Lock和Condition线程间通信—JDK1.5之后的解决办法

1. JDK1.5之前对于**多生产者-多消费者**中的线程间通信问题，且解决办法是利用锁对象的notifyAll唤醒所有的处于等待中的线程，但是为了能够实现同一个锁上能区别出**生产同步**还是**消费同步，即同一个锁上具有多个不同的监视器对象**，JDK1.5做出了改进，利用**Lock接口及子类**替代原来的任意对象的锁，每个锁可以通过**newCondition方法**获取多个锁的Condition对象**即监视器对象**，利用新的监视器对象中的**await、signal、signalAll方法**替代了Object类（所对象）中定义的**wait、notify、notifyAll方法**。

**简单记忆：一个锁上可以挂着多个监视器对象。**

1. JDK1.5后将**同步和锁**都封装到了对象中，将操作锁的隐式方式封装到了对象中，将**隐式动作**变成了**显式动作**。之前利用synchronized关键字声明同步，获取锁和释放锁都是隐式的，现在利用**lock方法和unlock方法**显式地获取锁和释放锁。
2. **接口Lock**：存在于**java.util.concurrent.locks**包中，没有子接口，但是有3个直接实现子类，即**ReentrantLock**, ReentrantReadWriteLock.ReadLock, ReentrantReadWriteLock.WriteLock。（后两个为**内部类**）
3. **ReentrantLock类：**java.util.concurrent.locks.ReentrantLock。直接继承 与java.lang.Object。

是一个类，有构造方法，可以直接new出对象，实现了Lock接口中的lock和unlock等方法。可以通过newCondition（）方法创建一个**Condition对象**。



1. **接口Condition：也存在于**java.util.concurrent.locks包中，**没有子接口**，只有两个**直接实现子类**，且是**内部类，**即：AbstractQueuedLongSynchronizer.ConditionObject,AbstractQueuedSynchronizer.ConditionObject 。  
   故没有直接new出Condition对象，而是利用Lock的子类中的newCondition（）方法获取一个Condition对象。
2. **接口Lock和接口Condition共同用于线程间通信。**
3. **接口Lock中的成员方法**：
4. **lock()方法：获取锁。**



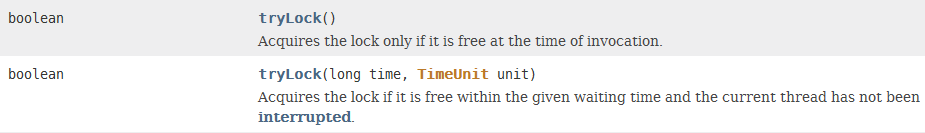
1. **unlock()方法：释放锁。**



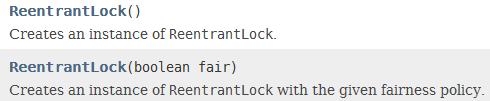
1. **newCondition()方法**：获取此锁的一个**Condition对象**。



1. **tryLock()方法**：



1. ReentrantLock类：存在于java.util.concurrent.locks包中，直接继承于Object类，**Lock接口的直接实现类**，可通过创建此类的一个对象获取一把锁对象。
2. **构造方法**：



1. **成员方法**：主要方法与Lock接口相同，不再赘述。
2. **使用示例：**

**为了防止获取锁后，代码出现异常，锁不能释放的情况发生，使用try{}finally{}或try{}catch{}finally{}语句。必须在finally中释放锁。**

示例:

public class Demo {

private final ReentrantLock lock = new ReentrantLock();//通常将锁定义为常量

public void show() {

**lock.lock();**

try {

//方法体

}catch(Exception e) {//可以对异常不进行捕捉

//异常处理代码

}finally {

**lock.unlock();**

}

}

}

1. **Condition接口**中的方法：**主要是await、signal、signalAll方法**。
2. **signal方法**：唤醒一个线程。

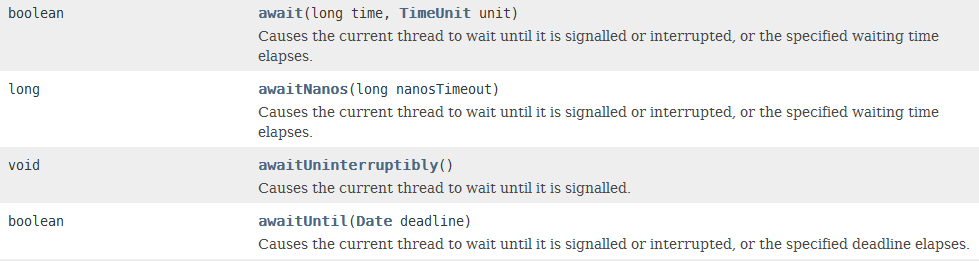


1. **signalAll方法**：唤醒所有线程。



1. **await**：





1. **Lock接口**：的出现替代了**同步代码块或者同步方法**；将同步的隐式锁操作变成了**显式的锁操作**，变得更加灵活，并且**可以在一个锁上设置多个监视器**。**lock（）方法获取锁，unlock（）释放锁，通常将xxx.unlock（）；放在finally语句中。**
2. **Condition 接口**：的出现替代了Object类中的**wait、notify、notifyAll方法**，将这些**监视器方法**单独进行了封装，变成**Condition监视器对象**，可以和任意锁进行组合使用，方法变成 **await、signal、signalAll**。
3. **分析：之前设计的多生产者多消费者模式，都是在需要停止生产的时候，利用this.notifyAll方法唤醒所有的线程包括生产者和消费者，可是如果唤醒生产者，该生产这仍然再次进入wait，从而浪费了时间。引入Lock和Condition后，在1把锁上可以设置多个监视器，这样生产者持有一个监视器，而消费者持有一个监视器，确保了生产结束后，叫醒消费者即可；同理消费结束后，叫醒生产者即可。**
4. **改进的多生产者-多消费者模式：**

仅仅是在资源类中做了改进：

public class Resource2 {

String name;

boolean flag;

int num = 0;

**private final ReentrantLock lock = new ReentrantLock();**

**private final Condition con1 = lock.newCondition();**

**private final Condition con2 = lock.newCondition();**

public Resource2(String name, boolean flag) {

super();

this.name = name;

this.flag = flag;

}

public void produce() {

**lock.lock();**

try {

while(flag) {

**con1.await();**

}

System.out.println(Thread.currentThread().getName()+"生产的"+this.name+"的编号是"+(++this.num));

flag = true;

**con2.signal();**

} catch (InterruptedException e) {

}finally {

**lock.unlock();**

}

}

public void consume() {

lock.lock();

try {

while(!flag) {

**con2.await();**

}

System.out.println(Thread.currentThread().getName()+"消费的......."+this.name+"的编号是"+(this.num));

flag = false;

**con1.signal();**

}catch(Exception e){

}finally {

lock.unlock();

}

}

}

1. **多消费者—多生产者：**

**public** **class** BoundBuffer {

**private** **int** putptr,takeptr,count = 0;

**private** Lock lock = **new** ReentrantLock();

**private** Condition put = lock.newCondition();

**private** Condition take = lock.newCondition();

**private** Object[] objs = **new** Object[100];

**public** **void** put(Object obj) {

**try** {

lock.lock();

**while**(count == objs.length) {

put.await();

}

objs[putptr] = obj;

**if** (++putptr == objs.length ) {

putptr = 0;

}

count++;

take.signalAll();

} **catch** (Exception e) {

}**finally** {

lock.unlock();

}

}

**public** Object take() {

Object obj = **null**;

**try** {

lock.lock();

**while**(count == 0) {

take.await();

}

obj = objs[takeptr];

**if**(++takeptr == objs.length) {

takeptr =0;

}

count--;

put.signalAll();

} **catch** (Exception e) {

}**finally** {

lock.unlock();

}

**return** obj;

}

}

**class** Put **implements** Runnable{

**private** BoundBuffer bb;

**public** Put(BoundBuffer bb) {

**super**();

**this**.bb = bb;

}

@Override

**public** **void** run() {

**for** (**int** i = 0; i < 200; i++) {

String str = "北京"+i;

bb.put(str);

System.***out***.println(Thread.*currentThread*().getName()+str);

}

}

}

**class** Take **implements** Runnable{

**private** BoundBuffer bb;

**public** Take(BoundBuffer bb) {

**super**();

**this**.bb = bb;

}

@Override

**public** **void** run() {

**for** (**int** i = 0; i < 200; i++) {

String str = (String) bb.take();

System.***out***.println(Thread.*currentThread*().getName()+str);

}

}

}

主方法：

**public** **class** Test222 {

**public** **static** **void** main(String[] args) {

BoundBuffer bb = **new** BoundBuffer();

**new** Thread(**new** Put(bb),"put1").start();

**new** Thread(**new** Take(bb),"take1").start();

**new** Thread(**new** Put(bb),"put2").start();

**new** Thread(**new** Take(bb),"take2").start();

**new** Thread(**new** Put(bb),"put3").start();

**new** Thread(**new** Take(bb),"take3").start();

**new** Thread(**new** Put(bb),"put4").start();

**new** Thread(**new** Take(bb),"take4").start();

}

}