
 if (tab != null && (f instanceof ForwardingNode) &&  
 (nextTab = ((ForwardingNode<K,V>)f).nextTable) != null) {  
 //计算操作栈校验码  
 int rs = *resizeStamp*(tab.length);  
 while (nextTab == nextTable && table == tab &&  
 (sc = sizeCtl) < 0) {  
 if ((sc >>> *RESIZE\_STAMP\_SHIFT*) != rs || sc == rs + 1 ||  
 sc == rs + *MAX\_RESIZERS* || transferIndex <= 0)  
 break;  
 if (*U*.compareAndSwapInt(this, *SIZECTL*, sc, sc + 1)) {  
 transfer(tab, nextTab);  
 break;  
 }  
 }  
 return nextTab;  
 }  
 return table;  
}

4.4.4.2.3 treeifyBin(Node<K,V>[] , index)

* 1. 如果检测到要插入的节点是非空且不是forward节点，就对这个节点加锁，这样就保证了线程安全。尽管这个有一些影响效率，但是还是会比hashTable的synchronized要好得多。

整体流程就是首先利用spread方法对key的hashcode进行一次hash计算，由此来确定这个值在table中的位置。

如果这个位置是空的，那么直接放入，而且不需要加锁操作。

如果这个位置存在结点，说明发生了hash碰撞，首先判断这个节点的类型。如果是链表节点（fh>0）,则得到的结点就是hash值相同的节点组成的链表的头节点。需要依次向后遍历确定这个新加入的值所在位置。如果遇到hash值与key值都与新加入节点是一致的情况，则只需要更新value值即可。否则依次向后遍历，直到链表尾插入这个结点。如果这个节点的类型已经是树节点的话，直接调用树节点的插入方法进行插入新的值. 如果加入链表节点以后链表长度大于8，就调用treeifyBin() 方法来决定扩容还是把当前节点链表转为红黑树,源码:

private final void treeifyBin(Node<K,V>[] tab, int index) {  
 Node<K,V> b; int n, sc;  
 if (tab != null) {  
 //当数组长度还未超过64,优先数组的扩容,否则将链表转为红黑树  
 if ((n = tab.length) < *MIN\_TREEIFY\_CAPACITY*)  
 //两倍扩容  
 tryPresize(n << 1);  
 else if ((b = *tabAt*(tab, index)) != null && b.hash >= 0) {  
 synchronized (b) {  
 if (*tabAt*(tab, index) == b) {  
 TreeNode<K,V> hd = null, tl = null;  
 for (Node<K,V> e = b; e != null; e = e.next) {  
 TreeNode<K,V> p =  
 new TreeNode<K,V>(e.hash, e.key, e.val,  
 null, null);  
 if ((p.prev = tl) == null)  
 hd = p;  
 else  
 tl.next = p;  
 tl = p;  
 }  
 *setTabAt*(tab, index, new TreeBin<K,V>(hd));  
 }  
 }  
 }  
 }  
}

**说明:** 从源码中可以看出,当数组长度未超过MIN\_TREEIFY\_CAPACITY = 64时,就调用tryPresize()尝试对对table进行扩容,源码如下:

*/\*\*  
 \* Tries to presize table to accommodate the given number of elements.  
 \*  
 \** ***@param*** *size number of elements (doesn't need to be perfectly accurate)  
 \*/*private final void tryPresize(int size) {  
 int c = (size >= (*MAXIMUM\_CAPACITY* >>> 1)) ? *MAXIMUM\_CAPACITY* :  
 *tableSizeFor*(size + (size >>> 1) + 1);  
 int sc;  
 while ((sc = sizeCtl) >= 0) {  
 Node<K,V>[] tab = table; int n;  
 //未初始化  
 if (tab == null || (n = tab.length) == 0) {  
 n = (sc > c) ? sc : c;  
 if (*U*.compareAndSwapInt(this, *SIZECTL*, sc, -1)) {  
 try {  
 if (table == tab) {  
 @SuppressWarnings("unchecked")  
 Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];  
 table = nt;  
 sc = n - (n >>> 2);  
 }  
 } finally {  
 sizeCtl = sc;  
 }  
 }  
 }  
 //已达到最大容量  
 else if (c <= sc || n >= *MAXIMUM\_CAPACITY*)  
 break;  
 else if (tab == table) {  
 int rs = *resizeStamp*(n);  
 //正在进行扩容操作  
 if (sc < 0) {  
 Node<K,V>[] nt;  
 if ((sc >>> *RESIZE\_STAMP\_SHIFT*) != rs || sc == rs + 1 ||  
 sc == rs + *MAX\_RESIZERS* || (nt = nextTable) == null ||  
 transferIndex <= 0)  
 break;  
 if (*U*.compareAndSwapInt(this, *SIZECTL*, sc, sc + 1))  
 transfer(tab, nt);  
 }  
 else if (*U*.compareAndSwapInt(this, *SIZECTL*, sc,  
 (rs << *RESIZE\_STAMP\_SHIFT*) + 2))  
 transfer(tab, null);  
 }  
 }  
}

4.4.4.2.4 addCount(long,int)

* 1. 最后调用addCount(long, int)方法尝试更新元素个数.源码如下:

*/\*\*  
 \* 更新元素个数  
 \* 可能产生扩容操作(@param check>=0)  
 \* \*/*private final void addCount(long x, int check) {  
 CounterCell[] as; long b, s;  
 if ((as = counterCells) != null ||  
 !*U*.compareAndSwapLong(this, *BASECOUNT*, b = baseCount, s = b + x)) {  
 //未成功执行到CAS的线程会执行以下逻辑  
 CounterCell a; long v; int m;  
 boolean uncontended = true;  
 if (as == null || (m = as.length - 1) < 0 ||  
 (a = as[ThreadLocalRandom.*getProbe*() & m]) == null ||  
 !(uncontended =  
 *U*.compareAndSwapLong(a, *CELLVALUE*, v = a.value, v + x))) {  
 fullAddCount(x, uncontended);  
 return;  
 }  
 if (check <= 1)  
 return;  
 s = sumCount();  
 }  
 if (check >= 0) {  
 Node<K,V>[] tab, nt; int n, sc;  
 while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&  
 (n = tab.length) < *MAXIMUM\_CAPACITY*) {  
 int rs = *resizeStamp*(n);  
 if (sc < 0) {  
 if ((sc >>> *RESIZE\_STAMP\_SHIFT*) != rs || sc == rs + 1 ||  
 sc == rs + *MAX\_RESIZERS* || (nt = nextTable) == null ||  
 transferIndex <= 0)  
 break;  
 if (*U*.compareAndSwapInt(this, *SIZECTL*, sc, sc + 1))  
 transfer(tab, nt);  
 }  
 else if (*U*.compareAndSwapInt(this, *SIZECTL*, sc,  
 (rs << *RESIZE\_STAMP\_SHIFT*) + 2))  
 transfer(tab, null);  
 s = sumCount();  
 }  
 }  
}

4.4.4.2.5 transfer(Node<K,V> tab, Node<K,V> nextTab)

**发生扩容的条件:**

* 当某个Node的链表节点数>=TREEIFY\_THRESHOLD(8)时:

调用treeifyBin方法尝试把链表转为红黑树,如果此时数组长度小于MIN\_TREEIFY\_CAPACITY(64)时优先进行扩容;否则转换链表为红黑树. 源码如下:

//当数组长度还未超过64,优先数组的扩容,否则将链表转为红黑树  
if ((n = tab.length) < *MIN\_TREEIFY\_CAPACITY*)  
 //两倍扩容  
 tryPresize(n << 1);  
else if ((b = *tabAt*(tab, index)) != null && b.hash >= 0) {  
 synchronized (b) {  
 if (*tabAt*(tab, index) == b) {  
 TreeNode<K,V> hd = null, tl = null;  
 //遍历找到头节点  
 for (Node<K,V> e = b; e != null; e = e.next) {  
 TreeNode<K,V> p =  
 new TreeNode<K,V>(e.hash, e.key, e.val,  
 null, null);  
 if ((p.prev = tl) == null)  
 hd = p;  
 else  
 tl.next = p;  
 tl = p;  
 }  
 *setTabAt*(tab, index, new TreeBin<K,V>(hd));  
 }  
 }  
}

* 所有节点元素总数(通过sumCount()计算) > sizeCtl( 0.75\*n )时进行扩容操作

并且数组长度不超过MIN\_TREEIFY\_CAPACITY(64)时进行扩容操作,

整个扩容操作分为两部分:

* 1. 构建一个nextTable,它的容量是原来的两倍
  2. 将原来table中的元素复制到nextTable中，这里允许多线程进行操作。

具体源码如下:

private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {  
 int n = tab.length, stride;  
 //任务分片处理  
 if ((stride = (*NCPU* > 1) ? (n >>> 3) / *NCPU* : n) < *MIN\_TRANSFER\_STRIDE*)  
 stride = *MIN\_TRANSFER\_STRIDE*; // subdivide range  
 if (nextTab == null) { // initiating  
 try {  
 //根据当前数组长度,新建一个两倍长度的数组nextTab  
 @SuppressWarnings("unchecked")  
 Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];  
 nextTab = nt;  
 } catch (Throwable ex) { // try to cope with OOME  
 sizeCtl = Integer.*MAX\_VALUE*;  
 return;  
 }  
 nextTable = nextTab;  
 transferIndex = n;  
 }  
 int nextn = nextTab.length;  
 //初始化ForwardingNode节点,持有nextTab的引用,在处理完每个槽位的节点之后当做占位节点，表示该槽位已经处理过了；  
 ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);  
 boolean advance = true;//并发扩容的关键属性 如果等于true 说明这个节点已经处理过  
 boolean finishing = false; // to ensure sweep before committing nextTab  
 //通过for自循环处理每个槽位中的链表元素，默认advace为true，通过CAS设置transferIndex属性值，  
 // 并初始化i和bound值，i指当前处理的槽位序号，bound指需要当前线程处理的扩容任务数  
 for (int i = 0, bound = 0;;) {  
 //f:当前处理i位置的node; fh:f.hash  
 Node<K,V> f; int fh;  
 //这个while循环体的作用就是在控制i-- 通过i--可以依次遍历原hash表中的节点  
 while (advance) {  
 //nextIndex:剩余未分配的任务; nextBound:领取的扩容任务数  
 int nextIndex, nextBound;  
 if (--i >= bound || finishing)  
 advance = false;  
 //全部任务已经被分配出去  
 else if ((nextIndex = transferIndex) <= 0) {  
 i = -1;  
 advance = false;  
 }  
 //transferIndex（初值为最后一个桶），表示从transferIndex开始到后面所有的桶的迁移任务已经被分配出去了。  
 // 所以每次线程领取扩容任务，则需要对该属性进行CAS的减操作，即一般是transferIndex-stride。  
 else if (*U*.compareAndSwapInt  
 (this, *TRANSFERINDEX*, nextIndex,  
 nextBound = (nextIndex > stride ?  
 nextIndex - stride : 0))) {  
 bound = nextBound;  
 i = nextIndex - 1;  
 advance = false;  
 }  
 }  
 if (i < 0 || i >= n || i + n >= nextn) {  
 int sc;  
 if (finishing) {  
 //如果所有的节点都已经完成复制工作 就把nextTab赋值给table 清空过渡对象nextTable  
 nextTable = null;  
 table = nextTab;  
 sizeCtl = (n << 1) - (n >>> 1);//1.5\*n 扩容阈值设置为原来容量的1.5倍 依然相当于现在容量的0.75倍  
 return;  
 }  
 //参与扩容的线程未全部结束时,已经结束扩容任务的线程的处理逻辑  
 //每个线程完成自己的扩容操作后就对sizeCtl-1  
 if (*U*.compareAndSwapInt(this, *SIZECTL*, sc = sizeCtl, sc - 1)) {  
 //判断是否全部任务已经完成,sizeCtl初始值为一个负值=(rs << RESIZE\_STAMP\_SHIFT) + 2)  
 if ((sc - 2) != *resizeStamp*(n) << *RESIZE\_STAMP\_SHIFT*)  
 return;  
 finishing = advance = true;  
 i = n; // recheck before commit  
 }  
 }  
 //如果需要处理的槽位边界中没有节点，则通过CAS插入在上面初始化的ForwardingNode节点，用于告诉其它线程该槽位已经处理过了  
 else if ((f = *tabAt*(tab, i)) == null)  
 advance = *casTabAt*(tab, i, null, fwd);  
 //如果槽位边界已经被其他线程处理了，那么当前线程处理到这个节点时，取到该节点的hash值应该为MOVED，则直接跳过，继续处理下一个槽位的节点  
 else if ((fh = f.hash) == *MOVED*)  
 advance = true; // already processed  
 else {  
 synchronized (f) {  
 if (*tabAt*(tab, i) == f) {  
 //处理当前槽位的节点,构建两个node:ln/hn,分别放在低位和高位  
 //ln:原位置; hn:i+n位置  
 Node<K,V> ln, hn;  
 //如果这个位置是Node(链表)节点（fh>=0）  
 if (fh >= 0) {  
 //使用fn&n快速把链表中的元素分成两份  
 //原槽位i上的节点会分为两类，设原表长度为n=2^x，根据槽位计算公式i=(n-1)&hash，在表扩张后，  
 // 若hash在第x位是1，那么槽位就是(n-1)&hash+2^x，若hash在第x为是0，那么槽位不变。  
 int runBit = fh & n;  
 //最下一个节点  
 Node<K,V> lastRun = f;  
 //获取当前处理节点f的最下一个节点  
 for (Node<K,V> p = f.next; p != null; p = p.next) {  
 int b = p.hash & n;  
 if (b != runBit) {  
 runBit = b;  
 lastRun = p;  
 }  
 }  
 if (runBit == 0) {  
 ln = lastRun;  
 hn = null;  
 }  
 else {  
 hn = lastRun;  
 ln = null;  
 }  
 for (Node<K,V> p = f; p != lastRun; p = p.next) {  
 int ph = p.hash; K pk = p.key; V pv = p.val;  
 //在原位置  
 if ((ph & n) == 0)  
 ln = new Node<K,V>(ph, pk, pv, ln);  
 //i+n位置  
 else  
 hn = new Node<K,V>(ph, pk, pv, hn);  
 }  
 //nextTab的i位置插入一个链表  
 *setTabAt*(nextTab, i, ln);  
 //nextTab的i+n位置插入一个链表  
 *setTabAt*(nextTab, i + n, hn);  
 //在table的i位置上插入forwardNode节点 表示已经处理过该节点  
 *setTabAt*(tab, i, fwd);  
 advance = true;  
 }  
 */\*\*  
 \* 如果该槽位是红黑树结构，则构造树节点lo和hi，遍历红黑树中的节点，同样根据hash&tab.length算法，  
 \* 把节点分为两类，分别插入到lo和hi为头的链表中，根据lo和hi链表中的元素个数分别生成ln和hn节点，  
 \* 其中ln节点的生成逻辑如下：  
 \*（1）如果lo链表的元素个数小于等于UNTREEIFY\_THRESHOLD，默认为6，则通过untreeify方法把树节点链表转化成普通节点链表  
 \*（2）否则判断hi链表中的元素个数是否等于0：如果等于0，表示lo链表中包含了所有原始节点，则设置原始红黑树给ln，  
 \* 否则根据lo链表重新构造红黑树。  
 \*/* else if (f instanceof TreeBin) {  
 //转为头结点  
 TreeBin<K,V> t = (TreeBin<K,V>)f;  
 TreeNode<K,V> lo = null, loTail = null;  
 TreeNode<K,V> hi = null, hiTail = null;  
 int lc = 0, hc = 0;  
 //  
 for (Node<K,V> e = t.first; e != null; e = e.next) {  
 int h = e.hash;  
 TreeNode<K,V> p = new TreeNode<K,V>  
 (h, e.key, e.val, null, null);  
 //原位置  
 if ((h & n) == 0) {  
 if ((p.prev = loTail) == null)  
 lo = p;  
 else  
 loTail.next = p;  
 loTail = p;  
 ++lc;  
 }  
 //i+n位置  
 else {  
 if ((p.prev = hiTail) == null)  
 hi = p;  
 else  
 hiTail.next = p;  
 hiTail = p;  
 ++hc;  
 }  
 }  
 //如果扩容后已经不再需要tree的结构 反向转换为链表结构  
 ln = (lc <= *UNTREEIFY\_THRESHOLD*) ? *untreeify*(lo) :  
 (hc != 0) ? new TreeBin<K,V>(lo) : t;  
 hn = (hc <= *UNTREEIFY\_THRESHOLD*) ? *untreeify*(hi) :  
 (lc != 0) ? new TreeBin<K,V>(hi) : t;  
 *setTabAt*(nextTab, i, ln);  
 *setTabAt*(nextTab, i + n, hn);  
 *setTabAt*(tab, i, fwd);  
 advance = true;  
 }  
 }  
 }  
 }  
 }  
}

说明:

* 1. 线程是靠sizeCtl的值来感知其他线程是否也在进行扩容操作, 它初始值是一个负值=(rs << RESIZE\_STAMP\_SHIFT) + 2)，每当一个线程参与进来执行迁移工作，则该值进行CAS自增，该线程的任务执行完毕要退出时对该值进行CAS自减操作，所以当sizeCtl的值等于上述初值则说明了此时没有其他线程还在执行迁移工作，可以去执行收尾工作了
  2. 任务分片: stride的计算依赖于CPU的核数，如果只有1个核，那么此时就不用分片，即stride=n,其他情况就是 (n >>> 3) / NCPU
  3. ConcurrentHashMap含有一个属性transferIndex（初值为最后一个桶），表示从transferIndex开始到后面所有的桶的迁移任务已经被分配出去了。所以每次线程领取扩容任务，则需要对该属性进行CAS的减操作，即一般是transferIndex-stride。
  4. 线程处理分片任务时,第一个获取到分片的线程会创建一个新的Node<K,V>[],即nextTab,长度为原来的2倍,容量(sizeCtl)为原来的1.5倍(还是0.75\*n).然后遍历自己所分到的任务:



* + - 桶中元素不存在,则通过cas操作设置桶中的第一个元素为ForwardingNode,其hash值为MOVED(-1),同时该元素含有新的数组引用
    - 此时若其它线程进行put操作,发现第一个元素的hash值为-1即正在进行扩容操作（并且表明该桶已经完成扩容操作了，可以直接在新的数组中重新进行hash和插入操作），该线程就可以去参与进去，或者没有任务则不用参与，此时可以去直接操作新的数组了
    - 桶中为链表或者红黑树结构，则需要获取桶锁，防止其他线程对该桶进行put操作，然后处理方式同HashMap的处理方式一样，对桶中元素分为2类，分别代表当前桶中和要迁移到新桶中的元素。设置完毕后代表桶迁移工作已经完成，旧数组中该桶可以设置成ForwardingNode了

###### get(key)

public V get(Object key) {  
 Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;  
 //计算hash值  
 int h = *spread*(key.hashCode());  
 //通过tabAt找到元素位置  
 if ((tab = table) != null && (n = tab.length) > 0 &&  
 (e = *tabAt*(tab, (n - 1) & h)) != null) {  
 if ((eh = e.hash) == h) {  
 if ((ek = e.key) == key || (ek != null && key.equals(ek)))  
 return e.val;  
 }  
 //小于0可能为红黑树或者fwd节点  
 else if (eh < 0)  
 return (p = e.find(h, key)) != null ? p.val : null;  
 while ((e = e.next) != null) {  
 if (e.hash == h &&  
 ((ek = e.key) == key || (ek != null && key.equals(ek))))  
 return e.val;  
 }  
 }  
 return null;  
}

**说明**: 具体看源码

###### size()

*/\*\*  
 \* Base counter value, used mainly when there is no contention,  
 \* but also as a fallback during table initialization  
 \* races. Updated via CAS.  
 \*/*private transient volatile long baseCount;

*/\*\*  
 \* Table of counter cells. When non-null, size is a power of 2.  
 \* 保存部分元素的变化个数  
 \* size = baseCount + CounterCell[i].value  
 \*/*private transient volatile CounterCell[] counterCells;

public long mappingCount() {  
 long n = sumCount();  
 return (n < 0L) ? 0L : n; // ignore transient negative values  
}

public int size() {  
 long n = sumCount();  
 return ((n < 0L) ? 0 :  
 (n > (long)Integer.*MAX\_VALUE*) ? Integer.*MAX\_VALUE* :  
 (int)n);  
}

final long sumCount() {  
 CounterCell[] as = counterCells; CounterCell a;  
 long sum = baseCount;  
 if (as != null) {  
 for (int i = 0; i < as.length; ++i) {  
 if ((a = as[i]) != null)  
 sum += a.value;  
 }  
 }  
 return sum;  
}

**说明**: 对于ConcurrentHashMap来说，这个table里到底装了多少东西其实是个不确定的数量，因为不可能在调用size()方法的时候像GC的“stop the world”一样让其他线程都停下来让你去统计，因此只能说这个数量是个估计值。对于这个估计值，ConcurrentHashMap也是大费周章才计算出来的。提供了两个计算size的方法,mappingCount和size,都是调用的sumCount计算,可以看出在sumCount中,总数的计算是通过baseCount+counterCells[i].value 计算出来的.

#### ConcurrentSkipListMap

##### 概述

ConcurrentSkipListMap是线程安全的有序的哈希表，适用于高并发的场景。  
ConcurrentSkipListMap和TreeMap，它们虽然都是有序的哈希表。但是，第一，它们的线程安全机制不同，TreeMap是非线程安全的，而ConcurrentSkipListMap是线程安全的。第二，ConcurrentSkipListMap是通过跳表实现的，而TreeMap是通过红黑树实现的。  
关于跳表(Skip List)，它是平衡树的一种替代的数据结构，但是和红黑树不相同的是，跳表对于树的平衡的实现是基于一种随机化的算法的，这样也就是说跳表的插入和删除的工作是比较简单的。

##### 函数列表

// 构造一个新的空映射，该映射按照键的自然顺序进行排序。  
ConcurrentSkipListMap()  
// 构造一个新的空映射，该映射按照指定的比较器进行排序。  
ConcurrentSkipListMap(Comparator<? super K> comparator)  
// 构造一个新映射，该映射所包含的映射关系与给定映射包含的映射关系相同，并按照键的自然顺序进行排序。  
ConcurrentSkipListMap(Map<? extends K,? extends V> m)  
// 构造一个新映射，该映射所包含的映射关系与指定的有序映射包含的映射关系相同，使用的顺序也相同。  
ConcurrentSkipListMap(SortedMap<K,? extends V> m)  
  
// 返回与大于等于给定键的最小键关联的键-值映射关系；如果不存在这样的条目，则返回 null。  
Map.Entry<K,V> ceilingEntry(K key)  
// 返回大于等于给定键的最小键；如果不存在这样的键，则返回 null。  
K ceilingKey(K key)  
// 从此映射中移除所有映射关系。  
void clear()  
// 返回此 ConcurrentSkipListMap 实例的浅表副本。  
ConcurrentSkipListMap<K,V> clone()  
// 返回对此映射中的键进行排序的比较器；如果此映射使用键的自然顺序，则返回 null。  
Comparator<? super K> comparator()  
// 如果此映射包含指定键的映射关系，则返回 true。  
boolean containsKey(Object key)  
// 如果此映射为指定值映射一个或多个键，则返回 true。  
boolean containsValue(Object value)  
// 返回此映射中所包含键的逆序 NavigableSet 视图。  
NavigableSet<K> descendingKeySet()  
// 返回此映射中所包含映射关系的逆序视图。  
ConcurrentNavigableMap<K,V> descendingMap()  
// 返回此映射中所包含的映射关系的 Set 视图。  
Set<Map.Entry<K,V>> entrySet()  
// 比较指定对象与此映射的相等性。  
boolean equals(Object o)  
// 返回与此映射中的最小键关联的键-值映射关系；如果该映射为空，则返回 null。  
Map.Entry<K,V> firstEntry()  
// 返回此映射中当前第一个（最低）键。  
K firstKey()  
// 返回与小于等于给定键的最大键关联的键-值映射关系；如果不存在这样的键，则返回 null。  
Map.Entry<K,V> floorEntry(K key)  
// 返回小于等于给定键的最大键；如果不存在这样的键，则返回 null。  
K floorKey(K key)  
// 返回指定键所映射到的值；如果此映射不包含该键的映射关系，则返回 null。  
V get(Object key)  
// 返回此映射的部分视图，其键值严格小于 toKey。  
ConcurrentNavigableMap<K,V> headMap(K toKey)  
// 返回此映射的部分视图，其键小于（或等于，如果 inclusive 为 true）toKey。  
ConcurrentNavigableMap<K,V> headMap(K toKey, boolean inclusive)  
// 返回与严格大于给定键的最小键关联的键-值映射关系；如果不存在这样的键，则返回 null。  
Map.Entry<K,V> higherEntry(K key)  
// 返回严格大于给定键的最小键；如果不存在这样的键，则返回 null。  
K higherKey(K key)  
// 如果此映射未包含键-值映射关系，则返回 true。  
boolean isEmpty()  
// 返回此映射中所包含键的 NavigableSet 视图。  
NavigableSet<K> keySet()  
// 返回与此映射中的最大键关联的键-值映射关系；如果该映射为空，则返回 null。  
Map.Entry<K,V> lastEntry()  
// 返回映射中当前最后一个（最高）键。  
K lastKey()  
// 返回与严格小于给定键的最大键关联的键-值映射关系；如果不存在这样的键，则返回 null。  
Map.Entry<K,V> lowerEntry(K key)  
// 返回严格小于给定键的最大键；如果不存在这样的键，则返回 null。  
K lowerKey(K key)  
// 返回此映射中所包含键的 NavigableSet 视图。  
NavigableSet<K> navigableKeySet()  
// 移除并返回与此映射中的最小键关联的键-值映射关系；如果该映射为空，则返回 null。  
Map.Entry<K,V> pollFirstEntry()  
// 移除并返回与此映射中的最大键关联的键-值映射关系；如果该映射为空，则返回 null。  
Map.Entry<K,V> pollLastEntry()  
// 将指定值与此映射中的指定键关联。  
V put(K key, V value)  
// 如果指定键已经不再与某个值相关联，则将它与给定值关联。  
V putIfAbsent(K key, V value)  
// 从此映射中移除指定键的映射关系（如果存在）。  
V remove(Object key)  
// 只有目前将键的条目映射到给定值时，才移除该键的条目。  
boolean remove(Object key, Object value)  
// 只有目前将键的条目映射到某一值时，才替换该键的条目。  
V replace(K key, V value)  
// 只有目前将键的条目映射到给定值时，才替换该键的条目。  
boolean replace(K key, V oldValue, V newValue)  
// 返回此映射中的键-值映射关系数。  
int size()  
// 返回此映射的部分视图，其键的范围从 fromKey 到 toKey。  
ConcurrentNavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)  
// 返回此映射的部分视图，其键值的范围从 fromKey（包括）到 toKey（不包括）。  
ConcurrentNavigableMap<K,V> subMap(K fromKey, K toKey)  
// 返回此映射的部分视图，其键大于等于 fromKey。  
ConcurrentNavigableMap<K,V> tailMap(K fromKey)  
// 返回此映射的部分视图，其键大于（或等于，如果 inclusive 为 true）fromKey。  
ConcurrentNavigableMap<K,V> tailMap(K fromKey, boolean inclusive)  
// 返回此映射中所包含值的 Collection 视图。  
Collection<V> values();

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*以下为1.8新增方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
  
/\*\*对于指定key做remappingFunction函数调用，remappingFunction函数返回值即为新的value，  
如果返回值为null，则从map中删除对应的key。compute返回key更新后的值（remappingFunction函数返回值）\*/*public V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction);  
  
*/\*\*如果指定的key不存在，对该key做mappingFunction函数操作，mappingFunction函数返回值不为null，则将对应的k-v放到map中，否则不操作。  
如果key存在返回key对应的value（此时mappingFunction不会调用）\*/*public V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction);  
*/\*\*类似与computeIfAbsent，仅对已经存在的key才计算新value。同样，如果remappingFunction返回值为null，会删除对应的k-v。\*/*public V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction);  
*/\*\*当key不存在，直接插入对应value，remappingFunction不会被调用；否则，对oldValue与value做remappingFunction函数，结果作为新的newValue插入到map中。  
 同样null结果会删除对应的k-v。\*/*public V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction);

##### 数据结构







**说明**：

先以数据“7,14,21,32,37,71,85”序列为例，来对跳表进行简单说明。

跳表分为许多层(level)，每一层都可以看作是数据的索引，这些索引的意义就是加快跳表查找数据速度。每一层的数据都是有序的，上一层数据是下一层数据的子集，并且第一层(level 1)包含了全部的数据；层次越高，跳跃性越大，包含的数据越少。  
跳表包含一个表头，它查找数据时，是从上往下，从左往右进行查找。现在“需要找出值为32的节点”为例，来对比说明跳表和普遍的链表。

**情况1：链表中查找“32”节点**  
路径如下图1-02所示：

[http://images.cnitblog.com/blog/497634/201312/30222059-f01604b1b24d429e940f7a002947c143.jpg](http://images.cnitblog.com/blog/497634/201312/30222059-f01604b1b24d429e940f7a002947c143.jpg)

需要4步(红色部分表示路径)。

**情况2：跳表中查找“32”节点**  
路径如下图1-03所示：

[](http://images.cnitblog.com/blog/497634/201312/30222128-045c88b7e992443395a540ba2eb740f3.jpg)

忽略索引垂直线路上路径的情况下，只需要2步(红色部分表示路径)。

下面说说Java中ConcurrentSkipListMap的数据结构。  
(01) ConcurrentSkipListMap继承于AbstractMap类，也就意味着它是一个哈希表。  
(02) Index是ConcurrentSkipListMap的内部类，它与“跳表中的索引相对应”。HeadIndex继承于Index，ConcurrentSkipListMap中含有一个HeadIndex的对象head，head是“跳表的表头”。  
(03) Index是跳表中的索引，它包含“右索引的指针(right)”，“下索引的指针(down)”和“哈希表节点node”。node是Node的对象，Node也是ConcurrentSkipListMap中的内部类。

###### Node<K,V>

static final class Node<K,V> {  
 final K key;  
 volatile Object value;  
 volatile Node<K,V> next;  
  
 */\*\*  
 \* Creates a new regular node.  
 \*/* Node(K key, Object value, Node<K,V> next) {  
 this.key = key;  
 this.value = value;  
 this.next = next;  
 }  
  
 */\*\*  
 \* Creates a new marker node. A marker is distinguished by  
 \* having its value field point to itself. Marker nodes also  
 \* have null keys, a fact that is exploited in a few places,  
 \* but this doesn't distinguish markers from the base-level  
 \* header node (head.node), which also has a null key.  
 \*/* Node(Node<K,V> next) {  
 this.key = null;  
 this.value = this;  
 this.next = next;  
 }

//以下代码省略

}

**说明**: 表示节点数据,存储K-V,持有下一个节点的引用

###### Index<K,V>

static class Index<K,V> {  
 final Node<K, V> node;  
 final Index<K, V> down;  
 volatile Index<K, V> right;  
  
 */\*\*  
 \* Creates index node with given values.  
 \*/* Index(Node<K, V> node, Index<K, V> down, Index<K, V> right) {  
 this.node = node;  
 this.down = down;  
 this.right = right;  
 }  
 //以下代码省略  
}

**说明:** 跳表中的索引,包含了右指针(right),向下的索引down和node节点的引用

###### HeadIndex<K,V>

*/\*\*  
 \* Nodes heading each level keep track of their level.  
 \* 表示跳表的表头,内部标识了跳表的层级level  
 \*/*static final class HeadIndex<K,V> extends Index<K,V> {  
 final int level;  
 HeadIndex(Node<K,V> node, Index<K,V> down, Index<K,V> right, int level) {  
 super(node, down, right);  
 this.level = level;  
 }  
}

说明: 跳表的表头header,内部标识了当前的层级

##### 源码解析

###### 构造函数

public ConcurrentSkipListMap() {  
 this.comparator = null;  
 initialize();  
}  
  
*/\*\*  
 \* Constructs a new, empty map, sorted according to the specified  
 \* comparator.  
 \*  
 \** ***@param*** *comparator the comparator that will be used to order this map.  
 \* If {****@code*** *null}, the {****@linkplain*** *Comparable natural  
 \* ordering} of the keys will be used.  
 \*/*public ConcurrentSkipListMap(Comparator<? super K> comparator) {  
 this.comparator = comparator;  
 initialize();  
}  
  
*/\*\*  
 \* Constructs a new map containing the same mappings as the given map,  
 \* sorted according to the {****@linkplain*** *Comparable natural ordering} of  
 \* the keys.  
 \*  
 \** ***@param*** *m the map whose mappings are to be placed in this map  
 \** ***@throws*** *ClassCastException if the keys in {****@code*** *m} are not  
 \* {****@link*** *Comparable}, or are not mutually comparable  
 \** ***@throws*** *NullPointerException if the specified map or any of its keys  
 \* or values are null  
 \*/*public ConcurrentSkipListMap(Map<? extends K, ? extends V> m) {  
 this.comparator = null;  
 initialize();  
 putAll(m);  
}  
  
*/\*\*  
 \* Constructs a new map containing the same mappings and using the  
 \* same ordering as the specified sorted map.  
 \*  
 \** ***@param*** *m the sorted map whose mappings are to be placed in this  
 \* map, and whose comparator is to be used to sort this map  
 \** ***@throws*** *NullPointerException if the specified sorted map or any of  
 \* its keys or values are null  
 \*/*public ConcurrentSkipListMap(SortedMap<K, ? extends V> m) {  
 this.comparator = m.comparator();  
 initialize();  
 buildFromSorted(m);  
}  
*/\*\*  
 \* Initializes or resets state. Needed by constructors, clone,  
 \* clear, readObject. and ConcurrentSkipListSet.clone.  
 \* (Note that comparator must be separately initialized.)  
 \*/*private void initialize() {  
 keySet = null;  
 entrySet = null;  
 values = null;  
 descendingMap = null;  
 head = new HeadIndex<K,V>(new Node<K,V>(null, *BASE\_HEADER*, null),  
 null, null, 1);  
}

**说明:** 可以看到构造函数都是调用了initialize()方法,该方法在构造器、clone、clear、readObject和ConcurrentSkipListSet.clone中都有用到,注意在构造函数中,comparator需单独初始化。

###### put(K,V)

public V put(K key, V value) {  
 if (value == null)  
 throw new NullPointerException();  
 return doPut(key, value, false);  
}  
private V doPut(K key, V value, boolean onlyIfAbsent) {  
 Node<K,V> z; // added node  
 if (key == null)  
 throw new NullPointerException();  
 Comparator<? super K> cmp = comparator;  
 //自旋  
 outer: for (;;) {  
 //找到最底层(base-level)节点链中key的上一个节点(比key小(左边))的节点b,n为当前节点  
 for (Node<K,V> b = findPredecessor(key, cmp), n = b.next;;) {  
 if (n != null) {  
 Object v; int c;  
 Node<K,V> f = n.next;  
 if (n != b.next) //读不一致,跳出重试 // inconsistent read  
 break;  
 if ((v = n.value) == null) { // n is deleted  
 //帮助删除n节点  
 n.helpDelete(b, f);  
 break;  
 }  
 if (b.value == null || v == n) // b is deleted  
 break;  
 if ((c = *cpr*(cmp, key, n.key)) > 0) {//当前key大于n.key,把之前节点往后移动  
 b = n;  
 n = f;  
 continue;  
 }  
 if (c == 0) {//key相同处理  
 if (onlyIfAbsent || n.casValue(v, value)) {  
 @SuppressWarnings("unchecked") V vv = (V)v;  
 return vv;  
 }  
 break; // restart if lost race to replace value  
 }  
 // else c < 0; fall through  
 }  
 //新建一个节点  
 z = new Node<K,V>(key, value, n);  
 //cas替换  
 if (!b.casNext(n, z))  
 break; // restart if lost race to append to b  
 break outer;  
 }  
 }  
 //在跳表上层添加index  
 int rnd = ThreadLocalRandom.*nextSecondarySeed*();  
 if ((rnd & 0x80000001) == 0) { //生成的32位的rnd最高位和最低位不为1时执行(极大概率) test highest and lowest bits  
 int level = 1, max;  
 //计算跳表level  
 while (((rnd >>>= 1) & 1) != 0)//判断从右到左有多少个连续的1  
 ++level;  
 Index<K,V> idx = null;  
 HeadIndex<K,V> h = head;  
 if (level <= (max = h.level)) {  
 for (int i = 1; i <= level; ++i)  
 //从下到上依次赋值,节点持有新节点z和down节点idx的引用  
 idx = new Index<K,V>(z, idx, null);  
 }  
 else { // try to grow by one level  
 level = max + 1; // hold in array and later pick the one to use  
 //构建一个level+1长度的Index数组  
 @SuppressWarnings("unchecked")Index<K,V>[] idxs =  
 (Index<K,V>[])new Index<?,?>[level+1];  
 //从下到到上构建Index  
 for (int i = 1; i <= level; ++i)  
 idxs[i] = idx = new Index<K,V>(z, idx, null);  
 //自旋  
 for (;;) {  
 h = head;  
 //保存head之前的层级  
 int oldLevel = h.level;  
 if (level <= oldLevel) // lost race to add level  
 break;  
  
 HeadIndex<K,V> newh = h;  
 Node<K,V> oldbase = h.node;  
 //为每层生成一个头节点,每层的右节点为idxs[j]  
 for (int j = oldLevel+1; j <= level; ++j)  
 newh = new HeadIndex<K,V>(oldbase, newh, idxs[j], j);  
 //cas替换头节点  
 if (casHead(h, newh)) {  
 h = newh;  
 //idx赋值为之前层级的头结点，并将level赋值为之前的层级  
 idx = idxs[level = oldLevel];  
 break;  
 }  
 }  
 }  
 // find insertion points and splice in  
 //为每一层插入数据  
 splice: for (int insertionLevel = level;;) {  
 //获取新跳表headIndex的层级  
 int j = h.level;  
 //q:新跳表; r:q.right; t:未操作之前跳表  
 for (Index<K,V> q = h, r = q.right, t = idx;;) {  
 if (q == null || t == null)  
 break splice;  
 if (r != null) {  
 Node<K,V> n = r.node;  
 // compare before deletion check avoids needing recheck  
 int c = *cpr*(cmp, key, n.key);  
 if (n.value == null) {// 结点的值为空，表示需要删除  
 if (!q.unlink(r))// 删除q的right结点  
 break;  
 r = q.right;  
 continue;  
 }  
 if (c > 0) {// key大于结点的key,向右寻找  
 q = r;  
 r = r.right;  
 continue;  
 }  
 }  
  
 if (j == insertionLevel) {  
 if (!q.link(r, t))//r结点插入q与t之间  
 break; // restart  
 if (t.node.value == null) {  
 findNode(key);  
 break splice;  
 }  
 if (--insertionLevel == 0)//到达最底层  
 break splice;  
 }  
  
 if (--j >= insertionLevel && j < level)//移动到下一层level  
 t = t.down;  
 q = q.down;  
 r = q.right;  
 }  
 }  
 }  
 return null;

说明:put调用了doput方法,主要分两个个步骤:

1. 找到最底层(base-level)需要插入的节点位置,在最底层插入node,此处用findPredecessor方法找到最底层比给定key小(在给定节点左边)的节点,源码如下

*/\*\*  
 \* Returns a base-level node with key strictly less than given key,  
 \* or the base-level header if there is no such node. Also  
 \* unlinks indexes to deleted nodes found along the way. Callers  
 \* rely on this side-effect of clearing indices to deleted nodes.  
 \** ***@param*** *key the key  
 \** ***@return*** *a predecessor of key  
 \* 返回最底层(base-level)节点链中比给定key小(在“给定节点”左边)的节点，  
 \* 如果没找到，那么返回底层链的头节点。  
 \* 在查找过程中会帮助删除一些标记为删除的节点。  
 \*/*private Node<K,V> findPredecessor(Object key, Comparator<? super K> cmp) {  
 if (key == null)  
 throw new NullPointerException(); // don't postpone errors  
 for (;;) {  
 //q:head节点; r:右节点; d:下节点  
 for (Index<K,V> q = head, r = q.right, d;;) {  
 //右节点不为空  
 if (r != null) {  
 Node<K,V> n = r.node;  
 K k = n.key;  
 //判断r.node是否被删除  
 if (n.value == null) {  
 //如果n已经删除,尝试以CAS更新q的右节点为r.right  
 if (!q.unlink(r))  
 break; // restart  
 r = q.right; // reread r  
 continue;  
 }  
 //比较key和k  
 if (*cpr*(cmp, key, k) > 0) {  
 //继续向右循环  
 q = r;  
 r = r.right;  
 continue;  
 }  
 }  
 //right节点为空,则向下找  
 if ((d = q.down) == null)  
 //下节点为空,返回当前q节点的node  
 return q.node;  
 q = d;  
 r = d.right;  
 }  
 }  
}

1. 在跳表上层添加index
   1. 生成随机层级level,level小于最大层级直接从下到上构建index;大于最大层级时首先构建head(最上层headIndex)节点
   2. 通过双层嵌套循环为每一层添加index,这个操作中如果最上层Index(idx)的value为空,证明节点可能被删除了,就调用findNode()方法消除删除的节点,源码如下:

private Node<K,V> findNode(Object key) {  
 if (key == null)  
 throw new NullPointerException(); // don't postpone errors  
 Comparator<? super K> cmp = comparator;  
 outer: for (;;) {  
 for (Node<K,V> b = findPredecessor(key, cmp), n = b.next;;) {  
 Object v; int c;  
 if (n == null)  
 break outer;  
 Node<K,V> f = n.next;  
 if (n != b.next) // inconsistent read  
 break;  
 if ((v = n.value) == null) { // n is deleted  
 n.helpDelete(b, f);  
 break;  
 }  
 if (b.value == null || v == n) // b is deleted  
 break;  
 if ((c = *cpr*(cmp, key, n.key)) == 0)  
 return n;  
 if (c < 0)  
 break outer;  
 b = n;  
 n = f;  
 }  
 }  
 return null;  
}

###### remove(K)

public V remove(Object key) {  
 return doRemove(key, null);  
}  
final V doRemove(Object key, Object value) {  
 if (key == null)  
 throw new NullPointerException();  
 Comparator<? super K> cmp = comparator;  
 outer: for (;;) {  
 //b:比key对应的node小(左侧)的节点  
 for (Node<K,V> b = findPredecessor(key, cmp), n = b.next;;) {  
 Object v; int c;  
 if (n == null)  
 break outer;  
 Node<K,V> f = n.next;  
 if (n != b.next) // inconsistent read  
 break;  
 if ((v = n.value) == null) { // n is deleted  
 n.helpDelete(b, f);  
 break;  
 }  
 if (b.value == null || v == n) // b is deleted  
 break;  
 if ((c = *cpr*(cmp, key, n.key)) < 0)  
 break outer;  
 if (c > 0) {  
 //继续往右寻找  
 b = n;  
 n = f;  
 continue;  
 }  
 if (value != null && !value.equals(v))  
 break outer;  
 if (!n.casValue(v, null))//当前节点的value设为空  
 break;  
 //n替换为f,b的next替换为f  
 if (!n.appendMarker(f) || !b.casNext(n, f))  
 findNode(key); // retry via findNode  
 else {  
 //删除n节点对应的index  
 findPredecessor(key, cmp); // clean index  
 if (head.right == null)  
 //减少跳表层级  
 tryReduceLevel();  
 }  
 @SuppressWarnings("unchecked") V vv = (V)v;  
 return vv;  
 }  
 }  
 return null;  
}

**说明:** 从源码中可以看出,remove方法调用了doRemove(). 首先找到需要删除节点的前节点,如果中途发现有已经删除的节点,就帮助清除,然后删除base-level的node,再用findPredecessor()的辅助作用删除node节点对应的index,最后判断head是否还有right节点,没有的话就需要通过tryReduceLevel()减少跳表层级,tryReduceLevel()源码如下:

private void tryReduceLevel() {  
 HeadIndex<K,V> h = head;  
 HeadIndex<K,V> d;  
 HeadIndex<K,V> e;  
 if (h.level > 3 &&  
 (d = (HeadIndex<K,V>)h.down) != null &&  
 (e = (HeadIndex<K,V>)d.down) != null &&  
 e.right == null &&  
 d.right == null &&  
 h.right == null &&  
 casHead(h, d) && // try to set  
 h.right != null) // recheck  
 casHead(d, h); // try to backout  
}

###### get(K)

public V get(Object key) {  
 return doGet(key);  
}  
private V doGet(Object key) {  
 if (key == null)  
 throw new NullPointerException();  
 Comparator<? super K> cmp = comparator;  
 outer: for (;;) {  
 //找到最底层比key小(左边)的node  
 for (Node<K,V> b = findPredecessor(key, cmp), n = b.next;;) {  
 Object v; int c;  
 if (n == null)  
 break outer;  
 Node<K,V> f = n.next;  
 if (n != b.next) // inconsistent read  
 break;  
 if ((v = n.value) == null) { // n is deleted  
 n.helpDelete(b, f);  
 break;  
 }  
 if (b.value == null || v == n) // b is deleted  
 break;  
 if ((c = *cpr*(cmp, key, n.key)) == 0) {  
 @SuppressWarnings("unchecked") V vv = (V)v;  
 return vv;  
 }  
 if (c < 0)  
 break outer;  
 b = n;  
 n = f;  
 }  
 }  
 return null;  
}

**说明:** 很简单,看源码

###### size()

public int size() {  
 long count = 0;  
 for (Node<K,V> n = findFirst(); n != null; n = n.next) {  
 if (n.getValidValue() != null)  
 ++count;  
 }  
 return (count >= Integer.*MAX\_VALUE*) ? Integer.*MAX\_VALUE* : (int) count;  
}

**说明:** 找到首节点,依次向后遍历.用到了findFirst(),源码如下:

final Node<K,V> findFirst() {  
 for (Node<K,V> b, n;;) {  
 if ((n = (b = head.node).next) == null)  
 return null;  
 if (n.value != null)  
 return n;  
 n.helpDelete(b, n.next);  
 }  
}

#### ConcurrentSkipListSet

##### 概述

ConcurrentSkipListSet是线程安全的有序的集合，适用于高并发的场景。  
ConcurrentSkipListSet和TreeSet，它们虽然都是有序的集合。但是，

第一，它们的线程安全机制不同，TreeSet是非线程安全的，而ConcurrentSkipListSet是线程安全的。

第二，ConcurrentSkipListSet是通过ConcurrentSkipListMap实现的，而TreeSet是通过TreeMap实现的。

##### 函数列表

// 构造一个新的空 set，该 set 按照元素的自然顺序对其进行排序。  
ConcurrentSkipListSet()  
// 构造一个包含指定 collection 中元素的新 set，这个新 set 按照元素的自然顺序对其进行排序。  
ConcurrentSkipListSet(Collection<? extends E> c)  
// 构造一个新的空 set，该 set 按照指定的比较器对其元素进行排序。  
ConcurrentSkipListSet(Comparator<? super E> comparator)  
// 构造一个新 set，该 set 所包含的元素与指定的有序 set 包含的元素相同，使用的顺序也相同。  
ConcurrentSkipListSet(SortedSet<E> s)  
  
// 如果此 set 中不包含指定元素，则添加指定元素。  
boolean add(E e)  
// 返回此 set 中大于等于给定元素的最小元素；如果不存在这样的元素，则返回 null。  
E ceiling(E e)  
// 从此 set 中移除所有元素。  
void clear()  
// 返回此 ConcurrentSkipListSet 实例的浅表副本。  
ConcurrentSkipListSet<E> clone()  
// 返回对此 set 中的元素进行排序的比较器；如果此 set 使用其元素的自然顺序，则返回 null。  
Comparator<? super E> comparator()  
// 如果此 set 包含指定的元素，则返回 true。  
boolean contains(Object o)  
// 返回在此 set 的元素上以降序进行迭代的迭代器。  
Iterator<E> descendingIterator()  
// 返回此 set 中所包含元素的逆序视图。  
NavigableSet<E> descendingSet()  
// 比较指定对象与此 set 的相等性。  
boolean equals(Object o)  
// 返回此 set 中当前第一个（最低）元素。  
E first()  
// 返回此 set 中小于等于给定元素的最大元素；如果不存在这样的元素，则返回 null。  
E floor(E e)  
// 返回此 set 的部分视图，其元素严格小于 toElement。  
NavigableSet<E> headSet(E toElement)  
// 返回此 set 的部分视图，其元素小于（或等于，如果 inclusive 为 true）toElement。  
NavigableSet<E> headSet(E toElement, boolean inclusive)  
// 返回此 set 中严格大于给定元素的最小元素；如果不存在这样的元素，则返回 null。  
E higher(E e)  
// 如果此 set 不包含任何元素，则返回 true。  
boolean isEmpty()  
// 返回在此 set 的元素上以升序进行迭代的迭代器。  
Iterator<E> iterator()  
// 返回此 set 中当前最后一个（最高）元素。  
E last()  
// 返回此 set 中严格小于给定元素的最大元素；如果不存在这样的元素，则返回 null。  
E lower(E e)  
// 获取并移除第一个（最低）元素；如果此 set 为空，则返回 null。  
E pollFirst()  
// 获取并移除最后一个（最高）元素；如果此 set 为空，则返回 null。  
E pollLast()  
// 如果此 set 中存在指定的元素，则将其移除。  
boolean remove(Object o)  
// 从此 set 中移除包含在指定 collection 中的所有元素。  
boolean removeAll(Collection<?> c)  
// 返回此 set 中的元素数目。  
int size()  
// 返回此 set 的部分视图，其元素范围从 fromElement 到 toElement。  
NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)  
// 返回此 set 的部分视图，其元素从 fromElement（包括）到 toElement（不包括）。  
NavigableSet<E> subSet(E fromElement, E toElement)  
// 返回此 set 的部分视图，其元素大于等于 fromElement。  
NavigableSet<E> tailSet(E fromElement)  
// 返回此 set 的部分视图，其元素大于（或等于，如果 inclusive 为 true）fromElement。  
NavigableSet<E> tailSet(E fromElement, boolean inclusive);

##### 数据结构



**说明**：  
(01) ConcurrentSkipListSet继承于AbstractSet。因此，它本质上是一个集合。  
(02) ConcurrentSkipListSet实现了NavigableSet接口。因此，ConcurrentSkipListSet是一个有序的集合。  
(03) ConcurrentSkipListSet是通过ConcurrentSkipListMap实现的。它包含一个ConcurrentNavigableMap对象m，而m对象实际上是ConcurrentNavigableMap的实现类ConcurrentSkipListMap的实例。ConcurrentSkipListMap中的元素是key-value键值对；而ConcurrentSkipListSet是集合，它只用到了ConcurrentSkipListMap中的key

##### 源码解析

见ConcurrentSkipListMap源码

#### ArrayBlockingQueue

##### 概述

ArrayBlockingQueue是数组实现的线程安全的有界的阻塞队列。  
线程安全是指，ArrayBlockingQueue内部通过“互斥锁”保护竞争资源，实现了多线程对竞争资源的互斥访问。而有界，则是指ArrayBlockingQueue对应的数组是有界限的。 阻塞队列，是指多线程访问竞争资源时，当竞争资源已被某线程获取时，其它要获取该资源的线程需要阻塞等待；而且，ArrayBlockingQueue是按 FIFO（先进先出）原则对元素进行排序，元素都是从尾部插入到队列，从头部开始返回。

注意：ArrayBlockingQueue不同于ConcurrentLinkedQueue，ArrayBlockingQueue是数组实现的，并且是有界限的；而ConcurrentLinkedQueue是链表实现的，是无界限的。

##### 函数列表

public ArrayBlockingQueue(int capacity);  
//创建ArrayBlockingQueue,指定锁类型  
public ArrayBlockingQueue(int capacity, boolean fair);  
//指定Collection元素  
public ArrayBlockingQueue(int capacity, boolean fair, Collection<? extends E> c);  
//添加元素到尾部,队列已满抛出IllegalStateException  
public boolean add(E e);  
//添加元素到尾部,队列已满返回false  
public boolean offer(E e);  
//添加元素到尾部,队列已满等待timeout后无空间返回false  
public boolean offer(E e, long timeout, TimeUnit unit);  
//添加元素到尾部,队列已满就等待队列可用  
public void put(E e);  
//获取并移除队列头数据  
public E poll();  
//获取并移除队列头数据,等待指定时间  
public E poll(long timeout, TimeUnit unit);  
//获取并移除队列头数据,无数据一直等待  
public E take();  
//获取不移除队列都数据  
public E peek();  
//返回队列元素数量  
public int size();  
//返回剩余空间  
public int remainingCapacity();  
//移除元素  
public boolean remove(Object o);  
//返回是否包含o元素  
public boolean contains(Object o);  
//返回队列的元素数组  
public Object[] toArray();  
//返回队列的元素数组,指定类型  
public <T> T[] toArray(T[] a);  
//清空队列  
public void clear();  
//移除队列所有可用元素,并添加到c中  
public int drainTo(Collection<? super E> c);  
//移除队列中指定数量的元素,并添加到c中  
public int drainTo(Collection<? super E> c, int maxElements);

##### 数据结构



**说明:**1. 内部存储通过一个Object数组items实现2.线程安全是通过ReentrantLock和Condition实现的,内部持有ReentrantLock和两个Condition( notEmpty和notFull )的引用,分别控制队列的插入和取出,构造时也可以指定锁性质(fair或者nofair),源码如下:

//存储元素的数组  
final Object[] items;  
  
*/\*\* items index for next take, poll, peek or remove \*/*//下一个要取的元素索引  
int takeIndex;  
  
*/\*\* items index for next put, offer, or add \*/*//下一个要插入的元素索引  
int putIndex;  
  
*/\*\* Number of elements in the queue \*/*//队列中的项  
int count;  
  
/\*  
 \* Concurrency control uses the classic two-condition algorithm  
 \* found in any textbook.  
 \*/  
  
*/\*\* Main lock guarding all access \*/*final ReentrantLock lock;  
  
*/\*\* Condition for waiting takes \*/*//等待获取条件  
private final Condition notEmpty;  
  
*/\*\* Condition for waiting puts \*/*//等待插入条件  
private final Condition notFull;  
  
*/\*\*  
 \* Shared state for currently active iterators, or null if there  
 \* are known not to be any. Allows queue operations to update  
 \* iterator state.  
 \*/*//当前活动迭代器共享状态,没有的话为null.允许队列操作时更新迭代器状态  
transient Itrs itrs = null;

##### 源码解析

###### 构造函数

public ArrayBlockingQueue(int capacity) {  
 this(capacity, false);  
}  
public ArrayBlockingQueue(int capacity, boolean fair) {  
 if (capacity <= 0)  
 throw new IllegalArgumentException();  
 this.items = new Object[capacity];  
 lock = new ReentrantLock(fair);  
 notEmpty = lock.newCondition();  
 notFull = lock.newCondition();  
}  
public ArrayBlockingQueue(int capacity, boolean fair,  
 Collection<? extends E> c) {  
 this(capacity, fair);  
  
 final ReentrantLock lock = this.lock;  
 lock.lock(); // Lock only for visibility, not mutual exclusion  
 try {  
 int i = 0;  
 try {  
 for (E e : c) {  
 *checkNotNull*(e);  
 items[i++] = e;  
 }  
 } catch (ArrayIndexOutOfBoundsException ex) {  
 throw new IllegalArgumentException();  
 }  
 count = i;  
 putIndex = (i == capacity) ? 0 : i;  
 } finally {  
 lock.unlock();  
 }  
}

**说明:** 构造一个指定数组大小,指定锁类型的ArrayBlockingQueue,默认为非公平锁.

###### 添加(入列)

ArrayBlockingQueue的插入元素方法包括:add(E),offer(E),put(E),源码如下:

*/\*\*添加元素到尾部,队列已满抛出IllegalStateException\*/*public boolean add(E e) {  
 return super.add(e);  
}  
public boolean add(E e) {  
 if (offer(e))  
 return true;  
 else  
 throw new IllegalStateException("Queue full");  
}

*/\*\*添加元素到尾部,队列已满返回false\*/*public boolean offer(E e) {  
 *checkNotNull*(e);  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 if (count == items.length)  
 return false;  
 else {  
 enqueue(e);  
 return true;  
 }  
 } finally {  
 lock.unlock();  
 }  
}

*/\*\*插入元素到队列尾部\*/*private void enqueue(E x) {  
 // assert lock.getHoldCount() == 1;  
 // assert items[putIndex] == null;  
 final Object[] items = this.items;  
 items[putIndex] = x;  
 if (++putIndex == items.length)  
 putIndex = 0;  
 count++;  
 notEmpty.signal();  
}

*/\*\*添加元素到尾部,队列已满就一直等待可用\*/*public void put(E e) throws InterruptedException {  
 *checkNotNull*(e);  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 while (count == items.length)  
 notFull.await();  
 enqueue(e);  
 } finally {  
 lock.unlock();  
 }  
}

*/\*\*添加元素到队列尾部,队列已满等待timeout后无空间返回false\*/*public boolean offer(E e, long timeout, TimeUnit unit)  
 throws InterruptedException {  
  
 *checkNotNull*(e);  
 long nanos = unit.toNanos(timeout);  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 while (count == items.length) {  
 if (nanos <= 0)  
 return false;  
 nanos = notFull.awaitNanos(nanos);  
 }  
 enqueue(e);  
 return true;  
 } finally {  
 lock.unlock();  
 }  
}

**说明:** p

1. add内部调用了offer,不同的是add在队列已满的时候会抛出IllegalStateException.
2. Put插入元素时,如果队列已满,则会一直等待队列可用,并且lock是响应中断的.

###### 获取(出列)

获取元素的方法包括:poll(),take(),peek(),drainTo(Collection)源码如下:

*/\*\*获取并移除列表头元素\*/*public E poll() {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 return (count == 0) ? null : dequeue();  
 } finally {  
 lock.unlock();  
 }  
}

*/\*\*获取并移除列表头元素,无元素一直等待\*/*public E take() throws InterruptedException {  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 while (count == 0)  
 notEmpty.await();  
 return dequeue();  
 } finally {  
 lock.unlock();  
 }  
}

*/\*\*获取并移除列表头元素,无元素等待timeout后返回null\*/*public E poll(long timeout, TimeUnit unit) throws InterruptedException {  
 long nanos = unit.toNanos(timeout);  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 while (count == 0) {  
 if (nanos <= 0)  
 return null;  
 nanos = notEmpty.awaitNanos(nanos);  
 }  
 return dequeue();  
 } finally {  
 lock.unlock();  
 }  
}

*/\*\*获取但不移除头元素\*/*public E peek() {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 return itemAt(takeIndex); // null when queue is empty  
 } finally {  
 lock.unlock();  
 }  
}

**说明:**

1. poll()会获取并移除头部元素
2. take()获取并移除头部元素,在队列为空时一直等待队列可用
3. peek()获取元素,但不移除
4. 元素移除时都调用dequeue()方法,源码如下:

//出队列  
private E dequeue() {  
 // assert lock.getHoldCount() == 1;  
 // assert items[takeIndex] != null;  
 final Object[] items = this.items;  
 @SuppressWarnings("unchecked")  
 E x = (E) items[takeIndex];  
 items[takeIndex] = null;  
 if (++takeIndex == items.length)  
 takeIndex = 0;  
 count--;  
 if (itrs != null)  
 itrs.elementDequeued();  
 notFull.signal();//唤醒notFull  
 return x;  
}

###### remove(O)

*/\*\*移除o元素\*/*public boolean remove(Object o) {  
 if (o == null) return false;  
 final Object[] items = this.items;  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 if (count > 0) {  
 final int putIndex = this.putIndex;  
 int i = takeIndex;  
 do {  
 if (o.equals(items[i])) {  
 removeAt(i);  
 return true;  
 }  
 if (++i == items.length)  
 i = 0;  
 } while (i != putIndex);  
 }  
 return false;  
 } finally {  
 lock.unlock();  
 }  
}  
void removeAt(final int removeIndex) {  
 // assert lock.getHoldCount() == 1;  
 // assert items[removeIndex] != null;  
 // assert removeIndex >= 0 && removeIndex < items.length;  
 final Object[] items = this.items;  
 if (removeIndex == takeIndex) {//移除元素的索引为取的索引,直接进行出队列操作  
 // removing front item; just advance  
 items[takeIndex] = null;  
 if (++takeIndex == items.length)  
 takeIndex = 0;  
 count--;  
 if (itrs != null)  
 itrs.elementDequeued();  
 } else {  
 // an "interior" remove  
  
 // slide over all others up through putIndex.  
 final int putIndex = this.putIndex;  
 for (int i = removeIndex;;) {  
 int next = i + 1;  
 if (next == items.length)  
 next = 0;  
 if (next != putIndex) {//移除的元素不是队尾  
 items[i] = items[next];  
 i = next;  
 } else {  
 items[i] = null;  
 this.putIndex = i;  
 break;  
 }  
 }  
 count--;  
 if (itrs != null)  
 itrs.removedAt(removeIndex);  
 }  
 notFull.signal();  
}

**说明:** 找到移除对象的索引,通过removeAt(i)移除对象,移除之后释放等待notFull条件的线程

###### drainTo(Collection)

*/\*\*移除队列所有可用元素,并添加到Collection中\*/*public int drainTo(Collection<? super E> c) {  
 return drainTo(c, Integer.*MAX\_VALUE*);  
}  
*/\*\*移除队列指定数量的可用元素,并添加到Collection中\*/*public int drainTo(Collection<? super E> c, int maxElements) {  
 *checkNotNull*(c);  
 if (c == this)  
 throw new IllegalArgumentException();  
 if (maxElements <= 0)  
 return 0;  
 final Object[] items = this.items;  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 int n = Math.*min*(maxElements, count);  
 int take = takeIndex;  
 int i = 0;  
 try {  
 while (i < n) {  
 @SuppressWarnings("unchecked")  
 E x = (E) items[take];  
 c.add(x);  
 items[take] = null;  
 if (++take == items.length)  
 take = 0;  
 i++;  
 }  
 return n;  
 } finally {  
 // Restore invariants even if c.add() threw  
 //当collection添加元素出错时修复队列属性  
 if (i > 0) {  
 count -= i;  
 takeIndex = take;  
 if (itrs != null) {  
 if (count == 0)  
 itrs.queueIsEmpty();//队列置空  
 else if (i > take)  
 itrs.takeIndexWrapped();//出列索引置为0时调用,更新迭代器状态  
 }  
 for (; i > 0 && lock.hasWaiters(notFull); i--)  
 notFull.signal();  
 }  
 }  
 } finally {  
 lock.unlock();  
 }  
}

说明: 移除指定数量的可用元素,并放入指定collection中,操作完成之后循环唤醒等待notFull条件的线程.

#### LinkedBlockingQueue

##### 概述

LinkedBlockingQueue是一个单向链表实现的阻塞队列。该队列按 FIFO（先进先出）排序元素，新元素插入到队列的尾部，并且队列获取操作会获得位于队列头部的元素。链接队列的吞吐量通常要高于基于数组的队列，但是在大多数并发应用程序中，其可预知的性能要低。

此外，LinkedBlockingQueue还是可选容量的(防止过度膨胀)，即可以指定队列的容量。如果不指定，默认容量大小等于Integer.MAX\_VALUE。

##### 函数列表

public LinkedBlockingQueue() {  
 this(Integer.*MAX\_VALUE*);  
}  
public LinkedBlockingQueue(int capacity)

public LinkedBlockingQueue(Collection<? extends E> c)

//队列长度  
public int size()

//剩余空间  
public int remainingCapacity()

//尾部插入节点,队列满时会一直等待可用,响应中断  
public void put(E e)

*/\*\*插入节点,成功返回true,等待timeout时间后无空间返回false\*/*public boolean offer(E e, long timeout, TimeUnit unit)

*/\*\*插入节点\*/*public boolean offer(E e)

*/\*\*获取并消除头节点,会一直等待队列可用,响应中断\*/*public E take()

*/\*\*获取并消除头节点,等待timeout时间后无数据返回null,响应中断\*/*public E poll(long timeout, TimeUnit unit)

*/\*\*获取并消除头节点,忽略中断\*/*public E poll()

*/\*\*获取头节点,不消除\*/*public E peek()

*/\*\*移除o元素\*/*public boolean remove(Object o)

*/\*\*返回是否包含o元素\*/*public boolean contains(Object o)

*/\*\*返回元素列表(数组)\*/*public Object[] toArray()

*/\*\*返回元素列表,指定类型\*/*public <T> T[] toArray(T[] a)

*/\*\*清空队列\*/*public void clear()

*/\*\*移除指定数量的元素,并添加到collection中\*/*public int drainTo(Collection<? super E> c, int maxElements)

##### 数据结构



static class Node<E> {  
 E item;  
  
 */\*\*  
 \* One of:  
 \* - the real successor Node  
 \* - this Node, meaning the successor is head.next  
 \* - null, meaning there is no successor (this is the last node)  
 \*/* Node<E> next;  
  
 Node(E x) { item = x; }  
}  
  
*/\*\* The capacity bound, or Integer.MAX\_VALUE if none \*/*//容量  
private final int capacity;  
  
*/\*\* Current number of elements \*/*//元素个数  
private final AtomicInteger count = new AtomicInteger();  
  
*/\*\*  
 \* Head of linked list.  
 \* Invariant: head.item == null  
 \*/*//链表头  
transient Node<E> head;  
  
*/\*\*  
 \* Tail of linked list.  
 \* Invariant: last.next == null  
 \*/*//链表尾  
private transient Node<E> last;  
  
*/\*\* Lock held by take, poll, etc \*/*//出列锁  
private final ReentrantLock takeLock = new ReentrantLock();  
  
*/\*\* Wait queue for waiting takes \*/*//等待获取(出队)条件  
private final Condition notEmpty = takeLock.newCondition();  
  
*/\*\* Lock held by put, offer, etc \*/*//入列锁  
private final ReentrantLock putLock = new ReentrantLock();  
  
*/\*\* Wait queue for waiting puts \*/*//等待插入(入列)条件  
private final Condition notFull = putLock.newCondition();

**说明**：  
1. LinkedBlockingQueue继承于AbstractQueue，它本质上是一个FIFO(先进先出)的队列。  
2. LinkedBlockingQueue实现了BlockingQueue接口，它支持多线程并发。当多线程竞争同一个资源时，某线程获取到该资源之后，其它线程需要阻塞等待。  
3. LinkedBlockingQueue是通过单链表实现的。  
(01) head是链表的表头。取出数据时，都是从表头head处插入。  
(02) last是链表的表尾。新增数据时，都是从表尾last处插入。  
(03) count是链表的实际大小，即当前链表中包含的节点个数。  
(04) capacity是列表的容量，它是在创建链表时指定的。  
(05) putLock是插入锁，takeLock是取出锁；notEmpty是“非空条件”，notFull是“未满条件”。通过它们对链表进行并发控制。  
       LinkedBlockingQueue在实现“多线程对竞争资源的互斥访问”时，对于“插入”和“取出(删除)”操作分别使用了不同的锁。对于插入操作，通过“插入锁putLock”进行同步；对于取出操作，通过“取出锁takeLock”进行同步。  
       此外，插入锁putLock和“非满条件notFull”相关联，取出锁takeLock和“非空条件notEmpty”相关联。通过notFull和notEmpty更细腻的控制锁。

-- 若某线程(线程A)要取出数据时，队列正好为空，则该线程会执行notEmpty.await()进行等待；当其它某个线程(线程B)向队列中插入了数据之后，会调用notEmpty.signal()唤醒“notEmpty上的等待线程”。此时，线程A会被唤醒从而得以继续运行。 此外，线程A在执行取操作前，会获取takeLock，在取操作执行完毕再释放takeLock。

-- 若某线程(线程H)要插入数据时，队列已满，则该线程会它执行notFull.await()进行等待；当其它某个线程(线程I)取出数据之后，会调用notFull.signal()唤醒“notFull上的等待线程”。此时，线程H就会被唤醒从而得以继续运行。 此外，线程H在执行插入操作前，会获取putLock，在插入操作执行完毕才释放putLock。

##### 源码解析

###### 构造函数

public LinkedBlockingQueue() {  
 this(Integer.*MAX\_VALUE*);  
}  
public LinkedBlockingQueue(int capacity) {  
 if (capacity <= 0) throw new IllegalArgumentException();  
 this.capacity = capacity;  
 last = head = new Node<E>(null);  
}

public LinkedBlockingQueue(Collection<? extends E> c) {  
 this(Integer.*MAX\_VALUE*);  
 final ReentrantLock putLock = this.putLock;  
 putLock.lock(); // Never contended, but necessary for visibility  
 try {  
 int n = 0;  
 for (E e : c) {  
 if (e == null)  
 throw new NullPointerException();  
 if (n == capacity)  
 throw new IllegalStateException("Queue full");  
 enqueue(new Node<E>(e));  
 ++n;  
 }  
 count.set(n);  
 } finally {  
 putLock.unlock();  
 }  
}

**说明:** 链表实现,可选容量

###### 添加(入列)

插入方法包括:put(E),offer(E),源码如下:

//尾部插入节点,队列满时会一直等待可用,响应中断  
public void put(E e) throws InterruptedException {  
 if (e == null) throw new NullPointerException();  
 // Note: convention in all put/take/etc is to preset local var  
 // holding count negative to indicate failure unless set.  
 int c = -1;  
 Node<E> node = new Node<E>(e);  
 final ReentrantLock putLock = this.putLock;  
 final AtomicInteger count = this.count;  
 putLock.lockInterruptibly();  
 try {  
 /\*  
 \* Note that count is used in wait guard even though it is  
 \* not protected by lock. This works because count can  
 \* only decrease at this point (all other puts are shut  
 \* out by lock), and we (or some other waiting put) are  
 \* signalled if it ever changes from capacity. Similarly  
 \* for all other uses of count in other wait guards.  
 \*/  
 while (count.get() == capacity) {  
 notFull.await();  
 }  
 enqueue(node);  
 c = count.getAndIncrement();  
 if (c + 1 < capacity)  
 notFull.signal();  
 } finally {  
 putLock.unlock();  
 }  
 if (c == 0)  
 signalNotEmpty();  
}

*/\*\*插入节点,成功返回true,等待timeout时间后无空间返回false,响应中断\*/*public boolean offer(E e, long timeout, TimeUnit unit)  
 throws InterruptedException {  
  
 if (e == null) throw new NullPointerException();  
 long nanos = unit.toNanos(timeout);  
 int c = -1;  
 final ReentrantLock putLock = this.putLock;  
 final AtomicInteger count = this.count;  
 putLock.lockInterruptibly();  
 try {  
 while (count.get() == capacity) {  
 if (nanos <= 0)  
 return false;  
 nanos = notFull.awaitNanos(nanos);  
 }  
 enqueue(new Node<E>(e));  
 c = count.getAndIncrement();  
 if (c + 1 < capacity)  
 notFull.signal();  
 } finally {  
 putLock.unlock();  
 }  
 if (c == 0)  
 signalNotEmpty();  
 return true;  
}

*/\*\*插入节点\*/*public boolean offer(E e) {  
 if (e == null) throw new NullPointerException();  
 final AtomicInteger count = this.count;  
 if (count.get() == capacity)  
 return false;  
 int c = -1;  
 Node<E> node = new Node<E>(e);  
 final ReentrantLock putLock = this.putLock;  
 putLock.lock();  
 try {  
 if (count.get() < capacity) {  
 enqueue(node);  
 c = count.getAndIncrement();  
 if (c + 1 < capacity)  
 notFull.signal();  
 }  
 } finally {  
 putLock.unlock();  
 }  
 if (c == 0)  
 signalNotEmpty();  
 return c >= 0;  
}

**说明:** put和offer都是在队列尾插入节点,区别是:

1. Put在队列满时会一直等待队列可用,lock是响应中断的
2. Offer分了两个方法,有等待时间和无等待时间,无等待时间的在队列满时直接返回false,lock是忽略中断的.

###### 获取(出列)

获取方法包括:take(),poll(),peek().源码如下:

*/\*\*获取并消除头节点,会一直等待队列可用,响应中断\*/*public E take() throws InterruptedException {  
 E x;  
 int c = -1;  
 final AtomicInteger count = this.count;  
 final ReentrantLock takeLock = this.takeLock;  
 takeLock.lockInterruptibly();  
 try {  
 while (count.get() == 0) {  
 notEmpty.await();  
 }  
 x = dequeue();  
 c = count.getAndDecrement();  
 if (c > 1)  
 notEmpty.signal();  
 } finally {  
 takeLock.unlock();  
 }  
 if (c == capacity)  
 signalNotFull();  
 return x;  
}  
*/\*\*获取并消除头节点,等待timeout时间后无数据返回null,响应中断\*/*public E poll(long timeout, TimeUnit unit) throws InterruptedException {  
 E x = null;  
 int c = -1;  
 long nanos = unit.toNanos(timeout);  
 final AtomicInteger count = this.count;  
 final ReentrantLock takeLock = this.takeLock;  
 takeLock.lockInterruptibly();  
 try {  
 while (count.get() == 0) {  
 if (nanos <= 0)  
 return null;  
 nanos = notEmpty.awaitNanos(nanos);  
 }  
 x = dequeue();  
 c = count.getAndDecrement();  
 if (c > 1)  
 notEmpty.signal();  
 } finally {  
 takeLock.unlock();  
 }  
 if (c == capacity)  
 signalNotFull();  
 return x;  
}  
*/\*\*获取并消除头节点,忽略中断\*/*public E poll() {  
 final AtomicInteger count = this.count;  
 if (count.get() == 0)  
 return null;  
 E x = null;  
 int c = -1;  
 final ReentrantLock takeLock = this.takeLock;  
 takeLock.lock();  
 try {  
 if (count.get() > 0) {  
 x = dequeue();  
 c = count.getAndDecrement();  
 if (c > 1)  
 notEmpty.signal();  
 }  
 } finally {  
 takeLock.unlock();  
 }  
 if (c == capacity)  
 signalNotFull();  
 return x;  
}  
  
*/\*\*获取头节点,不消除\*/*public E peek() {  
 if (count.get() == 0)  
 return null;  
 final ReentrantLock takeLock = this.takeLock;  
 takeLock.lock();  
 try {  
 Node<E> first = head.next;  
 if (first == null)  
 return null;  
 else  
 return first.item;  
 } finally {  
 takeLock.unlock();  
 }  
}

**说明:**

1. take()获取并移除头部元素,在队列为空时一直等待队列可用,lock响应中断
2. poll()会获取并移除头部元素
3. peek()获取元素,但不移除

###### remove(O)

*/\*\*移除o元素\*/*public boolean remove(Object o) {  
 if (o == null) return false;  
 fullyLock();  
 try {  
 for (Node<E> trail = head, p = trail.next;  
 p != null;  
 trail = p, p = p.next) {  
 if (o.equals(p.item)) {  
 unlink(p, trail);//移除p节点  
 return true;  
 }  
 }  
 return false;  
 } finally {  
 fullyUnlock();  
 }  
}

**说明:**   
找到o元素所在的节点,移除节点调用了unlink方法,源码如下:

*/\*\*  
 \* Unlinks interior Node p with predecessor trail.  
 \*/*//移除p节点,trail:p的前节点  
void unlink(Node<E> p, Node<E> trail) {  
 // assert isFullyLocked();  
 // p.next is not changed, to allow iterators that are  
 // traversing p to maintain their weak-consistency guarantee.  
 p.item = null;  
 trail.next = p.next;  
 if (last == p)  
 last = trail;  
 if (count.getAndDecrement() == capacity)  
 notFull.signal();  
}

###### drainTo(C,int)

*/\*\*移除指定数量的元素,并添加到collection中\*/*public int drainTo(Collection<? super E> c, int maxElements) {  
 if (c == null)  
 throw new NullPointerException();  
 if (c == this)  
 throw new IllegalArgumentException();  
 if (maxElements <= 0)  
 return 0;  
 boolean signalNotFull = false;  
 final ReentrantLock takeLock = this.takeLock;  
 takeLock.lock();  
 try {  
 int n = Math.*min*(maxElements, count.get());  
 // count.get provides visibility to first n Nodes  
 Node<E> h = head;  
 int i = 0;  
 try {  
 while (i < n) {  
 Node<E> p = h.next;  
 c.add(p.item);  
 p.item = null;  
 h.next = h;  
 h = p;  
 ++i;  
 }  
 return n;  
 } finally {  
 // Restore invariants even if c.add() threw  
 if (i > 0) {  
 // assert h.item == null;  
 head = h;  
 signalNotFull = (count.getAndAdd(-i) == capacity);  
 }  
 }  
 } finally {  
 takeLock.unlock();  
 if (signalNotFull)  
 signalNotFull();  
 }  
}

很简单 不说明了

#### LinkedBlockingDeque

##### 概述

**LinkedBlockingDeque是双向链表实现的双向并发阻塞队列。该阻塞队列同时支持FIFO和FILO两种操作方式，即可以从队列的头和尾同时操作(插入/删除)；并且，该阻塞队列是支持线程安全。**

**此外，LinkedBlockingDeque还是可选容量的(防止过度膨胀)，即可以指定队列的容量。如果不指定，默认容量大小等于Integer.MAX\_VALUE。**

##### 函数列表

*/\*\*插入到队列头,队列满时抛出IllegalStateException\*/*public void addFirst(E e)  
*/\*\*插入到队列尾,队列满时抛出IllegalStateException\*/*public void addLast(E e)

*/\*\*插入到队列头,失败返回false\*/*public boolean offerFirst(E e)

*/\*\*插入到队列尾,失败返回false\*/*public boolean offerLast(E e)

*/\*\*插入到队列头,队列满时一直等待可用\*/*public void putFirst(E e) throws InterruptedException

*/\*\*插入到队列尾,队列满时一直等待可用\*/*public void putLast(E e)

*/\*\*插入队列头,等待timeout时间后若队列无空间返回false,lock响应中断\*/*public boolean offerFirst(E e, long timeout, TimeUnit unit)

*/\*\*插入队列尾,等待timeout时间后若队列无空间返回false,lock响应中断\*/*public boolean offerLast(E e, long timeout, TimeUnit unit)

*/\*\*获取并移除头节点,队列为空抛出NoSuchElementException\*/*public E removeFirst()

*/\*\*获取并移除尾节点,队列为空抛出NoSuchElementException\*/*public E removeLast()

*/\*\*获取并移除头节点,成功返回节点,失败返回null\*/*public E pollFirst()

*/\*\*获取并移除尾节点,成功返回节点,失败返回null\*/*public E pollLast()

*/\*\*获取并移除头节点,队列为空则一直等待队列可用\*/*public E takeFirst()

*/\*\*获取并移除尾节点,队列为空则一直等待队列可用\*/*public E takeLast()

*/\*\*获取并移除头节点,等待timeout时间,lock响应中断\*/*public E pollFirst(long timeout, TimeUnit unit)

*/\*\*获取并移除尾节点,等待timeout时间,lock响应中断\*/*public E pollLast(long timeout, TimeUnit unit)

*/\*\*获取头节点,不移除,队列为空抛出NoSuchElementException\*/*public E getFirst()

*/\*\*获取尾节点,不移除,队列为空抛出NoSuchElementException\*/*public E getLast()

*/\*\*获取头节点,不移除,队列为空返回null\*/*public E peekFirst()

*/\*\*获取尾节点,不移除,队列为空返回null\*/*public E peekLast()

*/\*\*从队列头开始找,移除指定元素节点\*/*public boolean removeFirstOccurrence(Object o)

*/\*\*从队列尾开始找,移除指定元素节点\*/*public boolean removeLastOccurrence(Object o)

*/\*\*插入节点到队列尾{****@link*** *#addLast}\*/*public boolean add(E e)

*/\*\*插入节点到队列尾{****@link*** *#offerLast}\*/*public boolean offer(E e)

*/\*\*插入节点到队列尾{****@link*** *#putLast}\*/*public void put(E e)

*/\*\*插入节点到队列尾{****@link*** *#offerLast}\*/*public boolean offer(E e, long timeout, TimeUnit unit)

*/\*\*获取并移除头节点{****@link*** *#removeFirst}\*/*public E remove()

*/\*\*获取并移除头节点{****@link*** *#pollFirst}\*/*public E poll()

*/\*\*获取并移除头节点{****@link*** *#takeFirst}\*/*public E take()

*/\*\*获取并移除头节点{****@link*** *#pollFirst}\*/*public E poll(long timeout, TimeUnit unit)

*/\*\*获取头节点,不移除{****@link*** *#getFirst}\*/*public E element()

*/\*\*获取头节点,不移除{****@link*** *#peekFirst}\*/*public E peek()

*/\*\*返回剩余容量\*/*public int remainingCapacity()

*/\*\*移除指定数量的元素,并放入指定collection中,返回移除成功的元素数\*/*public int drainTo(Collection<? super E> c, int maxElements)

*/\*\*插入节点到队列头{****@link*** *#addFirst}\*/*public void push(E e)

*/\*\*获取并移除队列头{****@link*** *#removeFirst}\*/*public E pop()

*/\*\*从队列头开始找,移除指定元素节点{****@link*** *#removeFirstOccurrence(Object)}\*/*public boolean remove(Object o)

*/\*\*返回队列元素数量\*/*public int size()

*/\*\*返回是否包含指定元素\*/*public boolean contains(Object o)

*/\*\*返回元素列表\*/*public Object[] toArray()

*/\*\*返回元素列表,指定类型\*/*public <T> T[] toArray(T[] a)

*/\*\*清空队列\*/*public void clear()

##### 数据结构



*/\*\* Doubly-linked list node class \*/*static final class Node<E> {  
 */\*\*  
 \* The item, or null if this node has been removed.  
 \*/* E item;  
  
 */\*\*  
 \* One of:  
 \* - the real predecessor Node  
 \* - this Node, meaning the predecessor is tail  
 \* - null, meaning there is no predecessor  
 \*/* Node<E> prev;  
  
 */\*\*  
 \* One of:  
 \* - the real successor Node  
 \* - this Node, meaning the successor is head  
 \* - null, meaning there is no successor  
 \*/* Node<E> next;  
  
 Node(E x) {  
 item = x;  
 }  
}  
  
*/\*\*  
 \* Pointer to first node.  
 \* Invariant: (first == null && last == null) ||  
 \* (first.prev == null && first.item != null)  
 \*/*transient Node<E> first;  
  
*/\*\*  
 \* Pointer to last node.  
 \* Invariant: (first == null && last == null) ||  
 \* (last.next == null && last.item != null)  
 \*/*transient Node<E> last;  
  
*/\*\* Number of items in the deque \*/*private transient int count;  
  
*/\*\* Maximum number of items in the deque \*/*private final int capacity;  
  
*/\*\* Main lock guarding all access \*/*final ReentrantLock lock = new ReentrantLock();  
  
*/\*\* Condition for waiting takes \*/*private final Condition notEmpty = lock.newCondition();  
  
*/\*\* Condition for waiting puts \*/*private final Condition notFull = lock.newCondition();

**说明**：  
1. LinkedBlockingDeque继承于AbstractQueue，它本质上是一个支持FIFO和FILO的双向的队列。  
2. LinkedBlockingDeque实现了BlockingDeque接口，它支持多线程并发。当多线程竞争同一个资源时，某线程获取到该资源之后，其它线程需要阻塞等待。  
3. LinkedBlockingDeque是通过双向链表实现的。  
first是双向链表的表头。  
last是双向链表的表尾。  
count是LinkedBlockingDeque的实际大小，即双向链表中当前节点个数。  
capacity是LinkedBlockingDeque的容量，它是在创建LinkedBlockingDeque时指定的。  
lock是控制对LinkedBlockingDeque的互斥锁，当多个线程竞争同时访问LinkedBlockingDeque时，某线程获取到了互斥锁lock，其它线程则需要阻塞等待，直到该线程释放lock，其它线程才有机会获取lock从而获取cpu执行权。  
notEmpty和notFull分别是“非空条件”和“未满条件”。通过它们能够更加细腻进行并发控制。

4. LinkedBlockingDeque包含两个迭代器Itr和DescendingItr, Itr是正向迭代,从头开始; DescendingItr与之相反.

##### 源码解析

###### 构造函数

*/\*\*  
 \* Creates a {****@code*** *LinkedBlockingDeque} with a capacity of  
 \* {****@link*** *Integer#MAX\_VALUE}.  
 \*/*public LinkedBlockingDeque() {  
 this(Integer.*MAX\_VALUE*);  
}  
  
*/\*\*  
 \* Creates a {****@code*** *LinkedBlockingDeque} with the given (fixed) capacity.  
 \*  
 \** ***@param*** *capacity the capacity of this deque  
 \** ***@throws*** *IllegalArgumentException if {****@code*** *capacity} is less than 1  
 \*/*public LinkedBlockingDeque(int capacity) {  
 if (capacity <= 0) throw new IllegalArgumentException();  
 this.capacity = capacity;  
}  
  
*/\*\*  
 \* Creates a {****@code*** *LinkedBlockingDeque} with a capacity of  
 \* {****@link*** *Integer#MAX\_VALUE}, initially containing the elements of  
 \* the given collection, added in traversal order of the  
 \* collection's iterator.  
 \*  
 \** ***@param*** *c the collection of elements to initially contain  
 \** ***@throws*** *NullPointerException if the specified collection or any  
 \* of its elements are null  
 \*/*public LinkedBlockingDeque(Collection<? extends E> c) {  
 this(Integer.*MAX\_VALUE*);  
 final ReentrantLock lock = this.lock;  
 lock.lock(); // Never contended, but necessary for visibility  
 try {  
 for (E e : c) {  
 if (e == null)  
 throw new NullPointerException();  
 if (!linkLast(new Node<E>(e)))  
 throw new IllegalStateException("Deque full");  
 }  
 } finally {  
 lock.unlock();  
 }  
}

**说明:** 可选容量

###### 添加(入列)

插入方法包括:  
addFirst(E),addLast(E),offerFirst(E),offerLast(E),putFirst(E),putLast(E),add(E),put(E),offer(E),  
push(E).

源码都是基于linkFirst(node) linkLast(node)实现,具体方法说明见函数列表.这里挑一个插入方法进行分析:

*/\*\*插入节点到队列尾{****@link*** *#addLast}\*/*public boolean add(E e) {  
 addLast(e);  
 return true;  
}

*/\*\*插入到队列尾,队列满时抛出IllegalStateException\*/*public void addLast(E e) {  
 if (!offerLast(e))  
 throw new IllegalStateException("Deque full");  
}

*/\*\*  
 \** ***@throws*** *NullPointerException {****@inheritDoc****}  
 \*/  
/\*\*插入到队列尾,失败返回false\*/*public boolean offerLast(E e) {  
 if (e == null) throw new NullPointerException();  
 Node<E> node = new Node<E>(e);  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 return linkLast(node);  
 } finally {  
 lock.unlock();  
 }  
}

*/\*\*插入节点到队列头\*/*private boolean linkFirst(Node<E> node) {  
 // assert lock.isHeldByCurrentThread();  
 if (count >= capacity)  
 return false;  
 Node<E> f = first;  
 node.next = f;  
 first = node;  
 if (last == null)  
 last = node;  
 else  
 f.prev = node;  
 ++count;  
 notEmpty.signal();  
 return true;  
}  
  
*/\*\*  
 \* Links node as last element, or returns false if full.  
 \*/  
/\*\*插入节点到队列尾\*/*private boolean linkLast(Node<E> node) {  
 // assert lock.isHeldByCurrentThread();  
 if (count >= capacity)  
 return false;  
 Node<E> l = last;  
 node.prev = l;  
 last = node;  
 if (first == null)  
 first = node;  
 else  
 l.next = node;  
 ++count;  
 notEmpty.signal();  
 return true;  
}

**说明:** 由于是双向链表,所以LinkedBlockingDeque提供了插入头和尾的方法,分别是由linkFirst和linkLast实现:

首先对队列加锁,判断队列容量是否超出,然后插入节点,修改前节点(prev)或后节点(next),最后唤醒等待notEmpty条件的线程.

###### 获取(出列)

获取方法包括:

removeFirst(),removeLast(),pollFirst(),pollLast(),takeFirst(),takeLast(),getFirst(),getLast(),peekFirst(),peekLast(),remove(),poll(),take(),element(),peek(),pop()

源码都是基于unlinkFirst()和unlinkLast()实现,具体方法说明看函数列表,这里挑一个获取方法进行说明:

*/\*\*获取并移除头节点{****@link*** *#pollFirst}\*/*public E poll(long timeout, TimeUnit unit) throws InterruptedException {  
 return pollFirst(timeout, unit);  
}

*/\*\*获取并移除头节点,等待timeout时间,lock响应中断\*/*public E pollFirst(long timeout, TimeUnit unit)  
 throws InterruptedException {  
 long nanos = unit.toNanos(timeout);  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 E x;  
 while ( (x = unlinkFirst()) == null) {  
 if (nanos <= 0)  
 return null;  
 nanos = notEmpty.awaitNanos(nanos);  
 }  
 return x;  
 } finally {  
 lock.unlock();  
 }  
}

*/\*\*移除头结点,成功返回节点,队列空时返回null\*/*private E unlinkFirst() {  
 // assert lock.isHeldByCurrentThread();  
 Node<E> f = first;  
 if (f == null)  
 return null;  
 Node<E> n = f.next;  
 E item = f.item;  
 f.item = null;  
 f.next = f; // help GC  
 first = n;  
 if (n == null)  
 last = null;  
 else  
 n.prev = null;  
 --count;  
 notFull.signal();  
 return item;  
}  
  
*/\*\*  
 \* Removes and returns last element, or null if empty.  
 \*/  
/\*\*移除尾结点,成功返回节点,队列空时返回null\*/*private E unlinkLast() {  
 // assert lock.isHeldByCurrentThread();  
 Node<E> l = last;  
 if (l == null)  
 return null;  
 Node<E> p = l.prev;  
 E item = l.item;  
 l.item = null;  
 l.prev = l; // help GC  
 last = p;  
 if (p == null)  
 first = null;  
 else  
 p.next = null;  
 --count;  
 notFull.signal();  
 return item;  
}

**说明:** 提供了获取头结点和尾节点的方法,都是由unlinkFirst和unlinkLast来实现:

首先对队列加锁,判断队列是否为空,移除头或尾节点,重新定义头或尾节点,然后唤醒等待notFull条件的线程,最后返回获取到(出列)的节点.

###### remove\*(O)

移除方法包括:  
removeFirstOccurrence(O), removeLastOccurrence(),remove(O)

public boolean remove(Object o) {  
 return removeFirstOccurrence(o);  
}

*/\*\*从队列头开始找,移除指定元素节点\*/*public boolean removeFirstOccurrence(Object o) {  
 if (o == null) return false;  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 for (Node<E> p = first; p != null; p = p.next) {  
 if (o.equals(p.item)) {  
 unlink(p);  
 return true;  
 }  
 }  
 return false;  
 } finally {  
 lock.unlock();  
 }  
}

*/\*\*移除给定节点\*/*void unlink(Node<E> x) {  
 // assert lock.isHeldByCurrentThread();  
 Node<E> p = x.prev;  
 Node<E> n = x.next;  
 if (p == null) {  
 unlinkFirst();  
 } else if (n == null) {  
 unlinkLast();  
 } else {  
 p.next = n;  
 n.prev = p;  
 x.item = null;  
 // Don't mess with x's links. They may still be in use by  
 // an iterator.  
 --count;  
 notFull.signal();  
 }  
}

**说明:** 移除给定元素所在的节点,都是由unlink(E)方法实现:  
首先对队列加锁,找到元素所在的节点,通过unlink(E)方法移除节点,最后唤醒等待notFull条件的线程.

###### drainTo(C,int)

*/\*\*移除指定数量的元素,并放入指定collection中,返回移除成功的元素数\*/*public int drainTo(Collection<? super E> c, int maxElements) {  
 if (c == null)  
 throw new NullPointerException();  
 if (c == this)  
 throw new IllegalArgumentException();  
 if (maxElements <= 0)  
 return 0;  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 int n = Math.*min*(maxElements, count);  
 for (int i = 0; i < n; i++) {  
 c.add(first.item); // In this order, in case add() throws.  
 unlinkFirst();  
 }  
 return n;  
 } finally {  
 lock.unlock();  
 }  
}

**说明:** 从头部开始移除,通过unlinkFirst()实现.

#### ConcurrentLinkedQueue

##### 概述

**ConcurrentLinkedQueue是基于单向链表的线程安全的并发队列;与LinkedBlockingQueue就是并发和阻塞的区别, ConcurrentLinkedQueue使用cas实现并发, LinkedBlockingQueue使用ReentrantLock的机制实现阻塞.适用场景:  
单生产,单消费: LinkedBlockingQueue  
多生产,单消费: LinkedBlockingQueue  
单生产,多消费: ConcurrentLinkedQueue  
单生产,多消费: ConcurrentLinkedQueue**

##### 数据结构



private static class Node<E> {  
 volatile E item;  
 volatile Node<E> next;  
  
 */\*\*  
 \* Constructs a new node. Uses relaxed write because item can  
 \* only be seen after publication via casNext.  
 \*/* Node(E item) {  
 *UNSAFE*.putObject(this, *itemOffset*, item);  
 }  
  
 boolean casItem(E cmp, E val) {  
 return *UNSAFE*.compareAndSwapObject(this, *itemOffset*, cmp, val);  
 }  
  
 void lazySetNext(Node<E> val) {  
 *UNSAFE*.putOrderedObject(this, *nextOffset*, val);  
 }  
  
 boolean casNext(Node<E> cmp, Node<E> val) {  
 return *UNSAFE*.compareAndSwapObject(this, *nextOffset*, cmp, val);  
 }  
  
 // Unsafe mechanics  
  
 private static final sun.misc.Unsafe *UNSAFE*;  
 private static final long *itemOffset*;  
 private static final long *nextOffset*;  
  
 static {  
 try {  
 *UNSAFE* = sun.misc.Unsafe.*getUnsafe*();  
 Class<?> k = Node.class;  
 *itemOffset* = *UNSAFE*.objectFieldOffset  
 (k.getDeclaredField("item"));  
 *nextOffset* = *UNSAFE*.objectFieldOffset  
 (k.getDeclaredField("next"));  
 } catch (Exception e) {  
 throw new Error(e);  
 }  
 }  
}

*/\*\*  
 \* A node from which the first live (non-deleted) node (if any)  
 \* can be reached in O(1) time.  
 \* Invariants:  
 \* - all live nodes are reachable from head via succ()  
 \* - head != null  
 \* - (tmp = head).next != tmp || tmp != head  
 \* Non-invariants:  
 \* - head.item may or may not be null.  
 \* - it is permitted for tail to lag behind head, that is, for tail  
 \* to not be reachable from head!  
 \*/  
/\*\*  
 \* 在执行方法之前和之后，head 必须保持的不变式：  
 所有“活着”的节点（指未删除节点），都能从 head 通过调用 succ() 方法遍历可达。  
 head 不能为 null。  
 head 节点的 next 域不能引用到自身。  
 在执行方法之前和之后，head 的可变式：  
 head 节点的 item 域可能为 null，也可能不为 null。  
 允许 tail 滞后（lag behind）于 head，也就是说：从 head 开始遍历队列，不一定能到达 tail。  
 \*/*private transient volatile Node<E> head;  
  
*/\*\*  
 \* A node from which the last node on list (that is, the unique  
 \* node with node.next == null) can be reached in O(1) time.  
 \* Invariants:  
 \* - the last node is always reachable from tail via succ()  
 \* - tail != null  
 \* Non-invariants:  
 \* - tail.item may or may not be null.  
 \* - it is permitted for tail to lag behind head, that is, for tail  
 \* to not be reachable from head!  
 \* - tail.next may or may not be self-pointing to tail.  
 \*/  
/\*\*  
 \* 在执行方法之前和之后，tail 必须保持的不变式：  
 通过 tail 调用 succ() 方法，最后节点总是可达的。  
 tail 不能为 null。  
 在执行方法之前和之后，tail 的可变式：  
 tail 节点的 item 域可能为 null，也可能不为 null。  
 允许 tail 滞后于 head，也就是说：从 head 开始遍历队列，不一定能到达 tail。  
 tail 节点的 next 域可以引用到自身。  
 \*/*private transient volatile Node<E> tail;

**说明：**在后面的源代码分析中，我们将会看到队列有时会处于不一致状态。为此，ConcurrentLinkedQueue 使用三个不变式 ( 基本不变式，head 的不变式和 tail 的不变式 )，来约束队列中方法的执行。通过这三个不变式来维护非阻塞算法的正确性。  
**基本不变式**

在执行方法之前和之后，队列必须要保持的不变式：

* 当入队插入新节点之后，队列中有一个 next 域为 null 的（最后）节点。
* 从 head 开始遍历队列，可以访问所有 item 域不为 null 的节点。

**head 的不变式和可变式**

在执行方法之前和之后，head 必须保持的不变式：

* 所有“活着”的节点（指未删除节点），都能从 head 通过调用 succ() 方法遍历可达。
* head 不能为 null。
* head 节点的 next 域不能引用到自身。

在执行方法之前和之后，head 的可变式：

* head 节点的 item 域可能为 null，也可能不为 null。
* 允许 tail 滞后（lag behind）于 head，也就是说：从 head 开始遍历队列，不一定能到达 tail。

**tail 的不变式和可变式**

在执行方法之前和之后，tail 必须保持的不变式：

* 通过 tail 调用 succ() 方法，最后节点总是可达的。
* tail 不能为 null。

在执行方法之前和之后，tail 的可变式：

* tail 节点的 item 域可能为 null，也可能不为 null。
* 允许 tail 滞后于 head，也就是说：从 head 开始遍历队列，不一定能到达 tail。
* tail 节点的 next 域可以引用到自身。

##### 函数列表

public ConcurrentLinkedQueue() {  
 head = tail = new Node<E>(null);  
}  
public ConcurrentLinkedQueue(Collection<? extends E> c)  
*/\*\*添加节点到队列尾,由于队列是无界的,所以不会返回false或者抛出IllegalStateException\*/*public boolean add(E e) {  
 return offer(e);  
}  
*/\*\*添加节点到队列尾\*/*public boolean offer(E e)  
*/\*\*获取队列头节点\*/*public E poll()  
*/\*\*获取队列头节点,不移除\*/*public E peek()

*/\*\*返回队列是否为空\*/*public boolean isEmpty()

*/\*\*返回队列元素个数(不一定准确)\*/*public int size()

*/\*\*返回是否包含元素o\*/*public boolean contains(Object o)

*/\*\*移除元素\*/*public boolean remove(Object o)

*/\*\*把Collection的元素添加到队列尾\*/*public boolean addAll(Collection<? extends E> c)

*/\*\*返回队列元素列表\*/*public Object[] toArray()

*/\*\*按指定类型返回队列元素列表\*/*public <T> T[] toArray(T[] a)

##### 源码解析

###### add(E)

*/\*\*添加节点到队列尾,由于队列是无界的,所以不会返回false或者抛出IllegalStateException\*/*public boolean add(E e) {  
 return offer(e);  
}  
*/\*\*添加节点到队列尾\*/*public boolean offer(E e) {  
 *checkNotNull*(e);  
 final Node<E> newNode = new Node<E>(e);  
 //自旋,t:尾节点  
 for (Node<E> t = tail, p = t;;) {  
 Node<E> q = p.next;  
 if (q == null) {//p为尾节点  
 // p is last node  
 if (p.casNext(null, newNode)) {//cas替换p的next节点为新节点  
 // 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跳两个节点时才修改tail  
 casTail(t, newNode); // Failure is OK.cas替换尾节点  
 return true;  
 }  
 // Lost CAS race to another thread; re-read next  
 }  
 else if (p == q)//p=p.next  
 // 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.  
 //如果尾节点变了,返回p=t,否则p=head,继续循环  
 p = (t != (t = tail)) ? t : head;  
 else  
 // Check for tail updates after two hops.  
 //重新寻找tail节点  
 p = (p != t && t != (t = tail)) ? t : q;  
 }  
}

**说明:** 插入节点到队列尾调用了offer(E)方法，由于队列是无界的，所以插入时不会返回false或者抛出IllegalStateException,首先找到尾节点自旋,定义q为尾节点的下一个节点,这里循环中分三种情况:  
1、q为空，证明p就是尾节点，通过cas把p的next替换为新的节点，成功后判断p!=t，这里为了尽量减少cas次数，只有在跳两个节点时才通过cas替换尾节点，成功返回true；否则证明有其他线程修改了尾节点，继续循环。  
2、p==q，由第三种情况引起，如果尾节点改变返回p=t，否则（头节点改变）返回p=head继续循环。  
3、重新寻找tail节点，如果p==t返回p=q；否则先比较t是否还是尾节点，然后重新给t赋值tail，如果t!=tail，返回p=t(tail)继续循环。

**注意：为了尽量减少cas次数，在入列时，每两次入列才调用casTail更新tail节点，这个控制的实现就是通过p!=t来实现的**

###### poll()

*/\*\*获取并移除队列头节点\*/*public E poll() {  
 restartFromHead:  
 for (;;) {  
 //h：头结点  
 for (Node<E> h = head, p = h, q;;) {  
 E item = p.item;//获取头节点元素  
  
 if (item != null && p.casItem(item, null)) {//cas修改p节点的元素为空  
 // Successful CAS is the linearization point  
 // for item to be removed from this queue.  
 if (p != h) // hop two nodes at a time弹出两个节点时才修改head  
 //cas修改head节点  
 updateHead(h, ((q = p.next) != null) ? q : p);  
 return item;  
 }  
 else if ((q = p.next) == null) {//head.next为空  
 updateHead(h, p);//cas修改head   
 return null;  
 }  
 else if (p == q)  
 continue restartFromHead;//跳转到restartFromHead重新循环  
 else  
 p = q;  
 }  
 }  
}  
*/\*\*修改头节点\*/*final void updateHead(Node<E> h, Node<E> p) {  
 if (h != p && casHead(h, p))  
 h.lazySetNext(h);//设置h的next为自己（链接到自身，做为哨兵节点方便垃圾回收）  
}  
void lazySetNext(Node<E> val) {//惰性设置 next 域的值,可能延迟执行  
 *UNSAFE*.putOrderedObject(this, *nextOffset*, val);  
}

**说明：**获取并移除队列头节点，成功返回节点元素，失败返回null。使用了两个自旋操作，分四种情况：  
1. 找到head节点，如果节点元素不为空则通过cas修改节点元素为null，成功后再利用cas修改head节点（也是在弹出两个节点时才修改head），最后返回节点元素item；  
2. head.next为空，说明队列是只有一个head的的链表，更新head后返回null；  
3. p==q,head的next指向自己，跳转到第一个自旋重新循环；  
4. 设置p=p.next继续下一次循环，到步骤1，此时p=head.next，h=head，p!=h，执行cas函数updatHead，返回item。

**注意：为了尽量减少cas次数，在出列时，每两次弹出才调用casHead更新head节点，这个控制的实现就是通过p!=h来实现的。**

###### peek()

*/\*\*获取队列头节点,不移除\*/*public E peek() {  
 restartFromHead:  
 for (;;) {  
 for (Node<E> h = head, p = h, q;;) {  
 E item = p.item;  
 if (item != null || (q = p.next) == null) {  
 updateHead(h, p);//修改头节点  
 return item;  
 }  
 else if (p == q)//head的next指向自己  
 continue restartFromHead;  
 else  
 p = q;  
 }  
 }  
}

**说明：**获取队列头节点的元素，不移除头节点。具体分析见上小节poll()

#### ConcurrentLinkedDeque

##### 概述

**ConcurrentLinkedQueue是基于双向链表的线程安全的无界并发队列; 与ConcurrentLinkedDeque的区别是该阻塞队列同时支持FIFO和FILO两种操作方式，即可以从队列的头和尾同时操作(插入/删除)**

##### 数据结构



*/\*\*  
 \* A node from which the first node on list (that is, the unique node p  
 \* with p.prev == null && p.next != p) can be reached in O(1) time.  
 \* Invariants:  
 \* - the first node is always O(1) reachable from head via prev links  
 \* - all live nodes are reachable from the first node via succ()  
 \* - head != null  
 \* - (tmp = head).next != tmp || tmp != head  
 \* - head is never gc-unlinked (but may be unlinked)  
 \* Non-invariants:  
 \* - head.item may or may not be null  
 \* - head may not be reachable from the first or last node, or from tail  
 \*/  
/\*\*  
 \* 在执行方法之前和之后，head 必须保持的不变式：  
 第一个节点总是能以O(1)的时间复杂度从head通过prev链接到达  
 所有“活着”的节点（指未删除节点），都能从第一个节点通过调用 succ() 方法遍历可达。  
 head 不能为 null。  
 head 节点的 next 域不能引用到自身。  
 head节点不会是gc-unlinked状态（但可能处于unlinked状态）  
 在执行方法之前和之后，head 的可变式：  
 head 节点的 item 域可能为 null，也可能不为 null。  
 head 节点可能从第一个或最后一个节点或 tail 节点访问时不可达  
 \*/*private transient volatile Node<E> head;  
  
*/\*\*  
 \* A node from which the last node on list (that is, the unique node p  
 \* with p.next == null && p.prev != p) can be reached in O(1) time.  
 \* Invariants:  
 \* - the last node is always O(1) reachable from tail via next links  
 \* - all live nodes are reachable from the last node via pred()  
 \* - tail != null  
 \* - tail is never gc-unlinked (but may be unlinked)  
 \* Non-invariants:  
 \* - tail.item may or may not be null  
 \* - tail may not be reachable from the first or last node, or from head  
 \*/  
/\*\*  
 \* 在执行方法之前和之后，tail 必须保持的不变式：  
 最后一个节点总是能以O(1)的时间复杂度从 tail 通过 next 链接到达  
 通过 tail 调用 succ() 方法，最后节点总是可达的。  
 tail 不能为 null。  
 tail节点不会是gc-unlinked状态（但可能处于unlinked状态）  
 在执行方法之前和之后，tail 的可变式：  
 tail 节点的 item 域可能为 null，也可能不为 null。  
 允许 tail 滞后于 head，也就是说：从 head 开始遍历队列，不一定能到达 tail。  
 \*/*private transient volatile Node<E> tail;  
*/\*\*  
 \* pre的终止节点(PREV\_TERMINATOR.next = PREV\_TERMINATOR)  
 \* next的终止节点(NEXT\_TERMINATOR.pre = NEXT\_TERMINATOR)  
 \* if(x.prev==first) x.prev=PREV\_TERMINATOR  
 \* if(x.next==last) x.next=NEXT\_TERMINATOR  
 \*/*  
private static final Node<Object> *PREV\_TERMINATOR*, *NEXT\_TERMINATOR*;  
@SuppressWarnings("unchecked")  
Node<E> prevTerminator() {  
 return (Node<E>) *PREV\_TERMINATOR*;  
}  
  
@SuppressWarnings("unchecked")  
Node<E> nextTerminator() {  
 return (Node<E>) *NEXT\_TERMINATOR*;  
}  
  
static final class Node<E> {  
 volatile Node<E> prev;  
 volatile E item;  
 volatile Node<E> next;  
  
 Node() { // default constructor for NEXT\_TERMINATOR, PREV\_TERMINATOR  
 }  
  
 */\*\*  
 \* Constructs a new node. Uses relaxed write because item can  
 \* only be seen after publication via casNext or casPrev.  
 \*/* Node(E item) {  
 *UNSAFE*.putObject(this, *itemOffset*, item);  
 }  
  
 boolean casItem(E cmp, E val) {  
 return *UNSAFE*.compareAndSwapObject(this, *itemOffset*, cmp, val);  
 }  
  
 void lazySetNext(Node<E> val) {  
 *UNSAFE*.putOrderedObject(this, *nextOffset*, val);  
 }  
  
 boolean casNext(Node<E> cmp, Node<E> val) {  
 return *UNSAFE*.compareAndSwapObject(this, *nextOffset*, cmp, val);  
 }  
  
 void lazySetPrev(Node<E> val) {  
 *UNSAFE*.putOrderedObject(this, *prevOffset*, val);  
 }  
  
 boolean casPrev(Node<E> cmp, Node<E> val) {  
 return *UNSAFE*.compareAndSwapObject(this, *prevOffset*, cmp, val);  
 }  
  
 // Unsafe mechanics  
  
 private static final sun.misc.Unsafe *UNSAFE*;  
 private static final long *prevOffset*;  
 private static final long *itemOffset*;  
 private static final long *nextOffset*;  
  
 static {  
 try {  
 *UNSAFE* = sun.misc.Unsafe.*getUnsafe*();  
 Class<?> k = Node.class;  
 *prevOffset* = *UNSAFE*.objectFieldOffset  
 (k.getDeclaredField("prev"));  
 *itemOffset* = *UNSAFE*.objectFieldOffset  
 (k.getDeclaredField("item"));  
 *nextOffset* = *UNSAFE*.objectFieldOffset  
 (k.getDeclaredField("next"));  
 } catch (Exception e) {  
 throw new Error(e);  
 }  
 }  
}  
*/\*\*更新 head/tail 的阀值\*/*private static final int *HOPS* = 2;

**说明：**1. 相对于CLQ来说，由于是双向链表组成，在迭代器上多了一个反向迭代器descendingItr；  
2. 和CLQ一样，由于在操作中队列可能处于不一致的状态，所以CLD也使用了针对head和tail的三个不变式来约束方法的执行，维护非阻塞算法的正确性。这一点在上一节的“数据结构”中已经分析，具体看本节上面源码。  
3. CLD设定了一个更新head和tail的阀值HOPS，在执行出队/入队操作时，不是每次都调用cas更新head/tail，只有从head/tail到头/尾节点的距离达到指定阀值，入队/出队操作才会更新他们。在CLQ中是通过方法控制跳两个节点后才会更新

CLD的node遵循以下规则：  
 1.live node：节点的item!=null被称为live节点，当节点的item被cas操作为null，逻辑上来讲这个节点已经从链表中移除；  
 2. first node & last node：首节点(first node)总会有一个空的prev引用，并且是prev的终止节点；同样的最后一个节点(last node)是next的终止节点。first和last节点可能是live node，也可能不是live node。  
 3. first和last节点总是相互可达的。  
 4. 一个新的元素通过cas添加到一个包含空prev或空next的first或last节点，这个元素的节点在这时是live节点。  
 5. active node：live节点、first和last节点也被称为活跃节点(active node)，活跃节点一定是被链接的，如果p节点为active节点，则：p.item != null || (p.prev == null && p.next != p) || (p.next == null && p.prev != p)  
 6. 自链接节点(self-node)：prev或last指向自己的节点，用在解除链接操作中，自链接节点都不是active node。  
 7. 节点删除时经历三个阶段：逻辑删除("logical deletion"),未链接( "unlinking"), 和gc未链接( "gc-unlinking")  
 logical deletion：移除元素时cas修改item为null，被移除元素的节点可被unlink  
 unlinking：使被删除的节点从active变的不可达，最终被GC回收。未链接节点可能从迭代器中能被读取到。节点unlinking仅仅是为了我们能在方便的时候进行操作。链表中的live节点通过prev和next保持恒等，也就是说，从first节点通过next链接找到的节点和从last节点通过prev链接找到的节点相同，但是这对被删除的节点(logically deleted)是不适用的，这种节点可能只从一个方向可达。  
 gc-unlinking：gc-unlinking使删除节点从active node不可达，更进一步解除链接，使之更容易被GC回收。gc-unlinked节点也可能从迭代器中读取到，与unlinked节点不一样的是：从head或tail查找一定不可达。  
4. first和last是双向链表的表头和表尾，它们的item可能为null，但都是活动节点(active node)；head和tail的item则不能为空，它们是链表中的有效节点(live node)，关于节点特性的说明在“数据结构”中已经分析过。

##### 函数列表

public ConcurrentLinkedDeque() {  
 head = tail = new Node<E>(null);  
}  
public ConcurrentLinkedDeque(Collection<? extends E> c) {  
 // Copy c into a private chain of Nodes  
 Node<E> h = null, t = null;  
 for (E e : c) {  
 *checkNotNull*(e);  
 Node<E> newNode = new Node<E>(e);  
 if (h == null)  
 h = t = newNode;  
 else {  
 t.lazySetNext(newNode);  
 newNode.lazySetPrev(t);  
 t = newNode;  
 }  
 }  
 initHeadTail(h, t);  
}

*/\*\*添加元素到队列头\*/*public void addFirst(E e)

*/\*\*添加元素到队列尾\*/*public void addLast(E e)

*/\*\*添加元素到队列头，成功返回true\*/*public boolean offerFirst(E e)

*/\*\*添加元素到队列尾，成功返回true\*/*public boolean offerLast(E e)

*/\*\*获取队列首节点\*/*public E peekFirst()

*/\*\*获取队列尾节点\*/*public E peekLast()

*/\*\*获取队列首节点，失败抛出NoSuchElementException\*/*public E getFirst()

*/\*\*获取队列尾节点，失败抛出NoSuchElementException\*/*public E getLast()

*/\*\*获取并移除队列首节点\*/*public E pollFirst()

*/\*\*获取并移除队列尾节点\*/*public E pollLast()

*/\*\*获取并移除队列首节点，NoSuchElementException\*/*public E removeFirst()

*/\*\*获取并移除队列尾节点，NoSuchElementException\*/*public E removeLast()

*/\*\*添加节点到队列尾{****@link*** *Queue#offer}\*/*public boolean offer(E e)

*/\*\*添加节点到队列尾{****@link*** *Collection#add}\*/*public boolean add(E e)

*/\*\*获取并移除队列头节点\*/*public E poll() { return pollFirst(); }  
*/\*\*获取队列头节点，不移除\*/*public E peek() { return peekFirst(); }

*/\*\*获取并移除队列头节点，失败抛出NoSuchElementException\*/*public E remove() { return removeFirst(); }

*/\*\*获取并移除队列头节点，失败抛出NoSuchElementException\*/*public E pop() { return removeFirst(); }

*/\*\*获取队列头节点，不移除\*/*public E element() { return getFirst(); }

*/\*\*添加节点到队列头\*/*public void push(E e) { addFirst(e); }

*/\*\*从队列头开始寻找，移除指定元素所在的节点\*/*public boolean removeFirstOccurrence(Object o)

*/\*\*从队列尾开始向前寻找，移除指定元素所在的节点\*/*public boolean removeLastOccurrence(Object o)

*/\*\*返回是否包含指定元素\*/*public boolean contains(Object o)

*/\*\*返回是否为空队列\*/*public boolean isEmpty() {  
 return peekFirst() == null;  
}

*/\*\*返回队列元素数量，不一定准确\*/*public int size()

*/\*\*从队列头开始寻找，移除指定元素\*/*public boolean remove(Object o) {  
 return removeFirstOccurrence(o);  
}

*/\*\*添加指定Collection到队列中\*/*public boolean addAll(Collection<? extends E> c)

*/\*\*清空队列\*/*public void clear() {  
 while (pollFirst() != null)  
 ;  
}

*/\*\*返回队列的所有元素数组\*/*public Object[] toArray() {  
 return toArrayList().toArray();  
}

*/\*\*返回队列的所有元素数组，指定元素类型\*/*public <T> T[] toArray(T[] a) {  
 return toArrayList().toArray(a);  
}

*/\*\*正序迭代器\*/*public Iterator<E> iterator() {  
 return new Itr();  
}

*/\*\*反序迭代器\*/*public Iterator<E> descendingIterator() {  
 return new DescendingItr();  
}

##### 源码解析

###### 添加(入列)

CLD的添加方法包括：offer(E)、add(E)、push(E)、0addFirst(E)、addLast(E)、offerFirst(E)、offerLast(E)，入列操作都是通过linkFirst(E)和linkLast(E)来实现的，这里挑一个入列方法进行解析：

*/\*\*添加节点到队列尾{****@link*** *Collection#add}\*/*public boolean add(E e) {  
 return offerLast(e);  
}

*/\*\*添加元素到队列尾，成功返回true\*/*public boolean offerLast(E e) {  
 linkLast(e);  
 return true;  
}  
*/\*\*  
 \* Inserts the specified element at the front of this deque.  
 \* As the deque is unbounded, this method will never return {****@code*** *false}.  
 \*  
 \** ***@return*** *{****@code*** *true} (as specified by {****@link*** *Deque#offerFirst})  
 \** ***@throws*** *NullPointerException if the specified element is null  
 \*/  
/\*\*添加元素到队列头，成功返回true\*/*public boolean offerFirst(E e) {  
 linkFirst(e);  
 return true;  
}

**说明：**CLD的添加方法都依赖于linkFirst(E)和linkLast(E)，分别是添加到队列头和队列尾，源码如下：

*/\*\*  
 \* Links e as first element.  
 \*/  
/\*\*入列，插入到队列头\*/*private void linkFirst(E e) {  
 *checkNotNull*(e);  
 final Node<E> newNode = new Node<E>(e);  
  
 restartFromHead:  
 for (;;)  
 //从head节点往前寻找first节点  
 for (Node<E> h = head, p = h, q;;) {  
 if ((q = p.prev) != null &&  
 (q = (p = q).prev) != null)  
 // Check for head updates every other hop.  
 // If p == q, we are sure to follow head instead.  
 p = (h != (h = head)) ? h : q;  
 else if (p.next == p) // PREV\_TERMINATOR  
 continue restartFromHead;  
 else {  
 // p is first node  
 newNode.lazySetNext(p); // CAS piggyback  
 if (p.casPrev(null, newNode)) {  
 // Successful CAS is the linearization point  
 // for e to become an element of this deque,  
 // and for newNode to become "live".  
 if (p != h) // hop two nodes at a time 跳两个节点时才修改head  
 casHead(h, newNode); // Failure is OK.  
 return;  
 }  
 // Lost CAS race to another thread; re-read prev  
 }  
 }  
}

*/\*\*  
 \* Links e as last element.  
 \*/  
/\*\*入列，插入到队列尾\*/*private void linkLast(E e) {  
 *checkNotNull*(e);  
 final Node<E> newNode = new Node<E>(e);  
  
 restartFromTail:  
 for (;;)  
 //从tail节点往后寻找last节点  
 for (Node<E> t = tail, p = t, q;;) {  
 if ((q = p.next) != null &&  
 (q = (p = q).next) != null)  
 // Check for tail updates every other hop.  
 // If p == q, we are sure to follow tail instead.  
 p = (t != (t = tail)) ? t : q;  
 else if (p.prev == p) // NEXT\_TERMINATOR  
 continue restartFromTail;  
 else {  
 // p is last node  
 newNode.lazySetPrev(p); // CAS piggyback  
 if (p.casNext(null, newNode)) {  
 // Successful CAS is the linearization point  
 // for e to become an element of this deque,  
 // and for newNode to become "live".  
 if (p != t) // hop two nodes at a time 跳两个节点时才修改tail  
 casTail(t, newNode); // Failure is OK.  
 return;  
 }  
 // Lost CAS race to another thread; re-read next  
 }  
 }  
}

**说明：**linkFirst是插入新节点到队列头，首先从head节点开始向前循环找到first节点(p.prev==null&&p.next!=p)；然后通过lazySetNext设置新节点的next节点为first；然后cas修改first的prev为新节点。注意这里cas指令成功后会判断first节点是否已经跳了两个节点，只有在跳了两个节点才会cas更新head。linkLast跟linkFirst不同的是插入新节点到队列尾，在此不做说明，具体见源码。  
 LazySetNext调用了UNSAFE类的putOrderedObject方法：源码如下：

void lazySetNext(Node<E> val) {  
 *UNSAFE*.putOrderedObject(this, *nextOffset*, val);  
}

Oracle的JDK中提供了Unsafe. putOrderedObject，Unsafe. putOrderedInt，Unsafe. putOrderedLong这三个方法，JDK会在执行这三个方法时插入StoreStore内存屏障，避免发生写操作重排序。而在Intel 64/IA-32架构下，StoreStore屏障并不需要，Java编译器会将StoreStore屏障去除。比起写入volatile变量之后执行StoreLoad屏障的巨大开销，采用这种方法除了避免重排序而带来的性能损失以外，不会带来其它的性能开销

###### 获取(出列)

CLD的出列方法有：peekFirst 、peekLast、getFirst、getLast、pollFirst、pollLast、removeFirst、removeLast、poll、peek、remove、pop，这里挑一种出列方法进行分析：

*/\*\*  
 \** ***@throws*** *NoSuchElementException {****@inheritDoc****}  
 \*/  
/\*\*获取并移除队列头节点，失败抛出NoSuchElementException\*/*public E pop() { return removeFirst(); }

*/\*\*  
 \** ***@throws*** *NoSuchElementException {****@inheritDoc****}  
 \*/  
/\*\*获取并移除队列首节点，NoSuchElementException\*/*public E removeFirst() {  
 return screenNullResult(pollFirst());  
}

*/\*\*获取并移除队列首节点\*/*public E pollFirst() {  
 for (Node<E> p = first(); p != null; p = succ(p)) {  
 E item = p.item;  
 if (item != null && p.casItem(item, null)) {  
 unlink(p);  
 return item;  
 }  
 }  
 return null;  
}

**说明：**（1）首先通过first()方法找到first节点，first节点必须为活跃节点(p.prev==null&&p.next!=p)  
（2）如果first.item==null（这里是允许的，具体见“数据结构”小节的介绍），则继续调用succ方法继续寻找下一个节点。  
first()和succ(p)方法源码如下：

*/\*\*  
 \* Returns the first node, the unique node p for which:  
 \* p.prev == null && p.next != p  
 \* The returned node may or may not be logically deleted.  
 \* Guarantees that head is set to the returned node.  
 \*/  
/\*\*返回首节点\*/*Node<E> first() {  
 restartFromHead:  
 for (;;)  
 //从head开始往前找  
 for (Node<E> h = head, p = h, q;;) {  
 if ((q = p.prev) != null &&  
 (q = (p = q).prev) != null)  
 // Check for head updates every other hop.  
 // If p == q, we are sure to follow head instead.  
 //如果head被其他修改返回新的head否则返回q继续往前循环寻找  
 p = (h != (h = head)) ? h : q;  
 else if (p == h  
 // It is possible that p is PREV\_TERMINATOR,  
 // but if so, the CAS is guaranteed to fail.  
 //p!=h需要cas替换head  
 || casHead(h, p))  
 return p;  
 else  
 continue restartFromHead;  
 }  
}

*/\*\*  
 \* Returns the successor of p, or the first node if p.next has been  
 \* linked to self, which will only be true if traversing with a  
 \* stale pointer that is now off the list.  
 \*/  
/\*\*返回指定节点的的后继节点，如果指定节点的next指向自己，返回first节点\*/*final Node<E> succ(Node<E> p) {  
 // *TODO: should we skip deleted nodes here?* Node<E> q = p.next;  
 return (p == q) ? first() : q;  
}

（3）item不为null则用cas修改first节点的item为null，然后调用unlink(p)方法解除节点链接。源码如下：

*/\*\*  
 \* Unlinks non-null node x.  
 \*/  
/\*\*移除给定节点\*/*void unlink(Node<E> x) {  
 // assert x != null;  
 // assert x.item == null;  
 // assert x != PREV\_TERMINATOR;  
 // assert x != NEXT\_TERMINATOR;  
  
 final Node<E> prev = x.prev;  
 final Node<E> next = x.next;  
 if (prev == null) {  
 //如果为操作节点为first  
 unlinkFirst(x, next);  
 } else if (next == null) {  
 unlinkLast(x, prev);  
 } else {  
 // Unlink interior node.  
 //  
 // This is the common case, since a series of polls at the  
 // same end will be "interior" removes, except perhaps for  
 // the first one, since end nodes cannot be unlinked.  
 //  
 // At any time, all active nodes are mutually reachable by  
 // following a sequence of either next or prev pointers.  
 //  
 // Our strategy is to find the unique active predecessor  
 // and successor of x. Try to fix up their links so that  
 // they point to each other, leaving x unreachable from  
 // active nodes. If successful, and if x has no live  
 // predecessor/successor, we additionally try to gc-unlink,  
 // leaving active nodes unreachable from x, by rechecking  
 // that the status of predecessor and successor are  
 // unchanged and ensuring that x is not reachable from  
 // tail/head, before setting x's prev/next links to their  
 // logical approximate replacements, self/TERMINATOR.  
 Node<E> activePred, activeSucc;  
 boolean isFirst, isLast;  
 int hops = 1;  
  
 // Find active predecessor  
 //从被操作节点的prev节点开始找到前继活动节点  
 for (Node<E> p = prev; ; ++hops) { //b  
 if (p.item != null) {  
 activePred = p;  
 isFirst = false;  
 break;  
 }  
 Node<E> q = p.prev;  
 if (q == null) {  
 if (p.next == p)  
 return;  
 activePred = p;  
 isFirst = true;  
 break;  
 }  
 else if (p == q)  
 return;  
 else  
 p = q;  
 }  
  
 // Find active successor  
 for (Node<E> p = next; ; ++hops) { //c  
 if (p.item != null) {  
 activeSucc = p;  
 isLast = false;  
 break;  
 }  
 Node<E> q = p.next;  
 if (q == null) {  
 if (p.prev == p)  
 return;  
 activeSucc = p;  
 isLast = true;  
 break;  
 }  
 else if (p == q)  
 return;  
 else  
 p = q;  
 }  
  
 // *TODO: better HOP heuristics* if (hops < *HOPS* // always squeeze out interior deleted nodes  
 && (isFirst | isLast)) //d  
 return;  
  
 // Squeeze out deleted nodes between activePred and  
 // activeSucc, including x.  
 skipDeletedSuccessors(activePred); //e  
 skipDeletedPredecessors(activeSucc);  
  
 // Try to gc-unlink, if possible  
 if ((isFirst | isLast) &&  
  
 // Recheck expected state of predecessor and successor  
 (activePred.next == activeSucc) &&  
 (activeSucc.prev == activePred) &&  
 (isFirst ? activePred.prev == null : activePred.item != null) &&  
 (isLast ? activeSucc.next == null : activeSucc.item != null)) { //f  
  
 updateHead(); // Ensure x is not reachable from head  
 updateTail(); // Ensure x is not reachable from tail  
  
 // Finally, actually gc-unlink  
 x.lazySetPrev(isFirst ? prevTerminator() : x);  
 x.lazySetNext(isLast ? nextTerminator() : x);  
 }  
 }  
}

*/\*\*  
 \* Unlinks non-null first node.  
 \*/  
/\*\*出列，解除头结点链接\*/*private void unlinkFirst(Node<E> first, Node<E> next) {  
 // assert first != null;  
 // assert next != null;  
 // assert first.item == null;  
 //从next节点开始向后寻找有效节点  
 //如果first.next.item!=null，直接返回  
 for (Node<E> o = null, p = next, q;;) {  
 if (p.item != null || (q = p.next) == null) {  
 //找到合适节点p，cas替换first.next节点为p  
 if (o != null && p.prev != p && first.casNext(next, p)) {  
 //更新p的prev节点  
 skipDeletedPredecessors(p);  
 if (first.prev == null &&  
 (p.next == null || p.item != null) &&  
 p.prev == first) {  
 //更新head节点，确保已删除节点o从head不可达(unlinking)  
 updateHead(); // Ensure o is not reachable from head  
 //更新tail节点，确保已删除节点o从tail不可达(unlinking)  
 updateTail(); // Ensure o is not reachable from tail  
  
 // Finally, actually gc-unlink  
 //使未链接节点指向自身  
 o.lazySetNext(o);  
 //设置移除节点的prev为PREV\_TERMINATOR  
 o.lazySetPrev(prevTerminator());  
 }  
 }  
 return;  
 }  
 else if (p == q)  
 return;  
 else {  
 o = p;  
 p = q;  
 }  
 }  
}  
  
*/\*\*  
 \* Unlinks non-null last node.  
 \*/  
/\*\*出列，移除队列尾节点\*/*private void unlinkLast(Node<E> last, Node<E> prev) {  
 // assert last != null;  
 // assert prev != null;  
 // assert last.item == null;  
 for (Node<E> o = null, p = prev, q;;) {  
 if (p.item != null || (q = p.prev) == null) {  
 if (o != null && p.next != p && last.casPrev(prev, p)) {  
 skipDeletedSuccessors(p);  
 if (last.next == null &&  
 (p.prev == null || p.item != null) &&  
 p.next == last) {  
  
 updateHead(); // Ensure o is not reachable from head  
 updateTail(); // Ensure o is not reachable from tail  
  
 // Finally, actually gc-unlink  
 o.lazySetPrev(o);  
 o.lazySetNext(nextTerminator());  
 }  
 }  
 return;  
 }  
 else if (p == q)  
 return;  
 else {  
 o = p;  
 p = q;  
 }  
 }  
}

**说明：**  
1. 如果此时操作的是first节点，则调用unlinkFirst()方法解除头结点链接，这里分几种情况说明：  
 a. 如果first.next.item!=null，则直接返回，此时没有对first节点做任何操作；  
 b. 如果first.next.item==null，first.next.next==null，此时first.next节点为last节点，直接返回；  
 c. 非上述情况则继续往后循环查找活动节点，使o等于跳过的节点，cas替换first的next节点为新找到的活动节点p，成功后调用skipDeletedPredecessors(p)更新p的前继节点为first；  
 d. 判断上述操作是否正常完成，成功cas更新head和tail节点；更新o(item==null)节点的next指向自身，prev节点指向PREV\_TERMINATOR

2. 如果此时操作的是last节点，则调用unlinkLast()方法解除尾节点链接，参考上面的说明，这里不作赘述。

**注意：在上面的两个操作之后，此时链表结构first或last节点还是活动节点，只是在出列操作时改变了item=null；同样的这个操作也会把后续的无效节点的链接剔除。**

3. 非first和last节点 操作说明：  
 a. 回到pollFirst()方法，如果此时的队列是经过上面两步操作之后的队列(first.item=null)，在进入unlink()方法后操作的节点就是first.next节点了  
 b. 从被操作节点的prev节点开始向前自旋查找前继活动节点activePred，继续前面的步骤，此时活动前节点为first节点，isfirst=true。  
 c. 从被操作节点的next节点开始向后自旋查找后继活动节点activeSucc，这里假设查找next跳跃了一个节点（e操作中说明）。  
 d. 这里涉及到一个变量HOPS，如果在b c两步中的查找跳跃数小于HOPS，并且找到的有效节点中有first或last节点，直接返回。  
 e. 非d中所述条件继续执行；通过skipDeletedSuccessors(activePred)和skipDeletedPredecessors(activeSucc)方法分别更新(b和c操作中)找到的两个活动节点的后继和前继live(item!=null)节点（此时在正常进行之后，activePred和activeSucc应该是相互链接的）。源码如下：

*/\*\*更新指定节点的前节点\*/*private void skipDeletedPredecessors(Node<E> x) {  
 whileActive:  
 do {  
 Node<E> prev = x.prev;  
 // assert prev != null;  
 // assert x != NEXT\_TERMINATOR;  
 // assert x != PREV\_TERMINATOR;  
 Node<E> p = prev;  
 //找到有效活跃节点(item不为空或者前节点为first的节点)  
 findActive:  
 for (;;) {  
 if (p.item != null)  
 break findActive;  
 Node<E> q = p.prev;  
 if (q == null) {  
 if (p.next == p)//p可能为self-node  
 continue whileActive;  
 break findActive;  
 }  
 else if (p == q)  
 continue whileActive;  
 else  
 p = q;//继续往前寻找有效节点  
 }  
  
 // found active CAS target  
 if (prev == p || x.casPrev(prev, p))//更新前节点  
 return;  
  
 } while (x.item != null || x.next == null);  
}  
  
*/\*\*更新指定节点的后继节点\*/*private void skipDeletedSuccessors(Node<E> x) {  
 whileActive:  
 do {  
 Node<E> next = x.next;  
 // assert next != null;  
 // assert x != NEXT\_TERMINATOR;  
 // assert x != PREV\_TERMINATOR;  
 Node<E> p = next;  
 //找到有效活跃节点(item不为空或者后节点为last的节点)  
 findActive:  
 for (;;) {  
 if (p.item != null)  
 break findActive;  
 Node<E> q = p.next;  
 if (q == null) {  
 if (p.prev == p)//p可能为last节点  
 continue whileActive;  
 break findActive;  
 }  
 else if (p == q)  
 continue whileActive;  
 else  
 p = q;//继续往后寻找  
 }  
  
 // found active CAS target  
 if (next == p || x.casNext(next, p))//cas更新next节点  
 return;  
  
 } while (x.item != null || x.prev == null);  
}

f. 判断操作的活动节点是否有first或last节点，有则更新head和tail，最后通过UNSAFE的putOrderedObject更新出列节点的prev和next分别为PREV\_TERMINATOR和NEXT\_TERMINATOR。

#### DelayQueue

##### 概述

**DelayQueue是一个延时无界阻塞队列，锁是由ReentrantLock实现的，队列中的元素必须实现Delayed接口，也就是说只允许放入可以“延期”的元素。**

##### 数据结构



//放入的元素必须实现Delayed接口  
public class DelayQueue<E extends Delayed> extends AbstractQueue<E>  
 implements BlockingQueue<E> {  
  
 private final transient ReentrantLock lock = new ReentrantLock();  
 private final PriorityQueue<E> q = new PriorityQueue<E>();  
  
 */\*\*  
 \* Thread designated to wait for the element at the head of  
 \* the queue. This variant of the Leader-Follower pattern  
 \* (http://www.cs.wustl.edu/~schmidt/POSA/POSA2/) serves to  
 \* minimize unnecessary timed waiting. When a thread becomes  
 \* the leader, it waits only for the next delay to elapse, but  
 \* other threads await indefinitely. The leader thread must  
 \* signal some other thread before returning from take() or  
 \* poll(...), unless some other thread becomes leader in the  
 \* interim. Whenever the head of the queue is replaced with  
 \* an element with an earlier expiration time, the leader  
 \* field is invalidated by being reset to null, and some  
 \* waiting thread, but not necessarily the current leader, is  
 \* signalled. So waiting threads must be prepared to acquire  
 \* and lose leadership while waiting.  
 \*/* private Thread leader = null;  
  
 */\*\*  
 \* Condition signalled when a newer element becomes available  
 \* at the head of the queue or a new thread may need to  
 \* become leader.  
 \*/* private final Condition available = lock.newCondition();

说明：  
1. 存储元素必须实现Delayed接口  
2. 内部持有一个ReentrantLock  
3. 使用优先级队列PriorityQueue实现元素存储  
4. 持有一个优化内部阻塞通知的线程leader  
5. 用于实现阻塞的Condition对象

**关于Delayed和PriorityQueue：**

**Delayed是一个具有过期时间的元素  
PriorityQueue是二叉堆实现的根据队列里元素的某些属性排序的的优先级队列，内部持有一个比较器comparator**

6. delayQueue其实就是在每次往优先级队列中添加元素,然后以元素的delay/过期值作为排序的因素,以此来达到先过期的元素会拍在队首,每次从队列里取出来都是最先要过期的元素

##### 函数列表

public DelayQueue() {}

public DelayQueue(Collection<? extends E> c) {  
 this.addAll(c);  
}

*/\*\*添加元素{****@link*** *Collection#add}\*/*public boolean add(E e) {  
 return offer(e);  
}public boolean offer(E e)

public void put(E e) {  
 offer(e);  
}  
public boolean offer(E e, long timeout, TimeUnit unit)

*/\*\*获取并移除头元素\*/*public E poll()   
public E poll(long timeout, TimeUnit unit) throws InterruptedException   
*/\*\*获取并移除头元素，一直等待元素可用\*/*public E take() throws InterruptedException   
*/\*\*获取元素，不移除\*/*public E peek()  
*/\*\*返回队列元素数量\*/*public int size()  
*/\*\*移除队列中可用元素，添加到指定Collection中\*/*public int drainTo(Collection<? super E> c)  
public int drainTo(Collection<? super E> c, int maxElements)  
*/\*\*清空队列\*/*public void clear()  
*/\*\*返回队列剩余空间\*/*public int remainingCapacity()

public Object[] toArray()

public <T> T[] toArray(T[] a)

public boolean remove(Object o)

##### 源码解析

###### offer(E)

*/\*\*  
 \* Inserts the specified element into this delay queue.  
 \*  
 \** ***@param*** *e the element to add  
 \** ***@return*** *{****@code*** *true}  
 \** ***@throws*** *NullPointerException if the specified element is null  
 \*/  
/\*\*添加元素\*/*public boolean offer(E e) {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 q.offer(e);  
 if (q.peek() == e) {  
 leader = null;  
 available.signal();  
 }  
 return true;  
 } finally {  
 lock.unlock();  
 }  
}

**说明：**首先执行锁操作，把元素添加到优先级队列priorityQueue中；然后查看元素是否为头元素，是的话则设置leader为空，唤醒所有等待available的线程；最后释放锁。这里用到了PriorityQueue的offer()和peek()方法。siftUp()在下一章介绍priorityBlockingQueue的时候会具体分析。方法在源码如下：

*/\*\*  
 \* Inserts the specified element into this priority queue.  
 \*  
 \** ***@return*** *{****@code*** *true} (as specified by {****@link*** *Queue#offer})  
 \** ***@throws*** *ClassCastException if the specified element cannot be  
 \* compared with elements currently in this priority queue  
 \* according to the priority queue's ordering  
 \** ***@throws*** *NullPointerException if the specified element is null  
 \*/*public boolean offer(E e) {  
 if (e == null)  
 throw new NullPointerException();  
 modCount++;//代表队列结构被修改的次数  
 int i = size;  
 if (i >= queue.length)  
 //扩容操作  
 grow(i + 1);  
 size = i + 1;  
 if (i == 0)  
 //队列元素数为0直接添加  
 queue[0] = e;  
 else  
 //通过comparator找到合适位置添加  
 siftUp(i, e);  
 return true;  
}  
  
@SuppressWarnings("unchecked")  
public E peek() {  
 return (size == 0) ? null : (E) queue[0];  
}

*/\*\*  
 \* Inserts item x at position k, maintaining heap invariant by  
 \* promoting x up the tree until it is greater than or equal to  
 \* its parent, or is the root.  
 \*  
 \* To simplify and speed up coercions and comparisons. the  
 \* Comparable and Comparator versions are separated into different  
 \* methods that are otherwise identical. (Similarly for siftDown.)  
 \*  
 \** ***@param*** *k the position to fill  
 \** ***@param*** *x the item to insert  
 \*  
 \* 把x元素插入到指定位置，从父节点向上查找到合适位置(大于等于父节点)  
 \* 插入过程中保持堆的性质不变  
 \*/*

private void siftUp(int k, E x) {  
 if (comparator != null)  
 siftUpUsingComparator(k, x);  
 else  
 siftUpComparable(k, x);  
}  
  
@SuppressWarnings("unchecked")  
private void siftUpComparable(int k, E x) {  
 Comparable<? super E> key = (Comparable<? super E>) x;  
 while (k > 0) {  
 //从中段开始查找合适位置  
 int parent = (k - 1) >>> 1;  
 Object e = queue[parent];  
 if (key.compareTo((E) e) >= 0)  
 break;  
 queue[k] = e;  
 k = parent;  
 }  
 queue[k] = key;  
}  
  
@SuppressWarnings("unchecked")  
private void siftUpUsingComparator(int k, E x) {  
 while (k > 0) {  
 int parent = (k - 1) >>> 1;  
 Object e = queue[parent];  
 if (comparator.compare(x, (E) e) >= 0)  
 break;  
 queue[k] = e;  
 k = parent;  
 }  
 queue[k] = x;  
}

###### take()

*/\*\*  
 \* Retrieves and removes the head of this queue, or returns {****@code*** *null}  
 \* if this queue has no elements with an expired delay.  
 \*  
 \** ***@return*** *the head of this queue, or {****@code*** *null} if this  
 \* queue has no elements with an expired delay  
 \*/  
/\*\*获取并移除头元素\*/*public E poll() {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 E first = q.peek();  
 if (first == null || first.getDelay(*NANOSECONDS*) > 0)  
 return null;  
 else  
 return q.poll();  
 } finally {  
 lock.unlock();  
 }  
}

**说明：**首先对队列加（响应中断）锁；  
调用priorityQueue.peek()方法获取首元素；  
首元素为空等待元素可用；  
首元素可用并且元素已到延时时间调用priorityQueue.poll()方法返回；  
首元素未到延时时间，则首先释放first的引用，防止内存泄漏；  
leader不为空，证明有其他线程已经获取到leader，当前线程加入等待条件队列  
leader为空，使leader指向当前线程，等待delay时间；  
检查leader是否被其他线程改变，没有就重置leader继续循环  
unlock  
priorityQueue的poll()方法源码如下：

public E poll() {  
 if (size == 0)  
 return null;  
 int s = --size;  
 modCount++;  
 E result = (E) queue[0];  
 E x = (E) queue[s];  
 queue[s] = null;  
 if (s != 0)  
 siftDown(0, x);  
 return result;  
}

*/\*\*  
 \* Inserts item x at position k, maintaining heap invariant by  
 \* demoting x down the tree repeatedly until it is less than or  
 \* equal to its children or is a leaf.  
 \*  
 \** ***@param*** *k the position to fill  
 \** ***@param*** *x the item to insert  
 \*  
 \* 把x元素插入到指定位置，从子节点向下查找到合适位置(小于等于子节点)  
 \* 插入过程中保持堆的性质不变  
 \*/*private void siftDown(int k, E x) {  
 if (comparator != null)  
 siftDownUsingComparator(k, x);  
 else  
 siftDownComparable(k, x);  
}  
  
@SuppressWarnings("unchecked")  
private void siftDownComparable(int k, E x) {  
 Comparable<? super E> key = (Comparable<? super E>)x;  
 int half = size >>> 1; // loop while a non-leaf  
 while (k < half) {  
 int child = (k << 1) + 1; // assume left child is least  
 Object c = queue[child];  
 int right = child + 1;  
 if (right < size &&  
 ((Comparable<? super E>) c).compareTo((E) queue[right]) > 0)  
 c = queue[child = right];  
 if (key.compareTo((E) c) <= 0)  
 break;  
 queue[k] = c;  
 k = child;  
 }  
 queue[k] = key;  
}  
  
@SuppressWarnings("unchecked")  
private void siftDownUsingComparator(int k, E x) {  
 int half = size >>> 1;  
 while (k < half) {  
 int child = (k << 1) + 1;  
 Object c = queue[child];  
 int right = child + 1;  
 if (right < size &&  
 comparator.compare((E) c, (E) queue[right]) > 0)  
 c = queue[child = right];  
 if (comparator.compare(x, (E) c) <= 0)  
 break;  
 queue[k] = c;  
 k = child;  
 }  
 queue[k] = x;  
}

#### PriorityBlockingQueue

##### 概述

**PriorityBlockingQueue（以下称PBD）是一个基于平衡二元堆实现的无界优先级阻塞队列，与priorityQueue的排序规则一样；由于队列是无界的，企图向一个资源耗尽的队列中添加元素会引起OutOfMemoryError；PBD不允许添加null值；优先级队列依赖于comparable，并且不允许插入不能排序的对象（抛ClassCastException）。**

##### 数据结构



public class PriorityBlockingQueue<E> extends AbstractQueue<E>  
 implements BlockingQueue<E>, java.io.Serializable {  
 private static final long *serialVersionUID* = 5595510919245408276L;  
  
 /\*  
 \* The implementation uses an array-based binary heap, with public  
 \* operations protected with a single lock. However, allocation  
 \* during resizing uses a simple spinlock (used only while not  
 \* holding main lock) in order to allow takes to operate  
 \* concurrently with allocation. This avoids repeated  
 \* postponement of waiting consumers and consequent element  
 \* build-up. The need to back away from lock during allocation  
 \* makes it impossible to simply wrap delegated  
 \* java.util.PriorityQueue operations within a lock, as was done  
 \* in a previous version of this class. To maintain  
 \* interoperability, a plain PriorityQueue is still used during  
 \* serialization, which maintains compatibility at the expense of  
 \* transiently doubling overhead.  
 \*/  
  
 */\*\*  
 \* Default array capacity.  
 \*/* //默认容量  
 private static final int *DEFAULT\_INITIAL\_CAPACITY* = 11;  
  
 */\*\*  
 \* The maximum size of array to allocate.  
 \* Some VMs reserve some header words in an array.  
 \* Attempts to allocate larger arrays may result in  
 \* OutOfMemoryError: Requested array size exceeds VM limit  
 \*/* //数组最大长度  
 private static final int *MAX\_ARRAY\_SIZE* = Integer.*MAX\_VALUE* - 8;  
  
 */\*\*  
 \* Priority queue represented as a balanced binary heap: the two  
 \* children of queue[n] are queue[2\*n+1] and queue[2\*(n+1)]. The  
 \* priority queue is ordered by comparator, or by the elements'  
 \* natural ordering, if comparator is null: For each node n in the  
 \* heap and each descendant d of n, n <= d. The element with the  
 \* lowest value is in queue[0], assuming the queue is nonempty.  
 \*  
 \* 基于一个平衡的二元堆实现  
 \*/* private transient Object[] queue;  
  
 */\*\*  
 \* The number of elements in the priority queue.  
 \*/* private transient int size;  
  
 */\*\*  
 \* The comparator, or null if priority queue uses elements'  
 \* natural ordering.  
 \*/* private transient Comparator<? super E> comparator;  
  
 */\*\*  
 \* Lock used for all public operations  
 \*/* private final ReentrantLock lock;  
  
 */\*\*  
 \* Condition for blocking when empty  
 \*/* private final Condition notEmpty;  
  
 */\*\*  
 \* Spinlock for allocation, acquired via CAS.  
 \*/  
 /\*\*初始0为可获取状态，用于控制扩容操作\*/* private transient volatile int allocationSpinLock;  
  
 */\*\*  
 \* A plain PriorityQueue used only for serialization,  
 \* to maintain compatibility with previous versions  
 \* of this class. Non-null only during serialization/deserialization.  
 \*/* private PriorityQueue<E> q;

**说明：**  
1. PBD默认容量11，最大容量Integer.*MAX\_VALUE* – 8，这里是因为有些虚拟机会在数组中存储一些头数据，这里是为头数据保留储存空间，防止内存溢出。  
2. 阻塞是通过ReentrantLock和Condition实现的  
3. 内部持有一个priorityQueue的引用，主要是为了保持兼容性，只有在序列化的时候用到  
4. 持有一个控制扩容操作时的锁allocationSpinLock，初始状态为0，通过cas操作获取状态。  
5. 元素存储使用了一个平衡的**二叉最小堆**，关于二叉堆，这里简单介绍一下，以后开数据结构篇再详细分析。

###### 二叉堆（扩展）

**二叉堆**是一种特殊的[堆](https://zh.wikipedia.org/wiki/%E5%A0%86_(%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84))，二叉堆是完全二叉树或者是近似完全二叉树。二叉堆满足堆特性：父节点的键值总是保持固定的序关系于任何一个子节点的键值，且每个节点的左子树和右子树都是一个二叉堆。

当父节点的键值总是大于或等于任何一个子节点的键值时为**最大堆**。 当父节点的键值总是小于或等于任何一个子节点的键值时为**最小堆**。

二叉堆一般用数组来表示。如果根节点在数组中的位置是1，第n个位置的子节点分别在2n和 2n+1。因此，第1个位置的子节点在2和3，第2个位置的子节点在4和5。以此类推。这种基于1的数组存储方式便于寻找父节点和子节点。

如果存储数组的下标基于0，那么下标为i的节点的子节点是2i + 1与2i + 2；其父节点的下标是⌊(i − 1) ∕ 2⌋。

如下图的两个堆：

1 11

/ \ / \

2 3 9 10

/ \ / \ / \ / \

4 5 6 7 5 6 7 8

/ \ / \ /\ /\

8 9 10 11 1 2 3 4

将这两个堆保存在以1开始的数组中：

位置: 1 2 3 4 5 6 7 8 9 10 11

左图: 1 2 3 4 5 6 7 8 9 10 11

右图: 11 9 10 5 6 7 8 1 2 3 4

对于一个很大的堆，这种存储是低效的。因为节点的子节点很可能在另外一个内存页中。[B-heap](https://zh.wikipedia.org/wiki/B-heap)是一种效率更高的存储方式，把每个子树放到同一内存页。

如果用指针链表存储堆，那么需要能访问叶节点的方法。可以对二叉树“穿线”(threading)方式，来依序遍历这些节点。

##### 函数列表

public PriorityBlockingQueue() {  
 this(*DEFAULT\_INITIAL\_CAPACITY*, null);  
}

public PriorityBlockingQueue(int initialCapacity) {  
 this(initialCapacity, null);  
}

public PriorityBlockingQueue(int initialCapacity,  
 Comparator<? super E> comparator) {  
 if (initialCapacity < 1)  
 throw new IllegalArgumentException();  
 this.lock = new ReentrantLock();  
 this.notEmpty = lock.newCondition();  
 this.comparator = comparator;  
 this.queue = new Object[initialCapacity];  
}

public PriorityBlockingQueue(Collection<? extends E> c)

*/\*\*  
 \* Inserts the specified element into this priority queue.  
 \*  
 \** ***@param*** *e the element to add  
 \** ***@return*** *{****@code*** *true} (as specified by {****@link*** *Collection#add})  
 \** ***@throws*** *ClassCastException if the specified element cannot be compared  
 \* with elements currently in the priority queue according to the  
 \* priority queue's ordering  
 \** ***@throws*** *NullPointerException if the specified element is null  
 \*/*public boolean add(E e) {  
 return offer(e);  
}

*/\*\*  
 \* Inserts the specified element into this priority queue.  
 \* As the queue is unbounded, this method will never return {****@code*** *false}.  
 \*  
 \** ***@param*** *e the element to add  
 \** ***@return*** *{****@code*** *true} (as specified by {****@link*** *Queue#offer})  
 \** ***@throws*** *ClassCastException if the specified element cannot be compared  
 \* with elements currently in the priority queue according to the  
 \* priority queue's ordering  
 \** ***@throws*** *NullPointerException if the specified element is null  
 \*/*  
public boolean offer(E e)

*/\*\*  
 \* Inserts the specified element into this priority queue.  
 \* As the queue is unbounded, this method will never block.  
 \*  
 \** ***@param*** *e the element to add  
 \** ***@throws*** *ClassCastException if the specified element cannot be compared  
 \* with elements currently in the priority queue according to the  
 \* priority queue's ordering  
 \** ***@throws*** *NullPointerException if the specified element is null  
 \*/*public void put(E e) {  
 offer(e)

*/\*\*获取并移除首元素\*/*public E poll()

*/\*\*获取并移除首元素，一直等待首元素可用\*/*public E take()

public E poll(long timeout, TimeUnit unit)

*/\*\*获取首元素，不移除\*/*public E peek()

*/\*\*返回队列的comparator\*/*public Comparator<? super E> comparator()

public int size()

*/\*\*  
 \* Always returns {****@code*** *Integer.MAX\_VALUE} because  
 \* a {****@code*** *PriorityBlockingQueue} is not capacity constrained.  
 \** ***@return*** *{****@code*** *Integer.MAX\_VALUE} always  
 \*/*public int remainingCapacity() {  
 return Integer.*MAX\_VALUE*;  
}

*/\*\*  
 \* Removes a single instance of the specified element from this queue,  
 \* if it is present. More formally, removes an element {****@code*** *e} such  
 \* that {****@code*** *o.equals(e)}, if this queue contains one or more such  
 \* elements. Returns {****@code*** *true} if and only if this queue contained  
 \* the specified element (or equivalently, if this queue changed as a  
 \* result of the call).  
 \*  
 \** ***@param*** *o element to be removed from this queue, if present  
 \** ***@return*** *{****@code*** *true} if this queue changed as a result of the call  
 \*/  
/\*\*移除指定元素\*/*public boolean remove(Object o)

*/\*\*  
 \* Returns {****@code*** *true} if this queue contains the specified element.  
 \* More formally, returns {****@code*** *true} if and only if this queue contains  
 \* at least one element {****@code*** *e} such that {****@code*** *o.equals(e)}.  
 \*  
 \** ***@param*** *o object to be checked for containment in this queue  
 \** ***@return*** *{****@code*** *true} if this queue contains the specified element  
 \*/*public boolean contains(Object o)

*/\*\*  
 \* Returns an array containing all of the elements in this queue.  
 \* The returned array elements are in no particular order.  
 \*  
 \* <p>The returned array will be "safe" in that no references to it are  
 \* maintained by this queue. (In other words, this method must allocate  
 \* a new array). The caller is thus free to modify the returned array.  
 \*  
 \* <p>This method acts as bridge between array-based and collection-based  
 \* APIs.  
 \*  
 \** ***@return*** *an array containing all of the elements in this queue  
 \*/*public Object[] toArray()  
public <T> T[] toArray(T[] a)  
public String toString()  
public int drainTo(Collection<? super E> c, int maxElements)  
public void clear()

public Iterator<E> iterator()

##### 源码解析

###### offer(E)

*/\*\*  
 \* Inserts the specified element into this priority queue.  
 \* As the queue is unbounded, this method will never return {****@code*** *false}.  
 \*  
 \** ***@param*** *e the element to add  
 \** ***@return*** *{****@code*** *true} (as specified by {****@link*** *Queue#offer})  
 \** ***@throws*** *ClassCastException if the specified element cannot be compared  
 \* with elements currently in the priority queue according to the  
 \* priority queue's ordering  
 \** ***@throws*** *NullPointerException if the specified element is null  
 \*/*//添加元素  
public boolean offer(E e) {  
 if (e == null)  
 throw new NullPointerException();  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 int n, cap;  
 Object[] array;  
 while ((n = size) >= (cap = (array = queue).length))  
 //队列扩容  
 tryGrow(array, cap);  
 try {  
 Comparator<? super E> cmp = comparator;  
 //找到合适位置插入元素  
 if (cmp == null)  
 *siftUpComparable*(n, e, array);  
 else  
 *siftUpUsingComparator*(n, e, array, cmp);  
 size = n + 1;  
 notEmpty.signal();  
 } finally {  
 lock.unlock();  
 }  
 return true;  
}

说明：首先对队列加锁，判断队列是否需要扩容，如果需要调用tryGrow()方法进行扩容；然后调用siftUpComparable()方法找到合适位置插入元素；更新队列元素数size，唤醒等待notEmpty的线程，最后unlock。

1. PBQ的扩容说明：当队列元素数大于等于数组的长度时，会触发扩容操作，扩容是由单线程完成的。如果数组长度cap小于64，扩容长度为2\*(cap+1);否则扩容长度为原来的1.5倍(1.5\*cap)。

2. siftUpComparable()和siftUpUsingComparator()，由于队列中的元素都是有优先级(基于comparator排序)的，所以如果有新元素进来不会像其他队列一样直接放在队尾，而是通过这两个方法找到新增元素在队列中的排序位置然后插入。

tryGrow和siftUpComparable方法源码如下：

*/\*\*  
 \* Tries to grow array to accommodate at least one more element  
 \* (but normally expand by about 50%), giving up (allowing retry)  
 \* on contention (which we expect to be rare). Call only while  
 \* holding lock.  
 \*  
 \* 队列扩容（一般是原数组长度的1.5倍，如果数组长度小于64，扩容至2\*(cap+1)）  
 \** ***@param*** *array the heap array  
 \** ***@param*** *oldCap the length of the array  
 \*/*private void tryGrow(Object[] array, int oldCap) {  
 lock.unlock(); // must release and then re-acquire main lock  
 Object[] newArray = null;  
 if (allocationSpinLock == 0 &&  
 *UNSAFE*.compareAndSwapInt(this, *allocationSpinLockOffset*,  
 0, 1)) {  
 try {  
 //计算新的数组容量（1.5\*cap或2\*(cap+1)）  
 int newCap = oldCap + ((oldCap < 64) ?  
 (oldCap + 2) : // grow faster if small  
 (oldCap >> 1));  
 if (newCap - *MAX\_ARRAY\_SIZE* > 0) { // possible overflow  
 int minCap = oldCap + 1;  
 if (minCap < 0 || minCap > *MAX\_ARRAY\_SIZE*)  
 throw new OutOfMemoryError();  
 newCap = *MAX\_ARRAY\_SIZE*;  
 }  
 if (newCap > oldCap && queue == array)  
 newArray = new Object[newCap];  
 } finally {  
 allocationSpinLock = 0;  
 }  
 }  
 //如果有其他线程已经在进行扩容操作，当前线程让出执行时间片  
 if (newArray == null) // back off if another thread is allocating  
 Thread.*yield*();  
 lock.lock();  
 if (newArray != null && queue == array) {  
 queue = newArray;  
 System.*arraycopy*(array, 0, newArray, 0, oldCap);  
 }  
}

*/\*\*  
 \* Inserts item x at position k, maintaining heap invariant by  
 \* promoting x up the tree until it is greater than or equal to  
 \* its parent, or is the root.  
 \*  
 \* To simplify and speed up coercions and comparisons. the  
 \* Comparable and Comparator versions are separated into different  
 \* methods that are otherwise identical. (Similarly for siftDown.)  
 \* These methods are static, with heap state as arguments, to  
 \* simplify use in light of possible comparator exceptions.  
 \*  
 \** ***@param*** *k the position to fill  
 \** ***@param*** *x the item to insert  
 \** ***@param*** *array the heap array  
 \*/  
/\*\*在k位置插入元素x，从父节点开始向上找到合适位置，保持二元堆的性质不变\*/*private static <T> void siftUpComparable(int k, T x, Object[] array) {  
 Comparable<? super T> key = (Comparable<? super T>) x;  
 while (k > 0) {  
 int parent = (k - 1) >>> 1;  
 Object e = array[parent];  
 if (key.compareTo((T) e) >= 0)  
 break;  
 array[k] = e;  
 k = parent;  
 }  
 array[k] = key;  
}  
  
private static <T> void siftUpUsingComparator(int k, T x, Object[] array,  
 Comparator<? super T> cmp) {  
 while (k > 0) {  
 int parent = (k - 1) >>> 1;  
 Object e = array[parent];  
 if (cmp.compare(x, (T) e) >= 0)  
 break;  
 array[k] = e;  
 k = parent;  
 }  
 array[k] = x;  
}

###### poll()

*/\*\*获取并移除首元素\*/*public E poll() {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 return dequeue();  
 } finally {  
 lock.unlock();  
 }  
}

*/\*\*  
 \* Mechanics for poll(). Call only while holding lock.  
 \*/*//出列  
private E dequeue() {  
 int n = size - 1;  
 if (n < 0)  
 return null;  
 else {  
 Object[] array = queue;  
 E result = (E) array[0];  
 E x = (E) array[n];  
 array[n] = null;  
 Comparator<? super E> cmp = comparator;  
 if (cmp == null)  
 *siftDownComparable*(0, x, array, n);  
 else  
 *siftDownUsingComparator*(0, x, array, n, cmp);  
 size = n;  
 return result;  
 }  
}

**说明：** 从源码中可以看到在出列操作时，首先移除数组的最后一个元素，然后调用siftDownComparable或siftDownUsingComparator方法进行二叉堆的重组。  
siftDownComparable和siftDownUsingComparator源码如下：

*/\*\*  
 \* Inserts item x at position k, maintaining heap invariant by  
 \* demoting x down the tree repeatedly until it is less than or  
 \* equal to its children or is a leaf.  
 \*  
 \** ***@param*** *k the position to fill  
 \** ***@param*** *x the item to insert  
 \** ***@param*** *array the heap array  
 \** ***@param*** *n heap size  
 \*/  
/\*\*在k位置插入元素x，从子节点开始向下找到合适位置，保持二元堆的性质不变\*/*private static <T> void siftDownComparable(int k, T x, Object[] array,  
 int n) {  
 if (n > 0) {  
 Comparable<? super T> key = (Comparable<? super T>)x;  
 int half = n >>> 1; // loop while a non-leaf  
 while (k < half) {  
 //从左叶子节点查找  
 int child = (k << 1) + 1; // assume left child is least  
 Object c = array[child];  
 int right = child + 1;  
 if (right < n &&  
 ((Comparable<? super T>) c).compareTo((T) array[right]) > 0)  
 c = array[child = right];  
 if (key.compareTo((T) c) <= 0)  
 break;  
 array[k] = c;  
 k = child;  
 }  
 array[k] = key;  
 }  
}  
  
private static <T> void siftDownUsingComparator(int k, T x, Object[] array,  
 int n,  
 Comparator<? super T> cmp) {  
 if (n > 0) {  
 int half = n >>> 1;  
 while (k < half) {  
 int child = (k << 1) + 1;  
 Object c = array[child];  
 int right = child + 1;  
 if (right < n && cmp.compare((T) c, (T) array[right]) > 0)  
 c = array[child = right];  
 if (cmp.compare(x, (T) c) <= 0)  
 break;  
 array[k] = c;  
 k = child;  
 }  
 array[k] = x;  
 }  
}

**小结：**PBD的操作主要是针对内部二元堆的操作，这里理解二元堆的结构和原理，其他源码就很简单了。

#### SynchronousQueue

##### 概述

* 1. **SynchronousQueue是一个同步阻塞队列，它的每一个插入操作都要等待其他线程响应的移除操作，反之亦然。**
  2. **它本身是没有容量的。不能调用peek方法获取元素，因为元素只有在被移除的时候才会呈现，在一个线程移除元素之前，也不能插入元素；由于队列中没有元素，所以也不能迭代。**
  3. **整个 queue 分为 公平(TransferQueue FIFO)与非公平模式(TransferStack LIFO 默认)**
  4. **若使用 TransferQueue, 则队列中永远会存在一个 dummy node**
  5. **SynchronousQueue的一个使用场景是在线程池里。Executors.newCachedThreadPool()就使用了SynchronousQueue，这个线程池根据需要（新任务到来时）创建新的线程，如果有空闲线程则会重复使用，线程空闲了60秒后会被回收。**

##### 数据结构



*/\*\*  
 \* Shared internal API for dual stacks and queues.  
 \*/*abstract static class Transferer<E> {

//提供put和take操作  
 abstract E transfer(E e, boolean timed, long nanos);  
}

*/\*\* The number of CPUs, for spin control \*/*static final int *NCPUS* = Runtime.*getRuntime*().availableProcessors();

*/\*\*  
 \* The number of times to spin before blocking in timed waits.  
 \* The value is empirically derived -- it works well across a  
 \* variety of processors and OSes. Empirically, the best value  
 \* seems not to vary with number of CPUs (beyond 2) so is just  
 \* a constant.  
 \* 阻塞等待的自旋次数  
 \*/*static final int *maxTimedSpins* = (*NCPUS* < 2) ? 0 : 32;  
  
*/\*\*  
 \* The number of times to spin before blocking in untimed waits.  
 \* This is greater than timed value because untimed waits spin  
 \* faster since they don't need to check times on each spin.  
 \* 非计时的阻塞等待自旋次数  
 \*/*static final int *maxUntimedSpins* = *maxTimedSpins* \* 16;  
  
*/\*\*  
 \* The number of nanoseconds for which it is faster to spin  
 \* rather than to use timed park. A rough estimate suffices.  
 \* 自旋超时阀值  
 \*/*static final long *spinForTimeoutThreshold* = 1000L;

*/\*\*  
 \* 在构造时初始化，但在没有更为复杂的序列化时不能声明为final。  
 \* 由于只有公用方法中访问一次，所以用volatile代替final不会有显著的性能损失  
 \*/*private transient volatile Transferer<E> transferer;

*/\*\*  
 \* Creates a {****@code*** *SynchronousQueue} with nonfair access policy.  
 \* 默认非公平模式  
 \*/*public SynchronousQueue() {  
 this(false);  
}

*/\*\*  
 \* Creates a {****@code*** *SynchronousQueue} with the specified fairness policy.  
 \*  
 \** ***@param*** *fair if true, waiting threads contend in FIFO order for  
 \* access; otherwise the order is unspecified.  
 \*/*public SynchronousQueue(boolean fair) {  
 transferer = fair ? new TransferQueue<E>() : new TransferStack<E>();  
}

**说明：**SQ内部有一个抽象类Transferer,为SQ提供take和put操作。Transferer分两种实现：

* 1. TransferStack，非公平模式，栈实现(FILO)，由链表组成（内部由一个SNode实现），主要属性源码如下：

*/\*\* Dual stack \*/  
/\*\*非公平模式，栈实现\*/*static final class TransferStack<E> extends Transferer<E> {

/\* Modes for SNodes, ORed together in node fields \*/  
*/\*\* Node represents an unfulfilled consumer \*/  
/\*\* 消费者请求数据 \*/*static final int *REQUEST* = 0;  
*/\*\* Node represents an unfulfilled producer \*/  
/\*\* 生产者生产数据 \*/*static final int *DATA* = 1;  
*/\*\* Node is fulfilling another unfulfilled DATA or REQUEST \*/  
/\*\* 数据正在匹配其他生产者或消费者 \*/*static final int *FULFILLING* = 2;

static final class SNode {  
 volatile SNode next; // next node in stack  
 volatile SNode match; // the node matched to this  
 volatile Thread waiter; // to control park/unpark  
 Object item; // data; or null for REQUESTs  
 int mode;  
 //item和mode不需要用volatile修饰，因为他们总是发生在  
 // 其他volatile/atomic操作之前写或之后读  
 // Note: item and mode fields don't need to be volatile  
 // since they are always written before, and read after,  
 // other volatile/atomic operations.  
  
 SNode(Object item) {  
 this.item = item;  
 }

}

*/\*\* The head (top) of the stack \*/*volatile SNode head;

}

* 1. TransferQueue，公平模式，队列实现(FIFO)，由链表组成，主要属性源码如下：

*/\*\* Dual Queue \*/  
/\*\* 公平模式，队列实现 \*/*static final class TransferQueue<E> extends Transferer<E> {  
 /\*  
 \* This extends Scherer-Scott dual queue algorithm, differing,  
 \* among other ways, by using modes within nodes rather than  
 \* marked pointers. The algorithm is a little simpler than  
 \* that for stacks because fulfillers do not need explicit  
 \* nodes, and matching is done by CAS'ing QNode.item field  
 \* from non-null to null (for put) or vice versa (for take).  
 \*/  
  
 */\*\* Node class for TransferQueue. \*/* static final class QNode {  
 volatile QNode next; // next node in queue  
 volatile Object item; // CAS'ed to or from null  
 volatile Thread waiter; // to control park/unpark  
 final boolean isData;  
  
 QNode(Object item, boolean isData) {  
 this.item = item;  
 this.isData = isData;  
 }

}

*/\*\* Head of queue \*/*transient volatile QNode head;  
*/\*\* Tail of queue \*/*transient volatile QNode tail;  
*/\*\*  
 \* Reference to a cancelled node that might not yet have been  
 \* unlinked from queue because it was the last inserted node  
 \* when it was cancelled.  
 \*/  
/\*\* 被取消引用的节点可能还没有被解除链接，因为它被取消引用时是最后被插入的节点 \*/*transient volatile QNode cleanMe;

*/\*\*  
 \* 构造一个 dummy node, 而整个 queue 中永远会存在这样一个 dummy node  
 \* dummy node 的存在使得 代码中不存在复杂的 if 条件判断  
 \*/*TransferQueue() {  
 QNode h = new QNode(null, false); // initialize to dummy node.  
 head = h;  
 tail = h;  
}

##### 函数列表

*/\*\*  
 \* Adds the specified element to this queue, waiting if necessary for  
 \* another thread to receive it.  
 \*  
 \** ***@throws*** *InterruptedException {****@inheritDoc****}  
 \** ***@throws*** *NullPointerException {****@inheritDoc****}  
 \*/*public void put(E e) throws InterruptedException

*/\*\*  
 \* Inserts the specified element into this queue, waiting if necessary  
 \* up to the specified wait time for another thread to receive it.  
 \*  
 \** ***@return*** *{****@code*** *true} if successful, or {****@code*** *false} if the  
 \* specified waiting time elapses before a consumer appears  
 \** ***@throws*** *InterruptedException {****@inheritDoc****}  
 \** ***@throws*** *NullPointerException {****@inheritDoc****}  
 \*/*public boolean offer(E e, long timeout, TimeUnit unit)  
 throws InterruptedException

*/\*\*  
 \* Inserts the specified element into this queue, if another thread is  
 \* waiting to receive it.  
 \*  
 \** ***@param*** *e the element to add  
 \** ***@return*** *{****@code*** *true} if the element was added to this queue, else  
 \* {****@code*** *false}  
 \** ***@throws*** *NullPointerException if the specified element is null  
 \*/*public boolean offer(E e)

*/\*\*  
 \* Retrieves and removes the head of this queue, waiting if necessary  
 \* for another thread to insert it.  
 \*  
 \** ***@return*** *the head of this queue  
 \** ***@throws*** *InterruptedException {****@inheritDoc****}  
 \*/*public E take() throws InterruptedException

*/\*\*  
 \* Retrieves and removes the head of this queue, waiting  
 \* if necessary up to the specified wait time, for another thread  
 \* to insert it.  
 \*  
 \** ***@return*** *the head of this queue, or {****@code*** *null} if the  
 \* specified waiting time elapses before an element is present  
 \** ***@throws*** *InterruptedException {****@inheritDoc****}  
 \*/*public E poll(long timeout, TimeUnit unit) throws InterruptedException

*/\*\*  
 \* Retrieves and removes the head of this queue, if another thread  
 \* is currently making an element available.  
 \*  
 \** ***@return*** *the head of this queue, or {****@code*** *null} if no  
 \* element is available  
 \*/*public E poll()

##### 源码解析

**SQ的插入元素方法包括：put(E)、offer(E);取元素的方法包括：take()、poll()。这些方法全部都是由Transferer的transfer方法实现，这里主要针对Transferer的两个实现(TransferStack和TransferQueue)的transfer方法分析。**

###### TransferStack.transfer()

* + - * 1. put(E)

*/\*\*  
 \* Adds the specified element to this queue, waiting if necessary for  
 \* another thread to receive it.  
 \*  
 \** ***@throws*** *InterruptedException {****@inheritDoc****}  
 \** ***@throws*** *NullPointerException {****@inheritDoc****}  
 \*/*public void put(E e) throws InterruptedException {  
 if (e == null) throw new NullPointerException();  
 if (transferer.transfer(e, false, 0) == null) {  
 Thread.*interrupted*();  
 throw new InterruptedException();  
 }  
}

**说明：**put方法调用了transfer方法

* + - * 1. transfer(E e, boolean timed, long nanos)

*/\*\*  
 \* Puts or takes an item.  
 \*/*@SuppressWarnings("unchecked")  
E transfer(E e, boolean timed, long nanos) {  
 /\*  
 \* Basic algorithm is to loop trying one of three actions:  
 \*  
 \* 1. If apparently empty or already containing nodes of same  
 \* mode, try to push node on stack and wait for a match,  
 \* returning it, or null if cancelled.  
 \*  
 \* 2. If apparently containing node of complementary mode,  
 \* try to push a fulfilling node on to stack, match  
 \* with corresponding waiting node, pop both from  
 \* stack, and return matched item. The matching or  
 \* unlinking might not actually be necessary because of  
 \* other threads performing action 3:  
 \*  
 \* 3. If top of stack already holds another fulfilling node,  
 \* help it out by doing its match and/or pop  
 \* operations, and then continue. The code for helping  
 \* is essentially the same as for fulfilling, except  
 \* that it doesn't return the item.  
 \*/  
  
 SNode s = null; // constructed/reused as needed  
 //根据所传元素判断为生产or消费  
 int mode = (e == null) ? *REQUEST* : *DATA*;  
  
 for (;;) {  
 SNode h = head;  
 if (h == null || h.mode == mode) { // empty or same-mode  
 if (timed && nanos <= 0) { // can't wait  
 if (h != null && h.isCancelled())//head不为空并且head==match  
 casHead(h, h.next); // pop cancelled node  
 else  
 return null;  
 //cas把head替换为新的s节点（item=e,next=h,mode=mode）  
 } else if (casHead(h, s = *snode*(s, e, h, mode))) {  
 //等待s节点被匹配，返回s.match节点m  
 SNode m = awaitFulfill(s, timed, nanos);  
 //s.match==s（s节点被取消）  
 if (m == s) { // wait was cancelled  
 clean(s);  
 return null;  
 }  
 if ((h = head) != null && h.next == s)  
 casHead(h, s.next); // help s's fulfiller  
 return (E) ((mode == *REQUEST*) ? m.item : s.item);  
 }  
 } else if (!*isFulfilling*(h.mode)) { //head节点还没有被匹配，尝试匹配 try to fulfill  
 if (h.isCancelled()) // already cancelled  
 //head节点被取消，替换head节点继续循环  
 casHead(h, h.next); // pop and retry  
  
 //cas把head替换为新的s节点（item=e,next=h,mode=2|mode）  
 else if (casHead(h, s=*snode*(s, e, h, *FULFILLING*|mode))) {  
 for (;;) { // loop until matched or waiters disappear  
 SNode m = s.next; // m is s's match  
 if (m == null) { // all waiters are gone  
 casHead(s, null); // pop fulfill node  
 s = null; // use new node next time  
 break; // restart main loop  
 }  
 SNode mn = m.next;  
  
 if (m.tryMatch(s)) {//尝试匹配s节点  
 casHead(s, mn); // pop both s and m  
 return (E) ((mode == *REQUEST*) ? m.item : s.item);  
 } else // lost match  
 s.casNext(m, mn); //匹配失败，替换next节点 help unlink  
 }  
 }  
 } else { //头节点正在匹配 help a fulfiller  
 SNode m = h.next; // m is h's match  
 if (m == null) // waiter is gone  
 casHead(h, null); // pop fulfilling node  
 else {  
 SNode mn = m.next;  
 if (m.tryMatch(h)) // help match  
 casHead(h, mn); // pop both h and m  
 else // lost match  
 h.casNext(m, mn); // help unlink  
 }  
 }  
 }  
}

**说明：**整个程序在一个循环内，只有满足情况才能跳出循环。 大概分三种情况。

* 1. 一种情况是当前栈为空或者当前模式相同的节点遇到一起。
  2. 第二种情况是尝试匹配当前的节点，先将当前节点s如栈，如果失败（栈顶节点可能会被其他线程匹配），则循环进行匹配。
  3. 第三种情况是辅助方法，清除匹配成功的节点，或者当节点所属线程消失后将其移除栈。
     + - 1. awaitFulfill (SNode s, boolean timed, long nanos)

*/\*\*  
 \* Spins/blocks until node s is matched by a fulfill operation.  
 \* 自旋/阻塞，直到节点被一个fulfill操作匹配  
 \** ***@param*** *s the waiting node  
 \** ***@param*** *timed true if timed wait  
 \** ***@param*** *nanos timeout value  
 \** ***@return*** *matched node, or s if cancelled  
 \*/*SNode awaitFulfill(SNode s, boolean timed, long nanos) {  
 /\*  
 \* When a node/thread is about to block, it sets its waiter  
 \* field and then rechecks state at least one more time  
 \* before actually parking, thus covering race vs  
 \* fulfiller noticing that waiter is non-null so should be  
 \* woken.  
 \*  
 \* When invoked by nodes that appear at the point of call  
 \* to be at the head of the stack, calls to park are  
 \* preceded by spins to avoid blocking when producers and  
 \* consumers are arriving very close in time. This can  
 \* happen enough to bother only on multiprocessors.  
 \*  
 \* The order of checks for returning out of main loop  
 \* reflects fact that interrupts have precedence over  
 \* normal returns, which have precedence over  
 \* timeouts. (So, on timeout, one last check for match is  
 \* done before giving up.) Except that calls from untimed  
 \* SynchronousQueue.{poll/offer} don't check interrupts  
 \* and don't wait at all, so are trapped in transfer  
 \* method rather than calling awaitFulfill.  
 \*/  
 //计算截止时间  
 final long deadline = timed ? System.*nanoTime*() + nanos : 0L;  
 Thread w = Thread.*currentThread*();  
 //计算自旋次数  
 int spins = (shouldSpin(s) ?  
 (timed ? *maxTimedSpins* : *maxUntimedSpins*) : 0);  
 for (;;) {  
 if (w.isInterrupted())//当前线程被中断  
 //取消对给定节点s的匹配节点的等待  
 s.tryCancel();  
 SNode m = s.match;//获取给定节点s的match节点  
 if (m != null)//只有在m不为空的时候才返回match  
 return m;  
 if (timed) {  
 //超时处理  
 nanos = deadline - System.*nanoTime*();  
 if (nanos <= 0L) {  
 s.tryCancel();  
 continue;  
 }  
 }  
 if (spins > 0)  
 //spins-1  
 spins = shouldSpin(s) ? (spins-1) : 0;  
 else if (s.waiter == null)  
 //设置给定节点s的waiter为当前线程  
 s.waiter = w; // establish waiter so can park next iter  
 else if (!timed)//没有设定超时，直接阻塞  
 LockSupport.*park*(this);  
 else if (nanos > *spinForTimeoutThreshold*)//阻塞指定超时时间  
 LockSupport.*parkNanos*(this, nanos);  
 }  
}

*/\*\*  
 \* Returns true if node s is at head or there is an active fulfiller.  
 \* 给定节点s为head或者有一个活动的fulfiller时返回true  
 \*/*boolean shouldSpin(SNode s) {  
 SNode h = head;  
 return (h == s || h == null || *isFulfilling*(h.mode));  
}

*/\*\*  
 \* Tries to cancel a wait by matching node to itself.  
 \* 试图取消对其本身的匹配节点的等待  
 \*/*void tryCancel() {  
 *UNSAFE*.compareAndSwapObject(this, *matchOffset*, null, this);  
}

**说明：**awaitFulfill的作用是当前线程一直自旋/阻塞直到给定节点s被匹配。返回匹配的node，如果给定节点s被取消则返回s节点。

* + - * 1. clean(SNode s)

*/\*\*  
 \* Unlinks s from the stack.  
 \* 移除从栈顶头结点开始到该结点s（不包括）之间的所有已取消结点。  
 \*/*void clean(SNode s) {  
 s.item = null; // forget item  
 s.waiter = null; // forget thread  
  
 /\*  
 \* At worst we may need to traverse entire stack to unlink  
 \* s. If there are multiple concurrent calls to clean, we  
 \* might not see s if another thread has already removed  
 \* it. But we can stop when we see any node known to follow  
 \* s. We use s.next unless it too is cancelled, in  
 \* which case we try the node one past. We don't check any  
 \* further because we don't want to doubly traverse just to  
 \* find sentinel.  
 \*/  
 /\*  
 在最坏的情况下可能需要遍历整个栈来解除给定节点的链接。  
 在并发情况下，如果有其他线程已经移除节点，当前线程可能无法看到，  
 但是我们可以在能看到的已知节点上停止。使用s.next作为past节点，  
 如果past节点已经取消，则使用past.next节点，在这里不做更深的检查，  
 因为为了找到失效节点而进行两次遍历是不值的。  
 \*/  
 SNode past = s.next;  
 if (past != null && past.isCancelled())  
 past = past.next;  
  
 // Absorb cancelled nodes at head  
 SNode p;  
 while ((p = head) != null && p != past && p.isCancelled())  
 casHead(p, p.next);  
  
 // Unsplice embedded nodes  
 while (p != null && p != past) {  
 SNode n = p.next;  
 if (n != null && n.isCancelled())  
 p.casNext(n, n.next);  
 else  
 p = n;  
 }  
}

**说明：**移除从栈头节点开始到给定节点s之间的所有已取消节点。

* + - * 1. tryMatch(SNode s)

*/\*\*  
 \* Tries to match node s to this node, if so, waking up thread.  
 \* Fulfillers call tryMatch to identify their waiters.  
 \* Waiters block until they have been matched.  
 \*  
 \* 尝试匹配给定节点s，如果匹配唤醒当前waiter线程  
 \** ***@param*** *s the node to match  
 \** ***@return*** *true if successfully matched to s  
 \*/*boolean tryMatch(SNode s) {  
 if (match == null &&  
 *UNSAFE*.compareAndSwapObject(this, *matchOffset*, null, s)) {  
 Thread w = waiter;  
 if (w != null) { // waiters need at most one unpark  
 waiter = null;  
 LockSupport.*unpark*(w);  
 }  
 return true;  
 }  
 return match == s;  
}

**说明：**尝试匹配给定节点s，如果匹配唤醒当前节点的waiter线程。

###### TransferQueue.transfer()

* + - * 1. transfer(E e, boolean timed, long nanos)

*/\*\*  
 \* Puts or takes an item.  
 \*/*@SuppressWarnings("unchecked")  
E transfer(E e, boolean timed, long nanos) {  
 /\* Basic algorithm is to loop trying to take either of  
 \* two actions:  
 \*  
 \* 1. If queue apparently empty or holding same-mode nodes,  
 \* try to add node to queue of waiters, wait to be  
 \* fulfilled (or cancelled) and return matching item.  
 \*  
 \* 2. If queue apparently contains waiting items, and this  
 \* call is of complementary mode, try to fulfill by CAS'ing  
 \* item field of waiting node and dequeuing it, and then  
 \* returning matching item.  
 \*  
 \* In each case, along the way, check for and try to help  
 \* advance head and tail on behalf of other stalled/slow  
 \* threads.  
 \*  
 \* The loop starts off with a null check guarding against  
 \* seeing uninitialized head or tail values. This never  
 \* happens in current SynchronousQueue, but could if  
 \* callers held non-volatile/final ref to the  
 \* transferer. The check is here anyway because it places  
 \* null checks at top of loop, which is usually faster  
 \* than having them implicitly interspersed.  
 \*  
 \* 1. 若队列为空 / 队列中的尾节点和自己的模式相同, 则添加 node  
 \* 到队列中, 直到 timeout/interrupt/其他线程和这个线程匹配  
 \* timeout/interrupt awaitFulfill方法返回的是 node 本身  
 \* 匹配成功的话, 要么返回 null (producer返回的), 或匹配到的值 (consumer 返回的)  
 \*  
 \* 2. 队列不为空, 且队列的 head.next 节点是当前节点匹配的节点,  
 \* 进行数据的传递匹配, 并且通过 advanceHead 方法帮助先前 block 的节点dequeue  
 \*/  
  
 QNode s = null; // constructed/reused as needed  
 boolean isData = (e != null);//判断put or take  
  
 for (;;) {  
 QNode t = tail;  
 QNode h = head;  
 if (t == null || h == null) // saw uninitialized value  
 continue; // spin  
  
 if (h == t || t.isData == isData) { // empty or same-mode  
 QNode tn = t.next;  
 if (t != tail) // inconsistent read  
 continue;  
 if (tn != null) { //尾节点滞后，替换尾节点 lagging tail  
 advanceTail(t, tn);  
 continue;  
 }  
 if (timed && nanos <= 0) // can't wait  
 return null;  
 if (s == null)  
 s = new QNode(e, isData);  
 if (!t.casNext(null, s)) // failed to link in  
 continue;  
  
 advanceTail(t, s); // swing tail and wait  
 Object x = awaitFulfill(s, e, timed, nanos);  
 if (x == s) { // wait was cancelled  
 clean(t, s); //等待被中断，清除s节点  
 return null;  
 }  
  
 if (!s.isOffList()) { // not already unlinked  
 //s节点尚未出列  
 advanceHead(t, s); // unlink if head  
 if (x != null) // and forget fields  
 s.item = s;  
 s.waiter = null;  
 }  
 return (x != null) ? (E)x : e;  
  
 //take  
 } else { // complementary-mode  
 QNode m = h.next; // node to fulfill  
 if (t != tail || m == null || h != head)  
 continue; // inconsistent read  
  
 Object x = m.item;  
 if (isData == (x != null) || // m already fulfilled  
 x == m || //m.item=m, m cancelled  
 !m.casItem(x, e)) { // lost CAS  
 advanceHead(h, m); // dequeue and retry  
 continue;  
 }  
  
 advanceHead(h, m); //匹配成功，head出列 successfully fulfilled  
 LockSupport.*unpark*(m.waiter); //唤醒m的put线程  
 return (x != null) ? (E)x : e;  
 }  
 }  
}

**说明：**循环尝试两个动作中的其中一个：

1. 若队列为空 / 队列中的尾节点和自己的模式相同, 则添加 node 到队列中, 直到 timeout/interrupt/其他线程和这个线程匹配timeout/interrupt awaitFulfill方法返回的是 node 本身匹配成功的话, 要么返回 null (producer返回的), 或匹配到的值 (consumer 返回的)  
2. 队列不为空, 且队列的 head.next 节点是当前节点匹配的节点,进行数据的传递匹配, 并且通过 advanceHead 方法帮助先前 block 的节点dequeue

transfer在put处理中调用了awaitFulfill，采用的算法跟TransferStack. awaitFulfill一致，这里不在赘述。

* + - * 1. clean(QNode pred, QNode s)

在等待被中断时会调用clean方法清除新增的节点s，源码如下：

*/\*\*  
 \* Gets rid of cancelled node s with original predecessor pred.  
 \* 移除已取消节点s  
 \*/*void clean(QNode pred, QNode s) {  
 s.waiter = null; // forget thread  
 /\*  
 \* At any given time, exactly one node on list cannot be  
 \* deleted -- the last inserted node. To accommodate this,  
 \* if we cannot delete s, we save its predecessor as  
 \* "cleanMe", deleting the previously saved version  
 \* first. At least one of node s or the node previously  
 \* saved can always be deleted, so this always terminates.  
 \*  
 \* 在任何时候都存在一个不能删除的节点-最后被插入的节点。为了满足  
 \* 这一点，如果我们无法删除s节点，就首先删除先前保存的节点s.pred，  
 \* 然后我们会保存它的前继节点作为"cleanMe" 节点。在s节点和s.pred  
 \* 节点中至少有一个是可以删除的,所以这总是终结节点。  
 \*/  
 while (pred.next == s) { // Return early if already unlinked  
 QNode h = head;  
 QNode hn = h.next; // Absorb cancelled first node as head  
 if (hn != null && hn.isCancelled()) {  
 advanceHead(h, hn);  
 continue;  
 }  
 QNode t = tail; // Ensure consistent read for tail  
 if (t == h)//队列为空，直接返回  
 return;  
 QNode tn = t.next;  
 if (t != tail)//tail节点被其他线程修改，重新循环  
 continue;  
 if (tn != null) {//tail.next不为空，修改tail，重新循环  
 advanceTail(t, tn);  
 continue;  
 }  
 if (s != t) { // If not tail, try to unsplice  
 QNode sn = s.next;  
 if (sn == s || pred.casNext(s, sn))//cas删除s节点  
 return;  
 }  
 //s是队列尾节点  
 //清除cleanMe节点  
 QNode dp = cleanMe;  
 if (dp != null) { // Try unlinking previous cancelled node  
 QNode d = dp.next;  
 QNode dn;  
 if (d == null || // d is gone or  
 d == dp || // d is off list or  
 !d.isCancelled() || // d not cancelled or  
 (d != t && // d not tail and  
 (dn = d.next) != null && // has successor  
 dn != d && // that is on list  
 dp.casNext(d, dn))) // d unspliced  
 casCleanMe(dp, null);  
 if (dp == pred)  
 return; // s is already saved node  
 } else if (casCleanMe(null, pred))//原cleanMe为空，标记pred为cleanMe，延迟清除s节点  
 return; // Postpone cleaning s  
 }  
}

**说明：**方法参数中s为线程中断已经取消的节点，pred为s的前继节点。  
在任何时候都存在一个**不能删除的节点-最后被插入的节点**。为了满足这一点，如果我们无法删除s节点，就首先删除先前保存的节点s.pred， 然后我们会保存它的前继节点作为"cleanMe" 节点。在s节点和s.pred节点中至少有一个是可以删除的,所以这总是终结节点。

**小结：**

SQ是一个内部没有容量的阻塞队列，适合一对一生产-消费的场景。后面的线程池源码解析中会使用到。

#### LinkedTransferQueue

##### 概述

LinkedTransferQueue(以下称LTQ)是1.7新增的基于链表实现的FIFO无界阻塞队列。  
采用一种**预占模式**。意思就是消费者线程取元素时，如果队列为空，那就生成一个节点（节点元素为null）入队，然后消费者线程被等待在这个节点上，后面生产者线程入队时发现有一个元素为null的节点，生产者线程就不入队了，直接就将元素填充到该节点，唤醒该节点等待的线程，被唤醒的消费者线程取走元素，从调用的方法返回。即找到匹配的节点不入队，找不到根据how参数入队

##### 数据结构



public class LinkedTransferQueue<E> extends AbstractQueue<E>  
 implements TransferQueue<E>, java.io.Serializable {  
 private static final long *serialVersionUID* = -3223113410248163686L;

*/\*\* True if on multiprocessor \*/  
/\*\*是否为多处理器\*/*private static final boolean *MP* =  
 Runtime.*getRuntime*().availableProcessors() > 1;

*/\*\*  
 \* The number of times to spin (with randomly interspersed calls  
 \* to Thread.yield) on multiprocessor before blocking when a node  
 \* is apparently the first waiter in the queue. See above for  
 \* explanation. Must be a power of two. The value is empirically  
 \* derived -- it works pretty well across a variety of processors,  
 \* numbers of CPUs, and OSes.  
 \*  
 \* 当一个节点是队列中的第一个waiter时，在多处理器上进行自旋的次数(随机穿插调用thread.yield)  
 \*/*private static final int *FRONT\_SPINS* = 1 << 7;

*/\*\*  
 \* The number of times to spin before blocking when a node is  
 \* preceded by another node that is apparently spinning. Also  
 \* serves as an increment to FRONT\_SPINS on phase changes, and as  
 \* base average frequency for yielding during spins. Must be a  
 \* power of two.  
 \*  
 \* 当前继节点正在处理，当前节点在阻塞之前的自旋次数，也为FRONT\_SPINS  
 \* 的位变化充当增量，也可在自旋时作为yield的平均频率  
 \*/*private static final int *CHAINED\_SPINS* = *FRONT\_SPINS* >>> 1;

*/\*\*  
 \* The maximum number of estimated removal failures (sweepVotes)  
 \* to tolerate before sweeping through the queue unlinking  
 \* cancelled nodes that were not unlinked upon initial  
 \* removal. See above for explanation. The value must be at least  
 \* two to avoid useless sweeps when removing trailing nodes.  
 \*  
 \* sweepVotes的阀值  
 \* 为了避免无用的扫描，当移除后续节点时值必须大于等于2  
 \*/*static final int *SWEEP\_THRESHOLD* = 32;

static final class Node {  
 final boolean isData; // false if this is a request node  
 volatile Object item; // initially non-null if isData; CASed to match  
 volatile Node next;  
 volatile Thread waiter; // null until waiting  
  
 // CAS methods for fields  
 final boolean casNext(Node cmp, Node val) {  
 return *UNSAFE*.compareAndSwapObject(this, *nextOffset*, cmp, val);  
 }  
  
 final boolean casItem(Object cmp, Object val) {  
 // assert cmp == null || cmp.getClass() != Node.class;  
 return *UNSAFE*.compareAndSwapObject(this, *itemOffset*, cmp, val);  
 }  
  
 */\*\*  
 \* Constructs a new node. Uses relaxed write because item can  
 \* only be seen after publication via casNext.  
 \* 由于item只有在通过casNext公开之后才会被看到，这里使用UNSAFE直接  
 \* 修改内存数据  
 \*/* Node(Object item, boolean isData) {  
 *UNSAFE*.putObject(this, *itemOffset*, item); // relaxed write  
 this.isData = isData;  
 }  
  
 */\*\*  
 \* Links node to itself to avoid garbage retention. Called  
 \* only after CASing head field, so uses relaxed write.  
 \* next节点指向自身避免垃圾保留。只有在cas修改head时调用  
 \*/* final void forgetNext() {  
 *UNSAFE*.putObject(this, *nextOffset*, this);  
 }  
  
 */\*\*  
 \* Sets item to self and waiter to null, to avoid garbage  
 \* retention after matching or cancelling. Uses relaxed writes  
 \* because order is already constrained in the only calling  
 \* contexts: item is forgotten only after volatile/atomic  
 \* mechanics that extract items. Similarly, clearing waiter  
 \* follows either CAS or return from park (if ever parked;  
 \* else we don't care).  
 \* 匹配或取消节点后使节点item指向自身，waiter设为null，避免垃圾保留。  
 \*/* final void forgetContents() {  
 *UNSAFE*.putObject(this, *itemOffset*, this);  
 *UNSAFE*.putObject(this, *waiterOffset*, null);  
 }  
  
 */\*\*  
 \* Returns true if this node has been matched, including the  
 \* case of artificial matches due to cancellation.  
 \*/* final boolean isMatched() {  
 Object x = item;  
 return (x == this) || ((x == null) == isData);  
 }  
  
 */\*\*  
 \* Returns true if this is an unmatched request node.  
 \*/* final boolean isUnmatchedRequest() {  
 return !isData && item == null;  
 }  
  
 */\*\*  
 \* Returns true if a node with the given mode cannot be  
 \* appended to this node because this node is unmatched and  
 \* has opposite data mode.  
 \*/* final boolean cannotPrecede(boolean haveData) {  
 boolean d = isData;  
 Object x;  
 return d != haveData && (x = item) != this && (x != null) == d;  
 }  
  
 */\*\*  
 \* Tries to artificially match a data node -- used by remove.  
 \*/* final boolean tryMatchData() {  
 // assert isData;  
 Object x = item;  
 if (x != null && x != this && casItem(x, null)) {  
 LockSupport.*unpark*(waiter);  
 return true;  
 }  
 return false;  
 }  
  
 private static final long *serialVersionUID* = -3375979862319811754L;  
  
 // Unsafe mechanics  
 private static final sun.misc.Unsafe *UNSAFE*;  
 private static final long *itemOffset*;  
 private static final long *nextOffset*;  
 private static final long *waiterOffset*;  
 static {  
 try {  
 *UNSAFE* = sun.misc.Unsafe.*getUnsafe*();  
 Class<?> k = Node.class;  
 *itemOffset* = *UNSAFE*.objectFieldOffset  
 (k.getDeclaredField("item"));  
 *nextOffset* = *UNSAFE*.objectFieldOffset  
 (k.getDeclaredField("next"));  
 *waiterOffset* = *UNSAFE*.objectFieldOffset  
 (k.getDeclaredField("waiter"));  
 } catch (Exception e) {  
 throw new Error(e);  
 }  
 }  
}

*/\*\* head of the queue; null until first enqueue \*/*//队列头节点，第一次入列之前为空  
transient volatile Node head;  
  
*/\*\* tail of the queue; null until first append \*/*//队列尾节点，第一次添加节点之前为空  
private transient volatile Node tail;  
  
*/\*\* The number of apparent failures to unsplice removed nodes \*/*//累计到一定次数再清除无效node  
private transient volatile int sweepVotes;

/\*  
 \* Possible values for "how" argument in xfer method.  
 \* xfer方法类型  
 \*/  
private static final int *NOW* = 0; // for untimed poll, tryTransfer  
private static final int *ASYNC* = 1; // for offer, put, add  
private static final int *SYNC* = 2; // for transfer, take  
private static final int *TIMED* = 3; // for timed poll, tryTransfer

##### 函数列表

*/\*\*  
 \* Creates an initially empty {****@code*** *LinkedTransferQueue}.  
 \*/*public LinkedTransferQueue() {  
}  
  
*/\*\*  
 \* Creates a {****@code*** *LinkedTransferQueue}  
 \* initially containing the elements of the given collection,  
 \* added in traversal order of the collection's iterator.  
 \*  
 \** ***@param*** *c the collection of elements to initially contain  
 \** ***@throws*** *NullPointerException if the specified collection or any  
 \* of its elements are null  
 \*/*public LinkedTransferQueue(Collection<? extends E> c) {  
 this();  
 addAll(c);  
}

*/\*\*添加指定元素到队列尾\*/*public void put(E e) {  
 xfer(e, true, *ASYNC*, 0);  
}

*/\*\*  
 \* 插入指定元素到队列尾，返回true  
 \*/*public boolean offer(E e, long timeout, TimeUnit unit) {  
 xfer(e, true, *ASYNC*, 0);  
 return true;  
}

*/\*\*  
 \* 添加指定元素到队列尾{****@link*** *Queue#offer}  
 \*/*public boolean offer(E e) {  
 xfer(e, true, *ASYNC*, 0);  
 return true;  
}

*/\*\*  
 \* 添加指定元素到队列尾{****@link*** *Collection#add}  
 \*/*public boolean add(E e) {  
 xfer(e, true, *ASYNC*, 0);  
 return true;  
}

*/\*\*  
 \* 如果有消费者正在等待接收元素，立即把给定元素转移给等待的消费者  
 \* 无消费者返回false  
 \*/*public boolean tryTransfer(E e) {  
 return xfer(e, true, *NOW*, 0) == null;  
}

*/\*\*  
 \* 如果有消费者正在等待接收元素，立即把给定元素转移给等待的消费者  
 \* 否则插入给定元素到队列尾并等待直到元素被消费者接收  
 \*/*public void transfer(E e) throws InterruptedException {  
 if (xfer(e, true, *SYNC*, 0) != null) {  
 Thread.*interrupted*(); // failure possible only due to interrupt  
 throw new InterruptedException();  
 }  
}

public boolean tryTransfer(E e, long timeout, TimeUnit unit)

public E take() throws InterruptedException {  
 E e = xfer(null, false, *SYNC*, 0);  
 if (e != null)  
 return e;  
 Thread.*interrupted*();  
 throw new InterruptedException();  
}  
  
public E poll(long timeout, TimeUnit unit) throws InterruptedException {  
 E e = xfer(null, false, *TIMED*, unit.toNanos(timeout));  
 if (e != null || !Thread.*interrupted*())  
 return e;  
 throw new InterruptedException();  
}  
  
public E poll() {  
 return xfer(null, false, *NOW*, 0);  
}

public int drainTo(Collection<? super E> c)

public int drainTo(Collection<? super E> c, int maxElements)

public E peek() {  
 return firstDataItem();  
}

public boolean remove(Object o) {  
 return findAndRemove(o);  
}

public boolean contains(Object o)

##### 源码解析

**通过上节“函数列表”中可以看出，LTQ的添加（add、put、offer）和获取（poll、take）操作以及转移（transfer）操作全部都是通过xfer(E e, boolean haveData, int how, long nanos)方法实现，所以这里值对xfer方法进行解析。**

###### xfer(E e, boolean haveData, int how, long nanos)

*/\*\*  
 \* Implements all queuing methods. See above for explanation.  
 \*  
 \** ***@param*** *e the item or null for take  
 \** ***@param*** *haveData true if this is a put, else a take  
 \** ***@param*** *how NOW, ASYNC, SYNC, or TIMED  
 \** ***@param*** *nanos timeout in nanosecs, used only if mode is TIMED  
 \** ***@return*** *an item if matched, else e  
 \** ***@throws*** *NullPointerException if haveData mode but e is null  
 \*/*private E xfer(E e, boolean haveData, int how, long nanos) {  
 if (haveData && (e == null))  
 throw new NullPointerException();  
 Node s = null; // the node to append, if needed  
  
 retry:  
 for (;;) { // restart on append race  
 //从head开始向后匹配  
 for (Node h = head, p = h; p != null;) { // find & match first node  
 boolean isData = p.isData;  
 Object item = p.item;  
 if (item != p && (item != null) == isData) { //未找到有效节点或不匹配，p=p.next继续循环 unmatched  
 if (isData == haveData) //节点与此次操作模式一致，无法匹配 can't match  
 break;  
 if (p.casItem(item, e)) { // match  
 for (Node q = p; q != h;) {  
 Node n = q.next; // update by 2 unless singleton  
 if (head == h && casHead(h, n == null ? q : n)) {  
 h.forgetNext();//老head节点指向自身等待回收  
 break;  
 } // advance and retry  
 if ((h = head) == null ||  
 (q = h.next) == null || !q.isMatched())  
 break; // unless slack < 2  
 }  
 LockSupport.*unpark*(p.waiter);  
 return LinkedTransferQueue.<E>*cast*(item);  
 }  
 }  
 Node n = p.next;  
 p = (p != n) ? n : (h = head); // Use head if p offlist  
 }  
  
 if (how != *NOW*) { // No matches available  
 if (s == null)  
 s = new Node(e, haveData);  
 Node pred = tryAppend(s, haveData);  
 if (pred == null)  
 continue retry; // lost race vs opposite mode  
 if (how != *ASYNC*)  
 return awaitMatch(s, pred, e, (how == *TIMED*), nanos);  
 }  
 return e; // not waiting  
 }  
}

**说明：**xfer的基本流程如下：

1. 从head开始向后匹配，找到一个节点模式跟本次操作的模式不同的节点（生产或消费）进行匹配；
2. 匹配节点成功CAS修改匹配节点的item为给定元素e；
3. 如果此时所匹配节点向后移动，则CAS更新head节点，旧head节点链接指向自身等待被回收；如果在并发情况下竞争失败，则继续向下寻找节点；
4. 唤醒匹配节点p的等待线程waiter，返回匹配的item。
5. 如果在上述操作中没有找到匹配节点，则根据how做不同的处理：
   1. NOW：立即返回。for untimed poll,tryTransfer.
   2. SYNC：插入一个新的节点s(item=e,isData = haveData)到队列尾，然后自旋或阻塞当前线程直到节点被匹配或者取消。for offer,put,add.
   3. ASYNC：插入一个新的节点s(item=e,isData = haveData)到队列尾，异步直接返回。For transfer,take.
   4. TIMED：插入一个新的节点s(item=e,isData = haveData)到队列尾，等待超时返回。for timed poll,tryTransfer.

###### tryAppend(Node s, Boolean haveData)

*/\*\*  
 \* Tries to append node s as tail.  
 \* 尝试添加给定节点s作为尾节点  
 \*  
 \** ***@param*** *s the node to append  
 \** ***@param*** *haveData true if appending in data mode  
 \** ***@return*** *null on failure due to losing race with append in  
 \* different mode, else s's predecessor, or s itself if no  
 \* predecessor  
 \*/*private Node tryAppend(Node s, boolean haveData) {  
 for (Node t = tail, p = t;;) { // move p to last node and append  
 Node n, u; // temps for reads of next & tail  
 if (p == null && (p = head) == null) {//head和tail都为null  
 if (casHead(null, s))//修改head为新节点s  
 return s; // initialize  
 }  
 else if (p.cannotPrecede(haveData))  
 return null; // lost race vs opposite mode  
 else if ((n = p.next) != null) // not last; keep traversing  
 p = p != t && t != (u = tail) ? (t = u) : // stale tail  
 (p != n) ? n : null; // restart if off list  
 else if (!p.casNext(null, s))  
 p = p.next; // re-read on CAS failure  
 else {  
 if (p != t) { // update if slack now >= 2  
 while ((tail != t || !casTail(t, s)) &&  
 (t = tail) != null &&  
 (s = t.next) != null && // advance and retry  
 (s = s.next) != null && s != t);  
 }  
 return p;  
 }  
 }  
}

*/\*\*  
 \* Returns true if a node with the given mode cannot be  
 \* appended to this node because this node is unmatched and  
 \* has opposite data mode.  
 \* 如果节点模式和给定mode不同返回true，因为此节点未匹配并且有相反的模式  
 \*/*final boolean cannotPrecede(boolean haveData) {  
 boolean d = isData;  
 Object x;  
 return d != haveData && (x = item) != this && (x != null) == d;  
}

**说明：**添加给定节点s到队列尾并返回s的前继节点，失败时（与其他不同模式线程竞争失败）返回null，没有前继节点返回自身。

###### awaitMatch (Node s, Node pred, E e, boolean timed, long nanos)

*/\*\*  
 \* Spins/yields/blocks until node s is matched or caller gives up.  
 \* 自旋/让步/阻塞,直到给定节点s匹配到或放弃匹配  
 \*  
 \** ***@param*** *s the waiting node  
 \** ***@param*** *pred the predecessor of s, or s itself if it has no  
 \* predecessor, or null if unknown (the null case does not occur  
 \* in any current calls but may in possible future extensions)  
 \** ***@param*** *e the comparison value for checking match  
 \** ***@param*** *timed if true, wait only until timeout elapses  
 \** ***@param*** *nanos timeout in nanosecs, used only if timed is true  
 \** ***@return*** *matched item, or e if unmatched on interrupt or timeout  
 \*/*private E awaitMatch(Node s, Node pred, E e, boolean timed, long nanos) {  
 final long deadline = timed ? System.*nanoTime*() + nanos : 0L;  
 Thread w = Thread.*currentThread*();  
 //在首个item和取消检查后初始  
 int spins = -1; // initialized after first item and cancel checks  
 ThreadLocalRandom randomYields = null; // bound if needed  
  
 for (;;) {  
 Object item = s.item;  
 if (item != e) { //matched  
 // assert item != s;  
 s.forgetContents(); // avoid garbage  
 return LinkedTransferQueue.<E>*cast*(item);  
 }  
 if ((w.isInterrupted() || (timed && nanos <= 0)) &&  
 s.casItem(e, s)) { //取消匹配，item指向自身 cancel  
 unsplice(pred, s);//解除s节点和前继节点的链接  
 return e;  
 }  
  
 if (spins < 0) { // establish spins at/near front  
 if ((spins = *spinsFor*(pred, s.isData)) > 0)  
 randomYields = ThreadLocalRandom.*current*();  
 }  
 else if (spins > 0) { // spin  
 --spins;  
 if (randomYields.nextInt(*CHAINED\_SPINS*) == 0)  
 Thread.*yield*(); //不定期让步，给其他线程执行机会 occasionally yield  
 }  
 else if (s.waiter == null) {  
 s.waiter = w; // request unpark then recheck  
 }  
 else if (timed) {  
 nanos = deadline - System.*nanoTime*();  
 if (nanos > 0L)  
 LockSupport.*parkNanos*(this, nanos);  
 }  
 else {  
 LockSupport.*park*(this);  
 }  
 }  
}

**说明：**当前操作为同步操作时，会调用awaitMatch方法阻塞等待匹配，成功返回匹配节点item，失败返回给定参数e（s.item）。在等待期间如果线程被中断或等待超时，则取消匹配，并调用unsplice方法解除节点s和其前继节点的链接。

###### unsplice (Node pred, Node s)

*/\*\*  
 \* Unsplices (now or later) the given deleted/cancelled node with  
 \* the given predecessor.  
 \*  
 \* 解除给定已经被删除/取消节点和前继节点的链接（可能延迟解除）  
 \** ***@param*** *pred a node that was at one time known to be the  
 \* predecessor of s, or null or s itself if s is/was at head  
 \** ***@param*** *s the node to be unspliced  
 \*/*final void unsplice(Node pred, Node s) {  
 s.forgetContents(); // forget unneeded fields  
 /\*  
 \* See above for rationale. Briefly: if pred still points to  
 \* s, try to unlink s. If s cannot be unlinked, because it is  
 \* trailing node or pred might be unlinked, and neither pred  
 \* nor s are head or offlist, add to sweepVotes, and if enough  
 \* votes have accumulated, sweep.  
 \* 原理：如果pred(s.pred)==s，尝试解除s的链接。如果s不能被解除(由于  
 \* 他是尾节点或者s.pred可能被解除链接，并且pred和s都不是head节点或已经出列)，  
 \* 则添加到sweepVotes，累计到一定数量再清除  
 \*/  
 if (pred != null && pred != s && pred.next == s) {  
 Node n = s.next;  
 if (n == null ||  
 (n != s && pred.casNext(s, n) && pred.isMatched())) {  
 for (;;) { // check if at, or could be, head  
 Node h = head;  
 if (h == pred || h == s || h == null)  
 return; // at head or list empty  
 if (!h.isMatched())  
 break;  
 Node hn = h.next;  
 if (hn == null)  
 return; // now empty  
 if (hn != h && casHead(h, hn))  
 h.forgetNext(); // advance head  
 }  
 if (pred.next != pred && s.next != s) { // recheck if offlist  
 for (;;) { // sweep now if enough votes  
 int v = sweepVotes;  
 if (v < *SWEEP\_THRESHOLD*) {  
 if (casSweepVotes(v, v + 1))  
 break;  
 }  
 else if (casSweepVotes(v, 0)) {//达到阀值，清除无效节点  
 sweep();  
 break;  
 }  
 }  
 }  
 }  
 }  
}

*/\*\*  
 \* Unlinks matched (typically cancelled) nodes encountered in a  
 \* traversal from head.  
 \* 解除(通常是取消)从头部遍历时遇到的已经被匹配的节点的链接  
 \*/*private void sweep() {  
 for (Node p = head, s, n; p != null && (s = p.next) != null; ) {  
 if (!s.isMatched())  
 // Unmatched nodes are never self-linked  
 p = s;  
 else if ((n = s.next) == null) // trailing node is pinned  
 break;  
 else if (s == n) // stale  
 // No need to also check for p == s, since that implies s == n  
 p = head;  
 else  
 p.casNext(s, n);  
 }  
}

**说明：**此方法的作用是解除给定已经被删除/取消节点和其前继节点的链接，可能会延迟解除（先指向自身等sweepVotes到达阀值调用sweep清除）。基本原理：如果pred(s.pred)==s，尝试解除s的链接。如果s不能被解除(由于他是尾节点或者s.pred可能被解除链接，并且pred和s都不是head节点或已经出列)，则增加sweepVotes，累计到一定数量再清除

**注意：LinkedTransfer会因中断造成数据暂失。具体分析见** [**LinkedTransferQueue的数据暂失和CPU爆满以及修复**](http://ifeve.com/buglinkedtransferqueue-bug/#more-11117)

### JUC 线程池篇

#### 框架



1. TheadPoolExecutor 线程池类，对于线程池，可以通俗的将它理解为"存放一定数量线程的一个线程集合。线程池允许多个线程同时运行，允许同时运行的线程数量就是线程池的容量；当添加的到线程池中的线程超过它的容量时，会有一部分线程阻塞等待。线程池会通过相应的调度策略和拒绝策略，对添加到线程池中的线程进行管理。
2. ScheduledThreadPoolExecutor 属于线程池的一种，它可以允许任务延迟或周期执行，类似java的Timer。
3. ForkJoinPool 也是线程池的一种，采用分治算法，把大任务分成若干个小任务执行，最后再合并结果集。

##### Excutor

线程池的最上级接口，只有一个方法execute()来接收执行提交的Runnable任务对象，源码如下：

public interface Executor {  
  
 */\*\*  
 \* Executes the given command at some time in the future. The command  
 \* may execute in a new thread, in a pooled thread, or in the calling  
 \* thread, at the discretion of the {****@code*** *Executor} implementation.  
 \* 在某个时刻执行给到的命令。可能是一个新的线程，在池中的线程，也可能是正在  
 \* 调用的线程，取决于具体实现  
 \** ***@param*** *command the runnable task  
 \** ***@throws*** *RejectedExecutionException if this task cannot be  
 \* accepted for execution  
 \** ***@throws*** *NullPointerException if command is null  
 \*/* void execute(Runnable command);  
}

##### ExcutorService

继承自Excutor，提供关闭线程池(shutdown)和返回异步回调(Future)函数的方法(submit)。函数列表如下：

public interface ExecutorService extends Executor {

/*\*\*  
 \* 发送一个shutdown指令，等待以前的任务执行完毕再关闭，但是不再接收  
 \* 新的任务  
 \*/*void shutdown();

*/\*\*  
 \* 尝试停止所有正在执行的任务，停止等待任务的处理，并返回等待执行的任务列表  
 \*/*List<Runnable> shutdownNow();

boolean isShutdown();

boolean isTerminated();

*/\*\*  
 \* 发送一个shutdown请求之后开始阻塞，直到所有任务都完成执行，或者超时，或  
 \* 当前线程被中断  
 \*/*  
boolean awaitTermination(long timeout, TimeUnit unit)

*/\*\*  
 \* 提交一个有返回值的任务并返回Future（表示任务的未决结果）  
 \* 任务执行成功后可调用Future.get获取任务返回的结果  
 \*/*  
<T> Future<T> submit(Callable<T> task);

<T> Future<T> submit(Runnable task, T result);

Future<?> submit(Runnable task);

<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)

<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,  
 long timeout, TimeUnit unit)

*/\*\*  
 \* 返回任意一个已经完成执行的结果  
 \*/*<T> T invokeAny(Collection<? extends Callable<T>> tasks)

<T> T invokeAny(Collection<? extends Callable<T>> tasks,  
 long timeout, TimeUnit unit)

##### AbstractExecutorService

AbstractExecutorService实现了ExcutorService接口，为ExcutorService提供默认的实现。函数列表与ExcutorService一致，不在赘述。

##### ScheduledExecutorService

继承自ExcutorService，提供任务延迟和周期执行。函数列表如下：

public interface ScheduledExecutorService extends ExecutorService {

*/\*\*  
 \* 创建并执行在给定延迟后启用的一次性操作  
 \*/*public ScheduledFuture<?> schedule(Runnable command,  
 long delay, TimeUnit unit);

public <V> ScheduledFuture<V> schedule(Callable<V> callable,  
 long delay, TimeUnit unit);

*/\*\*  
 \* 创建一个周期执行的任务，第一次执行延期时间为initialDelay，  
 \* 之后每隔period执行一次  
 \* \*/*public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,  
 long initialDelay,  
 long period,  
 TimeUnit unit);

*/\*\*  
 \* 创建一个周期执行的任务，第一次执行延期时间为initialDelay，  
 \* 在第一次执行完之后延迟delay后开始下一次执行  
 \* \*/*public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,  
 long initialDelay,  
 long delay,  
 TimeUnit unit);

##### ThreadPoolExecutor

###### 概述

线程池核心类，内部存储了一定数量的线程。线程池解决了两个不同的问题：由于每个任务的调用开销减少，它们通常在执行大量异步任务时提供更好的性能；当执行任务集合时也提供了绑定和管理资源的方式，包括线程、消费者。每一个ThreadPoolExecutor也都维护了一些基础的运行统计，例如已完成的任务数。

线程的创建和销毁需要很大的开销，线程池中预先提供了指定数量的可重用线程，所以使用线程池会节省系统资源，方便线程的管理和监控。

###### 数据结构和核心参数



*/\*\*当运行任务已满，新增任务的存储队列\*/*private final BlockingQueue<Runnable> workQueue;

*/\*\*线程运行期间的锁，在调用shutdown和shutdownNow之后依然持有\*/*private final ReentrantLock mainLock = new ReentrantLock();

*/\*\*工作线程池，只有在持有mainLock才存储\*/*private final HashSet<Worker> workers = new HashSet<Worker>();

*/\*\*awaitTermination的等待条件\*/*private final Condition termination = mainLock.newCondition();

*/\*\*最大池容量\*/*private int largestPoolSize;

*/\*\*已完成任务数量\*/*private long completedTaskCount;

*/\*\*线程工厂，所有线程都是用它来创建(通过addWorker方法)\*/*private volatile ThreadFactory threadFactory;

*/\*\*在执行期间调用饱和或关闭时的处理\*/*private volatile RejectedExecutionHandler handler;

*/\*\*空闲线程保活时长\*/*private volatile long keepAliveTime;

*/\*\*默认false，表示core线程空闲依然保活；  
 \* 如果为true，使用keepAliveTime确定等待超时时间\*/*private volatile boolean allowCoreThreadTimeOut;

*/\*\*核心线程池大小  
 \* 超过核心线程数之后提交的任务将被放到等待队列中  
 \* \*/*private volatile int corePoolSize;

*/\*\*最大线程池大小  
 \* 如果当前等待队列任务已满，继续提交的任务将继续创建新的线程执行，这个线程数最大为maximumPoolSize  
 \* \*/*private volatile int maximumPoolSize;

*/\*\*默认失败策略\*/*private static final RejectedExecutionHandler *defaultHandler* = new AbortPolicy();

*/\*\*针对shutdown和shutdownNow的运行权限许可\*/*private static final RuntimePermission *shutdownPerm* =  
 new RuntimePermission("modifyThread");

线程池状态

**RUNNING**：接收新的任务和队列任务  
**SHUTDOWN**：不接收新任务，允许队列任务  
**STOP**：不接收新任务，不接收队列任务，并且中断正在运行任务  
**TIDYING**：所有任务都被销毁，workCount=0，当调用terminated()方法时线程池状态转换为TIDYING  
**TERMINATED**：调用terminated方法成功



线程池状态码定义：

//代表两种概念：workerCount(有效线程数)和runState(线程池状态)  
//低29位表示线程池中线程数，高3位表示线程池的运行状态  
private final AtomicInteger ctl = new AtomicInteger(*ctlOf*(*RUNNING*, 0));  
private static final int *COUNT\_BITS* = Integer.*SIZE* - 3;//任务线程数量所占的int的位数  
private static final int *CAPACITY* = (1 << *COUNT\_BITS*) - 1;//最大任务线程数量为2^29-1  
  
// runState is stored in the high-order bits  
private static final int *RUNNING* = -1 << *COUNT\_BITS*;//对应高三位111  
private static final int *SHUTDOWN* = 0 << *COUNT\_BITS*;//对应高三位000  
private static final int *STOP* = 1 << *COUNT\_BITS*;//对应高三位001  
private static final int *TIDYING* = 2 << *COUNT\_BITS*;//对应高三位010  
private static final int *TERMINATED* = 3 << *COUNT\_BITS*;//对应高三位011  
  
// Packing and unpacking ctl  
private static int runStateOf(int c) { return c & ~*CAPACITY*; }//运行状态  
private static int workerCountOf(int c) { return c & *CAPACITY*; }//运行的任务线程数  
private static int ctlOf(int rs, int wc) { return rs | wc; }//包装运行状态和任务线程

线程池饱和策略

当线程池已经饱和（阻塞队列已满，且没有空闲的工作线程），继续提交的任务将通过一种策略来处理，线程池提供了四种饱和策略：

1. AbortPolicy：默认策略，直接抛出异常RejectedExecutionException；

*/\*\*  
 \* A handler for rejected tasks that throws a  
 \* {****@code*** *RejectedExecutionException}.  
 \*/*public static class AbortPolicy implements RejectedExecutionHandler {  
 */\*\*  
 \* Creates an {****@code*** *AbortPolicy}.  
 \*/* public AbortPolicy() { }  
  
 */\*\*  
 \* Always throws RejectedExecutionException.  
 \*  
 \** ***@param*** *r the runnable task requested to be executed  
 \** ***@param*** *e the executor attempting to execute this task  
 \** ***@throws*** *RejectedExecutionException always  
 \*/* public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {  
 throw new RejectedExecutionException("Task " + r.toString() +  
 " rejected from " +  
 e.toString());  
 }  
}

1. CallerRunsPolicy：直接在execute方法的调用线程中运行被拒绝的任务，如果线程池已经关闭，任务将被丢弃；

public static class CallerRunsPolicy implements RejectedExecutionHandler {  
 */\*\*  
 \* Creates a {****@code*** *CallerRunsPolicy}.  
 \*/* public CallerRunsPolicy() { }  
  
 */\*\*  
 \* Executes task r in the caller's thread, unless the executor  
 \* has been shut down, in which case the task is discarded.  
 \*  
 \** ***@param*** *r the runnable task requested to be executed  
 \** ***@param*** *e the executor attempting to execute this task  
 \*/* public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {  
 if (!e.isShutdown()) {  
 r.run();  
 }  
 }  
}

1. DiscardPolicy：直接丢弃任务；

public static class DiscardPolicy implements RejectedExecutionHandler {  
 */\*\*  
 \* Creates a {****@code*** *DiscardPolicy}.  
 \*/* public DiscardPolicy() { }  
  
 */\*\*  
 \* Does nothing, which has the effect of discarding task r.  
 \*  
 \** ***@param*** *r the runnable task requested to be executed  
 \** ***@param*** *e the executor attempting to execute this task  
 \*/* public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {  
 }  
}

1. DiscardOldestPolicy：丢弃队列中最前未处理的任务，并执行当前提交的任务，如果线程池已经关闭，任务将被丢弃。

public static class DiscardOldestPolicy implements RejectedExecutionHandler {  
 */\*\*  
 \* Creates a {****@code*** *DiscardOldestPolicy} for the given executor.  
 \*/* public DiscardOldestPolicy() { }  
  
 */\*\*  
 \* Obtains and ignores the next task that the executor  
 \* would otherwise execute, if one is immediately available,  
 \* and then retries execution of task r, unless the executor  
 \* is shut down, in which case task r is instead discarded.  
 \*  
 \** ***@param*** *r the runnable task requested to be executed  
 \** ***@param*** *e the executor attempting to execute this task  
 \*/* public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {  
 if (!e.isShutdown()) {  
 e.getQueue().poll();  
 e.execute(r);  
 }  
 }  
}

###### 函数列表

public ThreadPoolExecutor(int corePoolSize,  
 int maximumPoolSize,  
 long keepAliveTime,  
 TimeUnit unit,  
 BlockingQueue<Runnable> workQueue) {  
 this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,  
 Executors.*defaultThreadFactory*(), *defaultHandler*);  
}

public ThreadPoolExecutor(int corePoolSize,  
 int maximumPoolSize,  
 long keepAliveTime,  
 TimeUnit unit,  
 BlockingQueue<Runnable> workQueue,  
 ThreadFactory threadFactory) {  
 this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,  
 threadFactory, *defaultHandler*);  
}

public ThreadPoolExecutor(int corePoolSize,  
 int maximumPoolSize,  
 long keepAliveTime,  
 TimeUnit unit,  
 BlockingQueue<Runnable> workQueue,  
 RejectedExecutionHandler handler) {  
 this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,  
 Executors.*defaultThreadFactory*(), handler);  
}

public ThreadPoolExecutor(int corePoolSize,  
 int maximumPoolSize,  
 long keepAliveTime,  
 TimeUnit unit,  
 BlockingQueue<Runnable> workQueue,  
 ThreadFactory threadFactory,  
 RejectedExecutionHandler handler) {  
 if (corePoolSize < 0 ||  
 maximumPoolSize <= 0 ||  
 maximumPoolSize < corePoolSize ||  
 keepAliveTime < 0)  
 throw new IllegalArgumentException();  
 if (workQueue == null || threadFactory == null || handler == null)  
 throw new NullPointerException();  
 this.corePoolSize = corePoolSize;  
 this.maximumPoolSize = maximumPoolSize;  
 this.workQueue = workQueue;  
 this.keepAliveTime = unit.toNanos(keepAliveTime);  
 this.threadFactory = threadFactory;  
 this.handler = handler;  
}

*/\*\*提交任务\*/*public void execute(Runnable command)

*/\*\*关闭线程池，顺序关闭已经提交的任务，不接收新的任务  
 \* 此方法不等待已提交任务执行完毕  
 \*/*public void shutdown()

*/\*\*关闭线程池，尝试停止所有活跃任务，停止等待任务的执行，  
 \* 返回等待执行的任务列表  
 \*/*public List<Runnable> shutdownNow()

public boolean isShutdown()

*/\*\*调用shutdown或shutdownNow后返回true，但没有完全终止\*/*public boolean isTerminating()

public boolean isTerminated()

*/\*\*等待已提交任务执行完毕或超时再销毁\*/*public boolean awaitTermination(long timeout, TimeUnit unit)

public boolean allowsCoreThreadTimeOut() {  
 return allowCoreThreadTimeOut;  
}

public void allowCoreThreadTimeOut(boolean value)

*/\*\*获取等待队列\*/*public BlockingQueue<Runnable> getQueue() {  
 return workQueue;  
}

*/\*\*移除内部队列中的任务\*/*public boolean remove(Runnable task)

*/\*\*移除工作队列所有已被取消的Future任务\*/*public void purge()

###### 源码解析

* + - * 1. Worker

private final class Worker  
 extends AbstractQueuedSynchronizer  
 implements Runnable  
{  
 */\*\*  
 \* This class will never be serialized, but we provide a  
 \* serialVersionUID to suppress a javac warning.  
 \*/* private static final long *serialVersionUID* = 6138294804551838833L;  
  
 */\*\* Thread this worker is running in. Null if factory fails. \*/* final Thread thread;  
 */\*\* Initial task to run. Possibly null. \*/* Runnable firstTask;  
 */\*\* Per-thread task counter \*/* volatile long completedTasks;  
  
 */\*\*  
 \* Creates with given first task and thread from ThreadFactory.  
 \** ***@param*** *firstTask the first task (null if none)  
 \*/* Worker(Runnable firstTask) {  
 setState(-1); // inhibit interrupts until runWorker  
 this.firstTask = firstTask;  
 this.thread = getThreadFactory().newThread(this);  
 }  
  
 */\*\* Delegates main run loop to outer runWorker \*/* public void run() {  
 runWorker(this);  
 }  
  
 // Lock methods  
 //  
 // The value 0 represents the unlocked state.  
 // The value 1 represents the locked state.  
  
 protected boolean isHeldExclusively() {  
 return getState() != 0;  
 }  
  
 protected boolean tryAcquire(int unused) {  
 if (compareAndSetState(0, 1)) {  
 setExclusiveOwnerThread(Thread.*currentThread*());  
 return true;  
 }  
 return false;  
 }  
  
 protected boolean tryRelease(int unused) {  
 setExclusiveOwnerThread(null);  
 setState(0);  
 return true;  
 }  
  
 public void lock() { acquire(1); }  
 public boolean tryLock() { return tryAcquire(1); }  
 public void unlock() { release(1); }  
 public boolean isLocked() { return isHeldExclusively(); }  
  
 void interruptIfStarted() {  
 Thread t;  
 if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {  
 try {  
 t.interrupt();  
 } catch (SecurityException ignore) {  
 }  
 }  
 }  
}

**说明：**1. 继承自AQS实现同步器功能。在线程池中持有一个Worker集合，一个Worker对应一个工作线程，当线程池启动时，对应的worker会执行池中的任务，执行完毕后从阻塞队列里获取一个新的任务继续执行。  
2. 本身实现了Runnable接口，也就是说Worker本身也作为一个线程任务执行。  
3. 初始化时设置锁状态为-1（setState(-1)），防止中断。直到调用runWorker时放开。  
由runWorker实现run方法：

runWorker

//工作线程执行任务  
final void runWorker(Worker w) {  
 Thread wt = Thread.*currentThread*();  
 Runnable task = w.firstTask;  
 w.firstTask = null;  
 //任务线程的锁状态默认为-1，此时解锁+1，变为0，即锁打开状态，允许中断，在任务未执行之前，不允许中断。  
 w.unlock(); // allow interrupts  
 boolean completedAbruptly = true;//完成后是否可以中断  
 try {  
 while (task != null || (task = getTask()) != null) {  
 w.lock();  
 // If pool is stopping, ensure thread is interrupted;  
 // if not, ensure thread is not interrupted. This  
 // requires a recheck in second case to deal with  
 // shutdownNow race while clearing interrupt  
 //如果线程池正在stop，需要确保线程已经被中断  
 //否则，确保线程没有被中断。这里针对两种情况需要进行复查，  
 //以处理在清除中断时的shutdownNow事件  
 if ((*runStateAtLeast*(ctl.get(), *STOP*) ||  
 (Thread.*interrupted*() &&  
 *runStateAtLeast*(ctl.get(), *STOP*))) &&  
 !wt.isInterrupted())  
 wt.interrupt();  
 try {  
 beforeExecute(wt, task);//执行前逻辑，可用于日志记录  
 Throwable thrown = null;  
 try {  
 task.run();//执行任务  
 } catch (RuntimeException x) {  
 thrown = x; throw x;  
 } catch (Error x) {  
 thrown = x; throw x;  
 } catch (Throwable x) {  
 thrown = x; throw new Error(x);  
 } finally {  
 afterExecute(task, thrown);//执行后逻辑  
 }  
 } finally {  
 task = null;  
 w.completedTasks++;  
 w.unlock();  
 }  
 }  
 completedAbruptly = false;  
 } finally {  
 //处理工作线程退出逻辑  
 processWorkerExit(w, completedAbruptly);  
 }  
}

**说明：**工作线程核心运行方法，循环从等待队列获取任务并执行。  
在任务执行前后可调用beforeExecute和afterExecute处理执行前后的逻辑，这两个方法在线程池中都是空方法，可根据业务需求自定义实现。任务获取通过getTask实现，源码如下：

getTask

//获取任务，基于当前线程池配置执行任务阻塞或等待  
private Runnable getTask() {  
 boolean timedOut = false; // Did the last poll() time out?  
  
 for (;;) {  
 int c = ctl.get();  
 int rs = *runStateOf*(c);  
  
 // Check if queue empty only if necessary.  
 if (rs >= *SHUTDOWN* && (rs >= *STOP* || workQueue.isEmpty())) {  
 decrementWorkerCount();  
 return null;  
 }  
  
 int wc = *workerCountOf*(c);  
  
 // Are workers subject to culling?  
 boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;  
  
 if ((wc > maximumPoolSize || (timed && timedOut))  
 && (wc > 1 || workQueue.isEmpty())) {  
 if (compareAndDecrementWorkerCount(c))  
 return null;  
 continue;  
 }  
  
 try {  
 Runnable r = timed ?  
 workQueue.poll(keepAliveTime, TimeUnit.*NANOSECONDS*) :  
 workQueue.take();  
 if (r != null)  
 return r;  
 timedOut = true;  
 } catch (InterruptedException retry) {  
 timedOut = false;  
 }  
 }  
}

**说明：**获取等待队列中的任务，基于当前线程池的配置执行任务阻塞、等待或返回null。在以下四个情况下会引起worker退出，并返回null：  
 1.工作线程数大于maximumPoolSize  
 2.线程池已停止  
 3.线程池已关闭并且等待队列为空  
 4.工作线程等待任务超时

由于上述条件返回null后，需要递减workerCount

processWorkerExit(Worker, completedAbruptly)

工作线程任务执行完毕后，调用此方法来处理工作线程退出的逻辑。源码如下：

//处理工作线程退出逻辑  
private void processWorkerExit(Worker w, boolean completedAbruptly) {  
 if (completedAbruptly) // If abrupt, then workerCount wasn't adjusted  
 //如果任务线程被中断，则工作线程数量减1  
 decrementWorkerCount();  
  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
 completedTaskCount += w.completedTasks;  
 workers.remove(w);  
 } finally {  
 mainLock.unlock();  
 }  
  
 tryTerminate();//尝试终止线程池  
  
 int c = ctl.get();  
 if (*runStateLessThan*(c, *STOP*)) {//如果线程处于运行中  
 if (!completedAbruptly) {  
 //获取当前核心线程数  
 int min = allowCoreThreadTimeOut ? 0 : corePoolSize;  
 if (min == 0 && ! workQueue.isEmpty())  
 //如果允许空闲工作线程等待任务，且任务队列不为空，则min为1  
 min = 1;  
 if (*workerCountOf*(c) >= min)  
 return; // replacement not needed  
 }  
 addWorker(null, false);  
 }  
}

**说明：**

runWorker执行完毕后，调用此方法处理工作线程退出逻辑。为已经死亡的worker执行相关的清除操作。除非设置了completedAbruptly，否则假定workerCount已经被修改为退出状态。在下面几种情况下，此方法从worker集合中移除线程，并可能销毁线程池或者替换该worker：  
 1.用户任务执行异常导致线程退出  
 2.工作线程数少于corePoolSize  
 3.等待队列不为空但没有工作线程。

* + - * 1. tryTerminate

//尝试终止线程池  
final void tryTerminate() {  
 for (;;) {  
 int c = ctl.get();  
 if (*isRunning*(c) || //正在运行  
 *runStateAtLeast*(c, *TIDYING*) || //状态大于TIDYING  
 (*runStateOf*(c) == *SHUTDOWN* && ! workQueue.isEmpty())) //状态为shutdown并且等待队列不为空  
 return;  
 if (*workerCountOf*(c) != 0) { // Eligible to terminate  
 interruptIdleWorkers(*ONLY\_ONE*);//中断空闲线程  
 return;  
 }  
  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
 //线程池已经关闭，等待队列为空，并且工作线程等于0，更新池状态为TIDYING  
 if (ctl.compareAndSet(c, *ctlOf*(*TIDYING*, 0))) {  
 try {  
 terminated();//线程池销毁，需自定义实现  
 } finally {  
 ctl.set(*ctlOf*(*TERMINATED*, 0));  
 termination.signalAll();//唤醒等待池结束的线程  
 }  
 return;  
 }  
 } finally {  
 mainLock.unlock();  
 }  
 // else retry on failed CAS  
 }  
}

**说明：**如果线程池状态为SHUTDOWN并且等待队列和池任务都为空，或池状态为STOP且池任务为空，调用此方法可转换线程池状态为TERMINATED。

interruptIdleWorkers(boolean onlyOne)

*/\*\*  
 \* 中断空闲工作线程  
 \*/*private void interruptIdleWorkers(boolean onlyOne) {  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
 for (Worker w : workers) {  
 Thread t = w.thread;  
 if (!t.isInterrupted() && w.tryLock()) {  
 try {  
 t.interrupt();  
 } catch (SecurityException ignore) {  
 } finally {  
 w.unlock();  
 }  
 }  
 if (onlyOne)  
 break;  
 }  
 } finally {  
 mainLock.unlock();  
 }  
}

说明：如果onlyOne为true，只中断最多一个空闲工作线程。此方法在关闭线程池时或关闭的过程中工作线程完成任务后调用。

* + - * 1. addWorker (Runnable firstTask, boolean core)

*/\*\*  
 \* 检查当前池状态和给定界限中是否可以添加新worker  
 \*/*private boolean addWorker(Runnable firstTask, boolean core) {  
 //自旋，判断可以添加线程的前提条件  
 retry:  
 for (;;) {  
 int c = ctl.get();  
 int rs = *runStateOf*(c);  
  
 // Check if queue empty only if necessary.  
 //判断池状态  
 if (rs >= *SHUTDOWN* &&  
 ! (rs == *SHUTDOWN* &&  
 firstTask == null &&  
 ! workQueue.isEmpty()))  
 return false;  
  
 for (;;) {  
 int wc = *workerCountOf*(c);  
 if (wc >= *CAPACITY* ||  
 wc >= (core ? corePoolSize : maximumPoolSize))  
 return false;  
 if (compareAndIncrementWorkerCount(c))  
 break retry;  
 c = ctl.get(); // Re-read ctl  
 if (*runStateOf*(c) != rs)  
 continue retry;  
 // else CAS failed due to workerCount change; retry inner loop  
 }  
 }  
  
 boolean workerStarted = false;  
 boolean workerAdded = false;  
 Worker w = null;  
 try {  
 //创建新的工作线程  
 w = new Worker(firstTask);  
 final Thread t = w.thread;  
 if (t != null) {  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
 // Recheck while holding lock.  
 // Back out on ThreadFactory failure or if  
 // shut down before lock acquired.  
 int rs = *runStateOf*(ctl.get());  
  
 if (rs < *SHUTDOWN* ||  
 (rs == *SHUTDOWN* && firstTask == null)) {  
 if (t.isAlive()) // precheck that t is startable  
 throw new IllegalThreadStateException();  
 workers.add(w);  
 int s = workers.size();  
 if (s > largestPoolSize)  
 largestPoolSize = s;  
 workerAdded = true;  
 }  
 } finally {  
 mainLock.unlock();  
 }  
 if (workerAdded) {//添加成功，启动线程  
 t.start();  
 workerStarted = true;  
 }  
 }  
 } finally {  
 if (! workerStarted)  
 addWorkerFailed(w);//添加失败，回滚  
 }  
 return workerStarted;  
}

**说明：**检查当前池状态和给定界限中是否可以添加新worker，如果可以，需要对workercount做出相应调整。添加完毕后，运行firstTask。如果线程池已停止或正在关闭或thread factory创建线程失败返回false。  
  
如果由于thread factory返回null或创建线程过程中抛出异常导致线程创建失败，则调用addWorkerFailed回滚添加worker操作。addWorkerFailed源码如下：

addWorkerFailed(Worker w)

//回滚添加worker线程失败操作  
private void addWorkerFailed(Worker w) {  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
 if (w != null)  
 workers.remove(w);  
 decrementWorkerCount();//workerCount-1  
 tryTerminate();//尝试终止线程池  
 } finally {  
 mainLock.unlock();  
 }  
}

**说明：**回滚操作需要进行三步：  
 1.从worker集合中移除worker  
 2.workerCount-1  
 3.调用tryTerminate尝试终止线程

* + - * 1. execute(Runnable command)

1. */\*\*提交任务\*/*public void execute(Runnable command) {  
    if (command == null)  
    throw new NullPointerException();  
    /\*  
    \* Proceed in 3 steps:  
    \*  
    \* 1. If fewer than corePoolSize threads are running, try to  
    \* start a new thread with the given command as its first  
    \* task. The call to addWorker atomically checks runState and  
    \* workerCount, and so prevents false alarms that would add  
    \* threads when it shouldn't, by returning false.  
    \*  
    \* 2. If a task can be successfully queued, then we still need  
    \* to double-check whether we should have added a thread  
    \* (because existing ones died since last checking) or that  
    \* the pool shut down since entry into this method. So we  
    \* recheck state and if necessary roll back the enqueuing if  
    \* stopped, or start a new thread if there are none.  
    \*  
    \* 3. If we cannot queue task, then we try to add a new  
    \* thread. If it fails, we know we are shut down or saturated  
    \* and so reject the task.  
    \*/  
    int c = ctl.get();  
    if (*workerCountOf*(c) < corePoolSize) {  
    if (addWorker(command, true))//添加到工作线程  
    return;  
    c = ctl.get();  
    }  
    if (*isRunning*(c) && workQueue.offer(command)) {//池正在运行，添加到等待队列  
    int recheck = ctl.get();  
    if (! *isRunning*(recheck) && remove(command))//池状态>=SHUTDOWN，移除任务，执行拒绝策略  
    reject(command);  
    else if (*workerCountOf*(recheck) == 0)//工作线程为空，添加新的工作线程  
    addWorker(null, false);  
    }  
    else if (!addWorker(command, false))  
    reject(command);  
   }

**说明：**提交一个任务到线程池，可能不会立即执行。提交的任务可能在一个新的线程中执行，也可能在已经存在的空闲线程执行。如果由于池已经关闭或者池容量已满导致任务无法提交，那么就根据RejectedExecutionHandler处理提交过来的任务。分三种情况：

1. 如果正在运行线程少于corePoolSize，尝试开启一个新的线程并把提交的任务作为它的firstTask运行。调用addWorker原子性检查runState和workerCount，通过返回false来防止在不应该添加线程时添加线程的假警报
2. 如果任务可被添加到等待队列，我们依然需要检查是否应该添加一个线程（因为在上次检查之后存在一个死掉的线程），还是进入到这个方法之后池被关闭。所以需要再次检查池状态，以保证能在操作失败时回滚，或者在没有线程的时候创建一个新的线程。
3. 如果任务不能添加到等待队列，则尝试添加新的worker，失败执行拒绝策略。

Iterator spliterator