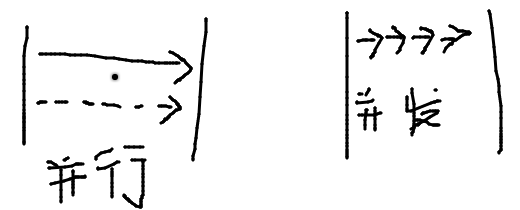
同步和异步通常用来形容一次方法调用。同步方法调用一旦开始，调用者必须等到方法调用返回后，才能继续后续的行为。异步方法调用更像一个消息传递，一旦开始，方法调用就会立即返回，调用者就可以继续后续的操作。而异步方法通常会在另外一个线程中“真实”地执行。

并发偏重于多个任务交替执行；而多个任务之间有可能还是串行的。并行是真正意义的“同时执行”。



毕竟一个cpu一次只能执行一条指令，这种情况下多进程或者多线程就是并发的，而不是并行的（操作系统会不停切换多个任务）。真实的并行也只可能出现在拥有多个cpu的系统中（比如多核CPU）。多cpu并行，单cpu并发。

临界区：临界区用来表示一种公共资源或者说是共享数据，可以被多个线程使用。但是每一次，只能有一个线程使用它，一旦临界区资源被占用，其他线程要想使用这个资源，就必须等待。

阻塞和非阻塞通常用来形容多线程间的相互影响。比如一个线程占用了临界区资源，那么其他所有需要这个资源的线程就必须在这个临界区中进行等待。等待会导致线程挂起，这种情况就是阻塞。

饥饿是指某一个或者多个线程因为无法获得所需要的资源，导致一直无法执行。比如它的线程优先级可能太低，而高优先级的线程不断抢占它需要的资源，导致低优先级线程无法工作。

阻塞：一个线程是阻塞的，那么在其他线程释放资源之前，当前线程无法继续执行。当我们使用synchronized关键字，或者重入锁时，我们得到的就是阻塞的线程。无论是synchronized或者重入锁，都会试图在执行后续代码前，得到临界区的锁，如果得不到，线程就会被挂起等待，直到占有了所需资源为止。

原子性：原子性是指一个操作是不可以中断的。即使是在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程干扰。

线程：

终止线程，线程中的stop()方法太过于暴力，强行把执行到一半的线程终止，可能会引起一些数据不一致的问题。

Thread.stop()方法 在结束线程时，会直接终止线程，并且会立即释放这个线程所持有的锁。而这些锁恰恰是用来维持对象一致性的。

正确的退出方法可以定义一个变量，当它为某一个值时退出程序,while(true){ if(变量为某值为退出)}.

严格地讲，线程中断并不会使线程立即退出，而是给线程发送一个通知，告知目标线程，有人希望你退出啦！至于目标线程接到通知后如何处理，则完全由目标线程自行决定。这点很重要，如果中断后，线程立即无条件退出，我们就又会遇到stop()方法的老问题。

public vloid Thread.interrupt() //中断线程

public boolean Thread.isInterrupted() //判断是否被中断

public static boolean Thread.interrupted() //判断是否被中断，并清除当前中断状态

Thread.interrupt()方法是一个实例方法。它通知目标线程中断，也就是设置中断标志位。中断标志位表示当前线程已经被中断了。Thread.isInterrupted()方法也是实例方法，它判断当前线程是否有被中断（通过检查中断标志位）。最后的静态方法Thread.interrupted()也是用来判断当前线程的中断状态，但同时会清除当前线程的中断标志位状态。

使用Thread.isInterrupted()函数判断当前线程是否被中断了，如果是，则退出循环体，结束线程。

Thread.sleep()方法会让当前线程休眠若干时间，它会抛出一个InterruptedException中断异常。InterruptedException不是运行时异常，也眩光是说程序必须捕获并且处理它，当线程在sleep()休眠时，如果被中断，这个异常就会产生。

执行了Thread.interrupt()方法再次中断自己，置上中断标记位，只有catch里再次中断，这样做才能发现当前线程已经被中断。

Thread.sleep()方法由于中断而抛出异常，此时，它会清除中断标记，如果不加处理，那么在下一次循环开始时，就无法捕获这个中断，故在异常处理中，再次设置中断标记位。

当在一个对象实例上调用wait()方法后，当前线程就会在这个对象上等待。比如，线程A中，调用了object.wait()方法 ，那么线程A就会停止继续执行，而转为等待状态。线程A会一直等到其他线程调用object.notify()方法为止。这时，object对象就成为了多个线程之间的有效通信手段。

如果一个线程调用了object.wait()，那么它就会进入object对象的等待队列。这个等待队列中，可能会有多个线程，因为系统运行多个线程同时等待某一个对象。当object.notify()被调用时，它就会从这个等待队列中，随机选择一个线程，并将其唤醒。notifyAll()会唤醒在这个等待队列中所有等待的线程。

假如有2个线程t1,t2;t2执行了notify()方法尝试唤醒一个等待线程，这里假设唤醒了t1.t1在被唤醒后，要做的第一件事并不是执行后续的代码，而是要尝试重新获得object的监视器；而这个监视器也正是t1在 wait()方法执行前所持有的那个。

Object.wait()和Thread.sleep()方法都可以让线程等待若干时间，除了wait()可以被唤醒外，另外一个主要区别就是wait()方法会释放hhsfcfqjr锁。而Thread.sleep()方法不会释放任何资源。

线程挂起(suspend)和继续执行(resume).不推荐使用suspend()去挂起线程的原因，是因为suspend()在导致线程暂停的同时，并不会去释放任何锁资源。直到对应的线程上进行了resume()操作，被挂起的线程才能继续，从而其他所有阻塞在相关锁上的线程也可以继续执行。但是，如果resume()操作意外地在suspend()前就执行了，那么被挂起的线程可能很难有机会被继续执行。更严重的是：它所占用的锁不会被释放，因此可能会导致整个系统工作不正常。

Thread.join()等待此线程结束。Thread.yield()让出线程，它会使当前线程让出cpu，但要注意，让出cpu并不代表当前线程不执行了。当前线程在让出cpu后，还会进行 cpu资源的争夺，但是是否能再次分配到就不一定了。一个线程的输入可能非常依赖于另一个或多个线程的输出，此时，这个线程就需要等待依赖线程执行完毕，才能继续执行。t1.join()其他线程等待，此线程执行一个人执行。

volatile与java内存模型：

当你用volatile去申明一个变量时，就等于告诉了虚拟机，这个变量极有可能会被某些者线程修改。为了确保这个变量被修改后，应用程序范围内的所有线程都能够”看到“这个改动，虚拟机就必须采用一些特殊的手段，保证这个变量的可见性等特点。

如果不使用volatile申明变量，那么这个变量被修改后，其他线程可能并不会被通知到，甚至在别的线程中，看到变量的修改顺序都会是反的。应该怎么处理才能保证每次写进去的数据不坏呢？最简单的一种方法就是加入volatile申明，告诉编译器，这个类型你要注意，因为它会不断地被修改。

volatile对于保证操作的原子性是有非常大的帮助的，但是注意的是，volatile并不能代替锁，它也无法保证一些复合操作的原子性。 volatile的变量可能被多个线程所见。volatile也能保证数据的可见性和有序性。

在一个系统中，如果线程数量很多，而且功能分配比较明确，就可以将相同功能的的线程放置在一个线程组里。

ThreadGroup tg = new ThreadGroup('xxx');

Thread t1 = new Thread(tg, ...);

线程组还有一个值得注意的方法stop()，它会停止线程组中所有的线程。

守护线程是一种特殊的线程，就和它的名字一样，它是系统的守护者，在后台默默地完成一些系统性的服务，比如垃圾回收线程，JIT线程就可以理解为守护线程。与之相对应的是用户线程，用户线程可以认为是系统的工作线程，它会完成这个程序应该要完成的业务操作。用户线程结束，守护线程也同样结束。

volatile并不能真正保证线程安全，它只能确保一个线程修改了数据后，其他线程能够看到这个改动。但当两个线程同时修改某一个数据时，却依然会产生冲突。

关键字synchronized可以有多种用法：

指定加锁对象，对给定对象加锁，进入同步代码前要获得给定对象的锁。

直接作用于实例方法：相当于对当前实例加锁，进入同步代码前要获得当前实例的锁。

直接作用于静态方法：相当于对当前类加锁，进入同步代码前要获得当前类的锁。

重入锁可以完全替代synchronized关键字。重入锁使用ReentrantLock类来实现。

重入锁对逻辑控制的灵活性要远远好于synchronized,在退出临界区时，必须记得释放锁。这种锁是可以反复进入的，这里的反复仅仅局限于一个线程。一个线程连续两次获得同一把锁。这是允许的，如果不允许这么操作，那么同一个线程在第2次获得锁时，将会和自己产生死锁。如果同一个线程多次获得锁，那么在释放锁的时候，也必须释放相同的次数。如果释放锁的次数多，那么会是得到一个java.lang.IllegalMonitorStateException异常，反之，如果释放锁的次数少了，那么相当于线程还持有这个锁，因此，其他线程也无法进入临界区。重入锁可以提供中断处理的能力。

使用重入锁，则提供另外一种可能，那就是线程可以被中断。也就是在等待锁的过程中，程序可以根据需要取消对锁的请求。lock.lockInterruptibly();

除了等待外部通知之外，要避免死锁还有另外一种方法，那就是限时等待。

lock.tryLock(5, timeUnit.SECONDS);//5少后申请锁。

ReentrantLock.tryLock()方法也可以不带参数直接运行。当前线程会尝试获得锁，如果锁并没有被其他线程占用，则申请锁会成功，立即返回true.如果锁被其他线程占用，则当前线程不会进行等待，而是立即返回false;

公平锁：在大多数情况下，锁的申请都是非公平的。线程1首先请求了锁A，接着线程2也请求了锁A.那么当锁A可用时，是线程1可以获得锁还是线程2可以获得锁呢？这是不一定的。系统只会从这个锁的等待队列中随机挑选一个。ReentrantLock(boolean fair); fair为true表示公平锁。公平锁的实现成本比较高，性能低，如果没有特别的需求，也不需要使用公平锁。

ReentrantLock有几个重要的方法：

lock(): 获得锁，如果锁已经被占用，则等待；

lockInterruptibly():获得锁，但优先响应中断；

tryLock():尝试获得锁，如果成功，返回true,失败返回false.此方法不等待，立即返回。

tryLock(long time,TimeUnit unit)：在给定时间内尝试获得锁；

unlock():释放锁；

重入锁的好搭档：Condition条件

通过Lock接口的Condition newCondition()方法可以生成一个与当前重入锁绑定的Condition实例。几个接口如下：

await()方法会使当前线程等待，同时释放当前锁，当其他线程中使用signal()或者signalAll()方法时，线程会得新获得锁并继续执行。或者当线程被中断时，也能跳出等待。这和Object.wait()方法相似。

awaitUninterruptibly()方法与await()方法基本相同，但是它并不会在等待过程中响应中断。

singal()方法用于唤醒一个在等待中的线程。相对的singalAll()方法会唤醒所有在等待的线程。

允许多个线程同时访问：信号量(Semaphore)

而信号量却可以指定多个线程，同时访问某一个资源。

在构造信号量对象时，必须要指定信号量的准入数，即同时能申请多个许可。当每个线程每次只申请一个许可时，这就相当于指定了同时有多少个线程可以访问某一个资源。

acquire()方法尝试获得一个准入的许可。若无法获得，则线程会等待，直到有线程释放一个许可或者当前线程被中断。acquireUninterruptibly()方法和acquire()方法类似，但是不响应中断。tryAcquire()尝试获得一个许可，如果成功返回true,失败则返回false,它不会进行等待，立即返回。release()用于在线程访问资源结束后，释放一个许可，以使其他等待许可的线程可以进行资源访问。

ReadWriteLock读写锁：

它是读写分离锁，可以有效地帮助减少锁竞争，以提升系统性能。用锁分离的机制来提升性能非常容易理解，比如线程A1,A2,A3进行写操作，B1,B2,B3进行读操作， 如果使用重入锁或者内部锁，则理论上说所有读之间，读与写之间，写和写之间都是串行操作。当B1进行读取时，B2，B3则需要等待。读写锁允许多个线程同时读，使得B1,B2,B3之间真正并行。写写操作和读写操作依然是需要相互等待和持有锁的。

读-读不互斥：读读之间不阻塞；

读-写互斥：读阻塞写，写也会阻塞读；

写-写互斥：写写阻塞；

如果系统中，读操作次数远远大于写操作，则读写锁就可以发挥最大的功效，提升系统的性能。

倒计时器：CountDownLatch

CountDownLatch是一个非常实用的多线程控制工具类。假设一个CountDownLatch实例，计数数量为10.这表示需要有10个线程完成任务，等待在CountDownLatch上的线程才能继续执行。

循环栅栏:\*

CyclicBarrier是另外一种多线程并发控制实用工具。和CountDownLatch非常类似，它也可以实现线程间的计数等待，但它的功能比CountDownLatch更加复杂强大。CyclicBarrier可以理解为循环栅栏。栅栏是一种障碍物。 多个线程拥有同一个CyclicBarrier变量。

场景：比如司令下达命令，要求10个士兵一起去完成一项任务。这时就会要求10个士兵先集合报道，接着，一起去完成任务（CyclicBarrier.await()等待其他线程集合）。只有士兵完成任务后，司令才宣布任务完成。

线程阻塞工具类:LockSupport

LockSupport是一个非常方便实用的线程阻塞工具，它可以在线程内任意让线程阻塞。和Thread.suspend()相比，它弥补了由于resume()在前发生，导致线程无法继续执行的情况。LockSupport的静态方法park()可以阻塞当前线程，类似的还有parkNanos(),parkUntil()等方法。它们实现了一个限时的等待。这里只是将原来的suspend()和resume()方法用park()和unpark()方法做了替换。park()类似暂停挂起，unpark()类似恢复。LockSupport类使用类似信号量的机制。它为每一个线程准备了一个许可，如果许可可用，那么park()函数会立即返回，并且消费这个许可（也就是将许可变为不可用），如果许可不可用，就会阻塞。而unpark()则使得一个许可变为可用（但是和信号量不同的是，许可不能累加，你不可能拥有起过一个许可，它永远只有一个）。除了定时阻塞功能外，LockSuport.park()还能支持中断影响。但是和其他接收中断的函数很不一样，LockSupport.park()不会抛出InterruptedException异常。它只是会默默的返回，但是我们可以从Thread.interrupted()等方法获得中断标记。

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。 可以根据时间需要对线程进行调度。

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。

超负载了怎么办：拒绝策略

ThreadPoolExecutor的最后一个参数指定了拒绝策略。也就是当任务数量超过系统实际承载能力时，要如何处理？这时就要用到拒绝策略。

Jdk内置的拒绝策略如下：

AbortPolicy策略：该策略会直接抛出异常，阻止系统正常工作。

CallerRunsPolicy策略：只要线程池未关闭，该策略直接在调用者线程中，运行当前被丢弃的任务。显然这样做不会真的丢弃任务，但是，任务提交线程的性能极有可能会急剧下降。

DiscardOledestPolicy策略：将丢弃最老的一个请求，也就是即将执行的一个任务，并尝试再次提交当前任务。

DiscardPolicy策略：此策略默默地丢弃无法处理的任务，不予任何处理。如果允许任务丢失，我觉得这可能是最好的一种方案了。

以上内置的策略均实现了RejectedExecutionHandler接口，若以上策略仍无法满足实际应用需要，完全可以自己扩展RejectedExecutionHandler接口。

ThreadPoolExecutor也是一个可以扩展的线程池。它提供了beforeExecute(),afterExecute()和terminated()三个接口对线程池进行控制。

自定义线程池只需要实现ThreadPoolExecutor.

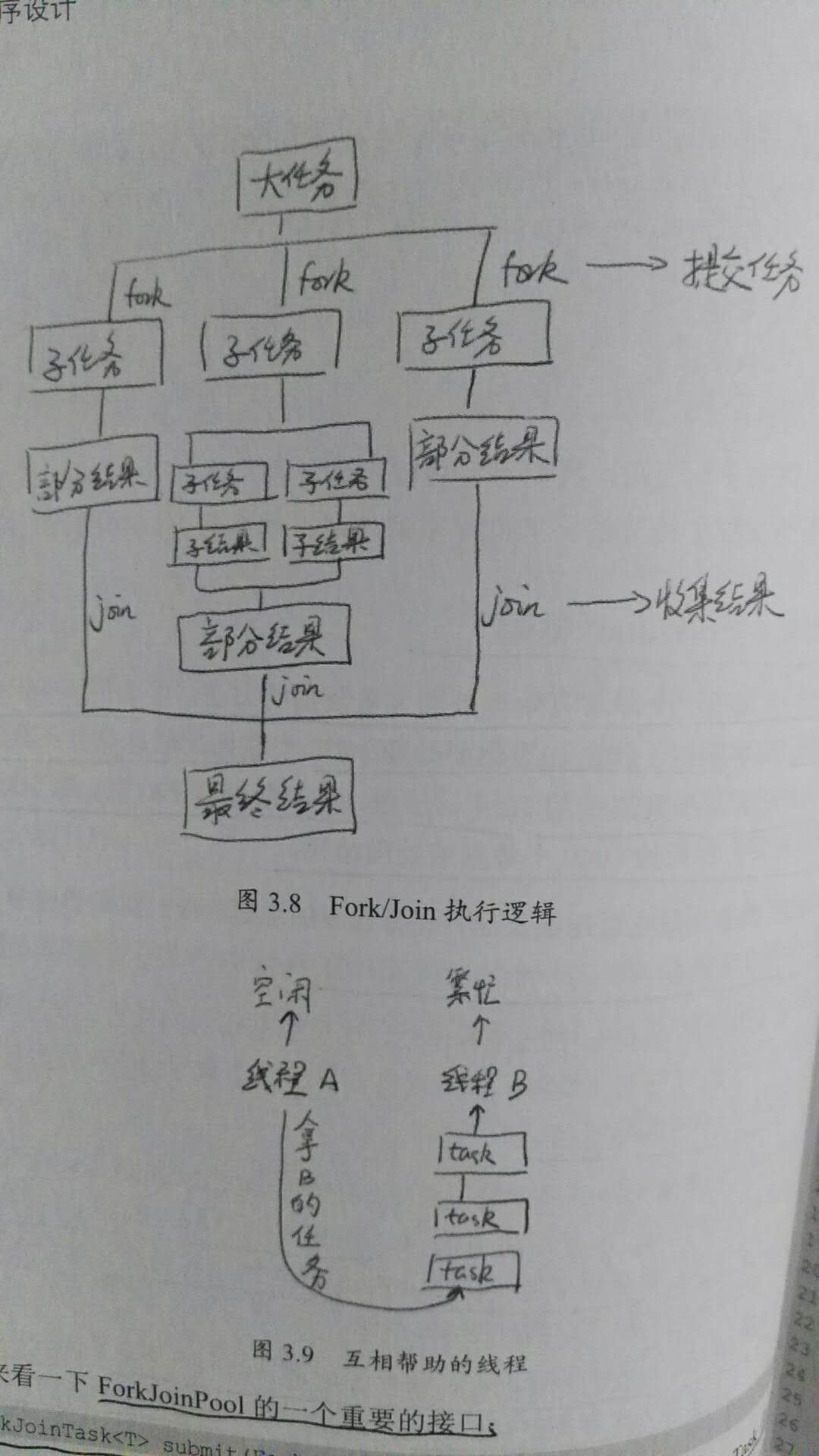
线程池的大小对系统的性能有一定的影响。过大或者过小的线程数量都无法发挥最优的系统性能，但是线程池大小的确定也不需要做得非常精确，因为只要避免极大和极小两种情况，线程池的大小对系统的性能并不会影响太大。确定线程池的大小需要考虑cpu数量，内存大小等因素。

执行线程时推荐使用execute()代替submit()，因为执行execute()出错时会得到错误的堆栈信息。

分而治之：Fork/Join框架

“分而治之”一直是一个非常有效地处理大量数据的方法。简单来说，就是如果你要处理1000个数据，但是你并不具备处理1000个数据的能力，那么你可以只处理其中的10个，然后，分阶段处理100次，将100次的结果进行合成，那就是最终想要的对原始1000个数据的处理结果。在Linux平台中，函数fork()用来创建子进程，使得系统进程可以多一个执行分支。join()表示等待。在jdk中，给出一个ForkJoinPool线程池，对于fork()方法并不急着开启线程，而是提交给ForkJoinPool线程池进行处理，以节省系统资源。

由于线程池的优化，提交的任务和线程数量并不是一对一的关系。在绝大多数情况下，一个物理线程实际上是需要处理多个逻辑任务的。因此，每个线程必然需要拥有一个任务队列。因此，在实际执行过程中，可能遇到这么一种情况：线程A已经把自己的任务都执行完了，而线程B还有一堆任务等着处理，此时，线程A就会“帮助”线程B，从线程B的任务队列中拿一个任务过来处理，尽可能达到平衡。当线程试图帮助别人时，总是从任务队列的底部开始拿数据，而线程试图执行自己的任务时，则是从相反的顶部开始拿。因此这种行为也十分有利于避免数据竞争。



重要接口: public <T> ForkJoinTask<T> submit(ForkJoinTask<T> task)

ForkJoinTask的任务就是支持fork()分解以及join()等待任务 。ForkJoinTask有两个重要的子类，RecursiveAction和RecursiveTask().它们分别表示没有返回值的任务和可以携带返回值的任务 。

在使用ForkJoin时需要注意，如果任务的划分层次很深，一直得不到返回，那么可能出现两种情况：第一，系统内的线程数量越积越多，导致性能严重下降。第二，函数的调用层次变得很深，最终导致栈溢出。ForkJoin线程池使用一个无锁的栈来管理空闲线程。如果一个工作线程暂时取不到可用的任务，则可能会被挂起，挂起的线程将会被压入由线程池维护的栈。待将来有任务可用时，再从栈中唤醒这些线程。

并发集合：

ConcurrentHashMap:这是一个高效的并发HashMap。线程安全的HashMap;

CopyOnWriteArrayList: 这是一个List,从名字看就是和ArrayList是一族的。在读多写少的场合，这个List的性能非常好，远远好于Vector.

ConcurrentLinkedQueue:高效的并发队列，使用链表实现，可以看做一个线程安全的LinkedList;

BlockingQueue:这是一个接口，JDK内部通过链表，数组等方式实现了这个接口。表示阻塞队列，非常适合用于作为数据共享的通道。

ConcurrentSkipListMap:跳表的实现。这是一个Map,使用跳表的数据结构进行快速查找。

如果要使一个HashMap变为线程安全的怎么做？Collections.synchronizedMap(new HashMap())方法可以。Collections.synchronizedMap()会生成一个名为SynchronizedMap()的Map。它使用委托，将自己所有的Map相关的功能交给传入的HashMap实现，而自己则主要负责保证线程安全。在内部实现中用一个对象作为锁，将里面的方法执行锁操作。如果并发级别不高，一般也够用。但是，在高并发环境中，我们也有必要寻求新的解决方案。同样ArrayList也有对应的Collections.synchronizedList();

高效读写的队列：ConcurrentLinkedQueue

在jdk中提供了一个ConcurrentLinkedQueue类用来实现高并发的队列。ConcurrentLinkedQueue应该算是在高并发环境中性能最好的队列。

高效读取：不变模式下的CopyOnWriteArrayList

在很多场景下读操作可能会远远大于写操作。希望读操作尽可能的快，写操作慢一点也没有关系。对它来说，读取是完全全不用加锁的，写入也不会阻塞读操作。只有写入和写入之间需要进行同步等待。所谓CopyOnWrite就是在写入操作时，进行一次自我复制。换句话说，当这个List需要修改时，我并不修改原有的内容（这对于保证当前在读线程的数据一致性非常重要），而是对原有的数据进行一次复制，将修改的内容写入副本中。写完之后，再将修改完的副本替换原来的数据。

数据共享通道：BlockingQueue

如果进行多个线程之间的数据共享？线程A希望给线程B发一个消息，用什么方式告知线程B是合理的？ArrayBlockingQueue和LinkedBlockingQueue。ArrayBlcokingQueue是基于数组实现的，而LinkedBlockingQueue是基于链表。ArrayBlockingQueue更适合做有界队列，因为队列中可容纳的最大元素需要在队列创建时指定（毕竟数组的动态扩展不太方便）。而LinkedBlockingQueue适合做无界队列，或者那些边界值非常大的队列，因为其内部元素可以动态增加，它不会因为初值容量很大，而一口气吃掉你一大半的内存。

而BlockingQueue之所有适合作为数据共享的通道，其关键还在于Blocking上。Blocking是阻塞的意思，当服务线程（服务线程不断获取队列中的消息，进行处理的线程）处理完成队列中所有的消息后，它如何知道下一条消息何时到来呢？ BlockingQueue它会让服务线程在队列为空时，进行等待，当有新的消息进入队列后，自动将线程唤醒。向队列中压入元素可以使用offer()方法和put()方法。对于offer()方法，如果当前队列已经满了，它就会立即反回false.如果没有满，则执行正常的入队操作。put()方法也是将元素压入队列末尾。但如果队列满了，它会一直等待，直到队列中有空闲的位置。从队列中弹出元素可以使用poll()方法和take()方法。如果队列为空poll()方法就直接返回null,而take()方法会等待，直到队列内有可用的元素。put()方法和take()方法者是体现Blocking()的关键。

随便数据结构：跳表SkipList

跳表是一种可以用来快速查找的数据结构，有点类似于平衡树。它们都可以对元素进行快速的查找。跳表的元素是有序的。ConcurrentSkipListMap.

锁的优化：

避免死锁，减小锁的粒度，锁分离等 。

在单核cpu上，采用并行算法的效率一般要低于原始的串行算法的，其根本原因也在于此。并行计算之所以能够提高系统的性能，是因为并行计算可以更合理的进行任务调度，充分利用各个cpu资源。

减少锁持有时间：如果线程持有锁的时间很长，锁的竞争程度也就越激烈。应该尽可能地减少对某个锁的占有时间，以减少线程间互斥的可能。如果有3个方法，只有其中一个方法需要同步，那就不要用synchronized来包3个方法，包需要同步的方法即可。只有在必要时进行同步，这样就能明显减少线程持有锁的时间，提高系统的吞吐量。减少锁的持有时间有助于降低锁冲突的可能性，进而提升系统的并发能力。

减小锁粒度：

对于ConcurrentHashMap,它内部进一步细分了若干个小的HashMap,称之为段(segment).默认情况下一个ConcurrentHashMap被进一步细分为16个段。如果需要在ConcurrentHashMap中增加一个新的表项，并不是将整个HashMap加锁，而是首先根据hashcode得到该表项应该被存放在哪个段中，然后对该段加锁，并完成put()操作。在多线程环境中，如果多个线程同时进行put()操作，只要被加入的表项不存放在同一个段中，则线程间便可以做到真正的并行。减少锁粒度会引入一个新的问题，即：当系统需要取得全局锁时，其消耗的资源会比较多。仍然以ConcurrentHashMap类为例，虽然其put()方法很好地分离了锁，但是当试图访问ConcurrentHash全局信息时，就会需要同时取得所有段的锁方能顺利实施。比如ConcurrentHashMap的size()方法，它将返回ConcurrentHashMap的有效表项的数量，即ConcurrentHashMap的全部有效表项之和。要获得这个信息需要取得所有子段的锁。

所谓减少锁粒度，就是指缩小锁定对象的范围，从而减少锁冲突的可能性，进而提高系统的并发能力。

读写分离锁来替换独占锁：

使用读写分离锁来替代独占锁是减小锁粒度的一种特殊情况。在大部分情况下，应该可以允许多线程同时读，读写锁正是实现了这种功能。在读多写少的场合，使用读写锁可以有效提升系统的并发能力。

锁分离：如果将读写锁的思想做进一步的延伸，就是锁分离。读写锁根据读写操作功能上的不同进行了有效的锁分离。依据应用程序的功能特点，使用类似的分离思想，也可以对独占锁进行分离。一个典型的案例就是LinkedBlockingQueue的实现。在LinkedBlockingQueue的实现中，take()函数和put()函数分别实现了从队列中取得数据和往队列中增加数据的功能。虽然两个函数都对当前队列进行了修改操作，但由于LinkedBlocingQueue是基于链表的，因此，两个操作分别作用于队列的前端和尾端。从理论上说，两者不冲突。

为了保证多线程间的有效并发，会要求每个线程持有锁的时间尽量短，即在使用完公共资源后，应该立即释放锁。

AtomicInteger它是可变的，并且是线程安全的。

AtomicReference表示对象引用，与AtomicInteger相似，不同之处就在于AtomicInteger是对整数的封装，而AtomicReference则对应普通的对象引用。也就是它可以保证你在修改对象引用时的线程安全性。

带有时间戳的对象引用：AtomicStampedReference

当AtomicStampedReference对应的数值被修改时，除了更新数据本身外，还必须要更新时间戳。当AtomicStampedReference设置对象值时，对象值以及时间戳都必须满足期望值，写入才会成功。即使对象值被反复读写，写回原值，只要时间戳发生变化，就能防止不恰当的写入。

数组也能无锁:AtomicIntegerArray

让普通变量也享受原子操作:AtomicIntegerFieldUpdater

因为在原子包里还有一个实用的工具类AtomicIntegerFieldUpdater.它可以让你在不改动（或者极少改动）原有代码的基础上，让变通的变量也享受CAS操作带来的线程安全性，这样你可以修改极少的代码，来获得线程安全的保证。这个Updater有三种，分别是AtomicIntegerFieldUpdater,AtomicLongFieldUpdater和AtomicReferenceFieldUpdater.

让线程之间互相帮助：SynchronousQueue，这是一个特殊的等待队列SynchronousQueue，它的容量是0,任何一个对SynchronousQueue的写需要等待一个对SynchronousQueue的读，反之亦然。因此SynchronousQueue与其说是一个队列，不如说是一个数据交换通道。必须执行一次put后再执行一次take，两者交替执行，不能同时执行多次put或take,可以用在生产者和消费者模型中。现在用transfer()方法来代替它们了。SynchronousQueue内部会维护一个线程等待队列。等待队列中会保存等待线程以及相关数据的信息。比如，生产者将数据放入SynchronousQueue时，如果没有消费者接收，那么数据本身和线程对象都会打包在队列中等待（因为SynchronousQueue容积为0，没有数据可以正常放入）。

transfer()函数的实现如下：

1.如果等待队列为空，或者队列中节点的类型和本次操作是一致的，那么将当前操作压入队列等待。比如，等待队列中是读线程等待，本次操作也是读，因此这两个读都需要等待。进入等待队列的线程可能会被挂起，它们会等待一个”匹配“操作。

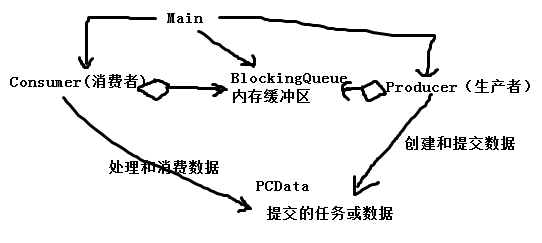
2.如果等待队列中的元素和本次操作是互补的（比如等待操作是读，而本次操作是写），那么就插入一个”完成“状态的节点，并且让他”匹配“到一个等待节点上。接着弹出这两个节点，并且使得对应的两个线程继续执行。

3.如果线程发现等待队列的节点就是”完成“节点，那么帮助这个节点完成任务。其流程步骤和1,2是一样的。

生产者-消费者模式：

生产者-消费者模式中的内存缓存区的主要功能是数据在多线程间的共享，此外，通过该缓冲区，可以缓解生产者和消费者的性能差。

|  |  |
| --- | --- |
| 角色 | 作用 |
| 生产者 | 用于提交用户请求，提取用户任务，并装入内存缓冲区 |
| 消费者 | 在内存缓冲区中提取并处理任务 |
| 内存缓冲区 | 缓存生产者提交的任务或数据，供消费者使用 |
| 任务 | 生产者向内存缓冲区提交的数据结构 |
| Main | 使用生产者和消费者的客户端 |



BlockingQueue充当了共享内存缓冲区，用于维护任务或数据队列(PCDATA).生产者对象和消费者对象均引用同一个BlockingQueue实例。生产者负责创建PCDATA对象，并将它加入BlockingQueue中，消费者则从BlockingQueue队列中获取PCDATA.

高性能的生产者-消费者：无锁的实现

BlockingQueue用于实现生产者和消费者是一个不错的选择。它可以很自然地实现作为生产者和消费者的内存缓冲区。但是BlockingQueue并不是一个高性能的实现，它完全使用锁和阻塞等待来实现线程间的同步。ConcurrentLinkedQueue是一个高性能的队列，但是BlockingQueue只是为了方便数据共享。

-------------------------------------

无锁框架：Disruptor

我们知道创建线程的方式有两种，一种是实现Runnable接口，另一种是继承Thread，但是这两种方式都有个缺点，那就是在任务执行完成之后无法获取返回结果，那如果我们想要获取返回结果该如何实现呢？还记上一篇Executor框架结构中提到的Callable接口和Future接口吗？，是的，从JAVA SE 5.0开始引入了Callable和Future，通过它们构建的线程，在任务执行完成后就可以获取执行结果，今天我们就来聊聊线程创建的第三种方式，那就是实现Callable接口。

<http://blog.csdn.net/javazejian/article/details/50896505>

java 并行流水线：<http://blog.csdn.net/jyxmust/article/details/70604206>

[java并行搜索 ：](http://blog.csdn.net/jyxmust/article/details/70604206)<http://blog.csdn.net/yangchangyong0/article/details/50528930>

CompletableFuture类实现了CompletionStage和Future接口。Future是Java 5添加的类，用来描述一个异步计算的结果，但是获取一个结果时方法较少,要么通过轮询isDone，确认完成后，调用get()获取值，要么调用get()设置一个超时时间。但是这个get()方法会阻塞住调用线程，这种阻塞的方式显然和我们的异步编程的初衷相违背。

为了解决这个问题，JDK吸收了guava的设计思想，加入了Future的诸多扩展功能形成了CompletableFuture。

<http://www.jianshu.com/p/6f3ee90ab7d3>

StampedLock 该类是一个读写锁的改进，它的思想是读写锁中读不仅不阻塞读，同时也不应该阻塞写。读不阻塞写的实现思路：在读的时候如果发生了写，则应当重读而不是在读的时候直接阻塞写！

<http://developer.51cto.com/art/201405/439149.htm>

  该类是一个读写锁的改进，它的思想是读写锁中读不仅不阻塞读，同时也不应该阻塞写。

**读不阻塞写的实现思路：**

在读的时候如果发生了写，则应当重读而不是在读的时候直接阻塞写！

        因为在读线程非常多而写线程比较少的情况下，写线程可能发生饥饿现象，也就是因为大量的读线程存在并且读线程都阻塞写线程，

因此写线程可能几乎很少被调度成功！当读执行的时候另一个线程执行了写，则读线程发现数据不一致则执行重读即可。所以读写都存在的情况下，

使用StampedLock就可以实现一种无障碍操作，即读写之间不会阻塞对方，但是写和写之间还是阻塞的！