**实现多线程的两种方法：Thread与Runable。**

Thread:

1，定义类继承Thread。

2，复写Thread类中的run方法

3，调用线程的start方法

Runnable

1.类A实现Runnable接口

2.覆盖run方法。

3.通过Thread类建立线程对象。

4.将类A对象作为实际参数传递给Thread类的构造函数。

不同：

Thread继承自runnable

使用runnable好处：

避免点继承的局限，一个类可以继承多个接口。

适合于资源的共享（同一个对象可作为多个thread的构造参数）

**ThreadPool用法与优势。**

ThreadPoolExecutor(corePoolSize,maximumPoolSize,keepAliveTime,milliseconds,runnableTaskQueue, handler);

corePoolSize（线程池的基本大小）

runnableTaskQueue（任务队列）：用于保存等待执行的任务的阻塞队列。

maximumPoolSize（线程池最大大小）：线程池允许创建的最大线程数。RejectedExecutionHandler（饱和策略）：当队列和线程池都满了，说明线程池处于饱和状态



优点：减少创建线程消耗的系统资源

Java通过Executors提供四种线程池，分别为：  
newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。  
newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。  
newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。  
newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

四种与ThreadPoolExecutor都是executorservice实现类

**wait()和sleep()的区别**

Object->wait thread->sleep

sleep()方法是Thread类的方法，因此它不能改变对象的机锁。

当在一个Synchronized方法中调用sleep（）时，线程虽然休眠了，但是对象的机锁没有被释放，其他线程仍然无法访问这个对象。

wait()方法则会在线程休眠的同时释放掉机锁，其他线程可以访问该对象(的**其他**synchronized方法)。

Wait() notify()只能在同步块调用:

因为wait和notify之间也存在着竞争关系。（只能有一个wait,notify不被争抢）

例子：生产者生产一个商品，消费者消费并notify,生产者wait(),这样就产生了不同步的问题

**CAS原理**

独占锁是一种悲观锁，synchronized就是一种独占锁，会导致其它所有需要锁的线程挂起，等待持有锁的线程释放锁。而另一个更加有效的锁就是乐观锁。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。

乐观锁用到的机制就是CAS，Compare and Swap。 CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。

++i实现机制：

*public final int incrementAndGet() {  
    for (;;) {  
        int current = get();//得到i的值  
        int next = current + 1;  
        if (compareAndSet(current, next))  
            return next;  
    }  
}//而compareAndSet利用JNI来完成CPU指令的操作*

*public final boolean compareAndSet(int expect, int update) {     
    return unsafe.compareAndSwapInt(this, valueOffset, expect, update);  
    }//* compareAndSwapInt就是借助C来调用CPU底层指令实现的

*i++： i. incrementAndGet*

*this就是v，expect是current，update是B*

问题：ABA一次修改第二次修改回去依旧可以执行，会有问题。

解决：控制变量值的版本来保证CAS atomicStampedRef.compareAndSet(xx)

Cpu层面也是通过锁来实现：

总线锁（处理器独占内存）和缓存锁（在操作期间保证某一变量被锁）

**Java两类锁：内置的锁和concurrent实现的一系列锁（它们的介绍与原理）**

Synchronized

ReentrantLocklock

Reentrantreadwritelock

重入锁是指同一个线程可以多次获取同一把锁 synchronized是非公平锁，而ReentrantLock的默认实现是非公平锁，但是也可以设置为公平锁

对于Synchronized来说，它是java语言的关键字，是原生语法层面的互斥，需要jvm实现,语法简单，不需要显式释放锁。而ReentrantLock它是JDK 1.5之后提供的API层面的互斥锁，需要lock()和unlock()方法配合try/finally语句块来完成，多线程时吞吐量高性能稳定，并且有很多特性。Reentrantreadwritelock适合读多写少，读时加读锁不阻塞读排斥写锁，写时排斥读。

原理：

Synchronized

每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

1、如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。

2、如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1.

3.如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

Synchronize可见性：先清空工作内存→在主内存中拷贝最新变量的副本到工作内存→执行完代码→将更改后的共享变量的值刷新到主内存中→释放互斥锁。

ReentrantLocklock:

ReentrantLock是基于AQS实现的，aqs（AbstractQueuedSynchronizer）基于cas

线程a基于cas获得锁，

线程b、c被堵塞加入队列（双向链表），基于cas创建队列，h为头，b、c为节点

将头status置为1，代表后面有线程

a结束去查看队列,head status==1,则唤醒头节点后一个任务。

ReentrantReadWriteLock

适合读多写少，

ReadLock是共享的，而WriteLock是独占的，有两个标识需要抢占。

也是基于aqs

/\*\* 返回共享锁（读锁）的数量 \*/

static int sharedCount(int c) { return c >>> SHARED\_SHIFT; }

/\*\* 返回排它锁（写锁）的数量 \*/

static int exclusiveCount(int c) { return c & EXCLUSIVE\_MASK;}

低位的16位用于保存写锁状态，而高位的16位用于保存读锁状态（保证原子性）

**锁的等级：方法锁、对象锁、类锁**

在修饰代码块的时候需要一个reference对象作为锁的对象.   
在修饰方法的时候默认是当前对象作为锁的对象.   
在修饰类(静态方法)时候默认是当前类的Class对象作为锁的对象.

ThreadLocal的设计理念与作用

当一个对象中有synchronized method或synchronized block的时候调用此对象的同步方法或进入其同步区域时，就必须先获得**对象锁**。

Concurrent包里的其他东西：ArrayBlockingQueue、CountDownLatch等等。

由于一个class不论被实例化多少次，其中的静态方法和静态变量在内存中都只有一份。所以，一旦一个静态的方法被申明为synchronized。此类所有的实例化对象在调用此方法，共用同一把锁，我们称之为**类锁**。

对象锁是用来控制实例方法之间的同步，类锁是用来控制**静态方法（或静态变量互斥体）**之间的同步。　**方法锁也是对象锁**

**Join**

某线程内调用t.join()，此线程挂起，等待t结束回复。

**Threadlocal**

ThreadLocal类用来提供线程内部的局部变量。这些变量在多线程环境下访问(通过get或set方法访问)时能保**证各个线程里的变量相对独立于其他线程内的变量**，ThreadLocal实例通常来说都是private static类型。

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

每个ThreadLocal**只有一个变量**，但是它内部包含一个map，针对每个thread保留一个entry，如果对应的thread不存在则会调用initialValue。

每个thread都有一个ThreadLocalMap，key为当前对象（一个线程可能有多个threadlocal），value为存储值

**Synchronized与volatile:**

Synchronized：保证可见性和原子性 Volatile：保证可见性，但不保证操作的原子性

解决死锁设计。

（可见性：一个线程对共享变量的修改可以即使被其他线程读取,

可见性 一：缓存，a线程读取了然后缓存，b线程修改，a线程使用的时候还是旧缓存

二：单一线程中，只要重排序不会影响到程序的执行结果，那么就不能保证其中的操作一定按照程序写定的顺序执行，即使重排序可能会对其它线程产生明显的影响。）

Volatile保证的是变量的原子性不是操作的原子性

不要将volatile用在getAndOperate场合，仅仅set或者get的场景是适合volatile的不要将volatile用在getAndOperate场合（这种场合不原子，需要再加锁），仅仅set或者get的场景是适合volatile的。int a=0;a++非原子性，因为依赖于之前的值

Volatile可见性原理：对Volatile变量进行写操作，JVM就会向处理器发送一条Lock前缀的指令，将这个变量所在**缓存行的数据写回到系统内存**。每个处理器通过嗅探在总线上传播的数据来检查自己**缓存的值是不是过期**了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器要对这个数据进行修改操作的时候，会强制重新从系统内存里把数据读到处理器缓存里。

**同步类**

AtomicLong，atomicxx用到了cas,atomic,atomicdouble还用到了volatile

Countdownlatch: public void CountDownLatch(int count) {...} 构造器中的计数值（count）实际上就是闭锁需要等待的线程数量 ，主线程必须在启动其他线程后立即调用CountDownLatch.await()方法。样主线程的操作就会在这个方法上阻塞，直到其他线程完成各自的任务。其他N 个线程必须引用闭锁对象，因为他们需要通知CountDownLatch对象，他们已经完成了各自的任务。这种通知机制是通过 CountDownLatch.countDown()方法来完成的；每调用一次这个方法，在构造函数中初始化的count值就减1

CyclicBarrier:一组任务并行执行任务，在下一个步骤之前等待直到所有任务完成。

Blockingqueue解决生产者消费者问题

Put and take

**死锁**

降低锁粒度

生产者消费者问题

消费者



生产者





While(conditionnotmet) wait()若变为 if(conditionnotmet) wait()

可能会有其他线程插足，抢走商品