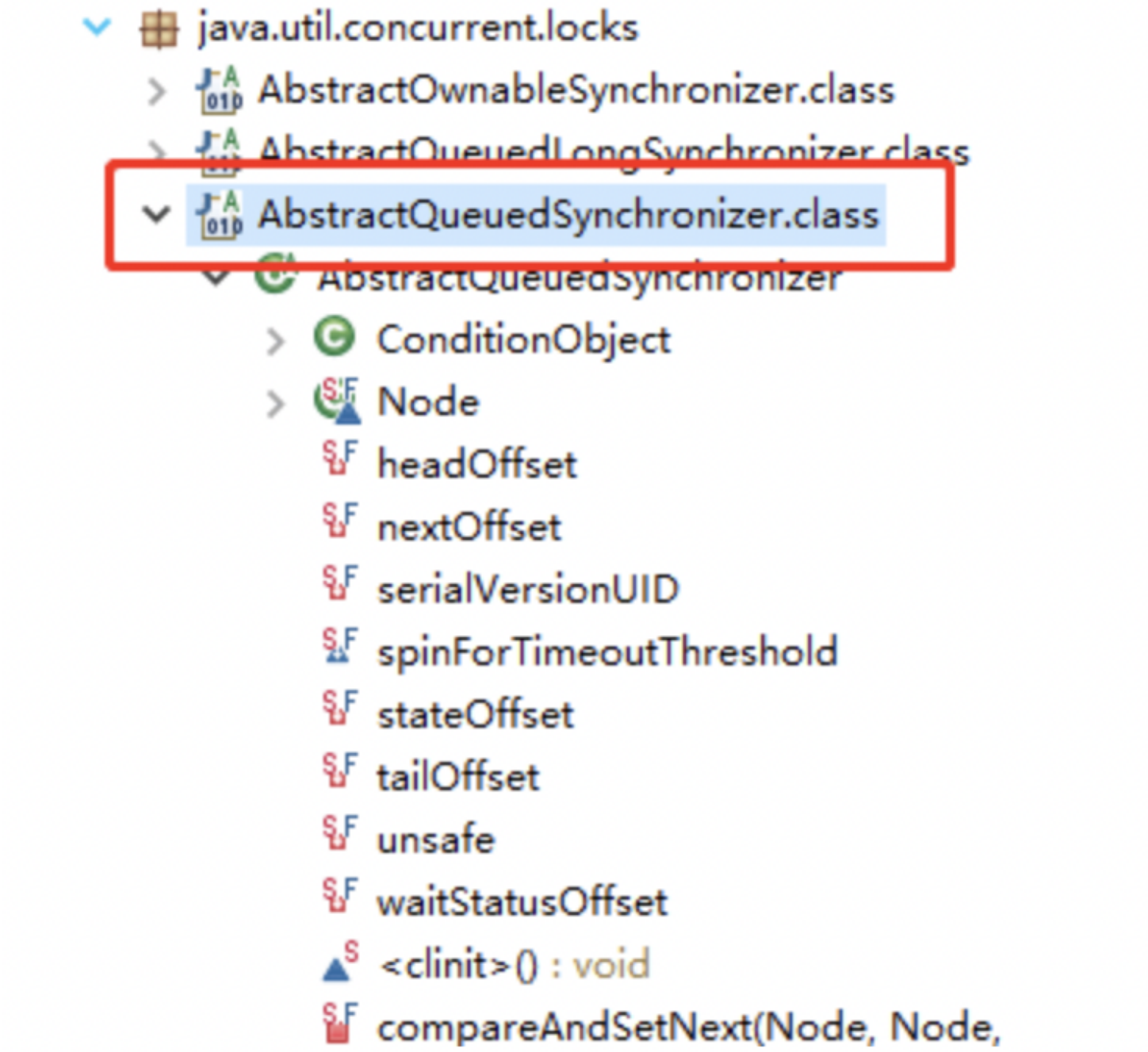
AQS学习

AQS 的全称为 AbstractQueuedSynchronizer



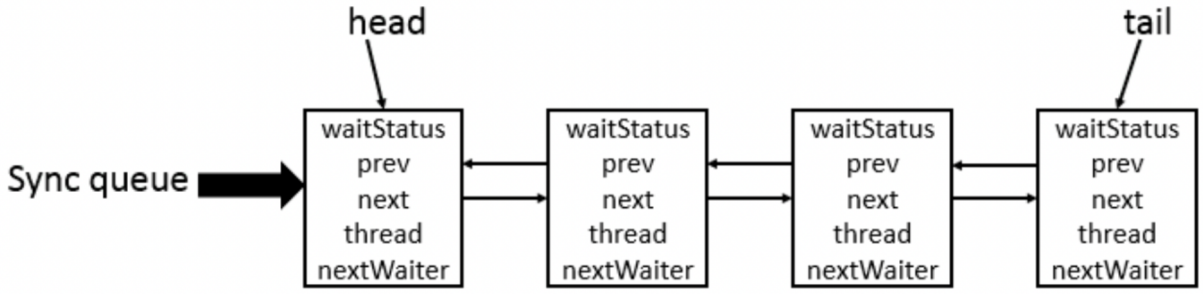
AQS是是一个抽象类，主要用来构建锁和同步器。后续在锁里会大量用大

ReentrantLock，Semaphore，其他的诸如 ReentrantReadWriteLock，SynchronousQueue等等皆是基于 AQS 的

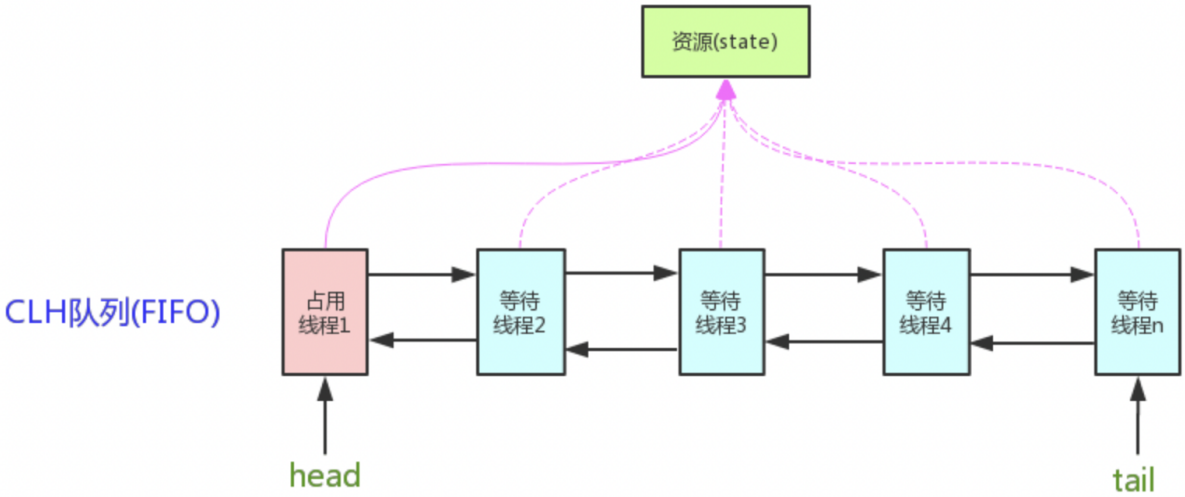
```
public abstract class AbstractQueuedSynchronizer extends AbstractOwnableSynchronizer im
```

AQS原理：如果被请求的资源空闲，则将当前线程设置为有效的工作线程，并将请求的共享资源设置为占用状态，但是若资源处于被占用的状态，那么就需要一套机制来实现等待和唤醒的操作，AQS就是使用CLH队列来实现（虚拟双向队列）：**将每一个请求资源的线程封装成一个队列节点**

CLH 队列锁结构如下图所示：



它保存着线程的引用（thread）、 当前节点在队列中的状态（waitStatus）、前驱节点（prev）、后继节点（next）



第一个核心点state

AQS 使用 int 成员变量 **state 表示同步状态**，通过内置的 线程等待队列 来完成获取资源线程的排队工作。

state 变量由 **volatile** 修饰，用于展示当前临界资源的获锁情况。

```
// 共享变量，使用volatile修饰保证线程可见性
private volatile int state;
```

另外，状态信息 state 可以通过 **protected** 类型的**getState()、setState()和compareAndSetState() 进行操作**。并且，这几个方法都是 final 修饰的，**在子类中无法被重写**。

```
//返回同步状态的当前值
protected final int getState() {
     return state;
}
 // 设置同步状态的值
protected final void setState(int newState) {
     state = newState;
}
//原子地（CAS操作）将同步状态值设置为给定值update如果当前同步状态的值等于expect（期望值）
protected final boolean compareAndSetState(int expect, int update) {
      return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
}
```

state 初始状态是0，加锁+1.调用方法tryAcquire枷锁，tryRelease释放锁
当然CountDownLatch这种，将任务分给多个线程执行，会一开始将state设置为n，每次countDown都会减少state，tryrelease，直到减少至0.await才会执行后续操作

第二个核心点tryAcquire-tryRelease、tryAcquireShared-tryReleaseShared

AQS两种资源共享方式：Exclusive和Share；一般来说，自定义同步器的共享方式要么是独占，要么是共享，他们也只需实现tryAcquire-tryRelease、tryAcquireShared-tryReleaseShared中的一种即可。但 AQS 也支持自定义同步器同时实现独占和共享两种方式，如**ReentrantReadWriteLock** 锁的篇幅中必须学会的一点。

常见的同步工具类

Semaphore(信号量)

synchronized 和 ReentrantLock 都是一次只允许一个线程访问某个资源，而Semaphore(信号量)可以用来控制同时访问特定资源的线程数量。
可以声明为公平模式或者非公平模式

```
public Semaphore(int permits) {
        sync = new NonfairSync(permits);
}

public Semaphore(int permits, boolean fair) {
        sync = fair ? new FairSync(permits) : new NonfairSync(permits);
}
```

Semaphore 是共享锁的一种实现，它默认构造 AQS 的 state 值为 permits，你可以将 permits 的值理解为许可证的数量，只有拿到许可证的线程才能执行。

```
/**
 *
 * @author Snailclimb
 * @date 2018年9月30日
 * @Description: 需要一次性拿一个许可的情况
 */
public class SemaphoreExample1 {
  // 请求的数量
  private static final int threadCount = 550;

  public static void main(String[] args) throws InterruptedException {
    // 创建一个具有固定线程数量的线程池对象（如果这里线程池的线程数量给太少的话你会发现执行的很慢）
    ExecutorService threadPool = Executors.newFixedThreadPool(300);
    // 初始许可证数量
    final Semaphore semaphore = new Semaphore(20);

    for (int i = 0; i < threadCount; i++) {
      final int threadnum = i;
      threadPool.execute(() -> {// Lambda 表达式的运用
        try {
          semaphore.acquire();// 获取一个许可，所以可运行线程数量为20/1=20
          test(threadnum);
          semaphore.release();// 释放一个许可
        } catch (InterruptedException e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
        }

      });
    }
    threadPool.shutdown();
    System.out.println("finish");
  }

  public static void test(int threadnum) throws InterruptedException {
    Thread.sleep(1000);// 模拟请求的耗时操作
    System.out.println("threadnum:" + threadnum);
    Thread.sleep(1000);// 模拟请求的耗时操作
  }
}
```

CountDownLatch

CountDownLatch 允许 count 个线程阻塞在一个地方，直至所有线程的任务都执行完毕。

CountDownLatch 是一次性的，计数器的值**只能在构造方法中初始化一次**，之后没有任何机制再次对其

设置值，当 CountDownLatch 使用完毕后，它不能再次被使用。

CountDownLatch就是将state的count设置为N，然后没吃countdown就trylease，最后state = 0 执行主方法

两种使用场景

1. 多个线程跑完以后主线程开始跑。类似多个组件加载完成以后，主线程就可以跑
2. 想让多个任务同时开始，并行开跑，这个时候就要将主线程的count设置为1，当主线程countdown以后，await会唤醒所有的子线程同时开跑

```
/**
 *
 * @author SnailClimb
 * @date 2018年10月1日
 * @Description: CountDownLatch 使用方法示例
 */
public class CountDownLatchExample1 {
  // 请求的数量
  private static final int threadCount = 550;

  public static void main(String[] args) throws InterruptedException {
    // 创建一个具有固定线程数量的线程池对象（如果这里线程池的线程数量给太少的话你会发现执行的很慢）
    ExecutorService threadPool = Executors.newFixedThreadPool(300);
    final CountDownLatch countDownLatch = new CountDownLatch(threadCount);
    for (int i = 0; i < threadCount; i++) {
      final int threadnum = i;
      threadPool.execute(() -> {// Lambda 表达式的运用
        try {
          test(threadnum);
        } catch (InterruptedException e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
        } finally {
          countDownLatch.countDown();// 表示一个请求已经被完成
        }

      });
    }
    countDownLatch.await();
    threadPool.shutdown();
    System.out.println("finish");
  }

  public static void test(int threadnum) throws InterruptedException {
    Thread.sleep(1000);// 模拟请求的耗时操作
    System.out.println("threadnum:" + threadnum);
    Thread.sleep(1000);// 模拟请求的耗时操作
  }
}
```

注意别死锁了，导致countdownlunch的大小永远减少不到0

CyclicBarrier(循环栅栏)

CyclicBarrier 的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是：让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，**屏障才会开门，所有被屏障拦截的线程才会继续干活**。

执行流程

1. CyclicBarrier 默认的构造方法是 CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，*每个线程调用 await()*方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞。

2. 当调用 CyclicBarrier 对象调用 await() 方法时，实际上调用的是 dowait(false, 0L)方法。 await() 方法就像树立起一个栅栏的行为一样，将线程挡住了，当拦住的线程数量达到 parties 的值时，栅栏才会打开，线程才得以通过执行。

```java
/**
 *
 * @author Snailclimb
 * @date 2018年10月1日
 * @Description: 测试 CyclicBarrier 类中带参数的 await() 方法
 */
public class CyclicBarrierExample1 {
  // 请求的数量
  private static final int threadCount = 550;
  // 需要同步的线程数量
  private static final CyclicBarrier cyclicBarrier = new CyclicBarrier(5);

  public static void main(String[] args) throws InterruptedException {
    // 创建线程池
    ExecutorService threadPool = Executors.newFixedThreadPool(10);

    for (int i = 0; i < threadCount; i++) {
      final int threadNum = i;
      Thread.sleep(1000);
      threadPool.execute(() -> {
        try {
          test(threadNum);
        } catch (InterruptedException e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
        } catch (BrokenBarrierException e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
        }
      });
    }
    threadPool.shutdown();
  }

  public static void test(int threadnum) throws InterruptedException, BrokenBarrierExce
    System.out.println("threadnum:" + threadnum + "is ready");
    try {
      /**等待60秒，保证子线程完全执行结束*/
      cyclicBarrier.await(60, TimeUnit.SECONDS);
    } catch (Exception e) {
      System.out.println("-----CyclicBarrierException------");
    }
    System.out.println("threadnum:" + threadnum + "is finish");
  }

}
```

总结下：

countdownlunch：类似于所有组件加载，最后主流程进行

ccyclicBarrier：类似于我需要此时执行的只有我指定的线程数。很向信号量，与信号量的区别又是，cycleBarrier会让你的线程要在同一时刻去跑

第二个特性：

另外，CyclicBarrier 还提供一个更高级的构造函数 CyclicBarrier(int parties, Runnable **barrierAction**)，**用于在线程到达屏障时，优先执行 barrierAction**，方便处理更复杂的业务场景。

```java
/**
 *
 * @author SnailClimb
 * @date 2018年10月1日
 * @Description: 新建 CyclicBarrier 的时候指定一个 Runnable
 */
public class CyclicBarrierExample2 {
  // 请求的数量
  private static final int threadCount = 550;
  // 需要同步的线程数量
  private static final CyclicBarrier cyclicBarrier = new CyclicBarrier(5, () -> {
    System.out.println("------当线程数达到之后，优先执行------");
  });

  public static void main(String[] args) throws InterruptedException {
    // 创建线程池
    ExecutorService threadPool = Executors.newFixedThreadPool(10);

    for (int i = 0; i < threadCount; i++) {
      final int threadNum = i;
      Thread.sleep(1000);
      threadPool.execute(() -> {
        try {
          test(threadNum);
        } catch (InterruptedException e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
        } catch (BrokenBarrierException e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
        }
      });
    }
    threadPool.shutdown();
  }

  public static void test(int threadnum) throws InterruptedException, BrokenBarrierExce
    System.out.println("threadnum:" + threadnum + "is ready");
    cyclicBarrier.await();
    System.out.println("threadnum:" + threadnum + "is finish");
  }

}
```