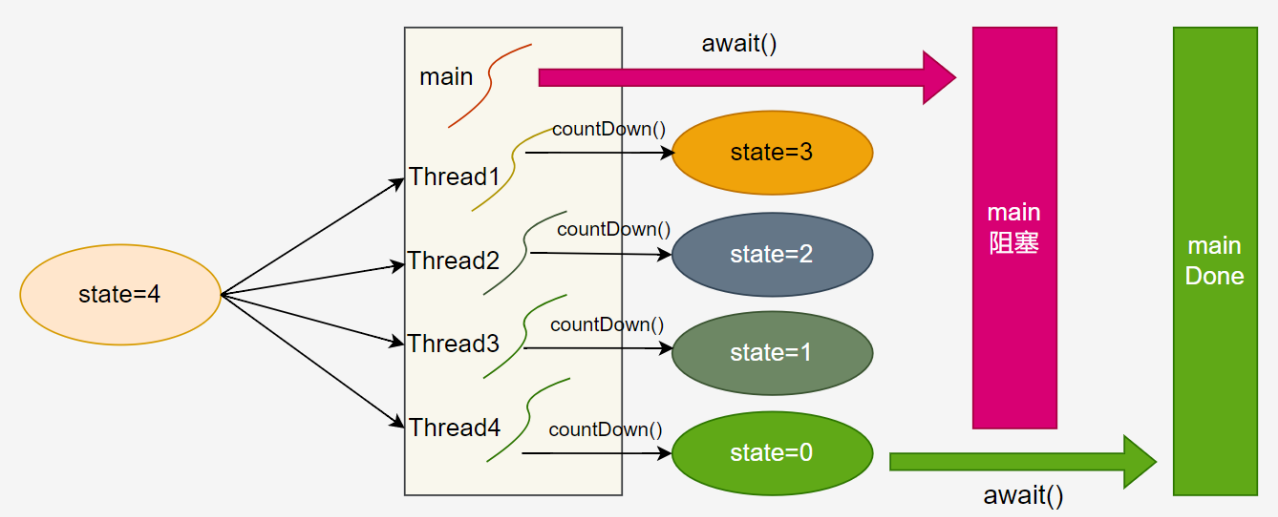
主讲老师：Fox

> 【有道云笔记】8.2 CountDownLatch源码分析
> https://note.youdao.com/s/RImvV3BD

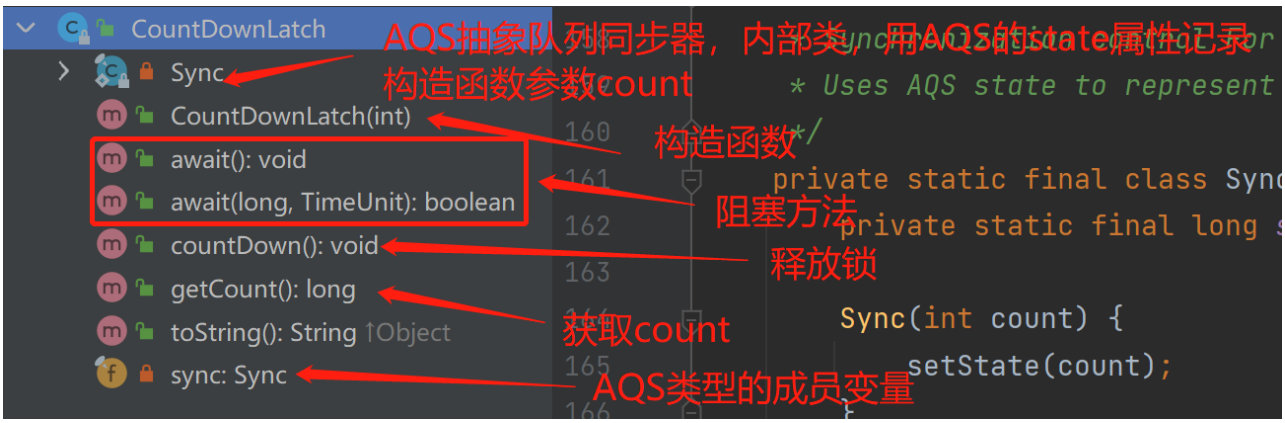
1、CountDownLatch介绍

CountDownLatch让一个或多个线程等待其他线程执行完成后再执行。在创建CountDownLatch对象时，必须指定线程数count，每当一个线程执行完成调用countDown()方法，线程数count减1，当count减到0时，await()方法就不再阻塞。



2、CountDownLatch源码分析

CountDownLatch详情如下：



2.1 构造函数

CountDownLatch没有无参构造函数，在有参构造函数中初始化了sync属性。

```java
public CountDownLatch(int count) {
    // count 合法校验
    if (count < 0) throw new IllegalArgumentException("count < 0");
    // 初始化sync属性
    this.sync = new Sync(count);
}
```

2.2 Sync - 队列同步器

```java
// 抽象队列同步器
private static final class Sync extends AbstractQueuedSynchronizer {
    private static final long serialVersionUID = 4982264981922014374L;

    // 将 count 赋值给 AQS 的 state 属性
    Sync(int count) {
        setState(count);
    }
    // 获取 AQS 的 state 属性
    int getCount() {
        return getState();
    }
    // 判断所有线程是否都执行完成， 1 -> 全部执行完成；-1 -> 仍有线程在执行
    protected int tryAcquireShared(int acquires) {
        return (getState() == 0) ? 1 : -1;
    }
    // 释放锁
    protected boolean tryReleaseShared(int releases) {
        // 自旋
        for (;;) {
            // 获取 AQS 的 state
            int c = getState();
            // 锁资源已经释放完毕，再次进入，直接返回false，什么也不做
            if (c == 0)
                return false;
            //  state - 1
            int nextc = c-1;
            // CAS 赋值操作
            if (compareAndSetState(c, nextc))
                // 最后一个线程执行完，state = 0 ，返回true。
                // countDown() 唤醒等待队列中的其他挂起线程
                return nextc == 0;
        }
    }
}
```

2.3 await() - 阻塞等待

CountDownLatch#await()，详情如下：

```
// AQS的state属性不为0， 阻塞
public void await() throws InterruptedException {
    // 调用AQS提供的获取共享锁并允许中断的方法
    sync.acquireSharedInterruptibly(1);
}
```

AbstractQueuedSynchronizer#acquireSharedInterruptibly()，详情如下：

```
// 获取共享锁，并允许其中断
public final void acquireSharedInterruptibly(int arg) throws InterruptedException {
    // 线程中断，抛出异常
    if (Thread.interrupted())
        throw new InterruptedException();
    // 获取共享锁，由CountDownLatch实现
    if (tryAcquireShared(arg) < 0)
        // state > 0，说明有线程在持有锁资源，将当前线程添加到AQS等待队列中
        doAcquireSharedInterruptibly(arg);
}
```

CountDownLatch#Sync#tryAcquireShared()，详情如下：

```
// 获取共享锁
protected int tryAcquireShared(int acquires) {
    // 线程全部执行完成，返回 1；未全部执行完成，返回-1
    return (getState() == 0) ? 1 : -1;
}
```

AbstractQueuedSynchronizer#acquireSharedInterruptibly()，详情如下：

```
// 将当前线程添加到AQS等待队列中
private void doAcquireSharedInterruptibly(int arg) throws InterruptedException {
    // 当前线程封装成Node，添加到AQS等待队列中
    final Node node = addWaiter(Node.SHARED);
    boolean failed = true;
    try {
        // 自旋
        for (;;) {
            // 获取当前线程节点的前驱节点
            final Node p = node.predecessor();
            // 前驱节点为等待队列头节点
            if (p == head) {
                // 调用 CountDownLatch 实现的方法
                int r = tryAcquireShared(arg);
                // 返回值为1，表示 state 为 0 ，所有线程都释放了锁，无其他线程持有锁资源
                if (r >= 0) {
                    // state = 0，将当前线程和后面所有排队的线程都唤醒。
                    setHeadAndPropagate(node, r);
                    p.next = null;
                    failed = false;
                    return;
                }
            }
            // *** 线程在此处被挂起，待所有线程释放锁资源后，即state = 0 ，线程被唤醒，再继续往下执行
            // 挂起获取锁资源失败的线程，并且挂起的线程被中断，抛出InterruptedException异常
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

2.4 countDown() - 释放锁资源

CountDownLatch#countDown()，详情如下：

```
// countDown方法，实际上调用了AQS的释放共享锁操作
2 public void countDown() {
3     sync.releaseShared(1);
4 }
```

AbstractQueuedSynchronizer#releaseShared()，详情如下：

```
// AQS提供的释放共享锁方法，CountDownLatch实现了 tryReleaseShared 方法
public final boolean releaseShared(int arg) {
    // 尝试释放锁资源
    if (tryReleaseShared(arg)) {
        // 没有线程持有锁资源，唤醒等待队列中的其他挂起线程
        doReleaseShared();
        return true;
    }
    return false;
}
```

CountDownLatch#Sync#tryReleaseShared()，详情如下：

```
protected boolean tryReleaseShared(int releases) {
    // 自旋
    for (;;) {
        // 获取当前持有锁资源的线程数
        int c = getState();
        // state已为0，返回false，那么再次执行countDown，什么事情也不做
        if (c == 0)
            return false;
        // count - 1
        int nextc = c-1;
        // CAS 完成赋值操作
        if (compareAndSetState(c, nextc))
            // 没有线程持有锁资源，返回true
            return nextc == 0;
    }
}
```

AbstractQueuedSynchronizer#doReleaseShared()，详情如下：

```java
// 没有线程持有锁资源的处理
private void doReleaseShared() {
    // 自旋
    for (;;) {
        // 获取等待队列的头节点
        Node h = head;
        // 等待队列中有挂起线程待唤醒
        if (h != null && h != tail) {
            int ws = h.waitStatus;
            // 线程待唤醒
            if (ws == Node.SIGNAL) {
                if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                    continue;
                // 唤醒线程
                unparkSuccessor(h);
            }
            // CAS失败
            else if (ws == 0 &&
                     !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
                continue;
        }
        // 等待队列头节点被改变，结束循环
        if (h == head)
            break;
    }
}
```

2.5 总结

CountDownLatch基于 AQS + CAS 实现，CountDownLatch的构造函数中必须指定count，同时初始继承AQS的内部类Sync，通过Sync对象将count赋值给AQS的state属性，这样就可以基于AQS提供的方法完成CountDownLatch的功能。

调用countDown()方法，实际上是将AQS中 state 减 1。所有线程执行完成，state 会被修改为 0，在countDown()中会唤醒等待队列中挂起的线程。

调用await()方法，实际上是判断AQS中的 state 是否为 0。state > 0，表示有线程仍在执行，此时await()会阻塞线程。当最后一个线程执行结束，state 变为 0，countDown()唤醒线程后，await()正常执行结束，不再阻塞。