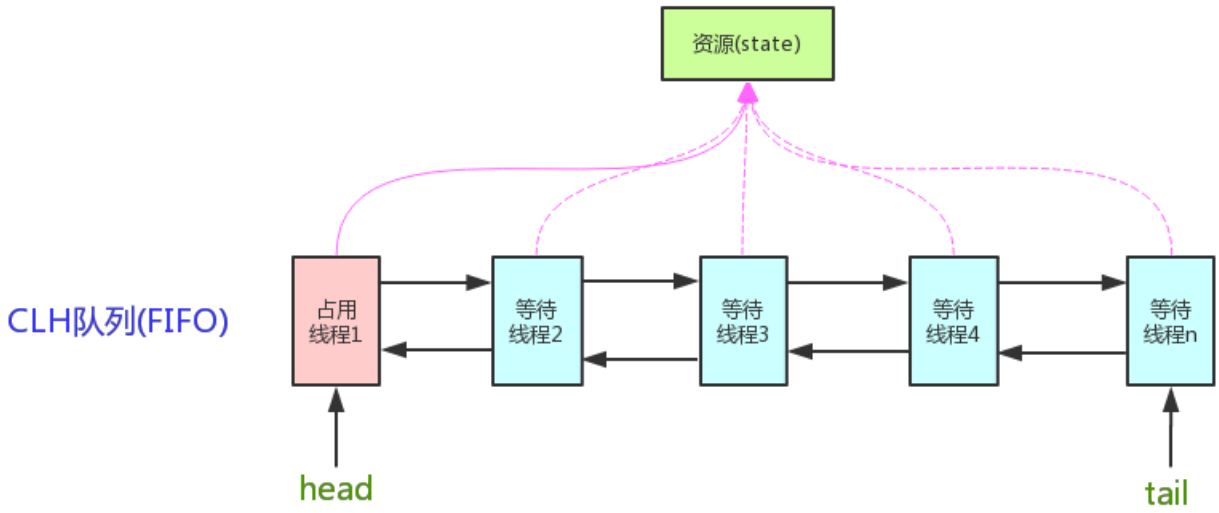
AQS原理以及AQS同步组件总结

AQS简单介绍

全称AbstractQueuedSynchronized AQS是一个用来构建锁和同步器的框架。使用AQS能简单且高效的构造出应用广泛分的大型同步器，比如我们提到的ReentrantLock，Semaphore等

AQS原理

AQS核心思想就是，如果被请求访问的资源空闲，则将当前请求资源的线程设置为有效的工作线程，将该资源状态变为锁定。如果被请求访问的资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配机制，这个机制AQS是用CLH队列实现的，即将暂时获取不到锁的资源加入到队列中。



AQS 使用一个 int 成员变量来表示同步状态，通过内置的 FIFO 队列来完成获取资源线程的排队工作。AQS 使用 CAS 对该同步状态进行原子操作实现对其值的修改。

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

具体的状态state信息通过getState，setState，compareAndSetState进行操作

```
//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);
}
```

AQS对资源的共享方式

Exclusive （独占） 只有一个线程能执行，如 ReentrantLock。又可分为公平锁和非公平锁,ReentrantLock 同时支持两种锁

ReentrantLock 默认采用非公平锁，因为考虑获得更好的性能，通过 boolean 来决定是否用公平锁（传入 true 用公平锁）。

```
private final Sync sync;
public ReentrantLock() {
    sync = new NonfairSync();
}
public ReentrantLock(boolean fair){
    sync = fair ? new FairSync() : new NonfairSync();
}
```

公平锁的lock方法：

```
static final class FairSync extends Sync {
    final void lock() {
        acquire(1);
    }
    public final void acquire(int arg){
        if(!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
    protected final boolean tryAcquire(int acquires) {
        final Thread current = Thread.currentThread();
        int c = getState();
        if (c == 0 ){
            // 和非公平锁相比多一个判断，是否有线程在等待
            if(! hasQueuedPredecessors() && compareAndSetState(0,acquires)) {
                setExclusiveOwnerThread(current);
                return true;
            }
        }
        else if(current == getExclusiveOwnerThread()){
            int nextc = c + acquires;
            if(nextc < 0){
                throws new Error("Maximum lock count exceeded");
            }
            setState(nextc);
```

```
            return true;
        }
        return false;
    }
}
```

非公平的lock方法

```
static final class NonfairSync extends Sync {
    final void lock() {
        // 2. 和公平锁相比，这里会直接先进行一次CAS，成功就返回了
        if (compareAndSetState(0, 1))
            setExclusiveOwnerThread(Thread.currentThread());
        else
            acquire(1);
    }
    // AbstractQueuedSynchronizer.acquire(int arg)
    public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
    protected final boolean tryAcquire(int acquires) {
        return nonfairTryAcquire(acquires);
    }
}
/**
* Performs non-fair tryLock.  tryAcquire is implemented in
* subclasses, but both need nonfair try for trylock method.
*/
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        // 这里没有对阻塞队列进行判断
        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;
}
```

公平和非公平锁只有两处不同：

1. 非公平锁在调用lock后，首先就会调用CAS进行一次抢锁，如果这个时候恰好说没有被占用，那么久直接获取到锁返回了。
2. 非公平锁在调用CAS失败后，和公平锁一样会进入到tryAcquire方法，在tryAcquire方法中，如果发现锁这个时候被释放了（state=0），非公平锁会直接CAS抢锁，但是公平锁会判断等待队列是否有现成处于等待状态，如果有则不抢锁，排在后面。

Share （共享）

多个线程同时执行，如Semaphore/CountDownLatch。Semaphore、CountDownLatCh、 CyclicBarrier、ReadWriteLock 我们都会在后面讲到。

AQS底层使用了模板方法模式

同步器的设计是基于模板方法模式的，如果需要自定义同步器一般的方法是这样的。 1、使用者继承AbstractQueuedSynchronizer并重写指定的方法。 2、将AQS组合在自定义同步组件的实现中，并调用其模板方法，而这些模板方法会调用使用者的重写的方法

模板方法模式是基于"继承"的，主要是为了在不改变模板结构的前提下在子类中重新定义模板中的内容以实现复用代码。举个很简单的例子假如我们要去一个地方的步骤是：购票buyTicket()->安检securityCheck()->乘坐某某工具回家ride()->到达目的地arrive()。我们可能乘坐不同的交通工具回家比如飞机或者火车，所以除了ride()方法，其他方法的实现几乎相同。我们可以定义一个包含了这些方法的抽象类，然后用户根据自己的需要继承该抽象类然后修改 ride()方法。

AQS使用了模板方法模式，自定义同步器时需要重写下面几个AQS提供的模板方法：

```
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。
```

默认情况下每个方法中都抛出UnsupportedOperationException.这些方法的实现必须是内部线程安全的，并且通常应该是简短而不是阻塞。AQS勒种其他方法都是final，无权被其他类使用。

以 CountDownLatch 以例，任务分为 N 个子线程去执行，state 也初始化为 N（注意 N 要与线程个数一致）。这 N 个子线程是并行执行的，每个子线程执行完后 countDown()一次，state 会 CAS(Compare and Swap)减 1。等到所有子线程都执行完后(即 state=0)，会 unpark()主调用线程，然后主调用线程就会从 await()函数返回，继续后余动作

Semaphore(信号量) 允许多个线程同时访问

synchronized和ReentrantLock都是一次只允许一个线程访问某个资源，Semaphore（信号量）可以指定多个线程同时访问某个资源。 示例： 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（如果这里线程池数量给太少的话，你会发现执行的很慢。
        ExecutorService threadPool = Executors.newFixedThreadPool(300);
//          一次只能允许执行的线程数量
        final Semaphore semaphore = new Semaphore(20);

        for(int i = 0;i < threadCount; i++){
            final int threadnum = i;
            threadPool.execute(()->{
                try {
                    semaphore.acquire();//获取一个许可，所以可运行的数量为20/1=20
                    test(threadnum);
                    semaphore.release();//释放一个许可
                }catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
        }
        threadPool.shutdown();
        System.out.println("finish");
    }
    public static void test(int threadnum) throws InterruptedException {
        Thread.sleep(1000);
        System.out.println("threadnum: " + threadnum);
        Thread.sleep(1000);
    }
}
```

执行acquire方法阻塞，知道一个许可证可以获得，然后拿走一个许可证；每个release方法增加一个许可证，这可能会释放一个阻塞的acquire方法，然而其实其实并没有实际的许可证这个对象，Semaphore 只是维持了一个可获得许可证的数量。 Semaphore 经常用于限制获取某种资源的线程数量

CountDownLatch(倒计时器)

同步工具类，允许一个或者多个线程一直等待，知道其他线程的操作执行完之后在执行。

CountDownLatch的三种典型用法

1. 某个线程再开始运行前等待n个线程执行完毕。将CountDownLatch的计数器初始化为n：new CountDownLatch(n); 每当一个任务线程执行完毕，就将计数器减 1 countdownlatch.countDown()，当计数器的值变为 0 时，在CountDownLatch上 await() 的线程就会被唤醒。一个典型应用场景就是启动一个服务时，主线程需要等待多个组件加载完毕，之后再继续执行。

2. 实现多个线程开始执行任务的最大并行性。注意是并行性，不是并发，强调的是多个线程在某一时刻同时开始执行，类似于赛跑，将夺冠线程放到起点，等待发令枪响，然后同时开跑。做法是初始化一个共享的 CountDownLatch对象，将其计数器初始化为1：new CountDownLatch(1),多个线程在开始执行前首先countdownLatch.await(),当主线程调用countDown时，计数器变为0，多个线程同时被唤醒。

3. 死锁检测:一个非常方便的使用场景就是，你可以使用n个线程访问共享资源，在每次测试阶段的线程数目是不同的，并尝试产生死锁。

使用示例

```
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
        ExecutorService threadPool = Executors.newFixedThreadPool(300);
        final CountDownLatch countDownLatch = new CountDownLatch(threadCount);
        for(int i = 0; i < threadCount;i++){
            final int threadnum = i;
            threadPool.execute(()->{
                try{
                    test(threadnum);
                }catch (InterruptedException e) {
                    e.printStackTrace();
                }finally {
                    countDownLatch.countDown();
                }
            });
        }
        countDownLatch.await();
        threadPool.shutdown();
        System.out.println("finish");
    }
    public static void test(int threadnum) throws InterruptedException {
        Thread.sleep(1000);
        System.out.println("threadnum: " + threadnum);
        Thread.sleep(1000);
    }
}
```

上面的代码中，我们定义了请求的数量为 550，当这 550 个请求被处理完成之后，才会执行System.out.println("finish");。

主线程在启动其他线程后立即调用CountDownLatch.await()方法，这样主线程就在这个方法上阻塞。

与 CountDownLatch 的第一次交互是主线程等待其他线程。主线程必须在启动其他线程后立即调用CountDownLatch.await()方法。这样主线程的操作就会在这个方法上阻塞，直到其他线程完成各自的任务。

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

CyclicBarrier（循环栅栏）

使用场景和CountDownLatch非常相似，CyclicBarrier字面意思可循环使用的屏障，要做的事情就是在一组线程到达一个屏障也可以叫同步点时被阻塞，知道最后一个线程到达屏障，屏障才会开门，所有被屏障拦截的线程才会继续干活。 默认的构造方法是CyclicBarrier(int parties) 参数是屏障拦截的线程数量，每个线程调用await方法告诉CyclicBarrier，我已经到达屏障，然后当前线程阻塞。

构造函数：

```
public CyclicBarrier(int parties) {
    this(parties, null);
}

public CyclicBarrier(int parties, Runnable barrierAction) {
    if (parties <= 0) throw new IllegalArgumentException();
    this.parties = parties;
    this.count = parties;
    this.barrierCommand = barrierAction;
}
```

CyclicBarrier可以用于多线程计算数据，最后合并计算结果的应用场景。比如我们用Excel保存了用户所有银行流水，每一个sheet中保存一个账户近一年的每笔银行流水，现在需要统计用户的日均银行流水，先用多线程处理每个sheet里的银行流水，都执行完之后，得到每个sheet的日均银行流水，最后再用barrierAction用这些线程的计算结果，计算出整个Excel的日均银行流水。

CyclicBarrier使用示例

```
public class CyclicBarrierExample2 {
    private static final int threadCount = 550;
    private static final CyclicBarrier cyclicBarrier = new CyclicBarrier(5);

    public static void main(String[] args) throws  InterruptedException{
        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){
                    e.printStackTrace();
                }catch (BrokenBarrierException e){
                    e.printStackTrace();
                }
            });
        }
        threadPool.shutdown();
    }
```

```
    public static void test(int threadnum) throws
InterruptedException,BrokenBarrierException{
        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");
    }
}
```

可以看到当线程数量也就是请求数量到达定义的5个时候，await方法之后的方法才会被执行。

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

```
public class CyclicBarrierExample3 {
    private static final int threadCount = 550;
    private static final CyclicBarrier cyclicBarrier = new CyclicBarrier(5,()->{
        Random random = new Random();
        int c = random.nextInt(100);
        System.out.println("------当线程到达之后，优先执行-------" + c); //线程到达栅
栏之后优先执行
    });

    public static void main(String[] args) throws InterruptedException{
        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 | BrokenBarrierException e){
                    e.printStackTrace();
                }
            });
        }
        threadPool.shutdown();
    }
    public static void test(int threadnum) throws
InterruptedException,BrokenBarrierException{
        System.out.println("threadnum:" + threadnum + "is ready");
        cyclicBarrier.await();
        System.out.println("threadnum:" + threadnum + "is finish");
    }
}
```

CyclicBarrier源码分析

当调用CyclicBarrier对象调用await()方法时，实际调用的是dowait(false,0L),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。

```
public int await() throws InterruptedException, BrokenBarrierException {
    try {
        return dowait(false, 0L);
    } catch (TimeoutException toe) {
        throw new Error(toe); // cannot happen
    }
}
// 当线程数量或者请求数量达到 count 时 await 之后的方法才会被执行。上面的示例中 count 的值就为 5。
private int count;
/**
 * Main barrier code, covering the various policies.
 */
private int dowait(boolean timed, long nanos)
    throws InterruptedException, BrokenBarrierException,TimeoutException {
    final ReentrantLock lock = this.lock;
    // 锁住
    lock.lock();
    try {
        final Generation g = generation;

        if (g.broken)
            throw new BrokenBarrierException();

        // 如果线程中断了，抛出异常
        if (Thread.interrupted()) {
            breakBarrier();
            throw new InterruptedException();
        }
        // cout减1
        int index = --count;
        // 当 count 数量减为 0 之后说明最后一个线程已经到达栅栏了，也就是达到了可以执行 await 方法之后的条件
        if (index == 0) {  // tripped
            boolean ranAction = false;
            try {
                final Runnable command = barrierCommand;
                if (command != null)
                    command.run();
                ranAction = true;
                // 将 count 重置为 parties 属性的初始化值
                // 唤醒之前等待的线程
                // 下一波执行开始
                nextGeneration();
```

```
                return 0;
            } finally {
                if (!ranAction)
                    breakBarrier();
            }
        }

        // loop until tripped, broken, interrupted, or timed out
        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();
                throw new TimeoutException();
            }
        }
    } finally {
        lock.unlock();
    }
}
```

CyclicBarrier内部通过一个count变量作为计数器，count初始值为parties的初始化值，每当一个线程执行到栅栏就将计数器减一，如果count值为0，表示这一代最后一个线程到达栅栏，就尝试执行我们构造方法中的任务。

CyclicBarrier 和 CountDownLatch 的区别

对于CountDownLatch来说，终点是一个线程或多个线程等待，而其他N个线程在完成某件事情之后，可以终止也可以等待，而对于CyclicBarrier，重点是多个线程在任意一个线程没有完成，所有的线程都必须等待。

ReentrantLock 和 ReentrantReadWriteLock

读写锁 ReentrantReadWriteLock 可以保证多个线程可以同时读，所以在读操作远大于写操作的时候，读写锁就非常有用了。