# ======线程的基本概念========

## 线程和进程的概念 √

1. 简单回答：

**进程**就是一个程序运行起来的状态，**线程**是一个进程中的不同的执行路径，进程的最小执行单元。

（你硬盘上一个QQ.exe是一个程序，双击运行，就叫一个进程，相对于程序来说进程是动态的概念。）

1. 专业回答：

进程是OS分配资源的基本单位，线程是执行调度的基本单位。

分配资源最重要的是：独立的内存空间，线程调度执行（线程共享进程的内存空间，没有自己独立的内存空间）

windows、linux、unix都是多进程、多线程的，dos是只支持单进程的，同一个时间点只能执行一个进程执行。

**纤程：**

用户态的线程，线程中的线程，切换和调度不需要经过OS

优势：1：占有资源很少 OS : 线程1M Fiber：4K 2：切换比较简单 3：启动很多个10W+

目前2020 3 22支持内置纤程的语言：Kotlin Scala Go Python(lib)... Java? （open jdk : loom）

Java中对于纤程的支持：没有内置，盼望内置

纤程 vs 线程池：很短的计算任务，不需要和内核打交道，并发量高！

## 创建线程的方式及实现 √

创建线程的三种方式：

* **new** Thread().start();
* **new** Thread(Runnable).start();
* 吹毛求疵：

线程池也是用的也是用的这两种之一

|  |
| --- |
| ExecutorService service = Executors.*newCachedThreadPool*();  service.execute(()->{  System.***out***.println("KKKKK");  });  service.shutdown();  ////// FutureTask+Callable ///////  **new** Thread(**new** FutureTask<Integer>(**new** MyCall())).start(); |

* JDK8: **new** Thread(()->{ }).start();

## 线程常见方法 √

### Thread.sleep();

线程睡眠指定的毫秒数，睡完后进入就绪状态

### Join

合并某个线程，等待某个线程执行结束，再恢复当前线程的运行。

在t1线程调用t2.join ，t1等待t2线程执行结束，t1再继续执行。

**问： 启动三个线程怎么让他们按顺序执行？**

在t1里调用t2.join，在t2里调t3.join



### Yield()

高风亮节，当前线程让出一下CPU，进到等待队列，给其他线程执行的机会，你能不能抢到cpu我就不管了。

### Priority

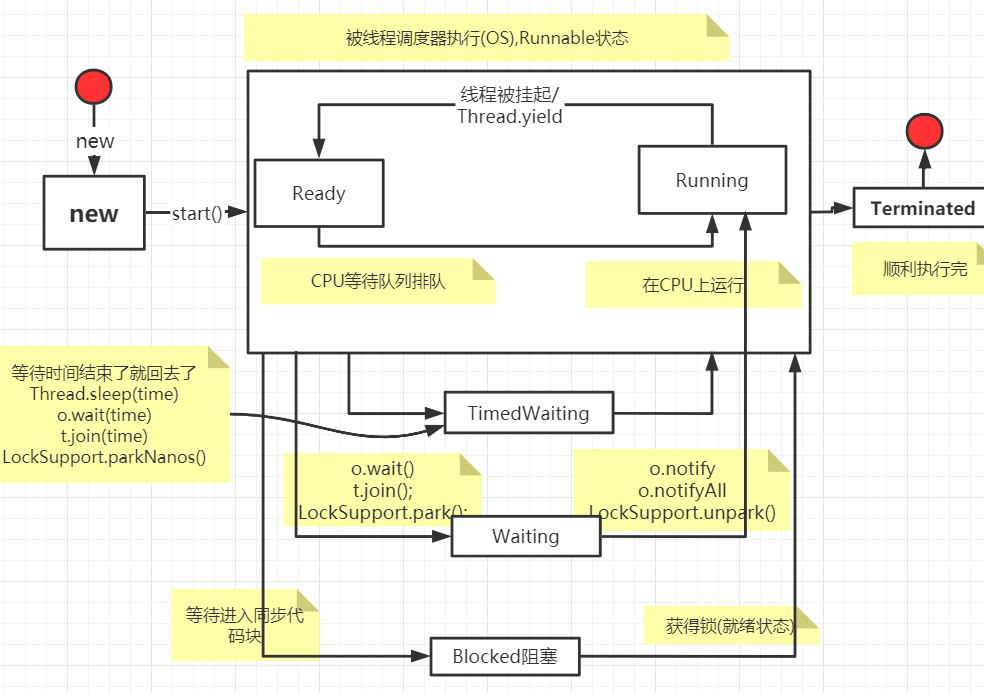
线程优先级

## 线程的生命周期，状态是如何转移的 √

六种状态

Thread.State 枚举类有6个状态：

***NEW，RUNNABLE，BLOCKED，WAITING，TIMED\_WAITING，TERMINATED***



* **（1）新建**

New一个线程，还没有调用start(); 时处于新建状态。

线程对象调用start();方法时，它会被线程调度器来执行（也就是交给操作系统执行），操作系统执行的时候，这整个状态叫**（2）Runnable状态**，Runnable内部又有两个状态：

* 1. **Ready就绪状态**

扔到CPU的等待队列里去等待CPU运行

* 1. **Running状态**

等真正扔到CPU上去运行的时候叫Running.（调用yield方法，从Running🡪Ready，线程调度器选中执行的时候 Ready🡪Running）

* **（3）Terminaled结束状态**

Runnable还有一些其他状态：

线程顺利执行完了进入Terminaled结束状态（不可以再回到new状态调用start，这就算结束了）

* **（4）TimedWaiting等待**

按照时间等待，等待时间结束了自己就到了Running状态。

Thread.sleep(time); 、o.wait(time)；、t.join(time)；、LockSupport.parkNanos()都是关于时间等待的方法

* **（5）Waiting等待**

在运行过程中，如果调用了o.wait()、t.join(); 、LockSupport.park();进入waiting状态；

调用o.notify();、o.notifyAll(); 、LockSupport.unpark();又回到Running状态

* **（6）Block阻塞**

同步代码块没有获得锁就会阻塞状态，获得锁就是就绪状态

### 追问：这些状态，哪些是JVM管理的，哪些是操作系统管理的？

这些状态全是由JVM管理的，因为JVM管理的时候也要通过操作系统，所以，哪个是JVM哪个是操作系统，他俩分不开。JVM是跑在操作系统上的一个普通程序。

### 追问：线程什么时候会被挂起？挂起是否也是一个状态

Running的时候，在一个CPU上会跑很多个线程，CPU会隔一段时间执行这个线程一下，在隔一段时间执行那个线程一下，这是CPU内部的一个调度，把这个状态线程扔出去，从Running扔出去，就叫线程被挂起，CPU控制它。

## 说说线程安全问题，什么是线程安全，如何保证线程安全

线程安全：当多个线程访问某一个类（对象或方法）时，这个类始终都能表现出正确的行为，那么这个类（对象或方法）就是线程安全的。

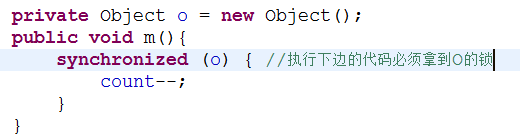
## Synchronized 基本概念√

### 概念：

可以在任意对象及方法上加锁，而加锁的这段代码成为“互斥区” 或 “临界区”

### 对象锁：

锁定的并不是count数字，可以任意指定，想锁谁就锁谁。



也可以不new对象锁定，可以**synchronized** (**this**)锁定当前对象。

### 类锁：

\*\*静态方法static是没有this对象的，static方法加synchronized就是synchronized(T.class);锁的是T类的对象

### 代码示例：

多个线程一把锁 。会有锁竞争问题，消耗CPU资源

|  |
| --- |
| **public** **class** MyThread **extends** Thread{    **private** **int** count = 5 ;    //加锁  @Override  **public** **synchronized** **void** run(){  count--;  System.***out***.println(**this**.*currentThread*().getName() + " count = "+ count);  }    **public** **static** **void** main(String[] args) {  /\*\*  \* 分析：当多个线程访问myThread的run方法时，以排队的方式进行处理（这里排对是按照CPU分配的先后顺序而定的），  \* 一个线程想要执行synchronized修饰的方法里的代码：  \* 1 尝试获得锁  \* 2 如果拿到锁，执行synchronized代码体内容；拿不到锁，这个线程就会不断的尝试获得这把锁，直到拿到为止，  \* 而且是多个线程同时去竞争这把锁。（也就是会有锁竞争的问题）  \*/  MyThread myThread = **new** MyThread();  Thread t1 = **new** Thread(myThread,"t1");  Thread t2 = **new** Thread(myThread,"t2");  Thread t3 = **new** Thread(myThread,"t3");  Thread t4 = **new** Thread(myThread,"t4");  Thread t5 = **new** Thread(myThread,"t5");  t1.start();  t2.start();  t3.start();  t4.start();  t5.start();  }  }  打印：和心里预期一样，就是线程安全的 |

扩展：如果是多个线程，每个线程锁定自己的对象，（多个线程多把锁）就不存在锁竞争问题，每个线程正常执行。

## synchronized方法和非synchronized方法能否同时被多个调用？√

有一个类T，有一个synchronized方法m1，一个普通方法m2，new T()；一个对象，多个线程能否同时调用synchronized方法和非synchronized方法？ 用大腿想想，肯定可以的。调m1的时候需要加锁，调m2的时候不需要加锁。

## 解释/为什么说Synchronized是可重入锁 \*\*\*\*\* √

Synchronized的一个重要属性：可重入。

一个对象里，一个同步方法可以调用另外一个同步方法，一个线程已经拥有某个对象的锁，再次申请的时候仍然会得到该对象的锁。也就是说synchronized获得锁是可重入的。

一个方法m1是synchronized的，另一个方法m2也是synchronized的，m1里调用m2，m1开始的时候获得了这把锁，然后m1里调m2，这时候m2发现是同一个线程，因为你m2也需要申请这把锁，他发现是同一个线程申请的这把锁，允许，这就叫可重入锁。

**原理：**

Synchronized一个对象的时候，在对象头上有2位标识是否加锁，一个线程锁定这个对象，这个线程又申请这把锁，发现当前持有这把锁的线程就是自己，就继续往下执行了。

## 异常锁 √

程序出现异常，锁将被释放，别的准备拿到这把锁的线程乱冲进来，程序乱入。

所以在并发处理的过程中，有异常要多加小心，不然可能发生不一致的情况。

**发生异常得处理：**

* 情况一：

所有任务不是一个整体，一个有问题不影响其他，只把出错的记录日志，下次执行,此时catch应该捕捉Exception。

* 情况二：

所有任务是一个整体，一个有问题影响其他，此时可以通过捕捉InterruptedException、catch里抛出RuntimeException();来终止线程继续执行。

例子程序：

|  |
| --- |
| **import** java.util.concurrent.TimeUnit;  **public** **class** T {  **int** count = 0;  **synchronized** **void** m() {  System.***out***.println(Thread.*currentThread*().getName() + " start");  **while**(**true**) {  count ++;  System.***out***.println(Thread.*currentThread*().getName() + " count = " + count);  **try** {  TimeUnit.***SECONDS***.sleep(1);    } **catch** (InterruptedException e) {  e.printStackTrace();  }  **if**(count % 5 == 0) {  **int** i = 1/0; //此处抛出异常，锁将被释放，要想不被释放，可以在这里进行catch，然后让循环继续  System.***out***.println(i);  }  }  }  **public** **static** **void** main(String[] args) {  T t = **new** T();  Runnable r = **new** Runnable() {  @Override  **public** **void** run() {  t.m();  }  };  **new** Thread(r, "t1").start();    **try** {  TimeUnit.***SECONDS***.sleep(3);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **new** Thread(r, "t3").start();  **new** Thread(r, "t2").start();  **new** Thread(r, "t4").start();  **new** Thread(r, "t5").start();  **new** Thread(r, "t6").start();  **new** Thread(r, "t7").start();  }  } |



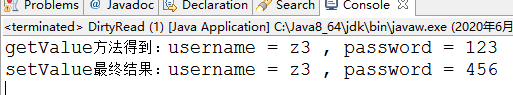
## 脏读问题 √

对于对象的同步和异步的方法，我们在设计自己的程序的时候，一定要考虑问题的整体，不然就会出现数据不一致的错误，很经典的错误就是脏读。

例子程序

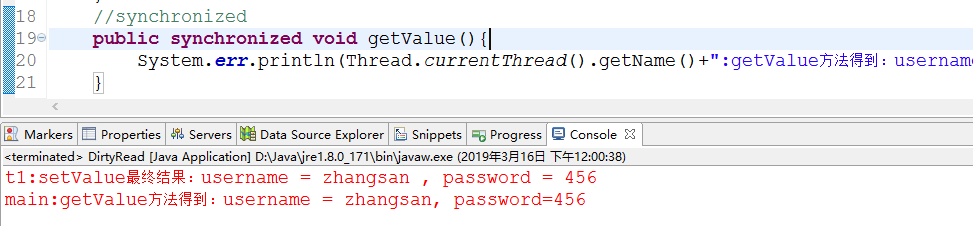
|  |
| --- |
| /\*\*  \* 业务整体需要使用完整的synchronized，保持业务的原子性。  \* **@author** alienware  \*  \*/  **public** **class** DirtyRead {  **private** String username = "bjsxt";  **private** String password = "123";    **public** **synchronized** **void** setValue(String username, String password){  **this**.username = username;  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **this**.password = password;  System.***out***.println("setValue最终结果：username = " + username + " , password = " + password);  }    **public** **void** getValue(){  System.***out***.println("getValue方法得到：username = " + **this**.username + " , password = " + **this**.password);  }      **public** **static** **void** main(String[] args) **throws** Exception{    **final** DirtyRead dr = **new** DirtyRead();  Thread t1 = **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  dr.setValue("z3", "456");  }  });  t1.start();  Thread.*sleep*(1000);    dr.getValue();  }  } |

执行结果：



t1线程先对username设置值，将初始值 lhy 修改为 zhangsan ，然后睡眠1秒，此时 password还是初始值 123，所以主线程在1秒的时候读取到的username是修改后的zhangsan，password是初始值123，等到2秒时，t1线程将password修改为456，t1线程执行结束。打印username=zhangsan 、password=456。

要想保证读写业务的一致性，getValue也应该加上synchornized关键字。加上之后打印的结果就是我们预期的：





## 并行和并发的概念

## 进程间通信的方式

## 一个线程连着调用start两次会出现什么情况？

1. （由于状态只有就绪、阻塞、执行，状态是无法由执行转化为执行的，所以会报不合法的状态！）
2. 如何调试多线程的程序；

# =======线程池======

## 讲讲线程池的实现原理

## 多线程是解决什么问题的？线程池解决什么问题？

## 线程池的四种实现方式，重要参数及原理？任务拒接策略有哪几种？

## 线程池，如何设计的，里面的参数有多少种，里面的工作队列和线程队列是怎样的结构，如果给你，怎样设计线程池？

## Java线程池的核心属性以及处理流程；

## Java线程池的几个参数的意义和实现机制；

## Java线程池使用无界任务队列和有界任务队列的优劣对比；

## 线程池，如何根据CPU的核数来设计线程大小，如果是计算机密集型的呢，如果是IO密集型的呢？

## 线程池内的线程如果全部忙，提交⼀个新的任务，会发⽣什么？队列全部塞满了之后，还是忙，再提交会发⽣什么？

## ExecutorService你一般是怎么⽤的？是每个Service放一个还是个项目放一个？有什么好处？

## 从源码详细说下 Java 里面的线程池吧，使用线程池有什么要注意的地方？你们公司有规范吗？

线程池：主要工作是控制工作线程的数量，处理过程中将任务放入队列，然后在线程中创建任务，如果线程超过了最大线程数量，则排队等候，等其他线程执行完再从队列中取出任务来执行

主要特点：线程复用，控制最大线程数，管理线程

优点：1降低资源消耗2提高响应速度3提高线程的可管理性

获取线程的方式1、继承Thread类2、实现Runable接口3、实现Callable接口4、线程池，newFixedThreadPoll固定大小 newsingleThreadExcute单线程 newCacheTreadPoll不固定大小

public static ExecutorService newFixedThreadPool(int var0) {

return new ThreadPoolExecutor(var0, var0, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue());

}

public static ExecutorService newSingleThreadExecutor() {

return new Executors.FinalizableDelegatedExecutorService(new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue()));

}

public static ExecutorService newCachedThreadPool() {

return new ThreadPoolExecutor(0, 2147483647, 60L, TimeUnit.SECONDS, new SynchronousQueue());

}

线程池7大参数：

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue) {

this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,

Executors.defaultThreadFactory(), defaultHandler);

ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler)

corePoolSize：线程池中常驻线程数量，线程池中不会被注销的线程

maximumPoolSize：线程池中能够容纳同时执行的最大线程数量，必须大于等于1

keepAliveTime：多余空闲线程存活时间，当前线程数量超过corePollSize时当空闲时间达到keepAliveTime时，多余空闲线程会被注销，直到剩下corepoolsize个线程为止

unit：keepalivetime的时间单位

workQueue：任务队列，被提交但是未被执行的任务

threadFactory：生产线程池中工作线程的线程工厂，用于创建线程，一般使用默认

handler：拒绝策略，当任务队列workqueue满了并且工作线程数大于等于线程池的最大线程数maximumpoolsize时的拒绝方式

1、abortpolicy，默认方式，直接抛出RejectedExecutorException异常

2、callerRunsPolicy:调节模式，既不会抛出异常，也不会抛弃任务，而是将任务回退到任务调用者，从而降低新的任务流量

3、DiscardOldestPolicy:抛弃队列中等待最久的任务，然后把当前任务加入队列中尝试再次提交任务

4、DiscardPolicy：直接抛弃任务，不予处理也不抛出异常，如果允许丢弃任务，是最佳选择

线程池大小配置原则：1、io密集型 由于io密集型线程不是一直执行任务，则应配置尽量大些，一般可设置为cpu核数\*2；cpu核数/（1-阻塞系数） 阻塞系数一般是0.8-0.9

2、cpu密集型 由于cpu密集型一直都处于高速的运算，而没有阻塞，一般公式 cpu核数+1

# 三、Synchronized

## synchronized 实现原理（对象监视器）

## synchronized 与 lock 的区别

## synchronized与ReentraLock哪个是公平锁；

## synchronized关键字锁住的是什么东西？在字节码中是怎么表示的？在内存中的对象上表现为什么？

## sync原理详细，sync内抛异常会怎样，死锁吗？还是释放掉？怎么排查死锁？死锁会怎样？有没有什么更好的替代方案？

## 讲下 ReentrantLock 可重入锁？ 什么是可重入锁？为什么要设计可重入锁？

## Synchronized 和 RecentLock 有什么区别？这两个有没有深入了解源码

## 讲下 Synchronized？

## wait/notify/notifyAll⽅法需不需要被包含在synchronized块中？这是为什么？

死锁：两个及以上个线程争执行过程中夺资源而造成的相互等待的现象，若无其他外力干涉将会一直僵持下去

原因：系统资源不足；进程运行推进的顺序不合适；资源分配不当

死锁分析:linux ps-ef|grep ；windows jps -l获取进程号； jstack 进程号

# ========= Volatile========

## volatile 轻量级同步机制 ：保证可见性，不保证原子性，禁止指令重排

## volatile关键字作用，他是怎样做到同步的？

## volatile 实现原理

（禁止指令重排、刷新内存）

## Volatile 为什么不支持原子性？举个例子

# ======常见的原子操作类=======

1. AtomicInteger底层实现原理；
2. 说说 CountDownLatch、CyclicBarrier 原理和区别

Atomic 怎么设计的？有用过里面的类吗？

# =======用过并发包下边的哪些类======

# =====一些高级概念=====

## 重入锁的概念，重入锁为什么可以防止死锁

## 解释一下自旋;

## 说说 Semaphore 原理

## 说说 Exchanger 原理

## ThreadLocal 原理分析，ThreadLocal为什么会出现OOM，出现的深层次原理

## 产生死锁的四个条件（互斥、请求与保持、不剥夺、循环等待）

## 如何检查死锁（通过jConsole检查死锁）

## AQS同步队列

## CAS无锁的概念、乐观锁和悲观锁

## CAS机制会出现什么问题；

## 什么是ABA问题，出现ABA问题JDK是如何解决的

## Java 8并法包下常见的并发类

## 偏向锁、轻量级锁、重量级锁、自旋锁的概念

## wait方法能不能被重写，wait能不能被中断；

## 常用的避免死锁方法；

## 写出一个必然会产生死锁的伪代码；

## Java中有哪些同步方案（重量级锁、显式锁、并发容器、并发同步器、CAS、volatile、AQS等）

## 多个线程同时读写，读线程的数量远远⼤于写线程，你认为应该如何解决并发的问题？你会选择加什么样的锁？

## 乐观悲观锁的设计，如何保证原子性，解决的问题；

## 乐观锁的业务场景及实现方式

## AQS原理，ReentranLock源码，设计原理，整体过程。

非公平锁/公平锁 Lock lock = new ReentrantLock({true：公平锁；空或false：非公平锁})，默认非公平锁;

公平锁：先来后到，顺序执行，按照申请锁的顺序；非公平锁：可加塞，会造成优先级翻转或饥饿现象，吞吐量大，synchronized也是一种非公平锁

可重入锁（递归锁）ReentrantLock和Synchronized 线程能够进入任何它已经拥有的锁的所同步的代码块，（获取外层方法锁，可直接获取内层方法的锁）可避免死锁，lock与unlock必须配对

自旋锁：尝试获取锁的线程不会立即阻塞，而是采用循环的方式尝试获取锁，可以减少线程上下文的切换的消耗，不过会增加cpu的消耗

## 怎么认为一个类是线程安全？线程安全的定义是什么？Java有多少个关键字进行同步？为什么这样设计？（聊了一大堆，一堆为什么）；

## 两个线程设计题。记得一个是：t1,t2,t3，让t1，t2执行完才执行t3，原生实现。

## 说下多线程，我们什么时候需要分析线程数，怎么分析，分析什么因素;

## 解释一下信号量

## 保证线程安全的还有其他的方式吗？

## 线程安全类和线程不安全的类，项目使用的时候你会怎么选择？怎么判断项目代码哪里会有线程不安全问题？

## cas 比较交换 自旋锁 unsafe

缺点:do while 循环开销大；只能保证一个共享数据；引发出ABA问题（版本号）

读写锁/互斥锁（独占锁：写锁；共享锁：读锁） reentrantLock和synchronized都是独占锁，只能同时被一个线程共享；reentrantReadWriteLock读写锁

countDownLatch 计数器 countdownlatch.countdown() 计数器减一;countdownlatch.await()主线程阻塞，等待计数器减为零。

CyclicBarrier new CyclicBarrier(上限屏障数，Runable接口) cyclicBarrier.await（）阻塞当前线程，待所有线程完成后，执行runable接口

Semaphore 信号量 semaphore.acquire()获得锁；semaphore.replace()释放锁；tryacquire（time，时间单位）尝试在时间内获取锁，若获取不到抛异常

|  |
| --- |
|  |

阻塞队列

ArrayBlockingQueue由有限数组组成的阻塞队列

LinkedBlockingQueue由有限（默认integer max）链表组成的阻塞队列

synchronousQueue：单元素队列

抛出异常 特殊值 阻塞 超时

插入 add（e） offer(e)true/false put(e) offer(e,time,unit)

移除 remove()删除队首 poll()队首 take() poll(time,unit)

检查 element()返回队首 peek()null 无 无

futureTask实现了runable接口，构造方法中需要传入callAble接口实现类，多个线程共用一个futureTask，只有一个线程执行runable

synchronized与lock的区别

1、原始构成

synchronized是关键字属于jvm层次，底层通过monitor对象来完成，wait和notify等方法也依赖monotor对象在同步块或方法中才能被调用monitorenter，monitorexist；

lock属于api层次的调用

2、使用方法

synchronized不需要手动释放锁，不会造成死锁

lock需手动释放，lock和unclock需配对使用

3、等待是否可中断

synchronized锁不可中断，只能正常结束或异常退出

renntranLock可设置超时退出tryLock(time,unit);lockInterruptibly()方法代码块中调用interrup()方法可中断

4、加锁是否公平

synchronized是非公平锁

lock可通过构造方法，true公平锁，false非公平锁，默认非公平锁

5、绑定多个条件condition

synchronized不支持，只能随机唤醒一个或所有

reentrantLock可分组唤醒，精确唤醒某个线程

非静态方法的锁默认是this，静态方法的锁是对应的Class实例

某一个时刻内，只能有一个线程持有锁，无论几个方法

三个线程循环输出ABC