一个进程中的多个线程可以被多个处理器并行执行

串行：事件A，B，C逐个逐个执行。(需要投入1个人)

并行：事件A,B,C一起同时执行。（需要投入3个人）

并发：A等待的时候做B，B等待的时候做C。（需要投入一个人）

临界区：

1. 表示一种公共资源/共享数据，可以被多个线程使用，但每一次只有一个线程能用它

一但被占用，其他线程想用就必须等待 //打印机

2. 在并行程序中，临界区资源就是保护的对象

阻塞：线程一直等资源，不干其他事

非 ：线程没有请求到资源，线程可以干其他事

死锁：两个进程互相等待对方资源释放

饥饿：线程一直无法获取资源

活锁：线程互相让资源，结果没有一个线程使用资源

并发级别：

阻塞： 当一个线程进入临界区，其他线程必须等待 / /悲观

无饥饿：不管线程优先级，根据先来后到顺序执行

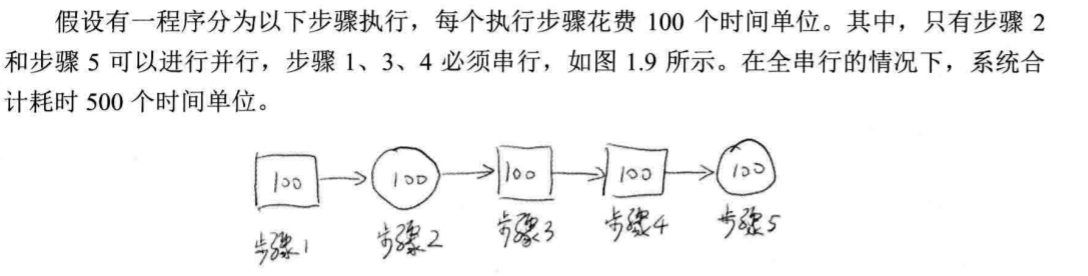
无障碍：都可以进入临界区，可能发生数据冲突--同时修改共享数据，会回滚操作，冲突严重可能导致全部失败回滚--一致性标记 //乐观

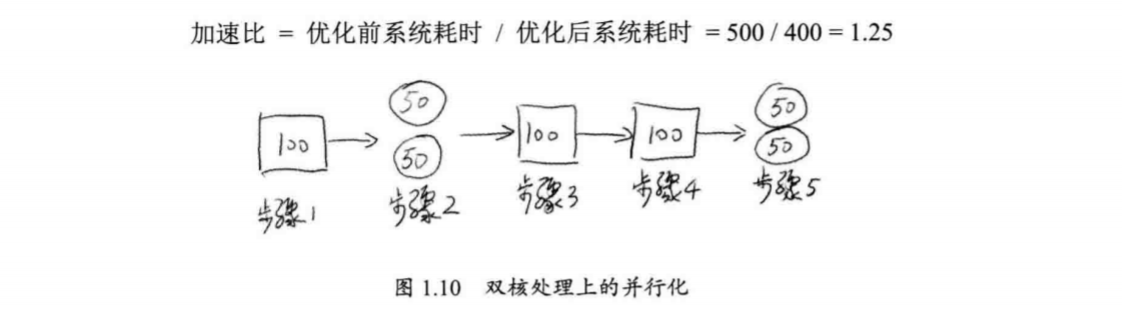
无锁：也是无障碍，但能保证每次有一个线程在有限步内完成操作离开临界区

无等待： 无锁只要求一个，它要求所有，这样不会引起饥饿问题

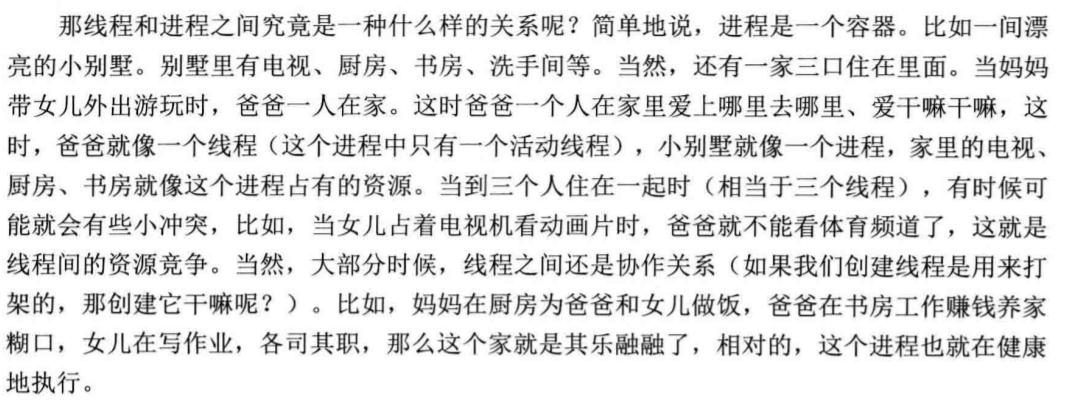
核越多越好吗？

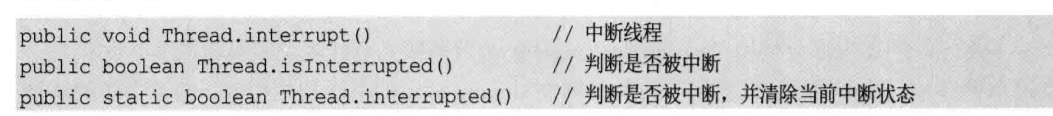
CPU越多，系统的串行程序比重越低，则性能越好

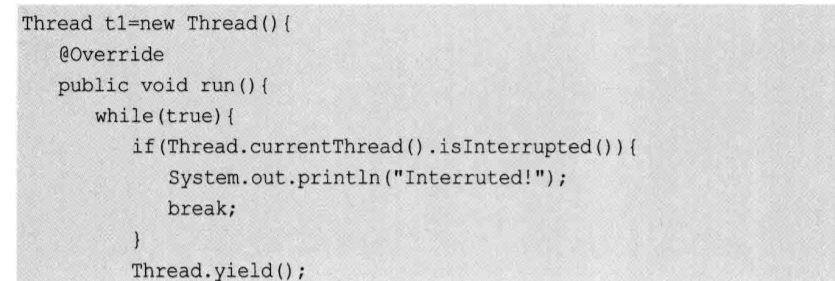
单核



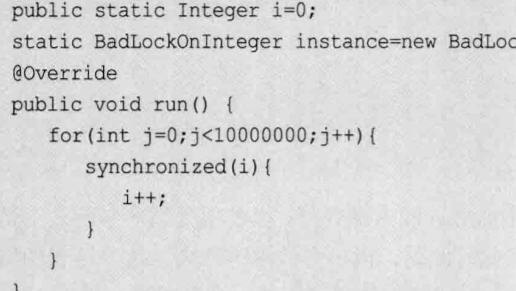
第二章



中断：  




与String类似，所有的包装类都是final类，即不可变类。虽然在代码A处看起来是改变了counter的值，但实际上是创建了另一个对象，并将方法内的counter参数的引用指向了这个新创建的对象



ReentrantLock(可重入锁)和synchronized

**同**

* 性能差不多
* 可重入性：ReentrantLock字面意思即为再进入锁，称为可重入锁，其实synchronize所使用的锁也是可以重入的，同一个线程可以多次进入同一个锁

两者关于这个区别不大，它们都是同一个线程进入一次，锁的计数器进行自增，要等到锁的计数器下降为零时，才能释放锁

**不同**

* 便利性：

synchronized更便利，它是由编译器保证加锁与释放

ReentrantLock需要手动声明与释放锁

所以为了避免忘记手工释放锁造成死锁，所以最好在finally中声明释放锁

* ReentrantLock可以指定是公平锁还是非公平锁，synchronized只能是非公平锁

所谓公平锁就是先等待的线程先获得锁）

* ReentrantLock提供了一个Condition类，可以分组唤醒需要唤醒的线程

即可以绑定多个Condition，每个方法需要的Conditon不同，A唤醒B->C

https://blog.csdn.net/lixinkuan328/article/details/94426872

synchronized要么随机唤醒一个线程，要么全部唤醒。

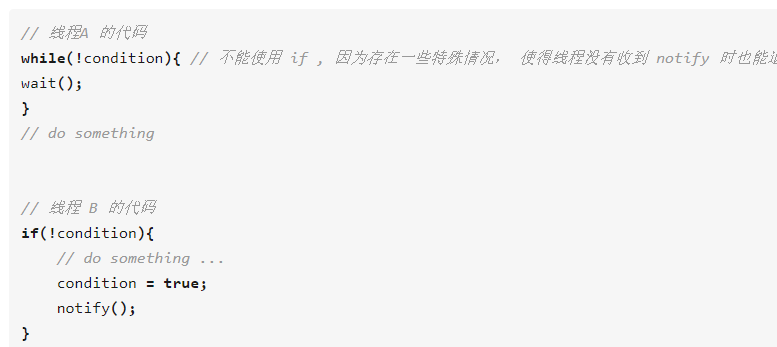
* 等待锁时可以中断，syn等待时不能被中断 lock.lockInterruptibly()

为什么wait()和notify()需要搭配synchonized关键字使用

每一个对象都有一个与之对应的监视器

每一个监视器里面都有一个该对象的锁和一个等待队列和一个同步队列

现在考虑， 如果wait() 和 notify() 的操作没有相应的同步机制， 则会发生如下情况



【线程A】 进入了 while 循环后（通过了 !condition 判断条件， 但尚未执行 wait 方法）, CPU 时间片耗尽， CPU 开始执行线程B的代码

【线程B】 执行完毕了 condition = true; notify(); 的操作， 此时【线程A】的 wait()操作尚未被执行， notify() 操作没有产生任何效果

【线程A】执行wait() 操作， 进入等待状态，如果没有额外的 notify() 操作， 该线程将持续在 condition = true 的情形下， 持续处于等待状态得不到执行。

由此看出， 在使用wait() 和 notify() 这种会挂起线程的操作时， 我们需要一种同步机制保证， condition 的检查与 wait() 操作， 以及 condition 的更新与 notify() 是互斥的。

所以 wait 和 notify 方法必须是要当前线程获得锁才可以实行的，否则会报错