1.Lock体系

1.1Lock简介

Lock：锁是用来控制多个线程访问共享资源的方式，一般来说能够防止多个线程访问共享资源。在Lock接口出现之前是靠synchronized关键字来实现锁的功能的，而JDK5之后，并发包中增加了lock接口，它提供了与
synchronized一样的锁功能。
lock和sychronized特点：
相比sychronized而言虽然失去了synchronized关键字隐式的加锁解锁得便捷性，但是却拥有了锁获取和释放得可操作性，可中断的获取锁以及超时获取锁等多种synchronized关键字锁具备的同步特性。通常使用显示使用lock的形式如下：

lock的标准使用形式

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

需要注意的是：lock必须使用ubLock()方法释放锁，因为此在final块中释放了锁

1.1.2 Lock 接口API

下面来看看Lock接口中定义了哪些方法

```
1. void lock(); // 获取锁
2. void lockInterruptibly() throws InterruptedException; // 获取锁的过程能够响应中断
3. boolean tryLock(); // 非阻塞式响应中断能立即返回，获取锁返回true反之为false
4. boolean tryLock(long time,TimeUnit unit);// 超时获取锁，在超时内或未中断的情况下能获取锁
5. Condition newCondition(); // 获取与lock绑定的等待通知组件，当前线程必须先获得了锁才能等待，等待会释放锁，再次获取到锁才能从等待中返回。
```

下面来看看Lock接口中定义了哪些方法

```
1. void lock(); // 获取锁
2. void lockInterruptibly() throws InterruptedException; // 获取锁的过程能够响应中断
3. boolean tryLock(); // 非阻塞式响应中断能立即返回，获取锁返回true反之为false
4. boolean tryLock(long time,TimeUnit unit);// 超时获取锁，在超时内或未中断的情况下能获取锁
5. Condition newCondition(); // 获取与lock绑定的等待通知组件，当前线程必须先获得了锁才能等待，等待会释放锁，再次获取到锁才能从等待中返回。
```

LocK 接口实现的子类

`ReadWiteLock`

`ReentrantLock`

`ReentrantReadWitrLock`

`StampedLock`

基本上所有方法的实现都是调用了其静态内存类Sync中的方法，而Sync类继承了AbstractQueuedSynchronizer(AQS)
则ReentrantLock关键的核心在于队列同步器AbstractQueuedSynchronizer(AQS)。（同步器的理解）

1.1.3初识AQS

同步器是用来构建和其他同步组件的基本框架，它的实现主要依赖一个int成员的变量来表示当前状态以及通过一个FIFO队列后成的等待队列。他的子类必须覆写AQS的几个protected修饰的用来改变同步状态的方法，其他方法主要是实现了排队和阻塞机制。状态更新使用getState，setState，以及compareAndSetState这三个方法。

子类被推荐定义为自定义同步组件的静态内部类（Sync）,同步器自身没有实现任何同步接口，它仅仅是定义了若干同步状 态的获取和释放方法来供自定义同步组件的使用，同步器既支持独占式获取同步状态，也可以支持共享式获取同步 状态，这样就可以方便的实现不同类型的同步组件。

同步器是实现锁（也可以是任意同步组件）的关键，在锁的实现中聚合同步器，利用同步器实现锁的语义。可以这 样理解二者的关系：(lock)锁是面向使用者，它定义了使用者与锁交互的接口，隐藏了实现细节；(AQS)同步器是面向锁的实现 者，它简化了锁的实现方式，屏蔽了同步状态的管理，线程的排队，等待和唤醒等底层操作。锁和同步器很好的隔离了使用者和实现者所需关注的领域。

1.1.4AQS的模板方法设计模式

AQS的设计是使用模板方法设计模式，它将一些方法开放给子类进行重写，而同步器给同步器组件所提供模板方法又会重新调用被子类所覆写的方法。举个列子，AQS中需要重写的方法tryAcquire:

```
protected boolean tryAcquire(int arg) {
throw new UnsupportedOperationException();
}
ReentrantLock中NonfairSync（继承AQS）

protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}
```

AQS中的模板方法acqire();会调用tryAcquire方法，而此时当继承AQS的NonfairSync（不公平锁）调用模板方法acquire时就会调用已经被NonfairSync重写的tryAcquire方法。这就是使用AQS的方式。

在弄懂这点后会lock的实现理解有很大的提升。可以归纳总结为这么几 点：

```
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

1.同步组件（这里不仅仅指锁，还包括CountDownLatch等）的实现依赖于同步器AQS。在同步组件实现中，使用 AQS的方式被推荐定义继承AQS的静态内存类；

例如：`ReentrantLock中 Sync类继承AQS`
但`ReentrantLock中有FairSync共公锁，NonfairSync非公平锁。所以又继承Sync类，来覆写AQS中的protected方法tryLock,tryRealse()` 等

在当其调用AQS 的acquire () 方法时就会调用 两个FairSync NonfairSync 所覆写的方法。达到自定义的目的。
实现其排队阻塞机制（即实现了同步的语义），并没有具体去实现AQS中的实际底层操作方法。

2 AQS采用模板方法进行设计，AQS的protected修饰的方法需要由继承AQS的子类进行重写实现，当调用AQS的子 类的方法时就会调用被重写的方法；
3. AQS负责同步状态的管理，线程的排队，等待和唤醒这些底层操作，而Lock等同步组件主要专注于实现同步语义；
4.在重写AQS的方式时，使用AQS提供的
getState(),setState(),compareAndSetState()方法进行修改同步状态

AQS可重写的方法如下图（protected方法）

方法名称	描　　述
protected boolean tryAcquire(int arg)	独占式获取同步状态，实现该方法需要查询当前状态并判断同步状态是否符合预期，然后再进行 CAS 设置同步状态
protected boolean tryRelease(int arg)	独占式释放同步状态，等待获取同步状态的线程将有机会获取同步状态

（续）

方法名称	描　　述
protected int tryAcquireShared(int arg)	共享式获取同步状态，返回大于等于 0 的值，表示获取成功，反之，获取失败
protected boolean tryReleaseShared(int arg)	共享式释放同步状态
protected boolean isHeldExclusively()	当前同步器是否在独占模式下被线程占用，一般该方法表示是否被当前线程所独占

方法名称	描　　述
protected boolean tryAcquire(int arg)	独占式获取同步状态，实现该方法需要查询当前状态并判断同步状态是否符合预期，然后再进行 CAS 设置同步状态
protected boolean tryRelease(int arg)	独占式释放同步状态，等待获取同步状态的线程将有机会获取同步状态

（续）

方法名称	描　　述
protected int tryAcquireShared(int arg)	共享式获取同步状态，返回大于等于 0 的值，表示获取成功，反之，获取失败
protected boolean tryReleaseShared(int arg)	共享式释放同步状态
protected boolean isHeldExclusively()	当前同步器是否在独占模式下被线程占用，一般该方法表示是否被当前线程所独占

在实现同步组件时AQS提供的模板方法如下图：（底层实际操作）

方法名称	描　述
void acquire(int arg)	独占式获取同步状态，如果当前线程获取同步状态成功，则由该方法返回，否则，将会进入同步队列等待，该方法将会调用重写的 tryAcquire(int arg) 方法
void acquireInterruptibly(int arg)	与 acquire(int arg) 相同，但是该方法响应中断，当前线程未获取到同步状态而进入同步队列中，如果当前线程被中断，则该方法会抛出 InterruptedException 并返回
boolean tryAcquireNanos(int arg, long nanos)	在 acquireInterruptibly(int arg) 基础上增加了超时限制，如果当前线程在超时时间内没有获取到同步状态，那么将会返回 false，如果获取到了返回 true
void acquireShared(int arg)	共享式的获取同步状态，如果当前线程未获取到同步状态，将会进入同步队列等待，与独占式获取的主要区别是在同一时刻可以有多个线程获取到同步状态
void acquireSharedInterruptibly(int arg)	与 acquireShared(int arg) 相同，该方法响应中断
boolean tryAcquireSharedNanos(int arg, long nanos)	在 acquireSharedInterruptibly(int arg) 基础上增加了超时限制
boolean release(int arg)	独占式的释放同步状态，该方法会在释放同步状态之后，将同步队列中第一个节点包含的线程唤醒
boolean releaseShared(int arg)	共享式的释放同步状态
Collection<Thread> getQueuedThreads()	获取等待在同步队列上的线程集合

方法名称	描　述
void acquire(int arg)	独占式获取同步状态，如果当前线程获取同步状态成功，则由该方法返回，否则，将会进入同步队列等待，该方法将会调用重写的 tryAcquire(int arg) 方法
void acquireInterruptibly(int arg)	与 acquire(int arg) 相同，但是该方法响应中断，当前线程未获取到同步状态而进入同步队列中，如果当前线程被中断，则该方法会抛出 InterruptedException 并返回
boolean tryAcquireNanos(int arg, long nanos)	在 acquireInterruptibly(int arg) 基础上增加了超时限制，如果当前线程在超时时间内没有获取到同步状态，那么将会返回 false，如果获取到了返回 true
void acquireShared(int arg)	共享式的获取同步状态，如果当前线程未获取到同步状态，将会进入同步队列等待，与独占式获取的主要区别是在同一时刻可以有多个线程获取到同步状态
void acquireSharedInterruptibly(int arg)	与 acquireShared(int arg) 相同，该方法响应中断
boolean tryAcquireSharedNanos(int arg, long nanos)	在 acquireSharedInterruptibly(int arg) 基础上增加了超时限制
boolean release(int arg)	独占式的释放同步状态，该方法会在释放同步状态之后，将同步队列中第一个节点包含的线程唤醒
boolean releaseShared(int arg)	共享式的释放同步状态
Collection<Thread> getQueuedThreads()	获取等待在同步队列上的线程集合

AQS提供的模板方法可分为3类

1.独占式获取与释放同步状态；

2.共享式获取与释放同步状态；

3.查询同步队列中等待线程情况；

同步组件通过过AQS提供的模板方法去实现自己的同步语义。

例如自己实现的简易Lock锁 ： LockSystem.Test 链接：
file:///D:/IDEA%20project/LockSystem/src/LockTest

1.实现同步组件时推荐定义继承AQS的静态内存类，**并重写需要的protected修饰的方法；**
2.同步组件语句的实现依赖于AQS的模板方法，而AQS模板方法又依赖于AQS子类所覆写的方法。

总的来说，同步组件通过重写AQS的方法**来实现自己想要表达的同步语义**，而AQS只需要**同步组件表达式的true和false**即可，**AQS会针对true和false不同情况下做不同的处理。**

因为AQS中acquire方法用tryAcquire方法作为if判断的语句块中的第一个判断条件，假如判断为真则CAS操作成功，直接退出。线程获取到锁。

```
AQS-acquire
public final void acquire(int arg) {
if (!tryAcquire(arg) &&
acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
selfInterrupt();
}
```