参考：<http://blog.csdn.net/u011116672/article/details/51064752>

[**Java并发编程系列之十七：Condition接口**](http://blog.csdn.net/u011116672/article/details/51064752)

版权声明：本文为博主原创文章，转载请注明出处。

通过前面的文章，我们知道任何一个Java对象，都拥有一组监视器方法，主要包括wait()、notify()、notifyAll()方法（me:只有同步监听锁才能执行wait和notify以及notifyAll方法），这些方法与synchronized关键字配合使用可以实现等待/通知机制（me：线程之间的通信）。而且前面我们已经使用这种方式实现了生产者-消费者模式。类似地，Condition接口也提供类似的Object的监视器的方法，主要包括await()、signal()、signalAll()方法，这些方法与Lock锁配合使用也可以实现等待/通知机制（me:即线程之间的通信）。

相比Object实现的监视器方法，Condition接口的监视器方法具有一些Object所没有的特性：

1. Condition接口可以支持多个等待队列，在前面已经提到一个Lock实例可以绑定多个Condition，所以自然可以支持多个等待队列了
2. Condition接口支持响应中断，前面已经提到过
3. Condition接口支持当前线程释放锁并进入等待状态到将来的某个时间，也就是支持定时功能

使用Condition接口配合Lock锁的使用实例如下：

Lock lock = new ReentrantLock();

Condition condition = lock.newCondition();

public void conditionWait() throws InterruptedException {

lock.lock();

try {

//....

condition.await();

}finally {

lock.unlock();

}

}

public void conditionSignal(){

lock.lock();

try {

//...

condition.signal();

}finally {

lock.unlock();

}

}

一般而言，都会将Condition变量作为成员变量。当调用await方法后，当前线程会释放锁并进入Condition变量的等待队列，而其他线程调用signal方法后，通知正在Condition变量等待队列的线程从await方法返回，并且**在返回前已经获得了锁**。

现在我们已经知道了如何配合Condition和Lock锁实现等待/通知机制，那么我们使用这种方式实现生产者-消费者模式：

package com.rhwayfun.concurrency.r0405;

import java.text.DateFormat;

import java.text.SimpleDateFormat;

import java.util.Date;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

/\*\*

\* Created by rhwayfun on 16-4-5.

\*/

public class ConditionProducerConsumerDemo {

//日期格式器

private static DateFormat format = new SimpleDateFormat("HH:mm:ss");

static class Info{

//作者

private String author;

//标题

private String title;

//是否开始生产的标志

private boolean produce = true;

//Lock锁

private Lock lock = new ReentrantLock();

//Condition变量

private Condition condition = lock.newCondition();

public Info(){}

public Info(String author, String title) {

this.author = author;

this.title = title;

}

public String getAuthor() {

return author;

}

public void setAuthor(String author) {

this.author = author;

}

public String getTitle() {

return title;

}

public void setTitle(String title) {

this.title = title;

}

/\*\*

\* 生产者执行的生产方法

\* @param author

\* @param title

\* @throws InterruptedException

\*/

public void set(String author,String title) throws InterruptedException {

lock.lock();

try {

//没有开始生产就等待

while (!produce){

condition.await();

}

//如果已经开始生产

this.setAuthor(author);

TimeUnit.SECONDS.sleep(1);

this.setTitle(title);

//表示已经停止了生产可以取数据了

produce = false;

//通知消费者

condition.signal();

}finally {

lock.unlock();

}

}

/\*\*

\* 消费者执行的消费方法

\* @throws InterruptedException

\*/

public void get() throws InterruptedException {

lock.lockInterruptibly();

try {

//如果已经开始生产就等待

while (produce){

condition.await();

}

//如果没有在生产就可以取数据

System.out.println(Thread.currentThread().getName() + ":" + this.getAuthor()

+ "=" + this.getTitle() + " at "

+ format.format(new Date()));

//表示我已经取了数据，生产者可以继续生产

produce = true;

//通知生产者

condition.signal();

}finally {

lock.unlock();

}

}

}

static class Producer implements Runnable{

private Info info;

public Producer(Info info) {

this.info = info;

}

public void run() {

boolean flag = true;

for (int i = 0; i < 5; i++){

if (flag){

try {

info.set("authorA","titleA");

System.out.println(Thread.currentThread().getName() + ":" + info.getAuthor() + "="

+ info.getTitle() + " at " + format.format(new Date()));

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

e.printStackTrace();

}

flag = false;

}else {

try {

info.set("authorB","titleB");

System.out.println(Thread.currentThread().getName() + ":" + info.getAuthor() + "="

+ info.getTitle() + " at " + format.format(new Date()));

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

e.printStackTrace();

}

flag = true;

}

}

}

}

static class Consumer implements Runnable{

private Info info;

public Consumer(Info info) {

this.info = info;

}

public void run() {

for (int i = 0; i < 5; i++){

try {

info.get();

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

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

Info info = new Info();

Thread producer = new Thread(new Producer(info),"Producer");

Thread consumer = new Thread(new Consumer(info),"Consumer");

producer.start();

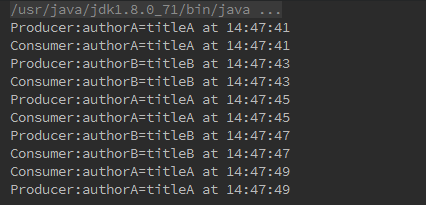
TimeUnit.SECONDS.sleep(1);

consumer.start();

}

}

运行结果如下：



与使用Object的监视器方法达到了同样的效果，也许看不出Condition配合Lock锁的优势何在。但是在复杂多线程的编程中，这种方式可以体现出其优势。所以一般使用的时候仍然是Object的监视器方法居多。