AQS抽象队列同步器

CountDownLatch

1. java1.5被引入的一个工具类，常被称为：倒计数器。创建对象时，传入指定数值作为线程参与的数量。await方法等待计数器值变为0，在这之前，线程进入等待状态。countdown方法计数器值减一，直到为0.经常用于等待其他线程执行到某一节点，再继续执行当前线程代码。
2. 使用场景
    - 统计线程执行的情况。
    - 压力测试中，使用countDownLatch实现最大程度的并发处理。
    - 多个线程之间互相通信，比如线程异步调用完接口，结果通知。

```
public class CountDownLatchDemo {
    AtomicInteger count;
    LinkedBlockingQueue<Thread> waiters = new LinkedBlockingQueue<>();

    public CountDownLatchDemo(int num) {
        this.count = new AtomicInteger(num);
    }

    public void await() {
        // 进入等待列表
        waiters.add(Thread.currentThread());
        while (this.count.get() != 0) {
            // 挂起线程
            LockSupport.park();
        }
        waiters.remove(Thread.currentThread());
    }

    public void countDown() {
        if (this.count.decrementAndGet() == 0) {
            Thread waiter = waiters.peek();
            LockSupport.unpark(waiter); // 唤醒线程继续 抢锁
        }
    }


    public static void main(String[] args) throws InterruptedException {
        // 一个请求，后台需要调用多个接口 查询数据
        CountDownLatchDemo cdLdemo = new CountDownLatchDemo(10); // 创建，计数数值
        for (int i = 0; i < 10; i++) { // 启动九个线程，最后一个两秒后启动
            int finalI = i;
            new Thread(() -> {
                try {
                    Thread.sleep(2000L);
                } catch (InterruptedException e) {
                    e.printStackTrace();
```

```
                }
                System.out.println("我是" + Thread.currentThread() + ".我执行接口-" + finalI
+ "调用了");
                cdLdemo.countDown(); // 参与计数
                // 不影响后续操作
            }).start();
        }

        cdLdemo.await(); // 等待计数器为0
        System.out.println("全部执行完毕.我来召唤神龙");
    }
}
```

CyclicBarrier

1. 1.5加入的，称为线程栅栏。
2. 创建对象时，制定栅栏线程数量。await方法等指定数量的线程都处于等待状态时，继续执行后续代码。barrierAction线程数量到了指定量之后，自动触发执行指定任务。和CountDownLatch重要区别在于，CyclicBarrier对象可以多次触发执行。
3. 应用场景：
    - 数据量比较大时，实现批量插入数据到数据库。
    - 数据统计，30个线程统计30天数据，全部统计完毕后，执行汇总。
    - 拼团。

```
public class CyclicBarrierTest {
    public static void main(String[] args) throws InterruptedException {
        LinkedBlockingQueue<String> sqls = new LinkedBlockingQueue<>();
        // 任务1+2+3...1000  拆分为100个任务（1+..10,  11+20） -> 100线程去处理。

        // 每当有4个线程处于await状态的时候，则会触发barrierAction执行
        CyclicBarrier barrier = new CyclicBarrier(4, new Runnable() {
            @Override
            public void run() {
                // 这是每满足4次数据库操作，就触发一次批量执行
                System.out.println("有4个线程执行了，开始批量插入： " + Thread.currentThread());
                for (int i = 0; i < 4; i++) {
                    System.out.println(sqls.poll());
                }
            }
        });

        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                try {
                    sqls.add("data - " + Thread.currentThread()); // 缓存起来
                    Thread.sleep(1000L); // 模拟数据库操作耗时
                    barrier.await(); // 等待栅栏打开,有4个线程都执行到这段代码的时候，才会继续往下执
行
                    System.out.println(Thread.currentThread() + "插入完毕");
                } catch (Exception e) {
                    e.printStackTrace();
```

```
                }
            }).start();
        }

        Thread.sleep(2000);
    }
}
```

Semaphore

1. 又称信号量，控制多个线程争抢许可。
2. acquire方法获取一个许可，如果没有就等待，release方法释放一个许可。availablePermits方法得到可用许可的数目。
3. 应用场景：
    - 代码并发处理限流

```
public class SemaphoreDemo {
    public static void main(String[] args) {
        SemaphoreDemo semaphoreTest = new SemaphoreDemo();
        int N = 9;            // 客人数量
        MySemaphore semaphore = new MySemaphore(5); // 手牌数量，限制请求数量
        for (int i = 0; i < N; i++) {
            String vipNo = "vip-00" + i;
            new Thread(() -> {
                try {
                    semaphore.acquire(); // 获取令牌,没拿到的就等
                    System.out.println(semaphore.count);
                    semaphoreTest.service(vipNo); // 实现了怼service方法的限流

                    semaphore.release(); // 释放令牌,令牌数+1
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }).start();
        }
    }

    // 限流 控制5个线程 同时访问
    public void service(String vipNo) throws InterruptedException {
        System.out.println("楼上出来迎接贵宾一位，贵宾编号" + vipNo + "，...");
        Thread.sleep(new Random().nextInt(3000));
        System.out.println("欢送贵宾出门，贵宾编号" + vipNo);
    }

}
```

```
public class MySemaphore {

    private volatile AtomicInteger count = new AtomicInteger();
```

```
    private volatile Queue<Thread> waiters = new LinkedBlockingQueue<>();

    public void acquire() {
        Thread currentThread = Thread.currentThread();
        waiters.add(currentThread);
        for (;;) {
            int current = count.get();
            int next = current - 1;
            if (current <= 0 || next < 0) {
                LockSupport.park(currentThread);
            }

            if (count.compareAndSet(current, next)) {
                break;
            }
        }
        waiters.remove(currentThread);
    }

    public void release() {
        if (count.incrementAndGet() > 0) {
            for (Thread waiter : waiters) {
                LockSupport.unpark(waiter);
            }
        }
    }
}
```

同步锁的本质-排队

1. 同步的方式：独享锁-单个队列窗口，共享锁-多个队列窗口。
2. 抢锁的方式：插队抢(不公平锁)，先来后到抢锁(公平锁)。
3. 没抢到锁的处理方式：快速尝试多次(CAS自旋锁)，阻塞等待。
4. 唤醒阻塞线程的方式(叫号器)：全部通知，通知下一个。

AQS抽象队列同步器

通过自己实现ReentrantLock\CountDownLatch\Semaphore，可以发现在获取资源时和释放资源时，这三个类 有类似的处理逻辑，将相同的部分进行抽象，可以得到一个简易的抽象队列同步器。

```
public class AQSdemo {
    // 同步资源状态
    volatile AtomicInteger state = new AtomicInteger(0);
    // 当前锁的拥有者
    protected volatile AtomicReference<Thread> owner = new AtomicReference<>();
    // java q 线程安全
    public volatile LinkedBlockingQueue<Thread> waiters = new LinkedBlockingQueue<>();

    // 独占
    public void acquire() {
        // 塞到等待锁的集合中
        waiters.offer(Thread.currentThread());
```

```
        while (!tryAcquire()) {
            // 挂起这个线程
            LockSupport.park();
        }
        // 后续，等待其他线程释放锁，收到通知之后继续循环
        waiters.remove(Thread.currentThread());
    }

    public void release() {
        // cas 修改 owner 拥有者
        if (tryRelease()) {
            Thread waiter = waiters.peek();
            LockSupport.unpark(waiter); // 唤醒线程继续 抢锁
        }
    }

    public boolean tryAcquire() {
        throw new UnsupportedOperationException();
    }

    public boolean tryRelease() {
        throw new UnsupportedOperationException();
    }


    // 共享资源获取
    public void acquireShared() {
        // 塞到等待锁的集合中
        waiters.offer(Thread.currentThread());
        while (tryAcquireShared() < 0) {
            // 挂起这个线程
            LockSupport.park();
        }
        // 后续，等待其他线程释放锁，收到通知之后继续循环
        waiters.remove(Thread.currentThread());
    }

    // 共享资源的释放
    public void releaseShared() {
        // cas 修改 owner 拥有者
        if (tryReleaseShared()) {
            Thread waiter = waiters.peek();
            LockSupport.unpark(waiter); // 唤醒线程继续 抢锁
        }
    }


    public int tryAcquireShared() {
        throw new UnsupportedOperationException();
    }

    public boolean tryReleaseShared() {
        throw new UnsupportedOperationException();
```

```
    }

    public AtomicInteger getState() {
        return state;
    }

    public void setState(AtomicInteger state) {
        this.state = state;
    }
}
```

1. 提供了对资源占用，释放，线程的等待，唤醒等接口和具体实现。
2. 可以用在各种需要控制资源争用的场景中。(ReentrantLock\CountDownLatch\Semphore)。
3. acquire，acquireShared：定义了资源争用的逻辑，如果没拿到，则等待。tryAcqure，tryAcquireShared：实际执行占用资源的操作，如何判定一个由使用者具体去实现。release，releaseShared：定义释放资源的逻辑，释放之后，通知后续节点进行争抢。tryRelease，tryReleaseShared：实际执行资源释放的操作，具体的使用者去实现。

委托AQS实现

1. ReentrantLock

```
public class NeteaseReadWriteLock implements ReadWriteLock {
    AQSdemo aqSdemo = new AQSdemo() {
        @Override
        public boolean tryAcquire() {
            // 有读的时候，不能写
            if (aqSdemo.getState().get() != 0) {
                return false;
            } else {
                return owner.compareAndSet(null, Thread.currentThread());
            }
        }

        @Override
        public boolean tryRelease() {
            return owner.compareAndSet(Thread.currentThread(), null);
        }

        @Override
        public boolean tryReleaseShared() {
```

```
            return aqSdemo.getState().decrementAndGet() >= 0;
        }

        // 加读锁
        @Override
        public int tryAcquireShared() {
            // 如果当前有线程占用了写锁，则不允许再加锁，除非是同一个线程
            if (owner.get() != null && !owner.get().equals(Thread.currentThread())) {
                return -1;
            }
            return aqSdemo.getState().incrementAndGet();
        }
    };

    @Override
    public Lock readLock() {
        return new Lock() {
            @Override
            public void lock() {
                aqSdemo.acquireShared();
            }

            @Override
            public void lockInterruptibly() throws InterruptedException {

            }

            @Override
            public boolean tryLock() {
                return false;
            }

            @Override
            public boolean tryLock(long time, TimeUnit unit) throws
InterruptedException {
                return false;
            }

            @Override
            public void unlock() {
                aqSdemo.releaseShared();
            }

            @Override
            public Condition newCondition() {
                return null;
            }
        };
    }

    @Override
    public Lock writeLock() {
        return new Lock() {
```

```
            @Override
            public void lock() {
                aqSdemo.acquire();
            }

            @Override
            public void lockInterruptibly() throws InterruptedException {

            }

            @Override
            public boolean tryLock() {
                return aqSdemo.tryAcquire();
            }

            @Override
            public boolean tryLock(long time, TimeUnit unit) throws
InterruptedException {
                return false;
            }

            @Override
            public void unlock() {
                aqSdemo.release();
            }

            @Override
            public Condition newCondition() {
                return null;
            }
        };
    }
}
```

2. CountDownLatch

```
public class CDLdemo {
    // Object  AQS = new Oject(state, queue);
    // AQS 具体实现对象（state、queue、owner）
    AQSdemo aqSdemo = new AQSdemo() {
        @Override
        public int tryAcquireShared() { // 如果非等于0，代表当前还有线程没准备就绪，则认为需要
等待
            return this.getState().get() == 0 ? 1 : -1;
        }

        @Override
        public boolean tryReleaseShared() { // 如果非等于0，代表当前还有线程没准备就绪，则不会
通知继续执行
            return this.getState().decrementAndGet() == 0;
        }
    };
```

```
    public CDLdemo(int count) {
        aqSdemo.setState(new AtomicInteger(count));
    }

    public void await() {
        aqSdemo.acquireShared();
    }

    public void countDown() {
        aqSdemo.releaseShared();
    }

    public static void main(String[] args) throws InterruptedException {
        // 一个请求，后台需要调用多个接口 查询数据
        CDLdemo cdLdemo = new CDLdemo(10); // 创建，计数数值
        for (int i = 0; i < 10; i++) { // 启动九个线程，最后一个两秒后启动
            int finalI = i;
            new Thread(() -> {
                try {
                    Thread.sleep(2000L);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("我是" + Thread.currentThread() + ".我执行接口-" +
finalI +"调用了");
                cdLdemo.countDown(); // 参与计数
                // 不影响后续操作
            }).start();
        }

        cdLdemo.await(); // 等待计数器为0
        System.out.println("全部执行完毕.我来召唤神龙");
    }
}
```

3. Semaphore

```
public class NeteaseSemaphore {
    AQSdemo aqs = new AQSdemo() {
        @Override
        public int tryAcquireShared() { // 信号量获取， 数量 - 1
            for(;;) {
                int count =  getState().get();
                int n = count - 1;
                if(count <= 0 || n < 0) {
                    return -1;
                }
                if(getState().compareAndSet(count, n)) {
                    return 1;
                }
            }
        }
```

```
        @Override
        public boolean tryReleaseShared() { // state + 1
            return getState().incrementAndGet() >= 0;
        }
    };

    /** 许可数量 */
    public NeteaseSemaphore(int count) {
        aqs.getState().set(count); // 设置资源的状态
    }

    public void acquire() {
        aqs.acquireShared();
    } // 获取令牌

    public void release() {
        aqs.releaseShared();
    } // 释放令牌
}
```