带你一次搞明白 Java 多线程（VIII）

5.2 ReentrantLock

5.2.1 ReentrantLock 的基本使用

调用 lock()方法获得锁，调用 unlock()释放锁

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

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

/**
 * Lock 锁的基本使用
 */
public class Test02 {
    //定义显示锁
    static Lock lock = new ReentrantLock();
    //定义方法
    public static void sm(){
        //先获得锁
        lock.lock();
        //for 循环就是同步代码块
        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName() + " -- " + i);
        }
        //释放锁
        lock.unlock();
    }

    public static void main(String[] args) {
        Runnable r = new Runnable() {
```

```
                @Override
                public void run() {
                    sm();
                }
            };
            //启动三个线程
            new Thread(r).start();
            new Thread(r).start();
            new Thread(r).start();
        }
}
```

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

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

/**
 * 使用 Lock 锁同步不同方法中的同步代码块
 */
public class Test03 {

    static Lock lock = new ReentrantLock();              //定义锁对象

    public static void sm1(){

        //经常在 try 代码块中获得 Lock 锁，在 finally 子句中释放锁

        try {

            lock.lock();            //获得锁

            System.out.println(Thread.currentThread().getName() + "-- method 1 -- " + System.currentTimeMillis() );
            Thread.sleep(new Random().nextInt(1000));
            System.out.println(Thread.currentThread().getName() + "-- method 1 -- " + System.currentTimeMillis() );
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {

            lock.unlock();              //释放锁
```

```
        }
    }

    public static void sm2(){
        try {
            lock.lock();            //获得锁
            System.out.println(Thread.currentThread().getName() + "-- method 22 -- " +
System.currentTimeMillis() );
            Thread.sleep(new Random().nextInt(1000));
            System.out.println(Thread.currentThread().getName() + "-- method 22 -- " +
System.currentTimeMillis() );
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();              //释放锁
        }
    }

    public static void main(String[] args) {
        Runnable r1 = new Runnable() {
            @Override
            public void run() {
                sm1();
            }
        };
        Runnable r2 = new Runnable() {
            @Override
            public void run() {
                sm2();
            }
        };

        new Thread(r1).start();
        new Thread(r1).start();
        new Thread(r1).start();
        new Thread(r2).start();
        new Thread(r2).start();
        new Thread(r2).start();
    }
}
```

5.2.2 ReentrantLock 锁的可重入性

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

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

/**
 *  ReentrantLock 锁的可重入性
 */
public class Test04 {
    static class Subthread extends Thread{
        private static Lock lock = new ReentrantLock();          //定义锁对象

        public static int num = 0;          //定义变量

        @Override
        public void run() {
            for (int i = 0; i <10000 ; i++) {
                try {
                    //可重入锁指可以反复获得该锁
                    lock.lock();
                    lock.lock();
                    num++;
                }finally {
                    lock.unlock();
                    lock.unlock();
                }
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Subthread t1 = new Subthread();
        Subthread t2 = new Subthread();
```

```
            t1.start();
            t2.start();
            t1.join();
            t2.join();
            System.out.println( Subthread.num );
        }
}
```

5.2.3 lockInterruptibly()方法

lockInterruptibly() 方法的作用:如果当前线程未被中断则获得锁,如果当前线程被中断则出现异常.

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

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

/**
 * lockInterruptibly() 方法的作用:如果当前线程未被中断则获得锁,如果当前线程被中断则出现异常.
 */
public class Test05 {
    static class Servier{

        private Lock lock = new ReentrantLock();            //定义锁对象
        public void serviceMethod(){
            try {
//                  lock.lock();            //获得锁定,即使调用了线程的 interrupt()方法,也没有真正的中断线程

                lock.lockInterruptibly();    //如果线程被中断了,不会获得锁,会产生异常
                System.out.println(Thread.currentThread().getName() + "-- begin lock");
```

```
                //执行一段耗时的操作
                for (int i = 0; i < Integer.MAX_VALUE; i++) {
                    new StringBuilder();
                }
                System.out.println( Thread.currentThread().getName() + " -- end lock");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                System.out.println( Thread.currentThread().getName() + " *****  释放锁");

                lock.unlock();          //释放锁
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Servier    s = new Servier();
        Runnable r = new Runnable() {
            @Override
            public void run() {
                s.serviceMethod();
            }
        };
        Thread t1 = new Thread(r);
        t1.start();

        Thread.sleep(50);
        Thread t2 = new Thread(r);
        t2.start();
        Thread.sleep(50);
        t2.interrupt();        //中断 t2 线程
    }
}
```

对于 synchronized 内部锁来说,如果一个线程在等待锁,只有两个结

果:要么该线程获得锁继续执行;要么就保持等待.

对于 ReentrantLock 可重入锁来说,提供另外一种可能,在等待锁的过程中,程序可以根据需要取消对锁的请求.

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

import com.wkcto.pipestream.Test2;

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

/**
 *  通过 ReentrantLock 锁的 lockInterruptibly()方法避免死锁的产生
 */
public class Test06 {
    static  class  IntLock implements Runnable{

        //创建两个 ReentrantLock  锁对象

        public static ReentrantLock lock1 = new ReentrantLock();
        public static ReentrantLock lock2 = new ReentrantLock();

        int lockNum;            //定义整数变量,决定使用哪个锁


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

        @Override
        public void run() {
            try {
                if ( lockNum % 2 == 1){            //奇数,先锁 1,再锁 2
                    lock1.lockInterruptibly();
                    System.out.println(Thread.currentThread().getName() + "获得锁 1,还需
要获得锁 2");
                    Thread.sleep(new Random().nextInt(500));
```

```
                    lock2.lockInterruptibly();
                    System.out.println(Thread.currentThread().getName() + "同时获得了锁
1 与锁 2....");
                }else {        //偶数,先锁 2,再锁 1
                    lock2.lockInterruptibly();
                    System.out.println(Thread.currentThread().getName() + "获得锁 2,还需
要获得锁 1");
                    Thread.sleep(new Random().nextInt(500));
                    lock1.lockInterruptibly();
                    System.out.println(Thread.currentThread().getName() + "同时获得了锁
1 与锁 2....");
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                if ( lock1.isHeldByCurrentThread())          //判断当前线程是否持有该锁
                    lock1.unlock();
                if (lock2.isHeldByCurrentThread())
                    lock2.unlock();
                System.out.println( Thread.currentThread().getName() + "线程退出");
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        IntLock intLock1 = new IntLock(11);
        IntLock intLock2    = new IntLock(22);

        Thread t1 = new Thread(intLock1);
        Thread t2 = new Thread(intLock2);
        t1.start();
        t2.start();
```

```
            //在 main 线程,等待 3000 秒,如果还有线程没有结束就中断该线程
            Thread.sleep(3000);

            //可以中断任何一个线程来解决死锁, t2 线程会放弃对锁 1 的申请,同时释放锁 2,
t1 线程会完成它的任务
            if (t2.isAlive()){ t2.interrupt();}
        }
}
```

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

5.3 ReentrantReadWriteLock 读写锁

synchronized 内部锁与 ReentrantLock 锁都是独占锁(排它锁)，同一时间只允许一个线程执行同步代码块,可以保证线程的安全性,但是执行效率低.

ReentrantReadWriteLock 读写锁是一种改进的排他锁,也可以称作共享/排他锁. 允许多个线程同时读取共享数据,但是一次只允许一个线程对共享数据进行更新.

读写锁通过读锁与写锁来完成读写操作. 线程在读取共享数据前必须先持有读锁,该读锁可以同时被多个线程持有,即它是共享的.线程在修改共享数据前必须先持有写锁,写锁是排他的，一个线程持有写锁时其他线程无法获得相应的锁

读锁只是在读线程之间共享,任何一个线程持有读锁时,其他线程都无法获得写锁，保证线程在读取数据期间没有其他线程对数据进行更新,使得读线程能够读到数据的最新值,保证在读数据期间共享变量不被修改

	获得条件	排他性	作用
读锁	写锁未被任意线程持有	对读线程是共享的，对写线程是排他的	允许多个读线程可以同时读取共享数据,保证在读共享数据时,没有其他线程对共享数据进行修改

写锁	该写锁未被其他线程持有,并且相应的读锁也未被其他线程持有	对读线程或者写线程都是排他的	保证写线程以独占的方式修改共享数据

读写锁允许读读共享，读写互斥,写写互斥

在 java.util.concurrent.locks 包中定义了 ReadWriteLock 接口,该接口中定义了 readLock()返回读锁,定义 writeLock()方法返回写锁. 该接口的实现类是 ReentrantReadWriteLock.

注意 readLock()与 writeLock()方法返回的锁对象是同一个锁的两个不同的角色，不是分别获得两个不同的锁. ReadWriteLock 接口实例可以充当两个角色.读写锁的其他使用方法

```
//定义读写锁
ReadWriteLock    rwLock = new ReentrantReadWriteLock();
//获得读锁
Lock readLock = rwLock.readLock();
//获得写锁
Lock writeLock = rwLock.writeLock();
//读数据
readLock.lock();        //申请读锁
try{
    读取共享数据
}finally{
```

```
        readLock.unlock();    //总是在 finally 子句中释放锁
    }
    //写数据
    writeLock.lock();        //申请写锁
    try{
        更新修改共享数据
    }finally{
        writeLock.unlock();    //总是在 finally 子句中释放锁
    }
```

5.3.1 读读共享

ReadWriteLock 读写锁可以实现多个线程同时读取共享数据,即读读共享,可以提高程序的读取数据的效率

```
package com.wkcto.lock.readwrite;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

/**
 * ReadWriteLock 读写锁可以实现读读共享,允许多个线程同时获得读锁
 */
public class Test01 {
    static class Service{
        //定义读写锁
        ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
```

```
        //定义方法读取数据
        public void read(){
            try {
                readWriteLock.readLock().lock();            //获得读锁
                System.out.println(Thread.currentThread().getName() + "获得读锁,开始读取
数据的时间--" + System.currentTimeMillis());
                TimeUnit.SECONDS.sleep(3);   //模拟读取数据用时
            } catch (InterruptedException e) {
                e.printStackTrace();
            }finally {
                readWriteLock.readLock().unlock();   //释放读锁
            }
        }
    }

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

        //创建 5 个线程,调用 read()方法
        for (int i = 0; i < 5; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    service.read();        //在线程中调用 read()读取数据
                }
            }).start();
        }

        //运行程序后,这多个 线程几乎可以同时获得锁读,执行 lock()后面的代码
    }
}
```

5.3.2 写写互斥

通过 ReadWriteLock 读写锁中的写锁,只允许有一个线程执行 lock()后面的代码.

```
package com.wkcto.lock.readwrite;

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

/**
 * 演示 ReadWriteLock 的 writeLock()写锁是互斥的,只允许有一个线程持有
 */
public class Test02 {
    static class Service{
        //先定义读写锁
        ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
        //定义方法修改数据
        public void write(){
            try {
                readWriteLock.writeLock().lock();          //申请获得写锁

                System.out.println(Thread.currentThread().getName() + "获得写锁,开始修改
数据的时间--" + System.currentTimeMillis());

                Thread.sleep(3000);       //模拟修改数据的用时
            } catch (InterruptedException e) {
                e.printStackTrace();
            }finally {
                System.out.println(Thread.currentThread().getName() + "读取数据完毕时的
时间==" + System.currentTimeMillis());

                readWriteLock.writeLock().unlock();       //释放写锁
```

```
                }
            }
        }

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

            //创建 5 个线程修改数据
            for (int i = 0; i < 5; i++) {
                new Thread(new Runnable() {
                    @Override
                    public void run() {

                        service.write();       //调用修改数据的方法

                    }
                }).start();
            }
            //从执行结果来看,同一时间只有一个线程获得写锁
        }
}
```

5.3.3 读写互斥

写锁是独占锁,是排他锁,读线程与写线程也是互斥的

```
package com.wkcto.lock.readwrite;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

/**
 * 演示 ReadWriteLock 的读写互斥

 *  一个线程获得读锁时,写线程等待;  一个线程获得写锁时,其他线程等待
```

```
 */
public class Test03 {
    static class Service{
        //先定义读写锁
        ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
        Lock readLock = readWriteLock.readLock();          //获得读锁
        Lock writeLock = readWriteLock.writeLock();        //获得写锁
        //定义方法读取数据
        public void read(){
            try {
                readLock.lock();          //申请获得读锁
                System.out.println(Thread.currentThread().getName() + "获得读锁,开始读取
数据的时间--" + System.currentTimeMillis());
                Thread.sleep(3000);       //模拟读取数据的用时
            } catch (InterruptedException e) {
                e.printStackTrace();
            }finally {
                System.out.println(Thread.currentThread().getName() + "读取数据完毕时的
时间==" + System.currentTimeMillis());
                readLock.unlock();        //释放读锁
            }
        }
        //定义方法修改数据
        public void write(){
            try {
                writeLock.lock();         //申请获得写锁
                System.out.println(Thread.currentThread().getName() + "获得写锁,开始修改
```

```
数据的时间--" + System.currentTimeMillis());

                    Thread.sleep(3000);        //模拟修改数据的用时
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }finally {
                    System.out.println(Thread.currentThread().getName() + "修改数据完毕时的
时间==" + System.currentTimeMillis());

                    writeLock.unlock();        //释放写锁
                }
            }
        }

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

            //定义一个线程读数据
            new Thread(new Runnable() {
                @Override
                public void run() {
                    service.read();
                }
            }).start();

            //定义一个线程写数据
            new Thread(new Runnable() {
                @Override
                public void run() {
                    service.write();
                }
            }).start();
        }
}
```

6 线程管理

6.1 线程组

类似于在计算机中使用文件夹管理文件,也可以使用线程组来管理线程. 在线程组中定义一组相似(相关)的线程,在线程组中也可以定义子线程组

Thread 类有几个构造方法允许在创建线程时指定线程组,如果在创建线程时没有指定线程组则该线程就属于父线程所在的线程组. JVM 在创建 main 线程时会为它指定一个线程组,因此每个 Java 线程都有一个线程组与之关联, 可以调用线程的 getThreadGroup()方法返回线程组.

线程组开始是出于安全的考虑设计用来 区分不同的 Applet,然而 ThreadGroup 并未实现这一目标,在新开发的系统中,已经不常用线程组, 现在一般会将一组相关的线程存入一个数组或一个集合中,如果仅仅是用来区分线程时,可以使用线程名称来区分, 多数情况下,可以忽略线程组.

6.1.1 创建线程组

```
package com.wkcto.threadgroup;
```

```
/**
 * 演示创建线程组
 */
public class Test01 {
    public static void main(String[] args) {
        // 1) 返回当前 main 线程的线程组
        ThreadGroup mainGroup = Thread.currentThread().getThreadGroup();
        System.out.println(mainGroup);

        //2) 定义线程组,如果不指定所属线程组,则自动归属当前线程所属的线程组中
        ThreadGroup group1 = new ThreadGroup("group1");
        System.out.println(group1);

        //3)定义线程组, 同时指定父线程组
        ThreadGroup group2 = new ThreadGroup(mainGroup, "group2");
        //现在 group1 与 group2 都是 maingroup 线程组中的子线程组, 调用线程组的 getParent()方法返回父线程组
        System.out.println( group1.getParent() == mainGroup);
```

```
//true
        System.out.println( group2.getParent() == mainGroup);


        //4) 在创建线程时指定所属线程组
        Runnable r = new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread());
            }
        };
        //在创建线程时,如果没有指定线程组,则默认线程归属到父线程的线程组中
        //在 main 线程中创建了 t1 线程,称 main 线程为父线程,t1 线程为子线程, t1 没有指定线程组则 t1 线程就归属到父线程 main 线程的线程组中
        Thread t1 = new Thread(r, "t1");
        System.out.println( t1 );     //Thread[t1,5,main], t1 的线程组是 main 线程组
        //创建线程时,可以指定线程所属线程组
        Thread t2 = new Thread(group1, r, "t2");
        Thread t3 = new Thread(group2, r, "t3");
```

```
        System.out.println(t2);
        System.out.println(t3);
    }
}
```

6.1.2 线程组的基本操作

activeCount() 返回当前线程组及子线程组中活动线程的数量(近似值)

activeGroupCount() 返回当前线程组及子线程组中活动线程组的数量(近似值)

int enumerate(Thread[] list) 将当前线程组中的活动线程复制到参数数组中

enumerate(ThreadGroup[] list) 将当前线程组中的活动线程组复制到参数数组中

getMaxPriority() 返回线程组的最大优先级,默认是 10

getName() 返回线程组的名称

getParent() 返回父线程组

interrupt() 中断线程组中所有的线程

isDaemon() 判断当前线程组是否为守护线程组

list() 将当前线程组中的活动线程打印出来

parentOf(ThreadGroup g) 判断当前线程组是否为参数线程组的父线程组

setDaemon(boolean daemon) 设置线程组为守护线程组

```
package com.wkcto.threadgroup;

/**
 * 演示线程组的基本操作
 */
public class Test02 {
    public static void main(String[] args) {
        ThreadGroup mainGroup = Thread.currentThread().getThreadGroup();       //返回当前线程组
        //再定义线程组
        ThreadGroup group = new ThreadGroup("group");    //默认 group 的父线程组是 main 线程组

        Runnable r = new Runnable() {
            @Override
            public void run() {
                while (true){
                    System.out.println("----------当前线程: " + Thread.currentThread());
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        };

        Thread t1 = new Thread(r, "t1");            //默认在 main 线程组中创建线程
```

```
            Thread t2 = new Thread(group, r, "t2");        //在指定的 group 线程组中创建线程
            t1.start();
            t2.start();

            //打印线程组的相关属性
            System.out.println("main 线程组中活动线程数量: " + mainGroup.activeCount());
//4, main 线程组中活动线程: main, t1, t2, 垃圾回收器
            System.out.println("group 子线程组中活动线程数量: " + group.activeCount());
//1,  t2
            System.out.println("main 线程组中子线程组数量: " + mainGroup.activeGroupCount());  //1,  group
            System.out.println("group 子线程组中子线程组数量: " + group.activeGroupCount());
//0
            System.out.println("main 线程组的父线程组: " + mainGroup.getParent());     //main 线程组的父线程组是 system
            System.out.println("group 线程组的父线程组: " + group.getParent());  //main
            System.out.println( mainGroup.parentOf(mainGroup));  //true, 线程组也是它自己的父线程组
            System.out.println( mainGroup.parentOf(group));        //true
            mainGroup.list();     //把 main 线程组中所有的线程打印输出
    }
}
```

6.1.3 复制线程组中的线程及子线程组

enumerate(Thread[] list) 把当前线程组和子线程组中所有的线程复制到参数数组中

enumerate(Thread[] list, boolean recursive)，如果第二个参数设置为 false,则只复制当前线程组中所有的线程,不复制子线程组中的线程

enumerate(ThreadGroup[] list) 把当前线程组和子线程组中所有的线程组复制到参数数组中

enumerate(ThreadGroup[] list, boolean recurse) 第二个参数设置 false,则只复制当前线程组的子线程组

```
package com.wkcto.threadgroup;

/**
 * 演示复制线程组中的内容
 */
public class Test03 {
    public static void main(String[] args) {
        ThreadGroup mainGroup = Thread.currentThread().getThreadGroup();       //返回 main 线程的 main 线程组

        //main 线程组中定义了两个子线程组
        ThreadGroup group1 = new ThreadGroup("group1");         //默认 group1 的父线程组就是当前线程组 main
        ThreadGroup group2 = new ThreadGroup(mainGroup, "group2");

        Runnable r = new Runnable() {
            @Override
```

```
            public void run() {
                while (true){
                    System.out.println("----当前线程: " + Thread.currentThread());
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        };

        //创建并启动三个线程
        Thread t1 = new Thread(r, "t1");       //默认在 main 线程组中创建线程
        Thread t2 = new Thread(group1, r, "t2");       //在 group1 线程组中创建线程
        Thread t3 = new Thread(group2, r, "t3");       //在 group2 线程组中创建线程
        t1.start();
        t2.start();
        t3.start();

        //1) 把 main 线程组中的线程复制到数组中
        //先定义存储线程的数组,数组的长度为 main 线程组中活动线程的数量
        Thread[] threadList = new Thread[mainGroup.activeCount()];
/*          //把 main 线程组包括子线程组中的所有的线程复制到数组中
        mainGroup.enumerate(threadList);
        //遍历 threadList 数组
        for (Thread thread : threadList) {
            System.out.println(thread);
        }
        System.out.println("----------------------------");*/

        //只把 main 线程组中的线程复制到数组中,不包含子线程组的线程
```

```
            mainGroup.enumerate(threadList, false);

            //遍历 threadList 数组

            for (Thread thread : threadList) {
                System.out.println(thread);
            }
            System.out.println("----------------------------");


            //2) 把 main 线程组中的子线程组复制到数组中

            //定义数组存储线程组

            ThreadGroup [] threadGroups = new ThreadGroup[mainGroup.activeGroupCount()];

            //把 main 线程组中的子线程组复制到数组中

            mainGroup.enumerate(threadGroups);
            System.out.println("============================");
            for (ThreadGroup threadGroup : threadGroups) {
                System.out.println(threadGroup);
            }
        }
}
```

6.1.4 线程组的批量中断

线程组的 interrupt() 可以给该线程组中所有的活动线程添加中断标志

```
package com.wkcto.threadgroup;

/**

 * 线程组的批量中断

 */
public class Test04 {
    public static void main(String[] args) throws InterruptedException {
        Runnable r = new Runnable() {
```

```
                @Override
                public void run() {

                    System.out.println("当前线程--" + Thread.currentThread() + "--开始循环");

                    //当线程没有被中断就一直循环

                    while ( !Thread.currentThread().isInterrupted()){
                        System.out.println(Thread.currentThread().getName()                    +
"------------------");
                      /*    try {
                            Thread.sleep(500);
                        } catch (InterruptedException e) {

                            //如果中断睡眠中的线程,产生中断异常，同时会清除中断标志

                            e.printStackTrace();
                        }*/
                    }

                    System.out.println(Thread.currentThread().getName() + "循环结束");

                }
            };


            //创建线程组

            ThreadGroup group = new ThreadGroup("group");


            //在 group 线程组中创建 5 个线程

            for (int i = 0; i < 5; i++) {
                new Thread(group,r).start();
            }


            //main 线程睡眠 2 秒

            Thread.sleep(50);

            //中断线程组，会中断线程组中所有的线程

            group.interrupt();
        }
}
```

6.1.5 设置守护线程组

守护线程是为其他线程提供服务的,当 JVM 中只有守护线程时,守护线程会自动销毁,JVM 会退出.

调用线程组的 setDaemon(true)可以把线程组设置为守护线程组,当守护线程组中没有任何活动线程时,守护线程组会自动销毁.

注意线程组的守护属性,不影响线程组中线程的守护属性,或者说守护线程组中的线程可以是非守护线程

```
package com.wkcto.threadgroup;

/**
 * 演示设置守护线程组
 */
public class Test05 {
    public static void main(String[] args) throws InterruptedException {
        //先定义线程组
        ThreadGroup group = new ThreadGroup("group");

        //设置线程组为守护线程组
        group.setDaemon(true);

        //向组中添加 3 个线程
        for (int i = 0; i < 3; i++) {
            new Thread(group, new Runnable() {
                @Override
                public void run() {
                    for (int j = 0; j < 20; j++) {
                        System.out.println(Thread.currentThread().getName() + " -- " + j);
                        try {
                            Thread.sleep(500);
                        } catch (InterruptedException e) {
```

```
                                e.printStackTrace();
                            }
                        }
                    }
                }).start();
            }

            //main 线程睡眠 5 秒
            Thread.sleep(5000);
            System.out.println("main...end....");
        }
}
```