[【死磕Java并发】—–深入分析synchronized 的实现原理](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483816&idx=1&sn=16c0f7f48bddb2d7ca6546d1cf0f20b2&chksm=ec505879db27d16f3586d05bd703afbbea9d8003ff03d1a395103358a53756769dc1f1bf0c4d&scene=21" \l "wechat_redirect" \t "_blank)

synchronized 可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性。深入分析 synchronized 的内在实现机制，锁优化、锁升级过程。

[【死磕Java并发】—–深入分析volatile的实现原理](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483668&idx=1&sn=fa89a1c92dd551aa0c14ed56ae015879&chksm=ec5058c5db27d1d3d9b0b4031b5dd32828d91631770583cb9b547b988bf7a5bdeb388a16b2a3&scene=21#wechat_redirect)

volatile 可以保证线程可见性且提供了一定的有序性，但是无法保证原子性。在 JVM 底层 volatile 是采用“内存屏障”来实现的。这篇博文将带你分析 volatile 的本质

[【死磕Java并发】—–Java内存模型之happens-before](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483678&idx=1&sn=ee05887d93ffa4d2ef38d95265b1769a&chksm=ec5058cfdb27d1d9cef96f3c728de98bc80d053e7230a8bb5277b5272e40595e1cac61474d03&scene=21#wechat_redirect)

happens-before 原则是判断数据是否存在竞争、线程是否安全的主要依据，保证了多线程环境下的可见性。

定义如下：

1. 如果一个操作happens-before另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前。
2. 两个操作之间存在happens-before关系，并不意味着一定要按照happens-before原则制定的顺序来执行。如果重排序之后的执行结果与按照happens-before关系来执行的结果一致，那么这种重排序并不非法。

[【死磕Java并发】—–Java内存模型之重排序](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483675&idx=1&sn=59bbd1bfba287cdce591362ccde66e33&chksm=ec5058cadb27d1dca8234ee80ecec0f7a485ba506ae3978596941bc954d8c4760140ff7647bb&scene=21#wechat_redirect)

在执行程序时，为了提供性能，处理器和编译器常常会对指令进行重排序，但是不能随意重排序，不是你想怎么排序就怎么排序，它需要满足以下两个条件：

* 在单线程环境下不能改变程序运行的结果；
* 存在数据依赖关系的不允许重排序

as-if-serial 语义保证在单线程环境下重排序后的执行结果不会改变。

[【死磕Java并发】—–Java内存模型之分析volatile](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483681&idx=1&sn=0e20f76f674c0840dc4828111fabf018&chksm=ec5058f0db27d1e6d2a10a328f0137e23e278b448121adcf106597078e2db364e9a8518b91ad&scene=21#wechat_redirect)

volatile的内存语义是：

* 当写一个 volatile 变量时，JMM 会把该线程对应的本地内存中的共享变量值立即刷新到主内存中。
* 当读一个 volatile 变量时，JMM 会把该线程对应的本地内存设置为无效，直接从主内存中读取共享变量

总是说 volatile 保证可见性，happens-before 是 JMM 实现可见性的基础理论，两者会碰撞怎样的火花？这篇博文给你答案。

[【死磕Java并发】—–Java内存模型之从JMM角度分析DCL](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483686&idx=1&sn=b698404703bafd6e0e4f0269e4ed5721&chksm=ec5058f7db27d1e12b99a9c8930ff4cea7253e172a19a40ebf5f580302692c7c30fcbc7c4e39&scene=21#wechat_redirect)

DCL，即Double Check Lock，双重检查锁定。是实现单例模式比较好的方式，这篇博客告诉你 DCL 中为何要加 volatile 这个关键字。

[【死磕Java并发】—–J.U.C之AQS：AQS简介](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483693&idx=1&sn=acf00a582ee54bd5f0d5f43005896385&chksm=ec5058fcdb27d1ea1b487e1f6f6f652168f3c955841cf05060833cff61c48a74549f1fa3544f&scene=21#wechat_redirect)

AQS，AbstractQueuedSynchronizer，即队列同步器。它是构建锁或者其他同步组件的基础框架（如ReentrantLock、ReentrantReadWriteLock、Semaphore等），为 JUC 并发包中的核心基础组件。

[【死磕Java并发】—–J.U.C之AQS：CLH同步队列](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483696&idx=1&sn=329316c01b00359e5c5ac3d8364f8ca4&chksm=ec5058e1db27d1f720d93951b4d14c5383add557b19b92f6cc5b11925fe9d14a4fa87e9ed787&scene=21#wechat_redirect)

前线程已经等待状态等信息构造成一个节点（Node）并将其加入到CLH同步队列，同时会阻塞当前线程，当同步状态释放时，会把首节点唤醒（公平锁），使其再次尝试获取同步状态。

[【死磕Java并发】—–J.U.C之AQS：同步状态的获取与释放](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483699&idx=1&sn=9bc31b9c74244d4387216a14045bcfab&chksm=ec5058e2db27d1f4d1a040d8be4f4f6cad1776a44cd6e718f366b13fcf2dfbf60879bfc5912c&scene=21#wechat_redirect)

AQS的设计模式采用的模板方法模式，子类通过继承的方式，实现它的抽象方法来管理同步状态，对于子类而言它并没有太多的活要做，AQS提供了大量的模板方法来实现同步，主要是分为三类：独占式获取和释放同步状态、共享式获取和释放同步状态、查询同步队列中的等待线程情况。

[【死磕Java并发】—–J.U.C之AQS：阻塞和唤醒线程](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483702&idx=1&sn=f17d18d4cb0205ab437628eadb2a6101&chksm=ec5058e7db27d1f1d9c0aa83a42c04c7054f2cdfd3e98132af48430ea7d8b96d454992283f5d&scene=21#wechat_redirect)

当需要阻塞或者唤醒一个线程的时候，AQS 都是使用 LockSupport 这个工具类来完成。

LockSupport是用来创建锁和其他同步类的基本线程阻塞原语。

[【死磕Java并发】—–J.U.C之重入锁：ReentrantLock](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483708&idx=1&sn=99bbbce46c5885b1875428a63c4d3731&chksm=ec5058eddb27d1fbef6b8b4a8a3f59c0a8fbe108b2ce3858baf2843824e1c7ec359343c5493d&scene=21#wechat_redirect)

一个可重入的互斥锁定 Lock，它具有与使用 synchronized 方法和语句所访问的隐式监视器锁定相同的一些基本行为和语义，但功能更强大。ReentrantLock 将由最近成功获得锁定，并且还没有释放该锁定的线程所拥有。当锁定没有被另一个线程所拥有时，调用 lock 的线程将成功获取该锁定并返回。如果当前线程已经拥有该锁定，此方法将立即返回。可以使用 isHeldByCurrentThread() 和 getHoldCount() 方法来检查此情况是否发生。

这篇博客带你理解 重入锁：ReentrantLock 内在本质。

[【死磕Java并发】—–J.U.C之读写锁：ReentrantReadWriteLock](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483734&idx=1&sn=b83faa922da639703e5cf956198ef850&chksm=ec505887db27d191851eef277a9378b51981df48f94327157068c40e78d862d283f0463e6552&scene=21#wechat_redirect)

读写锁维护着一对锁，一个读锁和一个写锁。通过分离读锁和写锁，使得并发性比一般的排他锁有了较大的提升：在同一时间可以允许多个读线程同时访问，但是在写线程访问时，所有读线程和写线程都会被阻塞。

读写锁的主要特性：

* 公平性：支持公平性和非公平性。
* 重入性：支持重入。读写锁最多支持65535个递归写入锁和65535个递归读取锁。
* 锁降级：遵循获取写锁、获取读锁在释放写锁的次序，写锁能够降级成为读锁

[【死磕Java并发】—–J.U.C之Condition](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483756&idx=1&sn=eae414fab1a6ba941708392ad4cf675b&chksm=ec5058bddb27d1ab12be95b903baa9bead40ce6bb46673910ea17214c320ec5ad564609cd587&scene=21#wechat_redirect)

在没有Lock之前，我们使用synchronized来控制同步，配合Object的wait()、notify()系列方法可以实现等待/通知模式。在Java SE5后，Java提供了Lock接口，相对于Synchronized而言，Lock提供了条件Condition，对线程的等待、唤醒操作更加详细和灵活

[【死磕Java并发】—-深入分析CAS](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483759&idx=1&sn=867d9ef27022b12367db2a60dbe9bce1&chksm=ec5058bedb27d1a86d02b485c6347058a248c5f8f40cbe1ff7880388acc47afc29f550ce4f31&scene=21#wechat_redirect)

CAS，Compare And Swap，即比较并交换。Doug lea大神在同步组件中大量使用 CAS 技术鬼斧神工地实现了Java 多线程的并发操作。整个 AQS 同步组件、Atomic 原子类操作等等都是以 CAS 实现的。可以说CAS是整个JUC的基石。

[【死磕Java并发】—–J.U.C之并发工具类：CyclicBarrier](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483762&idx=1&sn=4fc0f9ff9b1b2ef7760ba3cf753fcc78&chksm=ec5058a3db27d1b5c14a81b2c9d67087095b5ed648bd92d0323fb412e76c53f003f7871d5176&scene=21#wechat_redirect)

CyclicBarrier，一个同步辅助类。它允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。因为该 barrier 在释放等待线程后可以重用，所以称它为循环 的 barrier。

[【死磕Java并发】—–J.U.C之并发工具类：CountDownLatch](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483764&idx=1&sn=26148101b339206b8d12c8ba8ac05449&chksm=ec5058a5db27d1b381eb3cf1ddd1d0fc8e990ed69a36d0edcbef6cfa4a39cc4502e7eec3babc&scene=21#wechat_redirect)

CountDownLatch 所描述的是”在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待“。

用给定的计数 初始化 CountDownLatch。由于调用了 countDown() 方法，所以在当前计数到达零之前，await 方法会一直受阻塞。之后，会释放所有等待的线程，await 的所有后续调用都将立即返回。

[【死磕Java并发】—–J.U.C之并发工具类：Semaphore](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247484014&idx=1&sn=ebe2210c8a8c5055ad00b8eab7259c08&chksm=ec505bbfdb27d2a9f4def331ff95433033794b7bb855cfec529ac5eb3d4bbb9ec54e0fe1c310&scene=21#wechat_redirect)

Semaphore，信号量，是一个控制访问多个共享资源的计数器。从概念上讲，信号量维护了一个许可集。如有必要，在许可可用前会阻塞每一个 acquire()，然后再获取该许可。每个 release() 添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，Semaphore 只对可用许可的号码进行计数，并采取相应的行动。

[【死磕Java并发】—–J.U.C之并发工具类：Exchanger](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483767&idx=1&sn=07f8fe9143ee5fbba95d53e7240567d3&chksm=ec5058a6db27d1b06a9c2bcc8cee5be0746786e869437436562c21ac116f5ad0eef6e3beda70&scene=21#wechat_redirect)

可以在对中对元素进行配对和交换的线程的同步点。每个线程将条目上的某个方法呈现给 exchange 方法，与伙伴线程进行匹配，并且在返回时接收其伙伴的对象。Exchanger 可能被视为 SynchronousQueue 的双向形式。Exchanger 可能在应用程序（比如遗传算法和管道设计）中很有用。

[【死磕Java并发】—–J.U.C之Java并发容器：ConcurrentHashMap](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483820&idx=1&sn=556326307ce4d7b35891b20429ddeb88&chksm=ec50587ddb27d16bb49e9d55731673bfcfc9682efad4d0f6424701600ac46012ae08c667abfa&scene=21#wechat_redirect)

ConcurrentHashMap 作为 Concurrent 一族，其有着高效地并发操作。在1.8 版本以前，ConcurrentHashMap 采用分段锁的概念，使锁更加细化，但是 1.8 已经改变了这种思路，而是利用 CAS + Synchronized 来保证并发更新的安全，当然底层采用数组+链表+红黑树的存储结构。这篇博客带你彻底理解 ConcurrentHashMap。

[【死磕Java并发】—–J.U.C之ConcurrentHashMap红黑树转换分析](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483771&idx=1&sn=c91d38a343f111f7ecc953976cf63b65&chksm=ec5058aadb27d1bc14c0bd32dcc743520c71f076f6070c6abfe8ce3aae24e4a15a9d69a556f1&scene=21#wechat_redirect)

在 1.8 ConcurrentHashMap 的put操作中，如果发现链表结构中的元素超过了TREEIFY\_THRESHOLD（默认为8），则会把链表转换为红黑树，已便于提高查询效率。那么具体的转换过程是怎么样的？这篇博客给你答案。

[【死磕Java并发】—–J.U.C之Java并发容器：ConcurrentLinkedQueue](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483774&idx=1&sn=3b785aae9008613ff14204de4471de4c&chksm=ec5058afdb27d1b9e79320780eb492581db989de2f799f000a339e3ae5898a96d2193ff1e740&scene=21#wechat_redirect)

ConcurrentLinkedQueue是一个基于链接节点的无边界的线程安全队列，它采用FIFO原则对元素进行排序。采用“wait-free”算法（即CAS算法）来实现的。

CoucurrentLinkedQueue规定了如下几个不变性：

1. 在入队的最后一个元素的next为null
2. 队列中所有未删除的节点的item都不能为null且都能从head节点遍历到
3. 对于要删除的节点，不是直接将其设置为null，而是先将其item域设置为null（迭代器会跳过item为null的节点）
4. 允许head和tail更新滞后。这是什么意思呢？意思就说是head、tail不总是指向第一个元素和最后一个元素（后面阐述）。

[【死磕Java并发】—–J.U.C之Java并发容器：ConcurrentSkipListMap](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483776&idx=1&sn=9a61aa037a7e0519b0998fc4d12fb4a8&chksm=ec505851db27d14795ecf8503eba96166f21b61a82b5109013c3e1ec32e8a58ce9215499ee24&scene=21#wechat_redirect)

我们在Java世界里看到了两种实现key-value的数据结构：Hash、TreeMap，这两种数据结构各自都有着优缺点。

* Hash表：插入、查找最快，为O(1)；如使用链表实现则可实现无锁；数据有序化需要显式的排序操作。
* 红黑树：插入、查找为O(logn)，但常数项较小；无锁实现的复杂性很高，一般需要加锁；数据天然有序。

这里介绍第三种实现 key-value 的数据结构：SkipList。SkipList 有着不低于红黑树的效率，但是其原理和实现的复杂度要比红黑树简单多了。

ConcurrentSkipListMap 其内部采用 SkipLis 数据结构实现。

[【死磕Java并发】—–J.U.C之阻塞队列：ArrayBlockingQueue](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483779&idx=1&sn=da4fcc41599819a29e8ef41a4e1d8918&chksm=ec505852db27d1446e667a1820c63c3639490f8ef59fee06e992878af19d52f301c64f002c74&scene=21#wechat_redirect)

ArrayBlockingQueue，一个由数组实现的有界阻塞队列。该队列采用FIFO的原则对元素进行排序添加的。

ArrayBlockingQueue 为有界且固定，其大小在构造时由构造函数来决定，确认之后就不能再改变了。ArrayBlockingQueue 支持对等待的生产者线程和使用者线程进行排序的可选公平策略，但是在默认情况下不保证线程公平的访问，在构造时可以选择公平策略（fair = true）。公平性通常会降低吞吐量，但是减少了可变性和避免了“不平衡性”。

[【死磕Java并发】—–J.U.C之阻塞队列：PriorityBlockingQueue](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483782&idx=1&sn=5dc55736a787349075c20bf330dcb4a5&chksm=ec505857db27d14161e10e2e6169f673af15fd2a1d7472f2d0771986637cd3fcf6f2debffabe&scene=21#wechat_redirect)

PriorityBlockingQueue是一个支持优先级的无界阻塞队列。默认情况下元素采用自然顺序升序排序，当然我们也可以通过构造函数来指定Comparator来对元素进行排序。需要注意的是PriorityBlockingQueue不能保证同优先级元素的顺序。

[【死磕Java并发】—–J.U.C之阻塞队列：DelayQueue](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483794&idx=1&sn=1b3c0483ac5028cc1f2af5b779352c3c&chksm=ec505843db27d15528e74b8692497ea9892f61c8538866b9995f5b7409e2ee3c44de075afb0a&scene=21#wechat_redirect)

DelayQueue是一个支持延时获取元素的无界阻塞队列。里面的元素全部都是“可延期”的元素，列头的元素是最先“到期”的元素，如果队列里面没有元素到期，是不能从列头获取元素的，哪怕有元素也不行。也就是说只有在延迟期到时才能够从队列中取元素。

DelayQueue主要用于两个方面：

* 缓存：清掉缓存中超时的缓存数据
* 任务超时处理

[【死磕Java并发】—–J.U.C之阻塞队列：SynchronousQueue](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483812&idx=1&sn=453bad2d2c1300272e24c787e322cbe3&chksm=ec505875db27d163c6784d878dac9cd88656f04dfd880e396b13f06eb13619c8773c02fcdb82&scene=21#wechat_redirect)

SynchronousQueue与其他BlockingQueue有着不同特性：

1. SynchronousQueue没有容量。与其他BlockingQueue不同，SynchronousQueue是一个不存储元素的BlockingQueue。每一个put操作必须要等待一个take操作，否则不能继续添加元素，反之亦然。
2. 因为没有容量，所以对应 peek, contains, clear, isEmpty ... 等方法其实是无效的。例如clear是不执行任何操作的，contains始终返回false,peek始终返回null。
3. SynchronousQueue分为公平和非公平，默认情况下采用非公平性访问策略，当然也可以通过构造函数来设置为公平性访问策略（为true即可）。
4. 若使用 TransferQueue, 则队列中永远会存在一个 dummy node（这点后面详细阐述）。

SynchronousQueue非常适合做交换工作，生产者的线程和消费者的线程同步以传递某些信息、事件或者任务。

[【死磕Java并发】—–J.U.C之阻塞队列：LinkedTransferQueue](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483782&idx=1&sn=5dc55736a787349075c20bf330dcb4a5&chksm=ec505857db27d14161e10e2e6169f673af15fd2a1d7472f2d0771986637cd3fcf6f2debffabe&scene=21#wechat_redirect)

LinkedTransferQueue 是基于链表的 FIFO 无界阻塞队列，它出现在 JDK7 中。Doug Lea 大神说 LinkedTransferQueue 是一个聪明的队列。它是 ConcurrentLinkedQueue、SynchronousQueue (公平模式下)、无界的LinkedBlockingQueues 等的超集。

[【死磕Java并发】—–J.U.C之阻塞队列：LinkedBlockingDeque](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483786&idx=1&sn=b29eed9809502dd39f5bd76417705c35&chksm=ec50585bdb27d14ddcad1cf55e0a5b7968bda3fb126eac94a4c9adc7bcc440e291429ad6b1f3&scene=21#wechat_redirect)

LinkedBlockingDeque 是一个由链表组成的双向阻塞队列，双向队列就意味着可以从对头、对尾两端插入和移除元素，同样意味着 LinkedBlockingDeque 支持 FIFO、FILO 两种操作方式。

LinkedBlockingDeque 是可选容量的，在初始化时可以设置容量防止其过度膨胀，如果不设置，默认容量大小为 Integer.MAX\_VALUE。

[【死磕Java并发】—–深入分析ThreadLocal](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483840&idx=1&sn=fcd29305ad5afda707e86aed3c742118&chksm=ec505811db27d1072866d241c43428901eb7204655f0f48b2878cf126446a1829fe0c0638cee&scene=21#wechat_redirect)

ThreadLocal 提供了线程局部 (thread-local) 变量。这些变量不同于它们的普通对应物，因为访问某个变量（通过其get 或 set 方法）的每个线程都有自己的局部变量，它独立于变量的初始化副本。ThreadLocal实例通常是类中的 private static 字段，它们希望将状态与某一个线程（例如，用户 ID 或事务 ID）相关联。

所以ThreadLocal与线程同步机制不同，线程同步机制是多个线程共享同一个变量，而ThreadLocal是为每一个线程创建一个单独的变量副本，故而每个线程都可以独立地改变自己所拥有的变量副本，而不会影响其他线程所对应的副本。可以说ThreadLocal为多线程环境下变量问题提供了另外一种解决思路。

[【死磕Java并发】—–J.U.C之线程池：ThreadPoolExecutor](http://mp.weixin.qq.com/s?__biz=MzI5NTYwNDQxNA==&mid=2247483862&idx=1&sn=c0e874004e6f79eaf1a760331c4b3ed7&chksm=ec505807db27d111b4b9e8070896829589d3ff179907f7a88928608c070f74990509cbba79b2&scene=21#wechat_redirect)

鼎鼎大名的线程池。不需要多说!!!!!

这篇博客深入分析 Java 中线程池的实现。

【死磕Java并发】—–J.U.C之线程池：ScheduledThreadPoolExecutor

ScheduledThreadPoolExecutor 是实现线程的周期、延迟调度的。

ScheduledThreadPoolExecutor，继承 ThreadPoolExecutor 且实现了 ScheduledExecutorService 接口，它就相当于提供了“延迟”和“周期执行”功能的 ThreadPoolExecutor。在JDK API中是这样定义它的：ThreadPoolExecutor，它可另行安排在给定的延迟后运行命令，或者定期执行命令。需要多个辅助线程时，或者要求 ThreadPoolExecutor 具有额外的灵活性或功能时，此类要优于 Timer。 一旦启用已延迟的任务就执行它，但是有关何时启用，启用后何时执行则没有任何实时保证。按照提交的先进先出 (FIFO) 顺序来启用那些被安排在同一执行时间的任务。