1. Lock接口
2. 概述

锁是用来控制多个线程访问共享资源的方式，synchronized关键字可以实现锁的功能。JDK5之后，并发包中新增了Lock接口和相关实现类用来实现锁功能，它提供了同步工鞥你，只是在使用时需要显式地获取和释放锁。它的优势是灵活地获取和释放锁，可中断地获取锁，超时获取锁。

1. 基本使用

Lock锁的使用方式很简单，先获取锁，然后释放锁：

ReentrantLock lock = new ReentrantLock();  
lock.lock();  
try{}  
finally {  
 lock.unlock();  
}

在finally块中释放锁，目的是保证在获取到锁之后，最终能够释放锁，不要将获取锁的过程写在try块中，因为如果在获取锁的过程中发生异常，异常抛出的同时，锁也会被释放。

1. 队列同步器
2. 概述

队列同步器（AbstractQueuedSynchronized, AQS）是用来构建锁或者其他同步组件的基础框架，使用了一个int成员变量表示同步状态，通过内置的FIFO队列来完成资源获取线程的排队工作。子类通过继承AQS来实现它的抽象方法来管理同步状态，AQS提供了三个方法来管理同步状态：

getState(); // 获取同步状态

setState(int newState); //设置同步状态

compareAndSetState(int expect, int update); //使用CAS设置同步状态

它们能够保证状态的改变是安全的，同步器支持独占式地获取同步状态，也支持共享式地获取同步状态，方便实现不同类型的同步组件（ReentrantLock, ReentrantReadWriteLock）。同步器面向锁的实现者，简化了锁的实现方式，屏蔽了同步状态管理，线程的排队，等待与唤醒等底层操作。

1. AQS
2. 概述

AQS的设计是基于模板方法模式的，使用者需要继承同步器并重写指定的方法，然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，这些模板方法会调用使用者重写的方法。

1. 重写方法

独占式获取与释放同步状态，共享式获取与释放同步状态。

1. 模板方法

独占式获取与释放同步状态，共享式获取与释放同步状态，查询同步队列中等待线程情况。自定义同步组件将使用同步器提供的模板方法来实现自己的同步语义。

1. AQS的实现分析
2. 概述

AQS中包括同步队列，独占式同步状态获取与释放，共享式同步状态获取与释放。

1. 同步队列

AQS依赖内部的同步队列（FIFO双向队列）来完成同步状态管理，当前线程获取同步状态信息失败时，同步器会将当前线程以及等待状态信息构造为一个节点加入同步队列中，同时阻塞当前线程，当同步状态释放时，会将首节点的线程唤醒，使其尝试获取同步状态。

首节点是获取同步状态成功的节点，首节点的线程在释放同步状态时，将会唤醒后继节点，后继节点在获取同步状态成功时将自己设置为首节点。

1. 独占式同步状态获取与释放

调用同步器的acquire(int arg)方法可以获取同步状态，该方法对中断不敏感，即当前线程进入同步队列后，后续对该线程进行中断操作时，线程不会从同步队列中移出。acquire()方法首先调用了自定义同步器实现的tryAcquire()方法，该方法保证了线程安全地获取同步状态；如果获取失败，则构造同步节点，加入同步队列的尾部，最后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，被阻塞线程的唤醒依靠前驱节点的出队或阻塞线程被中断。在释放同步状态时，同步器调用tryRelease()方法释放，然后唤醒头节点的后续节点。

1. 共享式同步状态获取与释放

与独占式的区别就是同一时刻有多个线程同时获取到同步状态。在acquireShared()方法中，同步器调用tryAcquireShared()方法来尝试获取同步状态，返回值为int类型，当返回值>= 0时，表示能够获取到。因此在共享式获取的自旋过程中，如果当前节点的前驱为头结点时，尝试获取同步状态，如果返回值>=0，则获取成功从自旋退出。

1. 重入锁
2. 概述

支持重进入的锁，能够支持一个线程对资源的重复加锁，还支持获取锁时的公平和非公平选择。synchronized关键字隐式地支持冲进入，ReentrantLock锁在调用lock()方法时，已经获取到锁的线程，依然能够再次调用lock()方法获取锁。对于公平性，在绝对时间上，先对锁进行获取请求的线程一定先被满足，可以使用一个构造函数来选择公平性，非公平锁效率更高，因为线程切换次数较少。

1. 实现重进入

当线程再次获取锁时，锁需要识别获取锁的线程是否为当前占有锁的线程，如果是则再次获取。如果线程重复n次获取了锁，在第n次释放锁后，其他线程才可以获取到锁，这是通过对锁计数实现的，锁被释放时，计数等于0即表示成功释放。

ReentrantLock是通过组合自定义同步器来实现锁的获取与释放的，增加判断逻辑，如果当前线程已经获取了锁，则增加同步状态值并返回true，在释放同步状态时就需要减少同步状态值了。只有状态为0时，才会将占有线程设置为null，返回true，表示释放成功。

1. 公平锁实现

区别是在判断条件中多了一个方法，判断当前节点是否有前驱节点，如果有表示还有更早的线程没有获取锁，因此它需要等待。公平锁每次都是从同步队列的头结点获取锁，而非公平锁会出现线程连续获得锁的现象，这是因为当一个线程请求锁时，只要获取了同步状态即成功获取锁，那么刚刚释放的线程再次获取同步状态的几率很大。

非公平锁是默认实现，因为它的性能好，不需要过多的线程切换，公平锁为了保证FIFO，需要大量的线程切换。

1. 读写锁
2. 概述

读写锁在同一时刻允许多个读线程访问，在写线程访问时，所有的读线程和写线程都会被阻塞，读写锁维护了一对锁，一个读锁，一个写锁。因为读的需求远大于写，因此读写锁能够提供比排它锁更好的并发性和吞吐量，JUC中提供的实现是ReentrantReadWriteLock，支持重进入，支持锁降级，写锁降级为读锁。

1. 接口与实现
2. ReadWriteLock

接口ReadWriteLock仅仅定义了获取读锁和写锁两个方法。

1. ReentrantReadWriteLock

该接口的实现类读写锁，提供了一些便于外界监控其内部工作状态的方法，如返回当前锁被读取的次数等等。

1. 实现分析
2. 读写状态设计

读写锁同样依赖自定义同步器来实现同步功能，读写状态就是同步器中的同步状态，在ReentrantLock中同步状态是锁被一个线程重复获取的次数，而读写锁的同步状态需要为何多个读线程以及一个写线程的状态。在一个整型变量上维护多种状态，就需要按位切割，因此将状态分为两个部分，高16位表示读，低16位表示写，这样就能够表示两个值了。

1. 写锁的获取与释放

写锁是一个支持重进入的排他锁，如果当前线程获取了写锁，则增加写状态，如果读锁此时已获取或者没有获取到写锁，则进入等待状态。读锁存在时，写锁不能获取，因为读写锁要确保写锁的操作对读锁可见。只有等待所有的读锁释放后，写锁才能被获取。

1. 读锁的获取与释放

读锁是一个支持重进入的共享锁，能够被多个线程同时获取，在没有其他写线程访问时，读锁总会被成功地获取，增加读状态。如果当前线程在获取读锁时，写锁被其他线程获取，则当前线程进入等待。

JDK6的读锁实现更加复杂，添加了新的函数，如返回当前线程获取读锁的次数的函数，这个实现是通过ThreadLocal实现的，每个线程将各自获取读锁的次数保存在ThreadLocal中，由线程自身维护。

1. 锁降级

写锁降级为读锁，即持有写锁，获取读锁，最后释放写锁的过程。

1. Condition
2. 概述

任意一个Java对象都有一组监视器方法定义在Object类中，如wait()和notify()，这些方法和Synchronized配合可以实现等待/通知模式，Condition接口也可以和Lock配合实现这种功能。

1. 方法描述

首先要获取到Condition对象关联的锁，Condition对象是通过调用Lock对象的newCondition()方法创建出来的，即Condition是依赖Lock对象的。

Lock lock = new ReentrantLock();

Condition condition = lock.newCondition();

当调用condition的await()方法后，当前线程会释放锁，并再次等待；而其他线程则会调用condition对象的signal()方法，通知当前线程后，当前线程才从await()方法中返回，如果返回了则表示已经获得了锁。

1. Condition的实现

每个Condition对象都包含了一个等待队列，队列中的每个节点都是一个线程引用，即在Condition对象上等待的线程，线程调用了await()方法后，就会释放锁，构造成节点加入等待队列中并进入等待状态。Lock拥有一个同步队列和多个等待队列，当调用Conditon的await()方法后，相当于同步队列的首节点（获取锁的节点）移到Condition的等待队列中。

signal()方法则会唤醒等待队列中的首节点，移动到同步队列中，使用LockSupport唤醒线程，该线程会加入获取同步状态的竞争中。调用signal()方法的前置条件是当前线程必须获取了锁。