一、并发同步协作工具

1. 同步器类

读写锁ReentrantReadWriteLock

信号量Semaphore

倒计时门栓CountDownLatch

循环栅栏CyclicBarrier

(1) 基本原理：基于AQS（队列同步器）实现

2. 读写锁ReentrantReadWriteLock

2.1 定义

在Java并发包中，接口ReadWriteLock表示读写锁，主要实现类是可重入读写锁ReentrantReadWriteLock。

public interface ReadWriteLock {

Lock readLock();

Lock writeLock();

}

(1) 通过一个ReadWriteLock产生两个锁，一个读锁，一个写锁。读操作使用读锁，写操作使用写锁。

(2) 需要注意的是，只有"读-读"操作是可以并行的，"读-写"和"写-写"都不可以。只有一个线程可以进行写操作，在获取写锁时，只有在没有任何线程持有任何锁时才可以获取到，在持有写锁时，其他任何线程都获取不到任何锁。在没有其他线程持有写锁的情况下，多个线程可以获取和持有读锁

2.2 构造方法

ReentrantReadWriteLock是可重入的读写锁，它有两个构造方法，如下所示：

public ReentrantLock()

public ReentrantLock(boolean fair)

(1) fire表示是否公平，不传递的话是false。所谓公平是指，等待时间最长的线程优先获得锁。

(2) 可重入，一个线程在持有一个锁的前提下，可以继续获得该锁。

2.3 实例代码

public class MyCache {

private Map<String, Object> map = new HashMap<>();

private ReentrantReadWriteLock readWriteLock = new ReentrantReadWriteLock();

//先新建一个读写锁类，再由此获取读锁和写锁。

private Lock readLock = readWriteLock.readLock();

private Lock writeLock = readWriteLock.writeLock();

public Object get(String key) {

readLock.lock();

try {

return map.get(key);

} finally {

readLock.unlock();

}

}

public Object put(String key, Object value) {

writeLock.lock();

try {

return map.put(key, value);

} finally {

writeLock.unlock();

}

}

public void clear() {

writeLock.lock();

try {

map.clear();

} finally {

writeLock.unlock();

}

}

}

2.4 基本原理

(1) 内部，它们使用同一个整数变量表示锁的状态，16位给读锁用，16位给写锁用，使用一个变量便于进行CAS操作，锁的等待队列其实也只有一个。

(2) 写锁的获取，就是确保当前没有其他线程持有任何锁，否则就等待。写锁释放后，也就是将等待队列中的第一个线程唤醒，唤醒的可能是等待读锁的，也可能是等待写锁的。

(3) 读锁的获取不太一样，首先，只要写锁没有被持有，就可以获取到读锁，此外，在获取到读锁后，它会检查等待队列，逐个唤醒最前面的等待读锁的线程，直到第一个等待写锁的线程。如果有其他线程持有写锁，获取读锁会等待。读锁释放后，检查读锁和写锁数是否都变为了0，如果是，唤醒等待队列中的下一个线程。

3. 信号量Semaphore -- 限制并发访问数

3.1 背景

现实中，资源往往有多个，但每个同时只能被一个线程访问，比如，饭店的饭桌、火车上的卫生间。有的单个资源即使可以被并发访问，但并发访问数多了可能影响性能，所以希望限制并发访问的线程数。还有的情况，与软件的授权和计费有关，对不同等级的账户，限制不同的最大并发访问数。

3.2 信号量类Semaphore构造方法

public Semaphore(int permits)

public Semaphore(int permits, boolean fair)

3.3 基本方法：

Semaphore的方法与锁是类似的，主要的方法有两类，获取许可和释放许可，主要方法有：

//阻塞获取许可

public void acquire() throws InterruptedException

//阻塞获取许可，不响应中断

public void acquireUninterruptibly()

//批量获取多个许可

public void acquire(int permits) throws InterruptedException

public void acquireUninterruptibly(int permits)

//尝试获取

public boolean tryAcquire()

//限定等待时间获取

public boolean tryAcquire(int permits, long timeout, TimeUnit unit) throws

InterruptedException

//释放许可

public void release()

(1) 如果我们将permits的值设为1，你可能会认为它就变成了一般的锁，不过，它与一般的锁是不同的。一般锁只能由持有锁的线程释放，而Semaphore表示的只是一个许可数，任意线程都可以调用其release方法。主要的锁实现类ReentrantLock是可重入的，而Semaphore不是，每一次的acquire调用都会消耗一个许可。

(2) 信号量的基本原理比较简单，也是基于AQS实现的，permits表示共享的锁个数，acquire方法就是检查锁个数是否大于0，大于则减一，获取成功，否则就等待，release就是将锁个数加一，唤醒第一个等待的线程。

4. 倒计时门栓CountDownLatch

4.1 作用：

相当于是一个门栓，一开始是关闭的，所有希望通过该门的线程都需要等待，然后开始倒计时，倒计时变为0后，门栓打开，等待的所有线程都可以通过，它是一次性的，打开后就不能再关上了。

4.2 构造方法（计数变量）

public CountDownLatch(int count)

4.3 主要方法：

public void await() throws InterruptedException

public boolean await(long timeout, TimeUnit unit) throws InterruptedException

public void countDown()

(1) await()检查计数是否为0，如果大于0，就等待，await()可以被中断，也可以设置最长等待时间。countDown检查计数，如果已经为0，直接返回，否则减少计数，如果新的计数变为0，则唤醒所有等待的线程。

4.4 应用场景

(1) 在同时开始场景中，运行员线程等待主裁判线程发出开始指令的信号，一旦发出后，所有运动员线程同时开始，计数初始为1，运动员线程调用await，主线程调用countDown。

(2) 在主从协作模式中，主线程依赖工作线程的结果，需要等待工作线程结束，这时，计数初始值为工作线程的个数，工作线程结束后调用countDown，主线程调用await进行等待。

(3) 需要强调的是，在这里，countDown的调用应该放到finally语句中，确保在工作线程发生异常的情况下也会被调用，使主线程能够从await调用中返回。

5. 循环栅栏CyclicBarrier

5.1 作用

相当于是一个栅栏，所有线程在到达该栅栏后都需要等待其他线程，等所有线程都到达后再一起通过，它是循环的，可以用作重复的同步。CyclicBarrier特别适用于并行迭代计算，每个线程负责一部分计算，然后在栅栏处等待其他线程完成，所有线程到齐后，交换数据和计算结果，再进行下一次迭代。

5.2 构造方法：

//表示的是参与的线程个数

public CyclicBarrier(int parties)

//这个参数表示栅栏动作，当所有线程到达栅栏后，在所有线程执行下一步动作前，运行参数中的动作，这个动作由最后一个到达栅栏的线程执行。

public CyclicBarrier(int parties, Runnable barrierAction)

5.3 主要方法

public int await() throws InterruptedException, BrokenBarrierException

public int await(long timeout, TimeUnit unit) throws InterruptedException,

BrokenBarrierException, TimeoutException

(1) await在等待其他线程到达栅栏，调用await后，表示自己已经到达，如果自己是最后一个到达的，就执行可选的命令，执行后，唤醒所有等待的线程，然后重置内部的同步计数，以循环使用。

(2) await可以被中断，可以限定最长等待时间，中断或超时后会抛出异常。需要说明的是异常BrokenBarrierException，它表示栅栏被破坏了，什么意思呢？在CyclicBarrier中，参与的线程是互相影响的，只要其中一个线程在调用await时被中断了，或者超时了，栅栏就会被破坏，此外，如果栅栏动作抛出了异常，栅栏也会被破坏，被破坏后，所有在调用await的线程就会退出，抛出BrokenBarrierException。

5.4 CyclicBarrier与CountDownLatch的区别

(1) CountDownLatch的参与线程是有不同角色的，有的负责倒计时，有的在等待倒计时变为0，负责倒计时和等待倒计时的线程都可以有多个，它用于不同角色线程间的同步。

(2) CyclicBarrier的参与线程角色是一样的，用于同一角色线程间的协调一致。

(3) CountDownLatch是一次性的，而CyclicBarrier是可以重复利用的。

6. 总结

(1) 在读多写少的场景中使用ReentrantReadWriteLock替代ReentrantLock，以提高性能

(2) 使用Semaphore限制对资源的并发访问数

(3) 使用CountDownLatch实现不同角色线程间的同步

(4) 使用CyclicBarrier实现同一角色线程间的协调一致