**synchronized同步**

001/002同步代码块

003同步方法

004同步静态方法

005、006应用举例：下面两个程序，第一个程序，如果不加synchornize会出现原子性问题。5个线程抢得是桶一个对象的锁，因此会生产竞争，第二个程序每个线程拿到的锁都是不一样的，因此不存在竞争关系,而且变量也不是共享的的，不存在同步问题。

007同步方法和非同步方法可以同时调用

008账户读写案例：只对写方法加锁，读方法不加锁，产生脏读。

009 测试synchornize的可重入性，一个同步方法调用另一个同步方法

010子类同步方法可以调用父类的同步方法，这个也是可重入性的一种

011 线程抛出异常后锁被释放举例

------------------------------------------------------------------------------------------

016 采用细粒度的锁（尽量缩小锁的范围）

017 锁对象的引用变了的，就是另一个锁

018 不要以字符串为锁对象

**Volatile关键字---轻量级锁**

012 volatile可见性举例,JMM内存模型：

<http://www.cnblogs.com/nexiyi/p/java_memory_model_and_thread.html>

013 volatile不保证原子性列子：起多个线程进行累加，出现重复写

014 synchornized实现上个例子同步

015 使用原子类保证013例子同步---要扩展CAS内容

**wait、notify、CountDownLatch：**

019 实现一个容器，提供两个方法，add，size，写两个线程，线程1添加10个元素到容器中，线程2实现监控元素的个数，当个数到5个时，线程2给出提示，线程2结束,线程1继续运行

Mycontainer1:对list没有加volatile，没有可见性，程序无法达到目的

Mycontainer2：加入volatile，程序能实现功能但是t2在死循环中，占用cpu

Mycontainer3：使用notify 和 wait，notify不释放锁举例

Mycontainer4：使用notify 和 wait，线程1唤醒线程2之后要主动休眠释放锁，线程2处理完后再来释放1继续进行。

Mycontainer5：使用CountDownLatch来解决

**lock：**

020 reentrantlock的用法。

Reentrantlock1: synchornize用法复习

Reentrantlock2:lock的基本用法，使用 ReentrantLock实现上面相同功能，注意lock需要手动释放，使用syn锁定的话如果遇到异常，jvm会自动释放锁，但是lock必须手动释放锁，因此经常在finally中进行锁的释放。

Reentrantlock3：trylock的使用,常用的是指定时间去获取锁，如果时间到了，根据返回值我们写对应的逻辑代码

Reentrantlock4：lockInterruptibly可以从外部主动打断线程

ReentrantLock5：可以被指定为公平锁------谁先来谁先拿

**生产者消费者：**

021生产者消费者的两种实现方式

Mycontainer1:两个注意点：

用whlie判断不用if,

比如当生产者多个线程在阻塞的时候，消费者消费了之后唤醒所有线程，这个时候如果是if的话就会直接继续往下（因为if这个条件在进去之前已经判断过了，不会再判断），于是所有生产者都会认为应该生产，很容易导致容量溢出，如果用while的话在出wihle之前还会再判断一下，这是如果已经有生产者抢到了锁就会先生产一个，然后这个没抢到的再判断的时候已经满了，就进入阻塞，不去生产。

用notifyall不用notify，比如生产者生产完后如果用notify本来的目的是要唤醒消费者消费，但是这时候有可能唤醒的生产者，生产者被唤醒后发现已经满了，进入阻塞，这个时候整个都被阻塞了。因此用notifyall可以保证一定有消费者别唤醒。

Mycontainer2:使用lock和conditio来做，可以指定被唤醒对象。

**ThreadLocal：**

022 线程局部变量例子

**单例模式：**

023 介绍内部类方式实现情况

线程安全常用的三种见：

<http://www.cnblogs.com/xudong-bupt/p/3433643.html>

**抢票问题：**

024 抢票并发问题的解决

TicketSeller1：出现卖重（list的remove不是原子性的，可能在remove过程中其他线程也remove同一个元素）、卖超问题（判读和操作分离，不是原子性）

TicketSeller2：使用vector，可以解决卖重问题，但是卖超问题依旧

TicketSeller3：使用syn将判断和操作锁住，可以解决，但是效率低

TicketSeller4：使用并发容器concurentQueue,虽然判断和操作也分离了，但是这里的操作没有对queue进行修改。

**并发容器：**

**线程池：**