## 有了synchronized为什么还要Lock？

1. 尝试非阻塞地获取锁
2. 获取锁的过程可以被中断
3. 超时获取锁

## Lock的标准用法

参加代码

## Lock的常用API

Lock()

lockInterruptibly：可中断

tryLock尝试非阻塞地获取锁

unlock()

## 锁的可重入

递归的时候发生锁的重入，没有锁的可重入，就会死锁

## 公平和非公平锁

公平锁，先对锁发出获取请求的一定先被满足。公平锁的效率比非公平锁效率要低。

## 读写锁ReentrantReadWriteLock

允许多个读线程同时进行，但是只允许一个写线程(不允许其他读线程和写线程)，支持读多写少场景，性能会有提升。

## Condition接口有何用处？

Object wait,notify/all Condition接口和Lock配合来实现等待通知机制

## Condition常用方法和使用范式

参见代码

## 结合ReentrantLock和Condition实现线程安全的有界队列

参见代码

## 什么是AbstractQueuedSynchronizer？为什么我们要分析它？

AQS:抽象队列同步器，原理是：当多个线程去获取锁的时候，如果获取锁失败了，当前线程就会被打包成一个node节点放入同步队列里面使用LockSuport的park方法阻塞起来，如果有线程释放了锁，放入同步队列的线程就会被LockSupport的unpark方法唤醒再次去获取锁，如果获取锁又失败了就再次打包成node节点放入同步队列，这样不断的循环直到拿到锁

为什么药分析AQS：因为线程都是基于AQS实现的

## AQS的基本使用方法

同步器的主要使用方式是继承，子类通过继承同步器并实现它的抽象方法来管理同步状态。可重写的方法有：

tryAcquire 独占锁获取

tryRelease 独占锁释放

tryAcquireShared 共享锁获取

tryReleaseShared 共享锁释放

isHeldExclusively 快速判断被线程独占

同步器的设计是基于**模板方法模式,** 使用者需要继承同步器并重写指定的方法，随后将同步器组合在自定义同步组件的实现中，并调用同步器提供的模板方法，而这些模板方法将会调用使用者重写的方法。

对同步状态进行更改，这时就需要使用同步器提供的3个方法

getState() 获取同步状态

setState 设置同步状态

compareAndSetState 原子的设置同步状态

来进行操作。

## 了解LockSupport

LockSupport定义了一组的公共静态方法，这些方法提供了最基本的线程阻塞和唤醒功能，而LockSupport也成为构建同步组件的基础工具。LockSupport定义了一组以park开头的方法用来阻塞当前线程，以及unpark(Thread thread)方法来唤醒一个被阻塞的线程。

## 同步队列

同步器依赖内部的同步队列（一个FIFO双向队列）来完成同步状态的管理，当前线程获取同步状态失败时，同步器会将当前线程以及等待状态等信息构造成为一个节点（Node）并将其加入同步队列。同步器拥有首节点（head）和尾节点（tail），没有成功获取同步状态的线程将会成为节点加入该队列的尾部。

## 独占式同步状态获取与释放

通过调用同步器的acquire(int arg)方法可以获取同步状态，其主要逻辑是：首先调用自定义同步器实现的tryAcquire(int arg)方法，该方法保证线程安全的获取同步状态，如果同步状态获取失败，则构造同步节点（独占式Node.EXCLUSIVE，同一时刻只能有一个线程成功获取同步状态）并通过addWaiter(Node node)方法将该节点加入到同步队列的尾部，最后调用acquireQueued(Node node,int arg)方法，使得该节点以“死循环”的方式获取同步状态。如果获取不到则阻塞节点中的线程，而被阻塞线程的唤醒主要依靠前驱节点的出队或阻塞线程被中断来实现。

## 共享式同步状态获取与释放

共享式获取与独占式获取最主要的区别在于同一时刻能否有多个线程同时获取到同步状态。在acquireShared(int arg)方法中，同步器调用tryAcquireShared(int arg)方法尝试获取同步状态，tryAcquireShared(int arg)方法返回值为int类型，当返回值大于等于0时，表示能够获取到同步状态。因此，在共享式获取的自旋过程中，成功获取到同步状态并退出自旋的条件就是tryAcquireShared(int arg)方法返回值大于等于0。

**锁的可重入**

当一个线程获得锁之后，再次进入同步快任然能获取到锁执行，实现原理是先判断当前线程是不是已经获取到锁的线程，如果是就把状态加1；释放锁的时候拿了几次锁就要释放几次

如果没有实现锁的可重入则会出现一个线程拿到了锁之后再次获取锁，这样就会自己等待自己出现死锁

**公平锁和非公平锁**

公平锁：一个线程去竞争锁的时候先看前面有没有其他线程在等待获取锁，如果有就不参与

非公平锁：一个线程去获取锁的时候不管前面有没有线程在等待获取锁都有参与竞争

## 来实现一个共享式的同步工具类

参见代码。

## 了解ReentrantLock的实现

重进入是指任意线程在获取到锁之后能够再次获取该锁而不会被锁所阻塞，该特性的实现需要解决以下两个问题。

1）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。

2）锁的最终释放。

nonfairTryAcquire方法增加了再次获取同步状态的处理逻辑：通过判断当前线程是否为获取锁的线程来决定获取操作是否成功，如果是获取锁的线程再次请求，则将同步状态值进行增加并返回true，表示获取同步状态成功。同步状态表示锁被一个线程重复获取的次数。

如果该锁被获取了n次，那么前(n-1)次tryRelease(int releases)方法必须返回false，而只有同

步状态完全释放了，才能返回true。可以看到，该方法将同步状态是否为0作为最终释放的条件，当同步状态为0时，将占有线程设置为null，并返回true，表示释放成功。

## 了解ReentrantReadWriteLock的实现

读写锁的自定义同步器需要在同步状态（一个整型变量）上维护多个读线程和一个写线程的状态，使得该状态的设计成为读写锁实现的关键。如果在一个整型变量上维护多种状态，就一定需要“按位切割使用”这个变量，读写锁将变量切分成了两个部分，高16位表示读，低16位表示写。

读状态是所有线程获取读锁次数的总和，而每个线程各自获取读锁的次数只能选择保存在ThreadLocal中，由线程自身维护。

## 了解Condition的实现

等待队列是一个FIFO的队列，在队列中的每个节点都包含了一个线程引用，该线程就是在Condition对象上等待的线程，如果一个线程调用了Condition.await()方法，那么该线程将会释放锁、构造成节点加入等待队列并进入等待状态。

一个Condition包含一个等待队列，新增节点只需要将原有的尾节点nextWaiter指向它，并且更新尾节点即可。

调用Condition的signal()方法，将会唤醒在等待队列中等待时间最长的节点（首节点），在唤醒节点之前，会将节点移到同步队列中。

调用该方法的前置条件是当前线程必须获取了锁，可以看到signal()方法进行了isHeldExclusively()检查，也就是当前线程必须是获取了锁的线程。接着获取等待队列的首节点，将其移动到同步队列并使用LockSupport唤醒节点中的线程。

通过调用同步器的enq(Node node)方法，等待队列中的头节点线程安全地移动到同步队列。

Condition的signalAll()方法，相当于对等待队列中的每个节点均执行一次signal()方法，效果就是将等待队列中所有节点全部移动到同步队列中，并唤醒每个节点的线程。