## 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和Unsafe

JUC包大量使用了CAS，涉及并发或资源争用的地方都使用了sun.misc.Unsafe类的方法进行CAS操作。所以，在讲解JUC包之前，我们首先了解一下CAS和Unsafe类。

#### CAS介绍

**CAS,compare and swap比较并替换。** **CAS有三个操作数：内存值（V）、预期原值（A）和新值(B)。如果内存值与预期原值相匹配，那么处理器会自动将该位置值更新为新值并返回true。否则，不做任何操作返回false。**

**以AtomicInteger为例，内部的CAS实现如下：**

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*();  
 //value的偏移地址  
 private static final long *valueOffset*;  
  
 static {  
 try {  
 *valueOffset* = *unsafe*.objectFieldOffset  
 (AtomicInteger.class.getDeclaredField("value"));  
 } catch (Exception ex) {  
 throw new Error(ex);  
 }  
 }  
  
 private volatile int value;  
  
 public AtomicInteger(int initialValue) {  
 value = initialValue;  
 }  
   
 public final int getAndUpdate(IntUnaryOperator updateFunction) {  
 int prev, next;  
 do {  
 prev = get();  
 next = updateFunction.applyAsInt(prev);  
 } while (!compareAndSet(prev, next));  
 return prev;  
 }  
 public final boolean compareAndSet(int expect, int update) {  
 return *unsafe*.compareAndSwapInt(this, *valueOffset*, expect, update);  
 }  
  
}

说明： 可以看到AtomicInteger内部都是使用了Unsafe类来进行CAS操作，valueOffset表示的是value值的偏移地址，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使用volatile修饰，直接从共享内存中操作变量，保证多线程之间看到的value值是同一份。

以方法getAndUpdate()为例，执行步骤如下：  
1. 从内存中读取修改前的值prev，并执行给定lambda计算修改后的值next；  
2. 调用compareAndSet修改value值（内部是调用了unsafe的compareAndSwapInt方法）。如果此时有其他线程也在修改这个value值，那么CAS操作就会失败，继续进入do循环重新获取新值，再次执行CAS直到修改成功。

**ABA问题**

ABA问题是一种异常现象：如果算法中的节点可以被循环利用，那么在使用“比较并替换”指令时就可能出现这种问题（主要在没有垃圾回收机制的环境中）。

如果有两个线程x和y，如果x初次从内存中读取变量值为A；线程y对它进行了一些操作使其变成B，然后再改回A，那么线程x进行CAS的时候就会误认为这个值没有被修改过。尽管CAS操作会成功执行，但是不代表它是没有问题的，如果有一个单向链表A B组成的栈，栈顶为A，线程T1准备执行CAS操作head.compareAndSet(A,B)，在执行之前线程T2介入，T2将A、B出栈，然后又把C、A放入栈，T2执行完毕；切回线程T1，T1发现栈顶元素依然为A，也会成功执行CAS将栈顶元素修改为B，但因为B.next为null，所以栈结构就会丢弃C元素。

针对这种情况，有一种简单的解决方案：不是更新某个引用的值，而是更新两个值，包括一个引用和一个和版本号，即这个值由A变为B，然后又变成A，版本号也将是不同的。Java中提供了AtomicStampedReference和AtomicMarkableReference来解决ABA问题。他们支持在两个变量上执行原子的条件更新。AtomicStampedReference将更新一个“对象-引用”二元组，通过在引用上加上“版本号”，从而避免ABA问题。 类似地，AtomicMarkableReference将更新一个“对象引用-布尔值”二元组，在某些算法中将通过这种二元组使节点保存在链表中同时又将其标记为“已删除的节点”。不过目前来说，这两个类比较鸡肋，大部分情况下的ABA问题不会影响程序并发的正确性，如果需要解决ABA问题，改用传统的互斥同步可能会比原子类更高效。

#### Unsafe

Unsafe是实现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。Unsafe类提供了硬件级别的原子操作。

##### Unsafe函数列表

///--------------------- peek and poke operations--------------

//获取对象o中给定偏移地址(offset)的值。以下相关get方法作用相同  
public native int getInt(Object o, long offset);

//在对象o的给定偏移地址存储数值x。以下set方法作用相同  
public native void putInt(Object o, long offset, int x);

public native Object getObject(Object o, long offset);

public native void putObject(Object o, long offset, Object x);

*/\*\*篇幅原因，省略其他类型方法 \*/*//从给定内存地址获取一个byte。下同  
public native byte getByte(long address);

//在给定内存地址放置一个x。下同  
public native void putByte(long address, byte x);

*/\*\*篇幅原因，省略其他类型方法\*/*

//获取给定内存地址的一个本地指针  
public native long getAddress(long address);

//在给定的内存地址处存放一个本地指针x  
public native void putAddress(long address, long x);

///------------------内存操作----------------------

//在本地内存分配一块指定大小的新内存，内存的内容未初始化;它们通常被当做垃圾回收。  
//返回的本地指针永远不会为零，并将对所有值类型进行对齐。  
// 使用freeMemory来释放这块内存，或者使用reallocateMemory来重新分配大小  
public native long allocateMemory(long bytes);

//重新分配给定内存地址的本地内存  
public native long reallocateMemory(long address, long bytes);

//将给定内存块中的所有字节设置为固定值（通常是0）。  
//该方法通过两个参数确定内存块的地址，也就是说提供了一个double-register地址模型(指定对象，指定offset)。  
//当给定对象o为null，那么给定offset就是内存的base地址。  
public native void setMemory(Object o, long offset, long bytes, byte value);

//复制一块内存，double-register模型  
public native void copyMemory(Object srcBase, long srcOffset,  
 Object destBase, long destOffset,  
 long bytes);

//复制一块内存，single-register模型  
public void copyMemory(long srcAddress, long destAddress, long bytes) {  
 copyMemory(null, srcAddress, null, destAddress, bytes);  
}

//释放给定地址的内存  
public native void freeMemory(long address);

//获取给定对象的偏移地址  
public native long staticFieldOffset(Field f);

public native long objectFieldOffset(Field f);

//------------------数组操作---------------------------------

//获取给定数组的第一个元素的偏移地址  
public native int arrayBaseOffset(Class<?> arrayClass);

//获取给定数组的元素增量地址，也就是说每个元素的占位数  
public native int arrayIndexScale(Class<?> arrayClass);

//---------------------------------------------------

//告诉虚拟机去定义一个类。默认情况下，类加载器和保护域都来自这个方法  
public native Class<?> defineClass(String name, byte[] b, int off, int len,  
 ClassLoader loader,  
 ProtectionDomain protectionDomain);  
//定义匿名内部类  
public native Class<?> defineAnonymousClass(Class<?> hostClass, byte[] data, Object[] cpPatches);

//定位一个实例，但不运行构造函数  
public native Object allocateInstance(Class<?> cls) throws InstantiationException;

///--------------------锁指令（synchronized）-------------------------------

//对象加锁  
public native void monitorEnter(Object o);

//对象解锁  
public native void monitorExit(Object o);

public native boolean tryMonitorEnter(Object o);

//解除给定线程的阻塞  
public native void unpark(Object thread);

//阻塞当前线程  
public native void park(boolean isAbsolute, long time);

// CAS  
public final native boolean compareAndSwapObject(Object o, long offset,  
 Object expected,  
 Object x);

//获取对象o的给定偏移地址的引用值（volatile方式）  
public native Object getObjectVolatile(Object o, long offset);

public native void putObjectVolatile(Object o, long offset, Object x);

*/\*\* 省略其他类型方法 \*/*

//用于lazySet，适用于低延迟代码。实现非堵塞写入，避免指令重排序，这样它使用快速的存储-存储(store-store) barrier,  
// 而不是较慢的存储-加载(store-load) barrier, 后者总是用在volatile的写操作上，这种性能提升是有代价的，虽然便宜，  
// 也就是写后结果并不会被其他线程看到，甚至是自己的线程，通常是几纳秒后被其他线程看到，这个时间比较短，所以代价可以忍受。  
public native void putOrderedObject(Object o, long offset, Object x);

*/\*\* 省略其他类型方法 \*/*

//获取并加上给定delta，返回加之前的值  
public final int getAndAddInt(Object o, long offset, int delta)

*/\*\* 省略其他类型方法 \*/*

//为给定偏移地址设置一个新的值，返回设置之前的值  
public final int getAndSetInt(Object o, long offset, int newValue)

*/\*\* 省略其他类型方法 \*/*

///--------------------1.8新增指令-----------------------

// loadFence() 表示该方法之前的所有load操作在内存屏障之前完成  
public native void loadFence();

//表示该方法之前的所有store操作在内存屏障之前完成  
public native void storeFence();

//表示该方法之前的所有load、store操作在内存屏障之前完成，这个相当于上面两个的合体功能  
public native void fullFence();

Unsafe的方法比较简单，直接看方法字面意思就大概直到方法的作用了。

在Unsafe里有两个方法模型**：**double-register模型：给定对象，给定偏移地址offset。从给定对象的偏移地址取值。如getInt(Object o, long offset)；  
single-register模型：给定内存地址，直接从给定内存地址取值，如getInt(long)。

这里介绍一下几个比较重要的方法，在之后的源码阅读里会出现这几个方法。

1. **arrayBaseOffset：操作数组，用于获取数组的第一个元素的偏移地址**
2. **arrayIndexScale：操作数组，用于获取数组元素的增量地址，也就是说每个元素的占位数。打个栗子：如果有一个数组{1,2,3,4,5,6}，它第一个元素的偏移地址为16，每个元素的占位是4，如果我们要获取数组中“5”这个数字，那么它的偏移地址就是16+4\*4。**
3. **putOrderedObject：putOrderedObject是lazySet的实现，适用于低延迟代码。它能够实现非堵塞写入，避免指令重排序，这样它使用快速的存储-存储(store-store) barrier,而不是较慢的存储-加载(store-load) barrier, 后者总是用在volatile的写操作上。这种性能提升是有代价的，也就是写后结果并不会被其他线程看到，甚至是自己的线程，通常是几纳秒后被其他线程看到。类似的方法还有putOrderedInt、putOrderedLong。**
4. **loadFence、storeFence、fullFence：这三个方法是1.8新增，主要针对内存屏障定义，也是为了避免重排序。  
   loadFence() 表示该方法之前的所有load操作在内存屏障之前完成。  
   storeFence()表示该方法之前的所有store操作在内存屏障之前完成。  
   fullFence()表示该方法之前的所有load、store操作在内存屏障之前完成。**

### JUC锁

#### 框架



#### AQS ( AbstractQueuedSynchronizer )

##### 3.2.1 概述

AbstractQueuedSynchronizer，简称AQS。是java里的一个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等。支持独占锁（Exclusive）和共享锁（Share）两种模式：

独占锁：锁只能被一个线程获取到(ReentrantLock)；  
共享锁：锁可以被多个线程同时获取(CountDownLatch、ReadWriteLock的读锁)。

不管是独占锁还是共享锁，本质上都是对AQS内部的一个变量state的获取，state是一个原子性的int变量，可用来表示锁状态、资源数等



##### 3.2.2 数据结构和核心参数



AQS的内部实现了两个队列，一个同步队列和一个条件队列。其中条件队列是为Lock实现的一个基础同步器，并且一个线程可能会有多个条件队列，只有在使用了Condition才会存在条件队列。不管是同步队列还是条件队列，其内部都是由节点Node组成，首先介绍下AQS的内部类Node，源码如下：

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;  
}

说明：Node的实现很简单，就是一个普通双向链表的实现，这里主要说明一下内部的几个等待状态：

1. CANCELLED，值为1，当前节点由于超时或中断被取消。
2. SIGNAL，值为-1，表示当前节点的前节点被阻塞，当前节点在release或cancel时需要执行unpark来唤醒后继节点。
3. CONDITION，值为-2，当前节点正在等待condition，这个状态在同步队列里不会被用到。
4. PROPAGATE，值为-3，(针对共享锁) releaseShared()操作需要被传递到其他节点，这个状态在doReleaseShared中被设置，用来保证后续节点可以获取共享资源。
5. 0，初始状态，当前节点在sync queue中，等待获取锁。

AQS已经为我们提供了同步器的基础操作，如果要自定义同步器，必须实现以下几个方法：

* 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。
* isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。

##### 3.2.2源码解析

###### 3.2.2.1 acquire(int)

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

**说明：**独占模式下获取资源，忽略中断的影响。内部主要调用了三个方法，其中tryAcquire需要自定义实现。后面会对内部方法详细说明。函数流程如下：

1. tryAcquire() 尝试直接获取资源，如果成功则直接返回，失败进入第二步；
2. addWaiter() 获取资源失败后，将当前线程加入等待队列的尾部，并标记为独占模式；
3. acquireQueued() 使线程在等待队列中自旋等待获取资源，一直获取到资源后才返回。如果在等待过程中被中断过，则返回true，否则返回false。
4. 如果线程在等待过程中被中断(interrupt)是不响应的，在获取资源成功之后根据返回的中断状态调用selfInterrupt方法再把中断状态补上。

tryAcquire(int)

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；

//添加等待节点到尾部  
private Node addWaiter(Node mode) {  
 Node node = new Node(Thread.*currentThread*(), mode);  
 // Try the fast path of enq; backup to full enq on failure  
 //尝试快速入队  
 Node pred = tail;  
 if (pred != null) {  
 node.prev = pred;  
 if (compareAndSetTail(pred, node)) {  
 pred.next = node;  
 return node;  
 }  
 }  
 enq(node);  
 return node;  
}

//插入给定节点到队尾  
private Node enq(final Node node) {  
 for (;;) {  
 Node t = tail;  
 if (t == null) { // Must initialize  
 if (compareAndSetHead(new Node()))  
 tail = head;  
 } else {  
 node.prev = t;  
 if (compareAndSetTail(t, node)) {  
 t.next = node;  
 return t;  
 }  
 }  
 }  
}

acquireQueued()

线程进入等待队列后，在等待队列中自旋等待获取资源。如果在整个等待过程中被中断过，则返回true，否则返回false。具体流程如下：

1. 获取当前等待节点的前继节点，如果前继节点为head，说明可以尝试获取锁；

2. 调用`tryAcquire`获取锁，成功后更新`head`为当前节点；

3. 获取资源失败，调用`shouldParkAfterFailedAcquire`方法检查并更新等待状态。如果前继节点状态为`SIGNAL`，说明当前节点可以进入waiting状态等待唤醒；被唤醒后，继续自旋重复上述步骤。源码:

//自旋等待获取资源  
final boolean acquireQueued(final Node node, int arg) {  
 boolean failed = true;  
 try {  
 boolean interrupted = false;  
 for (;;) {  
 final Node p = node.predecessor();//获取前继节点  
 //前继节点为head，说明可以尝试获取资源  
 if (p == head && tryAcquire(arg)) {  
 setHead(node);//获取成功，更新head节点  
 p.next = null; // help GC  
 failed = false;  
 return interrupted;  
 }  
 if (*shouldParkAfterFailedAcquire*(p, node) && //检查是否可以park  
 parkAndCheckInterrupt())  
 interrupted = true;  
 }  
 } finally {  
 if (failed)  
 cancelAcquire(node);  
 }  
}  
  
//获取资源失败后，检查并更新等待状态  
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {  
 int ws = pred.waitStatus;  
 if (ws == Node.*SIGNAL*)  
 /\*  
 \* This node has already set status asking a release  
 \* to signal it, so it can safely park.  
 \*/  
 return true;  
 if (ws > 0) {  
 /\*  
 \* Predecessor was cancelled. Skip over predecessors and  
 \* indicate retry.  
 \*/  
 //如果前节点取消了，那就一直往前找到一个等待状态的节点，并排在它的后边  
 do {  
 node.prev = pred = pred.prev;  
 } while (pred.waitStatus > 0);  
 pred.next = node;  
 } else {  
 /\*  
 \* waitStatus must be 0 or PROPAGATE. Indicate that we  
 \* need a signal, but don't park yet. Caller will need to  
 \* retry to make sure it cannot acquire before parking.  
 \*/  
 //此时前节点状态为0或PROPAGATE，表示我们需要一个唤醒信号，但是不立即park,在park前调用者需要重试来确认它不能获取资源。  
 *compareAndSetWaitStatus*(pred, ws, Node.*SIGNAL*);  
 }  
 return false;  
}

//阻塞当前线程，返回中断状态  
private final boolean parkAndCheckInterrupt() {  
 LockSupport.*park*(this);  
 return Thread.*interrupted*();  
}

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)

*/\*\*独占锁释放资源\*/*public final boolean release(int arg) {  
 if (tryRelease(arg)) {//尝试释放资源  
 Node h = head;//头结点  
 if (h != null && h.waitStatus != 0)  
 unparkSuccessor(h);//唤醒head的下一个节点  
 return true;  
 }  
 return false;  
}

**说明：**独占模式下释放指定量的资源，成功释放后调用unparkSuccessor唤醒head的下一个节点。

tryRelease(int)

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

说明：和tryAcquire()一样，这个方法也需要自定义同步器去实现。一般来说，释放资源直接拿state减去给定的参数arg，释放后state==0说明释放成功。在ReentrantLock中实现如下：

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);//设置独占锁持有线程为null  
 }  
 setState(c);  
 return free;  
}

unparkSuccessor(Node)

说明: 唤醒等待队列里的下一个线程

private void unparkSuccessor(Node node) {  
 int ws = node.waitStatus;  
 if (ws < 0)//当前节点没有被取消,更新waitStatus为0。  
 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);  
}

**说明：**成功获取到资源后，调用此方法唤醒head的下一个节点。因为当前节点已经释放掉资源，下一个等待的线程可以被唤醒继续获取资源。

###### 3.2.2.3 acquireShared(int)

*/\*\*共享模式获取资源，忽略中断\*/*public final void acquireShared(int arg) {  
 if (tryAcquireShared(arg) < 0)  
 doAcquireShared(arg);  
}

**说明：**共享模式下获取资源/锁，忽略中断的影响。内部主要调用了两个个方法，其中tryAcquireShared需要自定义同步器实现。后面会对各个方法进行详细分析。acquireShared方法流程如下：

1. tryAcquireShared(arg) 尝试获取共享资源。成功获取并且还有可用资源返回正数；成功获取但是没有可用资源时返回0；获取资源失败返回一个负数。
2. 获取资源失败后调用doAcquireShared方法进入等待队列，获取资源后返回。

tryAcquiredShared(int)

*/\*\*共享模式获取资源\*/*protected int tryAcquireShared(int arg) {  
 throw new UnsupportedOperationException();  
}

**说明：**尝试获取共享资源，需同步器自定义实现。有三个类型的返回值：

* + 1. 正数：成功获取资源，并且还有剩余可用资源，可以唤醒下一个等待线程；
    2. 负数：获取资源失败，准备进入等待队列；
    3. 0：获取资源成功，但没有剩余可用资源。

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)

//获取共享锁  
private void doAcquireShared(int arg) {  
 final Node node = addWaiter(Node.*SHARED*);//添加一个共享模式Node到队列尾  
 boolean failed = true;  
 try {  
 boolean interrupted = false;  
 for (;;) {  
 final Node p = node.predecessor();//获取前节点  
 if (p == head) {  
 int r = tryAcquireShared(arg);//前节点为head，尝试获取资源  
 if (r >= 0) {  
 //获取资源成功，设置head为自己，如果有剩余资源可以在唤醒之后的线程  
 setHeadAndPropagate(node, r);  
 p.next = null; // help GC  
 if (interrupted)  
 *selfInterrupt*();  
 failed = false;  
 return;  
 }  
 }  
 if (*shouldParkAfterFailedAcquire*(p, node) && //检查获取失败后是否可以阻塞  
 parkAndCheckInterrupt())  
 interrupted = true;  
 }  
 } finally {  
 if (failed)  
 cancelAcquire(node);  
 }  
}

**说明：**在tryAcquireShared中获取资源失败后，将当前线程加入等待队列尾部等待唤醒，成功获取资源后返回。

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

//设置head，如果有剩余资源可以再唤醒之后的线程  
private void setHeadAndPropagate(Node node, int propagate) {  
 Node h = head; // Record old head for check below  
 setHead(node);  
 /\*  
 \* 如果满足下列条件可以尝试唤醒下一个节点：  
 \* 调用者指定参数(propagate>0)，并且后继节点正在等待或后继节点为空  
 \*/  
 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)

*/\*\*共享模式释放资源\*/*public final boolean releaseShared(int arg) {  
 if (tryReleaseShared(arg)) {  
 doReleaseShared();//释放锁，并唤醒后继节点  
 return true;  
 }  
 return false;  
}

**说明：**共享模式下释放给定量的资源，如果成功释放，唤醒等待队列的后继节点。tryReleaseShared需要自定义同步器去实现。方法执行流程：tryReleaseShared(int)尝试释放给定量的资源，成功释放后调用doReleaseShared()唤醒后继线程。

tryReleaseShared(int)

*/\*\*共享模式释放资源\*/*protected boolean tryReleaseShared(int arg) {  
 throw new UnsupportedOperationException();  
}

**说明：**释放给定量的资源，需自定义同步器实现。释放后如果允许后继等待线程获取资源返回true。

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)

//释放共享资源-唤醒后继线程并保证后继节点的传播  
private void doReleaseShared() {  
 //自旋，确保释放后唤醒后继节点  
 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);//唤醒后继节点  
 }  
 else if (ws == 0 &&  
 !*compareAndSetWaitStatus*(h, 0, Node.*PROPAGATE*))  
 continue; // loop on failed CAS  
 }  
 if (h == head) // loop if head changed  
 break;  
 }  
}

**说明：**在tryReleaseShared成功释放资源后，调用此方法唤醒后继线程并保证后继节点的release传播（通过设置head节点的waitStatus为PROPAGATE）。

###### 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**，也被称为**“独占锁”**。在上一篇讲解AQS的时候已经提到，“独占锁”在同一个时间点只能被一个线程持有；而可重入的意思是，ReentrantLock可以被单个线程多次获取。  
ReentrantLock又分为**“公平锁(fair lock)”和“非公平锁(non-fair lock)”**。它们的区别体现在获取锁的机制上是否公平。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。

##### 数据结构和核心参数



可以看到ReetrantLock继承自AQS，并实现了Lock接口。Lock源码如下：

public interface Lock {  
 //获取锁，如果锁不可用则线程一直等待  
 void lock();  
 //获取锁，响应中断，如果锁不可用则线程一直等待  
 void lockInterruptibly() throws InterruptedException;  
 //获取锁，获取失败直接返回  
 boolean tryLock();  
 //获取锁，等待给定时间后如果获取失败直接返回  
 boolean tryLock(long time, TimeUnit unit) throws InterruptedException;  
 //释放锁  
 void unlock();  
 //创建一个新的等待条件  
 Condition newCondition();  
}

在Lock提供的获取锁方法中，有lock()、lockInterruptibly()、tryLock()和tryLock(long time, TimeUnit unit)四种方式，他们的区别如下：

* lock() 获取失败后，线程进入等待队列自旋或休眠，直到锁可用，并且忽略中断的影响
* lockInterruptibly() 线程进入等待队列park后，如果线程被中断，则直接响应中断（抛出InterruptedException）
* tryLock() 获取锁失败后直接返回，不进入等待队列
* tryLock(long time, TimeUnit unit) 获取锁失败等待给定的时间后返回获取结果

ReetrantLock通过AQS实现了自己的同步器Sync，分为公平锁FairSync和非公平锁NonfairSync。在构造时，通过所传参数boolean fair来确定使用那种类型的锁。

本篇会以对比的方式分析两种锁的源码实现方式。

ReetrantLock通过AQS实现了自己的同步器Sync，分为公平锁FairSync和非公平锁NonfairSync。本篇会以对比的方式分析两种锁的源码实现方式。

##### lock()

//获取锁，一直等待锁可用  
public void lock() {  
 sync.lock();  
}

//公平锁获取  
final void lock() {  
 acquire(1);  
}

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

**说明：**公平锁的lock方法调用了AQS的acquire(1)；而非公平锁则直接通过CAS修改state值来获取锁，当获取失败时才会调用acquire(1)来获取锁。

关于acquire()方法，在上篇介绍AQS的时候已经讲过，印象不深的同学可以翻回去看一下，这里主要来看一下tryAcquire在ReetrantLock中的实现。

//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);  
}

AQS的acquire()源码:

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

公平锁tryAcquire():

//公平锁  
protected final boolean tryAcquire(int acquires) {  
 final Thread current = Thread.*currentThread*();  
 int c = getState();//获取锁状态state  
 if (c == 0) {  
 if (!hasQueuedPredecessors() && //判断当前线程是否还有前节点  
 compareAndSetState(0, acquires)) {//CAS修改state  
 //获取锁成功，设置锁的持有线程为当前线程  
 setExclusiveOwnerThread(current);  
 return true;  
 }  
 }  
 else if (current == getExclusiveOwnerThread()) {//当前线程已经持有锁  
 int nextc = c + acquires;//重入  
 if (nextc < 0)  
 throw new Error("Maximum lock count exceeded");  
 setState(nextc);//更新state状态  
 return true;  
 }  
 return false;  
}

**说明：**公平锁模式下的tryAcquire，执行流程如下：

如果当前锁状态state为0，说明锁处于闲置状态可以被获取，首先调用hasQueuedPredecessors方法判断当前线程是否还有前节点(prev node)在等待获取锁。如果有，则直接返回false；如果没有，通过调用compareAndSetState（CAS）修改state值来标记自己已经拿到锁，CAS执行成功后调用setExclusiveOwnerThread设置锁的持有者为当前线程。程序执行到现在说明锁获取成功，返回true；

如果当前锁状态state不为0，但当前线程已经持有锁（current == getExclusiveOwnerThread()），由于锁是可重入（多次获取）的，则更新重入后的锁状态state += acquires 。锁获取成功返回true。

//非公平锁获取  
protected final boolean tryAcquire(int acquires) {  
 return nonfairTryAcquire(acquires);  
}

final boolean nonfairTryAcquire(int acquires) {  
 final Thread current = Thread.*currentThread*();  
 int c = getState();  
 if (c == 0) {  
 if (compareAndSetState(0, acquires)) {//CAS修改state  
 setExclusiveOwnerThread(current);  
 return true;  
 }  
 }  
 else if (current == getExclusiveOwnerThread()) {  
 int nextc = c + acquires;//计算重入后的state  
 if (nextc < 0) // overflow  
 throw new Error("Maximum lock count exceeded");  
 setState(nextc);  
 return true;  
 }  
 return false;  
}

**说明：**通过对比公平锁和非公平锁tryAcquire的代码可以看到，非公平锁的获取略去了!hasQueuedPredecessors()这一操作，也就是说它不会判断当前线程是否还有前节点(prev node)在等待获取锁，而是直接去进行锁获取操作。

##### unlock()

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

protected final boolean tryRelease(int releases) {  
 int c = getState() - releases;//计算释放后的state值  
 if (Thread.*currentThread*() != getExclusiveOwnerThread())  
 throw new IllegalMonitorStateException();  
 boolean free = false;  
 if (c == 0) {  
 free = true;//锁全部释放，可以唤醒下一个等待线程  
 setExclusiveOwnerThread(null);//设置锁持有线程为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

// 使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()  
// 使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)  
// 使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)  
// 使当前线程在接到信号之前一直处于等待状态。  
void awaitUninterruptibly()  
// 使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()

#### 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由ReentrantReadWriteLock实现。

源码:

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

ReentrantReadWriteLock维护了一对相关的锁：共享锁readLock和独占锁writeLock。共享锁readLock用于读操作，能同时被多个线程获取；独占锁writeLock用于写入操作，只能被一个线程持有。独占锁的实现和我们上篇所讨论的ReentrantLock相似，共享锁我们接下来会详细分析。同样的，ReentrantReadWriteLock也支持公平和非公平两种模式：

**非公平模式：**当构造一个非公平锁时，读锁和写锁的顺序是未指定的，受重入的限制。一个持续争用的非公平锁，可能会使其他读线程或写线程无限延期，但它比公平锁有更高的吞吐量。

**公平模式：**当构造一个公平锁时，线程争用使用一个近似地顺序到达的策略。当目前持有的锁被释放，要么是等待时间最长的单个写入线程被分配写入锁，或者如果有一组读线程比所有等待写线程等待更长的时间，该组将被分配读取锁。  
 如果写锁被持有，或者有一个正在等待写的线程，一个尝试获取公平读锁的线程（非重入）将会阻塞。在等待时间最长的写线程获取并释放写锁之前，当前线程将不能获取读锁。如果一个等待写入的线程放弃等待，并且在队列中等待时间最长的一个或多个读线程正在等待写锁空闲，那么这些读线程将被分配读取锁。  
 当读锁和写锁都是空闲时（这意味着已经没有等待线程）， 一个尝试获取公平写锁（非重入）的线程才会获取成功。注意非阻塞方法tryLock()会立即尝试获取锁，它们并不会按照公平原则那样去等待前继节点。

除了公平性，ReentrantReadWriteLock还有以下几种特性：

* 1. **重入性：**ReentrantReadWriteLock允许读线程和写线程重复获取 一种ReentrantLock方式的读锁或写锁。当所有写锁都被释放，不可重入读线程才允许获取锁。  
     此外，一个写入线程可以获取读锁，但是一个读线程不能获取写锁。重入性的作用还体现在：当已经持有写锁，在方法调用或回调期间在读取锁中执行读操作如果一个读线程尝试获取写锁将不会成功。
  2. **锁降级：**重入性允许从写锁降级到读锁，首先获取写锁，然后获取读锁，然后释放写锁。不过，从一个读锁升级到写锁是不允许的。读锁和写锁在获取过程中都支持中断。
  3. **Condition支持：**Condition只有在写锁中用到，读锁是不支持Condition的。

###### 数据结构和核心参数



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，内部的自定义同步器也继承自AQS。Sync也包括"公平锁"FairSync和"非公平锁"NonfairSync。

Sync

abstract static class Sync extends AbstractQueuedSynchronizer {  
 private static final long *serialVersionUID* = 6317671515068378041L;  
 // 最多支持65535(1<<16 -1)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;  
 // 读锁高16位，读锁个数加1，其实是状态值加 2^16  
 static final int *SHARED\_UNIT* = (1 << *SHARED\_SHIFT*);  
 // 锁最大数量  
 static final int *MAX\_COUNT* = (1 << *SHARED\_SHIFT*) - 1;  
 // 写锁掩码，用于标记低16位  
 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*; }  
  
 //读锁线程数  
 static int sharedCount(int c) { return c >>> *SHARED\_SHIFT*; }  
 //写锁线程数  
 static int exclusiveCount(int c) { return c & *EXCLUSIVE\_MASK*; }  
 //当前线程持有的读锁重入数量private transient ThreadLocalHoldCounter readHolds;  
 //最近一个获取读锁成功的线程计数器private transient HoldCounter cachedHoldCounter;  
 // 第一个获取读锁的线程  
 private transient Thread firstReader = null;  
 //firstReader的持有数  
 private transient int firstReaderHoldCount;  
  
}  
// 构造函数  
Sync() {  
 readHolds = new ThreadLocalHoldCounter();  
 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()  
// 获取读锁。  
ReentrantReadWriteLock.ReadLock readLock()  
// 获取写锁。  
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;  
 int r = *sharedCount*(c);//获取读锁数量  
 if (!readerShouldBlock() &&  
 r < *MAX\_COUNT* &&  
 compareAndSetState(c, c + *SHARED\_UNIT*)) {  
 if (r == 0) {//首次获取读锁,初始化firstReader和firstReaderHoldCount  
 firstReader = current;  
 firstReaderHoldCount = 1;  
 } else if (firstReader == current) {//当前线程是首个获取读锁的线程  
 firstReaderHoldCount++;  
 } else {  
 //更新cachedHoldCounter  
 HoldCounter rh = cachedHoldCounter;  
 if (rh == null || rh.tid != *getThreadId*(current))  
 cachedHoldCounter = rh = readHolds.get();  
 else if (rh.count == 0)  
 readHolds.set(rh);  
 rh.count++;//更新获取的读锁数量  
 }  
 return 1;  
 }  
 return fullTryAcquireShared(current);  
}

**说明：**tryAcquireShared()的作用是尝试获取“读锁/共享锁”。函数流程如下：

1. 如果“写锁”已经被持有，这时候可以继续获取读锁，但如果持有写锁的线程不是当前线程，直接返回-1（表示获取失败）；
2. 如果在尝试获取锁时不需要阻塞等待（由公平性决定），并且读锁的共享计数小于最大数量MAX\_COUNT，则直接通过CAS函数更新读取锁的共享计数，最后将当前线程获取读锁的次数+1。
3. 如果第二步执行失败，则调用fullTryAcquireShared尝试获取读锁，源码如下：

fullTryAcquireShared(Thread current)

//获取读锁的完整版本，用于处理CAS失败、阻塞等待和重入读问题  
final int fullTryAcquireShared(Thread current) {  
 /\*  
 \* This code is in part redundant with that in  
 \* tryAcquireShared but is simpler overall by not  
 \* complicating tryAcquireShared with interactions between  
 \* retries and lazily reading hold counts.  
 \*/  
 HoldCounter rh = null;  
 for (;;) {//自旋  
 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) {  
 if (getExclusiveOwnerThread() != current)  
 return -1;  
 // else we hold the exclusive lock; blocking here  
 // would cause deadlock.  
 } else if (readerShouldBlock()) {//需要阻塞  
 // Make sure we're not acquiring read lock reentrantly  
 //当前线程如果是首个获取读锁的线程，则继续往下执行。  
 if (firstReader == current) {  
 // assert firstReaderHoldCount > 0;  
 } else {  
 //更新锁计数器  
 if (rh == null) {  
 rh = cachedHoldCounter;  
 if (rh == null || rh.tid != *getThreadId*(current)) {  
 rh = readHolds.get();  
 if (rh.count == 0)  
 readHolds.remove();//当前线程持有读锁数为0，移除计数器  
 }  
 }  
 if (rh.count == 0)  
 return -1;  
 }  
 }  
 if (*sharedCount*(c) == *MAX\_COUNT*)//超出最大读锁数量  
 throw new Error("Maximum lock count exceeded");  
 if (compareAndSetState(c, c + *SHARED\_UNIT*)) {//CAS更新读锁数量  
 if (*sharedCount*(c) == 0) {//首次获取读锁  
 firstReader = current;  
 firstReaderHoldCount = 1;  
 } else if (firstReader == current) {//当前线程是首个获取读锁的线程，更新持有数  
 firstReaderHoldCount++;  
 } else {  
 //更新锁计数器  
 if (rh == null)  
 rh = cachedHoldCounter;  
 if (rh == null || rh.tid != *getThreadId*(current))  
 rh = readHolds.get();//更新为当前线程的计数器  
 else if (rh.count == 0)  
 readHolds.set(rh);  
 rh.count++;  
 cachedHoldCounter = rh; // cache for release  
 }  
 return 1;  
 }  
 }  
}

**说明**：fullTryAcquireShared是获取读锁的完整版本，用于处理CAS失败、阻塞等待和重入读问题。相对于tryAcquireShared来说，执行流程上都差不多，不同的是，它增加了重试机制和对“持有读锁数的延迟读取”的处理。

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) {  
 Thread current = Thread.*currentThread*();  
 if (firstReader == current) {//当前为第一个获取读锁的线程  
 // assert firstReaderHoldCount > 0;  
 //更新线程持有数  
 if (firstReaderHoldCount == 1)  
 firstReader = null;  
 else  
 firstReaderHoldCount--;  
 } else {  
 HoldCounter rh = cachedHoldCounter;  
 if (rh == null || rh.tid != *getThreadId*(current))  
 rh = readHolds.get();//获取当前线程的计数器  
 int count = rh.count;  
 if (count <= 1) {  
 readHolds.remove();  
 if (count <= 0)  
 throw unmatchedUnlockException();  
 }  
 --rh.count;  
 }  
 for (;;) {//自旋  
 int c = getState();  
 int nextc = c - *SHARED\_UNIT*;//获取剩余资源/锁  
 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;  
 }  
}

**说明：在releaseShared中，首先调用tryReleaseShared尝试释放锁，方法流程很简单，主要包括两步：**

**更新当前线程计数器的锁计数；**

**CAS更新释放锁之后的state，这里使用了自旋，在state争用的时候保证了CAS的成功执行。**

doReleaseShared()

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

#### 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()分别调用了Unsafe类的park() 、 unpark()，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。

##### 函数列表

// 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:**由于所有方法都是调用UNSAFE类的native接口实现，这里省略源码解析部分，感兴趣的同学可以自己查看相关源码。

#### CountDownLatch

##### 概述

CountDownLatch是一一个同步辅助类，在其他线程完成它们的操作之前，允许一个多个线程等待。简单来说，CountDownLatch中有一个计数，在计数到达0之前，线程会一直等待。

**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

//构造函数  
public CountDownLatch(int count)

//使当前线程等待直到锁计数变为0或者线程被中断  
public void await() throws InterruptedException

//使当前线程在锁计数变为0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 boolean await(long timeout, TimeUnit unit)

//递减锁计数，如果锁计数变为0，释放所有等待线程  
public void countDown()

//返回当前计数  
public 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))

protected boolean tryReleaseShared(int releases) {  
 // Decrement count; signal when transition to zero  
 for (;;) {  
 int c = getState();  
 if (c == 0)  
 return false;  
 int nextc = c-1;  
 if (compareAndSetState(c, nextc))  
 return nextc == 0;  
 }  
}

##### 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)。如果一个程序中有固定的线程数，并且线程之间需要相互等待，这时候CyclicBarrier是一个很好的选择。之所以叫它cyclic，是因为在释放等待线程之后，它可以被重用。



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  
//count=0 -> tripped  
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;  
 lock.lock();  
 try {  
 //当前generation  
 final Generation g = generation;  
  
 if (g.broken)  
 throw new BrokenBarrierException();  
  
 if (Thread.*interrupted*()) {  
 breakBarrier();//线程被中断，终止Barrier，唤醒所有等待线程  
 throw new InterruptedException();  
 }  
  
 int index = --count;  
 if (index == 0) { // tripped  
 boolean ranAction = false;  
 try {  
 final Runnable command = barrierCommand;  
 if (command != null)  
 command.run();//如果有barrierCommand，在所有parties到达之后运行它  
 ranAction = true;  
 //更新barrier状态并唤醒所有线程  
 nextGeneration();  
 return 0;  
 } finally {  
 if (!ranAction)  
 breakBarrier();  
 }  
 }  
  
 // loop until tripped, broken, interrupted, or timed out  
 //自旋等待 所有parties到达 | generation被销毁 | 线程中断 | 超时  
 for (;;) {  
 try {  
 if (!timed)  
 trip.await();  
 else if (nanos > 0L)  
 nanos = trip.awaitNanos(nanos);  
 } catch (InterruptedException ie) {  
 if (g == generation && ! g.broken) {  
 breakBarrier();  
 throw ie;  
 } else {  
 // We're about to finish waiting even if we had not  
 // been interrupted, so this interrupt is deemed to  
 // "belong" to subsequent execution.  
 Thread.*currentThread*().interrupt();  
 }  
 }  
  
 if (g.broken)  
 throw new BrokenBarrierException();  
  
 if (g != generation)  
 return index;  
  
 if (timed && nanos <= 0L) {  
 breakBarrier();//超时，销毁当前barrier  
 throw new TimeoutException();  
 }  
 }  
 } finally {  
 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的一个内部类，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

#### Phaser

Phaser是JDK1.7新增的一个同步辅助类，在功能上跟CyclicBarrier和CountDownLatch差不多，但支持更丰富的用法。本篇将从多个方面分析这个同步类。

##### 概述

* 1. **Registration（注册）：**  
     跟其他barrier不同，在phaser上注册的parties会随着时间的变化而变化。任务可以随时注册(使用方法register,bulkRegister注册，或者由构造器确定初始parties)，并且在任何抵达点可以随意地撤销注册(方法arriveAndDeregister)。就像大多数基本的同步结构一样，注册和撤销只影响内部count；不会创建更深的内部记录，所以任务不能查询他们是否已经注册。(不过，可以通过继承来实现类似的记录)
  2. **Synchronization(同步机制)：**  
     和CyclicBarrier一样，Phaser也可以重复await。方法arriveAndAwaitAdvance的效果类似CyclicBarrier.await。phaser的每一代都有一个相关的phase number，初始值为0，当所有注册的任务都到达phaser时phase+1，到达最大值(Integer.MAX\_VALUE)之后清零。使用phase number可以独立控制 到达phaser 和 等待其他线程 的动作，通过下面两种类型的方法：
     + Arrival(到达机制)  
       arrive和arriveAndDeregister方法记录到达状态。这些方法不会阻塞，但是会返回一个相关的arrival phase number；也就是说，phase number用来确定到达状态。当所有任务都到达给定phase时，可以执行一个可选的函数，这个函数通过重写onAdvance方法实现，通常可以用来控制终止状态。重写此方法类似于为CyclicBarrier提供一个barrierAction，但比它更灵活。
     + Waiting(等待机制)  
       awaitAdvance方法需要一个表示arrival phase number的参数，并且在phaser前进到与给定phase不同的phase时返回。和CyclicBarrier不同，即使等待线程已经被中断，awaitAdvance方法也会一直等待。中断状态和超时时间同样可用，但是当任务等待中断或超时后未改变phaser的状态时会遭遇异常。如果有必要，在方法forceTermination之后可以执行这些异常的相关的handler进行恢复操作，Phaser也可能被ForkJoinPool中的任务使用，这样在其他任务阻塞等待一个phase时可以保证足够的并行度来执行任务。
  3. **Termination(终止机制)：**
     + 可以用isTerminated方法检查phaser的终止状态。在终止时，所有同步方法立刻返回一个负值。在终止时尝试注册也没有效果。当调用onAdvance返回true时Termination被触发。当deregistration操作使已注册的parties变为0时，onAdvance的默认实现就会返回true。也可以重写onAdvance方法来定义终止动作。forceTermination方法也可以释放等待线程并且允许它们终止。
  4. **Tiering(分层结构)：**Phaser支持分层结构(树状构造)来减少竞争。注册了大量parties的Phaser可能会因为同步竞争消耗很高的成本， 因此可以设置一些子Phaser来共享一个通用的parent。这样的话即使每个操作消耗了更多的开销，但是会提高整体吞吐量。  
     在一个分层结构的phaser里，子节点phaser的注册和取消注册都通过父节点管理。子节点phaser通过构造或方法register、bulkRegister进行首次注册时，在其父节点上注册。子节点phaser通过调用arriveAndDeregister进行最后一次取消注册时，也在其父节点上取消注册。
  5. **Monitoring(状态监控)：**  
     由于同步方法可能只被已注册的parties调用，所以phaser的当前状态也可能被任何调用者监控。在任何时候，可以通过getRegisteredParties获取parties数，其中getArrivedParties方法返回已经到达当前phase的parties数。当剩余的parties(通过方法getUnarrivedParties获取)到达时，phase进入下一代。这些方法返回的值可能只表示短暂的状态，所以一般来说在同步结构里并没有啥卵用。

###### 核心参数

private volatile long state;

*/\*\*  
 \* The parent of this phaser, or null if none  
 \*/*private final Phaser parent;

*/\*\*  
 \* The root of phaser tree. Equals this if not in a tree.  
 \*/*private final Phaser root;

//等待线程的栈顶元素，根据phase取模定义为一个奇数header和一个偶数header  
private final AtomicReference<QNode> evenQ;  
private final AtomicReference<QNode> oddQ;

**state状态说明**：

Phaser使用一个long型state值来标识内部状态：  
> 低0-15位表示未到达parties数；  
> 中16-31位表示等待的parties数；  
> 中32-62位表示phase当前代；  
> 高63位表示当前phaser的终止状态。

注意：子Phaser的phase在没有被真正使用之前，允许滞后于它的root节点。这里在后面源码分析的reconcileState方法里会讲解。

**Qnode**是Phaser定义的内部等待队列，用于在阻塞时记录等待线程及相关信息。实现了ForkJoinPool的一个内部接口ManagedBlocker，上面已经说过，Phaser也可能被ForkJoinPool中的任务使用，这样在其他任务阻塞等待一个phase时可以保证足够的并行度来执行任务(通过内部实现方法isReleasable和block)。

###### 函数列表

//构造方法

public Phaser() {  
 this(null, 0);  
}

public Phaser(int parties) {  
 this(null, parties);  
}

public Phaser(Phaser parent) {  
 this(parent, 0);  
}

public Phaser(Phaser parent, int parties)

//注册一个新的party  
public int register()

//批量注册  
public int bulkRegister(int parties)

//使当前线程到达phaser，不等待其他任务到达。返回arrival phase number  
public int arrive()

//使当前线程到达phaser并撤销注册，返回arrival phase number  
public int arriveAndDeregister()

/\*  
 \* 使当前线程到达phaser并等待其他任务到达，等价于awaitAdvance(arrive())。  
 \* 如果需要等待中断或超时，可以使用awaitAdvance方法完成一个类似的构造。  
 \* 如果需要在到达后取消注册，可以使用awaitAdvance(arriveAndDeregister())。  
 \*/  
public int arriveAndAwaitAdvance()

//等待给定phase数，返回下一个 arrival phase number  
public int awaitAdvance(int phase)

//阻塞等待，直到phase前进到下一代，返回下一代的phase number  
public int awaitAdvance(int phase)

//响应中断版awaitAdvance  
public int awaitAdvanceInterruptibly(int phase) throws InterruptedException

public int awaitAdvanceInterruptibly(int phase, long timeout, TimeUnit unit)  
 throws InterruptedException, TimeoutException

//使当前phaser进入终止状态，已注册的parties不受影响，如果是分层结构，则终止所有phaser  
public void forceTermination()

##### 源码分析

###### register()

//注册一个新的party  
public int register() {  
 return doRegister(1);  
}

private int doRegister(int registrations) {  
 // adjustment to state  
 long adjust = ((long)registrations << *PARTIES\_SHIFT*) | registrations;  
 final Phaser parent = this.parent;  
 int phase;  
 for (;;) {  
 long s = (parent == null) ? state : reconcileState();  
 int counts = (int)s;  
 int parties = counts >>> *PARTIES\_SHIFT*;//获取已注册parties数  
 int unarrived = counts & *UNARRIVED\_MASK*;//未到达数  
 if (registrations > *MAX\_PARTIES* - parties)  
 throw new IllegalStateException(badRegister(s));  
 phase = (int)(s >>> *PHASE\_SHIFT*);//获取当前代  
 if (phase < 0)  
 break;  
 if (counts != *EMPTY*) { // not 1st registration  
 if (parent == null || reconcileState() == s) {  
 if (unarrived == 0) // wait out advance  
 root.internalAwaitAdvance(phase, null);//等待其他任务到达  
 else if (*UNSAFE*.compareAndSwapLong(this, *stateOffset*,  
 s, s + adjust))//更新注册的parties数  
 break;  
 }  
 }  
 else if (parent == null) { // 1st root registration  
 long next = ((long)phase << *PHASE\_SHIFT*) | adjust;  
 if (*UNSAFE*.compareAndSwapLong(this, *stateOffset*, s, next))//更新phase  
 break;  
 }  
 else {  
 //分层结构，子phaser首次注册用父节点管理  
 synchronized (this) { // 1st sub registration  
 if (state == s) { // recheck under lock  
 phase = parent.doRegister(1);//分层结构，使用父节点注册  
 if (phase < 0)  
 break;  
 // finish registration whenever parent registration  
 // succeeded, even when racing with termination,  
 // since these are part of the same "transaction".  
 //由于在同一个事务里，即使phaser已终止，也会完成注册  
 while (!*UNSAFE*.compareAndSwapLong  
 (this, *stateOffset*, s,  
 ((long)phase << *PHASE\_SHIFT*) | adjust)) {//更新phase  
 s = state;  
 phase = (int)(root.state >>> *PHASE\_SHIFT*);  
 // assert (int)s == EMPTY;  
 }  
 break;  
 }  
 }  
 }  
 }  
 return phase;  
}

**说明：**register方法为phaser添加一个新的party，如果onAdvance正在运行，那么这个方法会等待它运行结束再返回结果。如果当前phaser有父节点，并且当前phaser上没有已注册的party，那么就会交给父节点注册。

register和bulkRegister都由doRegister实现，大概流程如下：

1. 如果当前操作不是首次注册，那么直接在当前phaser上更新注册parties数
2. 如果是首次注册，并且当前phaser没有父节点，说明是root节点注册，直接更新phase
3. 如果当前操作是首次注册，并且当前phaser由父节点，则注册操作交由父节点，并更新当前phaser的phase

上面说过，子Phaser的phase在没有被真正使用之前，允许滞后于它的root节点。非首次注册时，如果Phaser有父节点，则调用reconcileState()方法解决root节点的phase延迟传递问题， 源码如下：

reconcileState()

private long reconcileState() {  
 final Phaser root = this.root;  
 long s = state;  
 if (root != this) {  
 int phase, p;  
 // CAS to root phase with current parties, tripping unarrived  
 while ((phase = (int)(root.state >>> *PHASE\_SHIFT*)) !=  
 (int)(s >>> *PHASE\_SHIFT*) &&  
 !*UNSAFE*.compareAndSwapLong  
 (this, *stateOffset*, s,  
 s = (((long)phase << *PHASE\_SHIFT*) |  
 ((phase < 0) ? (s & *COUNTS\_MASK*) :  
 (((p = (int)s >>> *PARTIES\_SHIFT*) == 0) ? *EMPTY* :  
 ((s & *PARTIES\_MASK*) | p))))))  
 s = state;  
 }  
 return s;  
}

当root节点的phase已经advance到下一代，但是子节点phaser还没有，这种情况下它们必须通过更新未到达parties数 完成它们自己的advance操作(如果parties为0，重置为EMPTY状态)。

回到register方法的第一步，如果当前未到达数为0，说明上一代phase正在进行到达操作，此时调用internalAwaitAdvance()方法等待其他任务完成到达操作，源码如下：

internalAwaitAdvance(phase,QNode)

//阻塞等待phase到下一代  
private int internalAwaitAdvance(int phase, QNode node) {  
 // assert root == this;  
 releaseWaiters(phase-1); // ensure old queue clean  
 boolean queued = false; // true when node is enqueued  
 int lastUnarrived = 0; // to increase spins upon change  
 int spins = *SPINS\_PER\_ARRIVAL*;  
 long s;  
 int p;  
 while ((p = (int)((s = state) >>> *PHASE\_SHIFT*)) == phase) {  
 if (node == null) { // spinning in noninterruptible mode  
 int unarrived = (int)s & *UNARRIVED\_MASK*;//未到达数  
 if (unarrived != lastUnarrived &&  
 (lastUnarrived = unarrived) < *NCPU*)  
 spins += *SPINS\_PER\_ARRIVAL*;  
 boolean interrupted = Thread.*interrupted*();  
 if (interrupted || --spins < 0) { // need node to record intr  
 //使用node记录中断状态  
 node = new QNode(this, phase, false, false, 0L);  
 node.wasInterrupted = interrupted;  
 }  
 }  
 else if (node.isReleasable()) // done or aborted  
 break;  
 else if (!queued) { // push onto queue  
 AtomicReference<QNode> head = (phase & 1) == 0 ? evenQ : oddQ;  
 QNode q = node.next = head.get();  
 if ((q == null || q.phase == phase) &&  
 (int)(state >>> *PHASE\_SHIFT*) == phase) // avoid stale enq  
 queued = head.compareAndSet(q, node);  
 }  
 else {  
 try {  
 ForkJoinPool.*managedBlock*(node);//阻塞给定node  
 } catch (InterruptedException ie) {  
 node.wasInterrupted = true;  
 }  
 }  
 }  
  
 if (node != null) {  
 if (node.thread != null)  
 node.thread = null; // avoid need for unpark()  
 if (node.wasInterrupted && !node.interruptible)  
 Thread.*currentThread*().interrupt();  
 if (p == phase && (p = (int)(state >>> *PHASE\_SHIFT*)) == phase)  
 return abortWait(phase); // possibly clean up on abort  
 }  
 releaseWaiters(phase);  
 return p;  
}

简单介绍下第二个参数node，如果不为空，则说明等待线程需要追踪中断状态或超时状态。以doRegister中的调用为例，不考虑线程争用，internalAwaitAdvance大概流程如下：

1. 首先调用releaseWaiters唤醒上一代所有等待线程，确保旧队列中没有遗留的等待线程。
2. 循环SPINS\_PER\_ARRIVAL指定的次数或者当前线程被中断，创建node记录等待线程及相关信息。
3. 继续循环调用ForkJoinPool.managedBlock运行被阻塞的任务
4. 继续循环，阻塞任务运行成功被释放，跳出循环
5. 最后唤醒当前phase的线程

###### arrive()

//使当前线程到达phaser，不等待其他任务到达。返回arrival phase number  
public int arrive() {  
 return doArrive(*ONE\_ARRIVAL*);  
}

private int doArrive(int adjust) {  
 final Phaser root = this.root;  
 for (;;) {  
 long s = (root == this) ? state : reconcileState();  
 int phase = (int)(s >>> *PHASE\_SHIFT*);  
 if (phase < 0)  
 return phase;  
 int counts = (int)s;  
 //获取未到达数  
 int unarrived = (counts == *EMPTY*) ? 0 : (counts & *UNARRIVED\_MASK*);  
 if (unarrived <= 0)  
 throw new IllegalStateException(badArrive(s));  
 if (*UNSAFE*.compareAndSwapLong(this, *stateOffset*, s, s-=adjust)) {//更新state  
 if (unarrived == 1) {//当前为最后一个未到达的任务  
 long n = s & *PARTIES\_MASK*; // base of next state  
 int nextUnarrived = (int)n >>> *PARTIES\_SHIFT*;  
 if (root == this) {  
 if (onAdvance(phase, nextUnarrived))//检查是否需要终止phaser  
 n |= *TERMINATION\_BIT*;  
 else if (nextUnarrived == 0)  
 n |= *EMPTY*;  
 else  
 n |= nextUnarrived;  
 int nextPhase = (phase + 1) & *MAX\_PHASE*;  
 n |= (long)nextPhase << *PHASE\_SHIFT*;  
 *UNSAFE*.compareAndSwapLong(this, *stateOffset*, s, n);  
 releaseWaiters(phase);//释放等待phase的线程  
 }  
 //分层结构，使用父节点管理arrive  
 else if (nextUnarrived == 0) { //propagate deregistration  
 phase = parent.doArrive(*ONE\_DEREGISTER*);  
 *UNSAFE*.compareAndSwapLong(this, *stateOffset*,  
 s, s | *EMPTY*);  
 }  
 else  
 phase = parent.doArrive(*ONE\_ARRIVAL*);  
 }  
 return phase;  
 }  
 }  
}

**说明：**arrive方法手动调整到达数，使当前线程到达phaser。Arrive和arriveAndDeregister都调用了doArrive实现，大概流程如下：

1. 首先更新state(state - adjust)；
2. 如果当前不是最后一个未到达的任务，直接返回phase
3. 如果当前是最后一个未到达的任务：
   1. 如果当前是root节点，判断是否需要终止phaser，CAS更新phase，最后释放等待的线程；
   2. 如果是分层结构，并且已经没有下一代未到达的parties，则交由父节点处理doArrive逻辑，然后更新state为EMPTY。

###### arriveAndAwaitAdvance()

public int arriveAndAwaitAdvance() {  
 // Specialization of doArrive+awaitAdvance eliminating some reads/paths  
 final Phaser root = this.root;  
 for (;;) {  
 long s = (root == this) ? state : reconcileState();  
 int phase = (int)(s >>> *PHASE\_SHIFT*);  
 if (phase < 0)  
 return phase;  
 int counts = (int)s;  
 int unarrived = (counts == *EMPTY*) ? 0 : (counts & *UNARRIVED\_MASK*);//获取未到达数  
 if (unarrived <= 0)  
 throw new IllegalStateException(badArrive(s));  
 if (*UNSAFE*.compareAndSwapLong(this, *stateOffset*, s,  
 s -= *ONE\_ARRIVAL*)) {//更新state  
 if (unarrived > 1)  
 return root.internalAwaitAdvance(phase, null);//阻塞等待其他任务  
 if (root != this)  
 return parent.arriveAndAwaitAdvance();//子Phaser交给父节点处理  
 long n = s & *PARTIES\_MASK*; // base of next state  
 int nextUnarrived = (int)n >>> *PARTIES\_SHIFT*;  
 if (onAdvance(phase, nextUnarrived))//全部到达，检查是否可销毁  
 n |= *TERMINATION\_BIT*;  
 else if (nextUnarrived == 0)  
 n |= *EMPTY*;  
 else  
 n |= nextUnarrived;  
 int nextPhase = (phase + 1) & *MAX\_PHASE*;//计算下一代phase  
 n |= (long)nextPhase << *PHASE\_SHIFT*;  
 if (!*UNSAFE*.compareAndSwapLong(this, *stateOffset*, s, n))//更新state  
 return (int)(state >>> *PHASE\_SHIFT*); // terminated  
 releaseWaiters(phase);//释放等待phase的线程  
 return nextPhase;  
 }  
 }  
}

**说明：**使当前线程到达phaser并等待其他任务到达，等价于awaitAdvance(arrive())。如果需要等待中断或超时，可以使用awaitAdvance方法完成一个类似的构造。如果需要在到达后取消注册，可以使用awaitAdvance(arriveAndDeregister())。效果类似于CyclicBarrier.await。大概流程如下：

1. 更新state(state - 1)；
2. 如果未到达数大于1，调用internalAwaitAdvance阻塞等待其他任务到达，返回当前phase
3. 如果为分层结构，则交由父节点处理arriveAndAwaitAdvance逻辑
4. 如果未到达数<=1，判断phaser终止状态，CAS更新phase到下一代，最后释放等待当前phase的线程，并返回下一代phase。

###### awaitAdvance()

public int awaitAdvance(int phase) {  
 final Phaser root = this.root;  
 long s = (root == this) ? state : reconcileState();  
 int p = (int)(s >>> *PHASE\_SHIFT*);  
 if (phase < 0)  
 return phase;  
 if (p == phase)  
 return root.internalAwaitAdvance(phase, null);  
 return p;  
}

//响应中断版awaitAdvance

public int awaitAdvanceInterruptibly(int phase)  
 throws InterruptedException {  
 final Phaser root = this.root;  
 long s = (root == this) ? state : reconcileState();  
 int p = (int)(s >>> *PHASE\_SHIFT*);  
 if (phase < 0)  
 return phase;  
 if (p == phase) {  
 QNode node = new QNode(this, phase, true, false, 0L);  
 p = root.internalAwaitAdvance(phase, node);  
 if (node.wasInterrupted)  
 throw new InterruptedException();  
 }  
 return p;  
}

**说明：**awaitAdvance用于阻塞等待线程到达，直到phase前进到下一代，返回下一代的phase number。方法很简单，不多赘述。awaitAdvanceInterruptibly方法是响应中断版的awaitAdvance，不同之处在于，调用阻塞时，会记录线程的中断状态。

##### 小结

**关键点： 用法、分层结构原理。**

#### Semaphore

##### 概述

Semaphore是一个计数信号量，它完全由AQS实现，本质上是一个“共享锁”。Semaphore维护了一个许可集合。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。Semaphore常用来限制可以获取资源的线程数。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实现了自己的同步等待队列，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。

一个基于能力(capability-based)的锁，提供了三种模式来控制read/write的获取。StampedLock的状态由一个版本和模式构成。锁获取方法返回一个表示并控制对锁状态访问的 stamp；0表示没有写锁被授权访问。锁释放和转换方法需要用这个stamp作为参数，如果它与锁状态不匹配操作就会失败。

**写锁：**使用writeLock方法获取，当锁不可用时会阻塞，获取成功后返回一个与这个写锁对应的stamp，在unlockWrite方法中，需要通过这个stamp来释放与之对应的锁。在tryWriteLock同样也会提供这个stamp。当在write模式中获取到写锁时，读锁不能被获取，并且所有的乐观读锁验证(validate方法)都会失败。

**读锁：**使用readLock方法获取，当超出可用资源时（类似AQS的state设计）会阻塞。同样的，在获取锁成功后也会返回stamp，作用与上述相同。tryReadLock同样如此。

**乐观读锁：**使用tryOptimisticRead方法获取，只有在写锁可用时才能成功获取乐观读锁，获取成功后也会返回一个stamp。validate方法可以根据这个stamp来判断写锁是否被获取。这种模式可以理解为一个弱化的读锁(weak version of a read-lock)，它在任何时候都能被破坏。乐观读模式常被用在短的只读的代码段，用来减少争用并提高吞吐量。乐观读区域应该只读取字段，并将它们保存在本地变量中，以便在验证（validate方法）后使用。在乐观读模式中字段的读取可能会不一致，所以可能需要反复调用validate()来检查一致性。例如，当首次读取一个对象或数组引用，然后访问其中一个的字段、元素或方法时，这些步骤通常是必需的。

StampedLock还支持在三种模式中提供有条件地转换的方法。例如，tryConvertToWriteLock方法尝试升级一个锁模式，下面三种情况下可以升级模式并返回一个有效的write stamp：  
(1)已经在writing模式中   
(2)在reading模式中并且已经没有其他读线程   
(3)在乐观读模式中锁可用这些方法的表现形式旨在帮助减少由于基于重试(retry-based)设计造成的代码膨胀

StampedLock 被设计作为线程安全模型的内部工具类。它的使用依赖于对数据、对象和方法的内部属性有一定的了解。**StampedLock 是不可重入的**，所以在锁的内部不能调用其他尝试重复获取锁的方法。一个stamp如果在很长时间都没有使用或验证，在很长一段时间之后可能就会验证失败。StampedLocks是可序列化的，但是反序列化后变为初始的非锁定状态，所以在远程锁定中是不安全的。

StampedLock 的调度策略不会始终偏向读线程或写线程，所有的"try"方法都是尽最大努力获取，并不一定遵循任何调度或公平策略。从"try"方法获取或转换锁失败返回0时，不会携带任何锁的状态信息，随后的一个调用可能会成功获取。

由于StampedLock支持跨多个锁模式的协调使用，它不会直接实现Lock或ReadWriteLock接口。但是，如果应用程序需要Lock的相关功能，它可以通过asReadLock()、asWriteLock()和asReadWriteLock()方法返回一个Lock视图。

##### 核心参数

//获取锁失败入队之前的最大自旋次数（实际运行时并不一定是这个数）  
private static final int *SPINS* = (*NCPU* > 1) ? 1 << 6 : 0;

//头节点获取锁的最大自旋次数  
private static final int *HEAD\_SPINS* = (*NCPU* > 1) ? 1 << 10 : 0;

//头节点再次阻塞前的最大自旋次数  
private static final int *MAX\_HEAD\_SPINS* = (*NCPU* > 1) ? 1 << 16 : 0;

//等待自旋锁溢出的周期数  
private static final int *OVERFLOW\_YIELD\_RATE* = 7; // must be power 2 - 1

//在溢出之前读线程计数用到的bit数  
private static final int *LG\_READERS* = 7;

// Values for lock state and stamp operations  
private static final long *RUNIT* = 1L;//读锁单位  
private static final long *WBIT* = 1L << *LG\_READERS*;//写状态标识 1000 0000  
private static final long *RBITS* = *WBIT* - 1L;//读状态标识 111 1111  
private static final long *RFULL* = *RBITS* - 1L; //读锁最大资源数 111 1110  
private static final long *ABITS* = *RBITS* | *WBIT*; //用于获取锁状态 1111 1111  
private static final long *SBITS* = ~*RBITS*; //note overlap with ABITS

//锁状态初始值  
private static final long *ORIGIN* = *WBIT* << 1;

//中断标识  
private static final long *INTERRUPTED* = 1L;

//节点状态 等待/取消  
private static final int *WAITING* = -1;  
private static final int *CANCELLED* = 1;

//节点模型 读/写  
private static final int *RMODE* = 0;  
private static final int *WMODE* = 1;

**状态判断:**

**state & ABITS == 0L 写锁可用**

**state & ABITS < RFULL 读锁可用**

**state & ABITS == WBIT 写锁已经被其他线程获取**

**state & ABITS == RFULL 读锁饱和，可尝试增加额外资源数**

**(stamp & SBITS) == (state & SBITS) 验证stamp是否为当前已经获取的锁stamp**

**(state & WBIT) != 0L 当前线程已经持有写锁**

**(state & RBITS) != 0L 当前线程已经持有读锁**

**s & RBITS 读锁已经被获取的数量**

##### 函数列表

//构造函数  
public StampedLock() {  
 state = *ORIGIN*;  
}

//获取写锁，等待锁可用  
public long writeLock()

//获取写锁，直接返回  
public long tryWriteLock()

//获取写锁，等待指定的时间  
public long tryWriteLock(long time, TimeUnit unit)

//获取写锁，响应中断  
public long writeLockInterruptibly()

//获取读锁，等待锁可用  
public long readLock()

//尝试获取读锁，直接返回  
public long tryReadLock()

//获取读锁，限制等待时间  
public long tryReadLock(long time, TimeUnit unit)

//获取读锁，响应中断  
public long readLockInterruptibly()

//获取乐观读锁,如果写锁可用获取成功，不修改任何状态值  
public long tryOptimisticRead()

//验证stamp，如果在锁发出给定的stamp之后写锁没有被获取，或者给定stamp是当前已经获取的锁stamp，则返回true。一般用在乐观读锁中，用于判断是否可继续获取读锁。  
public boolean validate(long stamp)

//释放写锁  
public void unlockWrite(long stamp)

//释放读锁  
public void unlockRead(long stamp)

//释放给定stamp对应的锁  
public void unlock(long stamp)

//尝试升级给定stamp对应的锁为写锁  
public long tryConvertToWriteLock(long stamp)

//尝试降级给定stamp对应的锁为读锁  
public long tryConvertToReadLock(long stamp)

//尝试降级给定stamp对应的锁为乐观读锁  
public long tryConvertToOptimisticRead(long stamp)

//尝试释放写锁，一般用在异常复原  
public boolean tryUnlockWrite()

//尝试释放读锁，一般用在异常复原  
public boolean tryUnlockRead()

//写锁是否被持有  
public boolean isWriteLocked()

//读锁是否被持有  
public boolean isReadLocked()

//获取读锁数  
public int getReadLockCount()

//返回一个ReadLock  
public Lock asReadLock()

//返回一个WriteLock  
public Lock asWriteLock()

//返回一个ReadWriteLock  
public ReadWriteLock asReadWriteLock()

##### 源码解析

###### writeLock()

//获取写锁，等待可用  
public long writeLock() {  
 long s, next; // bypass acquireWrite in fully unlocked case only  
 return ((((s = state) & *ABITS*) == 0L &&  
 *U*.compareAndSwapLong(this, *STATE*, s, next = s + *WBIT*)) ?  
 next : acquireWrite(false, 0L));  
}

WNode node = null, p;  
 //第一个自旋，准备入队  
 for (int spins = -1;;) { // spin while enqueuing  
 long m, s, ns;  
 if ((m = (s = state) & *ABITS*) == 0L) {//锁可用  
 if (*U*.compareAndSwapLong(this, *STATE*, s, ns = s + *WBIT*))//获取锁 CAS修改锁状态  
 return ns;  
 }  
 else if (spins < 0)  
 spins = (m == *WBIT* && wtail == whead) ? *SPINS* : 0;//自旋次数  
 else if (spins > 0) {  
 if (LockSupport.*nextSecondarySeed*() >= 0)  
 --spins; //随机递减  
 }  
 else if ((p = wtail) == null) { // initialize queue  
 WNode hd = new WNode(*WMODE*, null);//初始化写锁等待队列  
 if (*U*.compareAndSwapObject(this, *WHEAD*, null, hd))  
 wtail = hd;  
 }  
 else if (node == null)  
 node = new WNode(*WMODE*, p);//创建新的等待节点  
 else if (node.prev != p)  
 node.prev = p;  
 else if (*U*.compareAndSwapObject(this, *WTAIL*, p, node)) {//更新tail节点  
 p.next = node;  
 break;  
 }  
 }  
  
 //第二个自旋，节点依次获取锁  
 for (int spins = -1;;) {  
 WNode h, np, pp; int ps;  
 if ((h = whead) == p) {//当前节点是最后一个等待节点  
 if (spins < 0)  
 spins = *HEAD\_SPINS*; //头结点自旋次数  
 else if (spins < *MAX\_HEAD\_SPINS*)  
 spins <<= 1; // spins=spins/2  
 for (int k = spins;;) { // spin at head  
 long s, ns;  
 if (((s = state) & *ABITS*) == 0L) {//锁可用  
 if (*U*.compareAndSwapLong(this, *STATE*, s,  
 ns = s + *WBIT*)) {//更新锁状态  
 //更新头结点，返回stamp  
 whead = node;  
 node.prev = null;  
 return ns;  
 }  
 }  
 else if (LockSupport.*nextSecondarySeed*() >= 0 &&  
 --k <= 0)//随机递减  
 break;  
 }  
 }  
 else if (h != null) { // help release stale waiters  
 WNode c; Thread w;  
 //依次唤醒头节点的cowait节点线程  
 while ((c = h.cowait) != null) {//有等待读的线程  
 if (*U*.compareAndSwapObject(h, *WCOWAIT*, c, c.cowait) && //CAS更新头结点的cowait  
 (w = c.thread) != null)  
 *U*.unpark(w);  
 }  
 }  
 if (whead == h) {  
 //检查队列稳定性  
 if ((np = node.prev) != p) {  
 if (np != null)  
 (p = np).next = node; // stale  
 }  
 else if ((ps = p.status) == 0)  
 *U*.compareAndSwapInt(p, *WSTATUS*, 0, *WAITING*);  
 else if (ps == *CANCELLED*) {//尾节点取消，更新尾节点的前继节点为p.prev，继续自旋  
 if ((pp = p.prev) != null) {  
 node.prev = pp;  
 pp.next = node;  
 }  
 }  
 else {  
 long time; // 0 argument to park means no timeout  
 if (deadline == 0L)  
 time = 0L;  
 else if ((time = deadline - System.*nanoTime*()) <= 0L)  
 return cancelWaiter(node, node, false);//超时，取消等待  
 Thread wt = Thread.*currentThread*();  
 *U*.putObject(wt, *PARKBLOCKER*, this);  
 node.thread = wt;  
 if (p.status < 0 && (p != h || (state & *ABITS*) != 0L) &&  
 whead == h && node.prev == p)  
 *U*.park(false, time); // emulate LockSupport.park  
 node.thread = null;  
 *U*.putObject(wt, *PARKBLOCKER*, null);  
 if (interruptible && Thread.*interrupted*())  
 return cancelWaiter(node, node, true);//中断，取消等待  
 }  
 }  
 }  
}

**说明：**获取写锁，如果锁可用((state & ABITS) == 0L)则直接获取写锁并返回stamp，否则调用acquireWrite等待锁可用，acquireWrite函数执行流程如下：

1. 第一个自旋，使当前线程进入等待队列的尾节点。
2. 第二个自旋，节点依次获取写锁，直到当前线程所在节点的前继节点(prev)为头结点时，如果锁可用，则说明可以获取锁，获取成功返回stamp

###### readLock()

//获取写锁，等待锁可用  
public long readLock() {  
 long s = state, next; // bypass acquireRead on common uncontended case  
 return ((whead == wtail && (s & *ABITS*) < *RFULL* && //还有可用资源  
 *U*.compareAndSwapLong(this, *STATE*, s, next = s + *RUNIT*)) ?  
 next : acquireRead(false, 0L));  
}

private long acquireRead(boolean interruptible, long deadline) {  
 WNode node = null, p;  
 //第一个自旋，入队  
 for (int spins = -1;;) {  
 WNode h;  
 if ((h = whead) == (p = wtail)) {//等待队列为空  
 for (long m, s, ns;;) {  
 if ((m = (s = state) & *ABITS*) < *RFULL* ? //有可用资源  
 *U*.compareAndSwapLong(this, *STATE*, s, ns = s + *RUNIT*) :  
 (m < *WBIT* && (ns = tryIncReaderOverflow(s)) != 0L))  
 return ns;  
 else if (m >= *WBIT*) {  
 if (spins > 0) {  
 if (LockSupport.*nextSecondarySeed*() >= 0)  
 --spins;//随机递减自旋数  
 }  
 else {  
 if (spins == 0) { //自旋结束，准备进入等待队列  
 WNode nh = whead, np = wtail;  
 if ((nh == h && np == p) || (h = nh) != (p = np))  
 break;  
 }  
 spins = *SPINS*;  
 }  
 }  
 }  
 }  
 if (p == null) { // initialize queue  
 //初始化等待队列  
 WNode hd = new WNode(*WMODE*, null);  
 if (*U*.compareAndSwapObject(this, *WHEAD*, null, hd))  
 wtail = hd;  
 }  
 else if (node == null)  
 node = new WNode(*RMODE*, p);//创建新的节点  
 else if (h == p || p.mode != *RMODE*) {  
 //到这里说明尾节点是写线程  
 if (node.prev != p)  
 node.prev = p;  
 else if (*U*.compareAndSwapObject(this, *WTAIL*, p, node)) {//更新tail节点为当前节点  
 p.next = node;  
 break;  
 }  
 }  
 else if (!*U*.compareAndSwapObject(p, *WCOWAIT*,  
 node.cowait = p.cowait, node))//到这里说明尾节点是等待读的节点，CAS把当前节点(node节点)转移到p节点的cowait上  
 node.cowait = null;  
 else {  
 //当前节点进入等待队列成功后的逻辑(当前节点已被转移到尾节点的cowait上)  
 for (;;) {  
 WNode pp, c; Thread w;  
 if ((h = whead) != null && (c = h.cowait) != null &&  
 *U*.compareAndSwapObject(h, *WCOWAIT*, c, c.cowait) &&  
 (w = c.thread) != null) // help release  
 *U*.unpark(w); //唤醒头节点等待读线程  
 if (h == (pp = p.prev) || h == p || pp == null) {//没有前继节点，可以尝试唤醒当前节点等待的线程  
 long m, s, ns;  
 do {  
 if ((m = (s = state) & *ABITS*) < *RFULL* ?  
 *U*.compareAndSwapLong(this, *STATE*, s,  
 ns = s + *RUNIT*) : //获取锁  
 (m < *WBIT* &&  
 (ns = tryIncReaderOverflow(s)) != 0L))//读锁饱和，尝试增加额外的读锁数量，只有在读锁数=RFULL时才可以增加  
 return ns; //返回stamp  
 } while (m < *WBIT*);  
 }  
 //超时及中断判断逻辑  
 if (whead == h && p.prev == pp) {//检查队列是否稳定  
 long time;  
 if (pp == null || h == p || p.status > 0) {  
 node = null; // throw away  
 break;  
 }  
 if (deadline == 0L)  
 time = 0L;  
 else if ((time = deadline - System.*nanoTime*()) <= 0L)  
 return cancelWaiter(node, p, false);//超时，取消等待  
 Thread wt = Thread.*currentThread*();  
 *U*.putObject(wt, *PARKBLOCKER*, this);  
 node.thread = wt;  
 if ((h != pp || (state & *ABITS*) == *WBIT*) &&  
 whead == h && p.prev == pp)  
 *U*.park(false, time);//阻塞等待  
 node.thread = null;  
 *U*.putObject(wt, *PARKBLOCKER*, null);  
 if (interruptible && Thread.*interrupted*())  
 return cancelWaiter(node, p, true);//被中断，取消等待  
 }  
 }  
 }  
 }  
  
 //第二个自旋，节点依次获取锁  
 for (int spins = -1;;) {  
 WNode h, np, pp; int ps;  
 if ((h = whead) == p) {//当前节点是最后一个等待节点  
 if (spins < 0)  
 spins = *HEAD\_SPINS*;//初始化自旋数  
 else if (spins < *MAX\_HEAD\_SPINS*)  
 spins <<= 1;  
 for (int k = spins;;) { // spin at head  
 long m, s, ns;  
 if ((m = (s = state) & *ABITS*) < *RFULL* ? //有可用资源  
 *U*.compareAndSwapLong(this, *STATE*, s, ns = s + *RUNIT*) :  
 (m < *WBIT* && (ns = tryIncReaderOverflow(s)) != 0L)) {  
 //获取读锁成功，更新头节点为当前节点  
 WNode c; Thread w;  
 whead = node;  
 node.prev = null;  
 //依次唤醒当前节点的cowait节点线程  
 while ((c = node.cowait) != null) {  
 if (*U*.compareAndSwapObject(node, *WCOWAIT*,  
 c, c.cowait) &&  
 (w = c.thread) != null)  
 *U*.unpark(w);  
 }  
 return ns;  
 }  
 else if (m >= *WBIT* &&  
 LockSupport.*nextSecondarySeed*() >= 0 && --k <= 0)//随机递减自旋次数  
 break;  
 }  
 }  
 else if (h != null) {  
 WNode c; Thread w;  
 while ((c = h.cowait) != null) {  
 //依次唤醒head节点的cowait节点线程  
 if (*U*.compareAndSwapObject(h, *WCOWAIT*, c, c.cowait) &&  
 (w = c.thread) != null)  
 *U*.unpark(w);  
 }  
 }  
 if (whead == h) {  
 //检查队列稳定性  
 if ((np = node.prev) != p) {  
 if (np != null)  
 (p = np).next = node; // stale  
 }  
 else if ((ps = p.status) == 0)  
 *U*.compareAndSwapInt(p, *WSTATUS*, 0, *WAITING*);  
 else if (ps == *CANCELLED*) {//尾节点取消，更新尾节点的前继节点为p.prev，继续自旋  
 if ((pp = p.prev) != null) {  
 node.prev = pp;  
 pp.next = node;  
 }  
 }  
 else {  
 //超时及中断判断逻辑  
 long time;  
 if (deadline == 0L)  
 time = 0L;  
 else if ((time = deadline - System.*nanoTime*()) <= 0L)  
 return cancelWaiter(node, node, false);//超时，取消等待  
 Thread wt = Thread.*currentThread*();  
 *U*.putObject(wt, *PARKBLOCKER*, this);  
 node.thread = wt;  
 if (p.status < 0 &&  
 (p != h || (state & *ABITS*) == *WBIT*) &&  
 whead == h && node.prev == p)  
 *U*.park(false, time);  
 node.thread = null;  
 *U*.putObject(wt, *PARKBLOCKER*, null);  
 if (interruptible && Thread.*interrupted*())  
 return cancelWaiter(node, node, true);//被中断，取消等待  
 }  
 }  
 }  
}

**说明：**获取读锁，代码比较多，但逻辑很简单，跟wirteLock()差不多。详细的流程这里就不细说了，有兴趣的同学可以参考笔者添加的注释一步一步阅读。  
如果有可用资源((state & ABITS) < RFULL)则直接获取读锁并返回stamp，否则调用acquireRead等待锁可用，acquireRead函数执行流程如下：

1. 第一个自旋，使当前线程进入等待队列的尾节点。注意这里跟获取写锁时的区别，在获取写锁时，把当前线程所在的节点直接放入队尾；但是在获取读锁时，是把当前线程所在的节点放入尾节点的cowait节点里。
2. 第二个自旋，节点依次获取读锁。直到当前线程所在节点的前继节点(prev)为头结点时，如果有可用资源，则说明可以获取锁，获取成功返回stamp
3. 如果在自旋中未能成功获取到锁，并且线程被中断或者等待超时，则调用cancelWaiter方法取消节点的等待，cancelWaiter后面会分析。

###### tryOptimisticRead()

//获取乐观读锁   
public long tryOptimisticRead() {  
 long s;  
 return (((s = state) & *WBIT*) == 0L) ? (s & *SBITS*) : 0L;  
}

说明：获取乐观读锁，取决于写锁，如果写锁空闲则获取成功。并且不修改任何状态值。函数比较简单，不多赘述。

###### cancelWaiter()

//取消给定节点  
private long cancelWaiter(WNode node, WNode group, boolean interrupted) {  
 if (node != null && group != null) {  
 Thread w;  
 node.status = *CANCELLED*;//修改节点状态  
 // unsplice cancelled nodes from group  
 //依次解除已经取消的cowait节点的链接  
 for (WNode p = group, q; (q = p.cowait) != null;) {  
 if (q.status == *CANCELLED*) {  
 *U*.compareAndSwapObject(p, *WCOWAIT*, q, q.cowait);  
 p = group; // restart  
 }  
 else  
 p = q;  
 }  
 if (group == node) {  
 //依次唤醒节点上的未取消的cowait节点线程  
 for (WNode r = group.cowait; r != null; r = r.cowait) {  
 if ((w = r.thread) != null)  
 *U*.unpark(w); // wake up uncancelled co-waiters  
 }  
 //  
 for (WNode pred = node.prev; pred != null; ) { // unsplice  
 WNode succ, pp; // find valid successor  
 while ((succ = node.next) == null ||  
 succ.status == *CANCELLED*) { //后继节点为空或者已经取消，则去查找一个有效的后继节点  
 WNode q = null; // find successor the slow way  
 //从尾节点开始往前查找距离node节点最近的一个有效节点q  
 for (WNode t = wtail; t != null && t != node; t = t.prev)  
 if (t.status != *CANCELLED*)  
 q = t; // don't link if succ cancelled  
 if (succ == q || // ensure accurate successor  
 //运行到这里说明从node到“距离node最近的一个有效节点q”之间可能存在已经取消的节点  
 // CAS替换node的后继节点为“距离node最近的一个有效节点”，也就是说解除了“所有已经取消但是还存在在链表上的无效节点”的链接  
 *U*.compareAndSwapObject(node, *WNEXT*,  
 succ, succ = q)) {  
 if (succ == null && node == wtail) {  
 //运行到这里说明node为尾节点，  
 //利用CAS先修改尾节点为node的前继有效节点，后面再解除node的链接  
 *U*.compareAndSwapObject(this, *WTAIL*, node, pred);  
 }  
 break;  
 }  
 }  
 //解除node节点的链接  
 if (pred.next == node) // unsplice pred link  
 *U*.compareAndSwapObject(pred, *WNEXT*, node, succ);  
 //唤醒后继节点的线程  
 if (succ != null && (w = succ.thread) != null) {  
 succ.thread = null;  
 *U*.unpark(w); // wake up succ to observe new pred  
 }  
 //如果前继节点已经取消，向前查找一个有效节点继续循环，如果这个节点为空则直接跳出循环  
 if (pred.status != *CANCELLED* || (pp = pred.prev) == null)  
 break;  
 node.prev = pp; // repeat if new pred wrong/cancelled  
 *U*.compareAndSwapObject(pp, *WNEXT*, pred, succ);  
 pred = pp;  
 }  
 }  
 }  
 //检查是否可唤醒head节点的后继节点线程  
 WNode h; // Possibly release first waiter  
 while ((h = whead) != null) {  
 long s; WNode q; // similar to release() but check eligibility  
 if ((q = h.next) == null || q.status == *CANCELLED*) {  
 //从尾节点向前查找一个未取消的节点，作为头节点的next节点  
 for (WNode t = wtail; t != null && t != h; t = t.prev)  
 if (t.status <= 0)  
 q = t;  
 }  
 if (h == whead) {  
 if (q != null && h.status == 0 &&  
 ((s = state) & *ABITS*) != *WBIT* && // waiter is eligible  
 (s == 0L || q.mode == *RMODE*))//锁可用，或者后继节点是读线程  
 release(h);//可以唤醒头节点的后继节点线程  
 break;  
 }  
 }  
 return (interrupted || Thread.*interrupted*()) ? *INTERRUPTED* : 0L;  
}

**说明：**如果节点线程被中断或者等待超时，需要取消节点的链接。大概的操作就是首先修改节点为取消状态，然后解除它在等待队列中的链接，并且唤醒节点上所有等待读的线程(也就是cowait节点)；最后如果锁可用，帮助唤醒头节点的后继节点的线程。其实也是AQS中取消获取锁方法的一种变体（详见AQS篇）。

重点介绍一下cancelWaiter的前两个参数node和group：

1. 如果node!=group，说明node节点是group节点上的一个cowait节点（如果不明白请见前面acquireRead方法中的U.compareAndSwapObject(p, WCOWAIT,node.cowait = p.cowait, node)这一行代码的注释），这种情况下首先修改node节点的状态(node.status = CANCELLED)，然后直接操作group节点，依次解除group节点上已经取消的cowait节点的链接。最后如果锁可用，帮助唤醒头节点的后继节点的线程。
2. 如果node==group，说明在node节点之前的节点为写线程节点，这时需要进行以下操作：
   1. 依次唤醒node节点上的未取消的cowait节点线程
   2. 解除node节点和一段节点（node节点到“距离node最近的一个有效节点”）的链接
   3. 最后如果锁可用，帮助唤醒头节点的后继节点的线程。

##### 小结

关于StampedLock的其他方法都比较简单，这里就不在详细介绍了，有兴趣的同学可以去我的git下载源码查阅，上面有笔者读源码时添加的一些注释。

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);  
 }  
 }  
}

### 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

CopyOnWriteArrayList是线程安全的ArrayList。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();//加锁  
 try {  
 Object[] elements = getArray();  
 int len = elements.length;  
 if (index > len || index < 0)  
 throw new IndexOutOfBoundsException("Index: "+index+  
 ", Size: "+len);  
 Object[] newElements;  
 int numMoved = len - index;//需要拷贝的后半段元素数  
 if (numMoved == 0)  
 newElements = Arrays.*copyOf*(elements, len + 1);  
 else {  
 newElements = new Object[len + 1];  
 //拷贝0-index数据  
 System.*arraycopy*(elements, 0, newElements, 0, index);  
 //拷贝index后数据  
 System.*arraycopy*(elements, index, newElements, index + 1,  
 numMoved);  
 }  
 newElements[index] = element;  
 setArray(newElements);  
 } finally {  
 lock.unlock();  
 }  
}

**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);  
}

说明： 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。

##### 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);

##### 数据结构和核心参数



首先来看一下用来存储K-V元素的Node节点类：



* + - 1. **Node<K, V>：** 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。
      2. **TreeNode<K, V>：**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完成红黑树的封装。
      3. **TreeBin<K, V>：**封装了TreeNode，红黑树的根节点，也就是说在ConcurrentHashMap中红黑树存储的是TreeBin对象。
      4. **ForwardingNode<K, V>：**在节点转移时用于连接两个table（table和nextTable）的节点类。包含一个nextTable指针，用于指向下一个table。而且这个节点的k-v和next指针全部为null，hash值为-1。只有在扩容时发挥作用，作为一个占位节点放在table中表示当前节点已经被移动。
      5. **ReservationNode<K,V>：**在computeIfAbsent和compute方法计算时当做一个占位节点，表示当前节点已经被占用，在compute或computeIfAbsent的function计算完成后插入元素。hash值为-3。

所以，在ConcurrentHashMap中，节点的分布如下：



###### 核心参数

//最大容量  
private static final int *MAXIMUM\_CAPACITY* = 1 << 30;

//初始容量  
private static final int *DEFAULT\_CAPACITY* = 16;

//数组最大容量  
static final int *MAX\_ARRAY\_SIZE* = Integer.*MAX\_VALUE* - 8;

//默认并发度，兼容1.7及之前版本  
private static final int *DEFAULT\_CONCURRENCY\_LEVEL* = 16;

//加载/扩容因子，实际使用n - (n >>> 2)  
private static final float *LOAD\_FACTOR* = 0.75f;

//链表转红黑树的节点数阀值  
static final int *TREEIFY\_THRESHOLD* = 8;

//红黑树转链表的节点数阀值  
static final int *UNTREEIFY\_THRESHOLD* = 6;

//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  
static final int *MIN\_TREEIFY\_CAPACITY* = 64;

//扩容时任务的最小转移节点数  
private static final int *MIN\_TRANSFER\_STRIDE* = 16;

//sizeCtl中记录stamp的位数  
private static int *RESIZE\_STAMP\_BITS* = 16;

//帮助扩容的最大线程数  
private static final int *MAX\_RESIZERS* = (1 << (32 - *RESIZE\_STAMP\_BITS*)) - 1;

//size在sizeCtl中的偏移量  
private static final int *RESIZE\_STAMP\_SHIFT* = 32 - *RESIZE\_STAMP\_BITS*;

//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;

//基础计数器值(size = baseCount + CounterCell[i].value)  
private transient volatile long baseCount;

//控制table初始化和扩容操作  
private transient volatile int sizeCtl;

//节点转移时下一个需要转移的table索引  
private transient volatile int transferIndex;

//元素变化时用于控制自旋  
private transient volatile int cellsBusy;

// 保存table中的每个节点的元素个数 2的幂次方  
// size = baseCount + CounterCell[i].value  
private transient volatile CounterCell[] counterCells;

**table**：Node数组，在第一次插入元素的时候初始化，默认初始大小为16，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，扩容时生成的新的数组，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
* 初始化数组或扩容完成后,将sizeCtl的值设为0.75\*n (n-(n >>> 2)) n=tab.length
* 在扩容操作在进行节点转移前，sizeCtl改为(hash << RESIZE\_STAMP\_SHIFT) + 2，这个值为负数，并且每有一个线程参与扩容操作sizeCtl就加1

**transferIndex**：扩容时用到，初始时为table.length，表示从索引0到transferIndex的节点还未转移

**counterCells：**ConcurrentHashMap的特定计数器，实现方法跟LongAdder类似。这个计数器的机制避免了在更新时的资源争用，但是如果并发读取太频繁会导致缓存超负荷，为了避免读取太频繁，只有在添加了两个以上节点时才可以尝试扩容操作。在统一hash分配的前提下，发生这种情况的概率在13%左右，也就是说只有大约1/8的put操作才会检查扩容（并且在扩容后会更少）。

**hash计算公式**：hash = (key.hashCode ^ (key.hashCode >>> 16)) & HASH\_BITS

**索引计算公式**：(table.length-1)&hash

##### 源码解析

###### 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;;) {//自旋  
 //f:索引节点; n:tab.length; i:新节点索引 (n - 1) & hash; fh:f.hash  
 Node<K,V> f; int n, i, fh;  
 if (tab == null || (n = tab.length) == 0)  
 //初始化  
 tab = initTable();  
 else if ((f = *tabAt*(tab, i = (n - 1) & hash)) == null) {//索引i节点为空，直接插入  
 //cas插入节点,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-其他线程正在进行节点转移操作  
 else if ((fh = f.hash) == *MOVED*)  
 //帮助转移  
 tab = helpTransfer(tab, f);  
 else {  
 V oldVal = null;  
 synchronized (f) {  
 if (*tabAt*(tab, i) == f) {//check stable  
 //f.hash>=0,说明f是链表的头结点  
 if (fh >= 0) {  
 binCount = 1;//记录链表节点数，用于后面是否转换为红黑树做判断  
 for (Node<K,V> e = f;; ++binCount) {  
 K ek;  
 //key相同 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;  
 //到这里说明已经是链表尾，把当前值作为新的节点插入到队尾  
 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;  
 }  
 }  
 }  
 //更新新元素个数  
 addCount(1L, binCount);  
 return null;  
}

**说明:**首先说一下put的流程，后面再详细分析每一个流程的具体实现：

* 1. 计算当前key的hash值，根据hash值计算索引i（i=(table.length - 1) & hash）；
  2. 如果当前table为null，说明是第一次进行put操作，调用initTable()初始化table；
  3. 如果索引i位置的节点f为空，则直接把当前值作为新的节点直接插入到索引i位置；
  4. 如果节点f.hash == MOVED(-1)，说明当前节点处于移动状态（或者说是其他线程正在对f节点进行转移操作），此时调用helpTransfer(tab, f)帮助转移；
  5. 如果不属于上述条件，此时需要对节点f进行操作，首先使用内置锁synchronized对f节点进行加锁；
     1. 如果f.hash>=0，说明i位置是一个链表，并且节点f是这个链表的头节点，则对f节点进行遍历，此时分两种情况：
        + 如果链表中某个节点e的hash与当前key的hash相同，则对这个节点e的value进行修改操作。
        + 如果遍历到链表尾都没有找到与当前key的hash相同的节点，则把当前K-V作为一个新的节点插入到这个链表尾部。
     2. 如果f节点是TreeBin节点(f instanceof TreeBin)，说明索引i位置的节点是一个红黑树，则调用putTreeVal方法找到一个已存在的节点进行修改，或者是把当前K-V放入一个新的节点（put or update）。
  6. 完成插入后，如果索引i处是一个链表，并且在插入新的节点后节点数>8，则调用treeifyBin把链表转换为红黑树。
  7. 最后，更新元素个数

1.4.4.2.1 initTable()

初始化操作，ConcurrentHashMap的初始化在第一次插入数据的时候(判断table是否为null)，注意初始化操作为单线程操作（如果有其他线程正在进行初始化，则调用Thread.yield()让出CPU时间片，自旋等待table初始完成）。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) {  
 if ((sc = sizeCtl) < 0)//其他线程正在进行初始化或转移操作，让出CPU执行时间片，继续自旋  
 Thread.*yield*(); // lost initialization race; just spin  
 else if (*U*.compareAndSwapInt(this, *SIZECTL*, sc, -1)) {//CAS设置sizectl为-1 表示当前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);//0.75\*n 设置扩容阈值  
 }  
 } finally {  
 sizeCtl = sc;//初始化sizeCtl=0.75\*n  
 }  
 break;  
 }  
 }  
 return tab;  
}

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

如果索引到的节点的hash == MOVED(-1)，说明当前节点处于移动状态（或者说是其他线程正在对f节点进行转移操作。这里主要是靠ForwardingNode节点来检测，在transfer方法中，被转移后的节点会改为ForwardingNode，它是一个占位节点，并且hash=MOVED，也就是说，我们可以通过判断hash是否为MOVED来确定当前节点的状态），此时调用helpTransfer(tab, f)帮助转移，主要操作就是更新帮助转移的线程数（sizeCtl+1），然后调用transfer方法进行转移操作。源码如下：

//帮助其他线程进行转移操作  
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)//不需要帮助转移，跳出  
 break;  
 if (*U*.compareAndSwapInt(this, *SIZECTL*, sc, sc + 1)) {//CAS更新帮助转移的线程数  
 transfer(tab, nextTab);  
 break;  
 }  
 }  
 return nextTab;  
 }  
 return table;  
}

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

在put操作完成后，如果当前节点为一个链表，并且链表长度>=TREEIFY\_THRESHOLD(8)，此时就需要调用treeifyBin方法来把当前链表转为一个红黑树。treeifyBin主要进行两步操作：

1. 如果当前table长度还未超过MIN\_TREEIFY\_CAPACITY(64)，则优先对数组进行扩容操作，容量为原来的2倍(n<<1)。
2. 否则就对当前节点进行转换操作（注意这个操作是单线程完成的）。遍历链表节点，把Node转换为TreeNode，然后在通过TreeBin来构造红黑树（红黑树的构造相信大家都已经很熟悉了，这里就不在详细介绍了）。

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) {//check stable  
 //hd：节点头  
 TreeNode<K,V> hd = null, tl = null;  
 //遍历转换节点  
 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));  
 }  
 }  
 }  
 }  
}

1.4.4.2.4 tryPresize(int size)

当table容量不足时，需要对其进行两倍扩容。tryPresize方法很简单，主要就是用来检查扩容前的必要条件（比如是否超过最大容量），真正的扩容其实也可以叫“节点转移”，主要是通过transfer方法完成，后面我们会详细解析transfer，首先来简单看一下tryPresize源码：

*/\*\*  
 \* 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);  
 }  
 }  
}

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

**transfer方法是table扩容的核心实现**。由于ConcurrentHashMap的扩容是新建一个table，所以主要问题就是如何把旧table的元素转移到新的table上。所以，扩容问题就演变成了“节点转移”问题。首先总结一下需要转移节点（调用transfer）的几个条件：

1. 对table进行扩容时
2. 在更新元素数目时(addCount方法)，元素总数>=sizeCtl（sizeCtl=0.75n，达到扩容阀值），此时也需要扩容
3. 在put操作时，发现索引节点正在转移(hash==MOVED)，此时需要帮助转移

transfer源码如下:

//转移或复制节点到新的table  
private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {  
 int n = tab.length, stride;  
 //转移幅度( tab.length/(NCPU\*8) )，最小为16  
 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;//初始为table的最后一个索引  
 }  
 int nextn = nextTab.length;  
 //初始化ForwardingNode节点,持有nextTab的引用,在处理完每个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;//节点是否已经处理  
 boolean finishing = false; // to ensure sweep before committing nextTab  
 //自旋移动每个节点，从transferIndex开始移动stride个节点到新的table。  
 //i：当前处理的Node索引；bound：需要处理节点的索引边界  
 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;  
 //通过while循环获取本次需要移动的节点索引i  
 while (advance) {  
 //nextIndex:下一个要处理的节点索引; nextBound:下一个需要处理的节点的索引边界  
 int nextIndex, nextBound;  
 if (--i >= bound || finishing)//通过--i控制下一个需要移动的节点  
 advance = false;  
 //节点已全部转移  
 else if ((nextIndex = transferIndex) <= 0) {  
 i = -1;  
 advance = false;  
 }  
 //transferIndex（初值为最后一个节点的索引），表示从transferIndex开始后面所有的节点都已分配，  
 //每次线程领取扩容任务后，需要更新transferIndex的值(transferIndex-stride)。  
 //CAS修改transferIndex，并更新索引边界  
 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) {//已完成转移，更新相关属性  
 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;  
 }  
 //当前线程已经完成转移，但可能还有其他线程正在进行转移操作  
 //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=(rs << RESIZE\_STAMP\_SHIFT) + 2)  
 //这里判断如果还有其他线程正在操作，直接返回，否则的话重新初始化i对原tab进行一遍检查然后再提交  
 if ((sc - 2) != *resizeStamp*(n) << *RESIZE\_STAMP\_SHIFT*)  
 return;  
 finishing = advance = true;  
 i = n; // recheck before commit  
 }  
 }  
 else if ((f = *tabAt*(tab, i)) == null)  
 advance = *casTabAt*(tab, i, null, fwd);//i位置节点为空，替换为ForwardingNode节点，用于通知其他线程该位置已经处理  
 else if ((fh = f.hash) == *MOVED*)//节点已经被其他线程处理过，继续处理下一个节点  
 advance = true; // already processed  
 else {  
 synchronized (f) {  
 if (*tabAt*(tab, i) == f) {//check stable  
 //处理当前拿到的节点,构建两个node:ln/hn。ln:原位置; hn:i+n位置  
 Node<K,V> ln, hn;  
  
 if (fh >= 0) {//当前为链表节点（fh>=0）  
 //使用fn&n把原链表中的元素分成两份（fn&n = n or 0）  
 //在表扩容2倍后，索引i可能发生改变，如果原table长度n=2^x，如果hash的x位为1，此时需要加上x位的值，也就是i+n；  
 //如果x位为0，索引i不变  
 int runBit = fh & n; // n or 0  
 //最后一个与头节点f索引不同的节点  
 Node<K,V> lastRun = f;  
 //从索引i的节点开始向后查找最后一个有效节点  
 for (Node<K,V> p = f.next; p != null; p = p.next) {  
 int b = p.hash & n;//n or 0  
 if (b != runBit) {  
 runBit = b;  
 lastRun = p;  
 }  
 }  
 if (runBit == 0) {  
 ln = lastRun;  
 hn = null;  
 } else {  
 hn = lastRun;  
 ln = null;  
 }  
 //把f链表分解为两个链表  
 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;  
 }  
 */\*\*  
 \* 如果该节点是红黑树结构，则构造树节点lo和hi，遍历红黑树中的节点，同样是根据hash&tab.length算法，  
 \* 把节点分为两类，分别插入索引i和(i+n)位置。  
 \*/* else if (f instanceof TreeBin) {  
 //转为根结点  
 TreeBin<K,V> t = (TreeBin<K,V>)f;  
 TreeNode<K,V> lo = null, loTail = null;//低位(i)节点和低位尾节点  
 TreeNode<K,V> hi = null, hiTail = null;//高位(i+n)节点和高位尾节点  
 int lc = 0, hc = 0;  
 //从首个节点向后遍历  
 for (Node<K,V> e = t.first; e != null; e = e.next) {  
 int h = e.hash;  
 //构建树节点  
 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：transfer方法相对比较复杂，在对其进行分析之前，请大家先记住这张图：**



在进行节点转移之前，首先要做的就是重新初始化sizeCtl的值（sizeCtl = (hash << RESIZE\_STAMP\_SHIFT) + 2），这个值是一个负值，用于标识当前table正在进行转移操作，并且每有一个线程参与转移，sizeCtl就加1。transfer执行步骤如下（结合源码注释）：

* 1. 计算转移幅度stride（或者说是当前线程需要转移的节点数），最小为16；
  2. 创建一个相当于当前table两倍容量的Node数组，转移完成后用作新的table；
  3. 从transferIndex（初始为table.length，也就是table的最后一个节点）开始，依次向前处理stride个节点。前面介绍过，table的每个节点都可能是一个链表结构，因为在put的时候是根据(table.length-1)&hash计算出的索引，当插入新值时，如果通过key计算出的索引已经存在节点，那么这个新值就放在这个索引位节点的尾部(Node.next)。所以，在进行节点转移后，由于table.length变为原来的两倍，所以相应的索引也会改变，这时候就需要对链表进行分割，我们来看一下这个分割算法：
     + 假设当前处理的节点table[i]=f，并且它是一个链表结构，原table容量为n=2^x，索引计算公式为i=(n - 1)&hash。在表扩张后，由于容量n变为2^(x+1)，所以索引计算就变为，i=(2n - 1)&hash。如果hash的x位为0，则hash&(2^x)=0，此时hash&((2^x)-1)== hash&((2^(x+1))-1)，索引位i不变；如果hash的x位为1，则hash&(2^x)=(2^x)==n，在扩容后x变为x+1，此时需要加上x位的值，即hash&((2^x)-1) + hash&(2^x)，也就是i+n。举个栗子：设n=100000(2^5)，x=5，hash为100101。n-1=011111，那么i=hash&(n-1)=000101；扩容后容量变为m=1000000(2^6)，m-1=0111111，那么i就变成了hash&(m-1)=100101，此时就需要加上x位的值，也就是hash&n。
     + 如果当前节点为红黑树结构，也是利用这个算法进行分割，不同的是，在分割完成之后，如果这两个新的树节点<=6，则调用untreeify方法把树结构转为链表结构。
  4. 最后把操作过的节点都设为ForwardingNode节点（hash= MOVED，这样别的线程就可以检测到）

1.4.4.2.6 addCount(long,int)

put操作全部完成后，别忘了更新元素数量。addCount用来更新ConcurrentHashMap的元素数，根据所传参数check决定是否检查扩容，如果需要，调用transfer方法进行扩容/节点转移。这里面有一个看起来比较复杂的方法fullAddCount，作用是在线程争用资源时，使用它来计算更新元素数。这个方法的实现类似于LongAdder的add（LongAdder在上面有简单介绍），源码在此就不再详细分析了，有兴趣的同学可以研究下。

addCount源码如下：

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)) {//counterCells为null，CAS更新baseCount  
 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);//在线程争用资源时，使用fullAddCount计算更新元素数  
 return;  
 }  
 if (check <= 1)  
 return;  
 s = sumCount();//计算元素总数，用于后面的扩容操作  
 }  
 if (check >= 0) {  
 //检查扩容  
 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);  
 //其他线程在进行扩容操作  
 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();  
 }  
 }  
}

###### 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

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。

首先来看一下另外两个有序Map：TreeMap 和 LinkedHashMap，哥仨特性如下：

TreeMap：红黑树实现的线程不安全的有序哈希表，既然是红黑树，那么优缺点就很明显了，高效读写，时间复杂度O(log n)。

LinkedHashMap：继承自HashMap，线程不安全的有序哈希表，跟HashMap一样，基于散列表实现。数据结构跟HashMap唯一的不同之处是它在内部增加了一个双向链表，这个链表的作用就是用来存储元素顺序。LinkedHashMap 的“有序性”不像TreeMap，LinkedHashMap是基于元素插入时的先后顺序(insertion-order)，也就是说**它的有序性需要我们自己来控制**；而TreeMap 则是通过更灵活的Comparator实现有序性。

ConcurrentSkipListMap：跳表实现的并发有序哈希表。本篇文章的主角。

##### 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

内部属性：

//用来定义最底层(base-level)的头节点。  
private static final Object *BASE\_HEADER* = new Object();

//跳表的最高层headIndex  
private transient volatile HeadIndex<K,V> head;

BASE\_HEADER：初始化时放到最底层头节点的value

Head：跳表最高层的HeadIndex，在跳表层级更新时需要随之更新





**说明**：

首先说一下跳表（Skip List）的特性：

* + - 1. 跳表分为多层，层级越高跳跃性越大，数据越少
      2. 跳表的层级是通过“掷硬币”方式来决定增长的，也就是说在增加元素时，概率增长层数
      3. 第一层包含所有元素
      4. 每层的元素集合必须包含序数最小的元素
      5. 查找数据时，按照从上到下，从左往右的顺序查找
      6. 时间复杂度O(log n)，空间复杂度O(n)

对比链表和跳表：



如果要找到元素25，在链表中我们需要五步才能查到；而在跳表中只需要三步（根据跳表层级分配，有可能更少），如下图（忽略向下路线）：



下面说说Java中ConcurrentSkipListMap的数据结构。

ConcurrentSkipListMap 继承自AbstractMap，实现了ConcurrentNavigableMap 接口，也就是说它是一个有序的可分割的并发安全的哈希表。内部数据存储通过三个内部类实现：

* + 1. Node:存储K-V数据，持有下一个节点的引用，也就是说Node是一个单向链表。
    2. Index: 跳表中的索引节点，包含了右指针(right)，向下的索引(down)和节点node。注意虽然 Node和Index都有向前的指针字段，但是它们类型不同，并且处理方式也不同，所以如果在共享的抽象类中放置这些字段不能很好地实现这一点。
    3. HeadIndex: 表示跳表的表头，继承自Index，内部标识了当前层级level

##### 源码解析

###### 构造函数

//空构造  
public ConcurrentSkipListMap() {  
 this.comparator = null;  
 initialize();  
}  
//指定比较器public ConcurrentSkipListMap(Comparator<? super K> comparator) {  
 this.comparator = comparator;  
 initialize();  
}  
//指定mappublic ConcurrentSkipListMap(Map<? extends K, ? extends V> m) {  
 this.comparator = null;  
 initialize();  
 putAll(m);  
}  
//指定有序mappublic ConcurrentSkipListMap(SortedMap<K, ? extends V> m) {  
 this.comparator = m.comparator();  
 initialize();  
 buildFromSorted(m);  
}  
//初始化内部相关属性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);  
}

###### 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;  
 //第一个自旋，更新或插入新Node  
 outer: for (;;) {  
 //从最底层(base-level)给定key节点的前继节点开始向后查找  
 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，重新查找  
 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，继续向后查找  
 b = n;  
 n = f;  
 continue;  
 }  
 if (c == 0) {//update处理  
 if (onlyIfAbsent || n.casValue(v, value)) {//更新value，返回更新前的值  
 @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，放到b和b.next之间  
 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;  
 }  
 }  
 //更新Index逻辑  
 int rnd = ThreadLocalRandom.*nextSecondarySeed*();  
 //生成随机数为正偶数才会更新层级（通过最高位和最低位不为1验证）  
 if ((rnd & 0x80000001) == 0) { // test highest and lowest bits  
 int level = 1, max;  
 //计算跳表level  
 while (((rnd >>>= 1) & 1) != 0)//判断从低2位开始向左有多少个连续的1  
 ++level;  
 //idx:新添加的index的level层index  
 Index<K,V> idx = null;  
 HeadIndex<K,V> h = head;  
 //构建Index逻辑  
 if (level <= (max = h.level)) {//不需要增加层级  
 for (int i = 1; i <= level; ++i)  
 //从下到上构建,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  
 //构建新层级（旧层级+1）  
 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HeadIndex  
 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;  
 //构建new head  
 for (int j = oldLevel+1; j <= level; ++j)  
 newh = new HeadIndex<K,V>(oldbase, newh, idxs[j], j);  
 //cas替换head节点  
 if (casHead(h, newh)) {  
 h = newh;  
 //idx赋值为之前层级的头节点，并将level赋值为之前的层级  
 idx = idxs[level = oldLevel];  
 break;  
 }  
 }  
 }  
 // find insertion points and splice in  
 //插入Index  
 splice: for (int insertionLevel = level;;) {  
 //获取新跳表head的层级  
 int j = h.level;  
 // 查找需要清除的节点  
 // q:新跳表head; r:q.right; t:新增的Index节点  
 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) {// 需要清除n节点  
 // 删除q的right节点r，并替换为r.right  
 if (!q.unlink(r))  
 break;  
 r = q.right;//重新对r赋值  
 continue;  
 }  
 if (c > 0) {  
 // 继续向右查找一个key小于当前遍历到的节点r  
 q = r;  
 r = r.right;  
 continue;  
 }  
 }  
  
 if (j == insertionLevel) { //新跳表head层级等于旧跳表head层级  
 if (!q.link(r, t))//把新增节点插入q和r之间  
 break; // restart  
 if (t.node.value == null) {//新增节点被删除  
 findNode(key);//清除已删除的节点  
 break splice;  
 }  
 if (--insertionLevel == 0)//到达最底层  
 break splice;  
 }  
  
 //移动到下一层level  
 if (--j >= insertionLevel && j < level)  
 t = t.down;  
 q = q.down;  
 r = q.right;  
 }  
 }  
 }  
 return null;  
}

说明:插入或更新元素都调用了doput方法,大致分两步:

1. 自旋查找索引位置，更新或插入给定节点（Node）元素。在这一步中也会帮助清除已经删除的节点。大概流程如下：
   1. 通过findPredecessor方法（后面会分析）首先找到最底层(base-level)给定key节点的前继节点b，从这个节点开始向后查找合适位置插入（或更新）
   2. 如果在查找过程中遇到已删除节点会调用helpDelete方法帮助清除节点链接，然后重复a步骤
2. 通过“掷硬币”方式决定是否更新跳表层级。在ConcurrentSkipListMap中这个“掷硬币”随机算法实现如下：生成一个随机数rnd，如果rnd为正偶数（通过最高位和最低位都不为1验证）才会更新Index。然后判断这个随机数rnd从低2位开始有多少连续的1，如果这个“连续数”小于或等于旧表层级，则不需要增加跳表层级，只需更新Index；否则的话就需要增加跳表层级（旧表层级+1），并且更新Index和HeadIndex。

###### findPredecessor

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;  
 }  
 }  
}

**说明：**findPredecessor用于查找最底层(base-level)节点链中比给定key小(在“给定节点”左边)的节点，也就是给定key节点的前继节点。如果没找到，那么返回底层链表的头节点。在查找过程中会帮助清除已经被删除的节点。在put、get、remove方法中都有调用findPredecessor，因为在这些方法遍历时，都是从这个节点开始向后遍历。  
从head开始按照从上到下，从左到右的顺序查找，找到right节点的key值大于给定key的节点，然后返回left节点。如下图，如果我们的key值为35，那么从左节点8开始向右遍历，一直到找一个比35大的节点39，然后返回39的left节点29。



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 (;;) {  
 //找到给定key节点的前继节点  
 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);//帮助清除已删除节点  
 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))//找到指定节点，把节点的value置空  
 break;  
 //给节点添加删除标识（next节点改为一个指向自身的节点）  
 //然后把前继节点的next节点CAS修改为next.next节点（彻底解除n节点的链接）  
 if (!n.appendMarker(f) || !b.casNext(n, f))  
 //如果cas失败，清除已删除的节点后重新循环  
 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;//返回对应value  
 }  
 }  
 return null;  
}

**说明:** doRemove是移除节点的主方法，函数逻辑如下：

1. 首先找到需要删除节点的前节点，如果在查找过程中发现有已经删除的节点，就帮助清除节点（解除节点链接）。
2. 在找到需要删除的节点时并不会直接移除它，而是先利用CAS给这个节点添加一个删除标识（next节点改为一个指向自身的节点），然后再利用CAS解除它的链接；如果途中CAS执行失败，则调用findNode方法来清除已经删除的节点。
3. 最后检查head.right如果已经被移除，就调用tryReduceLevel()方法尝试对跳表进行降级操作（跳表只有在层级大于3时才可以降级），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 (;;) {  
 //从最底层(base-level)给定key节点的前继节点开始向后查找  
 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  
 //节点n被删除，帮助清除已删除节点，继续循环outer  
 n.helpDelete(b, f);  
 break;  
 }  
 if (b.value == null || v == n) // b is deleted  
 break;  
 if ((c = *cpr*(cmp, key, n.key)) == 0) {//检查key是否相等  
 @SuppressWarnings("unchecked") V vv = (V)v;  
 return vv;  
 }  
 if (c < 0)  
 break outer;  
 //未找到合适节点，继续向后寻找  
 b = n;  
 n = f;  
 }  
 }  
 return null;  
}

**说明:** doGet 是通过key查找节点value的主方法，源码非常之简单，流程跟doRemove类似，不多赘述。

###### 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

Queue 是“生产者-消费者”模型的重要实现，在实际应用中，众多消息系统（例如RocketMQ、ActiveMQ等）都是基于Queue实现。从本章开始，我们会对JUC中所有的Queue进行逐一讲解。

ArrayBlockingQueue是数组结构的线程安全的有界的阻塞队列。元素操作按照FIFO (first-in-first-out 先入先出)的顺序。线程安全由显式锁ReentrantLock和两个Condition实现。本章我们从源码角度来看一下这个比较简单的Queue的实现。

在队列资源已满，对其的put操作会阻塞；相应地，在队列资源为空时，对其的take操作也会阻塞。

ArrayBlockingQueue 的线程安全由显式锁ReentrantLock和Condition实现，所以它也支持可选的公平策略。公平模式允许线程以FIFO模式操作元素，但是会减少吞吐量。

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，分别控制出队和入队操作。由于使用了显式锁ReentrantLock，所以它也支持可选的公平策略。默认为非公平模式，不保证元素操作顺序；公平模式允许线程以FIFO模式操作元素。公平模式可以降低队列变化性，并且可以避免线程饿死，但是也会减少吞吐量。

//存储元素的数组  
final Object[] items;  
//下一个要获取的元素索引  
int takeIndex;  
//下一个要插入的元素索引  
int putIndex;  
//元素数量  
int count;  
final ReentrantLock lock;  
//等待获取条件  
private final Condition notEmpty;  
//等待插入条件  
private final Condition notFull;  
//共享的活动迭代器,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:**

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是单向链表结构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,** **元素操作按照FIFO (first-in-first-out 先入先出)的顺序。 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: LinkedBlockingdDeque  
单生产,多消费: ConcurrentLinkedQueue  
多生产,多消费: ConcurrentLinkedDeque**

**ConcurrentLinkedQueue 的非阻塞算法简述**

**本文接下来将在分析 ConcurrentLinkedQueue 源代码实现的过程中，穿插讲解非阻塞算法的具体实现。为了便于读者理解本文，首先让我们对它的实现机制做个全局性的简述。ConcurrentLinkedQueue 的非阻塞算法实现可概括为下面 5 点：**

1. **使用 CAS 原子指令来处理对数据的并发访问，这是非阻塞算法得以实现的基础。**
2. **head/tail都允许滞后，也就是说它们并非总是指向队列的头/尾节点，这是因为并不是每次操作队列都更新head/tail，和LinkedTransferQueue一样，使用了一个“松弛阀值”，** **当前指针距离head/tail节点大于这个松弛阀值时才会更新head/tail，这也是一种优化方式。**
3. **由于队列有时会处于不一致状态。为此，ConcurrentLinkedQueue 使用三个不变式来维护非阻塞算法的正确性（后面会详细说明）。**
4. **以批处理方式来更新 head/tail，从整体上减少入队 / 出队操作的开销。**
5. **为了有利于垃圾收集，队列使用特有的 head 更新机制；为了确保从已删除节点向后遍历，可到达所有的非删除节点，队列使用了特有的向后推进策略。**

##### 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**

* 当入队插入新节点之后，队列中有一个 next 域为 null （最后一个）的节点。
* 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**

不变式：

* 所有live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。

可变性：

* 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**

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。

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。

**内存一致性：**对ConcurrentLinkedQueue的插入操作先行发生于(happen-before)访问或移除操作。

##### 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 跳两个节点以上时才修改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节点指向自身，说明p是一个自链节点，此时需要重新获取tail节点，  
 // 如果tail节点被其他线程修改，此时需要从head开始向后遍历，因为  
 // 从head可以到达所有的live节点。  
 p = (t != (t = tail)) ? t : head;  
 else  
 // Check for tail updates after two hops.  
 //继续向后查找，如果tail节点变化，重新获取tail  
 p = (p != t && t != (t = tail)) ? t : q;  
 }  
}

**说明：** ConcurrentLinkedQueue 提供了两个添加元素的方法（因为实现了 Queue 和 Collection 接口）：一个是 Queue 的offer方法， 另外一个是 Collection 的add方法，add也是调用offer来实现。新增元素时，把元素放到链表尾部，由于队列是无界的，所以插入时不会返回false或者抛出IllegalStateException。函数逻辑很简单：从tail节点向后自旋查找 next 为 null 的节点，也就是最后一个节点（因为 tail 节点并不是每次都更新，所以我们取到的 tail 节点有可能并不是最后一个节点），然后CAS插入新增节点。  
上面我们提到过：并不是每次操作都会更新 head/tail 节点，而是使用了一个“松弛阀值”，这个“松弛阀值”就体现在上面源码中if (p != t)这一行，p初始是等于tail的，如果向后查找了一次以上才找到最后一个节点，再加上新增的节点，说明tail已经跳跃了两个（或以上）节点，此时才会CAS更新tail，这里也算是一种编程技巧。

**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 (;;) {  
 //从head节点向后查找第一个live节点  
 for (Node<E> h = head, p = h, q;;) {  
 E item = p.item;  
  
 if (item != null && p.casItem(item, null)) {//找到第一个节点，cas修改节点item为null  
 // Successful CAS is the linearization point  
 // for item to be removed from this queue.  
 if (p != h) // 跳两个节点以上时才修改head  
 //cas修改head节点  
 updateHead(h, ((q = p.next) != null) ? q : p);  
 return item;  
 }  
 else if ((q = p.next) == null) {//队列已空，返回null  
 updateHead(h, p);//cas修改head节点为p  
 return null;  
 }  
 else if (p == q)//p为自链接节点，重新获取head循环  
 continue restartFromHead;//跳转到restartFromHead重新循环  
 else  
 p = q;//向后查找  
 }  
 }  
}

*/\*\*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是双向链表结构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  
 \*/*//头节点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。  
 \*/*//尾节点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);  
 }  
 }  
}  
//移除节点时更新链表属性的阀值private static final int *HOPS* = 2;

**说明：**

和ConcurrentLinkedQueue一样，内部也只维护了head和tail节点，对head/tail节点也使用了“不变性”和“可变性”约束，跟ConcurrentLinkedQueue有部分差异，我们来看一下：

head/tail的不变性：

* 1. 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
  2. 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
  3. 所有live节点（item不为null的节点），都能从第一个节点通过调用 succ() 方法遍历可达。
  4. 所有live节点（item不为null的节点），都能从最后一个节点通过调用 pred() 方法遍历可达。
  5. head/tail 不能为 null。
  6. **head** 节点的 next 域不能引用到自身。
  7. head/tail 不会是GC-unlinked节点（但它可能是unlink节点）

head/tail的可变性：

* 1. head/tail 节点的 item 域可能为 null，也可能不为 null。
  2. head/tail 节点可能从first/last/tail/head 节点访问时不可达
  3. **tail** 节点的 next 域可以引用到自身。

1. CLD中也对head/tail的更新也使用了“松弛阀值”的概念（在**ConcurrentLinkedQueue**一篇中已经分析），除此之外，CLD设定了一个“跳跃阀值”-HOPS，在执行出队操作时，跳跃节点数大于2或者操作的节点不是first/last节点时才会更新链表（后面源码中详细分析）。

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；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。  
 2. first node & last node：首节点(first node)总会有一个空的prev引用，终止任何从live节点开始的prev引用链；同样的最后一个节点(last node)是next的终止节点。first和last节点的item可以为null。并且first和last节点总是相互可达的。  
 3. 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)  
 4. self-node：自链接节点，prev或next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。  
 5. head/tail节点：head/tail也可能不是first/last节点。从head节点通过prev引用总是可以找到first节点，从tail节点通过next引用总是可以找到last节点。允许head和tail引用已删除的节点，这些节点没有链接，因此可能无法从live节点访问到。  
 6. 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：通过CAS修改节点item为null来完成，表示当前节点可以被解除链接(unlinking)  
 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访问到。节点unlinking仅仅是一个优化方式。在任何时候，从first通过next找到的live节点和从last通过prev找到的节点总是相等的。但是，在节点被逻辑删除时上述结论不成立，这些被逻辑删除的节点也可能从一端是可达的。  
 gc-unlinking： GC未链接使已经被解除链接的节点从active不可达，使GC更容易回收被删除的节点。gc-unlinking节点从head或tail访问不可达。这一步是为了使数据结构保持GC健壮性(gc-robust)，防止保守垃圾收集器对这些边界空间的使用。对保守的GC来说，使数据结构保持GC健壮性会消除内存无限滞留的问题，同时也提高了分代收机器的性能（只需要了解，后面会专门开篇讲解）。

##### 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)、addFirst(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.  
 //如果head被修改，返回head重新查找  
 p = (h != (h = head)) ? h : q;  
 else if (p.next == p) // 自链接节点，重新查找  
 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  
 }  
 }  
}

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.  
 //如果tail被修改，返回tail重新查找  
 p = (t != (t = tail)) ? t : q;  
 else if (p.prev == p) // 自链接节点，重新查找  
 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被修改则返回新的head重新查找，否则继续向前(prev)查找  
 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.  
 //找到的节点不是head节点，CAS修改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) {//操作节点为first节点  
 unlinkFirst(x, next);  
 } else if (next == null) {//操作节点为last节点  
 unlinkLast(x, prev);  
 } else {// common case  
 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;//自链接节点  
 activePred = p;  
 isFirst = true;  
 break;  
 }  
 else if (p == q)//自链接节点  
 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)//自链接节点  
 return;  
 else  
 p = q;  
 }  
  
 // *TODO: better HOP heuristics* //无节点跳跃并且操作节点有first或last节点时不更新链表  
 if (hops < *HOPS* // always squeeze out interior deleted nodes  
 && (isFirst | isLast)) //d  
 return;  
  
 // Squeeze out deleted nodes between activePred and  
 // activeSucc, including x.  
 //连接两个活动节点  
 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，o：已删除节点(item为null)  
 for (Node<E> o = null, p = next, q;;) {  
 if (p.item != null || (q = p.next) == null) {  
 //跳过已删除节点，CAS替换first的next节点为一个active节点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unlinking节点next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)//自链接节点  
 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：**

我们的策略是找到x节点的活跃（active）前继和后继节点。尝试修整它们之间的链接，让它们指向对方，留下一个从活跃(active)节点不可达的x节点。如果成功执行，或者x节点没有live的前继/后继节点，再尝试gc解除链接(gc-unlink)，在设置x节点的prev/next指向它们自己或TERMINATOR之前，需要检查x的前继和后继节点的状态未被改变，并保证x节点从head/tail不可达。

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);  
}

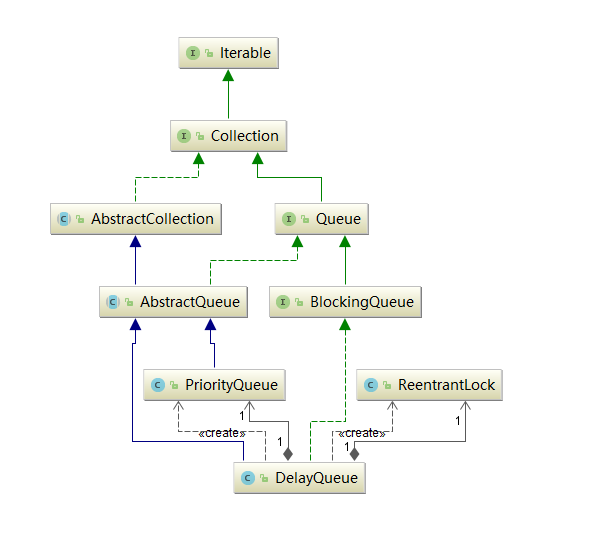
f. 判断操作的活动节点是否有first或last节点，有则更新head和tail，最后通过UNSAFE的putOrderedObject更新出列节点的prev和next分别为PREV\_TERMINATOR和NEXT\_TERMINATOR。

#### 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 {  
 //调用priorityQueue的offer出列  
 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()

*/\*\*获取并移除头元素，一直等待元素可用\*/*public E take() throws InterruptedException {  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 //自旋  
 for (;;) {  
 E first = q.peek();  
 if (first == null)//首节点为空，等待  
 available.await();  
 else {  
 long delay = first.getDelay(*NANOSECONDS*);//延时时间  
 if (delay <= 0)  
 return q.poll();  
 //释放first的引用，避免内存泄漏  
 first = null; // don't retain ref while waiting  
 if (leader != null)//leader不为空，证明有其他线程已经获取到leader，加入条件队列等到延时结束  
 available.await();  
 else {  
 Thread thisThread = Thread.*currentThread*();  
 leader = thisThread;//leader指向当前线程  
 try {  
 available.awaitNanos(delay);  
 } finally {  
 if (leader == thisThread)//检查是否被其他线程改变，没有就重置，再次循环  
 leader = null;  
 }  
 }  
 }  
 }  
 } finally {  
 if (leader == null && q.peek() != null)//leader为空并且队列不空，说明没有其他线程在等待，那就通知条件队列  
 available.signal();  
 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）是一个基于平衡二叉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.  
 \*  
 \* 基于一个平衡的二叉堆实现  
 \*/* 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，保持二叉堆的性质不变\*/*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，保持二叉堆的性质不变\*/*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叉堆的操作，这里理解二叉堆的结构和原理，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) {  
   
 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已经被匹配，修改head继续循环  
 casHead(h, h.next); // pop cancelled node  
 else  
 return null;  
 } else if (casHead(h, s = *snode*(s, e, h, mode))) {//构建新的节点s，放到栈顶  
 //等待s节点被匹配，返回s.match节点m  
 SNode m = awaitFulfill(s, timed, nanos);  
 //s.match==s（等待被取消）  
 if (m == s) { // wait was cancelled  
 clean(s);//清除s节点  
 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已经被匹配，修改head继续循环  
 casHead(h, h.next); // pop and retry  
  
 //构建新节点，放到栈顶  
 else if (casHead(h, s=*snode*(s, e, h, *FULFILLING*|mode))) {  
 for (;;) { // loop until matched or waiters disappear  
 //cas成功后s的match节点就是s.next，即m  
 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，唤醒m节点的线程  
 casHead(s, mn); //弹出匹配成功的两个节点，替换head pop both s and m  
 return (E) ((mode == *REQUEST*) ? m.item : s.item);  
 } else // lost match  
 s.casNext(m, mn); //匹配失败，删除m节点，重新循环 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 {//帮助头节点匹配  
 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：**基本算法是循环尝试三种行为之一：

* 1. 如果栈为空或者已经包含了一个相同的mode，就把当前节点压进栈顶等待匹配（通过awaitFulfill方法）。返回匹配节点的item，如果节点取消等待就返回null。
  2. 如果栈中有一个互补的等待节点，尝试与此节点进行匹配，然后从栈中弹出这两个节点，并返回匹配节点的数据。
  3. 如果栈顶节点已经持有另外一个数据节点，说明栈顶节点正在匹配，则帮助此节点进行匹配操作，然后继续循环。
     + - 1. awaitFulfill (SNode s, boolean timed, long nanos)

SNode awaitFulfill(SNode s, boolean timed, long nanos) {  
 //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)//已经匹配到，返回匹配节点  
 return m;  
 if (timed) {  
 //超时处理  
 nanos = deadline - System.*nanoTime*();  
 if (nanos <= 0L) {  
 s.tryCancel();//超时，取消s节点的匹配，match指向自身  
 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：**如果当前操作是一个不计时操作，或者是一个还未到超时时间的操作，就构建新的节点压入栈顶。然后调用此方法自旋/阻塞等待给定节点s被匹配。 当调用此方法时，所传参数节点s一定是在栈顶，节点真正阻塞前会先自旋，以防生产者和消费者到达的时间点非常接近时也被park。当节点/线程需要阻塞时，首先设置waiter字段为当前线程，然后在真正阻塞之前重新检查一下waiter的状态，因为在线程竞争中，需要确认waiter没有被其他线程占用。从主循环返回的检查顺序可以反映出中断优先于正常返回。除了不计时操作(poll/offer)不会检查中断，而是直接在transfer方法中入栈等待匹配。

* + - * 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  
  
 SNode past = s.next;  
 if (past != null && past.isCancelled())  
 past = past.next;  
  
 // Absorb cancelled nodes at head  
 //找到有效head  
 SNode p;  
 while ((p = head) != null && p != past && p.isCancelled())  
 casHead(p, p.next);  
  
 // Unsplice embedded nodes  
 //移除head到past中已取消节点的链接  
 while (p != null && p != past) {  
 SNode n = p.next;  
 if (n != null && n.isCancelled())  
 p.casNext(n, n.next);  
 else  
 p = n;  
 }  
}

**说明：**在最坏的情况下可能需要遍历整个栈来解除给定节点s的链接（给定节点在栈底）。在并发情况下，如果有其他线程已经移除给定节点s，当前线程可能无法看到，但是我们可以使用这样一种算法：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，然后依次解除从head到past中已取消节点的链接，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得的。

* + - * 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，匹配成功唤醒节点的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) {  
   
 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，更新尾节点 lagging tail  
 advanceTail(t, tn);  
 continue;  
 }  
 if (timed && nanos <= 0) // can't wait  
 return null;  
 //为当前操作构造新节点，并放到队尾  
 if (s == null)  
 s = new QNode(e, isData);  
 if (!t.casNext(null, s)) // failed to link in  
 continue;  
  
 //推进tail  
 advanceTail(t, s); // swing tail and wait  
 //等待匹配，并返回匹配节点的item，如果取消等待则返回该节点s  
 Object x = awaitFulfill(s, e, timed, nanos);  
 if (x == s) { // wait was cancelled  
 clean(t, s); //等待被取消，清除s节点  
 return null;  
 }  
  
 if (!s.isOffList()) { // s节点尚未出列 not already unlinked  
 advanceHead(t, s); // unlink if head  
 if (x != null) // and forget fields  
 s.item = s;//item指向自身  
 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)) { // 匹配，CAS修改item为给定元素e lost CAS  
 advanceHead(h, m); // 推进head，继续向后查找 dequeue and retry  
 continue;  
 }  
  
 advanceHead(h, m); //匹配成功，head出列 successfully fulfilled  
 LockSupport.*unpark*(m.waiter); //唤醒被匹配节点m的线程  
 return (x != null) ? (E)x : e;  
 }  
 }  
}

**说明：**循环尝试两个动作中的其中一个：

1. 若队列为空或者队列中的尾节点和自己的模式相同, 则把当前节点添加到队列尾,直到节点取消匹配或匹配成功，然后返回匹配节点的 item。  
2. 如果队列中有一个互补的等待节点，尝试通过CAS修改等待节点的item字段为给定元素e，然后使等待节点出列，并返回匹配节点的 item。

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

* + - * 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  
   
 while (pred.next == s) { // Return early if already unlinked  
 QNode h = head;  
 QNode hn = h.next; // Absorb cancelled first node as head  
 //找到有效head节点  
 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;  
 //找到tail节点  
 if (tn != null) {  
 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解除s的链接  
 return;  
 }  
 //s是队列尾节点，此时无法删除s，只能去清除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。

向后推进head节点

向后推进tail

**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) { // 找到有效节点，进入匹配  
 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)) { // 匹配成功，cas修改为指定元素 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)) {//更新head为匹配节点的next节点  
 h.forgetNext();//旧head节点指向自身等待回收  
 break;  
 } // head被其他线程修改，重新获取 head advance and retry  
 if ((h = head) == null ||  
 (q = h.next) == null || !q.isMatched())//如果head的next节点未被匹配，跳出循环，不更新head，也就是松弛度<2  
 break; // unless slack < 2  
 }  
 LockSupport.*unpark*(p.waiter);//唤醒在节点上等待的线程  
 return LinkedTransferQueue.<E>*cast*(item);  
 }  
 }  
 //匹配失败，继续向后查找节点  
 Node n = p.next;  
 p = (p != n) ? n : (h = head); // Use head if p offlist  
 }  
  
 //未找到匹配节点，把当前节点加入到队列尾  
 if (how != *NOW*) { // No matches available  
 if (s == null)  
 s = new Node(e, haveData);  
 //将新节点s添加到队列尾并返回s的前继节点  
 Node pred = tryAppend(s, haveData);  
 if (pred == null)  
 continue retry; //与其他不同模式线程竞争失败重新循环 lost race vs opposite mode  
 if (how != *ASYNC*)//同步操作，等待匹配  
 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;  
 }  
 }  
 }  
 }  
 }  
}

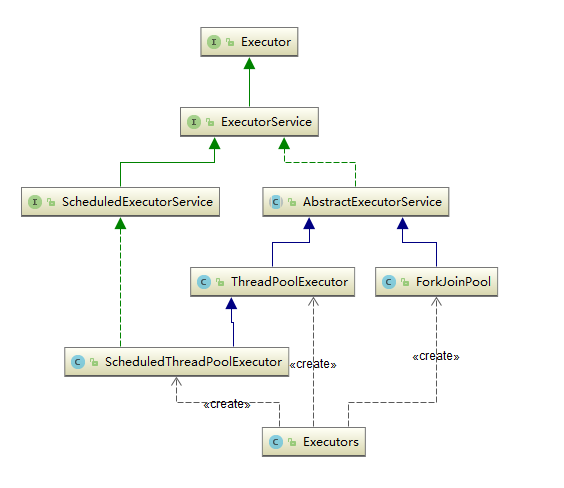
*/\*\*  
 \* Unlinks matched (typically cancelled) nodes encountered in a  
 \* traversal from head.  
 \* 解除(通常是取消)从头部遍历时遇到的已经被匹配的节点的链接  
 \*/*private void sweep() {  
 for (Node p = head, s, n; p != null && (s = p.next) != null; ) {  
 if (!s.isMatched())  
 // Unmatched nodes are never self-linked  
 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);

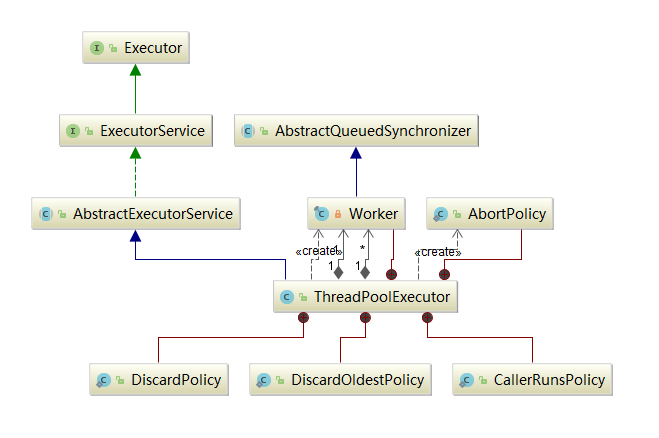
#### ThreadPoolExecutor

##### 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



*/\*\*当核心线程数已满，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;

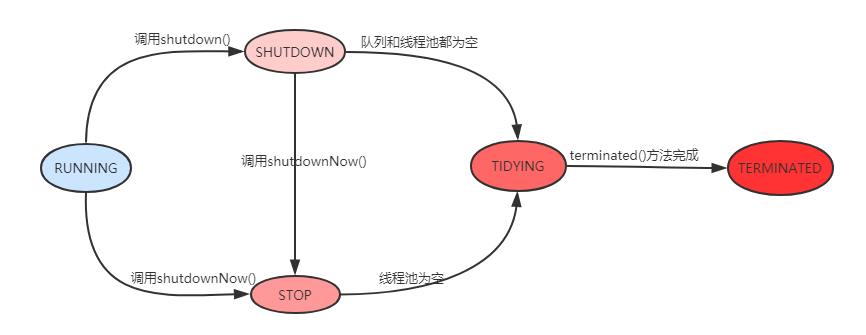
*/\*\*最大线程池大小  
 \* 如果当前等待队列任务已满，继续提交的任务将继续创建新的线程执行，这个线程数最大为maximumPoolSize  
 \* \*/*private volatile int maximumPoolSize;

*/\*\*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，当池状态为TIDYING时将会运行terminated()方法  
**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()

*/\*\*  
 \* 启动一个核心线程等待新任务，这将覆盖只在执行新任务时启动核心线程  
 \* 的默认策略。如果所有核心线程都已经开启，返回false  
 \*/*public boolean prestartCoreThread()

##### 源码解析

###### 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。

###### 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。

###### addWorker (Runnable firstTask, boolean core)

*/\*\*  
 \* 尝试添加新的工作线程  
 \* 检查当前池状态和给定界限（核心线程或最大线程）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);//获取runState  
  
 // Check if queue empty only if necessary.  
 //检查池状态  
 if (rs >= *SHUTDOWN* &&  
 ! (rs == *SHUTDOWN* &&  
 firstTask == null &&  
 ! workQueue.isEmpty()))  
 return false;  
  
 for (;;) {//检查工作线程数是否饱和  
 int wc = *workerCountOf*(c);//获取工作线程数  
 if (wc >= *CAPACITY* ||  
 wc >= (core ? corePoolSize : maximumPoolSize))  
 return false;  
 if (compareAndIncrementWorkerCount(c))//可以添加新的线程，递增ctl，跳出retry自旋  
 break retry;  
 //更新ctl失败，重读ctl继续循环检查  
 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();//加锁，准备添加新的工作线程  
 try {  
 // Recheck while holding lock.  
 // Back out on ThreadFactory failure or if  
 // shut down before lock acquired.  
 int rs = *runStateOf*(ctl.get());  
 //重新检查runState  
 if (rs < *SHUTDOWN* ||  
 (rs == *SHUTDOWN* && firstTask == null)) {  
 if (t.isAlive()) // precheck that t is startable  
 throw new IllegalThreadStateException();//线程已经被启动，抛出异常  
 workers.add(w);//添加工作线程  
 int s = workers.size();  
 if (s > largestPoolSize)  
 largestPoolSize = s;//更新最大池容量  
 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，回滚操作  
 }  
 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

###### execute(Runnable command)

*/\*\*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.  
 \* 如果运行线程数小于核心线程数，尝试开启一个新的线程执行给定任务。  
 \*  
 \* 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.  
 \* 如果任务可入列等待，仍然需要检查是否需要添加线程，因为从上次检查到现在可能有线程死掉，  
 \* 或者在进入此方法后线程池被关闭。所以我们要重新检查池状态，如果有必要，就回滚入队操作。  
 \*  
 \* 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.  
 \* 如果任务不能入队，我们就尝试添加一个新的线程。如果添加失败，就意味着池被关闭或已经饱和，  
 \* 这种情况就需要根据拒绝策略来处理任务。  
 \*/  
 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。

###### shutdown()

*/\*\*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() {  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
 checkShutdownAccess();//检查关闭权限  
 advanceRunState(*SHUTDOWN*);//修改运行状态runState  
 interruptIdleWorkers();//中断空闲工作线程  
 //为ScheduledThreadPoolExecutor提供的关闭钩子  
 onShutdown(); // hook for ScheduledThreadPoolExecutor  
 } finally {  
 mainLock.unlock();  
 }  
 tryTerminate();//销毁线程池  
}

说明：关闭线程池方法，此方法不会等待已提交任务执行完毕(可使用awaitTermination来实现这一需求)。分以下几步进行：  
1.检查关闭权限  
2.修改运行状态(runState)  
3.中断空闲工作线程  
4.执行关闭钩子，这里的onShutdown是扩展方法，在之后ScheduledThreadPoolExecutor的分析中会介绍这一方法。  
5.调用tryTerminate方法尝试销毁线程池

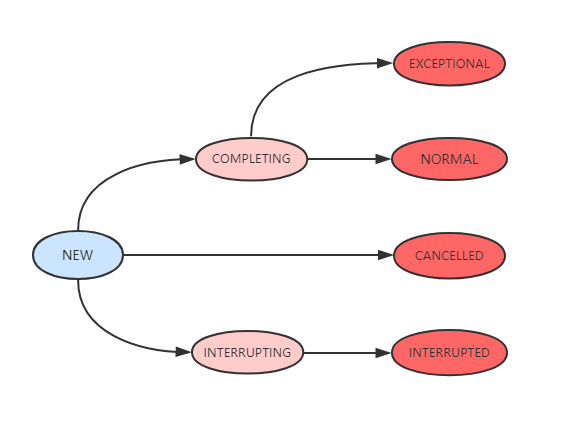
##### 小结

**关键点：线程池状态、corePoolSize（核心线程数）、maximumPoolSize（最大线程数）**

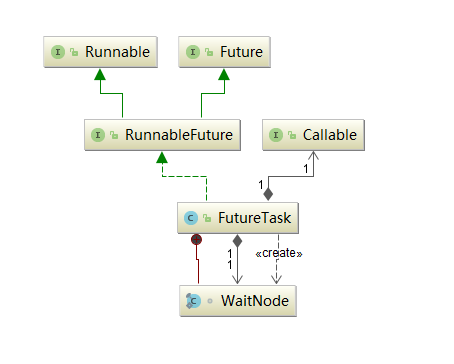
#### FutureTask

##### 概述

FutureTask，继承自Runnale和Future，是一个可取消的异步运算。此类为Future提供基础实现，如start和cancel，查询计算是否完成，获取执行结果。如果计算尚未完成，获取计算结果时将会阻塞。一旦执行结束，计算就不能被重启或取消（除非使用runAndReset执行计算）。  
FutureTask常用来封装Callable和Runnable，也可以作为一个任务提交到线程池中执行。  
除了作为一个独立的类之外，此类也提供了一些功能性函数供我们创建自定义task类使用。



##### 数据结构和核心参数



*/\*\*  
 \* The run state of this task, initially NEW. The run state  
 \* transitions to a terminal state only in methods set,  
 \* setException, and cancel. During completion, state may take on  
 \* transient values of COMPLETING (while outcome is being set) or  
 \* INTERRUPTING (only while interrupting the runner to satisfy a  
 \* cancel(true)). Transitions from these intermediate to final  
 \* states use cheaper ordered/lazy writes because values are unique  
 \* and cannot be further modified.  
 \*  
 \* Possible state transitions:  
 \* NEW -> COMPLETING -> NORMAL  
 \* NEW -> COMPLETING -> EXCEPTIONAL  
 \* NEW -> CANCELLED  
 \* NEW -> INTERRUPTING -> INTERRUPTED  
 \*/*

//state>COMPLETING处于结束状态

private volatile int state;  
private static final int *NEW* = 0;//新建  
private static final int *COMPLETING* = 1;//完成  
private static final int *NORMAL* = 2;//正常  
private static final int *EXCEPTIONAL* = 3;//异常  
private static final int *CANCELLED* = 4;//取消  
private static final int *INTERRUPTING* = 5;//正在中断  
private static final int *INTERRUPTED* = 6;//已中断  
  
*/\*\* The underlying callable; nulled out after running \*/*//底层调用，运行完毕后置空  
private Callable<V> callable;  
*/\*\* The result to return or exception to throw from get() \*/*//从get()中抛出或返回的异常  
private Object outcome; // non-volatile, protected by state reads/writes  
*/\*\* The thread running the callable; CASed during run() \*/*//运行callable线程，在run()期间保持原子性  
private volatile Thread runner;  
*/\*\* Treiber stack of waiting threads \*/*//使用Treiber栈保存等待线程  
private volatile WaitNode waiters;

##### 函数列表

public FutureTask(Callable<V> callable) {  
 if (callable == null)  
 throw new NullPointerException();  
 this.callable = callable;  
 this.state = *NEW*; // ensure visibility of callable  
}

public FutureTask(Runnable runnable, V result) {  
 this.callable = Executors.*callable*(runnable, result);  
 this.state = *NEW*; // ensure visibility of callable  
}

public boolean isCancelled() {  
 return state >= *CANCELLED*;  
}  
  
public boolean isDone() {  
 return state != *NEW*;  
}

public boolean cancel(boolean mayInterruptIfRunning)

//获取执行结果  
public V get()

public V get(long timeout, TimeUnit unit)

##### 源码分析

###### submit()

这里先从线程池的submit方法开始分析，一步一步进入FutureTask。submit方法默认实现在AbstractExecutorService，源码如下：

public Future<?> submit(Runnable task) {  
 if (task == null) throw new NullPointerException();  
 RunnableFuture<Void> ftask = newTaskFor(task, null);  
 execute(ftask);  
 return ftask;  
}

public <T> Future<T> submit(Runnable task, T result) {  
 if (task == null) throw new NullPointerException();  
 RunnableFuture<T> ftask = newTaskFor(task, result);  
 execute(ftask);  
 return ftask;  
}

public <T> Future<T> submit(Callable<T> task) {  
 if (task == null) throw new NullPointerException();  
 RunnableFuture<T> ftask = newTaskFor(task);  
 execute(ftask);  
 return ftask;  
}

protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) {  
 return new FutureTask<T>(runnable, value);  
}

public FutureTask(Runnable runnable, V result) {  
 this.callable = Executors.*callable*(runnable, result);  
 this.state = *NEW*; // ensure visibility of callable  
}

**说明：**方法比较简单，首先调用newTaskFor方法构造FutureTask，然后调用execute把任务放进线程池中，返回FutureTask。

###### 5.3.4.2 FutureTask.run()

public void run() {  
 //新建任务，CAS替换runner为当前线程  
 if (state != *NEW* ||  
 !*UNSAFE*.compareAndSwapObject(this, *runnerOffset*,  
 null, Thread.*currentThread*()))  
 return;  
 try {  
 Callable<V> c = callable;  
 if (c != null && state == *NEW*) {  
 V result;  
 boolean ran;  
 try {  
 result = c.call();  
 ran = true;  
 } catch (Throwable ex) {  
 result = null;  
 ran = false;  
 setException(ex);  
 }  
 if (ran)  
 set(result);//设置执行结果  
 }  
 } finally {  
 // runner must be non-null until state is settled to  
 // prevent concurrent calls to run()  
 runner = null;  
 // state must be re-read after nulling runner to prevent  
 // leaked interrupts  
 int s = state;  
 if (s >= *INTERRUPTING*)  
 handlePossibleCancellationInterrupt(s);//处理中断逻辑  
 }  
}

**说明：**

1. 运行任务，如果任务状态为NEW状态，则利用CAS修改为当前线程。执行完毕调用set方法设置执行结果。set源码如下：

set

//设置callable返回结果  
protected void set(V v) {  
 if (*UNSAFE*.compareAndSwapInt(this, *stateOffset*, *NEW*, *COMPLETING*)) {  
 outcome = v;  
 *UNSAFE*.putOrderedInt(this, *stateOffset*, *NORMAL*); // final state  
 finishCompletion();//执行完毕，唤醒等待线程  
 }  
}

1. 结果设置完毕后，调用finishCompletion()方法唤醒等待线程，源码如下：

finishCompletion

//移除并唤醒所有等待线程  
private void finishCompletion() {  
 // assert state > COMPLETING;  
 for (WaitNode q; (q = waiters) != null;) {  
 if (*UNSAFE*.compareAndSwapObject(this, *waitersOffset*, q, null)) {//移除等待线程  
 for (;;) {//自旋遍历等待线程  
 Thread t = q.thread;  
 if (t != null) {  
 q.thread = null;  
 LockSupport.*unpark*(t);//唤醒等待线程  
 }  
 WaitNode next = q.next;  
 if (next == null)  
 break;  
 q.next = null; // unlink to help gc  
 q = next;  
 }  
 break;  
 }  
 }  
 //任务完成后调用函数，自定义扩展  
 done();  
  
 callable = null; // to reduce footprint  
}

1. 如果在run期间调用了cancel(true)，此时需要调用handlePossibleCancellationInterrupt方法对线程中断做出处理，源码如下：

handlePossibleCancellationInterrupt

//确保调用cancel(true)时的任何中断只在run或runAndReset传递到task中  
private void handlePossibleCancellationInterrupt(int s) {  
 // It is possible for our interrupter to stall before getting a  
 // chance to interrupt us. Let's spin-wait patiently.  
 if (s == *INTERRUPTING*)  
 while (state == *INTERRUPTING*)  
 Thread.*yield*(); // wait out pending interrupt  
  
 // assert state == INTERRUPTED;  
  
 // We want to clear any interrupt we may have received from  
 // cancel(true). However, it is permissible to use interrupts  
 // as an independent mechanism for a task to communicate with  
 // its caller, and there is no way to clear only the  
 // cancellation interrupt.  
 //  
 // Thread.interrupted();  
}

###### runAndReset()

*/\*\*  
 \* 执行计算但不设置计算结果，然后重置future的state  
 \* 此方法是为执行不止一次的任务而设计的  
 \*/*protected boolean runAndReset() {  
 if (state != *NEW* ||  
 !*UNSAFE*.compareAndSwapObject(this, *runnerOffset*,  
 null, Thread.*currentThread*()))  
 return false;  
 boolean ran = false;  
 int s = state;  
 try {  
 Callable<V> c = callable;  
 if (c != null && s == *NEW*) {  
 try {  
 c.call(); // don't set result  
 ran = true;  
 } catch (Throwable ex) {  
 setException(ex);  
 }  
 }  
 } finally {  
 // runner must be non-null until state is settled to  
 // prevent concurrent calls to run()  
 runner = null;  
 // state must be re-read after nulling runner to prevent  
 // leaked interrupts  
 s = state;  
 if (s >= *INTERRUPTING*)  
 handlePossibleCancellationInterrupt(s);  
 }  
 return ran && s == *NEW*;  
}

**说明**：FutureTask的另外一个run方法，此方法不返回执行结果，而且在任务执行完之后会重置stat的状态为NEW，使任务可以多次执行。

###### get()

//获取执行结果  
public V get() throws InterruptedException, ExecutionException {  
 int s = state;  
 if (s <= *COMPLETING*)  
 s = awaitDone(false, 0L);  
 return report(s);  
}

1. 首先判断任务是否执行完毕，如果未完成，调用awaitDone方法等待任务完成，源码如下：

awaitDone(Boolean timed,long nanos)

//等待任务完成或中止，直到线程中断或超时  
private int awaitDone(boolean timed, long nanos)  
 throws InterruptedException {  
 final long deadline = timed ? System.*nanoTime*() + nanos : 0L;  
 WaitNode q = null;  
 boolean queued = false;  
 for (;;) {//自旋  
 if (Thread.*interrupted*()) {//获取并清除中断状态  
 removeWaiter(q);//移除等待WaitNode  
 throw new InterruptedException();  
 }  
  
 int s = state;  
 if (s > *COMPLETING*) {  
 if (q != null)  
 q.thread = null;//置空等待节点的线程  
 return s;  
 }  
 else if (s == *COMPLETING*) // cannot time out yet  
 Thread.*yield*();  
 else if (q == null)  
 q = new WaitNode();  
 else if (!queued)  
 //CAS修改waiter  
 queued = *UNSAFE*.compareAndSwapObject(this, *waitersOffset*,  
 q.next = waiters, q);  
 else if (timed) {  
 nanos = deadline - System.*nanoTime*();  
 if (nanos <= 0L) {  
 removeWaiter(q);//超时，移除等待节点  
 return state;  
 }  
 LockSupport.*parkNanos*(this, nanos);//阻塞当前线程  
 }  
 else  
 LockSupport.*park*(this);//阻塞当前线程  
 }  
}

awaitDone方法等待任务完成或终止，直到线程中断或超时。

* 1. 如果线程被中断，清除中断状态，调用removeWaiter移除等待的waitNode，然后抛出InterruptedException。removeWaiter源码如下：

//移除超时或已中断的等待节点  
private void removeWaiter(WaitNode node) {  
 if (node != null) {  
 node.thread = null;//首先置空线程  
 retry:  
 for (;;) { // restart on removeWaiter race  
 //依次遍历查找  
 for (WaitNode pred = null, q = waiters, s; q != null; q = s) {  
 s = q.next;  
 if (q.thread != null)  
 pred = q;  
 else if (pred != null) {  
 pred.next = s;  
 if (pred.thread == null) // check for race  
 continue retry;  
 }  
 else if (!*UNSAFE*.compareAndSwapObject(this, *waitersOffset*,  
 q, s))//cas替换  
 continue retry;  
 }  
 break;  
 }  
 }  
}

* 1. 如果当前状态为结束状态(state>COMPLETING),则根据需要置空等待节点的线程，并返回Future状态
  2. 如果当前状态为正在完成(COMPLETING)，说明此时Futre还不能做出超时动作，线程让步。
  3. 如果state为NEW，先新建一个WaitNode，然后CAS修改当前waiter
  4. 如果超时，移除等待节点，返回state，如果设置了超时时间但是尚未超时，则park阻塞当前线程
  5. 其他情况直接阻塞当前线程

1. 如果任务已经是完成状态(state>COMPLETING),则调用report获取执行结果，源码如下：

report(int state)

//返回执行结果或抛出异常  
private V report(int s) throws ExecutionException {  
 Object x = outcome;  
 if (s == *NORMAL*)  
 return (V)x;  
 if (s >= *CANCELLED*)  
 throw new CancellationException();  
 throw new ExecutionException((Throwable)x);  
}

###### cancel(boolean mayInterruptIfRunning)

public boolean cancel(boolean mayInterruptIfRunning) {  
 //如果当前Future状态为NEW，根据参数修改Future状态为INTERRUPTING或CANCELLED  
 if (!(state == *NEW* &&  
 *UNSAFE*.compareAndSwapInt(this, *stateOffset*, *NEW*,  
 mayInterruptIfRunning ? *INTERRUPTING* : *CANCELLED*)))  
 return false;  
 try { // in case call to interrupt throws exception  
 if (mayInterruptIfRunning) {//可以在运行时中断  
 try {  
 Thread t = runner;  
 if (t != null)  
 t.interrupt();  
 } finally { // final state  
 *UNSAFE*.putOrderedInt(this, *stateOffset*, *INTERRUPTED*);  
 }  
 }  
 } finally {  
 finishCompletion();//移除并唤醒所有等待线程  
 }  
 return true;  
}

**说明**：取消当前任务，如果mayInterruptIfRunning为true，修改为INTERRUPTING，否则CANCELLED。

###### shutdown()

public void shutdown() {  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
 checkShutdownAccess();//检查关闭权限  
 advanceRunState(*SHUTDOWN*);//修改运行状态runState  
 interruptIdleWorkers();//中断空闲工作线程  
 //为ScheduledThreadPoolExecutor提供的关闭钩子  
 onShutdown(); // hook for ScheduledThreadPoolExecutor  
 } finally {  
 mainLock.unlock();  
 }  
 tryTerminate();//销毁线程池  
}

**说明**：启动一个有序的关闭方式，在关闭之前已提交的任务会被执行，但不会接收新任务。此方法不会等待已提交任务执行完毕（通过awaitTermination方法可以等待已提交的任务完成之后再关闭）。方法内部的调用在上面都已经介绍过，不多赘述。

###### shutdownNow()

public List<Runnable> shutdownNow() {  
 List<Runnable> tasks;  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
 checkShutdownAccess();//检查关闭权限  
 advanceRunState(*STOP*);//修改运行状态runState  
 interruptWorkers();//中断所有线程  
 tasks = drainQueue();//移除所有等待队列的任务  
 } finally {  
 mainLock.unlock();  
 }  
 tryTerminate();  
 return tasks;  
}

public boolean awaitTermination(long timeout, TimeUnit unit)  
 throws InterruptedException {  
 long nanos = unit.toNanos(timeout);  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
 for (;;) {  
 if (*runStateAtLeast*(ctl.get(), *TERMINATED*))  
 return true;  
 if (nanos <= 0)  
 return false;  
 nanos = termination.awaitNanos(nanos);  
 }  
 } finally {  
 mainLock.unlock();  
 }  
}

##### 小结

**关键点：****任务状态、等待线程的链表（使用Treiber栈实现）**

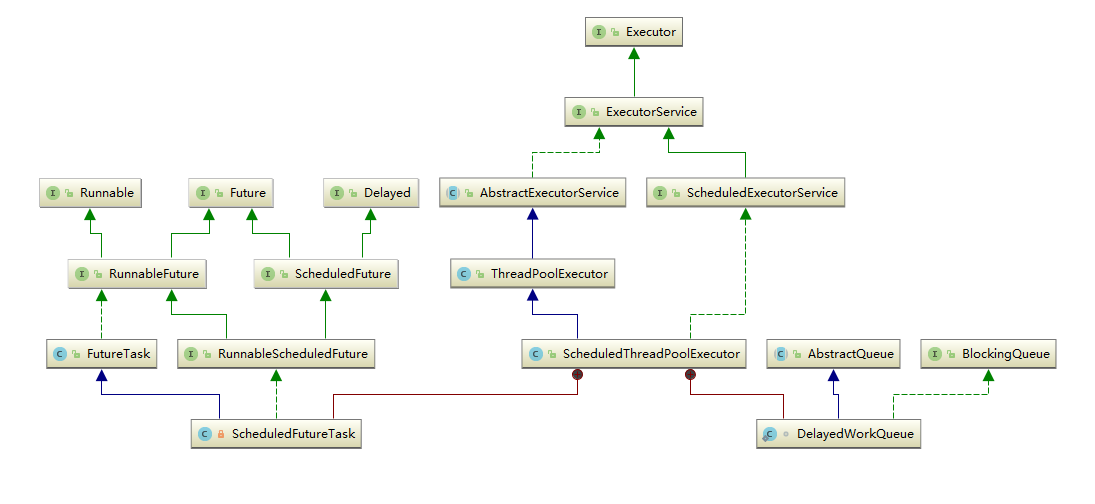
#### ScheduledThreadPoolExecutor

##### 概述

ScheduledThreadPoolExecutor继承自ThreadPoolExecutor，为线程池提供延期或者周期执行，功能与Timer类似，与Timer区别：

1. ScheduledThreadPoolExecutor线程会捕获任务重的异常，即使多个计划任务中存在某几个计划任务为捕获异常的情况，也不会影响ScheduledThreadPoolExecutor总线程的工作，不会影响其他计划任务的继续执行。
2. ScheduledThreadPoolExecutor是基于相对时间的，对系统时间的改变不敏感，但是如果执行某一绝对时间(如2017-10-16 11:32:45)执行任务，可能不好执行，此时可使用Timer。
3. ScheduledThreadPoolExecutor是线程池，如任务数过多或某些任务执行时间较长，可自动分配更多的线程来执行计划任务。

##### 数据结构和核心参数



//关闭后继续执行已经存在的周期任务  
private volatile boolean continueExistingPeriodicTasksAfterShutdown;

//关闭后继续执行已经存在的延时任务  
private volatile boolean executeExistingDelayedTasksAfterShutdown = true;

//取消任务后移除  
private volatile boolean removeOnCancel = false;

//为中断调度关系提供的顺序编号，保证在被绑定的entry之间的FIFO顺序  
private static final AtomicLong *sequencer* = new AtomicLong();

**说明：**

1. ScheduledThreadPoolExecutor继承自ThreadPoolExecutor，属于线程池的一种。
2. 内部实现了一个自定义的任务类型ScheduledFutureTask，即使是那些不需要调度的任务（也就是使用ExecutorService.execute执行的任务），也会被当作延迟为0的任务执行。
3. 内部实现了一个自定义的等待队列DelayedWorkQueue，是无界队列DelayQueue的一种变体，与ThreadPoolExecutor相比，缺乏容量约束和corePoolSize和maximumPoolSize，但实际上是简化了执行机制。

##### 函数列表

public ScheduledThreadPoolExecutor(int corePoolSize) {  
 super(corePoolSize, Integer.*MAX\_VALUE*, 0, *NANOSECONDS*,  
 new DelayedWorkQueue());  
}

public ScheduledThreadPoolExecutor(int corePoolSize,  
 ThreadFactory threadFactory) {  
 super(corePoolSize, Integer.*MAX\_VALUE*, 0, *NANOSECONDS*,  
 new DelayedWorkQueue(), threadFactory);  
}

public ScheduledThreadPoolExecutor(int corePoolSize,  
 RejectedExecutionHandler handler) {  
 super(corePoolSize, Integer.*MAX\_VALUE*, 0, *NANOSECONDS*,  
 new DelayedWorkQueue(), handler);  
}

public ScheduledThreadPoolExecutor(int corePoolSize,  
 ThreadFactory threadFactory,  
 RejectedExecutionHandler handler) {  
 super(corePoolSize, Integer.*MAX\_VALUE*, 0, *NANOSECONDS*,  
 new DelayedWorkQueue(), threadFactory, handler);  
}

*/\*\*  
 \* 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)

*/\*\*  
 \* 创建并执行在给定延迟后启用的ScheduledFuture任务  
 \*/*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)

public void execute(Runnable command) {  
 schedule(command, 0, *NANOSECONDS*);  
}

public Future<?> submit(Runnable task)

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

public <T> Future<T> submit(Callable<T> task)

public void shutdown() {  
 super.shutdown();  
}

public List<Runnable> shutdownNow() {  
 return super.shutdownNow();  
}

public BlockingQueue<Runnable> getQueue() {  
 return super.getQueue();  
}

##### 源码解析

###### schedule()

*/\*\*  
 \* 创建并执行在给定延迟后启用的ScheduledFuture任务  
 \*/*public <V> ScheduledFuture<V> schedule(Callable<V> callable,  
 long delay,  
 TimeUnit unit) {  
 if (callable == null || unit == null)  
 throw new NullPointerException();  
 RunnableScheduledFuture<V> t = decorateTask(callable,  
 new ScheduledFutureTask<V>(callable,  
 triggerTime(delay, unit)));//构造ScheduledFutureTask任务  
 delayedExecute(t);//任务执行主方法  
 return t;  
}

*/\*\*修改或替换用于执行callable的任务。此方法可被具体的类重载  
 \* 来管理内部任务。默认实现返回给定task  
 \*/*protected <V> RunnableScheduledFuture<V> decorateTask(  
 Callable<V> callable, RunnableScheduledFuture<V> task) {  
 return task;  
}

**说明：**schedule方法分两个版本（Runnable和Callable），这里拿其中一个schedule方法进行说明，其实内部都是先通过decorateTask组建任务，然后由delayedExecute方法来实现任务执行。delayedExecute源码如下：

delayedExecute

*/\*\*任务延迟或周期执行的主方法。如果池已经关闭，根据指定策略拒绝给定任务，  
 \* 否则添加给定任务到队列，并且开启一个线程。（由于给定任务可能还不能被运行，  
 \* 所以我们不能预先启动线程来运行任务）。如果在任务添加期间池被关闭，  
 \* 根据池状态和run-after-shutdown参数决定取消和移除任务。  
 \*/*private void delayedExecute(RunnableScheduledFuture<?> task) {  
 if (isShutdown())  
 reject(task);//池已关闭，执行拒绝策略  
 else {  
 super.getQueue().add(task);//任务入队  
 if (isShutdown() &&  
 !canRunInCurrentRunState(task.isPeriodic()) &&//判断run-after-shutdown参数  
 remove(task))//移除任务  
 task.cancel(false);  
 else  
 ensurePrestart();//启动一个新的线程等待任务  
 }  
}

*/\*\*  
 \* 修改或替换用于执行runnable的任务。此方法可被重写来管理内部任务。默认实现下返回给定task  
 \*/*protected <V> RunnableScheduledFuture<V> decorateTask(  
 Runnable runnable, RunnableScheduledFuture<V> task) {  
 return task;  
}

*/\*\*  
 \* 和TPE的prestartCoreThread作用一致，不同的是即使corePoolSize为0  
 \* 也会安排一个线程启动，  
 \*/*void ensurePrestart() {  
 int wc = *workerCountOf*(ctl.get());  
 if (wc < corePoolSize)  
 addWorker(null, true);  
 else if (wc == 0)  
 addWorker(null, false);  
}

//取消任务  
public boolean cancel(boolean mayInterruptIfRunning) {  
 boolean cancelled = super.cancel(mayInterruptIfRunning);  
 if (cancelled && removeOnCancel && heapIndex >= 0)  
 remove(this);  
 return cancelled;  
}

**说明：**任务延迟或周期执行的主方法。源码很简单，如果池可以继续接收任务，则通过ensurePrestart启动一个新的线程等待任务执行。ensurePrestart是TPE的方法，源码如下：

*/\*\*  
 \* 和TPE的prestartCoreThread作用一致，不同的是即使corePoolSize为0  
 \* 也会安排一个线程启动，  
 \*/*void ensurePrestart() {  
 int wc = *workerCountOf*(ctl.get());  
 if (wc < corePoolSize)  
 addWorker(null, true);  
 else if (wc == 0)  
 addWorker(null, false);  
}

###### ScheduledFutureTask.run()

//重写FutureTask的版本，以便在周期性的情况下重置/重排序  
public void run() {  
 boolean periodic = isPeriodic();//是否为周期任务  
 if (!canRunInCurrentRunState(periodic))//当前状态是否可以执行  
 cancel(false);  
 else if (!periodic)  
 //不是周期任务，直接执行  
 ScheduledFutureTask.super.run();  
 else if (ScheduledFutureTask.super.runAndReset()) {  
 setNextRunTime();//设置下一次运行时间  
 reExecutePeriodic(outerTask);//重排序一个周期任务  
 }  
}

//设置下一次执行任务的时间  
private void setNextRunTime() {  
 long p = period;  
 if (p > 0)  
 time += p;  
 else  
 time = triggerTime(-p);  
}

//计算固定延迟任务的执行时间  
long triggerTime(long delay) {  
 return now() +  
 ((delay < (Long.*MAX\_VALUE* >> 1)) ? delay : overflowFree(delay));  
}

说明：重写了FutureTask的run方法，如果不是周期任务，直接调用run方法执行；否则调用runAndReset方法运行任务，关于runAndReset的分析，请见[runAndReset()](#_runAndReset())。成功运行周期任务之后，调用setNextRunTime设置下一次执行的时间，然后调用reExecutePeriodic对当前周期任务进行重新排序。源码如下：

//设置下一次执行任务的时间  
private void setNextRunTime() {  
 long p = period;  
 if (p > 0)  
 time += p;  
 else  
 time = triggerTime(-p);  
}

//重排序一个周期任务  
void reExecutePeriodic(RunnableScheduledFuture<?> task) {  
 if (canRunInCurrentRunState(true)) {//池关闭后可继续执行  
 super.getQueue().add(task);//任务入列  
 //重新检查run-after-shutdown参数，不能运行，移除队列任务，取消任务的执行  
 if (!canRunInCurrentRunState(true) && remove(task))  
 task.cancel(false);  
 else  
 ensurePrestart();//启动一个新的线程等待任务  
 }  
}

###### scheduleAtFixedRate

*/\*\*  
 \* 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) {  
 if (command == null || unit == null)  
 throw new NullPointerException();  
 if (period <= 0)  
 throw new IllegalArgumentException();  
 //构建RunnableScheduledFuture任务类型  
 ScheduledFutureTask<Void> sft =  
 new ScheduledFutureTask<Void>(command,  
 null,  
 triggerTime(initialDelay, unit),//计算任务的延迟时间  
 unit.toNanos(period));//计算任务的执行周期  
 RunnableScheduledFuture<Void> t = decorateTask(command, sft);//执行用户自定义逻辑  
 sft.outerTask = t;//赋值给outerTask，准备重新入队等待下一次执行  
 delayedExecute(t);//执行任务  
 return t;  
}

*/\*\*  
 \* 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) {  
 if (command == null || unit == null)  
 throw new NullPointerException();  
 if (delay <= 0)  
 throw new IllegalArgumentException();  
 //构建RunnableScheduledFuture任务类型  
 ScheduledFutureTask<Void> sft =  
 new ScheduledFutureTask<Void>(command,  
 null,  
 triggerTime(initialDelay, unit),//计算任务的延迟时间  
 unit.toNanos(-delay));//计算任务的执行周期  
 RunnableScheduledFuture<Void> t = decorateTask(command, sft);//执行用户自定义逻辑  
 sft.outerTask = t;//赋值给outerTask，准备重新入队等待下一次执行  
 delayedExecute(t);//执行任务  
 return t;  
}

**说明：**跟schedule的执行步骤一样，这里主要说明一下scheduleAtFixedRate和scheduleWithFixedDelay这两个方法的不同：  
**scheduleAtFixedRate是首先延期执行一次，然后每隔period周期执行  
scheduleWithFixedDelay是首先延期执行一次，第一次执行完毕之后隔period开始下一次执行，之后每次执行都是等待前一个任务完成之后再延时period执行。**

###### onShutdown()

public void shutdown() {  
 super.shutdown();  
}

//取消并清除由于关闭策略不应该运行的所有任务  
@Override void onShutdown() {  
 BlockingQueue<Runnable> q = super.getQueue();  
 //获取run-after-shutdown参数  
 boolean keepDelayed =  
 getExecuteExistingDelayedTasksAfterShutdownPolicy();  
 boolean keepPeriodic =  
 getContinueExistingPeriodicTasksAfterShutdownPolicy();  
 if (!keepDelayed && !keepPeriodic) {//池关闭后不保留任务  
 //依次取消任务  
 for (Object e : q.toArray())  
 if (e instanceof RunnableScheduledFuture<?>)  
 ((RunnableScheduledFuture<?>) e).cancel(false);  
 q.clear();//清除等待队列  
 }  
 else {//池关闭后保留任务  
 // Traverse snapshot to avoid iterator exceptions  
 //遍历快照以避免迭代器异常  
 for (Object e : q.toArray()) {  
 if (e instanceof RunnableScheduledFuture) {  
 RunnableScheduledFuture<?> t =  
 (RunnableScheduledFuture<?>)e;  
 if ((t.isPeriodic() ? !keepPeriodic : !keepDelayed) ||  
 t.isCancelled()) { // also remove if already cancelled  
 //如果任务已经取消，移除队列中的任务  
 if (q.remove(t))  
 t.cancel(false);  
 }  
 }  
 }  
 }  
 tryTerminate(); //终止线程池  
}

**说明：**池关闭方法调用了TPE的shutdown，具体分析见[5.2.4.5 shutdown()](#_shutdown())。这里主要介绍以下在shutdown方法中调用的关闭钩子onShutdown方法，它的主要作用是在关闭线程池后取消并清除由于关闭策略不应该运行的所有任务，这里主要是根据run-after-shutdown参数（continueExistingPeriodicTasksAfterShutdown和executeExistingDelayedTasksAfterShutdown）来决定线程池关闭后是否关闭已经存在的任务。源码其实很简单，不多赘述。

##### 小结

**关键点：**

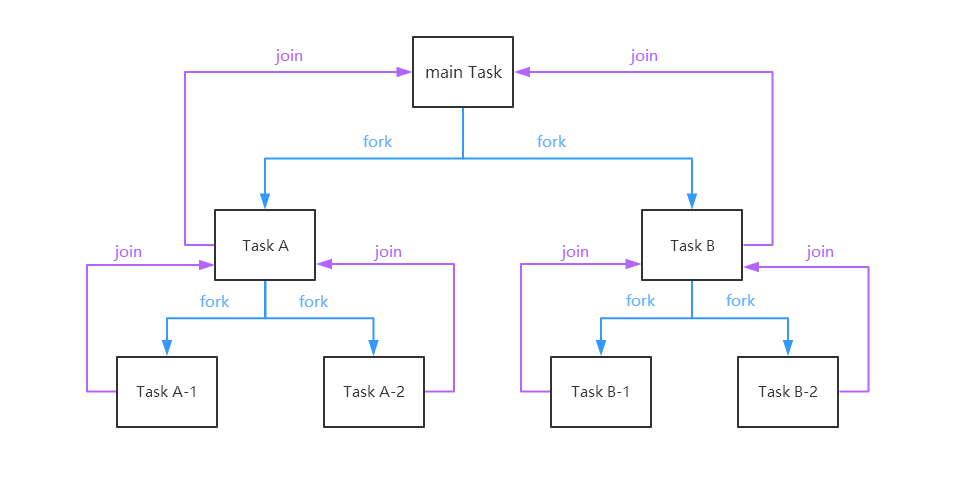
1. **任务延迟执行的原理**
2. **内部实现的自定义任务类型ScheduledFutureTask**
3. **内部实现的等待队列DelayedWorkQueue**
4. **常用方法如submit、execute、schedule、scheduleAtFixedRate、scheduleWithFixedDelay**

#### ForkJoinPool

##### 概述

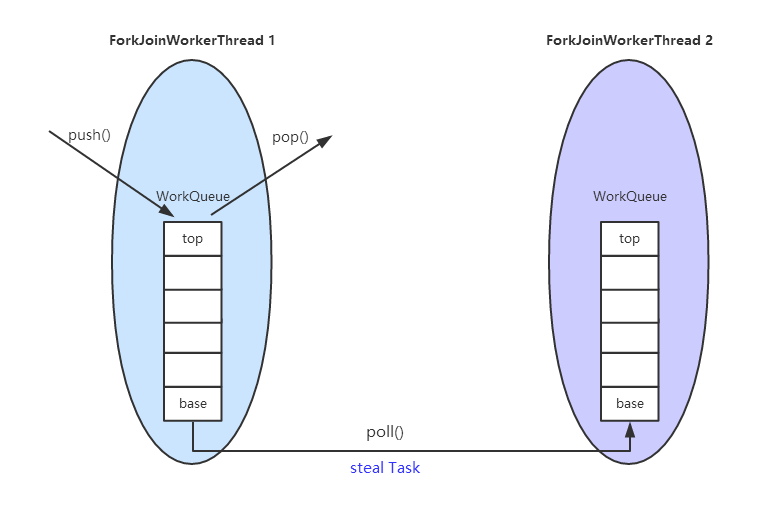
ForkJoinPool是1.7新增的一个线程池类，主要来运行ForkJoinTask任务，也为非ForkJoinTask客户端提供了入口，以及任务管理和监控功能。和其他线程池不同的是它使用了work-stealing算法：线程池中的所有线程尝试找到并执行其他已经提交的任务或者是被其他活动线程创建的任务（如果不存在就阻塞）。

fork/join框架是ExecutorService接口的一种具体实现，目的是为了帮助你更好地利用多处理器带来的好处。它是为那些能够被递归地拆解成子任务的工作类型量身设计的。其目的在于能够使用所有可用的运算能力来提升你的应用的性能。ForkJoinPool的实现就是如此：



ForkJoinPool 可以满足**并行**地实现**分治算法（Divide-and-Conquer）**的需要。

**Work-stealing（工作窃取）：**所谓 Work-Stealing，在 ForkJoinPool 中的实现为：线程池中每个线程都有一个互不影响的任务队列（双端队列），线程每次都从自己的任务队列的队头中取出一个任务来运行（**LIFO**）；如果某个线程对应的队列已空并且处于空闲状态，而其他线程的队列中还有任务需要处理但是该线程处于工作状态，那么空闲的线程可以从其他线程的队列的队尾取一个任务来帮忙运行 (**FIFO**)—— 就像是空闲的线程去偷人家的任务来运行一样，所以叫 “工作窃取”:



**适用场景：**Work-Stealing 的适用场景是不同的任务的耗时相差比较大，即某些任务需要运行较长时间，而某些任务会很快的运行完成，这种情况下用 Work-Stealing 很合适；但是如果任务的耗时很平均，则此时 Work-Stealing 并不适合，因为窃取任务时不同线程需要抢占锁，这可能会造成额外的时间消耗，而且每个线程维护双端队列也会造成更大的内存消耗。所以 ForkJoinPool 并不是 ThreadPoolExecutor 的替代品，而是作为对 ThreadPoolExecutor 的补充。

Submitters 提交任务的线程 只能执行自己提交的本地任务

Worker 工作线程 可执行子任务

|  |  |  |
| --- | --- | --- |
| Summary of task execution methods | | |
|  | **Call from non-fork/join clients** | **Call from within fork/join computations** |
| **Arrange async execution** | [execute(ForkJoinTask)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html#execute-java.util.concurrent.ForkJoinTask-) | [ForkJoinTask.fork()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html#fork--) |
| **Await and obtain result** | [invoke(ForkJoinTask)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html#invoke-java.util.concurrent.ForkJoinTask-) | [ForkJoinTask.invoke()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html#invoke--) |
| **Arrange exec and obtain Future** | [submit(ForkJoinTask)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html#submit-java.util.concurrent.ForkJoinTask-) | [ForkJoinTask.fork()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html#fork--) (ForkJoinTasks *are* Futures) |

###### 官方文档说明

FrokJoinPool的源码可以说是JUC中最难读的源码之一了，大量的位运算，方法注释少的可怜。 推荐大家还是先看一下官方的说明，这里笔者翻译了一份中文的说明，内容比较长，水平有限，如有不对的地方还请指正。

1. 实现概览

**ForkJoinPool和它的内部类为一组工作线程提供主要功能和控制：非FJ线程任务放在submission queue，工作线程拿到这些任务并拆分为多个子任务，队列任务也可能被其他工作线程偷取。工作线程优先处理来自自身队列的任务（LIFO或FIFO，参数mode决定），然后以FIFO的顺序随机窃取其他队列中的任务。work-stealing框架在刚开始是为了支持树状结构并行操作，随着时间的推移它的可伸缩性优势得到扩展和优化，以更好地支持更多样化的使用。由于很多内部方法和嵌套类相互关联，它们的主要原理和描述都在下面展示；个别方法和嵌套类只包含关于细节的简短注释。**

1. WorkQueues

**大多数操作都发生在work-stealing队列中(内部类WorkQueue)。此workQueue支持三种形式的出列操作：push、pop、poll(也叫steal)，push和pop只能被队列内部持有的线程调用，poll可被其他线程偷取任务时调用。和最初的 Work-Stealing 队列不同，考虑到GC问题，我们一有机会就将队列的获取槽位置空，即使是在生成大量任务的程序中，也要保持尽可能小的占用空间。为了实现这一点，我们使用CAS解决pop和poll(steal)的线程冲突问题。  
关于pop和poll的区别，下面用一段简单的伪代码说明：**

pop，持有队列的工作线程调用:  
 if ((base != top) and  
 (在top位的任务不为null) and  
 (通过CAS操作设置top位为null))  
 递减top值并返回任务;  
  
 poll，通常是偷取线程调用：  
 if ((base != top) and  
 (在base位的任务不为null) and  
 (base位没有被其他线程修改) and  
 (通过CAS操作设置base位为null))  
 增加base值并返回任务;

由于使用了CAS引用数组，所以在队列base和top不必使用标志位。这些也是数组结构的队列的特性（例如ArrayDeque）。对索引的更新保证了当top==base时代表这个队列为空，但是如果在push、pop或poll没有完全完成的情况下，可能出现即使base==top但队列为非空的错误情况（isEmpty方法可以检查“移除最后一个元素操作”部分完成的情况）。所以，单独考虑poll操作，它并不是wait-free的(无等待算法)。在一个线程正在偷取任务时，另外一个线程是无法完成偷取操作的。大体上讲，我们起码有一定概率保证了阻塞性。如果一个偷取操作失败，偷取线程会选择另外一个随机目标继续尝试。所以，为了保证偷取线程能够正常执行，它必须能够满足任何正在执行的对queue的poll或push操作都能完成（这就是为什么我们通常使用pollAt方法和它的变体，在已知的base索引中先尝试一次，然后再考虑可替代的操作，而不直接使用可以重试的poll方法）

这种方法同样也支持本地任务（非join任务）以FIFO模式运行，只需要使用poll而非pop。FIFO和LIFO这两种模式都不会考虑共用性、加载、缓存地址等，所以很少能在给定的机器上提供最好的性能，但通过对这些因素进行平均，可以提供良好的吞吐量。更进一步来讲，即使我们尝试使用这些信息，也没有能利用它的基础。例如，一些任务集从缓存共用中获取到良好的性能收益，但其他任务集会因此受到它的影响。另外，虽然队列中提供了扫描功能，但是从长远看来为了吞吐量通常最好使用随机选择，而非直接选择。所以，在ForkJoinPool 中我们也就使用了一种 XorShifts（一种随机算法，有些带有不同的偏移量） 随机算法。

WorkQueue 对于提交到池中的任务也使用类似的随机插入方式。我们不能把这些被工作线程使用的任务混合同一个队列中，所以，我们会**使用一种随机哈希算法（有点类似ConcurrentHashMap的随机算法）将工作队列与工作线程关联起来**。ThreadLocalRandom的probe(探针值)会为选中的已存在的队列提供一个哈希值，在与其他提交任务的线程（submitters）竞争时，也可以利用probe来随机移位。实际上，**submitters 就像工作线程（worker）一样，只不过他们被限制只能执行它们提交的本地任务（CountedCompleter类型任务也不能执行）**。在共享模式里提交的任务需要锁来控制（在扩容情况下提供保护），我们使用了一个简单的自旋锁(qlock字段，后面详细讲解)，因为 submitters 遇到一个繁忙队列时会继续尝试提交其他队列或创建新的队列-只有在 submitters 创建和注册新队列时阻塞。另外，"qlock"会在shutdown时饱和到不可锁定值(-1)，但是解锁操作依然可以执行。

1. 池管理

**work-stealing模式的吞吐量优势来自于分散控制：工作线程主要从它们自己的队列或其他worker的队列中获取任务，速度可以超过每秒十亿**。池本身的创建、激活(为运行中的任务提供扫描功能)、撤销、阻塞和销毁线程这些操作的信息都很小，只有一小部分属性我们可以全局追踪或维护，所以我们把他们封装成一系列小的数字变量，并且这些变量通常是没有阻塞或锁定的原子数。**几乎所有基本的原子控制状态都被保存在两个volatile变量中(ctl和runState)，这两个变量常被用来状态读取和一致性检查（此外，"config"字段持有了不可变的配置状态）。**

**ctl定义为一个64位的原子类型字段，用来标识对工作线程进行添加、灭活、重新激活和对队列出列、入列操作。**

**ctl原子性地维护了活跃线程数、工作线程总数，和一个放置等待线程的队列。活跃线程数同样也担任静止指标的角色，当工作线程确认已经没有任务可以执行时，就递减这个值。工作队列其实是一个Treiber栈（无锁并发栈），它可以以最近使用的顺序来存储活跃线程。这改善了线程执行性能和任务定位，只要任务在栈的顶端，在发生争用时也可以很好的释放工作线程。当找不到worker时，我们使用park/unpark来操作已经被放到空闲worker栈的工作线程（使用ctl低32位表示）。栈顶状态也维护了工作线程的“scanState”子域：它的索引和状态(ctl)，加上一个版本计数(SS\_SEQ)，再加上子域数（同样也可作为版本戳）提供对Treiber栈ABA问题的保护。**

**runState**表示当前池的运行状态（STARTED、STOP等），同时也可作为锁保护workQueues数组的更新。当作为一个锁使用时，通常只能被一部分操作持有(唯一的异常是数组进行一次性初始化和非常规扩容)，所以最多在一段自旋后总是处于可用状态。需要注意的是，awaitRunStateLock 方法(仅当使用CAS获取失败时调用)在自旋一段时间之后，使用 wait/notify 机制来实现阻塞(这种情况很少)。对于一个高竞争的锁来说这是一个很不好的设计，但多数pool在自旋限制之后都是在无锁竞争情况下运行的，所以作为一个保守的选择它运行的还是比较稳定的。因为我们没有其他的内部对象作为监视器使用，所以“stealCounter”(AtomicLong，也是在externalSubmit中延迟加载)充当了这个角色。

**"runState" 和 "ctl"的用法只有在一种情况中会相互影响： 当添加一个新的工作线程时（tryAddWorker），只有在获得锁时才可以对ctl进行CAS操作。**

**scanState**：工作线程(worker)和线程池(pool)都使用了 scanState，**通过 scanState 可以管理和追踪工作线程是否为 INACTIVE（可能正在阻塞等待唤醒）状态，也可以判断任务是否为 SCANNING 状态（当两者都不是时，它就是正在运行的任务）**。当一个工作线程处于灭活状态(INACTIVE)，它的scanState被设置为禁止执行任务，但是即便如此它也必须扫描一次以避免队列争用。注意，scanState 的更新在队列CAS释放之后（会有延迟）。在工作线程入队后，scanState的低16位必须持有它在池中的索引，所以我们在初始化时就将索引设置好(参考 registerWorker 方法)，并将其一直保存在那里或在必要时恢复它。

**WorkQueue记录**： ForkJoinPool 内部持有一个 WorkQueue 数组（"workQueues"）来记录任务工作队列。这个数组在第一次使用时(externalSubmit方法)创建，如果必要的话对其进行扩容。数组的更新操作受runState锁的保护，但可以并发读取。**为了简化基于索引的操作，数组大小一定为2的幂，并且可存储null值。工作任务(Worker queues)存放在奇数位索引;共享任务(submission/external queues)存放在偶数位索引，最多64个槽位。以这种方式将它们组合在一起可以简化和加速任务扫描。**

所有工作线程都是按需创建，被submission任务触发，然后被终止工作线程替代，或者作为阻塞线程的补偿线程。为了简化GC的工作，我们**并不会直接持有工作线程的引用，而是通过索引的方式来访问工作队列**(所以ForkJoinPool的代码结构看上去很麻烦)。**实际上，工作队列数组充当弱引用机制，例如使用ctl也维护了索引的栈顶(top)域而不是直接存储引用。**

**闲置工作线程排队**：跟高性能计算(HPC)work-stealing框架不同，当没有立刻扫描到任务时，我们不能让工作线程无限制的自旋扫描， **除非有任务可用，否则不能开启/重启工作线程**。另一方面，当新任务提交或生成时，我们必须快速地将它们推进行动。在许多情况下，通过增加时间来激活工作线程是限制整体性能的主要因素，并且JIT编译和分配使程序启动更加复杂，所以我们应尽可能地简化它。

**内存排序**：**由于我们必须不定时的唤醒某个工作线程，为了不会丢失signal信号指令，就需要更强的内存排序规则（full-fence）来避免指令重排序。一些核心操作：如出列、更新ctl状态也需要full-fence来进行CAS操作。数组的读取则是使用Unsafe类提供的仿照volatile的方式。从其他线程访问 WorkQueue 的base、top和任务数组时，也需要保证首次读取的可见性，所以，我们对"base"引用使用了volatile来修饰，并且在读取其他字段或引用之前总是先读取"base"。队列的持有线程也必须要保证变量的顺序更新，所以在更新时使用了内建指令（如putOrderedObject）。**

**Worker（工作线程）创建：创建worker时，首先增加worker总数(作为一个保留字段)，然后构造一个 ForkJoinWorkerThread。在构造新的ForkJoinWorkerThread期间，会调用 registerWorker 来创建对应的工作队列 workQueue，并在workQueues数组(如果需要就对数组扩容)中为之分配一个索引，然后启动线程。如果在这期间出现异常或线程工厂创建线程失败，则调用 deregisterWorker 回滚线程数和其他记录。**如果工作线程创建失败，线程池就会使用比目标数更少的工作线程继续运行。如果发生异常，异常通常被传递给其他外部调用者。为每个工作线程都分配索引避免了在扫描时发生偏移。我们可以把这个数组看作是一个简单的2幂哈希表，根据元素需要来进行扩容。当需要扩容或一个工作线程被撤销并替换时，增加seedIndex值以保证不会发生碰撞，在这之后要尽可能的保证低碰撞。

**灭活和等待：**任务入队时可能会遭遇到内部竞争：**当一个线程已经放弃查找任务，但还没有进入工作队列等待，此时另外一个生产任务的线程可能会错过查看（并唤醒）这个线程。**当一个工作线程偷取不到任务，它就会被灭活并入队等待。很多时候，由于GC或OS调度，缺少任务只是暂时性的；为了减少这种类型的灭活，扫描者在扫描期间会计算队列的 checksum (每次偷取时，对每个队列的base索引进行求和)。在扫描完成后并且 checksum 已经稳定，工作线程才可以放弃查找并尝试灭活；为了避免错过唤醒，在成功入队等候再次稳定之后会再重复这个扫描过程。在这种状态下，工作线程不能take/run任务，除非这个任务在队列中被释放，所以工作线程本身最终会尝试释放自己或其他继承者(参见tryRelease)。另外，在一次空的扫描过程中，一个失活的工作线程在阻塞(park)前会自旋指定的次数(参见 awaitWork)。**注意关于伴随着parking或其他阻塞的Thread.interrupt指令的不寻常的约定：线程的中断状态仅仅用来检查线程是否可以销毁，而且在线程阻塞时同样也会检查线程是否可销毁，所以在调用park阻塞线程之前我们会先清除中断状态（调用Thread.interrupted），这样做是主要是为了防止在用户代码中，会有一些与内部使用不相关的调用导致中断状态被设置，从而导致线程被错误地销毁。**

**唤醒和激活：在至少有一个可被找到并执行的任务时，工作线程才会被创建或激活。**在一个先前为空的队列中进行push(工作线程或外部提交)操作时，如果有空闲工作线程正在等待就唤醒它，如果工作线程少于给定并行度(parallelism)就创建一个新的工作线程。**在多数平台上，signal(unpark)指令的开销很大，在真正唤醒一个线程和它实际运行之间的时间会很长，因此，尽可能多地消除这些延迟是很有必要的。**并且，失活的工作线程并非都是被阻塞，它们经常会进行二次扫描或自旋，我们通过设置/清除WorkQueue的"parker"引用来减少不必要的unpark操作。

**缩减工作线程：**资源空闲一定的时间后需要释放。如果池保持静止状态超过一个周期时长（IDLE\_TIMEOUT，默认2秒），工作线程就会由于超时而被终止(参见awaitWork)，随着线程数量的减少和周期的增加，最终移除所有的工作线程。当剩余线程数多于2个，过量的线程在下一个静止点会被立即终止。

**关闭和终止：**   
shutdownNow()：直接终止，内部调用 tryTerminate 首先设置runState值，然后终止调用线程和其他所有工作线程，通过设置他们的 qlock 状态来帮助销毁，取消他们未处理的任务，并唤醒等待的线程，重复上述操作一直到池处于稳定状态（循环数被工作线程数量限制，至少3次）。  
shutdown()：首先检查池是否可以终止，依赖"ctl"内维护的的活动线程数—在awaitWork中如果池处于稳定状态并且活动线程数<=0也会调用tryTerminate进行销毁操作。不过，外部提交任务(submissions)并不依照上述条件。  
tryTerminate通过扫描队列(处于稳定状态)，以确保在触发"STOP"阶段之前没有正在进行的submission任务和work任务需要处理。（注意：如果在池关闭时调用 helpQuiescePool 会发生内部冲突，因为他们都会等待静止，在 helpQuiescePool 完成执行之前 tryTerminate 都不会被触发）

1. Join任务

**由于我们把许多任务都复用给一批工作线程，并且不能让它们阻塞，也不能为这些任务重新分配其他的运行时堆栈。所以就使用了一种"延续性"的形式**，即使这可能不是一个好的方案，因为我们可能既需要一个非阻塞的任务，也需要它的延续性来继续运行。为此我们总结了两种策略：

**Helping-帮助运行**：如果偷取还未开始，为这些 joiner 安排一些它可以执行的其它任务。

**Compensating-补偿运行**：如果没有足够的活动线程，tryCompensate()可能创建或重新激活一个备用的线程来为被阻塞的 joiner 补偿运行。

**第三种形式(在方法 tryRemoveAndExec 中实现)相当于帮助一个假想的补偿线程来运行任务**：如果补偿线程窃取并执行的是被join的任务,那么join线程不需要补偿线程就可以直接执行它（尽管牺牲了更大的运行时堆栈，但这种权衡通常是值得的）。设想一下这种互相帮助的场景：**补偿线程帮助join线程执行任务，反过来join线程也会帮助补偿线程执行任务。**

**helpStealer（补偿执行）** 使用了一种**“线性帮助(linear helping)”**的算法。每个工作线程都记录了最近一个从其他工作队列（或submission队列）偷取过来的任务（currentSteal引用），同样也记录了当前被join的任务（currentJoin引用）。helpStealer 方法使用这些标记去尝试找到偷取者并帮助它执行任务，（**也就是说，从偷取任务中拿到任务并执行，“偷取者偷我的任务执行，我去偷偷取者的任务执行”**），这样就可以加速任务的执行。关于这个算法可以查看Wagner & Calder的论文Leapfrogging(跨越式)：一种实现高效 Future 模式的便携式技巧。这种算法在ForkJoinPool中的大概实现方式如下：

* + 1. 从worker到steal之间我们只保存依赖关系，而不是记录每个steal任务。有时可能需要对workQueues进行线性扫描来定位偷取者，但是一般不需要，因为偷取者在偷取任务时会把他的索引存放在在 hint 引用里。一个worker可能进行了多个偷取操作，但只记录了其中一个偷取者的索引(通常是最近的那个)，为了节省开销，hint在需要时才会记录
    2. 它是相对“浅层的”，忽略了嵌套和可能发生的循环相互偷取。
    3. currentJoin引用只有在join的时候被更新，这意味着我们在执行生命周期比较长的任务时会丢失链接，导致GC停转(在这种情况下利用阻塞通常是一个好的方案)。
    4. 我们使用 checksum 限制查找任务的次数，然后挂起工作线程，必要时使用其他工作线程替换它。

**注意：CountedCompleter**的帮助动作不需要追踪currentJoin：helpComplete方法会获取并执行在同一个父节点下等待的所有任务。不过，这仍然需要对completer的链表进行遍历，所以使用CountedCompleters相对于直接定位currentJoin效率要低。

**补偿执行的目的不在于在任何给定的时间内保持未阻塞线程的目标并行数。这个类之前的版本为所有阻塞的join任务都提供即时补偿，然而，在实践中，绝大多数阻塞都是由GC和其他JVM或OS活动产生的瞬时的附带影响，这种情况下使用补偿线程替换工作线程会使情况变得更糟。**现在，通过检查WorkQueue.scanState的状态确认所有活动线程都正在运行，然后使用补偿操作消除多余的活跃线程数。补偿操作通常情况下是被忽略运行的(容忍少数线程)，因为它带来的利益很少：**当一个有着空等待队列的工作线程在join时阻塞，它仍然会有足够的线程来保证活性，所以不需要进行补偿。**

补偿机制可能是有界限的。commonPool的界限(commonMaxSpares)使JVM在资源耗尽之前能更好的处理程序错误和资源滥用。用户可能通过自定义工厂来限制线程的构造，所以界限的作用在这种pool中是不精确的。当线程撤销(deregister)时，工作线程的总数会随之减少，不用等到他们退出并且资源被JVM和OS回收时才减少工作线程数，所以活跃线程在此瞬间可能会超过界限。

1. Common Pool

**在ForkJoinPool静态初始化之后commonPool会一直存在，并且在应用中共享，使用common pool通常可以减少资源占用(它的线程在空闲一段时间后可以回收再使用)**。大多数初始操作会发生在首次提交任务时，在方法externalSubmit中进行。

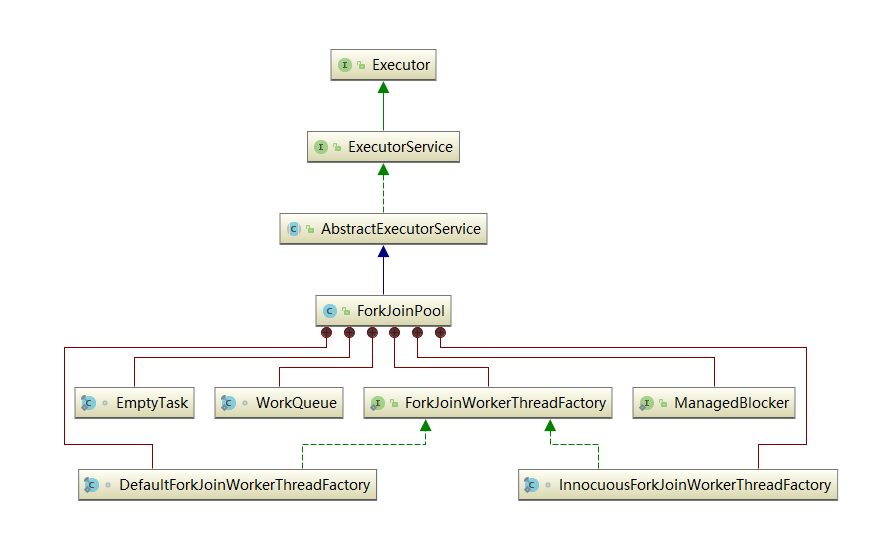
当外部线程提交任务到commonPool时，**在join过程中他们也可以帮助执行子任务**(参见externalHelpComplete和相关方法)。这种**caller-helps策略**是合理的，它会设置commonPool并行度为1(或更多，但小于可用内核的总数)，或者对于纯粹的caller-runs(调用者运行)直接设置为0。我们不需要记录外部提交任务是否属于commonPool—如果不是，外部帮助方法会快速返回结果。

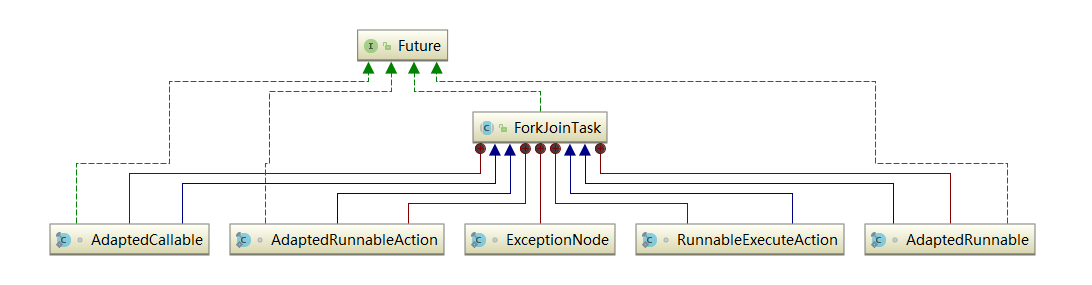
*当系统配置中有一个SecurityManager(安全管理)时，我们使用InnocuousForkJoinWorkerThread代替ForkJoinWorkerThread。这些工作线程没有设置许可，不属于任何明确的用户线程组，并且在执行完任何顶层任务后消除所有的ThreadLocal变量(参见WorkQueue.runTask)。这些相关的机制(主要在ForkJoinWorkerThread中)都是依赖JVM的，而且为了达到预期效果，必须访问特定的线程域*

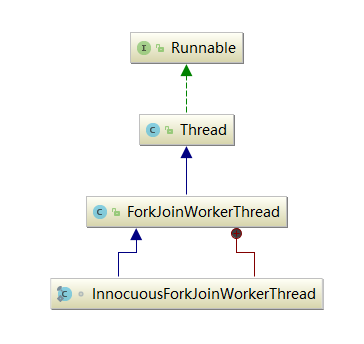
**以上就是官方文档的全部说明了，很多地方的翻译可能不太准确，还是推荐大家去读英文原版文档。**

**在java.util.concurrent包中，Fork/Join框架主要由ForkJoinPool、ForkJoinWorkerThread和ForkJoinTask来实现，它们之间有着很复杂的联系。所以本章我们会穿插讲解Fork/Join框架的运行机制。**

##### 数据结构和核心参数









首先说一下ForkJoinPool的各个内部类和相关术语，帮助源码阅读：

1. **ForkJoinWorkerThreadFactory** 内部线程工厂接口，用于创建工作线程ForkJoinWorkerThread
2. **DefaultForkJoinWorkerThreadFactory** ForkJoinWorkerThreadFactory的默认实现类
3. **InnocuousForkJoinWorkerThreadFactory** 实现了ForkJoinWorkerThreadFactory，无许可线程工厂，当系统变量中有系统安全管理相关属性时，默认使用这个工厂创建工作线程。
4. **EmptyTask** 内部占位类，用于替换队列中join的任务
5. **WorkQueue** work-stealing模式的双端任务队列，内部存放ForkJoinTask对象任务，使用@Contented注解修饰防止伪共享，如果构造中asyncMode(异步模型)为true，则使用FIFO队列
   1. 每个工作线程在运行中产生新的任务（通常是因为调用了 fork()）时，会放入工作队列的队尾（top位），并且工作线程在处理自己的工作队列时，使用的是 LIFO 方式，也就是说每次从队尾取出任务来执行。
   2. 每个工作线程在处理自己的工作队列同时，会尝试窃取一个任务（或是来自于刚刚提交到 pool 的任务，或是来自于其他工作线程的工作队列），窃取的任务位于其他线程的工作队列的队首（base位），也就是说工作线程在窃取其他工作线程的任务时，使用的是 FIFO 方式。
6. **任务分类：** 在ForkJoinPool中有两种任务，一种是直接通过FJP提交的任务（external/submissions task），存放在偶数索引位置；还有一种是通过内部fork分割的子任务（worker），存放在奇数索引位。
7. **伪共享状态：**缓存系统中是以缓存行（cache line）为单位存储的。缓存行是2的整数幂个连续字节，一般为32-256个字节。最常见的缓存行大小是64个字节。当多线程修改互相独立的变量时，如果这些变量共享同一个缓存行，就会无意中影响彼此的性能，这就是伪共享。
8. **Quiescence**:池静止状态，当活跃工作线程数等于0时，池处于Quiescence状态
9. **Stable：**稳定状态，多线程编程中，由于某个属性在同一时间可能被多个线程访问，所以在使用时应该验证属性的稳定性（例如在ForkJoinPool中，在使用ctl之前都会判断ctl值的一致性）

**常用位运算说明：**

1. **Parallelism**并行度 = config & SMASK
2. **定位偷取者索引**：((被偷取WorkQueue.hint|1) + 2passes) & (workQueues.length-1)
3. **获取base偏移地址**：(((ForkJoinTask<?>[].length-1) & base) << ASHIFT)+ABASE;
4. **ABASE ASHIFT：**

*ABASE* = *U*.arrayBaseOffset(ak);//数组第一个元素的偏移地址  
int scale = *U*.arrayIndexScale(ak);//数组中元素的增量地址，也就是说每个元素的占位数  
*ASHIFT* = 31 - Integer.*numberOfLeadingZeros*(scale);//numberOfLeadingZeros返回整数二进制补码中最高位（最左边）的位连续为0的个数

1. **Ctl**：((-parallelism) << AC\_SHIFT) & AC\_MASK | (((-parallelism) << TC\_SHIFT) & TC\_MASK)
2. **工作线程总数**：(config & SMASK) + (short) (c >>> TC\_SHIFT)
3. **活跃线程数：**(config & SMASK) + (int) (c >> AC\_SHIFT)

**ForkJoinPool、****ForkJoinTask、****RecursiveTask、RecursiveAction、ForkJoinWorkerThread之间的关系：****ForkJoinPool中使用ForkJoinWorkerThread来运行ForkJoinTask任务，ForkJoinPool只接收ForkJoinTask任务，RecursiveTask是ForkJoinTask的子类，是一个可以递归执行的ForkJoinTask，RecursiveAction是一个无返回值的RecursiveTask。**

// Bounds 限定参数  
static final int *SMASK* = 0xffff; // short bits == max index 低位掩码，也是最大索引位  
static final int *MAX\_CAP* = 0x7fff; // max #workers - 1 工作线程最大容量  
static final int *EVENMASK* = 0xfffe; // even short bits 偶数低位掩码  
static final int *SQMASK* = 0x007e; // max 64 (even) slots 最多64个槽位  
  
// Masks and units for WorkQueue.scanState and ctl sp subfield ctl子域和WorkQueue.scanState的掩码和标志位  
static final int *SCANNING* = 1; // false when running tasks 当任务在运行时为false  
static final int *INACTIVE* = 1 << 31; // must be negative 负数  
static final int *SS\_SEQ* = 1 << 16; // version count 版本戳，防止ABA问题  
  
// Mode bits for ForkJoinPool.config and WorkQueue.config. ForkJoinPool.config和WorkQueue.config的模式标记位  
static final int *MODE\_MASK* = 0xffff << 16; // top half of int 模式掩码  
static final int *LIFO\_QUEUE* = 0; //LIFO队列  
static final int *FIFO\_QUEUE* = 1 << 16;//FIFO队列  
static final int *SHARED\_QUEUE* = 1 << 31; // must be negative 共享模式队列，负数

//初始队列容量，2的幂  
static final int *INITIAL\_QUEUE\_CAPACITY* = 1 << 13;//最大队列容量  
static final int *MAXIMUM\_QUEUE\_CAPACITY* = 1 << 26; // 64M  
  
// Instance fields 实例字段  
volatile int scanState; // versioned, <0: inactive; odd:scanning  
int stackPred; // pool stack (ctl) predecessor 前一个栈顶的ctl  
int nsteals; // number of steals 偷取任务数  
int hint; // randomization and stealer index hint 偷取者索引  
int config; // pool index and mode 池索引和模式  
volatile int qlock; // 1: locked, < 0: terminate; else 0  
volatile int base; // index of next slot for poll 下一个poll操作的索引  
int top; // index of next slot for push 下一个push操作的索引  
ForkJoinTask<?>[] array; // the elements (initially unallocated) 任务数组  
final ForkJoinPool pool; // the containing pool (may be null)  
final ForkJoinWorkerThread owner; // owning thread or null if shared 当前工作队列的工作线程，共享模式下为null  
volatile Thread parker; // == owner during call to park; else null 调用park阻塞期间为owner，其他情况为null  
volatile ForkJoinTask<?> currentJoin; // task being joined in awaitJoin 记录被join过来的任务  
volatile ForkJoinTask<?> currentSteal; // mainly used by helpStealer 记录从其他工作队列偷取过来的任务

// Lower and upper word masks 高位和低位掩码  
private static final long *SP\_MASK* = 0xffffffffL;  
private static final long *UC\_MASK* = ~*SP\_MASK*;  
  
// Active counts 活跃线程数  
private static final int *AC\_SHIFT* = 48;  
private static final long *AC\_UNIT* = 0x0001L << *AC\_SHIFT*;  
private static final long *AC\_MASK* = 0xffffL << *AC\_SHIFT*;  
  
// Total counts 工作线程总数  
private static final int *TC\_SHIFT* = 32;  
private static final long *TC\_UNIT* = 0x0001L << *TC\_SHIFT*;  
private static final long *TC\_MASK* = 0xffffL << *TC\_SHIFT*;  
private static final long *ADD\_WORKER* = 0x0001L << (*TC\_SHIFT* + 15); // sign  
  
// runState bits: SHUTDOWN must be negative, others arbitrary powers of two 池状态  
private static final int *RSLOCK* = 1;  
private static final int *RSIGNAL* = 1 << 1;  
private static final int *STARTED* = 1 << 2;  
private static final int *STOP* = 1 << 29;  
private static final int *TERMINATED* = 1 << 30;  
private static final int *SHUTDOWN* = 1 << 31;//负数  
  
// Instance fields  
volatile long ctl; // main pool control  
volatile int runState; // lockable status  
final int config; // parallelism, mode 并行模式  
int indexSeed; // to generate worker index 生成工作线程索引  
volatile WorkQueue[] workQueues; // main registry 主对象注册信息，workQueue  
final ForkJoinWorkerThreadFactory factory;//线程工厂  
final UncaughtExceptionHandler ueh; // per-worker UEH 异常信息  
final String workerNamePrefix; // to create worker name string 用于创建工作线程的名称  
volatile AtomicLong stealCounter; // also used as sync monitor 偷取任务总数

//线程工厂  
public static final ForkJoinWorkerThreadFactory  
 *defaultForkJoinWorkerThreadFactory*;

//启动或杀死线程的方法调用者的权限  
private static final RuntimePermission *modifyThreadPermission*;

// 公共静态pool  
static final ForkJoinPool *common*;

//并行度，对应内部common池  
static final int *commonParallelism*;

//备用线程数，在tryCompensate中使用  
private static int *commonMaxSpares*;

//创建workerNamePrefix(工作线程名称前缀)时的序号  
private static int *poolNumberSequence*;

//线程阻塞等待新的任务的超时值(以纳秒为单位)，默认2秒  
private static final long *IDLE\_TIMEOUT* = 2000L \* 1000L \* 1000L; // 2sec

//空闲超时时间，防止timer未命中  
private static final long *TIMEOUT\_SLOP* = 20L \* 1000L \* 1000L; // 20ms

//默认备用线程数  
private static final int *DEFAULT\_COMMON\_MAX\_SPARES* = 256;

//阻塞前自旋等待的次数  
private static final int *SPINS* = 0;

//indexSeed的增量  
private static final int *SEED\_INCREMENT* = 0x9e3779b9;

##### 函数列表

###### ForkJoinPool

//无参构造，parallelism=CPU数  
public ForkJoinPool() {  
 this(Math.*min*(*MAX\_CAP*, Runtime.*getRuntime*().availableProcessors()),  
 *defaultForkJoinWorkerThreadFactory*, null, false);  
}

public ForkJoinPool(int parallelism)

public ForkJoinPool(int parallelism,  
 ForkJoinWorkerThreadFactory factory,  
 UncaughtExceptionHandler handler,  
 boolean asyncMode)

//获取内部common实例  
public static ForkJoinPool commonPool()

//执行给定任务，返回任务合并结果  
public <T> T invoke(ForkJoinTask<T> task)

//异步执行  
public void execute(ForkJoinTask<?> task)

//继承线程池AbstractExecutorService的方法  
public void execute(Runnable task)

//执行给定任务，返回任务  
public <T> ForkJoinTask<T> submit(ForkJoinTask<T> task)

//继承线程池AbstractExecutorService的方法  
public <T> ForkJoinTask<T> submit(Callable<T> task)

public <T> ForkJoinTask<T> submit(Runnable task, T result)

public ForkJoinTask<?> submit(Runnable task)

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

*/\*\*如果在当前pool中被一个ForkJoinTask操作调用，等价于ForkJoinTask.helpQuiesce  
 \* 否则的话，等待或尝试帮助执行中的任务直到当前pool中所有的工作线程都已经空闲  
 \*/*public boolean awaitQuiescence(long timeout, TimeUnit unit)

//运行给定可能会被阻塞的任务  
public static void managedBlock(ManagedBlocker blocker)

###### FrokJoinTask

//拆分任务  
public final ForkJoinTask<V> fork()

//合并任务  
public final V join()

//执行任务  
public final V invoke()

//拆分给定任务  
public static void invokeAll(ForkJoinTask<?> t1, ForkJoinTask<?> t2)

public static void invokeAll(ForkJoinTask<?>... tasks)

public static <T extends ForkJoinTask<?>> Collection<T> invokeAll(Collection<T> tasks)

//等待任务执行完成返回结果  
public final V get()

public final V get(long timeout, TimeUnit unit)

##### 源码解析

###### \*外部任务提交流程

**上面已经说过，ForkJoinPool中的任务分两种：一种是直接通过FJP提交的任务（external/submissions task），存放在偶数索引位置；还有一种是通过内部fork分割的子任务（worker），存放在奇数索引位。首先介绍一下外部任务提交流程：**

###### invoke(ForkJoinTask<T>)

//执行给定任务,返回任务合并结果  
public <T> T invoke(ForkJoinTask<T> task) {  
 if (task == null)  
 throw new NullPointerException();  
 externalPush(task);  
 return task.join();  
}

ForkJoinPool的提交任务方法有invoke()、execute()、submit()，内部都是调用externalPush()将给定任务添加到提交者队列中，源码如下：

externalPush(ForkJoinTask<?>)

//尝试添加给定任务到发送者线程的队列中  
final void externalPush(ForkJoinTask<?> task) {  
 WorkQueue[] ws; WorkQueue q; int m;  
 int r = ThreadLocalRandom.*getProbe*();//随机值  
 int rs = runState;  
 if ((ws = workQueues) != null && (m = (ws.length - 1)) >= 0 &&  
 (q = ws[m & r & *SQMASK*]) != null && r != 0 && rs > 0 &&  
 *U*.compareAndSwapInt(q, *QLOCK*, 0, 1)) {//锁定workQueue  
 ForkJoinTask<?>[] a; int am, n, s;  
 if ((a = q.array) != null &&  
 (am = a.length - 1) > (n = (s = q.top) - q.base)) {  
 int j = ((am & s) << *ASHIFT*) + *ABASE*;//找到合适位置  
 *U*.putOrderedObject(a, j, task);//任务入列  
 *U*.putOrderedInt(q, *QTOP*, s + 1);//更新push slot  
 *U*.putIntVolatile(q, *QLOCK*, 0);//解除锁定  
 if (n <= 1)  
 signalWork(ws, q);//创建或激活一个工作线程  
 return;  
 }  
 *U*.compareAndSwapInt(q, *QLOCK*, 1, 0);//解除锁定  
 }  
 externalSubmit(task);//提交任务  
}

如果活跃工作线程太少（n<=1），则调用signalWork创建或激活一个新的线程，这个后面再详细说明。在externalPush的最后调用了externalSubmit来提交任务，事实上大多数提交任务只需要执行externalPush的代码，除了第一次提交和少数workQueue还未初始化的情况。externalSubmit源码如下：

externalSubmit(ForkJoinTask<?>)

//任务提交  
private void externalSubmit(ForkJoinTask<?> task) {  
 //初始化调用线程的探针值  
 int r; // initialize caller's probe  
 if ((r = ThreadLocalRandom.*getProbe*()) == 0) {  
 ThreadLocalRandom.*localInit*();  
 r = ThreadLocalRandom.*getProbe*();  
 }  
 for (;;) {  
 WorkQueue[] ws; WorkQueue q; int rs, m, k;  
 boolean move = false;  
 if ((rs = runState) < 0) {  
 tryTerminate(false, false); // help terminate  
 throw new RejectedExecutionException();  
 }  
 //判断是否初始化  
 else if ((rs & *STARTED*) == 0 || // initialize  
 ((ws = workQueues) == null || (m = ws.length - 1) < 0)) {  
 int ns = 0;  
 rs = lockRunState();//锁定runState  
 try {  
 //初始化  
 if ((rs & *STARTED*) == 0) {  
 //初始化stealCounter  
 *U*.compareAndSwapObject(this, *STEALCOUNTER*, null,  
 new AtomicLong());  
 //创建workQueues  
 // create workQueues array with size a power of two  
 int p = config & *SMASK*; // ensure at least 2 slots  
 int n = (p > 1) ? p - 1 : 1;  
 n |= n >>> 1; n |= n >>> 2; n |= n >>> 4;  
 n |= n >>> 8; n |= n >>> 16; n = (n + 1) << 1;  
 workQueues = new WorkQueue[n];  
 ns = *STARTED*;  
 }  
 } finally {  
 unlockRunState(rs, (rs & ~*RSLOCK*) | ns);//解锁runState  
 }  
 }  
 else if ((q = ws[k = r & m & *SQMASK*]) != null) {//已经初始化，获取随机workQueue  
 if (q.qlock == 0 && *U*.compareAndSwapInt(q, *QLOCK*, 0, 1)) {//锁定workQueue  
 ForkJoinTask<?>[] a = q.array;  
 int s = q.top;  
 boolean submitted = false; // initial submission or resizing  
 try { // locked version of push  
 if ((a != null && a.length > s + 1 - q.base) ||  
 (a = q.growArray()) != null) {//扩容  
 int j = (((a.length - 1) & s) << *ASHIFT*) + *ABASE*;  
 *U*.putOrderedObject(a, j, task);//放入给定任务  
 *U*.putOrderedInt(q, *QTOP*, s + 1);//修改push slot  
 submitted = true;  
 }  
 } finally {  
 *U*.compareAndSwapInt(q, *QLOCK*, 1, 0);//解除锁定  
 }  
 if (submitted) {//任务提交成功，创建或激活工作线程  
 signalWork(ws, q);//创建或激活一个工作线程  
 return;  
 }  
 }  
 move = true; // move on failure  
 }  
 else if (((rs = runState) & *RSLOCK*) == 0) { // create new queue  
 q = new WorkQueue(this, null);  
 q.hint = r;  
 q.config = k | *SHARED\_QUEUE*;  
 q.scanState = *INACTIVE*;  
 rs = lockRunState(); // publish index  
 if (rs > 0 && (ws = workQueues) != null &&  
 k < ws.length && ws[k] == null)  
 ws[k] = q; // else terminated  
 unlockRunState(rs, rs & ~*RSLOCK*);  
 }  
 else  
 move = true; // move if busy  
 if (move)  
 r = ThreadLocalRandom.*advanceProbe*(r);//重新获取线程探针值  
 }  
}

externalSubmit可以说是externalPush的完全版本，处理一些非常规情况，例如第一次提交任务或workQueue尚未初始化完成。具体执行步骤如下（假设为单线程状态）：

1. 如果池为终止状态(runState<0)，调用tryTerminate来帮助终止，并抛出拒绝异常。
2. 如果尚未初始化，就为FJP执行初始化操作：初始化stealCounter、创建workerQueues，继续自旋
3. 初始化完成后，执行在externalPush中相同的操作：获取workQueue，放入指定任务。任务提交成功后调用signalWork方法创建或激活线程
4. 如果workQueue在步骤3中获取workQueue中锁失败，会在这一步中重新创建一个workQueue，创建成功继续自旋执行第三步操作。
5. 如果非上述情况，就重新获取线程探针值继续自旋。

###### signalWork(WorkQueue[],WorkQueue)

//如果活跃工作线程数很少，尝试创建或激活一个工作线程  
final void signalWork(WorkQueue[] ws, WorkQueue q) {  
 long c; int sp, i; WorkQueue v; Thread p;  
 while ((c = ctl) < 0L) { // too few active  
 if ((sp = (int)c) == 0) { // no idle workers  
 if ((c & *ADD\_WORKER*) != 0L) // too few workers  
 tryAddWorker(c);//添加工作线程  
 break;  
 }  
 if (ws == null) // unstarted/terminated  
 break;  
 if (ws.length <= (i = sp & *SMASK*)) // terminated  
 break;  
 if ((v = ws[i]) == null) // terminating  
 break;  
 int vs = (sp + *SS\_SEQ*) & ~*INACTIVE*; // next scanState  
 int d = sp - v.scanState; // screen CAS  
 long nc = (*UC\_MASK* & (c + *AC\_UNIT*)) | (*SP\_MASK* & v.stackPred);  
 if (d == 0 && *U*.compareAndSwapLong(this, *CTL*, c, nc)) {  
 v.scanState = vs; // activate v  
 if ((p = v.parker) != null)  
 *U*.unpark(p);//唤醒线程  
 break;  
 }  
 if (q != null && q.base == q.top) // no more work  
 break;  
 }  
}

新建或唤醒一个工作线程，在externalPush、externalSubmit、workQueue.push、scan中调用，如果工作线程过少，调用tryAddWorker添加一个新的工作线程，源码如下：

//添加新的工作线程  
private void tryAddWorker(long c) {  
 boolean add = false;  
 do {  
 long nc = ((*AC\_MASK* & (c + *AC\_UNIT*)) |  
 (*TC\_MASK* & (c + *TC\_UNIT*)));  
 if (ctl == c) {  
 int rs, stop; // check if terminating  
 if ((stop = (rs = lockRunState()) & *STOP*) == 0)  
 add = *U*.compareAndSwapLong(this, *CTL*, c, nc);  
 unlockRunState(rs, rs & ~*RSLOCK*);//释放锁  
 if (stop != 0)  
 break;  
 if (add) {  
 createWorker();//创建工作线程  
 break;  
 }  
 }  
 } while (((c = ctl) & *ADD\_WORKER*) != 0L && (int)c == 0);  
}

尝试添加一个新的工作线程，首先增加ctl值，然后调用createWorker创建新的工作线程，createWorker源码如下：

//尝试创建并开启一个新的工作线程  
private boolean createWorker() {  
 ForkJoinWorkerThreadFactory fac = factory;  
 Throwable ex = null;  
 ForkJoinWorkerThread wt = null;  
 try {  
 if (fac != null && (wt = fac.newThread(this)) != null) {  
 wt.start();  
 return true;  
 }  
 } catch (Throwable rex) {  
 ex = rex;  
 }  
 deregisterWorker(wt, ex);//线程创建失败处理  
 return false;  
}

createWorker的作用就是创建并启动一个新的工作线程，如果期间发生异常，调用deregisterWorker处理线程创建失败的逻辑。

###### deregisterWorker(ForkJoinWorkerThread, Throwable)

*/ /\*\*  
 \* 销毁工作线程的回调函数或创建工作线程异常时调用  
 \* 移除数组中工作线程记录，并调整线程数  
 \*/*final void deregisterWorker(ForkJoinWorkerThread wt, Throwable ex) {  
 WorkQueue w = null;  
 //1.移除workQueue  
 if (wt != null && (w = wt.workQueue) != null) {//获取ForkJoinWorkerThread的等待队列  
 WorkQueue[] ws; // remove index from array  
 int idx = w.config & *SMASK*;//计算workQueue索引  
 int rs = lockRunState();//获取runState锁和当前池运行状态  
 if ((ws = workQueues) != null && ws.length > idx && ws[idx] == w)  
 ws[idx] = null;//移除workQueue  
 unlockRunState(rs, rs & ~*RSLOCK*);//解除runState锁  
 }  
 //2.减少CTL数  
 long c; // decrement counts  
 do {} while (!*U*.compareAndSwapLong  
 (this, *CTL*, c = ctl, ((*AC\_MASK* & (c - *AC\_UNIT*)) |  
 (*TC\_MASK* & (c - *TC\_UNIT*)) |  
 (*SP\_MASK* & c))));  
 //3.处理被移除workQueue内部相关参数  
 if (w != null) {  
 w.qlock = -1; // ensure set  
 w.transferStealCount(this);  
 w.cancelAll(); // cancel remaining tasks  
 }  
 //4.如果线程未终止，替换被移除的workQueue并唤醒内部线程  
 for (;;) { // possibly replace  
 WorkQueue[] ws; int m, sp;  
 //尝试终止线程池  
 if (tryTerminate(false, false) || w == null || w.array == null ||  
 (runState & *STOP*) != 0 || (ws = workQueues) == null ||  
 (m = ws.length - 1) < 0) // already terminating  
 break;  
 //唤醒被替换的线程，依赖于下一步  
 if ((sp = (int)(c = ctl)) != 0) { // wake up replacement  
 if (tryRelease(c, ws[sp & m], *AC\_UNIT*))  
 break;  
 }  
 //创建工作线程替换  
 else if (ex != null && (c & *ADD\_WORKER*) != 0L) {  
 tryAddWorker(c); // create replacement  
 break;  
 }  
 else // don't need replacement  
 break;  
 }  
 //5.处理异常  
 if (ex == null) // help clean on way out  
 ForkJoinTask.*helpExpungeStaleExceptions*();  
 else // rethrow  
 ForkJoinTask.*rethrow*(ex);  
}

**说明：**在创建工作线程异常时调用，用于回滚创建线程之前的操作：移除数组中的工作线程，并调整线程数。

上面介绍了ForkJoinPool中提交任务的相关源码分析，下面将结合**ForkJoinPool、ForkJoinTask、ForkJoinWorkerThread**三个类来分析任务的执行流程

###### \*内部任务运行流程-run()



这里主要针对一些比较复杂的方法进行分析。

###### scan(WorkQueue,random)

*/\*\*  
 \* 扫描并尝试偷取一个最顶层任务。  
 \*/*private ForkJoinTask<?> scan(WorkQueue w, int r) {  
 WorkQueue[] ws;  
 int m;  
 if ((ws = workQueues) != null && (m = ws.length - 1) > 0 && w != null) {  
 int ss = w.scanState; // initially non-negative  
 for (int origin = r & m, k = origin, oldSum = 0, checkSum = 0; ; ) {  
 WorkQueue q;  
 ForkJoinTask<?>[] a;  
 ForkJoinTask<?> t;  
 int b, n;  
 long c;  
 if ((q = ws[k]) != null) {//取随机位置的workQueue  
 if ((n = (b = q.base) - q.top) < 0 &&  
 (a = q.array) != null) { // non-empty  
 //计算偏移量，取最顶任务  
 long i = (((a.length - 1) & b) << *ASHIFT*) + *ABASE*;  
 if ((t = ((ForkJoinTask<?>)  
 *U*.getObjectVolatile(a, i))) != null && //取base位置任务  
 q.base == b) {  
 if (ss >= 0) { //scanning  
 if (*U*.compareAndSwapObject(a, i, t, null)) {  
 q.base = b + 1;//更新next poll位置  
 if (n < -1) // signal others  
 signalWork(ws, q);//无可用任务，唤醒其他线程  
 return t;  
 }  
 } else if (oldSum == 0 && // try to activate  
 w.scanState < 0)  
 tryRelease(c = ctl, ws[m & (int) c], *AC\_UNIT*);  
 }  
 //base位置任务为空或base位置偏移，移位重新扫描  
 if (ss < 0) // refresh  
 ss = w.scanState;  
 r ^= r << 1;  
 r ^= r >>> 3;  
 r ^= r << 10;  
 origin = k = r & m; // move and rescan  
 oldSum = checkSum = 0;  
 continue;  
 }  
 checkSum += b;  
 }  
 //未扫描到任务  
 if ((k = (k + 1) & m) == origin) { // continue until stable  
 if ((ss >= 0 || (ss == (ss = w.scanState))) &&  
 oldSum == (oldSum = checkSum)) {  
 if (ss < 0 || w.qlock < 0) // already inactive  
 break;  
 int ns = ss | *INACTIVE*; // try to inactivate  
 long nc = ((*SP\_MASK* & ns) |  
 (*UC\_MASK* & ((c = ctl) - *AC\_UNIT*)));  
 w.stackPred = (int) c; // hold prev stack top  
 *U*.putInt(w, *QSCANSTATE*, ns);//修改scanState为inactive状态  
 if (*U*.compareAndSwapLong(this, *CTL*, c, nc))  
 ss = ns;  
 else  
 w.scanState = ss; // back out  
 }  
 checkSum = 0;  
 }  
 }  
 }  
 return null;  
}

**说明：**scan方法用来获取队列中某一个任务，具体步骤如下：

1. 取随机位置的一个workQueue
2. 获取base位置的ForkJoinTask，成功取到返回；如果取到的workQueue中没有可用任务，则调用signalWork新增或唤醒一个工作线程
3. 如果传入的workQueue处于不活跃状态，尝试唤醒一个工作线程
4. 如果base位置为空或偏移，则移位重新扫描
5. 如果扫描整个workQueues之后没有获取到任务，则设置状态为inactive，继续从初始位置循环
6. 再次扫描一圈之后还没有任务则跳出循环返回null

###### awaitWork(WorkQueue,random)

*/\*\*可能阻塞给定任务队列等待另一个任务的偷取，如果工作线程需要终止返回false  
 \* 如果由于失活的工作线程导致池变成静止状态，就检查是否可以终止线程池。  
 \* 只要它不是唯一的工作线程，就等待一个跟定的时间，等待期满后，如果ctl尚未  
 \* 改变就终止这个工作线程，并依次唤醒其他工作线程重复这个操作  
 \*/*private boolean awaitWork(WorkQueue w, int r) {  
 if (w == null || w.qlock < 0) // w is terminating  
 return false;  
 for (int pred = w.stackPred, spins = *SPINS*, ss; ; ) {  
 if ((ss = w.scanState) >= 0)//正在扫描，跳出循环  
 break;  
 else if (spins > 0) {  
 r ^= r << 6;  
 r ^= r >>> 21;  
 r ^= r << 7;  
 if (r >= 0 && --spins == 0) { // randomize spins  
 WorkQueue v;  
 WorkQueue[] ws;  
 int s, j;  
 AtomicLong sc;  
 if (pred != 0 && (ws = workQueues) != null &&  
 (j = pred & *SMASK*) < ws.length &&  
 (v = ws[j]) != null && // see if pred parking  
 (v.parker == null || v.scanState >= 0))  
 spins = *SPINS*; // continue spinning  
 }  
 } else if (w.qlock < 0) // recheck after spins  
 return false;  
 else if (!Thread.*interrupted*()) {//清除中断状态  
 long c, prevctl, parkTime, deadline;  
 int ac = (int) ((c = ctl) >> *AC\_SHIFT*) + (config & *SMASK*);//活跃线程数  
 if ((ac <= 0 && tryTerminate(false, false)) || //无active线程，尝试终止  
 (runState & *STOP*) != 0) // pool terminating  
 return false;  
 if (ac <= 0 && ss == (int) c) { // is last waiter  
 prevctl = (*UC\_MASK* & (c + *AC\_UNIT*)) | (*SP\_MASK* & pred);//获取前一个ctl  
 int t = (short) (c >>> *TC\_SHIFT*); // shrink excess spares  
 if (t > 2 && *U*.compareAndSwapLong(this, *CTL*, c, prevctl))//线程过量，缩减多余线程数  
 return false; // else use timed wait  
 parkTime = *IDLE\_TIMEOUT* \* ((t >= 0) ? 1 : 1 - t);  
 deadline = System.*nanoTime*() + parkTime - *TIMEOUT\_SLOP*;  
 } else  
 prevctl = parkTime = deadline = 0L;  
 Thread wt = Thread.*currentThread*();  
 *U*.putObject(wt, *PARKBLOCKER*, this); // emulate LockSupport  
 w.parker = wt;  
 if (w.scanState < 0 && ctl == c) // recheck before park  
 *U*.park(false, parkTime);//阻塞当前线程  
 *U*.putOrderedObject(w, *QPARKER*, null);  
 *U*.putObject(wt, *PARKBLOCKER*, null);  
 if (w.scanState >= 0)  
 break;  
 if (parkTime != 0L && ctl == c &&  
 deadline - System.*nanoTime*() <= 0L &&  
 *U*.compareAndSwapLong(this, *CTL*, c, prevctl))  
 return false; // shrink pool  
 }  
 }  
 return true;  
}

**说明：**如果没有扫描到任务，则调用waitWork阻塞给定队列等待其他线程偷取，如果工作线程需要销毁则返回false。如果当前无active线程，尝试终止线程池，只要当前不是唯一的工作线程，就等待一个给定的时间，等到时间过后线程唤醒如果发现自己是active状态，则跳出循环返回继续scan，如果是inactive状态销毁工作线程，然后依次唤醒其他工作线程继续执行这个操作。

###### ForkJoinTask.fork()

//拆分任务  
public final ForkJoinTask<V> fork() {  
 Thread t;  
 if ((t = Thread.*currentThread*()) instanceof ForkJoinWorkerThread)  
 ((ForkJoinWorkerThread)t).workQueue.push(this);  
 else  
 ForkJoinPool.*common*.externalPush(this);  
 return this;  
}

**说明：**如果当前线程是work线程，就直接放到自己的等待队列中，否则调用ForkJoinPool.externalPush提交到一个随机的等待队列中。

###### ForkJoinTask.join()/invoke()

//合并任务  
public final V join() {  
 int s;  
 if ((s = doJoin() & *DONE\_MASK*) != *NORMAL*)  
 reportException(s);  
 return getRawResult();  
}

**说明：**如果isDone为true，等待执行完成返回结果，否则抛出错误信息。调用了doJoin()方法计算返回结果，源码如下：

private int doJoin() {  
 int s; Thread t; ForkJoinWorkerThread wt; ForkJoinPool.WorkQueue w;  
 return (s = status) < 0 ? s :  
 ((t = Thread.*currentThread*()) instanceof ForkJoinWorkerThread) ?  
 (w = (wt = (ForkJoinWorkerThread)t).workQueue).  
 tryUnpush(this) && (s = doExec()) < 0 ? s :  
 wt.pool.awaitJoin(w, this, 0L) :  
 externalAwaitDone();  
}

如果当前线程是工作线程：  
首先调用tryUnpush判断当前task是否队列顶部，如果是，弹出给定任务并返回true；然后执行任务。  
如果当前任务不在队列顶部，或执行任务失败，则调用ForkJoinPool.awaitJoin帮助执行或阻塞任务。

如果当前线程不是工作线程，调用externalAwaitDone()阻塞none-Work-Thread直到执行完毕。

//执行任务  
public final V invoke() {  
 int s;  
 if ((s = doInvoke() & *DONE\_MASK*) != *NORMAL*)  
 reportException(s);  
 return getRawResult();  
}

private int doInvoke() {  
 int s; Thread t; ForkJoinWorkerThread wt;  
 return (s = doExec()) < 0 ? s :  
 ((t = Thread.*currentThread*()) instanceof ForkJoinWorkerThread) ?  
 (wt = (ForkJoinWorkerThread)t).pool.  
 awaitJoin(wt.workQueue, this, 0L) :  
 externalAwaitDone();  
}

invoke()与join()不同的是:invoke会先执行任务，如果执行失败才会进行awaitJoin或externalAwaitDone操作。而join需要先判断任务是否在队列顶部。

###### awaitJoin(WorkerQueue, ForkJoinTask, deadline)

//帮助执行或阻塞给定的join任务  
final int awaitJoin(WorkQueue w, ForkJoinTask<?> task, long deadline) {  
 int s = 0;  
 if (task != null && w != null) {  
 ForkJoinTask<?> prevJoin = w.currentJoin;//获取当前join的任务  
 *U*.putOrderedObject(w, *QCURRENTJOIN*, task);//替换join任务  
 CountedCompleter<?> cc = (task instanceof CountedCompleter) ?  
 (CountedCompleter<?>) task : null;  
 for (; ; ) {  
 if ((s = task.status) < 0)//已经完成|取消|异常 退出循环  
 break;  
  
 if (cc != null)  
 helpComplete(w, cc, 0);  
 else if (w.base == w.top || w.tryRemoveAndExec(task))//尝试执行  
 helpStealer(w, task);//队列为空或执行失败，任务可能被偷，帮助偷取者执行该任务  
  
 if ((s = task.status) < 0)  
 break;  
  
 long ms, ns;  
 if (deadline == 0L)  
 ms = 0L;  
 else if ((ns = deadline - System.*nanoTime*()) <= 0L)  
 break;  
 else if ((ms = TimeUnit.*NANOSECONDS*.toMillis(ns)) <= 0L)  
 ms = 1L;  
  
 if (tryCompensate(w)) {//执行补偿操作  
 task.internalWait(ms);//补偿执行成功，任务等待指定时间  
 *U*.getAndAddLong(this, *CTL*, *AC\_UNIT*);//还原活跃线程数  
 }  
 }  
 *U*.putOrderedObject(w, *QCURRENTJOIN*, prevJoin);//循环结束，替换为原来的join任务  
 }  
 return s;  
}

**说明：**在调用join或invoke时如果执行任务失败，就会调用awaitJoin方法帮助或阻塞任务的执行。并且每次awaitJoin都会挤掉原来的currentJoin。如果在此任务执行不成功，说明任务可能被偷取，调用helpStealer()方法帮助偷取者执行该任务，helpStealer源码如下：

* + - * 1. helpStealer(WorkQueue, ForkJoinTask)

//帮助 给定任务的偷取者 定位并执行任务（偷取者偷我的任务，我去偷偷取者的任务）  
private void helpStealer(WorkQueue w, ForkJoinTask<?> task) {  
 WorkQueue[] ws = workQueues;  
 int oldSum = 0, checkSum, m;  
 if (ws != null && (m = ws.length - 1) >= 0 && w != null &&  
 task != null) {  
 do { // restart point  
 checkSum = 0; // for stability check  
 ForkJoinTask<?> subtask;  
 WorkQueue j = w, v; // v is subtask stealer  
 descent:  
 for (subtask = task; subtask.status >= 0; ) {  
 //找到给定task的偷取者v，修改hint  
 for (int h = j.hint | 1, k = 0, i; ; k += 2) {  
 if (k > m) // can't find stealer  
 break descent;  
 if ((v = ws[i = (h + k) & m]) != null) {//定位偷取者  
 if (v.currentSteal == subtask) {  
 j.hint = i;//更新stealer索引  
 break;  
 }  
 checkSum += v.base;  
 }  
 }  
 //帮助偷取者v执行任务  
 for (; ; ) { // help v or descend  
 ForkJoinTask<?>[] a; //偷取者内部的任务  
 int b;  
 checkSum += (b = v.base);  
 ForkJoinTask<?> next = v.currentJoin;//获取偷取者的join任务  
 if (subtask.status < 0 || j.currentJoin != subtask ||  
 v.currentSteal != subtask) // stale  
 break descent;  
 if (b - v.top >= 0 || (a = v.array) == null) {  
 if ((subtask = next) == null) //偷取者的join任务为null，跳出descent循环  
 break descent;  
 j = v;  
 break; //偷取者内部任务为空，可能任务也被偷走了；跳出本次循环，查找偷取者的偷取者  
 }  
 int i = (((a.length - 1) & b) << *ASHIFT*) + *ABASE*;//获取base偏移地址  
 ForkJoinTask<?> t = ((ForkJoinTask<?>)  
 *U*.getObjectVolatile(a, i));//获取偷取者的base任务  
 if (v.base == b) {  
 if (t == null) // stale  
 break descent;  
 if (*U*.compareAndSwapObject(a, i, t, null)) {  
 v.base = b + 1; //更新base  
 ForkJoinTask<?> ps = w.currentSteal;//获取给定workQueue的偷来的任务  
 int top = w.top;  
 //首先更新给定workQueue的currentSteal为偷取者的base任务，然后执行该任务  
 //然后依次弹出给定workQueue的任务(LIFO)->更新currentSteal->执行该任务（注意这里是自己偷自己的任务执行）  
 do {  
 *U*.putOrderedObject(w, *QCURRENTSTEAL*, t);  
 t.doExec(); // clear local tasks too  
 } while (task.status >= 0 &&  
 w.top != top &&  
 (t = w.pop()) != null);  
 *U*.putOrderedObject(w, *QCURRENTSTEAL*, ps);//还原给定workQueue的currentSteal  
 if (w.base != w.top)  
 return; // can't further help  
 }  
 }  
 }  
 }  
 } while (task.status >= 0 && oldSum != (oldSum = checkSum));  
 }  
}

helpStealer的作用就是帮助偷取者执行任务：偷取者帮我执行任务，我去帮偷取者执行任务。首先会根据hint定位到偷取者，执行偷取者base任务(FIFO)，然后依次弹出给定WorkQueue(调用者)的任务(LIFO)执行。

回到awaitJoin方法，如果帮助执行失败（可能joiner被阻塞），会调用tryCompensate执行补偿操作。如果需要补偿，则唤醒或新建一个工作线程准备为join任务执行补偿运行。tryCompensate源码如下：

* + - * 1. tryCompensate (WorkQueue)

//执行补偿操作：尝试缩减活动线程量，可能释放或创建一个补偿线程来准备阻塞  
private boolean tryCompensate(WorkQueue w) {  
 boolean canBlock;  
 WorkQueue[] ws;  
 long c;  
 int m, pc, sp;  
 if (w == null || w.qlock < 0 || // caller terminating  
 (ws = workQueues) == null || (m = ws.length - 1) <= 0 ||  
 (pc = config & *SMASK*) == 0) // parallelism disabled  
 canBlock = false;  
 else if ((sp = (int) (c = ctl)) != 0) // release idle worker  
 canBlock = tryRelease(c, ws[sp & m], 0L);//唤醒空闲工作线程  
 else {//没有空闲线程  
 int ac = (int) (c >> *AC\_SHIFT*) + pc; //活跃线程数  
 int tc = (short) (c >> *TC\_SHIFT*) + pc;//总线程数  
 int nbusy = 0; // validate saturation  
 for (int i = 0; i <= m; ++i) { //奇数索引，存放worker任务 two passes of odd indices  
 WorkQueue v;  
 if ((v = ws[((i << 1) | 1) & m]) != null) {  
 if ((v.scanState & *SCANNING*) != 0)//正在扫描，跳出循环  
 break;  
 ++nbusy;  
 }  
 }  
 if (nbusy != (tc << 1) || ctl != c)  
 canBlock = false; // unstable or stale  
 else if (tc >= pc && ac > 1 && w.isEmpty()) {//总线程数大于并行度 & 活动线程数大于1 & 队列为空，不需要补偿  
 long nc = ((*AC\_MASK* & (c - *AC\_UNIT*)) |  
 (~*AC\_MASK* & c)); // uncompensated  
 canBlock = *U*.compareAndSwapLong(this, *CTL*, c, nc);//更新活跃线程数  
 } else if (tc >= *MAX\_CAP* ||  
 (this == *common* && tc >= pc + *commonMaxSpares*))//超出最大线程数  
 throw new RejectedExecutionException(  
 "Thread limit exceeded replacing blocked worker");  
 else { // similar to tryAddWorker  
 boolean add = false;  
 int rs; // CAS within lock  
 long nc = ((*AC\_MASK* & c) |  
 (*TC\_MASK* & (c + *TC\_UNIT*)));  
 if (((rs = lockRunState()) & *STOP*) == 0)//池尚未停止，更新ctl  
 add = *U*.compareAndSwapLong(this, *CTL*, c, nc);  
 unlockRunState(rs, rs & ~*RSLOCK*);  
 //运行到这里说明活跃工作线程数不足，需要创建一个新的工作线程  
 canBlock = add && createWorker(); // throws on exception  
 }  
 }  
 return canBlock;  
}

**说明：**这里总结一下需要和不需要补偿的情况：

1. 需要补偿
   1. 调用者队列不为空，并且有空闲工作线程，这时会唤醒空闲线程
   2. 池尚未停止，活跃线程数不足，这时会新建一个工作线程
2. 不需要补偿
   1. 池已终止
   2. 总线程数大于并行度 & 活动线程数大于1 & 队列为空

##### 小结

自此，JUC包下比较重要的类都已分析完毕。总的来说，FrokJoinPool的源码还是相对比较难的，建议大家多看下官方文档先理解本类的设计思想，然后再分析源码。

**关键点：**

1. **work-stealing模式实现原理**
2. **分治算法**
3. **ForkJoinPool、ForkJoinWorkerThread、ForkJoinTask、RecursiveTask的使用**