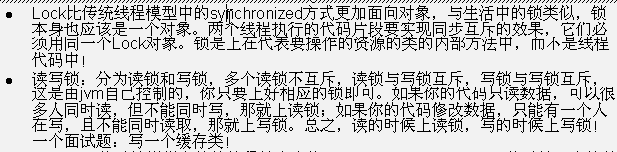
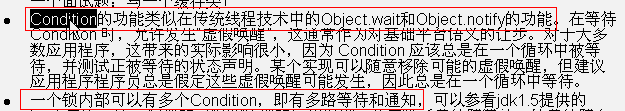
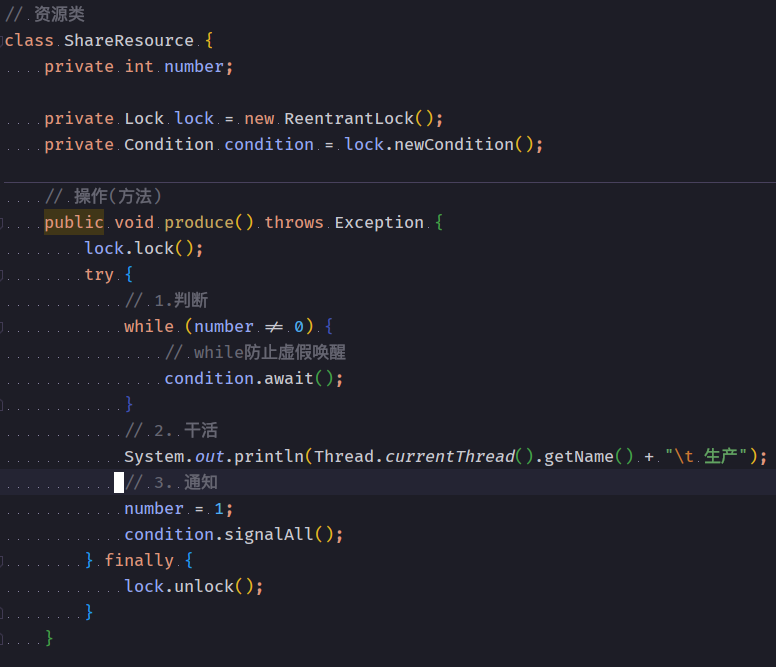
1. 锁java.util.concurrent.locks包



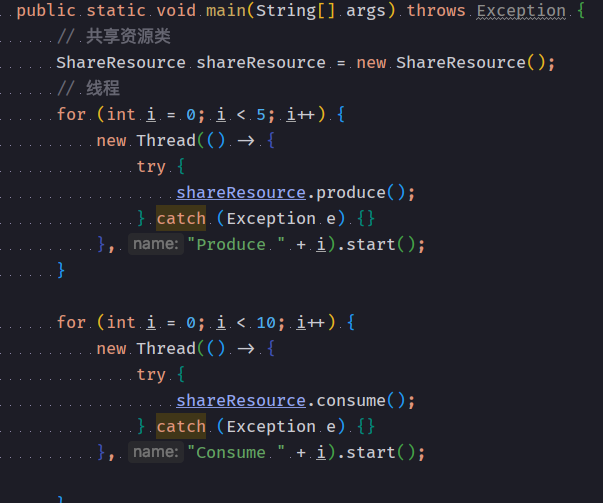
1. java.util.concurrent.locks.ReentrantLock，可重入锁
   1. new ReentrantLock(true). 公平锁，根据锁的请求顺序决定锁的获取顺序，满足FIFO。获取锁前先检查等待队列是否为空，若为空则直接获得锁，若不为空则加入等待队列队尾等待调度。
   2. new ReentrantLock(false). 非公平锁，新来的线程不用检查等待队列直接尝试获得锁，若此时正好有线程释放锁，则新来的线程可以直接获得锁；若获得锁失败，则加入等待队列队尾等待调度(后续等待调度与公平锁行为一致)。非公平锁会由于『频繁插队』导致『饥饿』问题，但吞吐量相对于公平锁也大大提高了
   3. lock.**lock**(). 获得锁。与synchronized不同，需要主动调用unlock释放锁。Synchronized遇到异常JVM会帮助释放锁，但是lock.lock()不会，注意在**finally**里面释放
   4. lock.tryLock(). 尝试获得锁。与lock不同，tryLock尝试获得锁，通过返回值true/false自行决定程序行为
   5. lock.tryLock(long time, TimeUnit unit)，尝试获得锁并指定等待时间，若指定等待时间内没有获得锁返回false
   6. lock.lockInterruptibly()，锁等待期间可以响应线程的interrupt()抛出异常InterruptedException
   7. lock.**unlock**(). 释放锁
2. Lock与synchronized区别
   1. 原始构成
      1. synchronized是Java关键字，属于JVM层面，底层通过monitorenter和monitorexit实现
      2. lock是具体类(java.util.concurrent.locks.Lock)，是API层面的锁
   2. 使用方法
      1. Synchronized不需要手动释放锁，当synchronized代码执行完后系统会自动让线程释放对锁的占用
      2. Lock需要用户手动释放锁，通常在try finally块中，需要lock()和unlock()方法配合try/finally语句块完成
   3. 等待是否可中断
      1. Synchronized不可中断，除非抛出异常或者正常执行完成
      2. Lock可中断。可以设置超时方法tryLock(time, unit)；也可以lockInterruptibly()，调用interrupt()方法中断
   4. 加锁是否公平
      1. Synchronized非公平锁
      2. Lock可以通过构造方法确定是公平还是非公平锁
   5. 绑定多个Condition
      1. Synchronized没有
      2. Lock用来实现分组唤醒需要被唤醒的线程们，可以通过多个Condition精确唤醒，而不是synchronized要么随机唤醒一个，要么全部唤醒
3. java.util.concurrent.locks中Condition接口



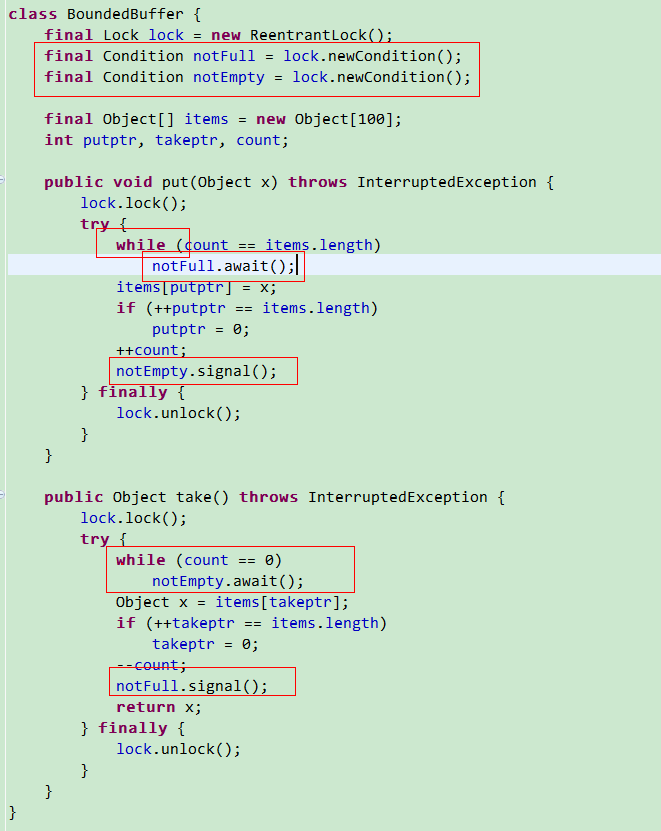
1. 线程锁模式
   1. 线程 操作(方法) 资源类(共享)
   2. 判断 干活 通知
   3. 防止虚假唤醒(while判断)



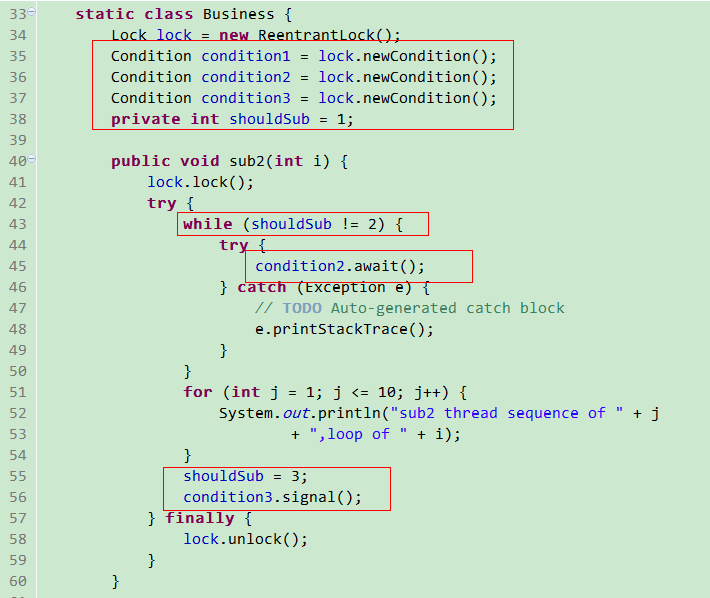


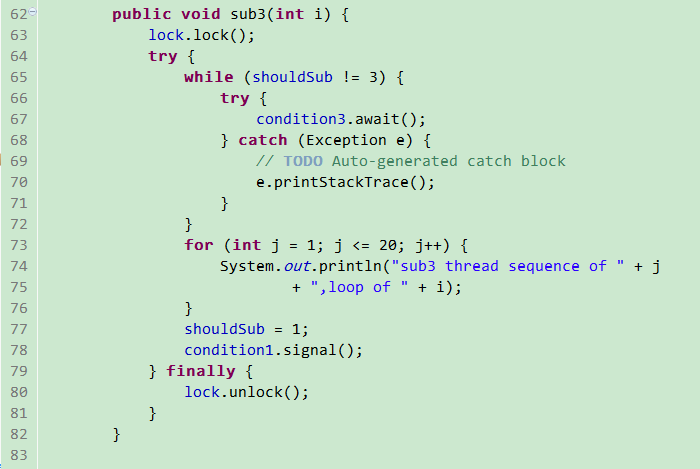


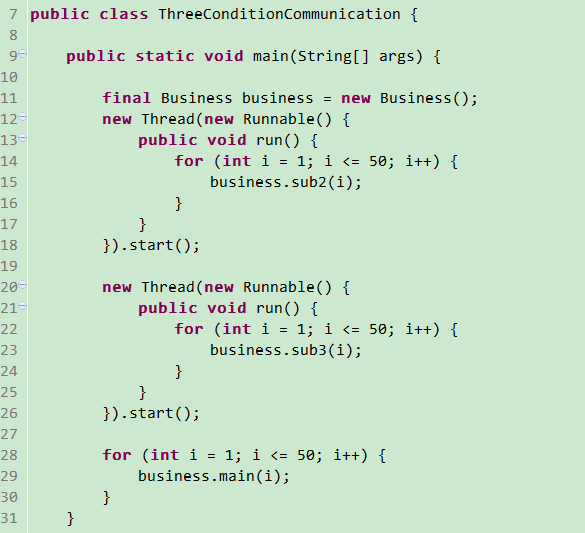
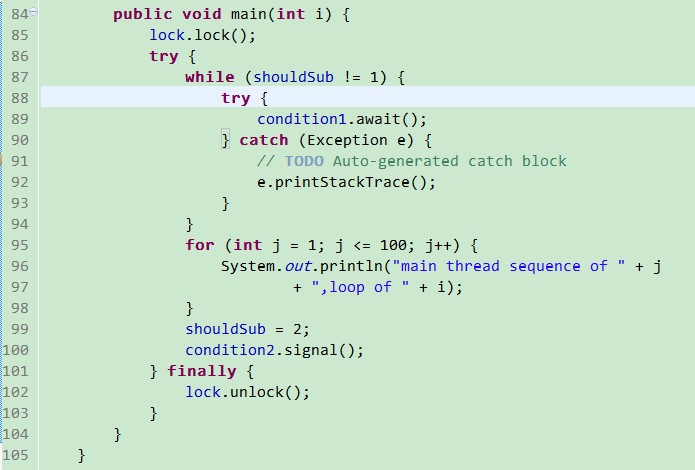
1. 两个线程通信简单实例



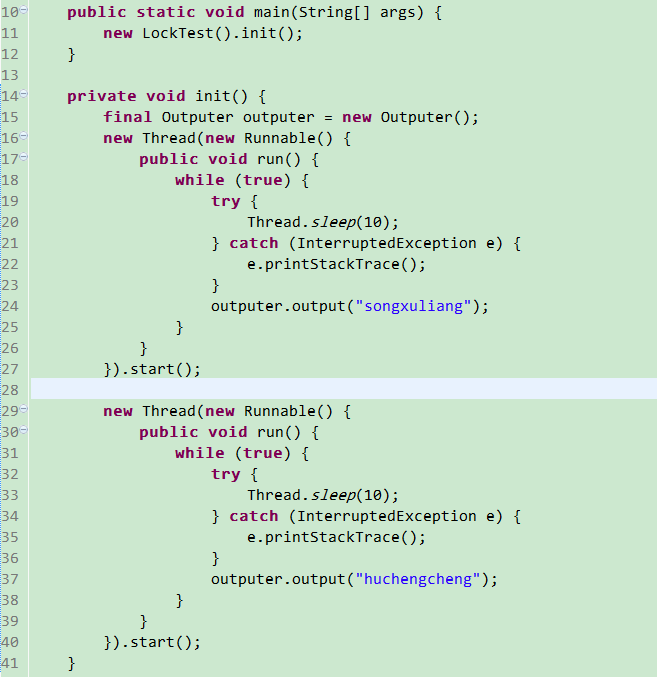
1. 三个线程通信简单实例

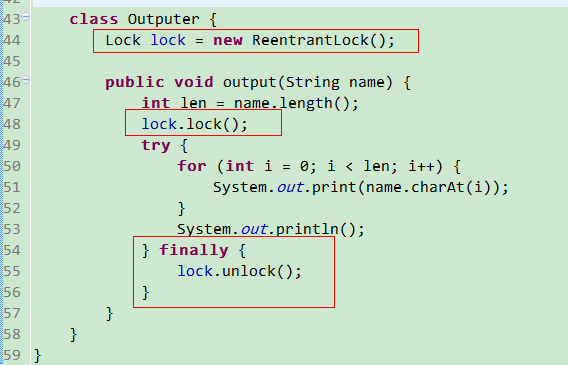






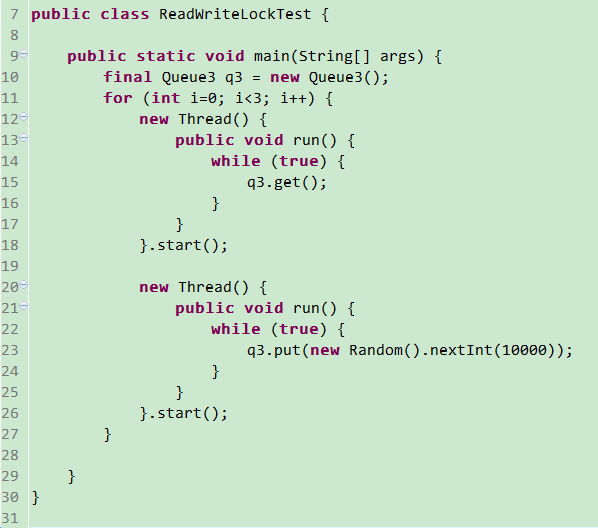
1. 锁的简单实例





1. 读写锁简单实例





1. 简易缓存系统

