并发包中锁的工具类LockSupport

简述

java.util.concurrent.locks.LockSupport 是 JDK 中的 rt.jar包里面的 工具类，它的主要作用是挂起和唤醒线程，该工具类是创建锁和其他同步

主要提供了阻塞，唤醒线程

方法名称	描述
void park()	阻塞当前线程，如果掉用unpark(Thread)方法或被中断(interrupt），才能从park()返回
void parkNanos(long nanos)	阻塞当前线程，超时返回，阻塞时间最长不超过nanos纳秒
void parkUntil(long deadline)	阻塞当前线程，直到deadline时间点
void unpark(Thread)	唤醒处于阻塞状态的线程

LockSupport 对比 synchronized

```
public class TestLock1 {

    public static void main(String[] args) throws Exception {
            final Object obj = new Object();
            Thread A = new Thread(new Runnable() {
                    @Override
                    public void run() {
                            int sum = 0;
                            for (int i = 0; i < 10; i++) {
                                    sum += i;
                            }
                            try {
                                    synchronized (obj) {  // 01
                                            obj.wait();
                                    }
                            } catch (Exception e) {
                                    e.printStackTrace();
                            }
                            System.out.println(sum);
                    }
            });
            A.start();
            // 睡眠一秒钟，保证线程A已经计算完成，阻塞在wait方法
            Thread.sleep(1000);
            synchronized (obj) { // 02
                    obj.notify();
            }
    }
}
```

wait和notify/notifyAll方法只能在同步代码块里用， 否则会抛出异常 （注意 01 和 02 两处的）

```
Exception in thread "main" java.lang.IllegalMonitorStateException
```

那么使用LockSupport来代替 synchronized

```
public class TestLock1 {

    public static void main(String[] args) throws Exception {
            Thread A = new Thread(new Runnable() {
                    @Override
                    public void run() {
                            int sum = 0;
                            for (int i = 0; i < 10; i++) {
                                    sum += i;
                            }
                            try {
                                    LockSupport.park();
                            } catch (Exception e) {
                                    e.printStackTrace();
                            }
                            System.out.println(sum);
                    }
```

```
            });
            A.start();
            // 睡眠一秒钟，保证线程A已经计算完成，阻塞在wait方法
            Thread.sleep(1000);
            LockSupport.unpark(A);
    }
}
```

改变使用同步块的代码

去掉Thread.sleep(1000); 进行测试，会发现 最后没办法结束了，主线程先执行了 obj.notify()；然后 A线程才执行 obj.wait();

改变使用LockSupport的代码

同样去掉Thread.sleep(1000); 进行测试，会发现没什么变化， 因为 unpark 一定在 park之后执行

小结

- LockSupport不需要在同步代码块里 。所以线程间也不需要维护一个共享的同步对象了，实现了线程间的解耦。
- unpark函数可以先于park调用，所以不需要担心线程间的执行的先后顺序

抽象同步队列 AQS —— AbstractQueuedSynchronizer

AbstractQueuedSynchronizer，它使整个java.util.concurrent包中众多并发工具类的灵魂

在并发包（JUC）中锁的底层就是基于AQS实现的，例如CountDownLatch、ReentrantLock、ReentrantReadWriteLock 等

代码结构简述

字段说明

```
private transient volatile Node head;

private transient volatile Node tail;

private volatile int state;
```

可以看到AQS是一个双向队列，通过head 和 tail记录 头元素和尾元素。队列的元素类型为Node。

Node讲解

```
// 模式Model，分为共享与独占
// 用来标记该线程是获取共享资源时被阻塞挂起后放入 AQS 队列的，还是获取独占资源时被挂起后放入 AQS 队列的
// 共享模式
static final Node SHARED = new Node();
// 独占模式
static final Node EXCLUSIVE = null;
// 结点状态
// CANCELLED，值为1，表示当前的线程被取消
// SIGNAL，值为-1，表示当前节点的后继节点包含的线程需要运行，也就是unpark
// CONDITION，值为-2，表示当前节点在等待condition，也就是在condition队列中
// PROPAGATE，值为-3，表示当前场景下后续的acquireShared能够得以执行
// 值为0，表示当前节点在sync队列中，等待着获取锁
static final int CANCELLED =  1;
```

```
static final int SIGNAL    = -1;
static final int CONDITION = -2;
static final int PROPAGATE = -3;

// 结点状态
volatile int waitStatus;
// 前驱结点
volatile Node prev;
// 后继结点
volatile Node next;
// 结点所对应的线程
volatile Thread thread;
// 下一个等待者
Node nextWaiter;
```

state队列内部的状态信息字段

ConditionObject说明

AQS 有个内部类 ConditionObject， 用来结合锁实现线程同步 。 ConditionObject 可以直接访问 AQS对象内部的变量，比如 state状

ConditionObject主要是为并发编程中的同步提供了等待通知的实现方式，可以在不满足某个条件的时候挂起线程等待。直到满足某个条
Condition 接口

```
public interface Condition {

    // 等待，当前线程在接到信号或被中断之前一直处于等待状态
    void await() throws InterruptedException;

    // 等待，当前线程在接到信号之前一直处于等待状态，不响应中断
    void awaitUninterruptibly();

    //等待，当前线程在接到信号、被中断或到达指定等待时间之前一直处于等待状态
     long awaitNanos(long nanosTimeout) throws InterruptedException;

     // 等待，当前线程在接到信号、被中断或到达指定等待时间之前一直处于等待状态。此方法在行为上等效于：awaitNanos(un
     boolean await(long time, TimeUnit unit) throws InterruptedException;

     // 等待，当前线程在接到信号、被中断或到达指定最后期限之前一直处于等待状态
     boolean awaitUntil(Date deadline) throws InterruptedException;

     // 唤醒一个等待线程。如果所有的线程都在等待此条件，则选择其中的一个唤醒。在从 await 返回之前，该线程必须重新获取
     void signal();

     // 唤醒所有等待线程。如果所有的线程都在等待此条件，则唤醒所有线程。在从 await 返回之前，每个线程都必须重新获取锁
     void signalAll();
}
```

总结

AQS是一个继续每个线程状态的双向队列，通过ConditionObject来操作队列， 用来结合锁实现线程同步。

可能永远都不会用到，但是很多用到的并发工具都是基于他实现的，例如：CountDownLatch、ReentrantLock、ReentrantReadWriteI

ReentrantLock

简述

jdk中独占锁的实现除了使用关键字synchronized外,还可以使用ReentrantLock。虽然在性能上ReentrantLock和synchronized没有什么区别而言功能更加丰富，使用起来更为灵活，也更适合复杂的并发场景

与synchronized的比较

1. ReentrantLock是独占锁且可重入的
2. ReentrantLock可以实现公平锁和非公平锁，而synchronized只能实现非公平锁

```
static class T1 implements Runnable {

    static ReentrantLock lock = new ReentrantLock(true);
    private int count = 1;

    @Override
    public void run() {
        for (int j = 0; j < 5; j++) {
            lock.lock();
             System.out.println(Thread.currentThread().getName() + " : " + count);
             count++;
             lock.unlock();
         }
     }
}

public static void main(String[] args) {
     for (int i = 0; i < 5; i++) {
         new Thread(new T1()).start();
     }
}

代码说明：公平锁，输出会很均匀，每个线程输出一次。
Thread-0 : 1
Thread-1 : 1
Thread-2 : 1
Thread-3 : 1
Thread-4 : 1
Thread-0 : 2
Thread-1 : 2
Thread-2 : 2
Thread-3 : 2
Thread-4 : 2
Thread-0 : 3
Thread-1 : 3
Thread-2 : 3
Thread-3 : 3
Thread-4 : 3
Thread-0 : 4
Thread-1 : 4
```

```
Thread-2 : 4
Thread-3 : 4
Thread-4 : 4
Thread-0 : 5
Thread-1 : 5
Thread-2 : 5
Thread-3 : 5
Thread-4 : 5
```

结合Condition实现等待通知机制

使用synchronized结合Object上的wait和notify方法可以实现线程间的等待通知机制。ReentrantLock结合Condition接口同样可以实现这
晰也更简单

使用说明

Condition接口在使用前必须先调用ReentrantLock的lock()方法获得锁。之后调用Condition接口的await()将释放锁,并且在该Condition上
signal()方法唤醒线程。使用方式和wait,notify类似

简单示例

```java
public class ConditionTest {

    static ReentrantLock lock = new ReentrantLock();
    static Condition condition = lock.newCondition();
    public static void main(String[] args) throws InterruptedException {

        lock.lock();
        new Thread(new SignalThread()).start();
        System.out.println("主线程等待通知");
        try {
            condition.await();
        } finally {
            lock.unlock();
        }
        System.out.println("主线程恢复运行");
    }
    static class SignalThread implements Runnable {

        @Override
        public void run() {
            lock.lock();
            try {
                TimeUnit.SECONDS.sleep(5);
                condition.signal();
                System.out.println("子线程通知");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }
    }
}
```

使用多个Condition实现交替打印A和B

```
static class Print {

    ReentrantLock lock = new ReentrantLock(true);

    Condition conditionA = lock.newCondition();

    Condition conditionB = lock.newCondition();

    public void printA() throws InterruptedException {
        while (true) {
            lock.lock();
            System.out.println("A");
            Thread.sleep(5000);
            conditionB.signal();
            conditionA.await();
            lock.unlock();
        }
    }

    public void printB() throws InterruptedException {
        while (true) {
            lock.lock();
            System.out.println("B");
            Thread.sleep(3000);
            conditionA.signal();
            conditionB.await();
            lock.unlock();
        }
    }
}

static class TA implements Runnable{
    private Print print;
    public TA(Print print){
        this.print = print;
    }

    @Override
    public void run() {
        try {
            print.printA();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

static class TB implements Runnable{
    private Print print;
    public TB(Print print){
        this.print = print;
```

```
    }

    @Override
    public void run() {
        try {
            print.printB();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}


public static void main(String[] args) {
    Print p = new Print();
    new Thread(new TA(p)).start();
    new Thread(new TB(p)).start();
}
```

使用Condition实现简单的阻塞队列

阻塞队列是一种特殊的先进先出队列,它有以下几个特点

1.入队和出队线程安全

2.当队列满时,入队线程会被阻塞;当队列为空时,出队线程会被阻塞

```
package threadlocal;

import java.util.LinkedList;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class MyBlockingQueue<E> {

    int size;//阻塞队列最大容量

     ReentrantLock lock = new ReentrantLock();

     LinkedList<E> list=new LinkedList<>();//队列底层实现

     Condition notFull = lock.newCondition();//队列满时的等待条件
     Condition notEmpty = lock.newCondition();//队列空时的等待条件

     public MyBlockingQueue(int size) {
         this.size = size;
     }

     public void enqueue(E e) throws InterruptedException {
         lock.lock();
         try {
             while (list.size() ==size)//队列已满,在notFull条件上等待
             {
                 notFull.await();
             }
             list.add(e);//入队:加入链表末尾
```

```
            System.out.println("入队: " +e);
            notEmpty.signal(); //通知在notEmpty条件上等待的线程
        } finally {
            lock.unlock();
        }
    }

    public E dequeue() throws InterruptedException {
        E e;
        lock.lock();
        try {
            while (list.size() == 0)//队列为空,在notEmpty条件上等待
            {
                notEmpty.await();
            }
            e = list.removeFirst();//出队:移除链表首元素
            System.out.println("出队: "+e);
            notFull.signal();//通知在notFull条件上等待的线程
            return e;
        } finally {
            lock.unlock();
        }
    }



    public static void main(String[] args) throws InterruptedException {

        MyBlockingQueue<Integer> queue = new MyBlockingQueue<>(2);
        for (int i = 0; i < 10; i++) {
            int data = i;
            new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        queue.enqueue(data);
                    } catch (InterruptedException e) {

                    }
                }
            }).start();

        }
        for(int i=0;i<10;i++){
            new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        Integer data = queue.dequeue();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }).start();
```

```
        }

    }
}
```

ReentrantReadWriteLock

读写锁拆成读锁和写锁来理解。读锁可以共享，多个线程可以同时拥有读锁，但是写锁却只能只有一个线程拥有，而且获取写锁的时候其他取写锁之后，其他线程不能再获取读锁。简单的说就是写锁是排他锁，读锁是共享锁

```
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class ReadWriteLockTest {
   private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

   public void read() {
      try {
         lock.readLock().lock();
         System.out.println(Thread.currentThread().getName() + "  开始读取");
          Thread.sleep(2000);
          System.out.println(Thread.currentThread().getName() + "  读取完毕");
      } catch (InterruptedException e) {
          e.printStackTrace();
      } finally {
         lock.readLock().unlock();
      }
   }

   public void write() {
      try {
         lock.writeLock().lock();
         System.out.println(Thread.currentThread().getName() + "  开始写数据");
         Thread.sleep(1000);
         System.out.println(Thread.currentThread().getName() + "  写数据完毕");
      } catch (InterruptedException e) {
         e.printStackTrace();
      } finally {
         lock.writeLock().unlock();
      }
   }

   public static void main(String[] args) {
      final ReadWriteLockTest rwlt = new ReadWriteLockTest();
      for(int i = 0 ;i < 10 ; i ++){
         new Thread() {
            @Override
            public void run() {
               rwlt.read();
            }
         }.start();
         new Thread() {
            @Override
```

```
            public void run() {
                rwlt.write();
            }
        }.start();
    }
  }
}
```

StampedLock

简述

StampedLock类，在JDK1.8时引入，是对读写锁ReentrantReadWriteLock的增强，该类提供了一些功能，优化了读锁、写锁的转换，更细粒度控制并发

为什么有了ReentrantReadWriteLock，还要引入StampedLock?

```
ReentrantReadWriteLock使得多个读线程同时持有读锁（只要写锁未被占用），而写锁是独占的。
但是，读写锁如果使用不当，很容易产生“饥饿”问题：
比如在读线程非常多，写线程很少的情况下，很容易导致写线程“饥饿”，虽然使用“公平”策略可以一定程度上缓解这个问题，
但是“公平”策略是以牺牲系统吞吐量为代价的
```

特点

StampedLock的主要特点概括一下，有以下几点：

1. 所有获取锁的方法，都返回一个邮戳（Stamp），Stamp为0表示获取失败，其余都表示成功；
2. 所有释放锁的方法，都需要一个邮戳（Stamp），这个Stamp必须是和成功获取锁时得到的Stamp一致；
3. StampedLock是不可重入的；（如果一个线程已经持有了写锁，再去获取写锁的话就会造成死锁）
4. StampedLock有三种访问模式：
   ①Reading（读模式）：功能和ReentrantReadWriteLock的读锁类似
   ②Writing（写模式）：功能和ReentrantReadWriteLock的写锁类似
   ③Optimistic reading（乐观读模式）：这是一种优化的读模式。读的时候发现有写操作，再去读多一次

5. StampedLock支持读锁和写锁的相互转换

   我们知道RRW中，当线程获取到写锁后，可以降级为读锁，但是读锁是不能直接升级为写锁的。StampedLock提供了读锁和写锁相互转换的功能，使得该类支持更多的应用场景。

6. 无论写锁还是读锁，都不支持Conditon等待