

Java多线程之ReentrantLock与Condition

一、ReentrantLock

1、ReentrantLock简介

ReentrantLock是一个可重入的互斥锁，又被称为“独占锁”。ReentrantLock 类实现了 Lock，它拥有与 synchronized 相同的并发性和内存语义，但是添加了类似锁投票、定时锁等候和可中断锁等候的一些特性。此外，它还提供了在激烈争用情况下更佳的性能。（换句话说，当许多线程都想访问共享资源时，JVM 可以花更少的时间来调度线程，把更多时间用在执行线程上。）

顾名思义，ReentrantLock锁在同一个时间点只能被一个线程锁持有；而可重入的意思是，ReentrantLock锁，可以被单个线程多次获取。ReentrantLock分为“公平锁”和“非公平锁”。它们的区别体现在获取锁的机制上是否公平。“锁”是为了保护竞争资源，防止多个线程同时操作线程而出错，ReentrantLock在同一个时间点只能被一个线程获取(当某线程获取到“锁”时，其它线程就必须等待)；ReentraantLock是通过一个FIFO的等待队列来管理获取该锁所有线程的。在“公平锁”的机制下，线程依次排队获取锁；而“非公平锁”在锁是可获取状态时，不管自己是不是在队列的开头都会获取锁。

2、ReentrantLock函数列表



```
// 创建一个 ReentrantLock，默认是“非公平锁”。
ReentrantLock()

// 创建策略是fair的 ReentrantLock。fair为true表示是公平锁，fair为false表示是非公平锁。
ReentrantLock(boolean fair)

// 查询当前线程保持此锁的次数。
int getHoldCount()
// 返回目前拥有此锁的线程，如果此锁不被任何线程拥有，则返回 null。
protected Thread getOwner()
// 返回一个 collection，它包含可能正等待获取此锁的线程。
protected Collection<Thread> getQueuedThreads()
// 返回正等待获取此锁的线程估计数。
int getQueueLength()
// 返回一个 collection，它包含可能正在等待与此锁相关给定条件的那些线程。
protected Collection<Thread> getWaitingThreads(Condition condition)
// 返回等待与此锁相关的给定条件的线程估计数。
int getWaitQueueLength(Condition condition)
// 查询给定线程是否正在等待获取此锁。
boolean hasQueuedThread(Thread thread)
// 查询是否有些线程正在等待获取此锁。
boolean hasQueuedThreads()
// 查询是否有些线程正在等待与此锁有关的给定条件。
boolean hasWaiters(Condition condition)
// 如果是“公平锁”返回true，否则返回false。
```

```

boolean isFair()
// 查询当前线程是否保持此锁。
boolean isHeldByCurrentThread()
// 查询此锁是否由任意线程保持。
boolean isLocked()
// 获取锁。
void lock()
// 如果当前线程未被中断，则获取锁。
void lockInterruptibly()
// 返回用来与此 Lock 实例一起使用的 Condition 实例。
Condition newCondition()
// 仅在调用时锁未被另一个线程保持的情况下，才获取该锁。
boolean tryLock()
// 如果锁在给定等待时间内没有被另一个线程保持，且当前线程未被中断，则获取该锁。
boolean tryLock(long timeout, TimeUnit unit)
// 试图释放此锁。
void unlock()

```



可重入锁指在同一个线程中，可以重入的锁。当然，当这个线程获得锁后，其他线程将等待这个锁被释放后，才可以获得这个锁。

通常的使用方法：



```

ReentrantLock lock = new ReentrantLock(); // not a fair lock
lock.lock();

try {

    // synchronized do something

} finally {
    lock.unlock();
}

```



3、重入的实现

reentrant 锁意味着什么呢？简单来说，它有一个与锁相关的获取计数器，如果拥有锁的某个线程再次得到锁，那么获取计数器就加1，然后锁需要被释放两次才能获得真正释放。这模仿了 synchronized 的语义；如果线程进入由线程已经拥有的监控器保护的 synchronized 块，就允许线程继续进行，当线程退出第二个（或者后续）synchronized 块的时候，不释放锁，只有线程退出它进入的监控器保护的第一个 synchronized 块时，才释放锁。

对于锁的重入，我们来想这样一个场景。当一个递归方法被synchronized关键字修饰时，在调用方法时显然没有发生问题，执行线程获取了锁之后仍能连续多次地获得该锁，也就是说synchronized关键字支持锁的重入。对于ReentrantLock，虽然没有像synchronized那样隐式地支持重入，但在调用lock()方法时，已经获取到锁的线程，能够再次调用lock()方法获取锁而不会被阻塞。

如果想要实现锁的重入，至少要解决一下两个问题：

- **线程再次获取锁**：锁需要去识别获取锁的线程是否为当前占据锁的线程，如果是，则再次成功获取。
- **锁的最终释放**：线程重复n次获取了锁，随后在n次释放该锁后，其他线程能够获取该锁。锁的最终释放要求锁对于获取进行计数自增，计数表示当前锁被重复获取的次数，而锁被释放时，计数自减，当计数等于0时表示锁已

经释放。

4、公平锁与非公平锁

在Java的ReentrantLock构造函数中提供了两种锁：创建公平锁和非公平锁（默认）。代码如下：

```

/**
 * 默认构造方法，非公平锁
 */
public ReentrantLock() {
    sync = new NonfairSync();
}

/**
 * true公平锁, false非公平锁
 * @param fair
 */
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}

```

如果获取一个锁是按照请求的顺序得到的，那么就是公平锁，否则就是非公平锁。

在没有深入了解内部机制及实现之前，先了解下为什么会存在公平锁和非公平锁。公平锁保证一个阻塞的线程最终能够获得锁，因为是有序的，所以总是可以按照请求的顺序获得锁。非公平锁意味着后请求锁的线程可能在其前面排列的休眠线程恢复前拿到锁，这样就有可能提高并发的性能。这是因为通常情况下挂起的线程重新开始与它真正开始运行，二者之间会产生严重的延时。因此非公平锁就可以利用这段时间完成操作。这是非公平锁在某些时候比公平锁性能要好的原因之一。

锁Lock分为“公平锁”和“非公平锁”，公平锁表示线程获取锁的顺序是按照线程加锁的顺序来分配的，即先来先得的FIFO先进先出顺序。而非公平锁就是一种获取锁的抢占机制，是随机获得锁的，和公平锁不一样的就是先来的不一定先得到锁，这个方式可能造成某些线程一直拿不到锁，结果也就是不公平的了。

5、ReentrantLock 扩展的功能

(1) 实现可轮询的锁请求

在内部锁中，死锁是致命的——唯一的恢复方法是重新启动程序，唯一的预防方法是在构建程序时不要出错。而可轮询的锁获取模式具有更完善的错误恢复机制，可以规避死锁的发生。

如果你不能获得所有需要的锁，那么使用可轮询的获取方式使你能够重新拿到控制权，它会释放你已经获得的这些锁，然后再重新尝试。可轮询的锁获取模式，由tryLock()方法实现。此方法仅在调用时锁为空闲状态才获取该锁。如果锁可用，则获取锁，并立即返回值true。如果锁不可用，则此方法将立即返回值false。

(2) 实现可定时的锁请求

当使用内部锁时，一旦开始请求，锁就不能停止了，所以内部锁给实现具有时限的活动带来了风险。为了解决这一问题，可以使用定时锁。当具有时限的活动调用了阻塞方法，定时锁能够在时间预算内设定相应的超时。如果活动在期待的时间内没能获得结果，定时锁能使程序提前返回。可定时的锁获取模式，由tryLock(long, TimeUnit)方法实现。

(3) 实现可中断的锁获取请求

可中断的锁获取操作允许在可取消的活动中使用。lockInterruptibly()方法能够使你获得锁的时候响应中断。

6、ReentrantLock 与 synchronized 的比较

相同：ReentrantLock提供了synchronized类似的功能和内存语义。

不同：

- (1) 与synchronized相比，ReentrantLock提供了更多，更加全面的功能，具备更强的扩展性。例如：时间锁等候，可中断锁等候，锁投票。
- (2) ReentrantLock还提供了条件Condition，对线程的等待、唤醒操作更加详细和灵活，所以在多个条件变量和高度竞争锁的地方，ReentrantLock更加适合（下面会阐述Condition）。
- (3) ReentrantLock提供了可轮询的锁请求。它会尝试着去获取锁，如果成功则继续，否则可以等到下次运行时处理，而synchronized则一旦进入锁请求要么成功，要么一直阻塞，所以相比synchronized而言，ReentrantLock会不容易产生死锁些。
- (4) ReentrantLock支持更加灵活的同步代码块，但是使用synchronized时，只能在同一个synchronized块结构中获取和释放。**注：ReentrantLock的锁释放一定要在finally中处理，否则可能会产生严重的后果。**
- (5) ReentrantLock支持中断处理，且性能较synchronized会好些。

7、ReentrantLock 不好与需要注意的地方

- (1) **lock 必须在 finally 块中释放。**否则，如果受保护的代码将抛出异常，锁就有可能永远得不到释放！这一点区别看起来可能没什么，但是实际上，它极为重要。忘记在 finally 块中释放锁，可能会在程序中留下一个定时炸弹，当有一天炸弹爆炸时，您要花费很大力气才有找到源头在哪。而使用同步，JVM 将确保锁会获得自动释放。
- (2) 当 JVM 用 synchronized 管理锁定请求和释放时，JVM 在生成线程转储时能够包括锁定信息。这些对调试非常有价值，因为它们能标识死锁或者其他异常行为的来源。Lock 类只是普通的类，JVM 不知道具体哪个线程拥有 Lock 对象。

8、示例分析

示例1：

首先来看第一个实例：用两个线程在控制台有序打出1，2，3。

```
package com.demo.test;

public class FirstReentrantLock {

    public static void main(String[] args) {
        Runnable runnable = new ReentrantLockThread();
        new Thread(runnable, "a").start();
        new Thread(runnable, "b").start();
    }
}
```

```
package com.demo.test;

public class ReentrantLockThread implements Runnable{

    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
```

```

        System.out.println(Thread.currentThread().getName() + "输出了:  " + i);
    }
}
}

```



执行FirstReentrantLock，查看控制台输出：



```

a输出了:  0
b输出了:  0
a输出了:  1
b输出了:  1
a输出了:  2
b输出了:  2

```



可以看到，并没有顺序，杂乱无章。

那使用ReentrantLock加入锁，代码如下：



```

package com.demo.test;

/**
 * 如何使用ReentrantLock
 * @author lxx
 */
public class FirstReentrantLock {

    public static void main(String[] args) {
        Runnable runnable = new ReentrantLockThread();
        new Thread(runnable, "a").start();
        new Thread(runnable, "b").start();
    }
}

```



```

package com.demo.test;

import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockThread implements Runnable{

    // 创建一个ReentrantLock对象
    ReentrantLock lock = new ReentrantLock();

    @Override
    public void run() {
        try{
            // 使用lock()方法加锁

```

```

        lock.lock();
        for (int i = 0; i < 3; i++) {
            System.out.println(Thread.currentThread().getName() + "输出了：  " + i);
        }
    }finally{
        // 别忘了执行unlock()方法释放锁
        lock.unlock();
    }
}
}
}

```

执行FirstReentrantLock，查看控制台输出：

```

a输出了： 0
a输出了： 1
a输出了： 2
b输出了： 0
b输出了： 1
b输出了： 2

```

有顺序的打印出了0, 1, 2。这就是锁的作用，它是互斥的，当一个线程持有锁的时候，其他线程只能等待，待该线程执行结束，再通过竞争得到锁。

示例2：测试可重入锁的重入特性。

```

package com.demo.test;

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

public class TestLock {

    private ReentrantLock lock = null;

    public TestLock() {
        // 创建一个自由竞争的可重入锁
        lock = new ReentrantLock();
    }

    public static void main(String[] args) {

        TestLock tester = new TestLock();

        try{
            // 测试可重入，方法testReentry() 在同一线程中，可重复获取锁，执行获取锁后，显示信息的功能
            tester.testReentry();
            // 能执行到这里而不阻塞，表示锁可重入
            tester.testReentry();
            // 再次重入
            tester.testReentry();
        }
    }
}

```

```

        }catch(Exception e){
            e.printStackTrace();
        }finally{
            // 释放重入测试的锁, 要按重入的数量解锁, 否则其他线程无法获取该锁。
            tester.getLock().unlock();
            tester.getLock().unlock();
            tester.getLock().unlock();
        }
    }

    public ReentrantLock getLock() {
        return lock;
    }

    public void testReentry() {
        lock.lock();

        Calendar now = Calendar.getInstance();

        System.out.println(now.getTime() + " " + Thread.currentThread().getName()
            + " get lock.");
    }
}

```



运行结果：

```

Thu Oct 12 22:01:47 CST 2017 main get lock.
Thu Oct 12 22:01:47 CST 2017 main get lock.
Thu Oct 12 22:01:47 CST 2017 main get lock.

```

示例3：此例可反应公平锁和非公平锁的差异。

(1) 公平锁



```

package com.demo.test;

import java.util.concurrent.locks.ReentrantLock;

public class Service {

    private ReentrantLock lock ;

    public Service(boolean isFair) {
        lock = new ReentrantLock(isFair);
    }

    public void serviceMethod() {
        try {
            lock.lock();
            System.out.println("ThreadName=" + Thread.currentThread().getName()
                + " 获得锁定");
        } finally {
            lock.unlock();
        }
    }
}

```

```

    }
}

```



```
package com.demo.test;
```

```
public class Run {
```

```

    public static void main(String[] args) throws InterruptedException {
        final Service service = new Service(true); //改为false就为非公平锁了
        Runnable runnable = new Runnable() {
            public void run() {
                System.out.println("**线程: " + Thread.currentThread().getName()
                    + " 运行了 " );
                service.serviceMethod();
            }
        };
    };

```

```
    Thread[] threadArray = new Thread[10];
```

```

    for (int i=0; i<10; i++) {
        threadArray[i] = new Thread(runnable);
    }
    for (int i=0; i<10; i++) {
        threadArray[i].start();
    }

```

```
}
```



运行结果：



```
**线程： Thread-0 运行了
```

```
**线程： Thread-2 运行了
```

```
ThreadName=Thread-0 获得锁定
```

```
**线程： Thread-4 运行了
```

```
**线程： Thread-3 运行了
```

```
**线程： Thread-6 运行了
```

```
ThreadName=Thread-2 获得锁定
```

```
**线程： Thread-8 运行了
```

```
ThreadName=Thread-4 获得锁定
```

```
**线程： Thread-7 运行了
```

```
ThreadName=Thread-3 获得锁定
```

```
**线程： Thread-1 运行了
```

```
ThreadName=Thread-6 获得锁定
```

```
ThreadName=Thread-8 获得锁定
```

```
**线程： Thread-9 运行了
```

```
**线程： Thread-5 运行了
```

```
ThreadName=Thread-7 获得锁定
```

```
ThreadName=Thread-1 获得锁定
```

```
ThreadName=Thread-9 获得锁定
```

```
ThreadName=Thread-5
```


获得锁定



打印的结果是按照线程加锁的顺序输出的，即线程运行了，则会先获得锁。

(2) 非公平锁

将下面语句中的参数true改为false就为非公平锁了。

```
final Service service = new Service(true); //改为false就为非公平锁了
```

运行结果：



```
**线程： Thread-1 运行了
**线程： Thread-2 运行了
ThreadName=Thread-2 获得锁定
ThreadName=Thread-1 获得锁定
**线程： Thread-6 运行了
ThreadName=Thread-6 获得锁定
**线程： Thread-7 运行了
ThreadName=Thread-7 获得锁定
**线程： Thread-0 运行了
ThreadName=Thread-0 获得锁定
**线程： Thread-4 运行了
**线程： Thread-9 运行了
**线程： Thread-5 运行了
**线程： Thread-3 运行了
ThreadName=Thread-4 获得锁定
**线程： Thread-8 运行了
ThreadName=Thread-8 获得锁定
ThreadName=Thread-9 获得锁定
ThreadName=Thread-5 获得锁定
ThreadName=Thread-3 获得锁定
```



是乱序的，说明先start()启动的线程不代表先获得锁。

运行结果反映：

在公平的锁上，线程按照他们发出请求的顺序获取锁，但在非公平锁上，则允许“插队”：当一个线程请求非公平锁时，如果在发出请求的同时该锁变成可用状态，那么这个线程会跳过队列中所有的等待线程而获得锁。非公平的ReentrantLock 并不提倡 插队行为，但是无法防止某个线程在合适的时候进行插队。

在公平的锁中，如果有另一个线程持有锁或者有其他线程在等待队列中等待这个锁，那么新发出的请求的线程将被放入到队列中。而非公平锁上，只有当锁被某个线程持有时，新发出请求的线程才会被放入队列中。

非公平锁性能高于公平锁性能的原因：在恢复一个被挂起的线程与该线程真正运行之间存在着严重的延迟。假设线程A持有一个锁，并且线程B请求这个锁。由于锁被A持有，因此B将被挂起。当A释放锁时，B将被唤醒，因此B会再次尝试获取这个锁。与此同时，如果线程C也请求这个锁，那么C很可能在B被完全唤醒之前获得、使用以及释放这个锁。这样就是一种双赢的局面：B获得锁的时刻并没有推迟，C更早的获得了锁，并且吞吐量也提高了。

当持有锁的时间相对较长或者请求锁的平均时间间隔较长，应该使用公平锁。在这些情况下，插队带来的吞吐量提升（当锁处于可用状态时，线程却还处于被唤醒的过程中）可能不会出现。

二、Condition

Condition是在java 1.5中才出现的，它用来替代传统的Object的wait()、notify()实现线程间的协作，相比使用Object的wait()、notify()，使用Condition的await()、signal()这种方式实现线程间协作更加安全和高效。因此通常来说比较推荐使用Condition。

Condition类能实现synchronized和wait、notify搭配的功能，另外比后者更灵活，Condition可以实现多路通知功能，也就是在一个Lock对象里可以创建多个Condition（即对象监视器）实例，线程对象可以注册在指定的Condition中，从而可以有选择的进行线程通知，在调度线程上更加灵活。而synchronized就相当于整个Lock对象中只有一个单一的Condition对象，所有的线程都注册在这个对象上。线程开始notifyAll时，需要通知所有的WAITING线程，没有选择权，会有相当大的效率问题。

1、Condition是个接口，基本的方法就是await()和signal()方法。

2、Condition依赖于Lock接口，生成一个Condition的基本代码是lock.newCondition()，参考下图。

```
// 实例化一个ReentrantLock对象
private ReentrantLock lock = new ReentrantLock();
// 为线程A注册一个Condition
public Condition conditionA = lock.newCondition();
// 为线程B注册一个Condition
public Condition conditionB = lock.newCondition();
```

3、调用Condition的await()和signal()方法，都必须在lock保护之内，就是说必须在lock.lock()和lock.unlock之间才可以使用。

4、Condition中的await()对应Object的wait()，Condition中的signal()对应Object的notify()，Condition中的signalAll()对应Object的notifyAll()。

接下来，使用Condition来实现等待/唤醒，并且能够唤醒制定线程。

先写业务代码：

```
package com.demo.test;

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

public class MyService {

    // 实例化一个ReentrantLock对象
    private ReentrantLock lock = new ReentrantLock();
    // 为线程A注册一个Condition
    public Condition conditionA = lock.newCondition();
    // 为线程B注册一个Condition
    public Condition conditionB = lock.newCondition();

    public void awaitA() {
        try {
            lock.lock();
            System.out.println(Thread.currentThread().getName() + "进入了awaitA方法");
            long timeBefore = System.currentTimeMillis();
            // 执行conditionA等待
            conditionA.await();
            long timeAfter = System.currentTimeMillis();
            System.out.println(Thread.currentThread().getName() + "被唤醒");
            System.out.println(Thread.currentThread().getName() + "等待了: " + (timeAfter - timeBefore)/1000+"s");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```

    } finally {
        lock.unlock();
    }
}

public void awaitB() {
    try {
        lock.lock();
        System.out.println(Thread.currentThread().getName() + "进入了awaitB方法");
        long timeBefore = System.currentTimeMillis();
        // 执行conditionB等待
        conditionB.await();
        long timeAfter = System.currentTimeMillis();
        System.out.println(Thread.currentThread().getName()+"被唤醒");
        System.out.println(Thread.currentThread().getName() + "等待了: " + (timeAfter -
timeBefore)/1000+"s");
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}

public void signalA() {
    try {
        lock.lock();
        System.out.println("启动唤醒程序");
        // 唤醒所有注册conditionA的线程
        conditionA.signalAll();
    } finally {
        lock.unlock();
    }
}

public void signalB() {
    try {
        lock.lock();
        System.out.println("启动唤醒程序");
        // 唤醒所有注册conditionB的线程
        conditionB.signalAll();
    } finally {
        lock.unlock();
    }
}
}

```

分别实例化了两个Condition对象，都是使用同一个lock注册。注意conditionA对象的等待和唤醒只对使用了conditionA的线程有用，同理conditionB对象的等待和唤醒只对使用了conditionB的线程有用。

继续写两个线程的代码：

```

package com.demo.test;

public class MyServiceThread1 implements Runnable{

```

```

private MyService service;

public MyServiceThread1(MyService service) {
    this.service = service;
}

@Override
public void run() {
    service.awaitA();
}
}

```



注意：MyServiceThread1 使用了awaitA()方法，持有的是conditionA！



```

package com.demo.test;

public class MyServiceThread2 implements Runnable{

    private MyService service;

    public MyServiceThread2(MyService service) {
        this.service = service;
    }

    @Override
    public void run() {
        service.awaitB();
    }
}

```



注意：MyServiceThread2 使用了awaitB()方法，持有的是conditionB！

最后看启动类：



```

package com.demo.test;

public class ApplicationCondition {

    public static void main(String[] args) throws InterruptedException {
        MyService service = new MyService();
        Runnable runnable1 = new MyServiceThread1(service);
        Runnable runnable2 = new MyServiceThread2(service);

        new Thread(runnable1, "a").start();
        new Thread(runnable2, "b").start();

        // 线程sleep2秒钟
        Thread.sleep(2000);
        // 唤醒所有持有conditionA的线程
        service.signalA();
    }
}

```

```

        Thread.sleep(2000);
        // 唤醒所有持有conditionB的线程
        service.signallB();
    }
}

```



执行ApplicationCondition，来看控制台输出结果：

Console Problems

<terminated> ApplicationCondi

a进入了awaitA方法

b进入了awaitB方法

启动唤醒程序

a被唤醒

a等待了：2s

启动唤醒程序

b被唤醒

b等待了：4s

a和b都进入各自的await()方法。首先执行的是

```

Thread.sleep(2000);
// 唤醒所有持有conditionA的线程
service.signallA();

```

使用conditionA的线程被唤醒，而后再唤醒使用conditionB的线程。学会使用Condition，那来用它实现生产者消费者模式。

生产者和消费者

首先来看业务类的实现：



```

package com.demo.test;

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

public class PCService {

    private Lock lock = new ReentrantLock();
    private boolean flag = false;
    private Condition condition = lock.newCondition();
    // 以此为衡量标志
    private int number = 1;

    /**
     * 生产者生产
     */
    public void produce() {
        try {
            lock.lock();

```

```

        while (flag == true) {
            condition.await();
        }
        System.out.println(Thread.currentThread().getName() + "-----生产-----");
        number++;
        System.out.println("number: " + number);
        System.out.println();
        flag = true;
        // 提醒消费者消费
        condition.signalAll();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}

/**
 * 消费者消费生产的物品
 */
public void consume() {
    try {
        lock.lock();
        while (flag == false) {
            condition.await();
        }
        System.out.println(Thread.currentThread().getName() + "-----消费-----");
        number--;
        System.out.println("number: " + number);
        System.out.println();
        flag = false;
        // 提醒生产者生产
        condition.signalAll();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
}

```



生产者线程代码：



```

package com.demo.test;

/**
 * 生产者线程
 * @author lixiaoxi
 */
public class MyThreadProduce implements Runnable{

    private PCSERVICE service;
}

```

```

public MyThreadProduce(PCService service) {
    this.service = service;
}

@Override
public void run() {
    for (;;) {
        service.produce();
    }
}
}

```



消费者线程代码：

```

package com.demo.test;

public class MyThreadConsume implements Runnable{

    private PCService service;

    public MyThreadConsume(PCService service) {
        this.service = service;
    }

    @Override
    public void run() {
        for (;;) {
            service.consume();
        }
    }
}

```



启动类：

```

package com.demo.test;

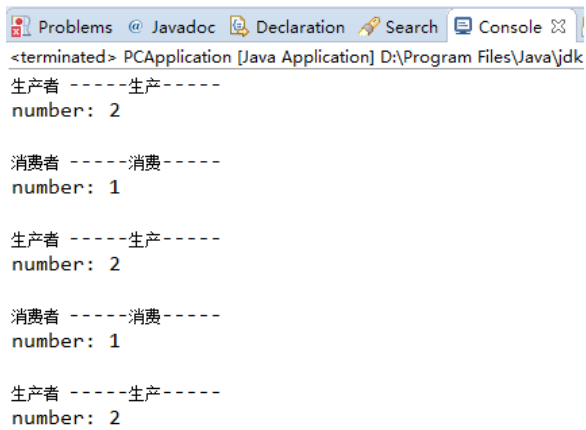
public class PCApplication {

    public static void main(String[] args) {
        PCService service = new PCService();
        Runnable produce = new MyThreadProduce(service);
        Runnable consume = new MyThreadConsume(service);
        new Thread(produce, "生产者 ").start();
        new Thread(consume, "消费者 ").start();
    }
}

```



执行PCApplication，看控制台的输出：



```

Problems Javadoc Declaration Search Console
<terminated> PCApplication [Java Application] D:\Program Files\Java\jdk
生产者 -----生产-----
number: 2

消费者 -----消费-----
number: 1

生产者 -----生产-----
number: 2

消费者 -----消费-----
number: 1

生产者 -----生产-----
number: 2

```

因为采用了无限循环，生产者线程和消费者线程会一直处于工作状态，可以看到，生产者线程执行完毕后，消费者线程就会执行，以这样的交替顺序，而且number也遵循着生产者生产+1，消费者消费-1的一个状态。这个就是使用ReentrantLock和Condition来实现的生产者消费者模式。

顺序执行线程

充分发掘Condition的灵活性，可以用它来实现顺序执行线程。

来看业务类代码：



```

package com.demo.test;

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

public class ConditionService {

    // 通过nextThread控制下一个执行的线程
    private static int nextThread = 1;
    private ReentrantLock lock = new ReentrantLock();
    // 有三个线程，所以注册三个Condition
    Condition conditionA = lock.newCondition();
    Condition conditionB = lock.newCondition();
    Condition conditionC = lock.newCondition();

    public void excuteA() {
        try {
            lock.lock();
            while (nextThread != 1) {
                conditionA.await();
            }
            System.out.println(Thread.currentThread().getName() + " 工作");
            nextThread = 2;
            conditionB.signalAll();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void excuteB() {
        try {

```



```

        lock.lock();
        while (nextThread != 2) {
            conditionB.await();
        }
        System.out.println(Thread.currentThread().getName() + " 工作");
        nextThread = 3;
        conditionC.signalAll();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}

public void excuteC() {
    try {
        lock.lock();
        while (nextThread != 3) {
            conditionC.await();
        }
        System.out.println(Thread.currentThread().getName() + " 工作");
        nextThread = 1;
        conditionA.signalAll();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
}

```



这里可以看到，注册了三个Condition，分别用于三个线程的等待和通知。

启动类代码：



```

package com.demo.test;

/**
 * 线程按顺序执行
 * @author lixiaoxi
 *
 */
public class ConditionApplication {

    private static Runnable getThreadA(final ConditionService service) {
        return new Runnable() {
            @Override
            public void run() {
                for (int i=0;i<10;i++) {
                    service.excuteA();
                }
            }
        };
    }
}

```

```

private static Runnable getThreadB(final ConditionService service) {
    return new Runnable() {
        @Override
        public void run() {
            for (int i=0;i<10;i++) {
                service.excuteB();
            }
        }
    };
}

private static Runnable getThreadC(final ConditionService service) {
    return new Runnable() {
        @Override
        public void run() {
            for (int i=0;i<10;i++) {
                service.excuteC();
            }
        }
    };
}

public static void main(String[] args) throws InterruptedException{
    ConditionService service = new ConditionService();
    Runnable A = getThreadA(service);
    Runnable B = getThreadB(service);
    Runnable C = getThreadC(service);

    new Thread(A, "A").start();
    new Thread(B, "B").start();
    new Thread(C, "C").start();
}
}

```

运行启动类, 查看控制台输出结果：

Problems @ Javadoc Declaration Search Console

<terminated> ConditionApplication [Java Application] D:\Program Files\J

```

A 工作
B 工作
C 工作
A 工作
B 工作
C 工作
A 工作
B 工作
C 工作
A 工作
B 工作
C 工作
A 工作
B 工作
C 工作

```

A,B,C三个线程一直按照顺序执行。

分类: [Java多线程](#)

好文要顶

关注我

收藏该文



平凡希

关注 - 1

粉丝 - 353

+加关注

0

推荐

0

反对

« 上一篇：[设计模式：工厂方法模式](#)

» 下一篇：[Java并发编程：线程池的使用](#)

posted @ 2017-10-15 20:31 平凡希 阅读(311) 评论(0) 编辑 收藏

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，访问[网站首页](#)。

【推荐】超50万VC++源码: 大型组态工控、电力仿真CAD与GIS源码库！

【福利】校园拼团福利，腾讯云1核2G云服务器10元/月！

【大赛】2018首届“顶天立地”AI开发者大赛

腾讯云

新注册用户域名抢购**1元起**

.com首年28元 .cn首年19元

立即抢购

Diagram showing domain names (.cn, .com, .xyz) and a laptop icon.

最新IT新闻:

- 摩拜单车宣布将在全国实行零门槛免押 发布摩拜助力车
 - 《我不是药神》大热 阿里影业大涨12% 北京文化连续3日涨停
 - 新一轮“黑车”治理开启 无资质网约车进入“严打周期”
 - Steam公布2018上半年畅销游戏榜：90款游戏上榜
 - 不让三星独美：LG开始准备折叠屏手机 国产厂商要跟进
- » 更多新闻...

最新知识库文章:

- 从Excel到微服务
 - 如何提升你的能力？给年轻程序员的几条建议
 - 程序员的那些反模式
 - 程序员的宇宙时间线
 - 突破程序员思维
- » 更多知识库文章...

昵称：平凡希
园龄：7年6个月
粉丝：353
关注：1
[+加关注](#)

2018年7月						
日	一	二	三	四	五	六
24	25	26	27	28	29	30
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4

搜索

常用链接

- 我的随笔
- 我的评论
- 我的参与
- 最新评论
- 我的标签

随笔分类

- Java IO(5)
- Java NIO(7)
- java8(3)
- Java多线程(20)
- java基础(11)
- Java集合(8)

Java虚拟机(9)
Linux
Mybatis(9)
mysql(15)
redis(9)
spring(8)
SpringMVC(8)
设计模式(5)

随笔档案
2018年6月 (1)
2018年4月 (1)
2018年1月 (3)
2017年12月 (2)
2017年11月 (10)
2017年10月 (3)
2017年9月 (6)
2017年8月 (9)
2017年7月 (6)
2017年6月 (12)
2017年5月 (3)
2017年4月 (3)
2017年3月 (24)
2017年2月 (14)
2017年1月 (2)

2016年12月 (5)
2016年11月 (4)
2016年10月 (7)
2016年9月 (5)
2016年8月 (1)
2016年7月 (6)

最新评论

1. Re:Java中的值传递和引用传递	
我是一个java小白看您的博客写的确实很专业，但是我对这个引用传递有我一点看法，如果说引用传递是传递变量的地址算是值传递的话，那么指针也算是一种值传递吧，java中是没有指针的，但是传递地址实际上和指.....	---色即是空
2. Re:SpringMVC+Spring+Mybatis框架集成	
大神啊，已收藏	--lslb
3. Re:SpringMVC工作原理	
终于看懂了	--一位不愿透露姓名的网友
4. Re:SpringMVC+Spring+Mybatis框架集成	
mark下!谢谢分享	--wizard_Q
5. Re:SpringMVC工作原理	
nice	--dengWeiHeng

阅读排行榜

1. SpringMVC工作原理(139623)
2. springmvc请求参数获取的几种方法(87383)
3. Spring系列之Spring常用注解总结(86852)
4. mysql 递归查询(41487)
5. 深入理解Java中的String(28612)

评论排行榜

1. Spring系列之Spring常用注解总结(16)
2. SpringMVC工作原理(11)
3. 深入理解Java中的String(10)
4. java集合框架综述(7)
5. springmvc请求参数获取的几种方法(6)

推荐排行榜

1. Spring系列之Spring常用注解总结(42)
2. SpringMVC工作原理(33)
3. 深入理解Java中的String(12)
4. java集合框架综述(10)
5. springmvc请求参数获取的几种方法(7)