## 可重入锁Shared Reentrant Lock

首先我们先看一个全局可重入的锁。 Shared意味着锁是全局可见的，客户端都可以请求锁。 Reentrant和JDK的ReentrantLock类似，意味着同一个客户端在拥有锁的同时，可以多次获取，不会被阻塞。  
它是由类InterProcessMutex来实现。它的主要方法：

1. // 构造方法
2. publicInterProcessMutex(CuratorFramework client,String path)
3. publicInterProcessMutex(CuratorFramework client,String path,LockInternalsDriver driver)
4. // 通过acquire获得锁,并提供超时机制：
5. publicvoid acquire()throwsException
6. publicboolean acquire(long time,TimeUnit unit)throwsException
7. // 撤销锁
8. publicvoid makeRevocable(RevocationListener<InterProcessMutex> listener)
9. publicvoid makeRevocable(finalRevocationListener<InterProcessMutex> listener,Executor executor)

可以使用ConnectionStateListener处理连接状态的改变。当连接LOST时你不再拥有锁.

可重入锁示例:

InterProcessMutex interProcessMutex=new InterProcessMutex(client, "/test");

if(interProcessMutex.acquire(60, TimeUnit.SECONDS)){

System.out.println("获取可重入锁成功");

}

if(interProcessMutex.acquire(60, TimeUnit.SECONDS)){

System.out.println("再次获取可重入锁成功");

}

interProcessMutex.release();

interProcessMutex.release();

运行结果: 两次都可以获取到锁.

既然是可重入锁，则可以在一个线程中多次调用acquire,在线程拥有锁时它总是返回true。

注意：使用InterProcessMutex时,不应该在多个线程中用同一个InterProcessMutex，可以在每个线程中都生成一个InterProcessMutex实例，它们的path都一样，这样它们可以共享同一个锁,如果多个线程中都使用同一个InterProcessMutex实例的话,会导致锁失效,即zookeeper认为请求获取锁的是同一线程,从而每次都返回true.此外,需要注意的是,既然获取了多次锁,那么最后一定要释放掉(获取到了几次锁,就应该释放几次),否则该线程仍会持有该锁.

### 应用场景:

#### 场景1：如果发现该操作已经在执行中则不再执行（有状态执行）

a、用在定时任务时，如果任务执行时间可能超过下次计划执行时间，确保该有状态任务只有一个正在执行，忽略重复触发。  
b、用在界面交互时点击执行较长时间请求操作时，防止多次点击导致后台重复执行（忽略重复触发）。

以上两种情况多用于进行非重要任务防止重复执行，（如：清除无用临时文件，检查某些资源的可用性，数据备份操作等）

可通过acquire方法

#### 场景2：如果发现该操作已经在执行，等待一个一个执行（同步执行，类似synchronized）

这种比较常见大家也都在用，主要是防止资源使用冲突，保证同一时间内只有一个操作可以使用该资源。

这种情况主要用于对资源的争抢（如：文件操作，同步消息发送，有状态的操作等）

可使用acquire方法

#### 场景3：如果发现该操作已经在执行，则尝试等待一段时间，等待超时则不执行（尝试等待执行）

这种其实属于场景2的改进，等待获得锁的操作有一个时间的限制，如果超时则放弃执行。  
用来防止由于资源处理不当长时间占用导致死锁情况（大家都在等待资源，导致线程队列溢出）。

可使用acquire方法设置等待时间

#### 场景4：如果发现该操作已经在执行，等待执行。这时可中断正在进行的操作立刻释放锁继续下一操作。

synchronized与Lock在默认情况下是不会响应中断(interrupt)操作，会继续执行完。curator提供了可撤销锁来解决此问题。（场景2的另一种改进，没有超时，只能等待中断或执行完毕）

这种情况主要用于取消某些操作对资源的占用。如：（取消正在同步运行的操作，来防止不正常操作长时间占用造成的阻塞）

可设置锁类型为可撤销

interProcessMutex.makeRevocable(new RevocationListener<InterProcessMutex>() {

@Override

public void revocationRequested(InterProcessMutex forLock) {

// TODO 撤销锁时会调用该监听

}

});

通过Revoker.attemptRevoke(client, "/test");来撤销锁，撤销时会调用revocationRequested监听。

## 不可重入锁Shared Lock

这个锁和上面的相比，就是少了Reentrant的功能，也就意味着它不能在同一个线程中重入。  
这个类是InterProcessSemaphoreMutex。  
使用方法和上面的类似。

InterProcessSemaphoreMutex interProcessSemaphoreMutex=new InterProcessSemaphoreMutex(client, "/test");

if(interProcessSemaphoreMutex.acquire(60, TimeUnit.SECONDS)){

System.out.println("获取锁成功");

}

if(interProcessSemaphoreMutex.acquire(60, TimeUnit.SECONDS)){

System.out.println("再次获取锁成功");

}

interProcessSemaphoreMutex.release();

运行结果:统一客户端下,第二次获取锁失败,无法重复获取锁

## 可重入读写锁Shared Reentrant Read Write Lock

类似JDK的ReentrantReadWriteLock.  
一个读写锁管理一对相关的锁。一个负责读操作，另外一个负责写操作。读操作在写锁没被使用时可同时由多个进程使用，而写锁使用时不允许读 (阻塞)。  
此锁是可重入的。一个拥有写锁的线程可重入读锁，但是读锁却不能进入写锁。  
这也意味着写锁可以降级成读锁，比如请求写锁 —->读锁 ——>释放写锁。从读锁升级成写锁是不成的。

主要由两个类实现：

* InterProcessReadWriteLock
* InterProcessLock

使用时首先创建一个InterProcessReadWriteLock实例，然后再根据你的需求得到读锁或者写锁，读写锁的类型是InterProcessLock。

InterProcessReadWriteLock lock=new InterProcessReadWriteLock(client, "/test");

InterProcessMutex writeLock = lock.writeLock();

InterProcessMutex readLock = lock.readLock();

if(writeLock.acquire(60, TimeUnit.SECONDS)){

System.out.println("get writeLock ");

}

if(readLock.acquire(60, TimeUnit.SECONDS)){

System.out.println("get readLock ");

}

writeLock.release();

readLock.release();

运行结果:先获得写锁成功,在写锁下获取读锁成功.若先获得读锁,在读锁下获得写锁,是无法获取到的

### 应用场景:

1）多个读者可以同时进行读  
2）写者必须互斥（只允许一个写者写，也不能读者写者同时进行）  
3）写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）

## 信号量Shared Semaphore

    一个计数的信号量类似JDK的Semaphore。 JDK中Semaphore维护的一组许可(permits)，而Cubator中称之为租约(Lease)。

       有两种方式可以决定semaphore的最大租约数。第一种方式是有用户给定的path决定。第二种方式使用SharedCountReader类。

       如果不使用SharedCountReader, 没有内部代码检查进程是否假定有10个租约而进程B假定有20个租约。所以所有的实例必须使用相同的numberOfLeases值.

       这次调用acquire会返回一个租约对象。客户端必须在finally中close这些租约对象，否则这些租约会丢失掉。但是，但是，如果客户端session由于某种原因比如crash丢掉，那么这些客户端持有的租约会自动close，这样其它客户端可以继续使用这些租约。

InterProcessSemaphoreV2 interProcessSemaphoreV2=new InterProcessSemaphoreV2(client, "/test", 10);

Collection<Lease> leases=interProcessSemaphoreV2.acquire(5);

if(leases!=null){

System.out.println("获取租约5个成功");

interProcessSemaphoreV2.returnAll(leases);

}else{

System.out.println("获取租约5个失败");

}

Collection<Lease> leases2=interProcessSemaphoreV2.acquire(8);

if(leases2!=null){

System.out.println("获取租约8个成功");

interProcessSemaphoreV2.returnAll(leases2);

}else{

System.out.println("获取租约8个失败");

}

Collection<Lease> lease3=interProcessSemaphoreV2.acquire(13, 10, TimeUnit.SECONDS);

if(lease3!=null){

System.out.println("获取租约13个成功");

}else{

System.out.println("获取租约13个失败");

}

interProcessSemaphoreV2.returnAll(lease3);

运行结果：生成租约10个，第一次获取租约5个成功，剩余5个租约，然后释放租约，剩余10个租约；第二次获取8个租约成功，剩余两个租约，然后释放租约；最后获取13个租约，因为租约数不足，所以获取锁失败。

#### 应用场景

比如说数据库允许10个连接,可使用信号量锁,租约为10,超过10个线程请求锁时,不会获取到锁.若采用可重入锁或其他锁,会导致每次都是获取了一个数据库连接后,其他线程都不会再次获得锁.

上面说讲的锁都是公平锁(fair)。从ZooKeeper的角度看，每个客户端都按照请求的顺序获得锁。相当公平。

## 多锁对象 Multi Shared Lock

   Multi Shared Lock是一个锁的容器。当调用acquire，所有的锁都会被acquire，如果请求失败，所有的锁都会被release。同样调用release时所有的锁都被release(失败被忽略)。  
基本上，它就是组锁的代表，在它上面的请求释放操作都会传递给它包含的所有的锁。

InterProcessLock lock1 = new InterProcessMutex(client, "/test"); // 可重入锁

InterProcessLock lock2 = new InterProcessSemaphoreMutex(client, "/test"); // 不可重入锁

InterProcessMultiLock lock=new InterProcessMultiLock(Arrays.asList(lock1,lock2));

if(lock.acquire(10, TimeUnit.SECONDS)){

System.out.println(lock1.isAcquiredInThisProcess());

System.out.println(lock2.isAcquiredInThisProcess());

}

if(lock.acquire(10, TimeUnit.SECONDS)){

System.out.println(lock1.isAcquiredInThisProcess());

System.out.println(lock2.isAcquiredInThisProcess());

}

lock.release();

运行结果：生成多种锁，将生成的锁放入多锁中，继续对这个多锁对象进行操作；第一次获取锁成功，第二次获取锁时因为含有不可重入锁，所以第二次获取锁失败，最后统一释放掉锁。