并发包类实现总结

2015-11-22 23:59 feiying 0 阅读 54

JSR166中有一些常用的API，总共可以分成5类，并发数据结构，并发工具包，原子类，锁，线程池，我们写代码主要是场景，有需求才有代码，这里就是一个整理的过程，对于每一个类，根据其源码思路，对其场景进行讲述；**1.并发数据结构**

a.阻塞队列 特性为生产者消费者模式，对于取来说，有就取，没有就阻塞，对于塞入动作也是一样； BlockingQueue为实现接口，有两个子类，ArrayBlockingQueue，LinkedBlockingQueue； 上述是单端的，双端是BlockingDeque，只不过是双端都可以操作的队列而已；

	实现思路
ArrayBlockingQueue	由数组实现的队列，队头指针，队尾指针，循环利用整个数组； 锁采用的是ReentrantLock，阻塞唤醒是用Condition来做的；
LinkedBlockingQueue	由链表实现的队列，比数组实现的好处一个是可以无限延长，而ArrayBlockingQueue队列仅有固定大小，容量初始化就固定了；其二，Node链接的并发压力小，每次锁定相关2，3个节点； 锁仍旧是ReentrantLock，但是拆分锁粒度，分为put锁和get锁，唤醒操作仍旧使用Condition； 因为大小可变，所以count是原子类，是AtomInteger；
LinkedBlockingDeque	基本思路和LinkedBlockingQueue一样，也是ReentrantLock+Condition，链表实现； 只不过双端都可以操作，相对比LinkedBlockingQueue操作方法*2；
PriorityBlockingQueue	优先队列的阻塞版本，PriorityQueue，同是最小堆实现，堆顶元素是最小的，每一次堆化都会把最小的那个元素弄到堆顶，左右子树与TreeMap这种红黑树和平衡二叉排序树比起来，并不平均，仅仅保证堆顶元素； 以数组作为存储实现，也是ReentrantLock，阻塞唤醒操作是Condition；

DelayQueue	实现了Delayed延迟接口的Queue，因为根据每一个Node节点的超时时间排序，最小的那个最先出队执行，因此DelayQueue没有实现，代理PriorityQueue作为实现； 仍旧是ReentrantLock+Condition的组合，只不过使用了一个蹩脚的Leader-Following模式(貌似没啥作用)
SynchronousQueue	各种资料都说其“没有容量”，这种说法其实是完全错误的，SynchronousQueue中实现的数据结构是著名的Dual Stack(不公平)和Dual Queue(公平)，二者内部都是有Node节点存储的，只不过对于外界看来，你put了n个Node，没有人take，它会造成当前客户端的阻塞而已； JDK5中SynchronousQueue使用的是ReentrantLock+Condition的组合,JDK6以后采用了上述的双数据结构的组合+CAS无锁，效率提升不少；对于阻塞和唤醒直接采用LockSupport操纵Node节点中的Thread线程；
LinkedTransferQueue	可以理解为SynchronousQueue的灵活版本，沿用Dual Queue数据结构，提供了xfer等更灵活的方法； 而这个LinkedTransferQueue中因为开放了Dual Queue数据结构，所以外界看来是有容量的，而其实实质和 SynchronousQueue的Dual Queue是一样的，因此，getsize等方法都会返回具体的Node节点；

b.非阻塞队列

与BlockingQueue相对应的是，非阻塞队列的“非阻塞”是对应ReentrantLock这种级别的锁，因为ReentrantLock虽然功能比synchronized关键字多很多，但大体上也是 调用操作系统的锁进行休眠，操作系统的锁为什么效率低呢？那是因为从用户态到内核态每一次休眠唤醒，需要错过非常多的时钟周期，而我们主要的程序是在几个进程之内 的，我们希望我们这几个进程（甚至就是一个JVM进程）能尽可能的“耗光和榨干”整个计算机的资源，因此，我们的做法就是不断的进行再用户态空转和等待，尽可能的少做 线程的切换，让cpu干我们自己的“正事”，这种做法也就大多数无锁算法的思路，对于队列也是一样；

	实现思路
ConcurrentLinkedQueue	非阻塞单端队列，链表实现； 采用CAS乐观锁，Node节点中都是cas算法的实现，整体效率在一定场景下比较高效；
ConcurrentLinkedDeque	非阻塞双端队列，链表实现，整体思路和ConcurrentLinkedQueue基本一样

c.key-value 从util包就可得知，键值对这种结构就是HashMap的专项， 而TreeMap是排序过的红黑树， 其次，JDK7中对SortedMap扩展的NavigableMap接口是针对排序和精细查找的键值对算法； 除此之外，Map这种数据结构的key部分，是Set这种数据结构的实现，因此对于key-value的这种结构也需要将Set算进来； 对于上述的这些内容，并发包一一对key-value的各种数据结构进行实现；

	实现思路
ConcurrentHashMap	扩展了HashMap的数组-链表的两级结构，增加了槽位(默认16个)，变成了数组-槽位-链表三级结构，这种数据结构的好处就在于，每一次操作锁定的只是一个槽位，而不是锁定整个HashMap； 每一个槽继承自ReentrantLock，因此每次锁定Segment，虽然降低了并发，但是上述的数据结构造成了查找的时候，进行了二次Hash，查找效率也大打折扣；
ConcurrentSkipListMap	基于NavigableMap和ConcurrentMap双接口的实现，目的就是替代单线程的TreeMap红黑树查找Map； 采用排序查找效率极高的跳表数据结构实现，存储结构纵向是链表，每一层也是链表，用Node节点堆砌； 采用CAS乐观锁，并且只对Node节点加锁，增删改效率和ConcurrentHashMap不可同日而语； 虽然查找效率不如ConcurrentHashMap，但因为排序的关系，其和单线程的TreeMap红黑树相差无几； 此类是多线程的排序Map的最佳选择；
ConcurrentSkipListSet	沿用跳表的数据结构，其实内部就是代理ConcurrentSkipListMap的实现，多线程的排序Set首选； 只不过key存的就是Set的Entry，value存的是null，和Set这个系列的制作思路是一样；

d.顺序存储 ArrayList的多线程并发版本，影子拷贝，主要解决Fail-Fast问题； 因为Set是指不重复的一串值，而List是可以重复的，所以影子拷贝也有Set的一个版本；

	实现思路

CopyOnWriteArrayList	ArrayList的并发版本，也是基于数组来实现的，但是使用了ReentrantLock的锁，适应于并发场景； 其次，针对于Fail-Fast的迭代过程中的问题，采用snap快照的方式，允许影子读；
CopyOnWriteArraySet	Set其实是一个很有意思的接口，当它排序的时候，它代理的是Map的实现，而它其实也是个顺序序列， 这个CopyOnWriteArraySet内部其实就是CopyOnWriteArrayList的实现，只不过元素不重复而已；

2.并发工具包

util包中提供一些单线程的工具，但是并没有针对一些并发的场景提供工具，这个util.concurrent包中都是非常重要的并发同步工具。

	实现思路
CountDownLatch	倒计时类，模拟火箭发射3,2,1； 内部实现是继承自AQS的Sync内部类，利用其AQS队列中的state属性作为count数字， Node节点类型是共享锁（每个线程都可以进行倒计时工作）；
CyclicBarrier	门槛类，也称之为集合点类，和操作系统中的门槛类类似； 采用的是ReentrantLock进行锁定，阻塞唤醒使用的是Condition；

Phaser	JDK7中搞出来的一个阶段的概念，和CyclicBarrier基本类似，但将越过门槛的动作拆解为达到，等待，碰撞(onAdvance回调)等3个动作，引入了Phase和party，功能颇多，甚至可以自动注册解注册party； 内部采用一个奇偶队列(类似于AQS)抽象party在每隔2个phase之间的等待和唤醒(防止并发冲突) 奇偶队列中的Qnode为单个party，调用LockSupport实现Qnode中的Thread的等待和唤醒； 对整个队列的几个属性聚焦在一个long的64位属性中，缩小CAS的范围，提高并发效率；
Semaphore	锁是针对于单个的，而Semaphore针对于一组实体，类似于厕所n个马桶蹲坑，或者几个路口的信号灯； 内部也是继承与AQS队列的Sync，队列中的state属性就是信号灯的个数，维持在一定的数量； 只不过这里Sync类分成公平和不公平，公平就是每一次加入乖乖的放到AQS队列的末尾，进行排队的操作，而不公平无视AQS队列，上来就调用CAS方法抢锁；整体思路和ReentrantLock的公平锁实现类似；
Exchanger	并发工具包的数据交换功能，可以用在启动两个线程，然后调用Exchanger.exchange进行数据互换； JDK5中使用Stack数据结构进行模拟交换场所，但因为并发压力都在栈顶，Douglea在JDK6的一篇论文中发表算法，搞出一个多slot槽位的栈，这样栈顶交换场所就可以多个进行，整体思路类似于ConcurrentHashMap中的多slot思路，减少并发的冲突；

并发工具包中，底层的实现是AQS队列，并用到了锁包中的Reentrant和Condition，并大量的使用了CAS；**3.锁**

锁在java内置语言中就有，但是synchronized就是一个关键字，只能代表进入区域中线程进入obj的对象监视器，从而产生排他性；正因为synchronized功能太弱，而性能也差（操作系统线程休眠），因此并发包才搞出个ReentrantLock，ReadWriteLock；对于object.wait/notify也是类似，引起操作系统线程的休眠和唤醒，和前面的CAS思路一样，能不能多在用户态进行停留，导致程序尽可能多“留”一会，这就是Condition的作用了；

	实现思路

AbstractQueuedSynchronizer	著名的AQS队列,几乎是所有同步工具包和部分lock包的底层实现； 使用链表的Node节点来模拟线程并发请求情况，每一个Node对应一个线程，根据Node中属性和传播的特性，可以模拟出独占和共享，其之所以在用户态模拟是为了更大程度的能利用CAS和在用户态的空转，让尽可能多的时间留在用户态中，实在等不起了，最终才发生操作系统级别的线程休眠； AQS队列其实有两个队列，一个是上面所述的，还有一个是为了Condition准备的wait队列； AQS队列最终在用户态等不起的时候，也是调用LockSupport进行线程阻塞和唤醒；
ReentrantLock	ReentrantLock首先值得说的是，它的性能与synchronized改进不大，尽管其使用了CAS尽可能多try了几次，但是在大并发压力下，仅仅是杯水车薪； ReentrantLock主要是功能上的改进，如和Semaphore一样，继承了AQS队列的公平/不公平Sync，导致多了公平锁的特性，其次基于CAS的try，响应线程的中断，锁的灵活性，线程信息获取和监视等有很大的提升，在功能上远远胜过synchronized关键字，可以被认为是细粒度的synchronized； 除此，ReentrantLock中可以创建Condition，就如同仅在synchronized能调用obj.wait一样；
Condition	相当于并发包的obj.wait/notify，必须由ReentrantLock进行创建； 其实Condition接口的具体实现类就是AQS队列中的等待队列ConditionObject，等待队列与AQS主队列配合，来完成线程wait后，进如等待区中的情况，这几乎和synchronized中的obj.wait进入等待区的概念一模一样（事实上这就是Condition之所以这么模拟的目的），线程阻塞唤醒使用的是LockSupport；
LockSupport	线程之前的同步要么是synchronized关键字，要么是obj.wait/notify，而主体应该是Thread才对，这就是LockSupport的目的，传入哪个线程，对应就park阻塞和unpark唤醒哪个线程；非常的好用，正因如此，此类在并发包中只要有线程相关的阻塞唤醒，LockSupport都会现身； 尽管在lock的过程中，也像ReentrantLock一样try了几次CAS，但是依然杯水车薪，最终还是调用的是操作系统的mutex_lock(linux系统调用)，整体思路就是为了方便，性能提升并不明显；

ReentrantReadWriteLock	读写锁，典型的读写分离的ReentrantLock，写写互斥，读读不互斥，但读写互斥； 此类是AQS队列的最佳实践，AQS队列中的共享模式就是读锁，独占模式就是写锁，读写都存在与AQS的队列中，符合AQS队列中的传播原则，除此，还有ReentrantLock的公平，线程中断等功能；

4.原子类

多线程并发中，不一定只有一行代码能编译成几条汇编指令，甚至1个操作都有可能编译成几条汇编指令；所以，我们理解的并发一定不要只局限于语句，一个类的操作，如HashMap的put/get就不是线程安全的，即使它是一条语句而已，而原子变量也一样，这也就是i++这种情况，我们需要在这种变量的级别控制多线程并发情况；而可见性是可以通过volatile修饰变量的，但是原子性却没有，而这个atomic包就是这个作用

	实现思路
AtomicInteger	int的原子性包装； 实现其实就是通过volatile修饰i，保证get方法拿到的一定是最新的值； 增长和set使用的Unsafe的CAS方法，保证在修改的过程中在用户态进行锁定，效率极高；
AtomicIntegerArray	int[]的原子性包装，实现思路和int差不多； 区别是volatile不能修饰int数组，所以需要调用Unsafe的volatile的CAS方法，其它几乎类似；
AtomicLong	Long的原子性包装，和int类似，volatile修饰的是long，也是CAS用于原子同步实现； 注意此类可能在C++实现中，因为64位中的前后32位分两次CAS，需要在C++级别加操作系统的锁来实现；
AtomicIntegerFieldUpdater	一个需求是如何不修改一个类的声明和定义，在外界包装出一个原子性的int属性， 这就是AtomicIntegerFieldUpdater的作用了，实现很简单，必然是类的反射拿到类中对象的int属性， 然后进行包装，其余内容和AtomicInteger类似；

AtomicReference	对一个对象也需要原子化的包装，这就是AtomicReference了，能包装引用一个对象； 实现也是volatile修饰V v,Unsafe的对Reference的CAS方法调用（C++）；
AtomicStampedReference	也是对象的原子化包装，比AtomicReference增强的是能解决CAS的ABA问题； 实现也是比原有的CAS多了一个时间戳的判断，这样能校验是否是修改过，其余和AtomicReference类似；

5.线程池

线程池是JDK中重要的多线程提升，在其它语言中一般都自己实现，而因为JAVA语言的强大，在JAVA内置就提供了性能强大，功能齐全的线程池；

	实现思路
Callable	与Runable的区别在于其可以返回结果，而结果通常为FutureTask； 一般都是把实现Callable的任务传入到线程池中，并通过FutureTask进行监控；
ThreadPoolExecuter	线程池的底层实体实现，通过参数变化可以演变为上层定制的各种功能和需求的线程池； 内部实现为一个ctl的多位的AtomicInteger状态控制，减少并发冲突； 使用ReentrantLock对增加减少线程进行同步控制； 内部维护一个Queue，主要目的是为了线程池数量已满，作为缓存，当有空闲线程，从这个队列当中取等待的线程，基于不同的需求Queue的类型也不同，可以实现为ArrayBlockingQueue和SychrounousQueue； 允许通过ThreadFactory个性化定制Thread，进行DIY的操作； 线程池关闭采用了Condition，其余还有n种线程池已满配置的拒绝策略，线程创建/销毁的回调函数等；

Executors	此类是对用户开发的工具类，用于创建不同类型的线程池，目前一共4种，实际实现就是通过构造方法的参数的不同，来传入到ThreadPoolExecuter，并最终构造出各不相同的线程； 固定线程池，最大最小都是一样的，单例线程池就是1个线程，而缓存线程池是面对大量的短连接复用（web服务器小图片等需求），将LinkedBlockingQueue换成了SychronousQueue；还可以调用定时器线程池；
ScheduledThreadPoolExecutor	和ThreadPoolExecuter其实差不多，但Queue换成了DelayedQueue这种带有延迟时间特性最小堆的实现，因此在Queue中第一个出队元素总是马上要到期的线程； 值得一提的是，quartz就是用的这个ScheduledThreadPoolExecutor；
FutureTask	Future是结果集，Runable是传入给线程池的任务，而FutureTask是二者的集合体，一个对象就可以搞定传入和接收执行结果的几个操作，和线程池配合极好； FutureTask的实现也是一个AQS队列的经典案例，继承自AQS的Sync内部类中维护着的一个state，这个state就是代表着future任务执行的四种状态，future不能立刻拿到结果，所以这里就利用AQS队列的乐观CAS不断的轮询结果，效率极高，不会造成操作系统的线程休眠和等待，一直到结果返回；
CompletionService	对FutureTask的Done机制的一种定制化和回调，对应场景就是提早看Future结果； 默认的实现是ExecutorCompletionService，这个实现搞出来一个完成队列，只要有一个Future结果返回，立刻塞入这个队列中，你在回调函数中立刻就可以看出这个结果；
ForkJoinPool	JDK7做出来的一个小型的mapreduce的线程池，

JSR166中有一些常用的API，总共可以分成5类，并发数据结构，并发工具包，原子类，锁，线程池，我们写代码主要是场景，有需求才有代码，这里就是一个整理的过程，对于每一个类，根据其源码思路，对其场景进行讲述；**1.并发数据结构**

a.阻塞队列 特性为生产者消费者模式，对于取来说，有就取，没有就阻塞，对于塞入动作也是一样； BlockingQueue为实现接口，有两个子类，ArrayBlockingQueue，LinkedBlockingQueue； 上述是单端的，双端是BlockingDeque，只不过是双端都可以操作的队列而已；

	实现思路
ArrayBlockingQueue	由数组实现的队列，队头指针，队尾指针，循环利用整个数组； 锁采用的是ReentrantLock，阻塞唤醒是用Condition来做的；

LinkedBlockingQueue	由链表实现的队列，比数组实现的好处一个是可以无限延长，而ArrayBlockingQueue队列仅有固定大小，容量初始化就固定了；其二，Node链接的并发压力小，每次锁定相关2，3个节点； 锁仍旧是ReentrantLock，但是拆分锁粒度，分为put锁和get锁，唤醒操作仍旧使用Condition； 因为大小可变，所以count是原子类，是AtomInteger；
LinkedBlockingDeque	基本思路和LinkedBlockingQueue一样，也是ReentrantLock+Condition，链表实现； 只不过双端都可以操作，相对比LinkedBlockingQueue操作方法*2；
PriorityBlockingQueue	优先队列的阻塞版本，PriorityQueue，同是最小堆实现，堆顶元素是最小的，每一次堆化都会把最小的那个元素弄到堆顶，左右子树与TreeMap这种红黑树和平衡二叉排序树比起来，并不平均，仅仅保证堆顶元素； 以数组作为存储实现，也是ReentrantLock，阻塞唤醒操作是Condition；
DelayQueue	实现了Delayed延迟接口的Queue，因为根据每一个Node节点的超时时间排序，最小的那个最先出队执行，因此DelayQueue没有实现，代理PriorityQueue作为实现； 仍旧是ReentrantLock+Condition的组合，只不过使用了一个蹩脚的Leader-Following模式(貌似没啥作用)
SynchronousQueue	各种资料都说其“没有容量”，这种说法其实是完全错误的，SynchronousQueue中实现的数据结构是著名的Dual Stack(不公平)和Dual Queue(公平)，二者内部都是有Node节点存储的，只不过对于外界看来，你put了n个Node，没有人take，它会造成当前客户端的阻塞而已； JDK5中SynchronousQueue使用的是ReentrantLock+Condition的组合,JDK6以后采用了上述的双数据结构的组合+CAS无锁，效率提升不少；对于阻塞和唤醒直接采用LockSupport操纵Node节点中的Thread线程；

LinkedTransferQueue	可以理解为SynchronousQueue的灵活版本，沿用Dual Queue数据结构，提供了xfer等更灵活的方法； 而这个LinkedTransferQueue中因为开放了Dual Queue数据结构，所以外界看来是有容量的，而其实实质和 SynchronousQueue的Dual Queue是一样的，因此，getsize等方法都会返回具体的Node节点；

b.非阻塞队列

与BlockingQueue相对应的是，非阻塞队列的“非阻塞”是对应ReentrantLock这种级别的锁，因为ReentrantLock虽然功能比synchronized关键字多很多，但大体上也是 调用操作系统的锁进行休眠，操作系统的锁为什么效率低呢？那是因为从用户态到内核态每一次休眠唤醒，需要错过非常多的时钟周期，而我们主要的程序是在几个进程之内 的，我们希望我们这几个进程（甚至就是一个JVM进程）能尽可能的“耗光和榨干”整个计算机的资源，因此，我们的做法就是不断的进行再用户态空转和等待，尽可能的少做 线程的切换，让cpu干我们自己的“正事”，这种做法也就大多数无锁算法的思路，对于队列也是一样；

	实现思路
ConcurrentLinkedQueue	非阻塞单端队列，链表实现； 采用CAS乐观锁，Node节点中都是cas算法的实现，整体效率在一定场景下比较高效；
ConcurrentLinkedDeque	非阻塞双端队列，链表实现，整体思路和ConcurrentLinkedQueue基本一样

c.key-value 从util包就可得知，键值对这种结构就是HashMap的专项， 而TreeMap是排序过的红黑树， 其次，JDK7中对SortedMap扩展的NavigableMap接口是针对排序和精细查找的键值对算法； 除此之外，Map这种数据结构的key部分，是Set这种数据结构的实现，因此对于key-value的这种结构也需要将Set算进来； 对于上述的这些内容，并发包一一对key-value的各种数据结构进行实现；

	实现思路
ConcurrentHashMap	扩展了HashMap的数组-链表的两级结构，增加了槽位(默认16个)，变成了数组-槽位-链表三级结构，这种数据结构的好处就在于，每一次操作锁定的只是一个槽位，而不是锁定整个HashMap； 每一个槽继承自ReentrantLock，因此每次锁定Segment，虽然降低了并发，但是上述的数据结构造成了查找的时候，进行了二次Hash，查找效率也大打折扣；

ConcurrentSkipListMap	基于NavigableMap和ConcurrentMap双接口的实现，目的就是替代单线程的TreeMap红黑树查找Map； 采用排序查找效率极高的跳表数据结构实现，存储结构纵向是链表，每一层也是链表，用Node节点堆砌； 采用CAS乐观锁，并且只对Node节点加锁，增删改效率和ConcurrentHashMap不可同日而语； 虽然查找效率不如ConcurrentHashMap，但因为排序的关系，其和单线程的TreeMap红黑树相差无几； 此类是多线程的排序Map的最佳选择；
ConcurrentSkipListSet	沿用跳表的数据结构，其实内部就是代理ConcurrentSkipListMap的实现，多线程的排序Set首选； 只不过key存的就是Set的Entry，value存的是null，和Set这个系列的制作思路是一样；

d.顺序存储 ArrayList的多线程并发版本，影子拷贝，主要解决Fail-Fast问题； 因为Set是指不重复的一串值，而List是可以重复的，所以影子拷贝也有Set的一个版本；

	实现思路
CopyOnWriteArrayList	ArrayList的并发版本，也是基于数组来实现的，但是使用了ReentrantLock的锁，适应于并发场景； 其次，针对于Fail-Fast的迭代过程中的问题，采用snap快照的方式，允许影子读；
CopyOnWriteArraySet	Set其实是一个很有意思的接口，当它排序的时候，它代理的是Map的实现，而它其实也是个顺序序列， 这个CopyOnWriteArraySet内部其实就是CopyOnWriteArrayList的实现，只不过元素不重复而已；

2.并发工具包

util包中提供一些单线程的工具，但是并没有针对一些并发的场景提供工具，这个util.concurrent包中都是非常重要的并发同步工具。

	实现思路
CountDownLatch	倒计时类，模拟火箭发射3,2,1； 内部实现是继承自AQS的Sync内部类，利用其AQS队列中的state属性作为count数字， Node节点类型是共享锁（每个线程都可以进行倒计时工作）；
CyclicBarrier	门槛类，也称之为集合点类，和操作系统中的门槛类类似； 采用的是ReentrantLock进行锁定，阻塞唤醒使用的是Condition；
Phaser	JDK7中搞出来的一个阶段的概念，和CyclicBarrier基本类似，但将越过门槛的动作拆解为达到，等待，碰撞(onAdvance回调)等3个动作，引入了Phase和party，功能颇多，甚至可以自动注册解注册party； 内部采用一个奇偶队列(类似于AQS)抽象party在每隔2个phase之间的等待和唤醒(防止并发冲突) 奇偶队列中的Qnode为单个party，调用LockSupport实现Qnode中的Thread的等待和唤醒； 对整个队列的几个属性聚焦在一个long的64位属性中，缩小CAS的范围，提高并发效率；
Semaphore	锁是针对于单个的，而Semaphore针对于一组实体，类似于厕所n个马桶蹲坑，或者几个路口的信号灯； 内部也是继承与AQS队列的Sync，队列中的state属性就是信号灯的个数，维持在一定的数量； 只不过这里Sync类分成公平和不公平，公平就是每一次加入乖乖的放到AQS队列的末尾，进行排队的操作，而不公平无视AQS队列，上来就调用CAS方法抢锁；整体思路和ReentrantLock的公平锁实现类似；

Exchanger	并发工具包的数据交换功能，可以用在启动两个线程，然后调用Exchanger.exchange进行数据互换； JDK5中使用Stack数据结构进行模拟交换场所，但因为并发压力都在栈顶，Douglea在JDK6的一篇论文中发表算法，搞出一个多slot槽位的栈，这样栈顶交换场所就可以多个进行，整体思路类似于ConcurrentHashMap中的多slot思路，减少并发的冲突；

并发工具包中，底层的实现是AQS队列，并用到了锁包中的Reentrant和Condition，并大量的使用了CAS；**3.锁**

锁在java内置语言中就有，但是synchronized就是一个关键字，只能代表进入区域中线程进入obj的对象监视器，从而产生排他性；正因为synchronized功能太弱，而性能也差（操作系统线程休眠），因此并发包才搞出个ReentrantLock，ReadWriteLock；对于object.wait/notify也是类似，引起操作系统线程的休眠和唤醒，和前面的CAS思路一样，能不能多在用户态进行停留，导致程序尽可能多“留”一会，这就是Condition的作用了；

	实现思路
AbstractQueuedSynchronizer	著名的AQS队列,几乎是所有同步工具包和部分lock包的底层实现； 使用链表的Node节点来模拟线程并发请求情况，每一个Node对应一个线程，根据Node中属性和传播的特性，可以模拟出独占和共享，其之所以在用户态模拟是为了更大程度的能利用CAS和在用户态的空转，让尽可能多的时间留在用户态中，实在等不起了，最终才发生操作系统级别的线程休眠； AQS队列其实有两个队列，一个是上面所述的，还有一个是为了Condition准备的wait队列； AQS队列最终在用户态等不起的时候，也是调用LockSupport进行线程阻塞和唤醒；
ReentrantLock	ReentrantLock首先值得说的是，它的性能与synchronized改进不大，尽管其使用了CAS尽可能多try了几次，但是在大并发压力下，仅仅是杯水车薪； ReentrantLock主要是功能上的改进，如和Semaphore一样，继承了AQS队列的公平/不公平Sync，导致多了公平锁的特性，其次基于CAS的try，响应线程的中断，锁的灵活性，线程信息获取和监视等有很大的提升，在功能上远远胜过synchronized关键字，可以被认为是细粒度的synchronized； 除此，ReentrantLock中可以创建Condition，就如同仅在synchronized能调用obj.wait一样；

Condition	相当于并发包的obj.wait/notify，必须由ReentrantLock进行创建； 其实Condition接口的具体实现类就是AQS队列中的等待队列ConditionObject，等待队列与AQS主队列配合，来完成线程wait后，进如等待区中的情况，这几乎和synchronized中的obj.wait进入等待区的概念一模一样（事实上这就是Condition之所以这么模拟的目的），线程阻塞唤醒使用的是LockSupport；
LockSupport	线程之前的同步要么是synchronized关键字，要么是obj.wait/notify，而主体应该是Thread才对，这就是LockSupport的目的，传入哪个线程，对应就park阻塞和unpark唤醒哪个线程；非常的好用，正因如此，此类在并发包中只要有线程相关的阻塞唤醒，LockSupport都会现身； 尽管在lock的过程中，也像ReentrantLock一样try了几次CAS，但是依然杯水车薪，最终还是调用的是操作系统的mutex_lock(linux系统调用)，整体思路就是为了方便，性能提升并不明显；
ReentrantReadWriteLock	读写锁，典型的读写分离的ReentrantLock，写写互斥，读读不互斥，但读写互斥； 此类是AQS队列的最佳实践，AQS队列中的共享模式就是读锁，独占模式就是写锁，读写都存在与AQS的队列中，符合AQS队列中的传播原则，除此，还有ReentrantLock的公平，线程中断等功能；

4.原子类

多线程并发中，不一定只有一行代码能编译成几条汇编指令，甚至1个操作都有可能编译成几条汇编指令；所以，我们理解的并发一定不要只局限于语句，一个类的操作，如HashMap的put/get就不是线程安全的，即使它是一条语句而已，而原子变量也一样，这也就是i++这种情况，我们需要在这种变量的级别控制多线程并发情况；而可见性是可以通过volatile修饰变量的，但是原子性却没有，而这个atomic包就是这个作用

	实现思路
AtomicInteger	int的原子性包装； 实现其实就是通过volatile修饰i，保证get方法拿到的一定是最新的值； 增长和set使用的Unsafe的CAS方法，保证在修改的过程中在用户态进行锁定，效率极高；

AtomicIntegerArray	int[]的原子性包装，实现思路和int差不多； 区别是volatile不能修饰int数组，所以需要调用Unsafe的volatile的CAS方法，其它几乎类似；
AtomicLong	Long的原子性包装，和int类似，volatile修饰的是long，也是CAS用于原子同步实现； 注意此类可能在C++实现中，因为64位中的前后32位分两次CAS，需要在C++级别加操作系统的锁来实现；
AtomicIntegerFieldUpdater	一个需求是如何不修改一个类的声明和定义，在外界包装出一个原子性的int属性， 这就是AtomicIntegerFieldUpdater的作用了，实现很简单，必然是类的反射拿到类中对象的int属性， 然后进行包装，其余内容和AtomicInteger类似；
AtomicReference	对一个对象也需要原子化的包装，这就是AtomicReference了，能包装引用一个对象； 实现也是volatile修饰V v,Unsafe的对Reference的CAS方法调用（C++）；
AtomicStampedReference	也是对象的原子化包装，比AtomicReference增强的是能解决CAS的ABA问题； 实现也是比原有的CAS多了一个时间戳的判断，这样能校验是否是修改过，其余和AtomicReference类似；

5.线程池

线程池是JDK中重要的多线程提升，在其它语言中一般都自己实现，而因为JAVA语言的强大，在JAVA内置就提供了性能强大，功能齐全的线程池；

	实现思路
Callable	与Runable的区别在于其可以返回结果，而结果通常为FutureTask； 一般都是把实现Callable的任务传入到线程池中，并通过FutureTask进行监控；

ThreadPoolExecuter	线程池的底层实体实现，通过参数变化可以演变为上层定制的各种功能和需求的线程池； 内部实现为一个ctl的多位的AtomicInteger状态控制，减少并发冲突； 使用ReentrantLock对增加减少线程进行同步控制； 内部维护一个Queue，主要目的是为了线程池数量已满，作为缓存，当有空闲线程，从这个队列当中取等待的线程，基于不同的需求Queue的类型也不同，可以实现为ArrayBlockingQueue和SychrounousQueue； 允许通过ThreadFactory个性化定制Thread，进行DIY的操作； 线程池关闭采用了Condition，其余还有n种线程池已满配置的拒绝策略，线程创建/销毁的回调函数等；
Executors	此类是对用户开发的工具类，用于创建不同类型的线程池，目前一共4种，实际实现就是通过构造方法的参数的不同，来传入到ThreadPoolExecuter，并最终构造出各不相同的线程； 固定线程池，最大最小都是一样的，单例线程池就是1个线程，而缓存线程池是面对大量的短连接复用（web服务器小图片等需求），将LinkedBlockingQueue换成了SychronousQueue；还可以调用定时器线程池；
ScheduledThreadPoolExecutor	和ThreadPoolExecuter其实差不多，但Queue换成了DelayedQueue这种带有延迟时间特性最小堆的实现，因此在Queue中第一个出队元素总是马上要到期的线程； 值得一提的是，quartz就是用的这个ScheduledThreadPoolExecutor；
FutureTask	Future是结果集，Runable是传入给线程池的任务，而FutureTask是二者的集合体，一个对象就可以搞定传入和接收执行结果的几个操作，和线程池配合极好； FutureTask的实现也是一个AQS队列的经典案例，继承自AQS的Sync内部类中维护着的一个state，这个state就是代表着future任务执行的四种状态，future不能立刻拿到结果，所以这里就利用AQS队列的乐观CAS不断的轮询结果，效率极高，不会造成操作系统的线程休眠和等待，一直到结果返回；

CompletionService	对FutureTask的Done机制的一种定制化和回调，对应场景就是提早看Future结果； 默认的实现是ExecutorCompletionService，这个实现搞出来一个完成队列，只要有一个Future结果返回，立刻塞入这个队列中，你在回调函数中立刻就可以看出这个结果；
ForkJoinPool	JDK7做出来的一个小型的mapreduce的线程池，