带你一次搞明白 Java 多线程（Ⅶ）

5 Lock 显示锁

在 JDK5 中增加了 Lock 锁接口，有 ReentrantLock 实现类,ReentrantLock 锁称为可重入锁, 它功能比 synchronized 多.

5.2 ReentrantLock

5.2.4 tryLock()方法

tryLock(long time, TimeUnit unit) 的作用在给定等待时长内锁没有被另外的线程持有,并且当前线程也没有被中断,则获得该锁.通过该方法可以实现锁对象的限时等待.

```
package com.wkcto.lock.reentrant;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;

/**
 *tryLock(long time, TimeUnit unit) 的基本使用
 */
public class Test07 {
    static class TimeLock implements Runnable{
        private static ReentrantLock lock = new ReentrantLock();     //定义锁对象


        @Override
        public void run() {
            try {
                if ( lock.tryLock(3, TimeUnit.SECONDS) ){          //获得锁返回 true
                    System.out.println(Thread.currentThread().getName() + "获得锁,执行耗时任务");
//                  Thread.sleep(4000);          //假设 Thread-0 线程先持有锁,完成任务需要 4 秒钟,Thread-1 线程尝试获得锁,Thread-1 线程在 3 秒内还没有获得锁的话,Thread-1 线程会放弃
                    Thread.sleep(2000);          //假设 Thread-0 线程先持有锁,完成任务需要 2 秒钟,Thread-1 线程尝试获得锁,Thread-1 线程会一直尝试,在它约定尝试的 3 秒内可以获得锁对象
                }else {          //没有获得锁
                    System.out.println(Thread.currentThread().getName() + "没有获得锁");
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                if (lock.isHeldByCurrentThread()){
                    lock.unlock();
                }
            }
        }
    }

    public static void main(String[] args) {
        TimeLock timeLock = new TimeLock();

        Thread t1 = new Thread(timeLock);
        Thread t2 = new Thread(timeLock);
        t1.start();
        t2.start();
    }
}
```

tryLock()仅在调用时锁定未被其他线程持有的锁,如果调用方法时,锁对象对其他线程持有,则放弃. 调用方法尝试获得没,如果该锁没有被其他线程占用则返回 true 表示锁定成功; 如果锁被其他线程占用则返回 false,不等待.

```
package com.wkcto.lock.reentrant;

import java.util.concurrent.locks.ReentrantLock;

/**
 *tryLock()
 *   当锁对象没有被其他线程持有的情况下才会获得该锁定
 */
public class Test08 {
    static class Service{
        private ReentrantLock lock = new ReentrantLock();
        public void serviceMethod(){
            try {
                if (lock.tryLock()){
                    System.out.println(Thread.currentThread().getName() + "获得锁定");
                    Thread.sleep(3000);      //模拟执行任务的时长
                }else {
                    System.out.println(Thread.currentThread().getName() + "没有获得锁定
");
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                if (lock.isHeldByCurrentThread()){
                    lock.unlock();
                }
            }
```

```
		}
	}

	public static void main(String[] args) throws InterruptedException {
		Service service = new Service();
		Runnable r = new Runnable() {
			@Override
			public void run() {
				service.serviceMethod();
			}
		};

		Thread t1 = new Thread(r);
		t1.start();
		Thread.sleep(50);		//睡眠 50 毫秒,确保 t1 线程锁定
		Thread t2 = new Thread(r);
		t2.start();
	}
}
```

```
package com.wkcto.lock.reentrant;

import java.util.Random;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 使用 tryLock()可以避免死锁
 */
public class Test09 {
	static class	IntLock implements Runnable{
		private static ReentrantLock lock1 = new ReentrantLock();
		private static ReentrantLock lock2 = new ReentrantLock();
		private int lockNum;		//用于控制锁的顺序


		public IntLock(int lockNum) {
			this.lockNum = lockNum;
		}

		@Override
```

```
        public void run() {
            if ( lockNum % 2 == 0 ){        //偶数先锁 1,再锁 2
                while (true){
                    try {
                        if (lock1.tryLock()){
                            System.out.println(Thread.currentThread().getName() + "获得
锁 1，还想获得锁 2");
                            Thread.sleep(new Random().nextInt(100));

                            try {
                                if (lock2.tryLock()){

System.out.println(Thread.currentThread().getName() + "同时获得锁 1 与锁 2 ----完成任务了");
                                    return;        //结束 run()方法执行,即当前线程
结束
                                }
                            } finally {
                                if (lock2.isHeldByCurrentThread()){
                                    lock2.unlock();
                                }
                            }
                        }
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    } finally {
                        if (lock1.isHeldByCurrentThread()){
                            lock1.unlock();
                        }
                    }
                }
            }else {        //奇数就先锁 2,再锁 1
                while (true){
                    try {
                        if (lock2.tryLock()){
```

```
                                        System.out.println(Thread.currentThread().getName() + "获得锁 2，还想获得锁 1");
                                        Thread.sleep(new Random().nextInt(100));

                                        try {
                                            if (lock1.tryLock()){
                                                System.out.println(Thread.currentThread().getName() + "同时获得锁 1 与锁 2 ----完成任务了");
                                                return;         //结束 run()方法执行,即当前线程结束
                                            }
                                        } finally {
                                            if (lock1.isHeldByCurrentThread()){
                                                lock1.unlock();
                                            }
                                        }
                                    }
                                } catch (InterruptedException e) {
                                    e.printStackTrace();
                                } finally {
                                    if (lock2.isHeldByCurrentThread()){
                                        lock2.unlock();
                                    }
                                }
                            }
                        }
                    }
                }
    public static void main(String[] args) {
        IntLock intLock1 = new IntLock(11);
        IntLock intLock2 = new IntLock(22);
        Thread t1 = new Thread(intLock1);
        Thread t2 = new Thread(intLock2);
        t1.start();
        t2.start();

        //运行后,使用 tryLock()尝试获得锁,不会傻傻的等待,通过循环不停的再次尝试,如果
```

```
等待的时间足够长,线程总是会获得想要的资源
        }
}
```

5.2.5 newCondition()方法

关键字 synchronized 与 wait()/notify()这两个方法一起使用可以实现等待/通知模式. Lock 锁的 newContition()方法返回 Condition 对象,Condition 类也可以实现等待/通知模式.

使用 notify()通知时, JVM 会随机唤醒某个等待的线程. 使用 Condition 类可以进行选择性通知. Condition 比较常用的两个方法:

await()会使当前线程等待,同时会释放锁,当其他线程调用 signal()时,线程会重新获得锁并继续执行.

signal()用于唤醒一个等待的线程

注意:在调用 Condition 的 await()/signal()方法前,也需要线程持有相关的 Lock 锁. 调用 await()后线程会释放这个锁,在 singal()调用后会从当前 Condition 对象的等待队列中,唤醒 一个线程,唤醒 的线程尝试获得锁, 一旦获得锁成功就继续执行.

```
package com.wkcto.lock.condition;

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

/**
 * Condition 等待与通知
 */
public class Test01 {

        //定义锁

        static Lock lock = new ReentrantLock();

        //获得 Condtion 对象

        static Condition condition = lock.newCondition();


        //定义线程子类

        static class SubThread extends Thread{
                @Override
                public void run() {
                        try {

                                lock.lock();        //在调用 await()前必须先获得锁

                                System.out.println("method lock");

                                condition.await();            //等待

                                System.out.println("method    await");
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        } finally {

                                lock.unlock();           //释放锁

                                System.out.println("method unlock");
                        }
                }
        }

        public static void main(String[] args) throws InterruptedException {
                SubThread t = new SubThread();
                t.start();

                //子线程启动后,会转入等待状态


                Thread.sleep(3000);

                //主线程在睡眠 3 秒后,唤醒子线程的等待

                try {
                        lock.lock();
                condition.signal();
            } finally {
                lock.unlock();
            }
        }
}
```

```
package com.wkcto.lock.condition;

import java.io.PipedOutputStream;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 多个 Condition 实现通知部分线程，使用更灵活
 */
public class Test02 {
    static class Service{
        private ReentrantLock lock = new ReentrantLock();        //定义锁对象

        //定义两个 Condtion 对象
        private Condition conditionA = lock.newCondition();
        private Condition conditionB = lock.newCondition();


        //定义方法,使用 conditionA 等待
        public void waitMethodA(){
            try {
                lock.lock();
                System.out.println(Thread.currentThread().getName() + " begin wait:" +
System.currentTimeMillis());
                conditionA.await();         //等待
                System.out.println(Thread.currentThread().getName() + " end wait:" +
System.currentTimeMillis());
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }

        //定义方法,使用 conditionB 等待
        public void waitMethodB(){
                try {
                        lock.lock();
                        System.out.println(Thread.currentThread().getName() + " begin wait:" +
System.currentTimeMillis());
                        conditionB.await();             //等待
                        System.out.println(Thread.currentThread().getName() + " end wait:" +
System.currentTimeMillis());
                } catch (InterruptedException e) {
                        e.printStackTrace();
                } finally {
                        lock.unlock();
                }
        }

        //定义方法唤醒 conditionA 对象上的等待
        public void signalA(){
                try {
                        lock.lock();
                        System.out.println(Thread.currentThread().getName() + " sigal A time = " +
System.currentTimeMillis());
                        conditionA.signal();
                        System.out.println(Thread.currentThread().getName() + " sigal A time = " +
System.currentTimeMillis());
                } finally {
                        lock.unlock();
                }
        }

        //定义方法唤醒 conditionB 对象上的等待
        public void signalB(){
                try {
                        lock.lock();
                        System.out.println(Thread.currentThread().getName() + " sigal A time = " +
System.currentTimeMillis());
```

```
                                conditionB.signal();
                                System.out.println(Thread.currentThread().getName() + " sigal A time = " +
System.currentTimeMillis());
                        } finally {
                                lock.unlock();
                        }
                }

        }

        public static void main(String[] args) throws InterruptedException {
                Service service = new Service();

                //开启两个线程,分别调用 waitMethodA(),waitMethodB()方法
                new Thread(new Runnable() {
                        @Override
                        public void run() {
                                service.waitMethodA();
                        }
                }).start();
                new Thread(new Runnable() {
                        @Override
                        public void run() {
                                service.waitMethodB();
                        }
                }).start();

                Thread.sleep(3000);              //main 线程睡眠 3 秒

//              service.signalA();               //唤醒  conditionA 对象上的等待,conditionB 上的等待
依然继续等待
                service.signalB();
        }
}
```

```
package com.wkcto.lock.condition;

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

/**
 *   使用 Condition 实现生产者/消费者设计模式，两个 线程交替打印
 */
public class Test03 {
    static class    MyService{

        private Lock lock = new ReentrantLock();            //创建锁对象

        private Condition condition = lock.newCondition();   //创建 Condition 对象

        private boolean flag = true;            //定义交替打印标志


        //定义方法只打印----横线

        public void printOne(){
            try {

                lock.lock();            //锁定

                while (flag){         //当 flag 为 true 等待

                    System.out.println(Thread.currentThread().getName() + " waiting...");
                    condition.await();
                }

                //flag 为 false 时打印

                System.out.println(Thread.currentThread().getName() + " ---------------- ");

                flag = true;            //修改交替打印标志

                condition.signal();        //通知另外的线程打印

            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {

                lock.unlock();          //释放锁对象

            }
        }
        //定义方法只打印***横线
        public void printTwo(){
            try {
                lock.lock();            //锁定
                while (!flag){          //当 flag 为 false 等待
                    System.out.println(Thread.currentThread().getName() + " waiting...");
                    condition.await();
                }
                //flag 为 true 时打印
                System.out.println(Thread.currentThread().getName() + " ****** ");
                flag = false;           //修改交替打印标志
                condition.signal();     //通知另外的线程打印
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();          //释放锁对象
            }
        }
    }

    public static void main(String[] args) {
        MyService myService = new MyService();
        //创建线程打印--
        new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    myService.printOne();
                }
            }
        }).start();
        //创建线程打印**
```

```
                new Thread(new Runnable() {
                        @Override
                        public void run() {
                                for (int i = 0; i < 100; i++) {
                                        myService.printTwo();
                                }
                        }
                }).start();
        }
}
```

5.2.6 公平锁与非公平锁

大多数情况下,锁的申请都是非公平的. 如果线程 1 与线程 2 都在请求锁 A, 当锁 A 可用时, 系统只是会从阻塞队列中随机的选择一个线程, 不能保证其公平性.

公平的锁会按照时间先后顺序,保证先到先得, 公平锁的这一特点不会出现线程饥饿现象.

synchronized 内部锁就是非公平的. ReentrantLock 重入锁提供了一个构造方法:ReentrantLock(boolean fair) ,当在创建锁对象时实参传递 true 可以把该锁设置为公平锁. 公平锁看起来很公平,但是要实现公平锁必须要求系统维护一个有序队列,公平锁的实现成本较高,性能也低. 因此默认情况下锁是非公平的. 不是特别的需求,一般不使用公平锁.

```
package com.wkcto.lock.method;

import java.util.concurrent.locks.ReentrantLock;

/**
```

```
 * 公平 锁与非公平锁
 */
public class Test01 {
//    static ReentrantLock lock = new ReentrantLock();          //默认是非公平锁
    static ReentrantLock lock = new ReentrantLock(true);          //定义公平锁

    public static void main(String[] args) {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                while (true){
                    try {
                        lock.lock();
                        System.out.println(Thread.currentThread().getName() + " 获得了
锁对象");
                    }finally {
                        lock.unlock();
                    }
                }
            }
        };

        for (int i = 0; i < 5; i++) {
            new Thread(runnable).start();
        }
        /*
            运行程序
                1)如果是非公平锁，系统倾向于让一个线程再次获得已经持有的锁，这种
分配策略是高效的,非公平的
                2)如果是公平锁，多个线程不会发生同一个线程连续多次获得锁的可能,
保证了公平性
         */
	}
}
```

5.2.7 几个常用的方法

int getHoldCount()　返回当前线程调用 lock()方法的次数

int getQueueLength()　返回正等待获得锁的线程预估数

int getWaitQueueLength(Condition condition)　返回与 Condition 条件相关的等待的线程预估数

boolean hasQueuedThread(Thread thread)　查询参数指定的线程是否在等待获得锁

boolean hasQueuedThreads()　查询是否还有线程在等待获得该锁

boolean hasWaiters(Condition condition)　查询是否有线程正在等待指定的 Condition 条件

boolean isFair()　判断是否为公平锁

boolean isHeldByCurrentThread()　判断当前线程是否持有该锁

boolean isLocked()　查询当前锁是否被线程持有；