J.U.C =Java.util.concurrent

> 更详细的文档，可以从2019年的并发编程的课后笔记中获得，那个是最最最详细的。

Lock (Synchronized)

在Lock接口出现之前，Java中的应用程序对于多线程的并发安全处理只能基于synchronized关键字来解决。但是synchronized在有些场景中会存在一些短板，也就是它并不适合于所有的并发场景。但是在Java5以后，Lock的出现可以解决synchronized在某些场景中的短板，它比synchronized更加灵活。

ReentrantLock（重入锁）

ReentrantReadWriteLock(重入读写锁)

StampedLock

读多写少的情况下.

读和读不互斥

读和写互斥

写和写互斥

思考锁的实现（设计思维）

{互斥}

- 锁的互斥特性 -> 共享资源（）-> 标记 （0 无锁， 1代表有锁）
- 没有抢占到锁的线程？-> 释放CPU资源 , [等待 -> 唤醒]
- 等待的线程怎么存储？ -> 数据结构去存储一些列等待中的线程，FIFO (等待队列)
- 公平和非公平（能否插队）
- 重入的特性（识别是否是同一个人？ThreadID）

{技术方案}

- volatile state =0 （无锁） ， 1代表是持有锁， >1代表重入
- wait/notify | condition 需要唤醒指定线程。[LockSupport.park(); ->unpark(thread)] unsafe类中提供的一个方法
- 双向链表
- 逻辑层面去实现
- 在某一个地方存储当前获得锁的线程的ID，判断下次抢占锁的线程是否为同一个。

```
private volatile int state;  //互斥资源 0
```

```
final void lock() {
    //抢占互斥资源（）
    if(cas()){ //线程并行。
        //有多个线程进入到这段代码？多个线程抢占到同一把锁.
        ...
    }
    //不管当前队列是否有人排队的？  临界点的情况.
    if (compareAndSetState(0, 1)) //乐观锁（ true / false） |  只有一个线程能够进入.
        //能够进入到这个方法 ，  表示无锁状态
        setExclusiveOwnerThread(Thread.currentThread());//保存当前的线程
    else
        acquire(1);
}
```

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

- ! tryAcquire(arg)
- addWaiter 将未获得锁的线程加入到队列
- acquireQueued(); 去抢占锁或者阻塞.

```
final boolean nonfairTryAcquire(int acquires) {
    //获得当前的线程
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) { //无锁
        if (compareAndSetState(0, acquires)) { //cas
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    //判断？重入
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires; //增加state的值
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc); //并不需要通过cas
        return true;
    }
    return false;
}
```