# Linux的补充

linux的组成：

Linux可以简单理解为很多库来构成的：

内核：直接与硬件交互；

运行在系统上的软件是不能直接与内核进行交互的，而是通过其他库，这些库再来调用内核;

eg:glibc里面就包含了pthread\_create()方法，java就是通过调用这个函数，他再去调用内核函数；

重编译glibc：

1. yum search java |grep -i --color jdk

yum命令查看java版本

1. yum install -y java-1.8.0-openjdk

安装完成可以使用java命令

1. yum install -y java-1.8.0-openjdk-devel.x86\_64

使用javac命令

1. 安装gcc：yum install -y gcc

用来编译C文件的，因为glibc就是c文件写的

1. 将需要安装的glibc复制到linux，eg 目录为/home/glibc
2. 解压到当前目录：

tar -zxvf glibc-2.19.tar.gz

1. 修改/home/glibc/glibc-2.19/nptl下的pthread\_mutex\_lock.c添加：

fprintf(stderr,"current mutex pid=%lu\n",pthread\_self());

//打印调用当前方法的线程ID

1. 创建一个编译后的目录eg glibc\_out，并cd到该目录
2. 编译：

../configure --prefix=/usr --disable-profile --enable-add-ons --with-headers=/usr/include --with-binutils=/usr/bin

--prefix=/usr 编译后的位置，覆盖系统的版本位置/usr/lib

1. make install

java调用执行本地方法：

1. 先编译JAVA文件：

javac SyncDemo.java

1. 将java文件编译为.h文件

javah SyncDemo

1. 把这个**GetOSThraedIdNative**.c编译成为一个动态链接库，这样在java代码里会被laod到内存：

生成.so文件

gcc -fPIC -I /usr/lib/jvm/java-1.8.0-openjdk/include -I /usr/lib/jvm/java-1.8.0-openjdk/include/linux -shared -o lib**GetOSThraedIdNative**.so **GetOSThraedIdNative**.c

libLubanThreadNative 此处的名称与System.*loadLibrary*的参数加lib+名字；

System.*loadLibrary*(**"GetOSThraedIdNative"**);

1. 做完这一系列事情之后需要把这个.so文件加入到path，这样java才能load到

export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:{libLubanThreadNative.so}所在的路径

1. 使用java命令执行java文件

java code：

**public class** SyncDemo {  
 Object o = **new** Object();  
 **static** {  
 //本地方法C文件变异成.so文件对应的文件名  
 System.*loadLibrary*(**"GetOSThraedIdNative"**);  
 }  
 //执行结果为 t1 t2两个线程轮流获取锁 依次执行  
 **public static void** main(String[] args) {  
 SyncDemo sync = **new** SyncDemo();  
 sync.start();  
 }  
 //因为这两个匿名内部类 所以字节码文件是三个  
 **public void** start(){  
 Thread t1 = **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **while** (**true**) {  
 **try** {  
 TimeUnit.*MILLISECONDS*.sleep(**500**);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 sync();  
 }  
 }  
 });  
 Thread t2 = **new** Thread(() -> {  
 **while** (**true**) {  
 **try** {  
 TimeUnit.*MILLISECONDS*.sleep(**500**);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 sync();  
 }  
 });  
 t1.setName(**"t1"**);  
 t2.setName(**"t2"**);  
 t1.start();  
 t2.start();  
 }  
   
 //因为无法通过java代码获取OS层面的线程ID进行比较 所以增加一个本地方法  
 **public native void** getOSThraedId();  
 **public void** sync(){  
 **synchronized** (o){  
 getOSThraedId();  
 System.*out*.println(Thread.*currentThread*().getName());  
 }  
 }  
}

C code:

**#include**<pthread.h>  
**#include**<stdio.h>  
**#include**<stdlib.h>  
*//调用这个本地方法的JAVA类类名编译成的.h文件***#include "SyncDemo.h"**JNIEXPORT **void** JNICALL Java\_SyncDemo\_getOSThraedId(JNIEnv \*env, jobject c1){  
 printf(**"current java native tid:%lu-----\n"**,pthread\_self());  
 usleep(**700**);  
}

# 并发编程模型

## 进程和线程的区别和联系

一个程序至少有一个进程,一个进程至少有一个线程。

1. 根本区别：进程是操作系统资源分配的基本单位，而线程是任务调度和执行的基本单位;
2. 开销方面：每个进程都有独立的代码和数据空间（程序上下文），程序之间的切换会有较大的开销；线程可以看做轻量级的进程，同一类线程共享代码和数据空间，每个线程都有自己独立的运行栈和程序计数器（PC），线程之间切换的开销小。
3. 所处环境：在操作系统中能同时运行多个进程（程序）；而在同一个进程（程序）中有多个线程同时执行（通过CPU调度，在每个时间片中只有一个线程执行）
4. 内存分配方面：系统在运行的时候会为每个进程分配不同的内存空间, 进程在执行过程中拥有独立的内存单元；

而对线程而言，除了CPU外，系统不会为线程分配内存（线程所使用的资源来自其所属进程的资源），线程组之间只能共享资源。

1. 包含关系：没有线程的进程可以看做是单线程的，如果一个进程内有多个线程，则执行过程不是一条线的，而是多条线（线程）共同完成的；线程是进程的一部分，所以线程也被称为轻权进程或者轻量级进程。

## java线程有OS(Linux)线程的关系

pthraed\_create()方法的四个参数说明：

|  |  |  |
| --- | --- | --- |
| pthread\_t \*thread | 传出参数，调用之后会传出被创建线程的id | 定义 pthread\_t pid; 继而取地址获取&pid |
| const pthread\_attr\_t \*attr | 线程属性，关于线程属性是linux的知识 | 在学习pthread\_create函数的时候一般传NULL，保持默认属性 |
| void \*(\*start\_routine) (void \*) | 线程的启动后的主体函数，相当于java当中的run方法 | 需要你定义一个函数，然后传函数名即可 |
| void \*arg | 前面参数主体函数的参数 | 如果没有可以传NULL |

通过C++来调用Linux的一个线程：

**#include** <pthread.h>*//头文件***#include** <stdio.h>  
pthread\_t pid;*//定义一个变量，接受创建线程后的线程id  
//定义线程的主体函数***void**\* thread\_entity(**void**\* arg){  
 printf(**"i am new Thread!"**);  
}  
*//main方法，程序入口，main和java的main一样会产生一个*进程*，继而产生一个main*线程**int** main() {  
 *//调用操作系统的函数*pthread\_create*创建线程，注意四个参数* pthread\_create(&pid,NULL,thread\_entity,NULL);  
 *//usleep是睡眠的意思，那么这里的睡眠是让谁睡眠呢？  
 //为什么需要睡眠？如果不睡眠会出现什么情况*  usleep(**100**);  
 printf(**"main\n"**);  
}

这就说明了系统内核也是支持多线程；

thread.start()方法调用的核心是：

**private native void** start0();

是一个native 本地方法

java调用的是start0(),这个本地方法肯定是一个C或者C++文件

JDK包含的内容：

1. 一系列sun公司提供的java库
2. 本地C文件(作用：1、实现调用操作系统的函数‘、；2、调用JVM的代码hotspot；)
3. C++写的hotspot虚拟机这个项目；

本地方法C文件调用的应该是hotspot虚拟机的代码，而hotspot的c++代码来调用os的函数；

重点 - java调用start方法来开启线程：

java通过start()方法开启一个线程，就是通过hotspot代理最终调用的os的pthread\_create()方法来创建一个新线程；所以可以说JAVA的线程和操作系统OS的线程是一一对应的，即java对线程的操作方法都可以在OS上找到对应的操作；

start()是来调用系统的函数的，所以start()方法不能写实现，而是通过调用对应的run()方法来实现相应的逻辑；

C语言调用java方法

java调用java方法：通过class对象来new 对应的java对象，然后调用它的方法；

C语言调用也是一样的，通过JVM找到对应的class文件，然后new对象来调用方法：

1. 定义变量jclass clas对象；
2. clas = (\*env)->FindClass(env,“Test”); //Test是对于java文件名；
3. 定义一个变量来承接对象jobject obj；jmethod cid对应java的方法；
4. 找到class之后调用cid = (\*env)->GetMethodID(env,cls,”<init>”,”()v”)

这里调用的方法<init>为java的构造方法，()v则说明是无参的；

这里的cid的c是constructor 构造方法的缩写；

1. 然后获取实例化对象：

obj = (\*env)->NewObject(env,clas,cid);

1. 调用方法：

定义对象jmethod rid；

int ret = (\*env)->CallIntMethod(env,obj,”run”,”()v”);

同获取构造函数的方法，第三个参数run是被调用的java方法名，第四个参数为该方法需传入的参数，()v表示传入参数为空；

1. 最终实现了C反调java方法；

重点 - JAVA的Thread调用start()方法是怎么调回run方法的？

1. thread调用start()，start调用的是本地方法start0()；
2. start0()方法调用的是操作系统的pthraed\_create()创建线程，并传入四个参数；
3. pthraed\_create()会调用一个主体函数java\_start，而这个主体函数通过JNI反射来实现调用JAVA的run();

java为什么不通过本地C文件直接调用OS的相关函数：

假如直接调用，则这个C文件会被十分频繁的调用去与OS函数进行交互；所以它调用了hotspot的c++的javaThread，而他去通过对应的方法与OS进行交互；

# 并发偏向锁的存在

## Thraed的API

1. 中断停止线程

线程一直处于运行的状态，比如阻塞(线程阻塞通常是指一个线程在执行过程中暂停，以等待某个条件的触发。)或者while(true);

JVM不推荐直接停止一个线程，因为会导致有些资源来不及释放，当前线程更改的变量来不及还原；一定要让一个线程执行结束；

当前线程一般有三个状态：

1. 正在执行：应该等执行结束终止；
2. 真的阻塞：解阻塞，然后继续；
3. 无限循环：改变循环条件，继续执行；
4. 修改静态变量

**private static** Thread *thread*;  
**private static boolean** *running* = **true**;  
**public static void** main(String[] args) **throws** InterruptedException {  
 *run*();  
 TimeUnit.*SECONDS*.sleep(**1**);  
 *running* = **false**;  
}  
**private static void** run() {  
 *thread* = **new** Thread(() -> {  
 **while** (*running*)  
 System.*out*.println(**"running"**);  
 });  
 *thread*.start();  
}

这样做的弊端，可能单前线程不能立即结束；

并且这样处理也是线程可见性变差；

**while** (*running*){  
 System.*out*.println(**"running"**);  
 **try** {  
 TimeUnit.*MILLISECONDS*.sleep(**1100**);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println(**"sleep over ..."**);  
}

线程休眠结束后的逻辑还是会被执行

## PS：指令重拍

如果while循环内不执行任何操作，这样即使*running改为false*也不会终止线程；只要有执行内容就会终止，原因：

因为JVM会发生指令重拍；

这里证明了指令重拍可以发生在编译阶段；它也可以执行在执行阶段；

指令重排是指JVM在编译Java代码的时候，或者CPU在执行JVM字节码的时候，对现有的指令顺序进行重新排序。

比如：代码int i=0;int k=0;i++;

执行时如果按顺序执行，到i++时，CPU需要去寄存器中重新寻找i的地址，比较麻烦，所以CPU在实际执行过程中会进行优化变成int i=0;i++;int k=0;这样直接可以i++；

分为保守优化和激进优化；

指令重排的目的是为了在不改变程序执行结果的前提下，优化程序的运行效率。需要注意的是，这里所说的不改变执行结果，指的是不改变单线程下的程序执行结果。

然而，指令重排是一把双刃剑，虽然优化了程序的执行效率，但是在某些情况下，会影响到多线程的执行结果。

如果while循环内没有执行内容，会被CPU将指令优化为：

*thread* = **new** Thread(**new** Runnable() {  
 **boolean** temp = *running*;  
 **public void** run() {  
 **while** (!temp){}  
 }  
});

用一个临时变量来承接，不用再频繁去寻找running的地址，但是也造成了temp一直为false而使得当前执行不会结束；

而如果里面有方法的调用，则CPU担心有方法溢出，就不再执行这个优化了；

如果只调用一个i++之类的也不会被终止；

但是如果main()方法中不休眠一定时间，while循环却会终止：

这是因为当主线程开启一个新线程，CPU 80%的概率会优先执行主线程，即先执行*running* = **false**;然后编译到指令重拍后的**boolean** temp = *running = false*;所以新开的线程就终止了；

## interrupt()

其作用是中断此线程（此线程不一定是当前线程，而是指调用该方法的Thread实例所代表的线程），但实际上只是给线程设置一个中断标志，线程仍会继续运行。

如果当前线程在执行过程(使用的是让当前线程休眠一定时间来模拟)中被直接调用interrupt()会抛出异常，并且当前线程不会终止:

java.lang.InterruptedException: sleep interrupted

所以终止线程推荐以下两种方法：

1. 假设当前线程被阻塞，就需要解阻塞，仍然让当前线程执行结束；
2. 处于while(flag)；flag=true循环中，JVM将flag改为false；

即调用*thread*.isInterrupted()属性，默认为true

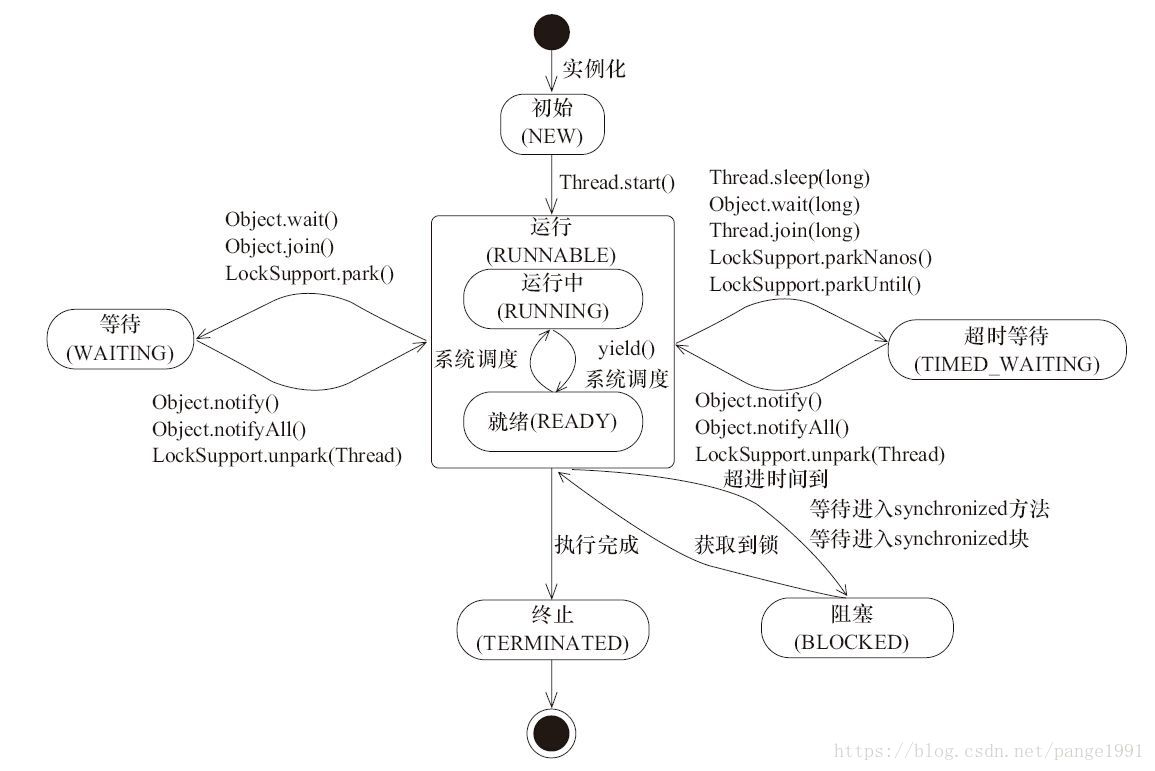
**private static** Thread *thread*;  
**public static void** main(String[] args) **throws** InterruptedException {  
 *run*();  
 TimeUnit.*MILLISECONDS*.sleep(**500**);  
 *thread*.interrupt();  
}  
**public static void** run() {  
 *thread* = **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **while** (!*thread*.isInterrupted()){  
 System.*out*.println(**"running"**);  
 }  
 }  
 });  
 *thread*.start();  
}

1. synchronized关键字

对象的五种状态

无锁、偏向锁、轻量锁、重量级锁和被GC标记状态

线程的五种状态



1. 初始状态

实现Runnable接口和继承Thread可以得到一个线程类，new一个实例出来，线程就进入了初始状态。

1. 2.1. 就绪状态 runnable

就绪状态只是说你资格运行，调度程序没有挑选到你，你就永远是就绪状态。

调用线程的start()方法，此线程进入就绪状态。

当前线程sleep()方法结束，其他线程join()结束，等待用户输入完毕，某个线程拿到对象锁，这些线程也将进入就绪状态。

当前线程时间片用完了，调用当前线程的yield()方法，当前线程进入就绪状态。

锁池里的线程拿到对象锁后，进入就绪状态。

2.2. 运行中状态 running

线程调度程序从可运行池中选择一个线程作为当前线程时线程所处的状态。这也是线程进入运行状态的唯一一种方式。

1. 阻塞状态 block

阻塞状态是线程阻塞在进入synchronized关键字修饰的方法或代码块(获取锁)时的状态。

1. 等待 waiting

处于这种状态的线程不会被分配CPU执行时间，它们要等待被显式地唤醒，否则会处于无限期等待的状态。

1. 超时等待 timed waiting

处于这种状态的线程不会被分配CPU执行时间，不过无须无限期等待被其他线程显示地唤醒，在达到一定时间后它们会自动唤醒。

1. 终止状态 terminatd

当线程的run()方法完成时，或者主线程的main()方法完成时，我们就认为它终止了。这个线程对象也许是活的，但是，它已经不是一个单独执行的线程。线程一旦终止了，就不能复生。

在一个终止的线程上调用start()方法，会抛出java.lang.IllegalThreadStateException异常。

synchronized关键字可以实现的锁分为三类：

synchlironized 1.6之前都是重量级锁；

使用synchronized关键字实现重量级锁的步骤：

1. 调用OS函数来实现锁，是通过int pthread\_mutex\_t mt;(PS:这里的int类型只是辅助理解它的参数类型为整型)这个参数，然后调用函数pthread\_mutex\_init(mt)来获取锁；
2. 里面pthread\_mutex\_lock(mt);对mt执行加1操作，如果+1操作不成功，就没有获取，等待前一个锁的释放；
3. 获取锁执行结束后执行pthread\_mutex\_unlock(mt)函数来-1变成0，释放锁；
4. 偏向锁

实际情况下方法一定要保证线程安全，但是实际情况不一定有线程互斥，所以偏向锁是synchronized锁的对象如果没有资源的竞争的情况下存在的；

偏向锁第一次会调用os函数，后续是不会再调用OS函数；

偏向锁是当前线程锁的对象获取锁，并在锁上打上标记；

比如：

**public synchronized void** sync(){  
 System.*out*.println(Thread.*currentThread*().getName());  
}

sync这个方法加入只有一个线程在运行，而且**synchronized**锁的对象是当前对象，相当于**synchronized(this)**，这时是没有竞争，**synchronized**就是偏向锁；

偏向锁的证明：

调用锁就必然调用os的pthread\_mutex\_lock(mt)函数来对锁对象上锁；

对于偏向锁，第一次会pthread\_mutex\_lock(mt)函数来上锁，而后续调用了synchronized关键字也不会再调用这个函数；

证明的思路：执行java代码时将Linux执行OS函数的线程ID都打印出来，同时在java代码synchronized代码块中也将当前的执行线程ID打印出来；因为实现锁会调用OS函数pthread\_mutex\_lock(mt)，此时会打印出对应的ID，如果与JAVA synchronized代码块中打印的线程ID一致；根据两个语句打印的次数可以判断锁的类型；

对于偏向锁，最初OS的pthread\_mutex\_lock(mt)会被调用一次，获得一个ID，java会同样适用这个线程，加锁成功后基本不再调用pthread\_mutex\_lock(mt)；

1. 轻量级锁
2. 重量级锁

借助于OS的函数来实现的锁；因为java调用OS函数比较麻烦，JVM需要从内核态切换为用户态；

synchronized作为重量级锁是为了多个线程在互斥时能够正常的运行；

线程互斥：指同一资源同时只允许一个访问者对其进行访问，具有唯一性和排他性；但是互斥无法限制访问者对资源的访问顺序，即访问是无序的；

# Synchronized和volatile关键字

## Java 内存模型中的可见性、原子性和有序性

1. 可见性

可见性是一种复杂的属性，因为可见性中的错误总是会违背我们的直觉。通常，我们无法确保执行读操作的线程能适时地看到其他线程写入的值，有时甚至是根本不可能的事情。为了确保多个线程之间对内存写入操作的可见性，必须使用同步机制。

可见性，是指线程之间的可见性，一个线程修改的状态对另一个线程是可见的。也就是一个线程修改的结果。另一个线程马上就能看到。比如：用volatile修饰的变量，就会具有可见性。volatile修饰的变量不允许线程内部缓存和重排序，即直接修改内存。所以对其他线程是可见的。但是这里需要注意一个问题，volatile只能让被他修饰内容具有可见性，但不能保证它具有原子性。比如 volatile int a = 0；之后有一个操作 a++；这个变量a具有可见性，但是a++ 依然是一个非原子操作，也就是这个操作同样存在线程安全问题。

在 Java 中 volatile、synchronized 和 final 实现可见性。

1. 原子性：

多线程原子性：指一个操作要么完成，要么不完成，不能中断；

原子是世界上的最小单位，具有不可分割性。比如 a=0；（a非long和double类型） 这个操作是不可分割的，那么我们说这个操作时原子操作。再比如：a++； 这个操作实际是a = a + 1；是可分割的，所以他不是一个原子操作。非原子操作都会存在线程安全问题，需要我们使用同步技术（sychronized）来让它变成一个原子操作。一个操作是原子操作，那么我们称它具有原子性。java的concurrent包下提供了一些原子类，我们可以通过阅读API来了解这些原子类的用法。比如：AtomicInteger、AtomicLong、AtomicReference等。

在 Java 中 synchronized 和在 lock、unlock 中操作保证原子性。

1. 有序性：

Java 语言提供了 volatile 和 synchronized 两个关键字来保证线程之间操作的有序性，volatile 是因为其本身包含“禁止指令重排序”的语义，synchronized 是由“一个变量在同一个时刻只允许一条线程对其进行 lock 操作”这条规则获得的，此规则决定了持有同一个对象锁的两个同步块只能串行执行。

## synchronized

synchronized锁的一个对象，一个对象只有一个锁信息pthread\_mutex\_lock(mt)对mt+1；

为什么锁对象？

synchronized锁的是类的实例和字节码，对象里面可以记录所信息；

Object o = new Object();

synchronized里面锁的不是引用对象o，而是堆里面new出来的对象；在对象头里面记录了锁信息；

锁对象的改变

锁定某对象o，如果o的属性发生改变，不影响锁的使用

但是如果o变成另外一个对象，则锁定的对象发生改变

应该避免将锁定对象的引用变成另外一个对象

PS：synchronized使用在静态方法上锁的是类对象；

一些案例：

1. 不能用一些常量作为锁：

eg:

String s1 = **"hello"**;  
String s2 = **"hello"**;

synchronized(s1) synchronized锁的是堆里面的对象，而s1和s2是放在常量池里面的，所以是无法被获取的；

1. synchronized里面的代码块越少越好；

面试问题：

1. 同步方法和非同步方法能否同时调用？

肯定是可以；非同步方法不需要锁

写的方法加锁，读不加锁，如果写没有执行完，读取是不会获得最新数据的，造成脏读；

eg：synchronized set(int value)方法对value执行操作，在操做完成前获取的数据都是value最初的数据；只有set结束后才是最新的数据；

1. 一个同步方法调用另一个同步方法，能否获取锁？ 重入锁

**synchronized void** test1(){  
 System.*out*.println(**"test1 start........."**);  
 test2();  
}  
**synchronized void** test2(){  
 System.*out*.println(**"test2 start......."**);  
}

调用test1(),test2()可以正常获取锁；

synchronized的锁的重入机制；

同理子类的同步方法中调用父类的方法也是可以的；

1. 出现异常需要及时处理来释放锁

\* T2线程能否执行？  
**public class** Demo {  
 **int** count = **0**;  
 **synchronized void** test(){  
 **while** (**true**) {  
 count ++;  
 **if** (count == **5**) {  
 //碰到异常的情况，如果没有处理，会自动释放锁，所以T2可以执行。  
 **int** i = **1**/**0**;  
 }  
 }  
 }  
 **public static void** main(String[] args) {  
 Demo demo11 = **new** Demo();  
 Runnable r = () -> demo11.test();  
 **new** Thread(r, **"t1"**).start();  
 **try** {  
 TimeUnit.*SECONDS*.sleep(**1**);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 **new** Thread(r, **"t2"**).start();  
 }  
}

t1获取偏向锁(已经证明)，导致t2一直获取不到无法执行，但是当t1抛出异常就自动释放锁，t2就可以运行；

所以synchronized代码块中的异常要处理不要抛出；

1. 多个Atomic类连续调用能否保证原子性？

连续调用是指一次调用两个方法

**public class** Demo {  
 AtomicInteger count = **new** AtomicInteger(**0**);   
 **public void** test(){  
 **for** (**int** i = **0**; i < **10000**; i++) {  
 **if**(count.get() < **1000**){  
 count.incrementAndGet();  
 }  
 }  
 }  
 **public static void** main(String[] args) {  
 Demo demo = **new** Demo();  
 List<Thread> threads = **new** ArrayList();  
 **for** (**int** i=**0**;i<**10**;i++) {threads.add(**new** Thread(demo::test, **"thread-"** + i)); }  
 threads.forEach((o)->o.start());  
 threads.forEach((o)->{  
 **try** { o.join(); } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 });  
 System.*out*.println(demo.count);  
 }  
}

运行结果：大于1000，尽管作了小于1000的判断；

所以说明连续调用不能保证原子性

比如count加到999了，这时候一个线程拿到count判断，虽然.get方法保证原子性，但是他阻止不了其它线程也来判断，所以第一个线程还没加完，第二个线程也进来了，这时候两个线程都给count加了，当前判断是小于1000，但是实行到incrementAndGet时count已经被其他线程执行了+操作，他再次执行就大于了1000；

1. 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给出提示并结束；

仅适用volatile关键字来实现List的线程看见是不行的：

**public class** Container2 {  
 **volatile** List lists = **new** ArrayList();  
 **public void** add(Object o){ lists.add(o); }  
 **public int** size(){ **return** lists.size(); }  
 **public static void** main(String[] args) {  
 Container2 c = **new** Container2();  
 **new** Thread(()->{  
 **for** (**int** i = **0**; i < **10**; i++) {  
 c.add(**new** Object());  
 System.*out*.println(**"add "** + i);  
 **try** {  
 TimeUnit.*SECONDS*.sleep(**1**);  
 } **catch** (Exception e) {}  
 }  
 },**" t1"**).start();  
 **new** Thread(()->{  
 **while** (**true**) {  
 **if** (c.size() == **5**) {  
 **break**;  
 }  
 }  
 System.*out*.println(**"t2线程结束"**);  
 }, **"t2"**).start();  
 }  
}

因为假定没有add后面的休眠，执行的很快，线程2获取到主内存中的list，可能第一次为1的时候获取到，判断重新去获取的时候list.size可能已经大于5了，导致线程2一直获取不到等于5的情况；同时线程2一直在循环占用大量的CPU资源，需要修改如下：

**public static void** main(String[] args) {  
 Container4 c = **new** Container4();  
 Object lock = **new** Object();  
 **new** Thread(()->{  
 **synchronized** (lock) {  
 System.*out*.println(**"t2启动"**);  
 **if** (c.size() != **5**) {lock.wait();}  
 System.*out*.println(**"t2结束"**);  
 lock.notify();  
 }  
 },**" t2"**).start();  
 **new** Thread(()->{  
 System.*out*.println(**"t1启动"**);  
 **synchronized** (lock) {  
 **for** (**int** i = **0**; i < **10**; i++) {  
 c.add(**new** Object());  
 System.*out*.println(**"add "** + i);  
 **if** (c.size() == **5**) {  
 lock.notify();  
 lock.wait();//要释放锁，T2才能得到锁得以执行

**//此处如果不执行wait()，则线程2会一直持有锁，直到全部代码执行结束，也就是只有for结束后线程1才能获取锁继续执行**  
 TimeUnit.*SECONDS*.sleep(**1**);  
 }  
 }  
 }  
 }, **"t1"**).start();  
}

当判断!=5时调用wait()方法进入等待，直到线程2执行到唤醒操作；

进一步优化，使用latch(门闩)： CountdownLatch

**public static void** main(String[] args) {  
 Container5 c = **new** Container5();  
 CountDownLatch latch = **new** CountDownLatch(**5**);  
 **new** Thread(()->{  
 System.*out*.println(**"t2启动"**);  
 **if** (c.size() != **5**) {  
 **try** {  
 latch.await();//latch准备 只有计数器到0才会继续操作  
 } **catch** (Exception e) {}  
 System.*out*.println(**"t2结束"**);  
 }  
 },**" t2"**).start();  
 **new** Thread(()->{  
 System.*out*.println(**"t1启动"**);  
 **for** (**int** i = **0**; i < **10**; i++) {  
 c.add(**new** Object());  
 System.*out*.println(**"add "** + i);  
 latch.countDown();//latch减一  
 }  
 }, **"t1"**).start();  
}

1. 写一个固定容量同步容器，拥有Put和get方法，以及getCount方法

能够支持两个生产者线程以及10个消费者线程的阻塞调用;

使用**private** Lock lock = **new** ReentrantLock();  
**private** Condition producer = lock.newCondition();  
**private** Condition consumer = lock.newCondition();

这样执行producer.await();后只有producer线程会处于wait状态；

同理consumer.signalAll()，只有consumer会被唤醒；

来优化；

**public class** Container1<T>{  
 **private final** LinkedList<T> lists = **new** LinkedList<>();  
 **private final int** MAX = **10**;  
 **private int** count;  
 **public synchronized void** put(T t){  
 **while** (lists.size() == MAX) {//  
 **try** {  
 **this**.wait(); 🡪 producer.await();  
 } **catch** (InterruptedException e) { }  
 }  
 lists.add(t);  
 ++count;  
 **this**.notifyAll(); 更改为consumer.signalAll();  
 }  
 **public synchronized** T get(){  
 T t = **null**;  
 **while** (lists.size() == **0**) {  
 **try** {  
 **this**.wait(); 🡪 consumer.await();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 t = lists.removeFirst();  
 count--;  
 **this**.notifyAll(); 🡪 consumer.await();  
 **return** t;  
 }  
 **public static void** main(String[] args) **throws** InterruptedException {  
 Container1<String> c = **new** Container1<>();  
 **for** (**int** i = **0**; i < **100**; i++) {  
 **new** Thread(()->{  
 **for** (**int** j = **0**; j < **5**; j++) {  
 System.*out*.println(c.get());  
 }  
 }, **"c"** + i).start();  
 }  
 TimeUnit.*SECONDS*.sleep(**2**);  
 **for** (**int** i = **0**; i < **2**; i++) {  
 **new** Thread(()->{  
 **for** (**int** j = **0**; j < **25**; j++) {  
 c.put(Thread.*currentThread*().getName() + **""** + j);  
 }  
 }, **"p"** + i).start();  
 }  
 }  
}

1. 为什么大多数情况下wait和while是一起使用的？

**public synchronized void** put(T t){  
 **while** (lists.size() == MAX) {//  
 **try** {  
 **this**.wait();  
 } **catch** (InterruptedException e) { }  
 }  
 lists.add(t);  
 ++count;  
 **this**.notifyAll();  
}

简单概括：wait被notify后继续执行wait,释放锁(所以线程B才也会获取锁)，并后面的代码，如果是if判断，线程A判断满足条件后开始执行后面的add操作，但是还没有线程B也执行了判断满足条件，执行add，就超过了最大容量；

所以使用while循环，wait被唤醒后还要在进入while判断是否满足循环条件；

因为这里是有两个生产者线程并且容器容量是固定的，生产方法加了锁。如果容器满了，这时候第一个生产者线程拿到了锁，他会判断有没有满，如果满了就等待，在等待的时候会释放掉锁资源，这时候第二个生产者线程就会拿到锁，然后他也会判断是否满，因为容器是满了，第二个生产者线程也会等待并且释放锁，当消费者消费之后唤醒所有线程，这时候两个生产者线程都醒来了，因为要竞争锁资源，比如第一个生产者线程拿到了锁，他给容器又加到十了，陷入等待状态，锁资源释放掉，第二个生产者线程这时候拿到锁资源，他会继续执行（从上次睡眠的地方继续）,如果是if的话，他在wait阻塞之前就已经执行了一次if，所以不会再执行，而是继续往下执行，那这时候就超过了容器的容量。所以为了让他再一次判断，这里使用while

## PS：wait()和notify()方法的使用

Thread类中把线程从running状态转化为非runnable状态有一个方法就是wait方法。等待之后立即释放当前锁，并且后续的代码是被挂起的，并且进入等待池中等待唤醒当等待池中的线程被唤醒notify()后，再次执行此语句之后的语句；

notify()唤醒，随机唤醒使用当前锁的线程池等待线程其中的一个。

notifyAll()：唤醒线程池所有等待线程。

使用wait()和notify()必须是有锁；

## PS:CountdownLatch

1. countDownLatch这个类使一个线程等待其他线程各自执行完毕后再执行。
2. countDownLatch是通过一个计数器来实现的，计数器的初始值是线程的数量。每当一个线程执行完毕后，计数器的值就-1，当计数器的值为0时，表示所有线程都执行完毕，然后在闭锁上等待的线程就可以恢复工作了。

API方法：

//调用await()方法的线程会被挂起，它会等待直到count值为0才继续执行

public void await() throws InterruptedException { };

//和await()类似，只不过等待一定的时间后count值还没变为0的话就会继续执行

public boolean await(long timeout, TimeUnit unit) throws InterruptedException { };

//将count值减1

public void countDown() { };

## PS：ReentrantLock实现更精细的wait和notify控制

如上面3.1.6里面有生产者和消费者处于wait状态，调用notifyAll()后所有使用这个锁的线程都会被唤醒，不够精细，可以使用ReentrantLock来实现更精细的wait和唤醒；

**private** Lock lock = **new** ReentrantLock();  
**private** Condition producer = lock.newCondition();  
**private** Condition consumer = lock.newCondition();

这样执行producer.await();后只有producer线程会处于wait状态；

同理consumer.signalAll()，只有consumer会被唤醒；

## volatile

1. volatile介绍

java语言提供了一种稍弱的同步机制，即volatile变量，用来确保将变量的更新操作通知到其他线程。当把变量声明为volatile类型后，编译器与运行时都会注意到这个变量是共享的，因此不会将该变量上的操作与其他内存操作一起重排序。volatile变量不会被缓存在寄存器或者对其他处理器不可见的地方，因此在读取volatile类型的变量时总会返回最新写入的值。

在访问volatile变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此volatile变量是一种比sychronized关键字更轻量级的同步机制。

当对非 volatile 变量进行读写的时候，每个线程先从内存拷贝变量到CPU缓存中。如果计算机有多个CPU，每个线程可能在不同的CPU上被处理，这意味着每个线程可以拷贝到不同的 CPU cache 中。

而声明变量是 volatile 的，JVM 保证了每次读变量都从内存中读，跳过 CPU cache 这一步。

1. 当一个变量定义为 volatile 之后，将具备两种特性：
2. 保证此变量对所有的线程的可见性，这里的“可见性”，如本文开头所述，当一个线程修改了这个变量的值，volatile 保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新。但普通变量做不到这点，普通变量的值在线程间传递均需要通过主内存来完成。
3. 禁止指令重排序优化。有volatile修饰的变量，赋值后多执行了一个“load addl $0x0, (%esp)”操作，这个操作相当于一个内存屏障（指令重排序时不能把后面的指令重排序到内存屏障之前的位置），只有一个CPU访问内存时，并不需要内存屏障；（什么是指令重排序：是指CPU采用了允许将多条指令不按程序规定的顺序分开发送给各相应电路单元处理）。
4. volatile 性能：

volatile 的读性能消耗与普通变量几乎相同，但是写操作稍慢，因为它需要在本地代码中插入许多内存屏障指令来保证处理器不发生乱序执行。

1. 案例分析

1、volatile的多次执行：

public class Demo {  
 volatile int count = 0;  
 public void test(){  
 for (int i = 0; i < 10000; i++) {  
 count ++;  
 }  
 }  
 public static void main(String[] args) {  
 Demo demo = new Demo();  
 List<Thread> threads = new ArrayList();  
 for (int i = 0; i < 10; i++) {  
 threads.add(new Thread(demo::test, "thread-" + i));  
 }  
 threads.forEach((o)->o.start());  
 threads.forEach((o)->{  
 try {

**/** /Thread类中的join方法的主要作用就是同步，它可以使得线程之间的并行执行变为串行执行。  
 o.join();  
 } **catch** (Exception e) { e.printStackTrace(); }  
 });  
 }  
}

每个线程循环10000次但是结果是小于十万，并且每次的执行结果都不一样；

原因：因为volatile只保证了线程之间的可见性，并不能保证原子性，当线程A从主内存中获取值为100并执行加操作，在执行的过程中线程B也获取了得到的也是100，加1为101后写入主内存，A也执行结束也写入主内存也是101；

## ReentrantLock

synchronized是手动上锁，自动释放锁；

而ReentrantLock是手动上锁，手动释放锁；可以reentrantlock用于替代synchronized

使用reentrantlock可以完成同样的功能，reentrantlock必须要手动释放锁，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；

如果想使用lock和unlock方法可以extends ReentrantLock

1. 作为手动锁来使用

成员变量：Lock lock = **new** ReentrantLock();

**try** {  
 lock.lock();//lock 锁的是this对象  
 **for** (**int** i = **0**; i < **3**; i++) {  
 System.*out*.println(i);  
 TimeUnit.*SECONDS*.sleep(**1**);  
 }  
} **catch** (InterruptedException e) {  
} **finally** {  
 lock.unlock();  
}

1. tryLock()方法

tryLock()是立马返回，执行后续代码，返回一个Boolean值，能获取到就获取到锁并返回true；不能就立即返回false；不阻塞；

tryLock()也可以设置等待时间

**public boolean** tryLock(**long** timeout, TimeUnit unit)；

lock()如果能获取到就获取锁；如果获取不到就阻塞直到获取到锁；

**public void** test2(){  
 **boolean** locked = **false**;  
 **try** {  
 locked = lock.tryLock(**3**, TimeUnit.*SECONDS*);

**//等待3s后继续执行**  
 System.*out*.println(**"test2...."** + locked);  
 } **catch** (InterruptedException e) {  
 } **finally** {  
 **if** (locked) {  
 System.*out*.println(**"test2 end"**);  
 lock.unlock();  
 }  
 }  
}

1. lockInterruptibly()方法

当前面线程A处于阻塞状态一直不释放锁，可以调用interrupt(),然后会执行线程B 执行lock.lockInterruptibly()方法，该线程B会获取锁，执行完成后释放锁，A会再次获取，A中的任务也会继续执行下去；

**public static void** main(String[] args) {  
 ReentrantLock lock = **new** ReentrantLock();  
 Thread t1 = **new** Thread(()->{  
 **try** {  
 lock.lock();  
 System.*out*.println(**"t1 start"**);  
 TimeUnit.*SECONDS*.sleep(Integer.*MAX\_VALUE*);//相当于线程被阻塞了  
 System.*out*.println(**"t1 end"**);  
 } **catch** (InterruptedException e) {  
 System.*out*.println(**"interrupted!"**);  
 } **finally** {  
 lock.unlock();  
 }  
 });  
 t1.start();  
 Thread t2 = **new** Thread(()->{  
 **boolean** locked = **false**;  
 **try** {  
// lock.lock(); //此处如果使用lock会一直获取不到锁  
 //使用lockInterruptibly来锁定可以对Interrupt方法作出响应  
 lock.lockInterruptibly();  
 System.*out*.println(**"t2 start"**);  
 TimeUnit.*SECONDS*.sleep(**3**);  
 System.*out*.println(**"t2 end"**);  
 locked = **true**;  
 } **catch** (InterruptedException e) {  
 System.*out*.println(**"interrupted!"**);  
 } **finally** {  
 **if** (locked){  
 lock.unlock();  
 }  
 }  
 });  
 t2.start();  
 **try** {  
 TimeUnit.*SECONDS*.sleep(**1**);  
 } **catch** (InterruptedException e) { }

**//t2必须执行**interrupt**方法才会触发**lockInterruptibly  
 t2.interrupt();  
 }

}

1. 公平锁 传参为true

构造方法传参数true，多个线程使用lock会依次获取锁，不用抢；

ReentrantLock *lock* = **new** ReentrantLock(**true**);

**public class** RLDemo4 **extends** Thread{  
 //ReentrantLock可以指定是否为公平锁，true为公平，默认为false  
 **private static** ReentrantLock *lock* = **new** ReentrantLock(**true**);  
  
 @Override  
 **public void** run() {  
 **for** (**int** i = **0**; i < **100** ; i++) {  
 *lock*.lock();  
 **try** {  
 System.*out*.println(Thread.*currentThread*().getName() + **"获得锁"**);  
 } **catch** (Exception e) {} **finally** {  
 *lock*.unlock();  
 }  
 }  
 }  
 **public static void** main(String[] args) {  
 RLDemo4 rlDemo4 = **new** RLDemo4();  
 Thread t1 = **new** Thread(rlDemo4);  
 Thread t2 = **new** Thread(rlDemo4);  
 t1.start();  
 t2.start();  
 }  
}

非公平锁时可能t1抢到多次，t2才抢到一次；公平锁则会依次获取，但是不保证效率；

# 同步容器ThreadLocal ConcurrentLinkedQueue CopyOnWriteList Vector

## ThreadLocal 线程本地变量

ThreadLocal中填充的变量属于当前线程，该变量对其他线程而言是隔离的。ThreadLocal为变量在每个线程中都创建了一个副本ThreadLocalMap，那么每个线程可以访问自己内部的副本变量。即当前线程并不能获取其他线程设置的值；

应用场景

1. 在进行对象跨层传递的时候，使用ThreadLocal可以避免多次传递，打破层次间的约束。
2. 线程间数据隔离
3. 进行事务操作，用于存储线程事务信息。
4. 数据库连接，Session会话管理。

方法：

1. set(T t)

**public void** set(T value) {  
 Thread t = Thread.*currentThread*();  
 ThreadLocalMap map = getMap(t);  
 **if** (map != **null**) map.set(**this**, value);  
 **else** createMap(t, value);  
}

现获取当前线程，然后调用getMap获取ThreadLocalMap，如果map存在，则将当前线程对象t作为key，要存储的对象作为value存到map里面去。如果该Map不存在，则初始化一个。

ThreadLocalMap其实就是ThreadLocal的一个静态内部类，里面定义了一个Entry来保存数据，而且还是继承的弱引用。在Entry内部使用ThreadLocal(this)作为key，使用我们设置的value作为value。

1. get();

**public** T get() {  
 Thread t = Thread.*currentThread*();  
 ThreadLocalMap map = getMap(t);  
 **if** (map != **null**) {  
 ThreadLocalMap.Entry e = map.getEntry(**this**);  
 **if** (e != **null**) {  
 @SuppressWarnings(**"unchecked"**)  
 T result = (T)e.value;  
 **return** result;  
 }  
 }  
 **return** setInitialValue();  
}

首先获取当前线程，然后调用getMap方法获取一个ThreadLocalMap，如果map不为null，那就使用当前线程作为ThreadLocalMap的Entry的键，然后值就作为相应的的值，如果没有那就设置一个初始值。

1. remove();

移除

ThreadLocal造成的内存泄露问题

1. Thread中有一个map，就是ThreadLocalMap
2. ThreadLocalMap的key是ThreadLocal，值是我们自己设定的。
3. ThreadLocal是一个弱引用，当为null时，会被当成垃圾回收
4. 重点来了，突然我们ThreadLocal是null了，也就是要被垃圾回收器回收了，但是此时我们的ThreadLocalMap生命周期和Thread的一样，它不会回收，这时候就出现了一个现象。那就是ThreadLocalMap的key没了，但是value还在，这就造成了内存泄漏。

解决办法：使用完ThreadLocal后，执行remove操作，避免出现内存溢出情况。

## Vector 向量 线程安全的List

1. 介绍

Vector可以实现可增长的对象数组。与数组一样，它包含可以使用整数索引进行访问的组件。

1. Vector 继承了AbstractList，实现了List接口。
2. Vector实现了RandmoAccess接口，即提供了随机访问功能。
3. Vector 实现了Cloneable接口，即实现克隆功能。
4. Vector 实现Serializable接口，表示支持序列化。
5. Vector的构造方法
6. 无参构造方法

**public** Vector() {**this**(**10**);}

默认初始容量是10；

1. 指定初始容量

**public** Vector(**int** initialCapacity) {**this**(initialCapacity, **0**);}

增长因子只能是int：

**int** newCapacity = oldCapacity + ((capacityIncrement > **0**) ?  
 capacityIncrement : oldCapacity);

1. 指定增长因子

**public** Vector(**int** initialCapacity, **int** capacityIncrement) {  
 **super**();  
 **if** (initialCapacity < **0**)  
 **throw new** IllegalArgumentException(**"Illegal Capacity: "**+  
 initialCapacity);  
 **this**.elementData = **new** Object[initialCapacity];  
 **this**.capacityIncrement = capacityIncrement;  
}

1. 传进一个Collection

**public** Vector(Collection<? **extends** E> c) {  
 elementData = c.toArray();  
 elementCount = elementData.length;  
 // c.toArray might (incorrectly) not return Object[] (see 6260652)  
 **if** (elementData.getClass() != Object[].**class**)  
 elementData = Arrays.*copyOf*(elementData, elementCount, Object[].**class**);  
}

就是将传入的集合转化为数组，然后copyOf生成Vector的数组

1. 相关方法
2. add(T t);

**public synchronized boolean** add(E e) {  
 modCount++; //记录操作次数 Fail-Fast 机制  
 ensureCapacityHelper(elementCount + **1**);  
 elementData[elementCount++] = e;  
 **return true**;  
}

PS：Fail-Fast 机制：对于线程不安全的集合

ConcurrentModificationException，这就是所谓fail-fast策略。这一策略在源码中的实现是通过 modCount 域，modCount 顾名思义就是修改次数，对HashMap 内容的修改都将增加这个值，那么在迭代器初始化过程中会将这个值赋给迭代器的 expectedModCount。在迭代过程中，判断 modCount 跟 expectedModCount 是否相等，如果不相等就表示已经有其他线程修改了 Map：注意到 modCount 声明为 volatile，保证线程之间修改的可见性。

1. get(int index)

**public synchronized** E get(**int** index) {  
 **if** (index >= elementCount)  
 **throw new** ArrayIndexOutOfBoundsException(index);  
 **return** elementData(index);  
}

1. 案例demo

\* 有10000张火车票,同时有10个窗口对外售票  
 \* 请写一个模拟程序  
**public class** SaleOfTickets {  
 **private static** Vector<Integer> *tickets* = **new** Vector<>(**10**);  
 **static** {  
 **for** (**int** i = **0**; i < **10000**; i++) { *tickets*.add(i); }  
 }  
 **public static void** main(String[] args) {  
 **for** (**int** i = **0**; i < **10**; i++) {  
 **new** Thread(()->{  
 **while**(*tickets*.size() > **0**){  
 **try** { TimeUnit.*MILLISECONDS*.sleep(**10**);  
 } **catch** (InterruptedException e) {}  
 System.*out*.println(**"销售票编号:"** + *tickets*.remove(**0**));  
 }  
 }).start();  
 }  
 }  
}

两个原子操作放一起就不能再保证整体的原子性，因为两个操作之间被其他线程打断了

这里仍然不能保证原子操作：

因为线程A判断size>0进行操作，但是在remove之前线程B也获取size，此时肯定也满足条件，就会导致list 越界；同多个Atomic类连续调用不能保证原子性；改为：

**while**(**true**){  
 //这里使用synchronized，使两个操作具备了原子性，不会出问题  
 **synchronized**(*tickets*){  
 **if**(*tickets*.size() <= **0**){  
 **break**;  
 }  
 System.*out*.println(**"销售票编号:"** + *tickets*.remove(**0**));  
 }  
}

PS：此处不能将判断条件放在synchronized代码块外面，要不仍然无法保证整体的原子性；

## ConcurrentLinkedQueue

ConcurrentLinkedQueue是一个基于链接节点的无界线程安全队列，它采用先进先出的规则对节点进行排序，当我们添加一个元素的时候，它会添加到队列的尾部，当我们获取一个元素时，它会返回队列头部的元素。它采用了“wait－free”算法来实现；

1. 方法：
2. 案例demo：

**public class** SaleOfTickets4 {  
 **private static** Queue<Integer> *tickets* = **new** ConcurrentLinkedQueue<>();  
 **static** {  
 **for** (**int** i = **0**; i < **10000**; i++) { *tickets*.add(i); }  
 }  
 **public static void** main(String[] args) {  
 **for** (**int** i = **0**; i < **10**; i++) {  
 **new** Thread(()->{  
 **while**(**true**){  
 Integer poll = *tickets*.poll();  
 **if**(poll == **null**) **break**;  
 System.*out*.println(**"销售票编号:"** + poll);  
 }  
 }).start();  
 }  
 }  
}

保证了原子性分析：如果有的话*tickets*.poll()方法已经取出来了，后续并没有对这个队列进行额外的操作；poll底层实现是cas；

## CopyOnWriteList 写时复制容器

写的效率不高，但是读的效率很高