CAS&AQS

一、CAS简介

> CAS （Compare And Swap）比较然后交换的意思，该操作具有原子性。通常来说一个CAS接收三个参数，数据的现值V，进行比较的值A，准备写入的值B。只有当V和A相等的时候，才会写入B。无论是否写入成功，都会返回V。翻译过来就是“我认为V现在的值是A，如果是那么将V的值更新为B，否则不修改V的值，并告诉我现在V的值是多少。”
>
> CAS是非阻塞的

非阻塞同步

非阻塞同步的意思是多个线程在竞争相同的数据时候不会发生阻塞，从而能够在更加细粒度的维度上进行协调，从而极大的减少线程调度的开销，从而提升效率。非阻塞算法不存在锁的机制也就不存在死锁的问题。

在基于锁的算法中，如果一个线程持有了锁，那么其他的线程将无法进行下去。使用锁虽然可以保证对资源的一致性访问，但是在挂起和恢复线程的执行过程中存在非常大的开销，如果锁上面存在着大量的竞争，那么有可能调度开销比实际工作开销还要高。

CAS实现源码追踪（如何查看源码）

java层面的实现

unsafe魔法类（后面有专题讲解）

```java
public final native boolean compareAndSwapInt(Object var1, long var2, int var4, int var5);
```

Hotspot c++实现 unsafe.cpp

```C++
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env, jobject unsafe, jobject obj, jlong offset, jint e, jint x))
  UnsafeWrapper("Unsafe_CompareAndSwapInt");
  oop p = JNIHandles::resolve(obj);
  jint* addr = (jint *) index_oop_from_field_offset_long(p, offset);
  return (jint)(Atomic::cmpxchg(x, addr, e)) == e;
UNSAFE_END

```

内联汇编语言 Atomic.cpp找到各大平台的实现（x86）

```Java

inline jint     Atomic::cmpxchg    (jint     exchange_value, volatile jint* dest, jint     compare_value) {
  int mp = os::is_MP();
  __asm__ volatile (LOCK_IF_MP(%4) "cmpxchgl %1,(%3)"
                    : "=a" (exchange_value)
                    : "r" (exchange_value), "a" (compare_value), "r" (dest), "r" (mp)
                    : "cc", "memory");
  return exchange_value;
}
```

1. 通过unsafe类查找到**native方法，可以发现通过jni调用c++的方法**
2. 通过在hotspot源码中搜索此方法，发现其中存在一个unsafe.cpp文件这个文件调用了Atomic::cmpxchg方法。
3. 进行跟踪发现cmpxchg有各大平台的实现我们以x86平台为例，可以看出在多处理器的情况下（os::is_MP）会使用内联汇编的方式使用lock指令去实现
4. Intel平台下 lock指令的说明
   - 确保对内存的读-改-写操作原子执行。在Pentium及Pentium之前的处理器中，带有lock前缀的指令在执行期间会锁住总线，使得其他处理器暂时无法通过总线访问内存。很显然，这会带来昂贵的开销。从Pentium 4，Intel Xeon及P6处理器开始，intel在原有总线锁的基础上做了一个很有意义的优化：如果要访问的内存区域（area of memory）在lock前缀指令执行期间已经在处理器内部的缓存中被锁定（即包含该内存区域的缓存行当前处于独占或以修改状态），并且该内存区域被完全包含在单个缓存行（cache line）中，那么处理器将直接执行该指令。由于在指令执行期间该缓

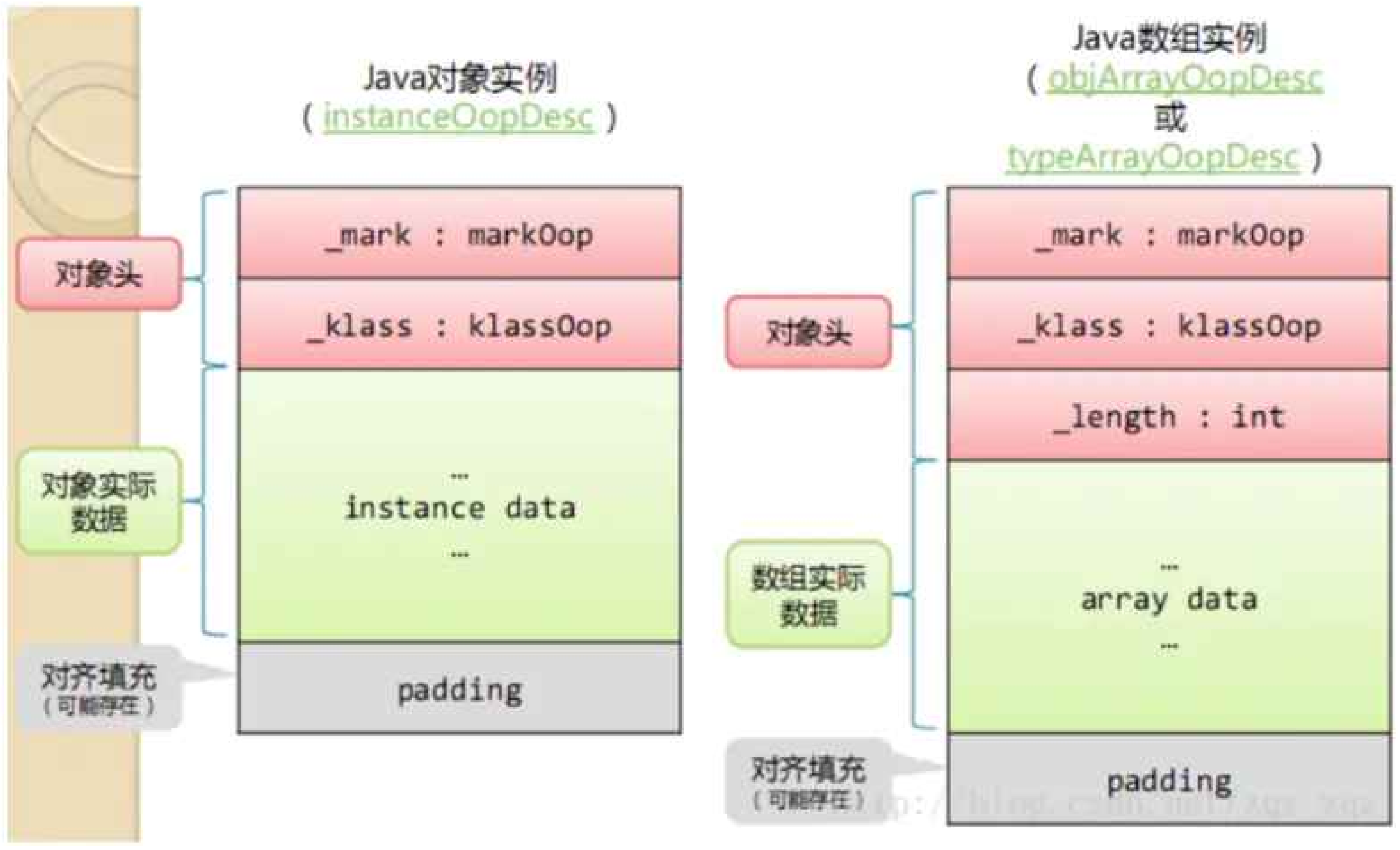
存行会一直被锁定，其它处理器无法读/写该指令要访问的内存区域，因此能保证指令执行的原子性。这个操作过程叫做缓存锁定（cache locking），缓存锁定将大大降低lock前缀指令的执行开销，但是当多处理器之间的竞争程度很高或者指令访问的内存地址未对齐时，仍然会锁住总线。

- 禁止该指令与之前和之后的读和写指令重排序。
- 把写缓冲区中的所有数据刷新到内存中。

二、UnSafe类

> 1. 获取当前对象的成员的偏移量 可以根据对象以及对应字段的偏移量获取最新的值
> 2. 提供CAS操作的方法
> 3. 线程的挂起与唤醒（LockSupport）
> 4. 其他,（后面专题讲）

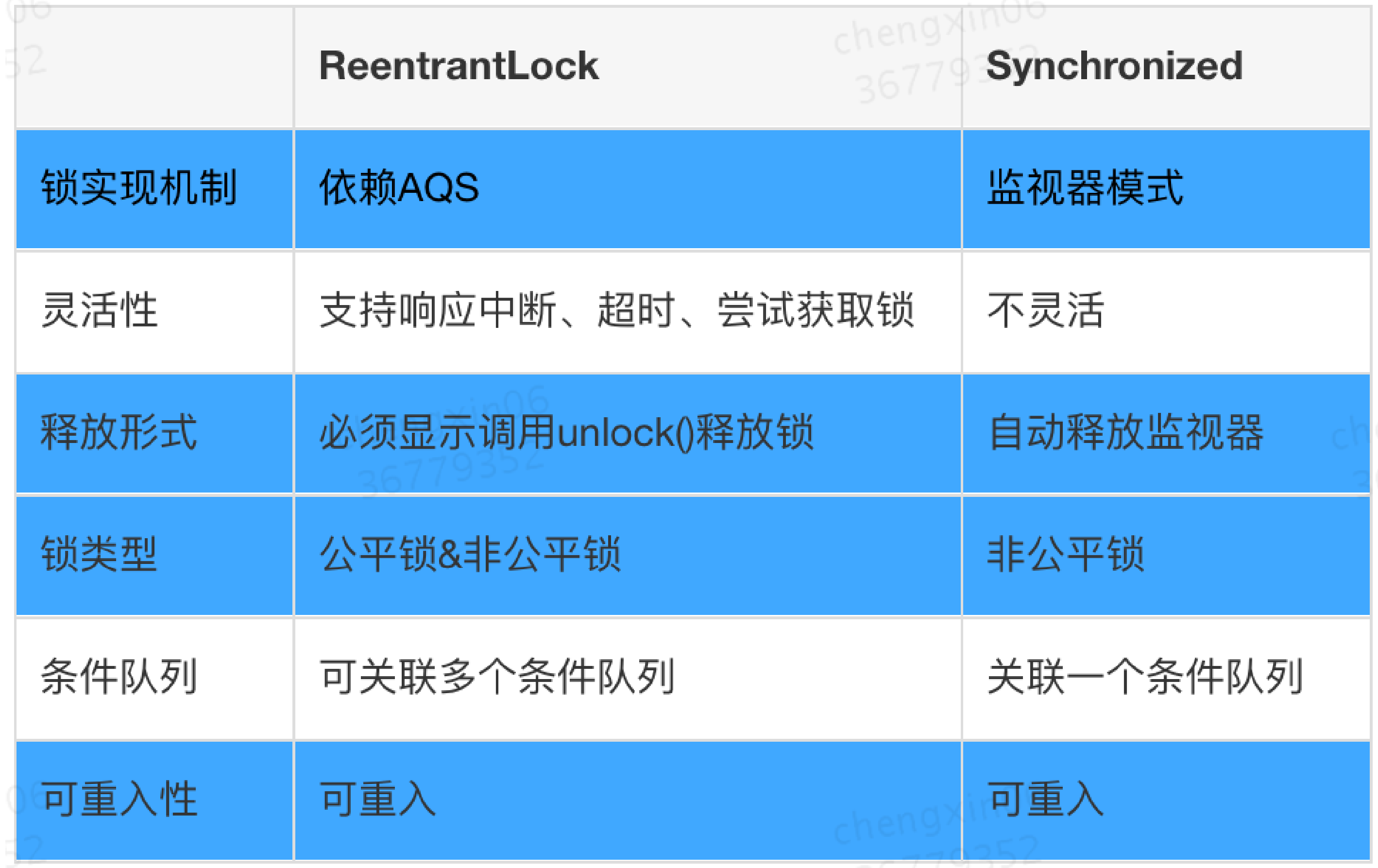
对象偏移量：



三、AQS

AQS的全称叫AbstractQueuedSynchronizer，是一个抽象队列同步器，java中ReentrantLock、Semaphore等都是由AQS来实现的。

AQS解决了什么问题?



	ReentrantLock	Synchronized
锁实现机制	依赖AQS	监视器模式
灵活性	支持响应中断、超时、尝试获取锁	不灵活
释放形式	必须显示调用unlock()释放锁	自动释放监视器
锁类型	公平锁&非公平锁	非公平锁
条件队列	可关联多个条件队列	关联一个条件队列
可重入性	可重入	可重入

AQS支持的几种模式

- 独占模式（exclusive mode）：同一时刻只允许一个线程访问共享资源，如`ReentrantLock`等
- 公平模式：获取锁失败的线程需要按照顺序排列，前面的先拿到锁
- 非公平模式： 当线程需要获取锁时，会尝试直接获取锁
- 共享模式（shared mode）：同一时刻允多个线程访问共享资源

Q1:如何线程安全的修改同步状态?

Q2:得不到资源的线程怎么处理？如何排队？

AQS的内部结构

同步状态

AQS中有个名为state的字段，代表同步状态，并提供了3个访问这个字段

Java

```java
protected final int getState() {return state;}
protected final void setState(int newState) {state = newState;}
protected final boolean compareAndSetState(int expect, int update) {
  //unsafe.compareAndSwapInt其实就是获取该对象中state在内存中的偏移量
  return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
}
//stateOffset是state在内存中的位置的偏移量
private static final long stateOffset;
static {
  try {
    stateOffset = unsafe.objectFieldOffset
      (AbstractQueuedSynchronizer.class.getDeclaredField("state"));
    headOffset = unsafe.objectFieldOffset
      (AbstractQueuedSynchronizer.class.getDeclaredField("head"));
    tailOffset = unsafe.objectFieldOffset
      (AbstractQueuedSynchronizer.class.getDeclaredField("tail"));
    waitStatusOffset = unsafe.objectFieldOffset
      (Node.class.getDeclaredField("waitStatus"));
    nextOffset = unsafe.objectFieldOffset
      (Node.class.getDeclaredField("next"));

  } catch (Exception ex) { throw new Error(ex); }
}
```

getState()和setState()都是普通的getter、setter方法，但是由于state是volatile修饰的，所以能够**保证可见性**，但是**不能保证原子性**，也就是说getState()只能获取state的相对新值，而不是最新值。

compareAndSetState()方法是**通过CAS来修改内存中state的值**，CAS能够**保证原子性**，state又是被volatile修饰的，能够**保证可见性和有序性**，故而是**线程安全的**。也就是说通过compareAndSetState()方法可以线程安全的修改同步状态、

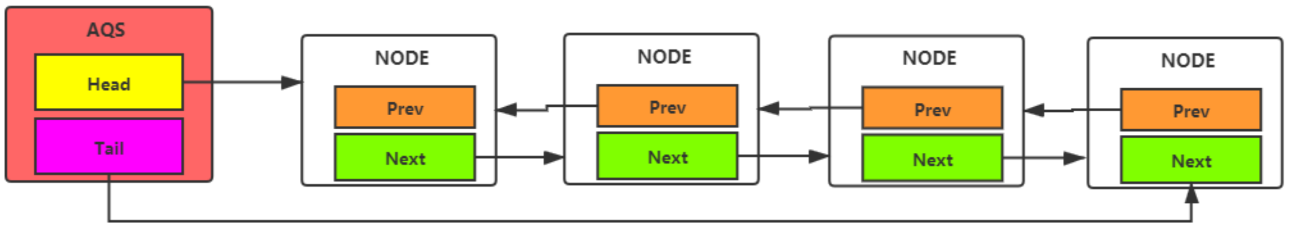
同步队列

Java

```java
static final class Node {
```

```java
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;

/** waitStatus value to indicate thread has cancelled */
static final int CANCELLED =  1;
/** waitStatus value to indicate successor's thread needs unparking */
static final int SIGNAL    = -1;
/** waitStatus value to indicate thread is waiting on condition */
static final int CONDITION = -2;
/**
     * waitStatus value to indicate the next acquireShared should
     * unconditionally propagate
     */
static final int PROPAGATE = -3;

/**
     * Status field, taking on only the values:
     *   SIGNAL:     The successor of this node is (or will soon be)
     *               blocked (via park), so the current node must
     *               unpark its successor when it releases or
     *               cancels. To avoid races, acquire methods must
     *               first indicate they need a signal,
     *               then retry the atomic acquire, and then,
     *               on failure, block.
     *   CANCELLED:  This node is cancelled due to timeout or interrupt.
     *               Nodes never leave this state. In particular,
     *               a thread with cancelled node never again blocks.
     *   CONDITION:  This node is currently on a condition queue.
     *               It will not be used as a sync queue node
     *               until transferred, at which time the status
     *               will be set to 0. (Use of this value here has
     *               nothing to do with the other uses of the
     *               field, but simplifies mechanics.)
     *   PROPAGATE:  A releaseShared should be propagated to other
     *               nodes. This is set (for head node only) in
     *               doReleaseShared to ensure propagation
     *               continues, even if other operations have
     *               since intervened.
     *   0:          None of the above
     *
     * The values are arranged numerically to simplify use.
     * Non-negative values mean that a node doesn't need to
     * signal. So, most code doesn't need to check for particular
     * values, just for sign.
     *
     * The field is initialized to 0 for normal sync nodes, and
     * CONDITION for condition nodes.  It is modified using CAS
     * (or when possible, unconditional volatile writes).
     */
volatile int waitStatus;

/**
     * Link to predecessor node that current node/thread relies on
     * for checking waitStatus. Assigned during enqueuing, and nulled
     * out (for sake of GC) only upon dequeuing.  Also, upon
     * cancellation of a predecessor, we short-circuit while
     * finding a non-cancelled one, which will always exist
     * because the head node is never cancelled: A node becomes
     * head only as a result of successful acquire. A
     * cancelled thread never succeeds in acquiring, and a thread only
     * cancels itself, not any other node.
     */
volatile Node prev;

/**
     * Link to the successor node that the current node/thread
     * unparks upon release. Assigned during enqueuing, adjusted
     * when bypassing cancelled predecessors, and nulled out (for
     * sake of GC) when dequeued.  The enq operation does not
     * assign next field of a predecessor until after attachment,
     * so seeing a null next field does not necessarily mean that
     * node is at end of queue. However, if a next field appears
     * to be null, we can scan prev's from the tail to
     * double-check.  The next field of cancelled nodes is set to
     * point to the node itself instead of null, to make life
     * easier for isOnSyncQueue.
     */
volatile Node next;

/**
     * The thread that enqueued this node.  Initialized on
     * construction and nulled out after use.
     */
volatile Thread thread;

/**
     * Link to next node waiting on condition, or the special
     * value SHARED.  Because condition queues are accessed only
     * when holding in exclusive mode, we just need a simple
     * linked queue to hold nodes while they are waiting on
     * conditions. They are then transferred to the queue to
     * re-acquire. And because conditions can only be exclusive,
     * we save a field by using special value to indicate shared
     * mode.
     */
Node nextWaiter;

/**
     * Returns true if node is waiting in shared mode.
     */
final boolean isShared() {
return nextWaiter == SHARED;
    }

/**
     * Returns previous node, or throws NullPointerException if null.
     * Use when predecessor cannot be null.  The null check could
     * be elided, but is present to help the VM.
     *
     * @return the predecessor of this node
     */
final Node predecessor() throws NullPointerException {
        Node p = prev;
if (p == null)
throw new NullPointerException();
else
            return p;
    }

    Node() {    // Used to establish initial head or SHARED marker
}

    Node(Thread thread, Node mode) {     // Used by addWaiter
this.nextWaiter = mode;
this.thread = thread;
    }

    Node(Thread thread, int waitStatus) { // Used by Condition
this.waitStatus = waitStatus;
this.thread = thread;
    }
```

```
}
```



同步队列结构

可以看出，该内部类是一个双向链表，保存前后节点，然后每个节点存储了当前的状态waitStatus、当前线程thread，还可以通过SHARED和EXCLUSIVE两个变量定义为共享模式或者独占模式，通过下面的方式：

```java
// 标识当前节点在共享模式
static final Node SHARED = new Node();
// 标识当前节点在独占模式
static final Node EXCLUSIVE = null;
```

然后定义了四个常量：

```java
CANCELLED，值为1，表示当前的线程被取消；
SIGNAL，值为-1，表示当前节点的后继节点包含的线程需要运行，也就是unpark；
CONDITION，值为-2，表示当前节点在等待condition，也就是在condition队列中；
PROPAGATE，值为-3，表示当前场景下后续的acquireShared能够得以执行；
默认值为0，表示当前节点在sync队列中，等待着获取锁。

waitStatus 表当前节点的状态值，取值为上面的四个常量。
```

独占模式

获取锁的过程

1. 调用acquire(**int** arg)，执行CAS操作尝试更新state状态
2. 如果更新成功说明获取到了锁，将当前线程为独占线程，继续处理相应的业务逻辑

3. 如果更新失败，则调用addWaiter(Node node)方法，创建独占Node，通过快速入队添加到队尾，快速入队失败后会执行enq(Node node)方法,自旋CAS,添加到阻塞队列的队尾
4. 然后通过判断acquireQueued(**final** Node node, **int** arg)判断是否需要中断，如果为true则执行selfInterrupt()

1. acquire(**int** arg)方法

Java

```java
public final void acquire(int arg) {
/**
* 1. tryAcquire方法尝试获取锁，如果获取失败则加入到同步队列队尾
* 2. addWaiter方法将节点加到同步队列的队尾
* 3. acquireQueued如果返回true则需要被中断
* 4. 如果需要被中断则调用selfInterrupt()方法处理
*/
if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
selfInterrupt();
}
```

2. addWaiter方法将获取失败的节点已独占的方式添加到队尾

Java

```java
private Node addWaiter(Node mode) {
    Node node = new Node(Thread.currentThread(), mode);
    // Try the fast path of enq; backup to full enq on failure
    /**
     * 尝试一次快速添加到队尾
    */
    Node pred = tail;
    if (pred != null) {
            node.prev = pred;
            if (compareAndSetTail(pred, node)) {
                pred.next = node;
                return node;
        }
    }
    /**
    * 如果快速尝试失败则进行自旋，保证节点被成功的添加到队尾
    */
    enq(node);
    return node;
}
```

enq方法进行自旋操作保证节点添加到队尾

Java

```java
private Node enq(final Node node) {
    for (;;) {
            Node t = tail;
            /**
                     * 如果tail为空，说明队列未初始化则进行初始化操作，设置头节点
            * tail不为空，则添加到队尾
            */
            if (t == null) { // Must initialize
                if (compareAndSetHead(new Node()))
                    tail = head;
            } else {
                node.prev = t;
                if (compareAndSetTail(t, node)) {
                        t.next = node;
                        return t;
                }
            }
        }
}
```

经过步骤2节点进入队列后，当前线程怎么办?

3. acquireQueued方法里面,这个方法内部除了处理中断，还做了其他的事情
    - 程序挂起
    - 移除CANCELLED状态的Node，什么时候节点会被标记为CANCELLED状态?
    - 判断当前线程是否需要中断

Java

```java
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            /**
             * 判断当前节点的前驱节点是否为head节点，如果是则尝试获取锁
            */
            if (p == head && tryAcquire(arg)) {
                    setHead(node);
                    p.next = null; // help GC
                    failed = false;
                    return interrupted;
                }
            /**
            * 如果当前节点获取锁失败
            * shouldParkAfterFailedAcquire判断是否需要挂起，如果需要挂起则parkAndCheckInterrupt()
            * parkAndCheckInterrupt()方法在被唤醒后，会返回当前线程在挂起期间有没有被执行interupt方法，并返回状态，
            * 如果interupted被标记为true,再次执行获取到锁之后执行线程中断selfInterupt()
            */
                if (shouldParkAfterFailedAcquire(p, node) && parkAndCheckInterrupt())
                                interrupted = true;
                }
         } finally {
            if (failed)
                    cancelAcquire(node);
         }
}
```

shouldParkAfterFailedAcquire判断是否需要挂起线程，防止一直在自旋获取锁，这个是非常消耗cpu的操作。

Java

```java
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    int ws = pred.waitStatus;
    /**
     * 如果前一个节点的waitStatus是 SIGNAL状态，则当前线程可以安全的park
     */
    if (ws == Node.SIGNAL)

        return true;
    /**
     * 如果前一个线程的等待状态>0 CANCELED说明是取消状态则移除该节点，循环移除前面所有的已取消的线程
     */
    if (ws > 0) {

            do {
                node.prev = pred = pred.prev;
            }
            while (pred.waitStatus > 0);
            pred.next = node;
      } else {

        /**
          * 等待状态必须为 0  或者PROPAGATE状态，说明我们需要一个信号，还不需要park,
        * 调用者将会继续尝试确认他无法获取在park之前
        */
            compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
        }
    return false;
}
```

如果线程需要挂起则执行parkAndCheckInterrupt()

Java

```java
private final boolean parkAndCheckInterrupt() {
    //执行挂起
    LockSupport.park(this);
    //线程执行unpark的时候返回线程的状态
    return Thread.interrupted();
}
```

如何处理中断

LockSupport.*park*(**this**)线程挂起但是会响应中断，但是不会抛出异常，如果线程在挂起的时候被调用了interupt方法，那么会将线程的中断状态置为true，Thread.*interrupted*()会返回true，并将线程中断状态重置false

线程唤醒后在acquireQueued方法会返回true。则执行*selfInterrupt()方法,等线程后续的工作处理完线程就会回收*

```Java
static void selfInterrupt() {
    Thread.currentThread().interrupt();
}
```

状态什么时候变成CANCLLED

Java

```java
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            /**
             * 判断当前节点的前驱节点是否为head节点，如果是则尝试获取锁
            */
            if (p == head && tryAcquire(arg)) {
                    setHead(node);
                    p.next = null; // help GC
                    failed = false;
                    return interrupted;
                }
            /**
            * 如果当前节点获取锁失败
            * shouldParkAfterFailedAcquire判断是否需要挂起，如果需要挂起则parkAndCheckInterrupt()
            * parkAndCheckInterrupt()方法在被唤醒后，会返回当前线程在挂起期间有没有被执行interupt方法，并返回状态，
            * 如果interupted被标记为true,再次执行获取到锁之后执行线程中断selfInterupt()
            */
                if (shouldParkAfterFailedAcquire(p, node) && parkAndCheckInterrupt())
                                interrupted = true;
               }
         } finally {
            if (failed)
                   cancelAcquire(node);
         }
}
```

其中执行cancelAcquire方法会将节点置为取消状态

> 1. 长时间获取不到锁，比如说获取锁超时
> 2. tryAcquire方法是交给实现者去实现的，如果存在异常，也会取消

Java

```java
private void cancelAcquire(Node node) {
  // 将无效节点过滤
	if (node == null)
		return;
  // 设置该节点不关联任何线程，也就是虚节点
	node.thread = null;
	Node pred = node.prev;
  // 通过前驱节点，跳过取消状态的node
	while (pred.waitStatus > 0)
		node.prev = pred = pred.prev;
  // 获取过滤后的前驱节点的后继节点
	Node predNext = pred.next;
  // 把当前node的状态设置为CANCELLED
	node.waitStatus = Node.CANCELLED;
  // 如果当前节点是尾节点，将从后往前的第一个非取消状态的节点设置为尾节点
  // 更新失败的话，则进入else，如果更新成功，将tail的后继节点设置为null
	if (node == tail && compareAndSetTail(node, pred)) {
		compareAndSetNext(pred, predNext, null);
	} else {
		int ws;
    // 如果当前节点不是head的后继节点，
    //1:判断当前节点前驱节点的是否为SIGNAL，
    //2:如果不是，则把前驱节点设置为SINGAL看是否成功
    // 如果1和2中有一个为true，再判断当前节点的线程是否为null
    // 如果上述条件都满足，把当前节点的前驱节点的后继指针指向当前节点的后继节点
		if (pred != head && ((ws = pred.waitStatus) == Node.SIGNAL || (ws <= 0 && compareAndSetWaitStatus(pred, ws, Node.SIGNAL))) && pred.thread != null) {
			Node next = node.next;
			if (next != null && next.waitStatus <= 0)
				compareAndSetNext(pred, predNext, next);
		} else {
      // 如果当前节点是head的后继节点，或者上述条件不满足，那就唤醒当前节点的后继节点
			unparkSuccessor(node);
		}
		node.next = node; // help GC
	}
}
```

释放锁的过程

1. 调用release方法

Java

```java
public final boolean release(int arg) {
        if (tryRelease(arg)) {
                Node h = head;
                if (h != null && h.waitStatus != 0)
                        unparkSuccessor(h);
                return true;
        }
        return false;
}
```

   - h == null Head还没初始化。初始情况下，head == null，第一个节点入队，Head会被初始化一个虚拟节点。所以说，这里如果还没来得及入队，就会出现head == null 的情况。
   - h != null && waitStatus == 0 表明后继节点对应的线程仍在运行中，不需要唤醒。
   - h != null && waitStatus < 0 表明后继节点可能被阻塞了，需要唤醒

2. 如果节点需要被唤醒则调用unparkSuccessor()唤醒后继节点

Java

```java
private void unparkSuccessor(Node node) {
    //获取头节点的状态
    int ws = node.waitStatus;
    //如果节点waitStatus<0
    if (ws < 0)
        //将节点状态修改为0，表示在运行中
        compareAndSetWaitStatus(node, ws, 0);

     //取后继节点
    Node s = node.next;
    // 如果后继节点是null或者后继节点被cancelled，就找到队列最开始的非cancelled的节点
    if (s == null || s.waitStatus > 0) {
        s = null;
        //从后往前找找到waitStatus需要被唤醒的节点进行唤醒
        for (Node t = tail; t != null && t != node; t = t.prev)
            if (t.waitStatus <= 0)
                s = t;
    }
    //后继节点不为空则唤醒节点的线程
    if (s != null)
        LockSupport.unpark(s.thread);
}
```

> Q：为什么要从后往前遍历查找呢？
>
> A：在入队的时候compareAndSetTail是原子操作，但是pred.**next** = node;是非原子操作，所以在操作的时候成为tail节点后，原tail节点的next节点还没有指向当前节点，所以只能反向递归，否则可能会递归到别的链

Java

```java
if (compareAndSetTail(pred, node)) {
                pred.next = node;
                return node;
 }
```

公平锁&非公平锁实现

> 公平锁与非公平锁的区别就在获取锁的过程，
>
> - 公平锁在获取锁的时候，如果队列中还有节点的话，就需要排队，依次获取
> - 非公平锁在获取锁的时候直接抢占，如果抢占失败才会进入队列排队

ReentrantLock支持公平锁与非公平锁的，那么以ReentrantLock为例子。

非公平锁的实现

```java
static final class NonfairSync extends Sync {
    private static final long serialVersionUID = 7316153563782823691L;

    /**
     * Performs lock.  Try immediate barge, backing up to normal
     * acquire on failure.
     */
    final void lock() {
        if (compareAndSetState(0, 1))
            setExclusiveOwnerThread(Thread.currentThread());
        else
            acquire(1);
    }

    protected final boolean tryAcquire(int acquires) {
            return nonfairTryAcquire(acquires);
    }
}
```

> 1. 首先调用lock方法的时候先尝试修改状态，如果修改成功，则说明获取到了锁，并设置当前的独占线程为当前线程
> 2. 如果快速获取失败则调用acquire(1)，调用此方法时会调用tryAcquire，最终调用nonfairTryAcquire方法

Java

```java
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {
                setExclusiveOwnerThread(current);
                return true;
            }
     }
     //可重入实现
    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;
}
```

> 1. 此方法先判断当前state是不是无锁状态，如果是则尝试获取锁，获取成功则返回
> 2. 如果当前state是加锁状态，则判断是否是当前线程占用如果是则设置状态state+1表示重入次数

公平锁的实现

Java

```java
static final class FairSync extends Sync {
    private static final long serialVersionUID = -3000897897090466540L;

    final void lock() {
            acquire(1);
        }

    /**
         * Fair version of tryAcquire.  Don't grant access unless
         * recursive call or no waiters or is first.
         */
    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;
  }
}
```

可重入锁实现

> 重入锁的实现，就是在判断独占线程是否是当前线程，如果是则说明当前线程已经获取了锁，那么只是将当前state+1表示重入次数

Java

```java
if (current == getExclusiveOwnerThread()) {
                int nextc = c + acquires;
                if (nextc < 0)
                    throw new Error("Maximum lock count exceeded");
                    setState(nextc);
                return true;
            }
```

> 重入锁的释放,也需要一次一次的释放直到state=0

Java

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

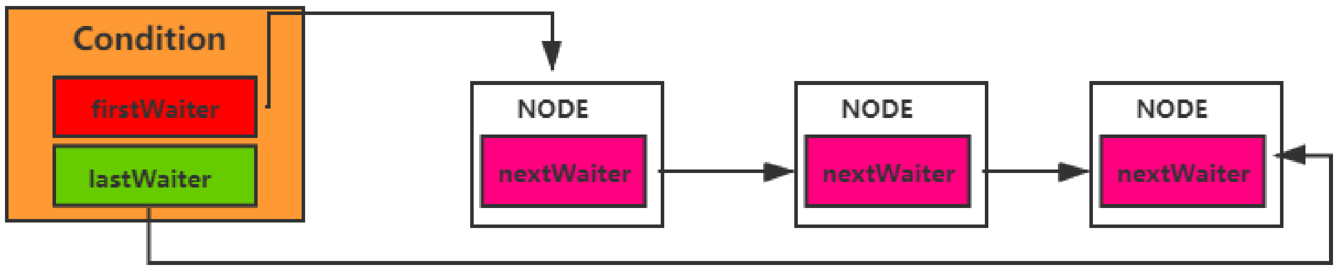
四、Condition

ConditionObject是AQS的内部类实现了Condition接口，这个对象构成了条件队列

> 1. ConditionObject 存在**firstWaiter、lastWaiter**
> 2. 每一个Waiter都是Node组成，每个Node中都有nextWaiter
> 3. 所以整个就构成了等待队列

Java

```java
public class ConditionObject implements Condition, java.io.Serializable {
    private static final long serialVersionUID = 1173984872572414699L;
    /** First node of condition queue. */
    private transient Node firstWaiter;
    /** Last node of condition queue. */
    private transient Node lastWaiter;
}
```



condition结构图

condition如何工作的?

> **DEMO，以阻塞队列为例**

Java

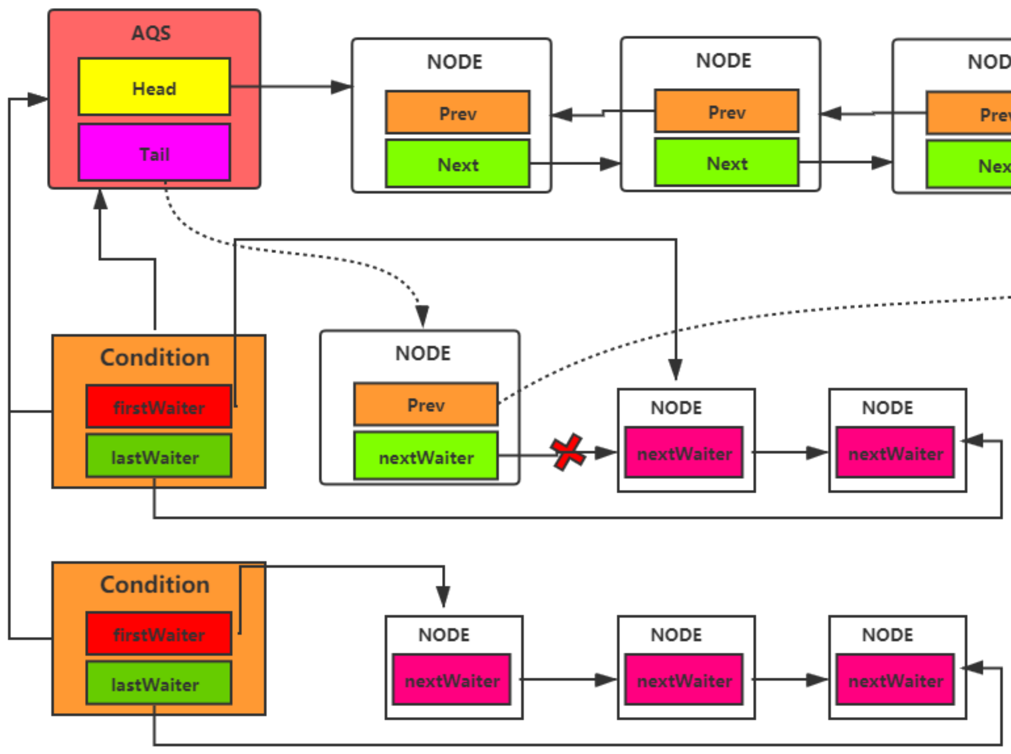
```java
final ReentrantLock lock;

/** Condition for waiting takes */
private final Condition notEmpty;

/** Condition for waiting puts */
private final Condition notFull;


public void put(E e) throws InterruptedException {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == items.length)
```

```java
                notFull.await();
            enqueue(e);
        } finally {
            lock.unlock();
        }
}

public E take() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            while (count == 0)
                notEmpty.await();
            return dequeue();
        } finally {
            lock.unlock();
        }
}

private void enqueue(E x) {
        final Object[] items = this.items;
        items[putIndex] = x;
        if (++putIndex == items.length)
            putIndex = 0;
        count++;
        notEmpty.signal();
}

private E dequeue() {
        final Object[] items = this.items;
        E x = (E) items[takeIndex];
        items[takeIndex] = null;
        if (++takeIndex == items.length)
            takeIndex = 0;
        count--;
        if (itrs != null)
            itrs.elementDequeued();
        notFull.signal();
        return x;
}
```

以上是阻塞队列的核心实现：

1. **put**方法在往阻塞队列内添加元素的时候先获取锁，如果当前**count**与数组长度相等则说明队列满了，那么调用**notFull.await()**，挂起**生产者**
2. 如果队列未满则执行**enqueue(e)**，入队，同时执行connt计数加1，以及执行**notEmpty.signal()**，队列不为空，唤醒**消费者**
3. take方法是从阻塞队列中取元素，如果当前数量count为0，以及调用**notEmpty.await()**;非空等待，说明当前队列中没有数据，无法获取，同时**挂起消费者**
4. 如果队列count>0 则执行**dequeue()**出队，同时count计数减1，以及调用**notFull.signal()；**未满，说明队列中还有空间，同时**唤醒生产者**

Signal



1. signal方法判断如果条件队列首节点不为空，说明有等待条件的线程，那么调用doSignal
2. doSignal方法做一个循环判断，首先从条件队列中取出第一个Node然后进行判断
3. transferForSignal为true时发生了什么？
    - *compareAndSetWaitStatus*(node, Node.***CONDITION***, 0)如果替换失败则返回false，说明已经被其他线程执行了signal了，需要移除，则继续取下一个Node重复直到取到一个为止
    - 取到后enq(node)进入同步队列，返回前驱节点，**只要前驱节点处于被取消的状态或者无法将前驱节点的状态修成Node.SIGNAL，那我们就将Node所代表的线程唤醒**，但这个条件并不意味着当前lock处于可获取的状态，有可能线程被唤醒了，但是锁还是被占有的状态，不过这样做至少是无害的，因为我们在线程被唤醒后还要去争锁，如果抢不到锁，则大不了再次被挂起

Java

```java
public final void signal() {
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    //如果firstWaiter为null说明队列中没有等待者则什么都不做
    Node first = firstWaiter;
    if (first != null)
    //如果first不为空则执行doSignal
    doSignal(first);
}
```

Java

```java
private void doSignal(Node first) {
    do {
            //第一个节点的指针指向下一个节点，下一个节点为空
            //说明队列为空，则设置lastWaiter为空
            if ( (firstWaiter = first.nextWaiter) == null)
                lastWaiter = null;
            //断开first节点指向nextWaiter 指针
            first.nextWaiter = null;
        } while (!transferForSignal(first) &&
                 (first = firstWaiter) != null);
}
```

Java

```java
final boolean transferForSignal(Node node) {
    /**
     * 如果将节点状态从CONDITION，换成0状态失败则返回false
     *
     */
    if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
        return false;

    //进入同步队列，并返回其前驱节点
    Node p = enq(node);
    int ws = p.waitStatus;

    if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))
            LockSupport.unpark(node.thread);
    return true;
}
```

Await

> 1. 当前线程肯定是已经获取到了锁，判断当前线程是否已经被中断过，如果被中断过则响应中断抛出异常
> 2. 通过addConditionWaiter方法添加Node到条件队列

Java

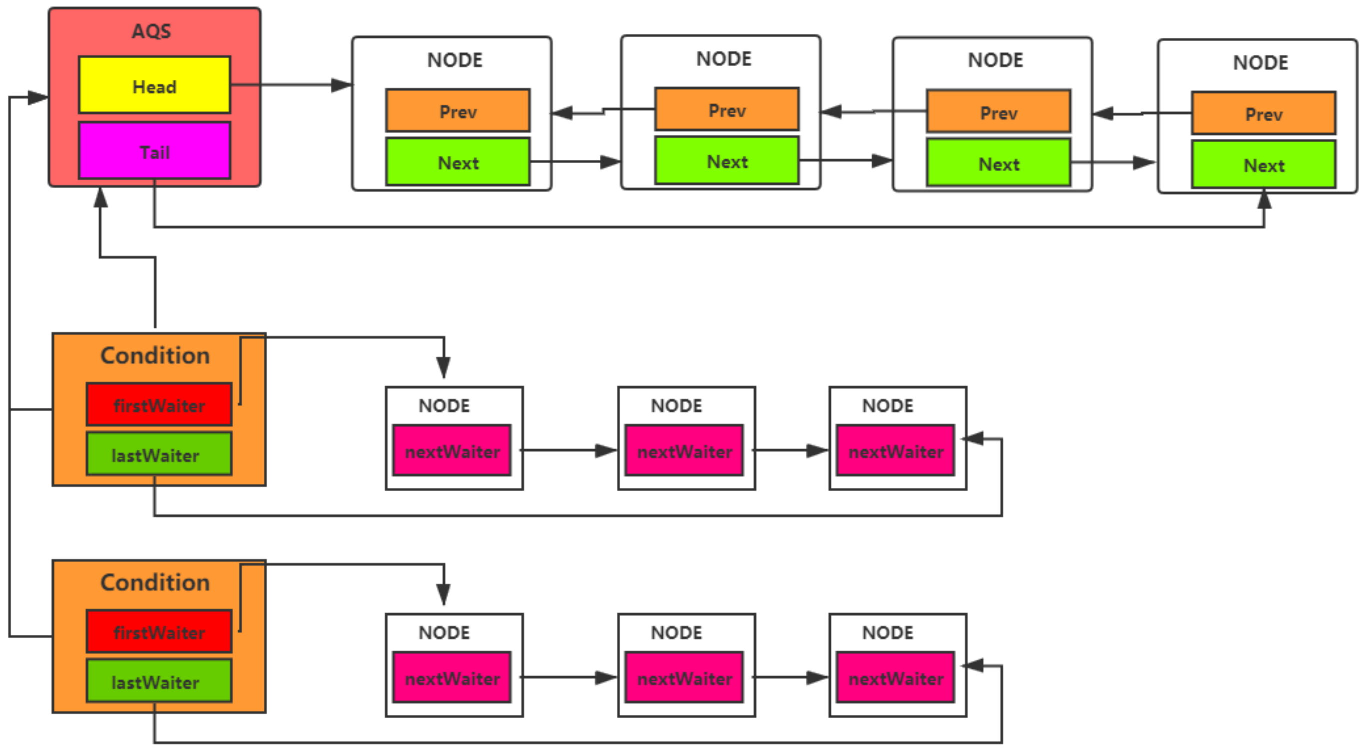
```java
public final void await() throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();

    //添加到条件队列
    Node node = addConditionWaiter();
    //释放当前节点占用的锁
    int savedState = fullyRelease(node);
    int interruptMode = 0;
    // 如果当前队列不在同步队列中，说明刚刚被await，还没有人调用signal方法，则直接将当前线程挂起
    while (!isOnSyncQueue(node)) {
        LockSupport.park(this);
        // 能执行到这里说明要么是signal方法被调用了，要么是线程被中断了
        // 所以检查下线程被唤醒的原因，如果是因为中断被唤醒，则跳出while循环
        if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
            break;
    }
    if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
            interruptMode = REINTERRUPT;
    if (node.nextWaiter != null) // clean up if cancelled
            unlinkCancelledWaiters();
    if (interruptMode != 0)
        reportInterruptAfterWait(interruptMode);
}
```

Java

```java
private Node addConditionWaiter() {
    Node t = lastWaiter;
    // 如果节点存在被取消的节点则移除
    if (t != null && t.waitStatus != Node.CONDITION) {
            unlinkCancelledWaiters();
            t = lastWaiter;
        }
    //创建CONDITION状态的节点并入队
    Node node = new Node(Thread.currentThread(), Node.CONDITION);
    if (t == null)
        firstWaiter = node;
    else
        t.nextWaiter = node;
    lastWaiter = node;
    return node;
}
```

五、全局结构

> 回头俯瞰全景图



AQS全貌图

六、结语

谢谢