

ReentrantLock有哪些特性？

2015-09-23 18:27 feiying 0 阅读 73

ReentrantLock是 Java API级别的锁，它基本和synchronized的关键字类似，也是独占锁；

但是synchronized关键字因为是语言机制提供的，因为只是一个标识符，作用是通知线程去看看对象的监视器中是否被其他线程独占，如果有的话，就等待排队，没有的话直接抢到对象独占权；

而ReentrantLock除了这些功能之外，它可以很灵活的加锁，解锁，公平锁，查看线程占用，线程中断等特性。

如下面几点内容：

1.加锁解锁的灵活性：

```
class X {
    private final ReentrantLock lock = new ReentrantLock();
    // ...

    public void m() {
        lock.lock(); // block until condition holds
        try {
            // ... method body
        } finally {
            lock.unlock()
        }
    }
}
```

类似于操作系统的mutex_lock的机制，加锁和解锁通过调用API函数来完成（操作系统的lock是直接调用的系统调用），

这样比较方便和灵活，我想加锁就加锁，我可以根据程序出现的异常情况，在一段代码根据程序的环境多次进行加解锁的操作，比synchronized包一块要灵活得多；

2.公平锁

ReentrantLock(boolean fair)
创建一个具有给定公平策略的 ReentrantLock。

看到ReentrantLock在构造方法的时候，可以传入一个boolean类的标识，它代表的策略就是公平策略；

所谓的公平策略就是此类的构造方法接受一个可选的公平 参数。当设置为 true 时，在多个线程的争用下，这些锁倾向于将访问权授予等待时间最长的线程。否则此锁将无法保证任何特定访问顺序。

下载《开发者大全》

下载 (/download/dev.apk)



与采用默认设置（使用不公平锁）相比，使用公平锁的程序在许多线程访问时表现为很低的总体吞吐量（即速度很慢，常常极其慢），但是在获得锁和保证锁分配的均衡性时差异较小。

不过要注意的是，公平锁不能保证线程调度的公平性，使用公平锁的众多线程中的一员可能获得多倍的成功机会，这种情况发生在其他活动线程没有被处理并且目前并未持有锁时。

通过上述的API中的注释进行的描述，可以总结的是，公平锁也仅仅是相对的公平，对成体吞吐量可能会降低，使用的时候一定需要慎重起见，不过如果你遇到的情况是“线程快饿死”的情况，

那么不妨设置一下公平锁，这样尽量会避免线程出现等不到对象而僵死的情况；

3.try一下试试

synchronized关键字没有试一试一说，我代码执行到synchronized这句话，这就是要等锁，一直在排队，直到等到锁为止；

这种思路倒是没有问题，因为同步确实有时候锁是必须的，但有一些情况，我可能准备了n套方案，即使拿不到锁，我去做一些别的事情，回来再看看能不能上上锁；

这种情况有时候效率是比较高的，API级别提供了tryLock这个机制

boolean	<code>tryLock()</code>	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	<code>tryLock(long timeout, TimeUnit unit)</code>	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。

仅在调用时锁未被另一个线程保持的情况下，才获取该锁。

如果该锁没有被另一个线程保持，并且立即返回 true 值，则将锁的保持计数设置为 1。即使已将此锁设置为使用公平排序策略，但是调用 tryLock() 仍将立即获取锁（如果有可用的），而不管其他线程当前是否正在等待该锁。在某些情况下，此“闯入”行为可能很有用，即使它会打破公平性也如此。如果希望遵守此锁的公平设置，则使用 tryLock(0, TimeUnit.SECONDS)，它几乎是等效的（也检测中断）。

如果当前线程已经保持此锁，则将保持计数加 1，该方法将返回 true。

如果锁被另一个线程保持，则此方法将立即返回 false 值。

这个机制也是和操作系统的机制差不多的机制，操作系统也有对应的函数，而synchronized是因为语言机制提供的，所以没法有这种功能；

4.快速响应中断请求

有一些时候，线程会有根据情况接受一个中断信号，在java领域这个就是由Thread.interrupter方法来调用操作系统的信号机制来发出这个信号，

而这个时候，你还要获取锁，在等待队列中傻等，显然程序如果这么写就太傻了，

对于synchronized关键字，和ReentrantLock.lock的方法，都是傻等，即使明知道此线程继续执行也无意义，也要固执己见的等着；

```
public class TestLock {
    public static void main(String[] args) {
        Thread t1 = new Thread(new RunIt());
        Thread t2 = new Thread(new RunIt());

        t1.start();
        t2.start();
        t2.interrupt();
    }
}

class RunIt implements Runnable{
    private static Lock lock = new ReentrantLock();
    @Override
    public void run() {
        try {
            runJob();
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName()+" Interrupted from runJob.");
        }
    }

    private void runJob() throws InterruptedException{
        lock.lockInterruptibly();
        //lock.lock();
        System.out.println(Thread.currentThread().getName()+" 开始！执...");
        try{
            System.out.println(Thread.currentThread().getName() + " running");
            TimeUnit.SECONDS.sleep(3);
            System.out.println(Thread.currentThread().getName() + " finished");
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName() + " interrupted");
        } finally {
            lock.unlock();
        }
    }
}
```

上述程序中，总共启动两个线程，而t2线程在线程启动的时候，就由main函数发了一个中断信号，因而，t2线程执行也没有意义，t2势必会执行到catch到InterruptedException异常的代码中；

那么，有没有办法让t2线程别在lock的时候一直等着，反正也没什么意义，

====》这就是lockInterruptibly方法的作用了，启动信号监听，一旦接受到中断信号，立即返回；

可以试试调用ReentrantLock.lock的输出结果：

```
Thread-0 开始！执...
Thread-0 running
Thread-0 finished
Thread-1 开始！执...
Thread-1 running
Thread-1 interrupted
```

因为有锁加入，所以Thread0会按部就班的执行完，把锁给Thread1，最后Thread1执行，过程中有一个中断信号，相当于Thread1白等了这么长时间，

从程序的执行上，你明显就可以感受到这一个延迟的效果，程序执行到结束很慢；

下载《开发者大全》 下载 (/download/dev.apk)

而如果采用ReentrantLock.lockInterruptibly的话:

```
Thread-0 运行 | 块...
Thread-1 Interrupted from runJob.
Thread-0 running
Thread-0 finished
```

Thread0执行过程中, Thread1有机会抢到了一些时间片执行一下, lockInterruptibly方法加了一个中断的信号监视, 发现这时候来了一个中断信号,

Thread1立刻就返回了, 也就是执行到catch到InterruptedException异常的代码中, 绝对不等了;

稍后Thread0继续抢到时间片, 继续执行;

上述的两个程序虽然只有一个lock方法的区别, 但是时间相差一倍, 对于有具体场景的时候, 需要使用这个lock Interruptibly方法;

5.与线程相关的相关信息的获取

synchronized关键字没有任何的办法, 与线程交互, 还是因为毕竟是一个关键字嘛;

而ReentrantLock就有很多的API方法, 可以与线程进行交互:

int	<code>getHoldCount()</code>	查询当前线程保持此锁的次数。
protected Thread	<code>getOwner()</code>	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 Collection<Thread>	<code>getQueuedThreads()</code>	返回一个 collection, 它包含可能正等待获取此锁的线程。
int	<code>getQueueLength()</code>	返回正等待获取此锁的线程估计数。
protected Collection<Thread>	<code>getWaitingThreads(Condition condition)</code>	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。
int	<code>getWaitQueueLength(Condition condition)</code>	返回等待与此锁相关的给定条件的线程估计数。
boolean	<code>hasQueuedThread(Thread thread)</code>	查询给定线程是否正在等待获取此锁。
boolean	<code>hasQueuedThreads()</code>	查询是否有些线程正在等待获取此锁。
boolean	<code>hasWaiters(Condition condition)</code>	查询是否有些线程正在等待与此锁有关的给定条件。
boolean	<code>isFair()</code>	如果此锁的公平设置为 true, 则返回 true。
boolean	<code>isHeldByCurrentThread()</code>	查询当前线程是否保持此锁。
boolean	<code>isLocked()</code>	查询此锁是否由任意线程保持。

方法很多, 甚至可以获取到等待队列的线程, 返回一个Collection<Thread>这样的集合, 可以查等待队列的深度, 这些都是synchronized可望而不可即的;

6.Condition

下载《开发者大全》

下载 (/download/dev.apk)

ReentrantLock可以对当前的对象监视器的语义进行扩充，也就是通过这个ReentrantLock 可以生成n个Condition

[Condition](#) | [newCondition\(\)](#)

应用服务器技术讨论圈
返回用来与此 [Lock](#) 实例一起使用的 [Condition](#) 实例。

具体说来，一个对象跟着一个监视器，这个监视器是JVM实现的，用以对当前独占这个对象所有权的线程进行计数，这个其实也就是JVM中对锁的C++实现了；

而如果一个线程中，调用到object.wait方法，那么这个线程主动就会进入监视器中的等待队列中，直到其他线程干完活了，发一个消息，使用notify或者notifyAll，

告诉在这个object等待区中的等着的线程都醒醒，你们抢下，看谁能拿到这个object的所有权，然后执行；

notify是每一次只唤醒1个等待线程，notifyall是全唤醒，让你们这些线程醒来后争一把，看谁能拿到所有权；

上述的过程，是wait和notify的实现原理，但是你有没有发现，当前拿到object所有权的线程干完后，我发出去的消息就是这一种类型，就是一个信号，

那么可以产生一个疑问，是不是可以对这个信号进行定制，

例如，我现在干完活了，发现我饿了，我直接发一个外卖的信号，让外卖那个线程醒来，给我送点吃的；

我觉得我需要打车了，我发一个滴滴打车的信号，让出租车线程赶紧过来，我还坐车；

。。。等等。

我可以对这个信号进行定制，这就是Condition的作用，也就是所谓的条件变量，

当出现某种条件的时候，我让符合这个条件的线程阻塞唤醒执行，或者让其线程进行阻塞；


```

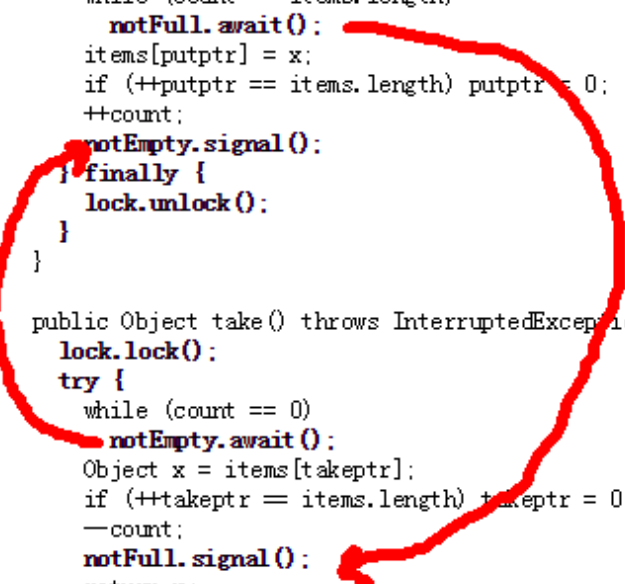
class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();

    final Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put(Object x) throws InterruptedException {
        lock.lock();
        try {
            while (count == items.length)
                notFull.await();
            items[putptr] = x;
            if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    public Object take() throws InterruptedException {
        lock.lock();
        try {
            while (count == 0)
                notEmpty.await();
            Object x = items[takeptr];
            if (++takeptr == items.length) takeptr = 0;
            --count;
            notFull.signal();
            return x;
        } finally {
            lock.unlock();
        }
    }
}

```



应用服务器技术讨论圈

上述的程序就是对这个信号的定制，notFull就是当满的时候发出的信号，不能继续往里put了，该take了，这时候put的线程执行到这里，调用await方法，全部阻塞住了，

当take了一个了，说明这里面没满了，你还可以往里面塞，这个时候notFull就发出一个信号，阻塞的那个put的线程直接又唤醒开始往buffer里面塞了；

这仅仅是一种情况，notEmpty是另外一种情况，它这个信号关注的是空的状态，类似于notFull；

值得一提的是，Condition信号可以进行定制，但Condition是依附与ReentrantLock的，他所操控的就是这同一把锁而已；

实质上来说，也就是把对象监视区的接受信号的条件定制化而已，什么情况阻塞相关线程，什么情况又唤醒这个线程，条件不再仅仅局限于wait和notify这一种，可以定义n种条件；

可以看到，synchronized打死都不可能有这种功能的；

=====总结=====》

ReentrantLock实际功能和synchronized差不多，但ReentrantLock功能相对来讲更多一些；

下载《开发者大全》

下载 (/download/dev.apk)



虽然如此，但不是说你以后遇到所有的锁的问题，都要用ReentrantLock，相反，使用ReentrantLock会让程序代码变得难以维护，并且性能在某种情况下(公平锁)还不一定就如synchronized；

结合具体的对象独占的场景，使用对应的锁，这才是java中锁的最佳实践；

分享🔗：

阅读 73 0

应用服务器技术讨论圈 更多文章

东方通加码大数据业务 拟募资8亿收购微智信业 (/html/308/201504/206211355/1.html)

玩转Netty – 从Netty3升级到Netty4 (/html/308/201504/206233287/1.html)

金蝶中间件2015招聘来吧！Come on！ (/html/308/201505/206307460/1.html)

GlassFish 4.1 发布，J2EE 应用服务器 (/html/308/201505/206323120/1.html)

Tomcat对keep-alive的实现逻辑 (/html/308/201505/206357679/1.html)

猜您喜欢

粉刷匠的自我修养——android的过度渲染介绍及优化 (/html/212/201606/2247483767/1.html)

大象旅游业背后，被损害的它们和他们 | 科学人 (/html/258/201606/2651584239/1.html)

扁平界面设计的一些技巧和注意问题 (/html/438/201403/200075605/1.html)

热门资讯 - 2013-09-12 (/html/292/201309/10000090/1.html)

我的 React Native 技能树点亮计划 の 代码风格统一工具 EditorConfig (/html/136/201606/2651036611/1.html)