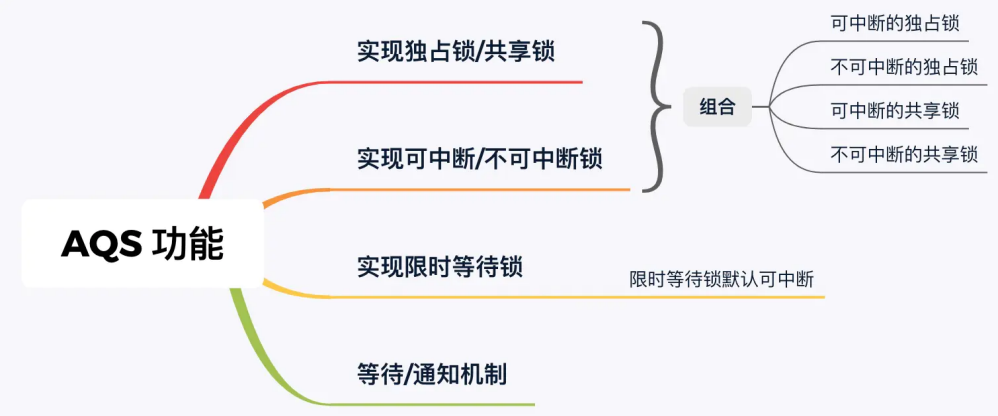
AQS（抽象队列同步器）

队列同步器是用来构建锁或者其它的同步组件的基础框架，它使用一个int变量来表示同步状态，通过内
置的FIFO队列(双向链表，逻辑上的链表，实质上只保留了的head、tail).使用方式主要是通过继承，实现tryAcquire/tryRelease、tryAcquireShared/tryReleaseShared、isHeldExclusively等方法来完成
自定义的同步组件的功能，从语义上来划分就是获取/释放、从模式来区分提供独占式、共享式，从响应中断
来区分就是支持中断、不支持中断.

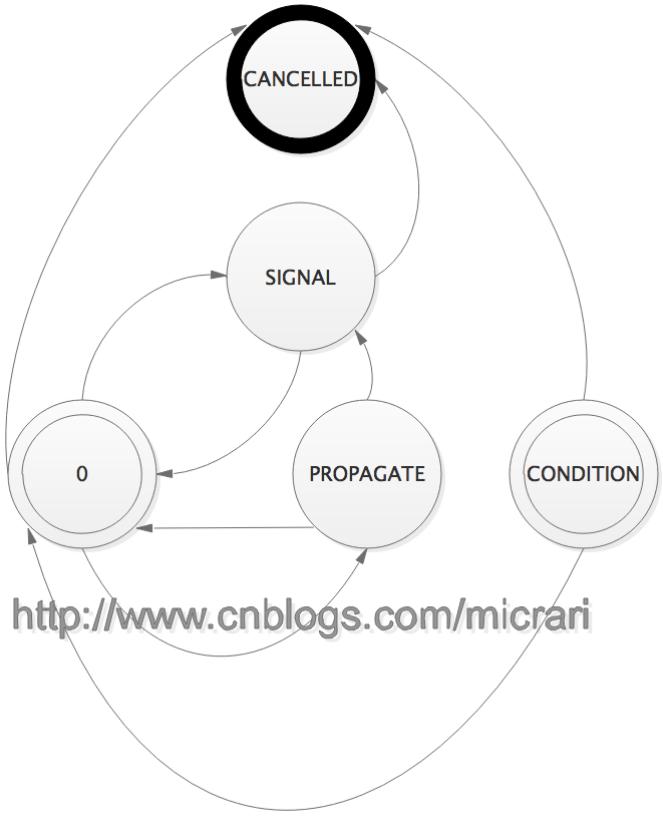
AQS包含了head和tail两个Node引用，其中head在逻辑上的含义是当前持有锁的线程，head节点实际上是一个虚节点，本身并不会存储线程信息(当节点获得同步状态时，会将自己设置head，并且将tread、pre清空).当一个线程无法获取同步状态而被加入到同步队列时，会用CAS来设置尾节点tail为当前线程对应的Node节点.head和tail在AQS中是延迟初始化的，也就是在需要的时候才会被初始化，也就意味着在所有线程都能获取到锁的情况下，队列中的head和tail都会是null.



Node

- prev、next、waitStatus、thread、nextWaiter
- CANCEL=1、SIGNAL=-1、CONDITION=-2、PROPAGATE=-3、0

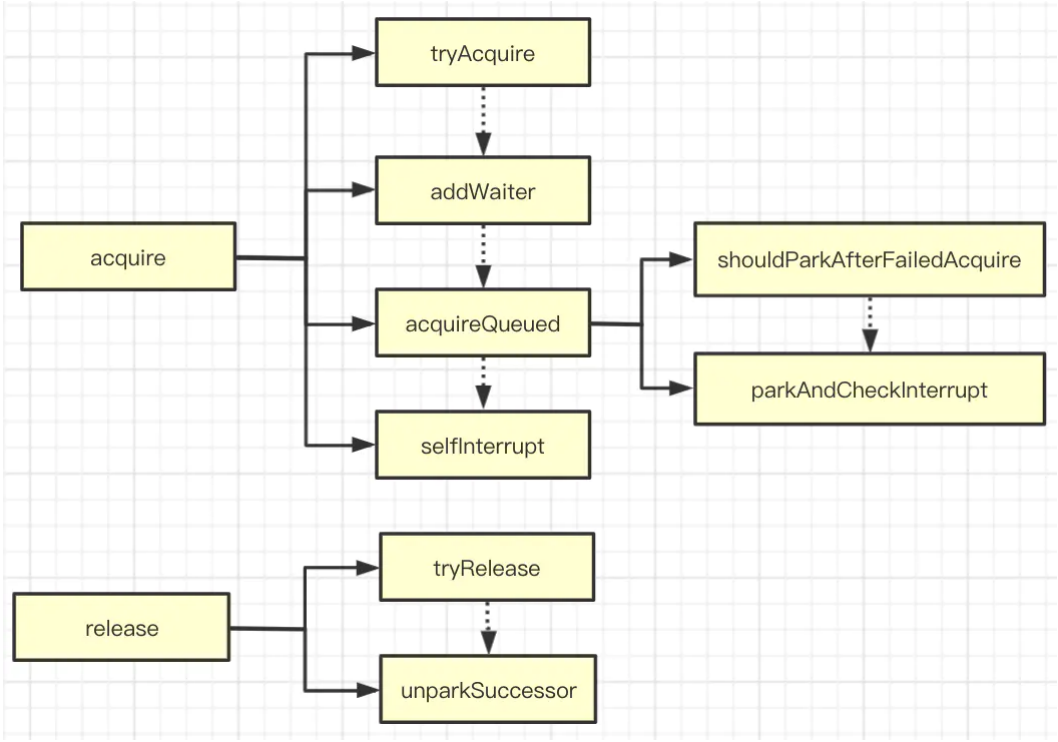
waitStatus



AQS内部类Node中的waitStatus字段有五种状态,0表示节点刚插入到同步队列，SIGNAL=-1表示当前节点释放后要唤醒后继等待节点、CANCELLED=1表示当前节点已经超时或者取消竞争同步状态、CONDITION=-2表示当前节点在condition队列中、PROPAGATE=-3表示将唤醒后继线程传递下去.

独占锁

- 自定义同步组件通过实现tryAcquire/tryRelease函数来完成独占锁功能.
- 独占锁函数调用关系



- 独占锁的获取

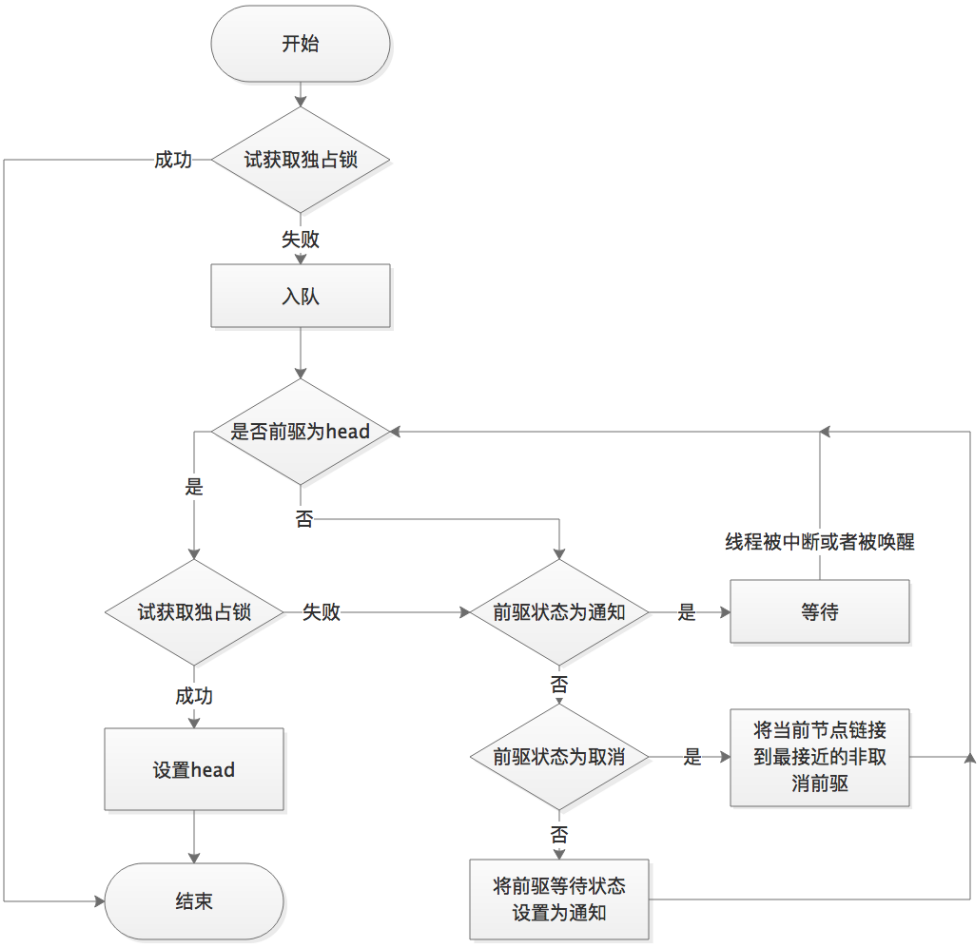
  AQS独占锁的获取如下：

  首先线程调用acquire函数去获取独占锁，然后acquire内部会先调用tryAcuqire(具体由用户自实现)，如果获取成功则返回；

  失败，就会调用addWaiter函数将包装当前线程信息生成一个独占模式的节点Node，添加到同步队列的尾部tail；然后调用acquireQueued函数，以自旋的方式(for死循环)去尝试获取同步状态，只有当前驱节
  点pre为head节点时，才去尝试竞争同步状态，成功了则将当前节点设置为head，原head节点置空回收；

  如果当前驱节点pre不为head，则调用shouldParkAfterFailedAcquire函数，该函数会判断前驱节点pre的状态waitStatus，如果状态为-1(SIGNAL)，则表明前面的节点**已经设置了释放锁后会激活当前节点**，因此当前节点可以安全的调用parkAndCheckInterreupt函数挂起等待；
  如果pre的wS=1，则表明pre已经取消，则循环寻找最近的未取消的pre置为当前节点的pre；如果wS为其它值，则直接将wS设置为SIGNAL；

  acquire函数是不响应中断的，因为在parkAndCheckInterreupt进行挂起时线程如果被中断了，只会激活后设置interrupt=true，而不会抛出中断异常，继续进行同步状态的竞争.



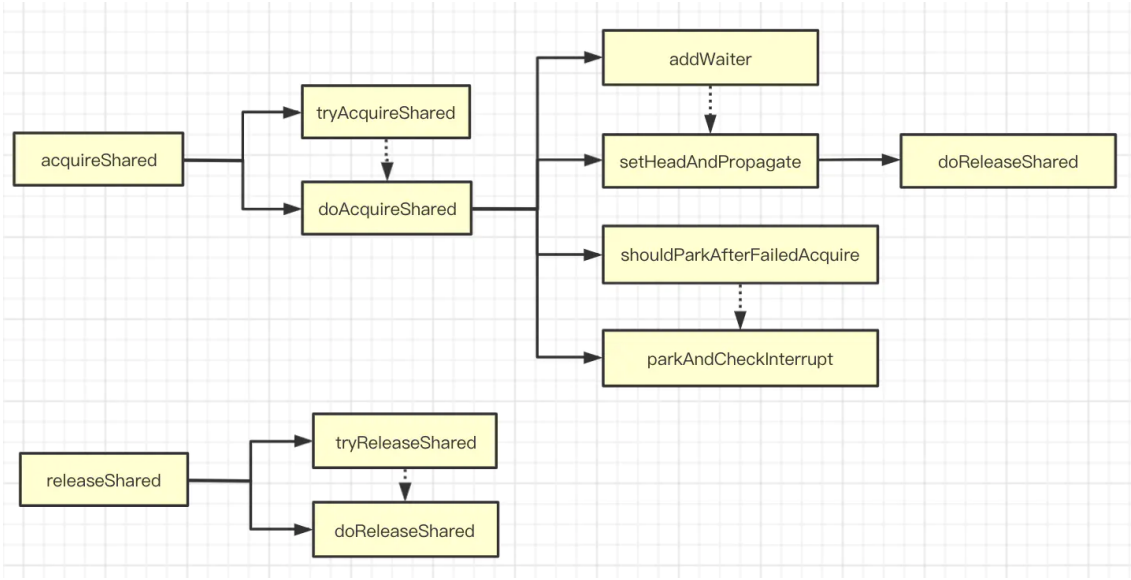
- 独占锁的释放

  通过release函数调用tryRelease函数判断是否完全释放了同步状态，是则唤醒后继线程(当head存在并且设wS<0才唤醒).

共享锁

共享锁允许多个线程持有，与独占锁的主要区别在于同步状态state可以大于1和PROPAGATE.

- 自定义同步组件通过实现tryAcquireShared/tryReleaseShared函数来完成共享锁功能.
- 共享锁的函数调用关系



- 共享锁的获取

  线程通过调用acquireShared函数去获取共享锁，然后内部会调用tryAcquireShared函数去获取共享锁，返回int值propagate，当propagate>=0时表示获取共享锁成功，则返回;

propagate<0，则调用doAcquireShared函数，内部调用addWaiter函数将线程信息包装生成SHARED模式的节点加入同步队列，以自旋的方式获取同步状态，只有当前驱节点pre为head节点时，才去尝试竞争同步状态，成功了则调用setHeadAndPropagate函数将当前节点设置为head，并且**propagate>0或者当head的状态为PROPAGATE**时(head.wS<0)[表示同步状态还可以获取、或者刚刚释放了资源]要调用doReleaseShared函数，该函数的作用是当前同步队列存在后继线程并且head.sW==SIGNAL时就去唤醒后继线程，否则则将当前head.sW置为PROPAGETE.

如果当前驱节点pre不为head，则调用shouldParkAfterFailedAcquire函数，和独占锁过程一样.

- 共享锁的释放

  线程调用releaseShared函数去释放共享锁，内部调用treReleaseShared函数去释放，成功后会调用doReleaseShared函数去唤醒后继线程或者设置head.sW置为PROPAGETE.

- tips(一些细节)
  - 节点插入同步队列操作顺序

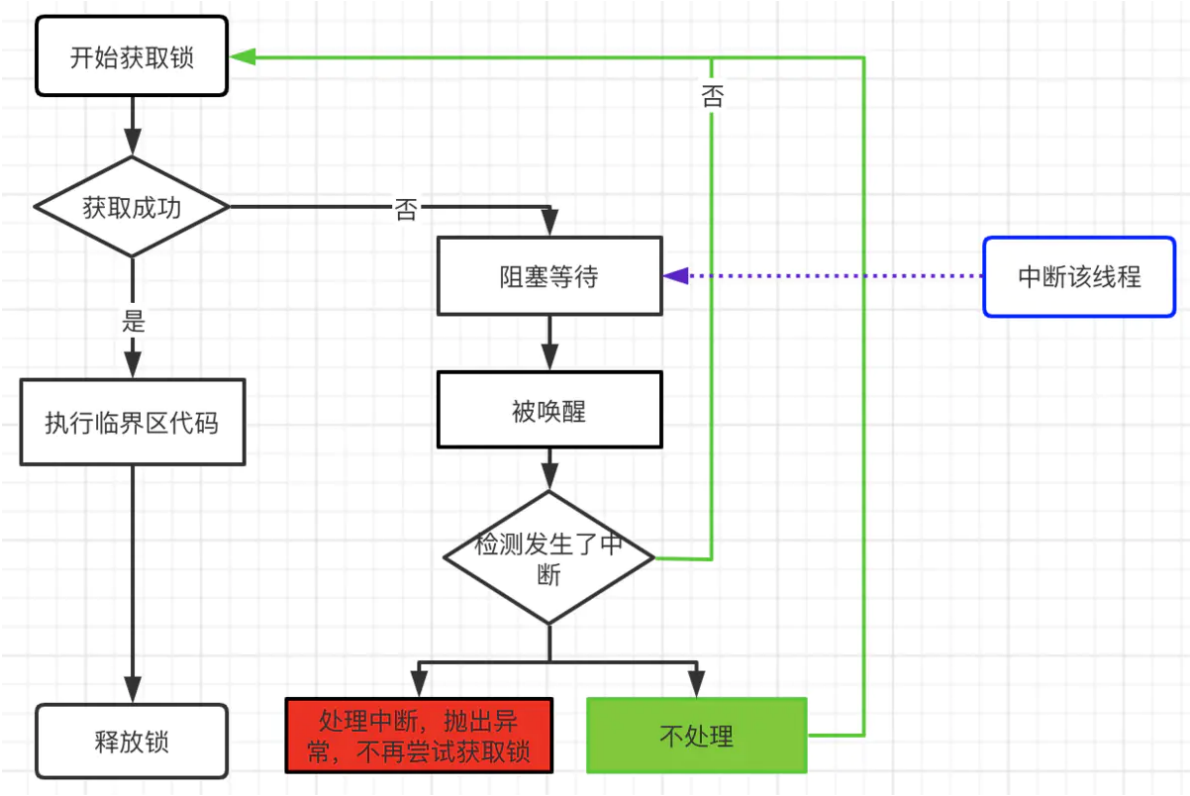
    节点插入同步队列时是先设置tail = node.pre，再CAS设置node为tail后，设置tail.next = node，这里是为了避免CAS设置node为tail后如果node.pre为设置，则其它线程读取当前tail node.pre == null的情况.
  - 唤醒节点时从tail从后向前遍历找第一个后继

    由于上面插入的操作，当从head遍历时，s = node.nex，如果CAS设置node为tail后tail.next = node操作还未完成，则当前s.next==null，实际上后面是有节点的，因此从tail向前遍历.
  - PROPAGATE

    由于共享锁可以允许多个线程持有，因此会出现多个线程竞争/释放锁时会同时持有head副本的情况，当其中一个获取锁的线程还未将自己设置为head时，另一个释放锁的线程由于旧版本的head.wS=0便不去唤醒下一个等待线程，获取的线程也因为propagate=0而不去唤醒下一个线程，此时会产生有锁资源而线程不被唤醒的情况，因此加入PROPAGETE状态，来规避这种情况，使得线程会被唤醒.

可中断/不可中断

可中断就是在获取锁的过程中会相应中断，不可中断就是不会相应中断.



- 可中断

  可中断的锁获取acquireInterruptibly/acquireSharedInterruptibly会在尝试获取锁之前(tryAcquire/trySharedAcquire)之前判断当前是否发生中断发生中断则抛出异常、也会在同步队列等待中被中断唤醒后直接抛出中断异常；

- 不可中断

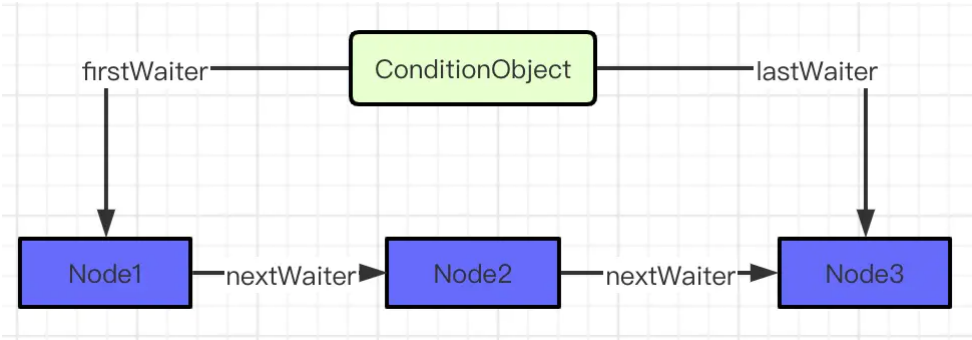
  不可中断的锁获取在同步队列中被中断唤醒后只会标记中断发生标记，然后继续进行获取锁的过程.

等待/通知机制

等待通知机制是实现同步功能的一种机制，在AQS中ConditionObject内部类就是用来实现这个机制的.

- ConditionObject

  ConditionObject中有firstWaiter、lastWaiter两个头尾节点，它们和节点中的nextWaiter共同组成了一个等待队列(条件队列)，它们不是volatile字段因为使用条件对象前默认已经获取锁.



- 实现

  ConditionObject通过await/signal、signalAll等函数来完成等待和通知功能.

- await

调用conditionObject的await函数会将当前线程的信息以CONDITION的方式生成节点加入到等待队列的尾部设置为lastWaiter，并且将当前线程的锁全部释放(重入锁至到state=0)；然后进入等待队列进行等待，直到有中断发生或者其它线程调用conditionObject的signal函数将当前线程唤醒(此时节点已经被加入同步队列)，调用acquireQueued函数进行锁的争抢，获得锁后退出或进行中断处理后退出.

- signal/signalAll

  先获取firstWaiter然后调用doSignal找到等待队列中首个条件节点(有效的等待节点)，将其从等待队列中移除，加入同步队列，修改同步队列中该节点的前驱wS=SIGNAL，如果前驱被取消或者修改失败则直接唤醒该节点的线程.

  - 从上面细节可知，signal的唤醒可能是直接唤醒或者是通过同步队列中的SIGNAL机制来唤醒，唤醒后await中判断当前节点已经在同步队列中，则会进入锁竞争正常退出(携锁退出)，进入临界区执行代码.
  - signalAll是将所有等待队列中的节点移到同步队列.
- 中断唤醒

  当await使得线程挂起等待的时候，可能会由于中断使得线程被唤醒，因此await唤醒后会判断唤醒的原因是SIGNA还是中断或者是两者都发生了(都会将节点加入同步队列)，如果只发生了中断，那么await获取锁后会先将节点从等待队列移出，然后直接抛出异常(await/signal成对出现signal未执行就中断，因此直接抛出异常)；如果发生了SIGNAL则会先确保SIGNAL操作将节点加入同步队列，然后await获取锁后，由于signal发生了，所有不会在await处理异常，只会重标记中断发生，留给调用者处理.
- 调用条件

  await/signal函数都是要保证在持有独占锁的情况执行，否则会报异常.
- tips
  - CONDITION与同步队列

    CONDITION和syn队列共用NODE节点，CONDITION是单向FIFO只利用了nextWaiter，并且CONDITION方法使用是默认获得独占锁条件，因此nextWaiter、firstWaiter、lastWaiter都是非volatile字段.
  - await/signal与wait/notify

    都需要加锁(synchronized、lock)，等待/通知成对出现，等待方法都可以响应中断，都支持超时返回;

    await/signal依赖lock是JDK实现，wait/notify是JVM实现，CONDITION等待方法也实现了不中断响应等待，等待超时可以设置时间点超时，可以一把锁生成多个CONDITION.
- **参考资料**
  - <<Java并发编程的艺术>>
  - AbstractQueuedSynchronizer源码解读 - 活在夢裡 - 博客园 (cnblogs.com)
  - Java并发之 AQS 深入解析(上) - 简书 (jianshu.com)
  - Java并发之 AQS 深入解析(下) - 简书 (jianshu.com)
  - Java并发之ReentrantLock源码解析 (四) - 北洛 - 博客园 (cnblogs.com)
  - 深入理解AbstractQueuedSynchronizer (三) | Idea Buffer