饥饿：某一个或者多个线程因为种种原因无法获得所需要的资源，导致一直无法执行。

活锁：如果线程都秉承着“谦让”的原则，主动将资源释放给他人使用，那么就会出现资源不断在两个线程中跳动，而没有一个线程可以同时拿到所有资源而正常执行。

阿姆达尔定律：使用多核 CPU 对系统进行优化，优化的效果取决于 CPU 的数量以及系统中的串行化程序的比重。CPU 数量越多，串行化比重越低，则优化效果越好。**仅提高 CPU 数量而不降低程序的串行化比重，也无法提高系统性能**。

古斯塔夫森定律：**如果串行化比例很小，并行化比例很大，那么加速比就是处理器的个数**。只要你不断地累加处理器，就能获得更快的速度。

阿姆达尔定律强调：**当串行比例一定时，加速比是有上限的，不管你堆叠多少个 CPU 参与计算，都不能突破这个上限**。古斯塔夫森定律强调：**如果可被并行化的代码所占比重足够多，那么加速比就能随着 CPU 的数量线性增长**。

进程是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。在早期面向进程设计的计算机结构中，进程是程序的基本执行实体；在当代面向线程设计的计算机结构中，**进程是线程的容器**。程序是指令、数据及其组织形式的描述，进程是程序的实体。

线程就是轻量级进程，是程序执行的最小单位。使用多线程而不是用多进程去进行并发程序的设计，是因为**线程间的切换和调度的成本远远小于进程**。

不要用 run()来开启新线程。它只会**在当前线程中，串行执行 run()中的代码**。

Thread.stop()方法太过暴力，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.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，也就是

说程序必须捕获并处理它，当线程在 sleep()休眠时，如果被中断，这个异常就会发生。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。

线程 A 中，调用了 obj.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了 obj.notify()方法为止。这时，obj 对象就俨然成为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**。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。这个选择是不公平的，并不是先等待的线程会优先被选择，这个选择完全是随机的。

object.wait()方法并不是可以随便调用的。它**必须包含在对应的 synchronized 语句中**，无论是 wait()或者 notify()都需要首先获得目标对象的一个监视器。

T1 在正确执行 wait()方法前，首先必须获得 object 对象的监视器。而 wait()方法在执行后，会释放这个监视器。这样做的目的是使得其他等待在 object 对象上的线程不至于因为 T1 的休眠而全部无法正常工作。

线程 T2 在 notify()调用前，也必须获得 object 的监视器。所幸，此时 T1 已经释放了这个监视器。因此，T2 可以顺利获得 object 的监视器。接着，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。如果暂时无法获得，T1 还必须要等待这个监视器。当监视器顺利获得后，T1 才可以真正意义上的继续执行。

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()方法会释放目标对象的锁，而 Thread.sleep()方法不会释放任何资源**。

线程挂起（suspend）和继续执行（resume）。这两个操作是一对相反的操作，被挂起的线程，必须要等到 resume()操作后，才能继续执行。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**。此时，其他任何线程想要访问被它占用的锁时，都会被牵连，导致无法正常进行。直到对应的线程上发生了 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()操作意外地发生在 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**。并且，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。而且，**对于被挂起的线程，从它的线程状态上看，居然还是 Runnable**，这也会严重影响我们对系统当前状态的判断。

join()的本质是让调用线程 wait()在当前线程对象实例上。join()方法让调用线程在当前线程对象上等待。当线程执行完成后，被等待的线程会在退出前调用 notifyAll()通知所有的等待线程继续执行。因此，值得注意的一点是：**不要在应用程序中，在 Thread 对象实例上使用 wait()或者 notify()等方法，因为这很有可能会影响系统 API 的工作，或者被系统 API 所影响**。

yield()方法会使当前线程让出 CPU。但要注意，让出 CPU 并不表示当前线程不执行了。**当前线程在让出 CPU 后，还会进行 CPU 资源的争夺**，但是是否能够再次被分配到，就不一定了。

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。在一个 Java 应用内，只有守护线程时，Java 虚拟机就会自然退出。

volatile 并不能真正地保证线程安全。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。

重入锁使用 java.util.concurrent.locks.**ReentrantLock** 类来实现。重入锁对逻辑控制的灵活性要远远好于 synchronized。但值得注意的是，**在退出临界区时，必须记得释放锁**，否则，其他线程就没有机会再访问临界区了。

中断响应。对于 synchronized 来说，如果一个线程在等待锁，那么结果只有两种，要么它获得这把锁继续执行，要么它就保持等待。而使用重入锁，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**。

锁申请等待限时。tryLock()方法进行一次限时的等待。ReentrantLock.tryLock()方法也可以不带参数直接运行。在这种情况下，当前线程会尝试获得锁，如果锁并未被其他线程占用，则申请锁会成功，并立即返回 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。这种模式不会引起线程等待，因此也不会产生死锁。

公平锁。如果我们使用 synchronized 关键字进行锁控制，那么产生的锁就是非公平的。而重入锁允许我们对其公平性进行设置。公平锁看起来很优美，但是要实现公平锁必然要求系统维护一个**有序队列**，因此公平锁的实现成本比较高，性能相对也非常低下。

就重入锁的实现来看，它主要集中在 Java 层面。在重入锁的实现中，主要包含三个要素：
第一，是**原子状态**。原子状态使用 CAS 操作来存储当前锁的状态，判断锁是否已经被别的线程持有。
第二，是**等待队列**。所有没有请求到锁的线程，会进入等待队列进行等待。待有线程释放锁后，系统就能从等待队列中唤醒一个线程，继续工作。
第三，是**阻塞原语 park()和 unpark()**，用来挂起和恢复线程。没有得到锁的线程将会被挂起。

wait()和 notify()方法是和 synchronized 关键字合作使用的，而 Condition 是与重入锁相关联的。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。**利用 Condition 对象，我们就可以让线程在合适的时间等待，或者在某一个特定的时刻得到通知**。

和 Object.wait()和 notify()方法一样，当线程使用 Condition.await()时，要求线程持有相关的重入锁，在 Condition.await()调用后，这个线程会释放这把锁。同理，在 Condition.signal()方法调用时，也要求线程先获得相关的锁。在 signal()方法调用之后，系统会从当前 Condition 对象的等待队列中，唤醒一个线程。一旦线程被唤醒，它会重新尝试获得与之绑定的重入锁，一旦成功获取，就可以继续执行

了。因此，**在 signal()方法调用之后，一般需要释放相关的锁，谦让给被唤醒的线程**。

信号量为多线程协作提供了更为强大的控制方法。广义上说，信号量是对锁的扩展。无论是内部锁 synchronized 还是重入锁 ReentrantLock，一次都只允许一个线程访问一个资源，而**信号量（Semaphore）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，即同时能申请多少个许可。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资源。

ReadWriteLock 是 JDK5 中提供的读写分离锁。读写分离锁可以有效地帮助减少锁竞争，以提升系统性能。如果在系统中，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。

CountDownLatch 是一个非常实用的多线程控制工具类。这个工具通常用来控制线程等待，它可以让某一个线程等待直到倒计时结束，再开始执行。CountDownLatch 的构造函数接收一个整数作为参数，即当前这个计数器的计数个数。

CyclicBarrier 可以理解为循环栅栏，这个计数器可以反复使用。比 CountDownLatch 略微强大一些的是，CyclicBarrier 可以接收一个参数作为 barrierAction，这个 barrierAction 就是当计数器一次计数完成后，系统会执行的动作。

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位置让线程阻塞。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。和 Object.wait()相比，它不需要先获得某个对象的锁，也不会抛出 InterruptedException 异常。LockSupport 的静态方法 park()可以阻塞当前线程。
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，你不可能拥有超过一个许可，它永远只有一个）。这个特点使得：**即使 unpark()操作发生在 park()之前，它也可以使下一次的 park()操作立即返回**。

同时，处于 park()挂起状态的线程不会像 suspend()那样还给出一个令人费解的 Runnable 的状态。它会非常明确地给出一个 **WAITING** 状态，甚至还会**标注是 park()引起的**。

除了有定时阻塞的功能外，**LockSupport.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**。

ThreadPoolExecutor 表示一个线程池。**Executors** 类则扮演着线程池工厂的角色。

public static ExecutorService newFixedThreadPool(int nThreads)

该方法返回一个**固定线程数量**的线程池。该线程池中的线程数量始终不变。当有一个新的任务提交时，线程池中若有空闲线程，则立即执行。若没有，则新的任务会被暂存在一个**任务队列**中，待有线程空闲时，便处理在任务队列中的任务。

public static ExecutorService newSingleThreadExecutor()

该方法返回一个只有一个线程的线程池。若多于一个任务被提交到该线程池，任务会被保存在一个**任务队列**中，待线程空闲，按**先入先出**的顺序执行队列中的任务。

public static ExecutorService newCachedThreadPool()

该方法返回一个可根据实际情况调整线程数量的线程池。线程池的线程数量不确定，但若有空闲线程可以复用，则会优先使用可复用的线程。若所有线程均在工作，又有新的任务提交，则会创建新的线程处理任务。**所有线程在当前任务执行完毕后，将返回线程池进行复用**。

public static ScheduledExecutorService newSingleThreadScheduleExecutor()

该方法返回一个 ScheduledExecutorService 对象，线程池大小为 1。ScheduledExecutorService 接口在 ExecutorService 接口之上扩展了在给定时间执行某任务的功能，如在某个固定的延时之后执行，或者周期性执行某个任务。

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
该方法返回一个 ScheduleExecutorService 对象，但该线程池可以指定线程数量。

public ThreadPoolExecutor(int **corePoolSize**, int **maximumPoolSize**, long **keepAliveTime**, TimeUnit **unit**, BlockingQueue<Runnable> **workQueue**, ThreadFactory **threadFactory**, RejectedExecutionHandler **handler**)
函数的参数定义如下：
corePoolSize：指定了**线程池中的线程数量**。
maximumPoolSize：指定了**线程池中的最大线程数量**。
keepAliveTime：**当线程池线程数量超过 corePoolSize 时，多余的空闲线程的存活时间**。即，超过 corePoolSize 的空闲线程，在多长时间内，会被销毁。
unit：keepAliveTime 的单位。
workQueue：**任务队列**，被提交但尚未被执行的任务。
threadFactory：**线程工厂，用于创建线程**，一般用默认的即可。
handler：**拒绝策略**。当任务太多来不及处理，如何拒绝任务。

参数 workQueue 指被提交但未执行的任务队列，它是一个 BlockingQueue 接口的对象，仅用于存放 Runnable 对象。根据队列功能分类，在 ThreadPoolExecutor 的构造函数中可使用以下几种 BlockingQueue。
直接提交的队列：该功能由 SynchronousQueue 对象提供。SynchronousQueue 是一个特殊的 BlockingQueue。**SynchronousQueue 没有容量**，每一个插入操作都要等待一个相应的删除操作，反之，每一个删除操作都要等待对应的插入操作。如果使用 SynchronousQueue，提交的任务不会被真实地保存，而总是将新任务提交给线程执行，如果没有空闲的线程，则尝试创建新的线程，如果线程数量已经达到最大值，则执行拒绝策略。因此，**使用 SynchronousQueue 队列，通常要设置很大的 maximumPoolSize 值，否则很容易执行拒绝策略**。
有界的任务队列：有界的任务队列可以使用 ArrayBlockingQueue 实现。ArrayBlockingQueue 的构造函数必须带一个**容量参数**，表示该队列的最大容量。当使用有界的任务队列时，若有新的任务需要执行，如果线程池的实际线程数小于 corePoolSize，则会优先创建新的线程，若大于 corePoolSize，则会将新任务加入等待队列。若等待队列已满，则无法加入，则在总线程数不大于

maximumPoolSize 的前提下，创建新的线程执行任务。若大于 maximumPoolSize，则执行拒绝策略。可见，**有界队列仅当在任务队列装满时，才可能将线程数提升到 corePoolSize 以上**，换言之，除非系统非常繁忙，否则确保核心线程数维持在 corePoolSize。

无界的任务队列：无界任务队列可以通过 LinkedBlockingQueue 实现。与有界队列相比，除非系统资源耗尽，否则无界的任务队列不存在任务入队失败的情况。当有新的任务到来，系统的线程数小于 corePoolSize 时，线程池会生成新的线程执行任务，但**当系统的线程数达到 corePoolSize 后，就不会继续增加**。若后续仍有新的任务加入，而又没有空闲的线程资源，则任务直接进入队列等待。若任务创建和处理的速度差异很大，无界队列会保持快速增长，直到耗尽系统内存。

优先任务队列：优先任务队列是带有执行优先级的队列。它通过 PriorityBlockingQueue 实现，可以控制任务的执行先后顺序。它是一个特殊的**无界**队列。无论是有界队列 ArrayBlockingQueue，还是未指定大小的无界队列 LinkedBlockingQueue 都是按照先进先出算法处理任务的。而 PriorityBlockingQueue 则可以根据任务自身的优先级顺序先后执行，在确保系统性能的同时，也能有很好的质量保证（总是确保高优先级的任务先执行）。

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。(**默认策略**)

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。

DiscardOldestPolicy 策略：该策略将**丢弃最老的一个请求**，也就是即将被执行的一个任务，并尝试**再次提交当前任务**。

DiscardPolicy 策略：该策略默默地丢弃无法处理的任务，不予处理。

ThreadFactory 是一个接口，它只有一个方法，用来创建线程：

```
Thread newThread(Runnable r);
```

当线程池需要新建线程时，就会调用这个方法。

自定义线程池可以帮助我们做不少事。比如，我们可以跟踪线程池究竟在何时创建了多少线程，也可以自定义线程的名称、组以及优先级等信息，甚至可以任性地将所有的线程设置为守护线程。总之，**使用自定义线程池可以让我们更加自由地设置池子中所有线程的状态**。

ThreadPoolExecutor 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。beforeExecute()、afterExecute()会在线程池中每个线程执行的时候被调用，而 terminated()方法只会在线程池销毁前被调用。

调用线程池的 shutdown()方法可以关闭线程池，在 shutdown()方法里会调用 terminated()方法。如果当前正有线程在执行，shutdown()方法并不会立即暴力地终止所有任务，它会等待所有任务执行完成后，再关闭线程池，但它并不会等待所有线程执行完成后再返回，因此，可以简单地理解成 shutdown()只是发送了一个关闭信号而已。但在 shutdown()方法执行后，这个线程池就不能再接受其他新的任务了。

在 JDK 中，给出了一个 **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成了，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，尽可能地达到平衡。一个值得注意的地方是，**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。

你可以向 ForkJoinPool 线程池提交一个 ForkJoinTask 任务。所谓 **ForkJoinTask** 任务就是**支持 fork()分解以及 join()等待的任务**。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**。此外，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中。待将来有任务可用时，再从栈中唤醒这些线程。

ConcurrentLinkedQueue 的内部实现非常复杂，它允许在运行时链表处于多个不同的状态。以 tail 为例，我们期望 tail 总是为链表的末尾，但实际上，tail 的更新并不是及时的，而是看你会产生**拖延**现象。插入时，tail 的更新会产生滞后，并且每次更新都会跳跃两个元素。

为了将读取的性能发挥到极致，JDK 中提供了 **CopyOnWriteArrayList** 类。对它来说，读取是完全不用加锁的，并且更好的消息是：**写入也不会阻塞读取操作**。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。

BlockingQueue 之所以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（服务线程指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？

BlockingQueue 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。

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。但一个重要的区别是：对平衡树的插入和删除往往很可能导致平衡树进行一次全局的调整。而对跳表的插入和删除只需要对整个数据结构的局部进行操作即可。这样带来的好处是：**在高并发的情况下，你会需要一个全局锁来保证整个平衡树的线程安全。而对于跳表，你只需要部分锁即可**。

不变式：并发对象需要一直保持的特性，不变式是并发对象的各个方法之间必须遵守的“契约”，每个方法在调用前和调用后都必须保持不变式。采用不变式，就可以**隔离地分析每个方法**，而不用考虑它们之间所有可能的交互。

ConcurrentLinkedQueue 的三个不变式：

基本不变式：在执行方法之前和之后，队列必须要保持的不变式：

（1）**当入队插入新节点之后，队列中有一个 next 域为 null 的（最后）节点**。

（2）**从 head 开始遍历队列，可以访问所有 item 域不为 null 的节点**。

head 的不变式和可变式：

在执行方法之前和之后，head 必须保持的不变式：

（1）**所有“活着”的节点（指未删除节点），都能从 head 通过调用 succ() 方法遍历可达**。

（2）**head 不能为 null**。

（3）**head 节点的 next 域不能引用到自身**。

在执行方法之前和之后，head 的可变式：

（1）**head 节点的 item 域可能为 null，也可能不为 null**。

（2）**允许 tail 滞后（lag behind）于 head，也就是说：从 head 开始遍历队列，不一定能到达 tail**。

tail 的不变式和可变式：

在执行方法之前和之后，tail 必须保持的不变式：

（1）**通过 tail 调用 succ()方法，最后节点总是可达的**。

（2）**tail 不能为 null**。

在执行方法之前和之后，tail 的可变式：

（1）**tail 节点的 item 域可能为 null，也可能不为 null**。

（2）**允许 tail 滞后于 head，也就是说：从 head 开始遍历队列，不一定能到达 tail**。

（3）**tail 节点的 next 域可以引用到自身**。

为了尽量减少执行 CAS 原子指令的次数，执行入队/出队操作时，ConcurrentLinkedQueue 并不总是更新 head/tail。只有从 head/tail 到头/尾节点之间的“距离”达到变量 HOPS 指定的阈值，入队/出队操作才会更新它们。

以批处理方式更新减少了更新 head/tail 的次数（减少了执行 CAS 原子指令的次数），但额外地增加了遍历队列，寻找头/尾节点的开销（增加了读 volatile 变量的开销）。在当前大多数的处理器系统中，volatile 读操作的开销非常低，几乎和非 volatile 变量的读操作一样。而执行一条 CAS 原子指令要消耗比普通加载或存储指令多得多的时钟周期。因为 CAS 原子指令的执行包含了内存屏障，防止乱序执行以及对各种编译器优化的限制。以批处理方式更新 head/tail，从整体上减少了入队/出队操作的开销。

减少锁的持有时间。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。所谓减小锁粒度，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。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。通过 takeLock 和 putLock 两把锁，LinkedBlockingQueue 实现了取数据和写数据的分离，使两者在真正意义上成为可并发的操作。

锁粗化。虚拟机在遇到一连串连续地对同一锁不断进行请求和释放的操作时，便会把所有的锁操作整合成对锁的一次请求，从而减少对锁的请求同步次数，这个操作叫做锁的粗化。**在循环体内请求锁，应该改为在外层只请求一次锁**。

JAVA 虚拟机对锁优化所做的努力：

锁偏向：**如果一个线程获得了锁，那么锁就进入偏向模式。当这个线程再次请求锁时，无须再做任何同步操作**。

轻量级锁：如果偏向锁失败，虚拟机并不会挂起线程。它还会使用一种称为轻量级锁的优化手段。轻量级锁的操作也很轻便，它只是简单地**将对象头部作为指针，指向持有锁的线程堆栈的内部，来判断一个线程是否持有对象锁**。如果线程获得轻量级锁成功，则可以顺利进入临界区。如果轻量级锁加锁失败，则表示其他线程抢先争夺到了锁，那么当前线程的锁请求就会膨胀为重量级锁。

自旋锁：锁膨胀后，虚拟机为了避免线程真实地在操作系统层面挂起，还会再做最后的努力——自旋锁。由于当前线程暂时无法获得锁，但是什么时候可以获得锁是一个未知数。因此，系统会进行一次赌注：它会假设不久的将来，线程可以得到这把锁。因此，虚拟机会让当前线程做几个**空循环**，在经过若干次循环后，如果还不能获得锁，才会真实地将线程在操作系统层面挂起。

锁消除。Java 虚拟机在 **JIT 编译**时，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁。锁消除涉及的一项关键技术为**逃逸分析**。所谓逃逸分析就是**观察某一个变量是否会逃出某一个作用域**。

为每一个线程分配不同的对象，需要在应用层面保证。ThreadLocal 只是起到了简单的容器作用。在 Thread 里有一个 ThreadLocalMap 对象，ThreadLocal

中的变量就设置在 Thread 里的 ThreadLocalMap 里，其中键为 ThreadLocal 对象，而值为我们需要的值（ThreadLocal 中 set 方法设置的值）。由于 ThreadLocal 涉及到的变量是维护在 Thread 类内部的，这也意味着只要线程不退出，对象的引用将一直存在。因此，如果我们使用线程池，那就意味着当前线程未必会退出（比如固定大小的线程池，线程总是存在）。如果这样，将一些太大的对象设置到 ThreadLocal 中（它实际保存在线程持有的 threadLocals Map 内），可能会使系统出现**内存泄漏**的可能（你设置了对象到 ThreadLocal，但是不清理它，在你使用几次后，这个对象也不再有用了，但是它却无法被回收）。此时，如果你希望及时回收对象，最好使用 **ThreadLocal.remove()**方法将这个变量移除。如果对于 ThreadLocal 的变量，我们也手动将其设置为 null，臂如 tl = null。那么，这个 ThreadLocal 对应的所有线程的局部变量都有可能被回收。

AtomicInteger 与 Integer 不同，它是可变的，并且是线程安全的，对其进行修改等任何操作，都是用 CAS 指令执行的。和 AtomicInteger 类似的类还有 **AtomicLong** 用来代表 long 型，**AtomicBoolean** 表示 boolean 型，**AtomicReference** 表示对象引用。

Unsafe 封装了一些类似指针的操作。compareAndSwapInt()方法是一个 native 方法，它的几个参数含义如下：

public final native Boolean compareAndSwapInt(Object o, long offset, int expected, int x);

第一个参数 o 为给定的对象，**offset 为对象内的偏移量（其实就是一个字段到对象头部的偏移量，通过这个偏移量可以快速定位字段）**，expected 表示期望值，x 表示要设置的值。如果指定的字段的值等于 expected，那么就会把它设置为 x。

AtomicReference 对应普通对象的引用，它可以保证你**在修改对象引用时的线程安全性**。但是 AtomicReference 无法解决 ABA 问题。

AtomicReference 无法解决 ABA 问题的根本因为是对象在修改过程中，丢失了状态信息。对象值本身与状态被画上了等号。因此，我们只要能够记录对象在修改过程中的状态值，就可以很好地解决对象被反复修改导致线程无法正确判断对象状态的问题。**AtomicStampedReference** 正是这么做的。它内部不仅维护了对象值，还维护了一个**时间戳**。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**

设置对象时，对象值以及时间戳都必须满足期望值，写入才会成功。

除了提供基本数据类型外，JDK 还为我们准备了数组等复合结构。当前可用的原子数组有：**AtomicIntegerArray**、**AtomicLongArray** 和 **AtomicReferenceArray**。

AtomicIntegerFieldUpdater、**AtomicLongFieldUpdater** 和 **AtomicReferenceFieldUpdater** 分别可以对 int、long 和普通对象进行 CAS 修改。

虽然 AtomicIntegerFieldUpdater 很好用，但有几个注意事项：

（1）**Updater 只能修改它可见范围内的变量**。因为 Updaer 使用反射得到这个变量。如果变量不可见，就会出错。比如 private 是不可行的。

（2）为了确保变量被正确地读取，它**必须是 volatile 类型**的。

（3）**由于 CAS 操作会通过对象实例中的偏移量直接进行赋值，因此，它不支持 static 字段**。

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）。

Transferer.transfer()函数的实现是 SynchronousQueue 的核心，它大体上分为三个步骤：

（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**。

从整个数据投递的过程中可以看到，在 SynchronousQueue 中，参与工作的所有线程不仅是竞争资源的关系。更重要的是，它们彼此之间还会**互相帮助**。在一个线程内部，可能会帮助其他线程完成它们的工作。这种模式可以更大程度上

减少饥饿的可能，提高系统整体的并行度。

在 Java 语言中，**不变模式**的实现很简单。为确保对象被创建后，不发生任何改变，并保证不变模式正常工作，只需要注意以下 4 点：

（1）**去除 setter 方法以及所有修改自身属性的方法**。

（2）**将所有属性设置为私有，并用 final 标记，确保其不可修改**。

（3）**确保没有子类可以重载修改它的行为**，final 修饰类。

（4）**有一个可以创建完整对象的构造函数**。

生产者-消费者模式是一个经典的多线程设计模式，它为多线程间的协作提供了良好的解决方案。在生产者-消费者模式中，通常有两类线程，即若干个生产者线程和若干个消费者线程。**生产者线程**负责**提交用户请求**，**消费者线程**则负责**具体处理生产者提交的任务**。生产者和消费者之间则通过**共享内存缓冲区**进行通信。

生产者和消费者模式中的内存缓冲区的主要功能是**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间的性能差**。

Disruptor 维护一个环形队列。为了能够快速从一个序列（sequence）对应到数组的实际位置（每次有元素入队，序列就加 1），Disruptor 要求我们必须将该环形队列的大小设置为 2 的整数次方。这样通过 **sequence & (queueSize – 1)**就能立即定位到实际的元素位置 index，这个要比取余操作快得多。

如果 queueSize 是 2 的整数次幂，则这个数字的二进制表示必然是 10、100、1000、10000 等形式。因此，queueSize – 1 的二进制则是一个全 1 的数字。因此它可以**将 sequence 限定在 queueSize – 1 范围内，并且不会有任何一位是浪费的**。

生产者写入数据时，使用 CAS 操作，消费者读取数据时，为了防止多个消费者处理同一个数据，也使用 CAS 操作进行数据保护。

为了提高 CPU 的速度，CPU 有一个高速缓存 Cache。在高速缓存中，读写数据的最小单位为缓存行（Cache Line），它是从主存（memory）复制到缓存（Cache）的最小单位。

如果两个变量存放在一个缓存行中时，在多线程访问中，可能会相互影响彼此的性能。假设 X 和 Y 在同一个缓存行。运行在 CPU1 上的线程更新了 X，那么 CPU2 上的缓存行就会失效，同一行的 Y 即使没有修改也会变成无效，导致 Cache

无法命中。接着，如果在 CPU2 上的线程更新了 Y。则导致 CPU1 上的缓存行又失效（此时，同一行的 X 又变得无法访问）。如果 CPU 经常不能命中缓存，那么系统的吞吐量就会急剧下降。

Future 模式是多线程开发中非常常见的一种设计模式，它的核心思想是异步调用。当我们需要调用一个函数方法时，如果这个函数执行很慢，那么我们就要进行等待。但有时候，我们可能并不急着要结果。因此，我们可以让被调者立即返回，让它在后台慢慢处理这个请求。对于调用者来说，则可以先处理一些其他任务，在真正需要数据的场合再去尝试获得需要的数据。

对于 Future 模式来说，虽然它无法立即给出你需要的数据。但是，它会返回给你一个契约，将来，你可以凭借着这个契约去重新获取你需要的信息。

流水线思想。

服务器**为每一个客户端连接开启一个线程**，这种模式倾向于让 CPU 进行 **IO 等待**。使用 Java 的 NIO 就可以将上面的网络 IO 等待时间从业务处理线程中抽取出来。

Channel（通道）是 NIO 中的一个关键组件。Channel 有点类似于流，一个 Channel 可以和文件或者网络 Socket 对应。如果 Channel 对应着一个 Socket，那么往这个 Channel 中写数据，就等同于向 Socket 中写入数据。

和 Channel 一起使用的另外一个重要组件是 **Buffer**，可以理解为一个内存区域或者 byte 数组。**数据需要包装成 Buffer 的形式才能和 Channel 交互**。

另外一个与 Channel 密切相关的是 **Selector**（选择器）。在 Channel 的众多实现中，有一个 SelectableChannel 实现，表示可被选择的通道。任何一个 SelectableChannel 都可以将自己注册到一个 Selector 中。这样，这个 Channel 就能被 Selector 管理。而一个 Selector 可以管理多个 SelectableChannel。当 SelectableChannel 的数据准备好时，Selector 就会接到通知，得到那些已经准备好的数据。而 SocketChannel 就是 SelectableChannel 的一种。

一个 Selector 可以由一个线程进行管理，而一个 SocketChannel 则可以表示一个客户端连接，因此这就构成由一个或者极少数线程，来处理大量客户端连接的结构。**当与客户端连接的数据没有准备好时，Selector 会处于等待状态**（不过幸好，用于管理 Selector 的线程数是极少量的），**而一旦有任何一个 SocketChannel 准备好了数据，Selector 就能立即得到通知，获取数据进行处理**。

AIO 是异步 IO 的缩写，即 Asynchronized。虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 **NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO 操作本身还是同步的**。

但**对于 AIO 来说，则更加进了一步，它不是在 IO 准备好时再通知线程，而是在 IO 操作已经完成后，再给线程发出通知**。因此，AIO 是完全不会阻塞的。此时，我们的业务逻辑将变成一个**回调函数**，等待 IO 操作完成后，由系统自动触发。

显式函数指**函数与外界交换数据的唯一渠道就是参数和返回值，显式函数不会去读取或者修改函数的外部状态**。与之相对的是**隐式函数**，隐式函数除了参数和返回值外，还**会读取外部信息，或者可能修改外部信息**。

函数式接口，就是只定义了单一抽象方法的接口。**函数式接口只能有一个抽象方法，而不是只能有一个方法**。首先，在 Java8 中，接口运行存在实例方法，其次任何被 java.lang.Object 实现的方法，都不能视为抽象方法。

在 Java8 中，使用 default 关键字，可以在接口内定义**实例**方法，这个方法并非抽象方法，而是拥有特定逻辑的具体**实例**方法。（不能是静态方法）

StampedLock 内部实现时，使用类似于 CAS 操作的死循环反复尝试的策略。在它挂起线程时，使用的是 Unsafe.park()函数，而 **park()函数在遇到线程中断时，会直接返回（注意，不同于 Thread.sleep()，它不会抛出异常）**。而在 StampedLock 的死循环逻辑中，没有处理有关中断的逻辑。因此，这就会导致**阻塞在 park()上的线程被中断后，会再次进入循环。而当退出条件得不到满足时，就会发生疯狂占用 CPU 的情况**。

StampedLock 的内部实现是基于 **CLH 锁**的。CLH 锁是一种**自旋锁**，它保证**没饥饿**发生，并且可以保证 **FIFO** 的服务顺序。

CLH 锁的基本思想如下：锁维护一个等待线程队列，所有申请锁，但是没有成功的线程都记录在这个队列中。每一个节点（一个节点代表一个线程），保存一个标记位（locked），用于判断当前线程是否已经释放锁。

当一个线程试图获得锁时，取得当前等待队列的尾部节点作为其前序节点，并使用类似如下代码判断前序节点是否已经成功释放锁：

```
while (pred.locked) {}
```

只要前序节点（pred）没有释放锁，则表示当前线程还不能继续执行，因此会自旋等待。反之，如果前序线程已经释放锁，则当前线程可以继续执行。释放锁时，也遵循这个逻辑，线程会将自身节点的 locked 位置标记为 false，那么后续等待的线程就能继续执行了。