多线程

Table of Contents

[多线程 1](#_Toc16971438)

[什么是线程？ 2](#_Toc16971439)

[线程和进程有什么区别？ 2](#_Toc16971440)

[如何在Java中创建线程？ 2](#_Toc16971441)

[什么时候使用Runnable还是Thread?你自己一般通过什么方式创建线程？ 2](#_Toc16971442)

[Thread 类中的start() 和 run() 方法有什么区别？ 2](#_Toc16971443)

[java中Runnable和Callable有什么不同？ 2](#_Toc16971444)

[Deamon线程 3](#_Toc16971445)

[Java中CyclicBarrier 和 CountDownLatch有什么不同？ 3](#_Toc16971446)

[物理计算机的并发问题 3](#_Toc16971447)

[Java内存模型是什么？ 5](#_Toc16971448)

[并发编程三个特性？ 6](#_Toc16971449)

[volatile变量 8](#_Toc16971450)

[说一下 synchronized 底层实现原理？ 9](#_Toc16971451)

[多线程中 synchronized 锁升级的原理是什么？ 11](#_Toc16971452)

[内存屏障如何实现，JMM对内存屏障做了哪些优化，volatile的语义增强 14](#_Toc16971453)

[什么是线程池，如何使用？为什么要用线程池？线程池运行流程，参数，策略？ 15](#_Toc16971454)

[为什么netty的线程池中，线程的个数是cpu的2倍？ 18](#_Toc16971455)

[用过ThreadLocal没？它的原理是什么？ 19](#_Toc16971456)

[多线程实现线程安全的方式 21](#_Toc16971457)

[乐观锁和悲观锁的区别？ 21](#_Toc16971458)

[reenterlocker和sychronize的区别？ 22](#_Toc16971459)

[final的可见性 23](#_Toc16971460)

[This的引用逃逸了解吗？ 23](#_Toc16971461)

[线程的生命周期 23](#_Toc16971462)

[speep（）和 wait()的区别 25](#_Toc16971463)

[Wait和notify知识点： 25](#_Toc16971464)

[线程池的调优策略？ 26](#_Toc16971465)

[为什么ConcurrentHashMap 1.8为什么要使用CAS+Synchronized取代Segment+ReentrantLock 26](#_Toc16971466)

[一个线程接着一个线程执行 有什么方法？ 27](#_Toc16971467)

[不可重入锁实现 28](#_Toc16971468)

什么是线程？

答：线程是操作系统能够进行运算调度的最小单位，它被包含在进程之中，是进程中的实际运作单位。程序员可以通过它进行多处理器编程，你可以使用多线程对运算密集型任务提速。

线程和进程有什么区别？

答：线程是进程的子集，一个进程可以有很多线程，每条线程并行执行不同的任务。不同的进程使用不同的内存空间，而所有的线程共享一片相同的内存空间。别把它和栈内存搞混，每个线程都拥有单独的栈内存用来存储本地数据。

如何在Java中创建线程？

答：在语言层面有两种方式。可以继承java.lang.Thread类或者直接实现Runnable接口来重写run()方法实现线程

当然也可以通过线程池创建线程。

什么时候使用Runnable还是Thread?你自己一般通过什么方式创建线程？

答：这个问题是上题的后续，大家都知道我们可以通过继承Thread类或者调用Runnable接口来实现线程，问题是，那个方法更好呢？什么情况下使用它？这个问题很容易回答，如果你知道**Java不支持类的多重继承，但允许你调用多个接口。所以如果你要继承其他类，当然是调用Runnable接口好了**。

自己一般通过线程池的方式创建线程。

Thread 类中的start() 和 run() 方法有什么区别？

答：这个问题经常被问到，但还是能从此区分出面试者对Java线程模型的理解程度。start()方法被用来启动新创建的线程，而且start()内部调用了run()方法，这和直接调用run()方法的效果不一样。当你调用run()方法的时候，只会是在原来的线程中调用，没有新的线程启动，start()方法才会启动新线程。

java中Runnable和Callable有什么不同？

答：它们的主要区别是Callable的 call() 方法可以返回值和抛出异常， Callable可以返回装载有计算结果的Future对象。

Deamon线程

守护线程是在程序运行时，在后台提供的一种通用服务，比如垃圾回收线程，当程序结束之后，deamon线程也会被杀掉。

与普通线程的区别就是：守护线程在用户线程退出运行之后，守护线程才会退出。

Thread.start之前设置thread.setdaemon(true),否者会抛错误。

Java中CyclicBarrier 和 CountDownLatch有什么不同？

答：

第一：CountDownLatch是只触发一次的事件，而CyclicBarrier可以多次重用直到ExecutorServiced调用shutdownNow或者shutdown。

第二：可以向CyclicBarrier提供一个"栅栏动作"，它是Runnable,当计数值到达0时自动执行，而CountDownLatch没有提供，不过在之前调用await()的任务线程可以在CountDown为0之后继续执行。

　1）CountDownLatch和CyclicBarrier都能够实现线程之间的等待，只不过它们侧重点不同：

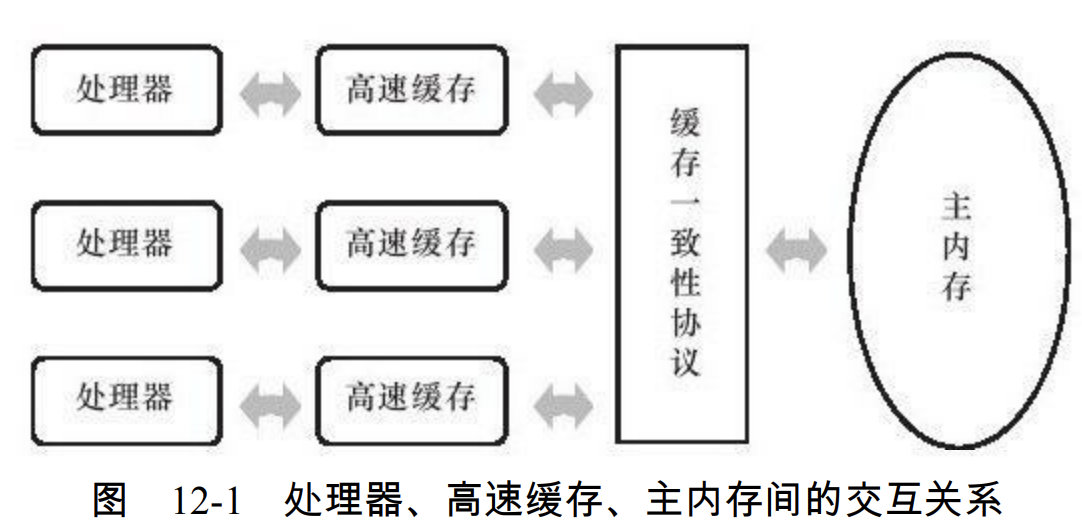
　　　　CountDownLatch一般用于某个线程A等待若干个其他线程执行完任务之后，它才执行；

　　　　而CyclicBarrier一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行；

　　　　另外，CountDownLatch是不能够重用的，而CyclicBarrier是可以重用的。

　　2）Semaphore其实和锁有点类似，它一般用于控制对某组资源的访问权限。

物理计算机的并发问题



由于cpu的处理速度和内存的读写速度有几个数量级的差距，所以计算机不得不加入高速缓存作为cpu和内存之间的缓冲：将运算要使用的数据复制到缓存中，让运算能快速进行，当运算结束之后，在将缓存同步到内存中，这样cpu就无需等待缓慢的内存读写了。

但是这样会引入一个问题：那就是缓存一致性问题，在多核系统中，由于cpu缓存数据更新时没有及时刷新到主内存中，这时就会导致不同处理器缓存的数据不一致的问题，为了解决缓存不一致的问题，就提出了对总线加锁或者缓存一致性协议，例如MESI

解决方案有：

1.通过在总线加LOCK#锁的方式

原理是通过锁住总线，保证每次只有一个cpu核能访问内存，其他cpu全部停止执行，只有该cpu完全执行完之后，其他cpu才能从内存中读取共享变量，但是在锁住总线期间，其他cpu无法访问内存，会导致效率低下。

2.通过缓存一致性协议

例如：MESI协议，MESI协议保证了每个缓存中使用的共享变量的副本是一致的。它核心的思想是：当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本（浅拷贝），会发出信号通知其他CPU将该变量的缓存行置为无效状态，因此当其他CPU需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它就会从主内存重新读取。

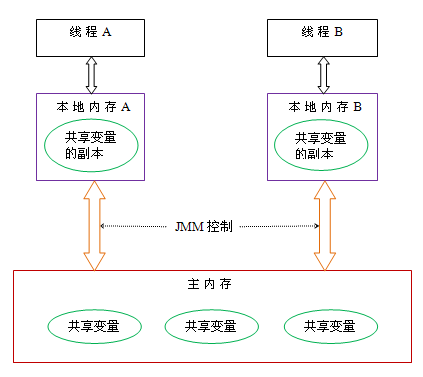
思考：既然cpu自己能够做到通过缓存一致性机制，保证各个处理器中的各自缓存和内存一致，那就不存在并发问题了，那Java就没有必要做内存模型去控制并发问题了？

思考答案：其实缓存一致性并不能保证立即的内存可见性，CPU为了提高效率，不会时时刻刻去刷新缓存，而是在合适的时机才会刷新缓存，只有在JVM声明了锁操作的时候（也就是Lock#信号或者内存屏障），才会立即刷新缓存。

Java内存模型是什么？

1.java内存模型(JMM)是线程间通信的控制机制.JMM定义了主内存和线程之间抽象关系。Java内存模型规定所有共享变量都存储在主内存（和物理机的主内存类比）中，每个线程都有自己的工作内存（与物理机的cpu高速缓存类比），当线程需要操作某个共享变量的时候，会把主内存中的共享变量浅拷贝一份到线程的工作内存来，再进行操作。

本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。Java内存模型的抽象示意图如下：



从上图来看，线程A与线程B之间如要通信的话，必须要经历下面2个步骤：

首先，线程A把本地内存A中更新过的共享变量刷新到主内存中去。

 然后，线程B到主内存中去读取线程A之前已更新过的共享变量。

2.当多个线程同时读写某个内存数据时，就会产生多线程并发问题，涉及到三个特 性：原子性，有序性，可见性。java从语法层面，应该给开发人员提供一种线程同步解决方案，这个方案就是诸如synchronized, volatile,锁机制（如同步块，就绪队 列，阻塞队列）等等。

* 原子性：是指一个操作或者多个操作 要么全部执行，要么就都不执行。java内存模型可以保证基本数据类型的读写是具备原子性的，java内存模型还提供了锁和sychronized保证对多个操作的原子性
* 可见性：是指当一个线程修改了共享变量的值，其他线程能立即看到修改的值，在java中，volatile变量，synchronized，lock，final都可以保证线程可见性。
* 有序性：为了优化程序性能，编译器和CPU会对代码进行指令重排序，在并发的过程中，是有可能出现线程安全问题的，在java中，volatile，sychronized，lock保证了有序性。

3.Java内存模型不可能总是通过volatile，sychronized保证有序性的，因此JMM具备一些先天的“有序性”：**先行发生原则**。即不需要通过任何手段就能够得到保证的有序性。如果两个操作的执行次序无法从先行发生原则推导出来，那么它们就不能保证它们的有序性，虚拟机可以随意地对它们进行重排序。在Java内存模型中，允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。

**下面就来具体介绍下先行发生原则：**

①程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作 （单线程无影响，多线程可能会发生指令重排序，对结果产生影响）

②锁定规则：一个unLock操作先行发生于后面对同一个锁的lock操作，在解锁之前会把线程中的工作内存写进主内存

③volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作 （比较重要的规则。直观地解释就是，如果一个线程先去写一个变量，然后一个线程去进行读取，那么写入操作肯定会先行发生于读操作。）

④传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C （体现happens-before原则具备传递性）

⑤线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作

⑥线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生

⑦线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行

⑧对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始

这8条规则中，前4条规则是比较重要的，后4条规则都是显而易见的。

并发编程三个特性？

**并发编程的三大概念：原子性，有序性，可见性**

**要想并发程序正确地执行，****必须要保证原子性、可见性以及有序性。只要有一个没有被保证，就有可能会导致程序运行不正确。**

* 原子性
  + 定义：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。
  + java中的原子性：**Java内存模型只保证了基本读取(print(x))和赋值（x=10）是原子性操作，如果要实现更大范围操作的原子性，可以通过synchronized和Lock来实现。由于synchronized和Lock能够保证任一时刻只有一个线程执行该代码块，那么自然就不存在原子性问题了，从而保证了原子性。**
  + 举例：之前做个一个实验，用十个线程对一个volatile修饰的一个共享变量x使用x++操作1000次，最后，这个数的值并不是10000，而是少于10000，这就是因为++操作并不是原子性的，导致，结果不准确。
* 可见性
  + 定义：可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。
  + java中的可见性：
    - Java提供了**volatile关键字**来保证可见性。当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。而普通的共享变量不能保证可见性，因为普通共享变量被修改之后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性。
    - **通过synchronized和Lock也能够保证可见性，synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中。因此可以保证可见性。**
  + 举例：这个问题我自己碰到过，当时用jemeter启动10个线程去启动hgos的时候，hgos启动时间比较长，这时当有一个线程一旦去启动hgos的时候状态会立马从可用变成正在启动中，其他有9个线程再进来，会报hgos正在启动中的错误，结果是只有4，5个才报错，这个时候通过把启动hgos的这段代码用synchronized锁住，防止了可见性的问题产生。
* 有序性
  + 定义：即程序执行的顺序按照代码的先后顺序执行。（指令重排序不会影响单个线程的执行，但是会影响到线程并发执行的正确性。）
  + java中的有序性：
    - 在Java内存模型中，允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。
    - 在Java里面，**可以通过volatile关键字来保证一定的“有序性”。另外可以通过synchronized和Lock来保证有序性，很显然，synchronized和Lock保证每个时刻是有一个线程执行同步代码，相当于是让线程顺序执行同步代码，自然就保证了有序性。**
  + 举例：在单例模式中的双重锁书写的例子中，如果不用volatile修饰单例对象，还是有可能多个单例，因为创建对象的过程分为三步。第一步，给instance分配内存,第二步，调用 Singleton 的构造函数来初始化成员变量，第三步，将instance对象指向分配的内存空间， JVM 的即时编译器中存在指令重排序的优化。也就是说上面的第二步和第三步的顺序是不能保证的，最终的执行顺序可能是 1-2-3 也可能是 1-3-2。如果是后者，则在 3 执行完毕、2 未执行之前，被线程二抢占了，这时 instance 已经是非 null 了（但却没有初始化），所以线程二会直接返回 instance，然后使用，然后顺理成章地报错。

### 理解原子性和可见性的区别

### **理解原子性和可见性（易变性）的概念这一点很重要。在非volatile域上的原子操作不必刷新到主存中。在缓存中完成原子操作，和把缓存数据刷新到主存中是两码事。**

### **当一个域的值依赖于它之前的值时（例如递增一个计数器），volatile就无法工作了。如果某个域的值受到其他域的值的限制，那么volatile也无法工作，例如Rangle类的lower和upper边界必须遵循lower<=upper的限制。**

### **使用volatile而不是synchronized的唯一安全的情况是类中只有一个可变的域。再次提醒，你的第一选择应该是使用synchronized关键字或者Lock，这是最安全的方式，而尝试其他任何方式都是有风险的。**

### **什么才是属于原子操作呢？对域中的值（除long和double 64 位之外的基本数据类型）做简单赋值和返回操作通常是原子性的。对基本类型的读取和复制操作被认为是安全的原子性操作。**

volatile变量

提到volatile就要提到java内存模型在并行时要处理的三大特性，要保证线程安全，就要保证有序性，可见性，原子性。Volatile关键字是用来保证有序性和可见性的，不保证原子性。

**有序性：**比如我们写的代码，不一定是按照我们自己书写的顺序来执行的，**编译器和CPU为了提高CPU执行效率会对指令进行重排序，需要一定的顺序和规则来保证，所有就有先行发生原则，其中就有条就是volatile变量规则：对一个volatile变量的写操作先行发生于后面对这个volatile变量的读操作；**有序性是通过插入内存屏障来保证的。

**可见性：**首先Java内存分主内存和每个线程的工作内存，当多个线程对共享变量操作时，比如线程A会把共享变量i从主内存中读取到自己的工作内存中，做了加1操作，但是此时没有把i的最新值刷新到主内存中，线程B此时读到的还是i的旧值。**当多个线程对共享变量操作时，volatile保证在一个线程对该变量修改之后会立马将新值刷新到主存，保证主存的值是最新的**，并且是通过缓存一致性协议保证其他线程在读取该值的时候，会发现缓存行无效，之后会重新从主存中读取该变量。

**为啥说volatile不保证原子性呢？** 比如多个线程对共享变量i做自增操作，自增操作分为三步：

1.读取变量的初始值（如果是第一次，还要讲该变量copy一份到工作内存中）

2.Cpu进行加1操作

3.Cpu将修改后的值写入工作内存。

假如当前i值为10，线程A从主存中读取到i值之后，线程A阻塞，此时，线程B抢占资源，也从主存读取i值，并且在+1操作，立即将i写入主存，此时已经为11了，这时，线程A不阻塞了，会刷到内存的i值也为11，而不是12.

解决方案：

* 可以通过synchronized或lock，进行加锁，来保证操作的原子性。
* 也可以通过AtomicInteger。atomic是利用CAS来实现原子性操作的（Compare And Swap），CAS实际上是利用处理器提供的CMPXCHG指令实现的，而处理器执行CMPXCHG指令是一个原子性操作

**volatile使用场景：**

volatile最适用一个线程写，多个线程读的场合，结合使用 volatile 和 synchronized 实现 “开销较低的读－写锁”，比如concurrenthashmap的保证线程安全设计。

**正确使用 volatile 变量的条件**

您只能在有限的一些情形下使用 volatile 变量替代锁。要使 volatile 变量提供理想的线程安全，必须同时满足下面两个条件：

* 对共享变量的写操作不依赖于共享变量的当前值。i++
* 该变量没有包含在具有其他变量的不变式中。

**一个变量如何从主内存拷贝到工作内存，又如何从工作内存同步回主内存？**

一般一个线程对变量的操作需要经历从主内存中读取Read--->加载到工作线程中load---->在工作线程中使用use--->该该共享变量进行赋值asign--->对该共享变量进行存储store--->将该共享变量回写到主内存中write。

**volatile实现**

观察加入volatile关键字和没有加入volatile关键字时所生成的汇编代码发现，加入volatile关键字时，会多出一个lock前缀指令

lock前缀指令实际上相当于一个内存屏障（也成内存栅栏），内存屏障会提供2个功能

1）禁止屏障两侧的指令重排序；

2）强制将对缓存的修改操作立即写入主存，保证让前面volatile变量的修改对其他CPU立即可见

具体看：内存屏障章节

说一下 synchronized 底层实现原理？

<http://bigdatadecode.club/JavaSynchronizedTheory.html>

简单说下：

Synchronized关键字经过编译之后，会在同步块的前后形成monitorenter和monitorexit这两个字节码指令，这两个字节码都需要一个引用类型的参数来指明要锁定和解锁的对象，synchronized语句指明的这个对象，如果没有指明，那根据synchronized修饰的实例方法还是类方法，去取对应的对象实例或者class对象来做锁对象。

在执行monitorenter指令的时候，首先要尝试获取对象的锁，如果这个对象没被锁定，或者当前线程已经拥有了那个对象的锁，那把锁的计数器加1，相应地，在执行monitorexit指令时，会将锁计数器减1，当计数器为0时，锁就释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到对象锁被另外一个线程释放为止。

详细说下：

（monitor就是锁的实现方式）

synchronized 语句块是由一对 monitorenter/monitorexit 指令来显示获取monitor对象和释放monitor对象来实现的，

sychronnized方法是通过判断是否设置常量池中ACC\_SYNCHRONIZED标示符来隐式地获得monitor对象和释放monitor对象来实现的，

不管是显示还是隐式的获取monitor，本质没有区别。

说到底synchronized是通过一个获取和释放monitor对象来实现的。每个对象和类都和一个监视器(monitor)相关联，锁住了一个对象，就是获得对象相关联的监视器。当monitor被占用时就会处于锁定状态，例如monitorenter/monitorexit：

* 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。
* 执行monitorexit的线程必须是objectref所对应的monitor的所有者。
  1. 指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个monitor的所有权。

通过这两个指令我们应该能很清楚的看出Synchronized的实现原理，Synchronized的语义底层是通过一个monitor对象来完成，其实wait/notify等方法也依赖于monitor对象，这就是为什么只有在同步的块或者方法中才能调用wait/notify等方法，否则会抛出java.lang.IllegalMonitorStateException的异常的原因。

什么是Monitor对象？它是一种同步机制，monitor是由ObjectMonitor实现的，其主要数据结构有四个属性，如下：

* \_count用来记录该线程获取锁的次数
* \_WaitSet存放处于wait状态的线程队列
* \_EntryList存放处于等待获取锁block状态的线程队列，即被阻塞的线程
* \_owner指向持有ObjectMonitor对象的线程

当多个线程同时访问一段同步代码时，首先会进入\_EntryList队列中，当某个线程获取到对象的monitor后进入\_Owner区域并把monitor中的\_owner变量设置为当前线程，同时monitor中的计数器\_count加1，若线程调用wait()方法，将释放当前持有的monitor，\_owner变量恢复为null，\_count自减1，同时该线程进入\_WaitSet集合中等待被唤醒。若当前线程执行完毕也将释放monitor(锁)并复位变量的值，以便其他线程进入获取monitor(锁)。如下图所示



在 Java 6 之前，synchronized是重量级锁，因为monitor对象的实现依赖于底层的操作系统的互斥锁来实现的，而操作系统实现线程之间的切换时需要从用户态转换到核心态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的synchronized效率低的原因。Java 6之后，为了减少获得锁和释放锁所带来的性能消耗，引入了偏向锁、轻量级锁和自旋锁，锁消除，锁粗化等概念，大大改进了性能。

多线程中 synchronized 锁升级的原理是什么？

**synchronized 锁升级原理：**在锁对象的对象头里面有一个 threadid 字段，在第一次访问的时候 threadid 为空，jvm 让其持有偏向锁，并将 threadid 设置为其线程 id，再次进入的时候会先判断 threadid 是否与其线程 id 一致，如果一致则可以直接使用此对象，如果不一致，则升级偏向锁为轻量级锁，通过自旋循环一定次数来获取锁，执行一定次数之后，如果还没有正常获取到要使用的对象，或者这时又有一个线程3过来竞争这个锁对象，此时就会把锁从轻量级升级为重量级锁，此过程就构成了 synchronized 锁的升级。

锁的升级的目的：线程的阻塞和唤醒需要CPU从用户态转为核心态，频繁的阻塞和唤醒会耗性能。锁升级是为了减低了锁带来的性能消耗。在 Java 6 之后优化 synchronized 的实现方式，使用了偏向锁升级为轻量级锁再升级到重量级锁的方式，从而减低了锁带来的性能消耗。

**为什么优化？**

线程的阻塞和唤醒需要CPU从用户态转为核心态，频繁的阻塞和唤醒会耗性能。

**为什么要引入偏向锁？**

同步块在大多数时候是不存在锁竞争的，常常是一个线程多次获得同一个锁，因此如果每次都要竞争锁会增大很多没有必要付出的代价，为了降低获取锁的代价，才引入的偏向锁。

**为什么要引入轻量级锁？****自旋锁和自适应自旋**

轻量级锁考虑的是竞争锁对象的线程不多，而且线程持有锁的时间也不长的情景。*因为阻塞线程需要CPU从用户态转到内核态，代价较大*，如果刚刚阻塞不久这个锁就被释放了，那这个代价就有点得不偿失了，因此这个时候就干脆不阻塞这个线程，让它自旋这等待锁释放。

**自旋锁和自适应自旋：**

* 自旋锁：为了避免进入内核态，在多个线程并发的时候，通过让阻塞的线程等待，只需要让线程执行一个忙循环，即自旋，这就是自旋锁。自旋的时间要控制在一定次数内，不然会浪费cpu资源。
* 自适应自旋：自适应的意思是下一次自旋锁的自旋时间由上一次在同一个锁上自旋时间及锁的拥有者状态来决定。如果在同一个锁对象刚刚通过自旋获取锁，那么下次自旋可能再次成功，因此下次自旋会等待相对更长的时间，如果对某个锁很少获得成功，则以后获取这个锁就省去自旋过程，以避免浪费处理器资源。

**锁消除和锁粗化**

**偏向锁，轻量级锁，重量级锁，这些锁是如何存储的？**

这些锁的存储，就要说到Java对象头

下面重点说下java对象头。

众所周知Java中万物皆对象，那对象在内存中是怎么存储的呢？

每个对象分为三块区域:对象头、实例数据和对齐填充。

* 对象头包含两部分，第一部分是Mark Word，用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳等等，这一部分占一个字节。第二部分是Klass Pointer（类型指针），是对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例，这部分也占一个字节。(如果对象是数组类型的，则需要3个字节来存储对象头，因为还需要一个字节存储数组的长度)
* 实例数据存放的是类属性数据信息，包括父类的属性信息，如果是数组的实例部分还包括数组的长度，这部分内存按4字节对齐。
* 填充数据是因为虚拟机要求对象起始地址必须是8字节的整数倍。填充数据不是必须存在的，仅仅是为了字节对齐。

对象头信息是与对象自身定义的数据无关的额外存储成本，考虑到虚拟机的空间效率，Mark Word被设计成一个非固定的数据结构以便在极小的空间内存储尽量多的信息，它会根据对象的状态复用自己的存储空间。例如在32位的HotSpot虚拟机 中对象未被锁定的状态下，Mark Word的32个Bits空间中的25Bits用于存储对象哈希码(HashCode)，4Bits用于存储对象分代年龄，2Bits用于存储锁标志 位，1Bit固定为0，在其他状态(轻量级锁定、重量级锁定、GC标记、可偏向)下对象的存储内容如下表所示对象头存储结构。  


Synchronized通常被称为重量级锁，但是1.6之后对其进行优化，新增了轻量级锁和偏向锁，这里重点说下重量级锁，随后对Synchronized的优化简单介绍下。

从对象头的存储内容可以看出锁的状态都保存在对象头中，Synchronized也不例外，当其从轻量级锁膨胀为重量级锁时，锁标识位为10，其中指针指向的是monitor对象(也称为管程或监视器锁)的起始地址。

这几种锁的优缺点（偏向锁、轻量级锁、重量级锁）



内存屏障如何实现，JMM对内存屏障做了哪些优化，volatile的语义增强

内存屏障，又称内存栅栏，是一个CPU指令，它的作用有两个，*一是保证特定操作的执行顺序，二是保证某些变量的内存可见性*（利用该特性实现volatile的内存可见性）。由于编译器和处理器都能执行指令重排优化。如果在指令间插入一条Memory Barrier则会告诉编译器和CPU，不管什么指令都不能和这条Memory Barrier指令重排序，也就是说通过插入内存屏障禁止在内存屏障前后的指令执行重排序优化。Memory Barrier的另外一个作用是强制刷出各种CPU的缓存数据，因此任何CPU上的线程都能读取到这些数据的最新版本。总之，volatile变量正是通过内存屏障实现其在内存中的语义，即可见性和禁止重排优化。

**内存屏障是什么**

* 硬件层的内存屏障分为两种：Load Barrier 和 Store Barrier即读屏障和写屏障。
* 内存屏障有两个作用：

1. 阻止屏障两侧的指令重排序；
2. 强制把写缓冲区/高速缓存中的脏数据等写回主内存，让缓存中相应的数据失效。

* 对于Load Barrier（读屏障）来说，在指令前插入Load Barrier，可以让高速缓存中的数据失效，强制从新从主内存加载数据；
* 对于Store Barrier（写屏障）来说，在指令后插入Store Barrier，能让写入缓存中的最新数据更新写入主内存，让其他线程可见。

 java的内存屏障通常所谓的四种即LoadLoad,StoreStore,LoadStore,StoreLoad实际上也是上述两种的组合，完成一系列的屏障和数据同步功能。

 LoadLoad屏障：对于这样的语句Load1; LoadLoad; Load2，在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。 **在后续读操作执行之前，****之前的读操作要读取的数据已被读取。**

 StoreStore屏障：对于这样的语句Store1; StoreStore; Store2，在Store2及后续写入操作执行前，保证Store1的写入操作对其它处理器可见。**在后续写操作执行之前，****之前****的写操作已对其他处理器可见。**

 LoadStore屏障：对于这样的语句Load1; LoadStore; Store2，在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。 **在后续写操作执行之前，之前的读操作要读的数据已被读取**

 StoreLoad屏障：对于这样的语句Store1; StoreLoad; Load2，在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。**在后续读操作执行之前，之前的写操作已对其他处理器可见。*它的开销是四种屏障中最大的。在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能***

***Volitle内存屏障规则：由于内存屏障的作用，避免了volatile变量和其它指令重排序、线程之间实现了通信，使得volatile表现出了锁的特性。***

内存屏障规则： SS,写操作,SL, 读操作,LL, 读操作，LS

（1）在每一个volatile写操作前面插入一个StoreStore屏障。这确保了在进行volatile写之前前面的所有普通的写操作都已经刷新到了内存。

（2）在每一个volatile写操作后面插入一个StoreLoad屏障。这样可以避免volatile写操作与后面可能存在的volatile读写操作发生重排序。

（3）在每一个volatile读操作后面插入一个LoadLoad屏障。这样可以避免volatile读操作和后面普通的读操作进行重排序。

（4）在每一个volatile读操作后面插入一个LoadStore屏障。这样可以避免volatile读操作和后面普通的写操作进行重排序。

*volatile的内存屏障插入策略非常保守，其实在实际中，只要不改变volatile写-读得内存语义，编译器可以根据具体情况优化，省略不必要的屏障。*

比如在volatile读操作后面，还是volatile读的话，那么前一个volatile读操作后面的LL，由于禁止下面所有的普通写与上面的volatile读重排序，但是由于存在第二个volatile读，那个普通的读根本无法越过第二个volatile读。所以可以省略。

再比如在volatile写操作后面，还是volatile写的话，那么前一个volatile写操作后面的SL，可以省略，因为后面跟的是volatile写，没有读操作。

再比如：在volatile读操作后面，是普通写的话，volatile读操作后面的LL就可以省了。因为下面不存在普通读了。

什么是线程池，如何使用？为什么要用线程池？线程池运行流程，参数，策略？

**为甚要用线程池？**

为了避免在执行大量异步任务的时候频繁的创建和销毁线程，所以才使用线程池。

**讲讲java线程池**

线程池参数主要有核心线程数，最大线程池数，任务队列，拒绝策略，空闲线程存活时长，线程工厂。

1.核心线程池，最大线程池，拒绝策略

1. 当运行的线程数少于corePoolSize核心线程数，则创建线程来处理任务，即使核心线程中有空闲的线程，也照样创建新的线程来处理任务。
2. 当运行的线程数大于corePoolSize核心线程数而少于maximumPoolSize最大线程数，且任务队列未满时，将任务放入任务队列。
3. 当运行的线程大于corePoolSize核心线程数，且任务队列已满
   1. 若运行的线程数小于最大线程数，创建线程。
   2. 若运行的线程数等于最大线程数，则执行拒绝策略。
      1. 丢弃，并抛RejectedExecutionException异常。（默认）
      2. 丢弃但不抛弃异常
      3. 丢弃最老的任务，再重试。
      4. 由调用executor方法自己的线程运行任务。这提供了一个简单的反馈控制机制，可以减慢新任务提交的速度。

2.任务队列，有三种类型：

1. 直接切换。（例如：SynchronousQueue）
   1. 工作队列默认是synchronousQueue，它将任务转手给没有其他任务持有的线程,在此，如果存在一个可立即运行任务的线程时，则加入队列，但是立即被可立即运行任务的线程给拿去运行，如果没有可立即运行任务的线程时，则试图把任务加入队列，此操作将会失败，因此会创建一个线程来运行任务。
   2. 此策略可以避免在处理可能具有内部依赖性的任务集时出现死锁的现象。直接切换策略通常要求maximumPoolSizes是无界的，以防止拒绝新提交的任务,反过来说，当任务持续到达的速度比线程池通常任务处理的速度更快时，线程则可能无边界的增长。
2. 无界队列（例如：LinkedBlockingQueue）
   1. 无界队列使用的队列是无特定容量的，例如LinkedBlockingQueue，无界队列将会导致在所有的核心线程都在忙碌时，新任务在都会在队列的等待。因此，所有创建的线程不会超过corePoolSize核心线程数。这样的话maximumPoolSize最大线程数的值也就是无效的。
   2. 应用场景：当每个任务完全独立于其他任务，即任务执行互不影响时，则适合使用无界队列；例如网页服务器。
   3. 然而这种排队方式对于解决短暂爆发的任务处理是有用的。当任务持续到达线程池的速度大于线程池通常任务处理的速度平均数时，任务队列将可能无边界的增长。
3. 有界队列（例如：ArrayBlockingQueue）
   1. 当使用一个有界的maximumPoolSizes最大线程数时，有界队列能够防止资源消耗殆尽。但是它更难去控制。
   2. 这就要权衡队列大小和最大线程数的大小了：
      1. 使用大型队列和小型池的话，将会降低CPU使用率，减少操作系统资源的使用和减少任务上下文切换的开销，但是会人为地导致低吞吐量。如果任务频繁地阻塞，其他任务等待时间也会变的更长。
      2. 使用小型队列和大型池，CPU使用率将会增加，同时因为大量任务在CPU上来回调度，就可能遇到不可接受的任务调度开销，也会降低吞吐量。

3.存活时间-keepAliveTime

如果池中当前有多于corePoolSize 的线程，则这些多出的线程在空闲时间超过 keepAliveTime 时将会终止。这提供了当池处于非活动状态时减少资源消耗的方法。如果池后来变得更为活动，则可以创建新的线程。

注意：

如果把值设为Long.MAX\_VALUE TimeUnit.NANOSECONDS 的话，空闲线程不会被回收直到ThreadPoolExecutor为Terminate。

默认情况下，保持活动策略只在有多于corePoolSizeThreads 的线程时应用。

只要 keepAliveTime 值非 0，并且把allowCoreThreadTimeOut设为true，也可将此超时策略应用于核心线程。

4.线程工厂

如果没有另外说明，则使用 Executors.defaultThreadFactory() 创建线程

5.线程池的关闭

ThreadPoolExecutor提供了两个方法，用于线程池的关闭，分别是shutdown()和shutdownNow()，其中：

* shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务
* shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

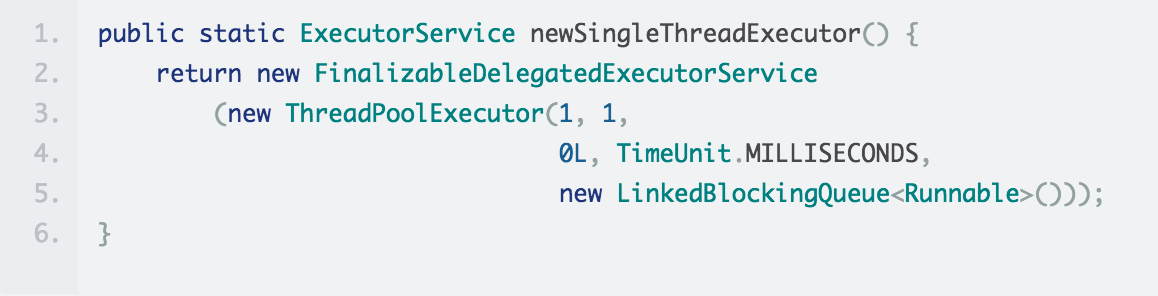
4. Eecutors中工厂方法的配置

1. Eecutors.newFixedThreadPool(n)



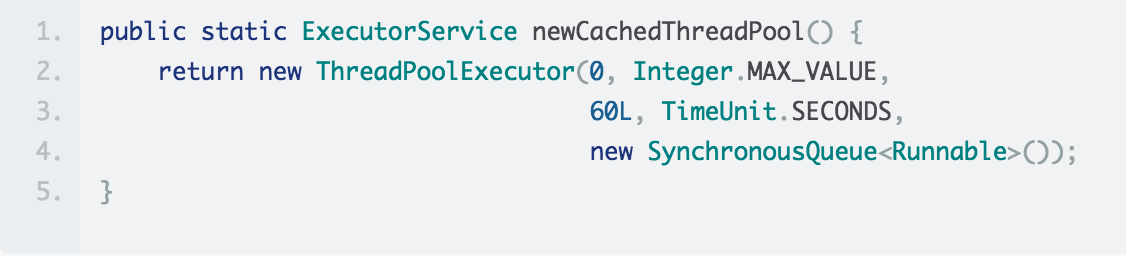
该工厂方法创建了一个固定大小为nThreads的任务队列，无界的任务队列的线程池，如果不设置allowCoreThreadTimeOut设为true，keepalivetime无意义，如果allowCoreThreadTimeOut设为true，则核心线程中一旦有空闲的线程，就会被销毁掉。

1. Eecutors.newSingleThreadExecutor()



该工厂方法创建了一个固定大小为1的任务队列，无界的任务队列的线程池，如果不设置allowCoreThreadTimeOut设为true，keepalivetime无意义，如果allowCoreThreadTimeOut设为true，则核心线程中一旦有空闲的线程，就会被销毁掉。

1. Eecutors.newCachedThreadPool()



该工厂方法创建了一个无界线程池，并且可以进行自动线程回收。keepalivetime是60，即表示，空闲线程空闲时间达到60秒则进行回收。

为什么netty的线程池中，线程的个数是cpu的2倍？

在IO优化中，这样的估算公式可能更适合：

线程执行计算，占用CPU的时间，

线程等待时间，不占用CPU的时间，执行IO

最佳线程数目 = （（线程等待时间/线程执行时间）+ 1）\* CPU数目

有个属性叫ioratio,作用是在一个eventloop中花在IO事件上的时间比例，默认值是50，设置50的目的是尝试保证在在eventloop中花在IO事件上的时间和非IO事件(CPU计算时间)的时间比例是1:1.

因为很显然，线程等待时间所占比例越高，需要越多线程。线程CPU时间所占比例越高，需要越少线程。

*Sets the percentage of the desired amount of time spent for I/O in the event loop. The default value is  
\* {****@code*** *50}, which means the event loop will try to spend the same amount of time for I/O as for non-I/O tasks.*

在netty中的eventloopthread的run方法是eventloopthread线程的主体，做的事情分为对三个步骤不断循环：

1. 首先轮询注册到reactor线程对用的selector上的所有的channel的IO事件

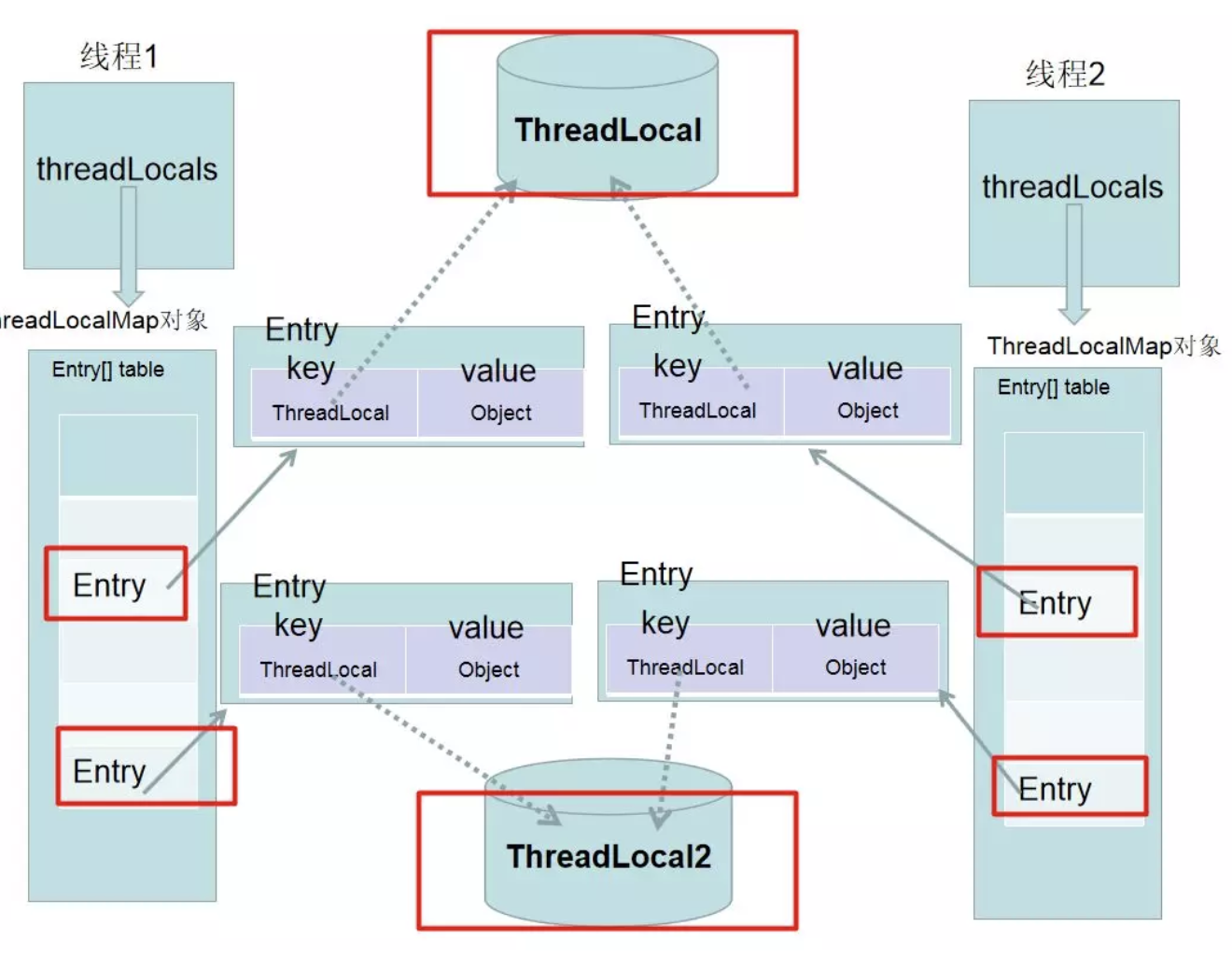
2. 处理产生网络IO事件的channel

3. 处理任务队列

在第二步和第三步开始前加了个判断,

* 如果ioratio等于100，直接处理所有IO时间和处理所有的任务队列，这会导致如果eventloop线程在处理任务队列的停留的时间过长，那么将积攒许多的IO事件无法处理，最终导致大量客户端请求阻塞。这条路一般不会走。
* 如果ioratio不等于100，默认是50，则会保证处理io事件的时间和处理任务队列的时间是相等的，第一，不会阻塞IO事件，第二，在IO优化中，有这么个公式，最佳线程数目 = （（线程等待时间/线程执行时间）+ 1）\* CPU数目，这样应用的性能会最优。所以netty进行了这个设计。­
  + 线程执行计算，占用CPU的时间，
  + 线程等待时间，不占用CPU的时间，执行IO

用过ThreadLocal没？它的原理是什么？



ThreadLocal是解决线程安全问题一个很好的思路，它通过为每个线程提供一个独立的变量副本解决了变量并发访问的冲突问题。在很多情况下，ThreadLocal比直接使用synchronized同步机制解决线程安全问题更简单，更方便，且结果程序拥有更高的并发性。

ThreadLocal是用于解决多线程并发问题的，

* 线程类里有个threadlocal.threadlocalMap类型的threadlocals属性变量，每个线程都有一个自己的ThreadLocalMap，
* Map中存储着一个个entry，每个entry由一个threadlocal会作为Key，变量副本作为value，所以每个线程往这个ThreadLocal中读写是隔离的，并且是互相不会影响的。
* 一个theadlocal只能存储一个对象，如果需要存储多个Object对象那么就需要多个ThreadLocal。

注意：通过线性探测的方式解决冲突，如果有大量不同的ThreadLocal对象放入map中时发送冲突，或者发生二次冲突，则效率很低。所以FTL使用了数组对此进行优化。

注意：由于ThreadLocalMap的key是弱引用，而Value是强引用。这就导致了一个问题，ThreadLocal在没有外部对象强引用时，发生GC时弱引用Key会被回收，而Value不会回收，如果创建ThreadLocal的线程一直持续运行，那么这个Entry对象中的value就有可能一直得不到回收，发生内存泄露。

* 如何避免泄漏：既然Key是弱引用，那么我们要做的事，就是在调用ThreadLocal的get()、set()方法时完成后再调用remove方法，将Entry节点和Map的引用关系移除，这样整个Entry对象在GC Roots分析后就变成不可达了，下次GC的时候就可以被回收。

注意：threadlocal对象应该使用static修饰，static的threadlocal变量能够保证，在一个线程内，static变量是被各个实例共同引用，但是在不同线程内，static变量是隔开的。

注意：FastThreadLocal

FastThreadLocal操作元素的时候，使用常量下标在数组中进行定位元素来替代ThreadLocal通过哈希和哈希表，这个改动特别在频繁使用的时候，效果更加显著！计算该ThreadLocal需要存储的位置是通过hash算法确定位置：

并且FastThreadLocal利用缓存行的特性，FastThreadLocal是通过indexedVariables数组进行存储数据的，如果有多个FastThreadLocal的时候，也可以利用缓存行，比如一次indexedVariables数组第3个位置数据，由于缓存的最小单位是缓存行，顺便把后面的4、5、6等也缓存了，下次刚刚好改线程需要读取另外的FastThreadLocal，这个FastThreadLocal的下标就是5的时候，进行读取的时候就直接走缓存了，比走主存可能快2个数量级而ThreadLocal通过hash是分散的。

引用场景：数据库连接，每个线程访问数据库都应当是一个独立的Session会话，如果多个线程共享同一个Session会话，有可能其他线程关闭连接了，当前线程再执行提交时就会出现会话已关闭的异常，导致系统异常。此方式能避免线程争抢Session，提高并发下的安全性。

使用ThreadLocal的典型场景正如上面的数据库连接管理，线程会话管理等场景，只适用于独立变量副本的情况，如果变量为全局共享的，则不适用在高并发下使用。

多线程实现线程安全的方式

1. 互斥同步(悲观锁)

1. sychronized

2. Reentrantlock

2. 非阻塞同步(乐观锁)

1. CAS：CAS指令需要有三个操作数，分别是内存位置（在java中可以简单理解为变量的内存地址用V表示），旧的预期值（用A表示）和新值（用B表示）。CAS指令执行时，当且仅当内存地址V等于预期值A时，处理器用新值B更新V的值，否知它就不执行更行，但是无论是否更行了V的值，都会返回V的值，并且这三个操作是原子性的，比锁的切换是要快很多的，并且如果则线程更新失败,失败的线程不会被挂起,通过自旋会重新准备再次尝试更新。

3. 无同步方案

1. 可重入代码

2. Threadlocal

乐观锁和悲观锁的区别？

**悲观锁（Pessimistic Lock）：**

每次获取数据的时候，都会担心数据被修改，所以每次获取数据的时候都会进行加锁，确保在自己使用的过程中数据不会被别人修改，使用完成后进行数据解锁。由于数据进行加锁，期间对该数据进行读写的其他线程都会进行等待。

代表：sychronize，lock

**乐观锁（Optimistic Lock）：**

每次获取数据的时候，都不会担心数据被修改，所以每次获取数据的时候都不会进行加锁，但是在更新数据的时候需要判断该数据是否被别人修改过。如果数据被其他线程修改，则不进行数据更新，如果数据没有被其他线程修改，则进行数据更新。由于数据没有进行加锁，期间该数据可以被其他线程进行读写操作。

实现：CAS实现和version方式

version方式：一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

核心SQL代码：

|  |  |
| --- | --- |
| 1 | update table set x=x+1, version=version+1 where id=#{id} and version=#{version}; |

CAS操作方式：即compare and swap 或者 compare and set，涉及到三个操作数，数据所在的内存值，预期值，新值。当需要更新时，判断当前内存值与之前取到的值是否相等，若相等，则用新值更新，若失败则重试，一般情况下是一个自旋操作，即不断的重试。

适用场景：

悲观锁：比较适合写入操作比较频繁的场景，如果出现大量的读取操作，每次读取的时候都会进行加锁，这样会增加大量的锁的开销，降低了系统的吞吐量。

乐观锁：比较适合读取操作比较频繁的场景，如果出现大量的写入操作，数据发生冲突的可能性就会增大，为了保证数据的一致性，应用层需要不断的重新获取数据，这样会增加大量的查询操作，降低了系统的吞吐量。

reenterlocker和sychronize的区别？

Synchronized和reenterlocker在基本用法上，很相似，都具有重入性，只是代码写法上有点区别，一个表现在API层面的互斥锁，一般在try{}finally{}中搭配使用的，另一个表现是原生语法层面的互斥锁，不过相比于synchronized，reenterlocker增加一些高级特性，主要是，等待可中断，可实现公平锁，以及锁可以绑定多个条件：

1. 等待可中断，持有锁的线程长期不释放的时候，正在等待的线程可以选择放弃等待，这相当于Synchronized来说可以避免出现死锁的情况。通过lock.lockInterruptibly()来实现这个机制。
2. 公平锁，多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁，可以防止饥饿，非公平锁则当锁释放的时候，其他线程随机抢占锁。Synchronized锁非公平锁，ReentrantLock默认的构造函数是创建的非公平锁，可以通过参数true设为公平锁，但公平锁表现的性能不是很好。
3. 锁可以绑定多个条件：一个ReentrantLock对象可以同时绑定多个condition。而sychronized，锁对象的wait和notify或者notifyall方法可以实现一个隐含的条件，如果要和多于一个的条件关联的话，就不得不额外加一把锁，而reentrantlock只需要多次调用newcondition方法既可
   1. await(); 造成当前线程在被中断和释放前一直处于等待状态。
   2. signal();随机唤醒一个等待的线程。
   3. signAll();唤醒所有等待的线程

**性能方面：**

* ReentrantLock属于乐观锁思想,是非阻塞式同步的,是轻量级的,其内部的CAS就是基于乐观锁的思想实现的,
  + CAS指令需要有三个操作数，分别是内存位置（在java中可以简单理解为变量的内存地址用V表示），旧的预期值（用A表示）和新值（用B表示）。CAS指令执行时，当且仅当内存地址V等于预期值A时，处理器用新值B更新V的值，否知它就不执行更行，但是无论是否更行了V的值，都会返回V的值，并且这三个操作是原子性的，比锁的切换是要快很多的，并且如果则线程更新失败,失败的线程不会被挂起,通过自旋会重新准备再次尝试更新。
* syncthronized属于悲观锁方式,是阻塞式同步的,采用独占的方式,每次获取和释放锁频繁,上下文资源切换较大,存在短暂调度延迟。在Synchronized优化以前，synchronized的性能是比ReenTrantLock差很多的，但是自从Synchronized引入了偏向锁，轻量级锁（自旋锁）,重量级锁后，两者的性能就差不多了，在两种方法都可用的情况下，官方甚至建议使用synchronized，其实synchronized的优化我感觉就借鉴了ReenTrantLock中的CAS技术。都是试图在用户态就把加锁问题解决，避免进入内核态的线程阻塞

**选择：**如果需要reenterlocker的高级特性，则优先选择reentrantlock。

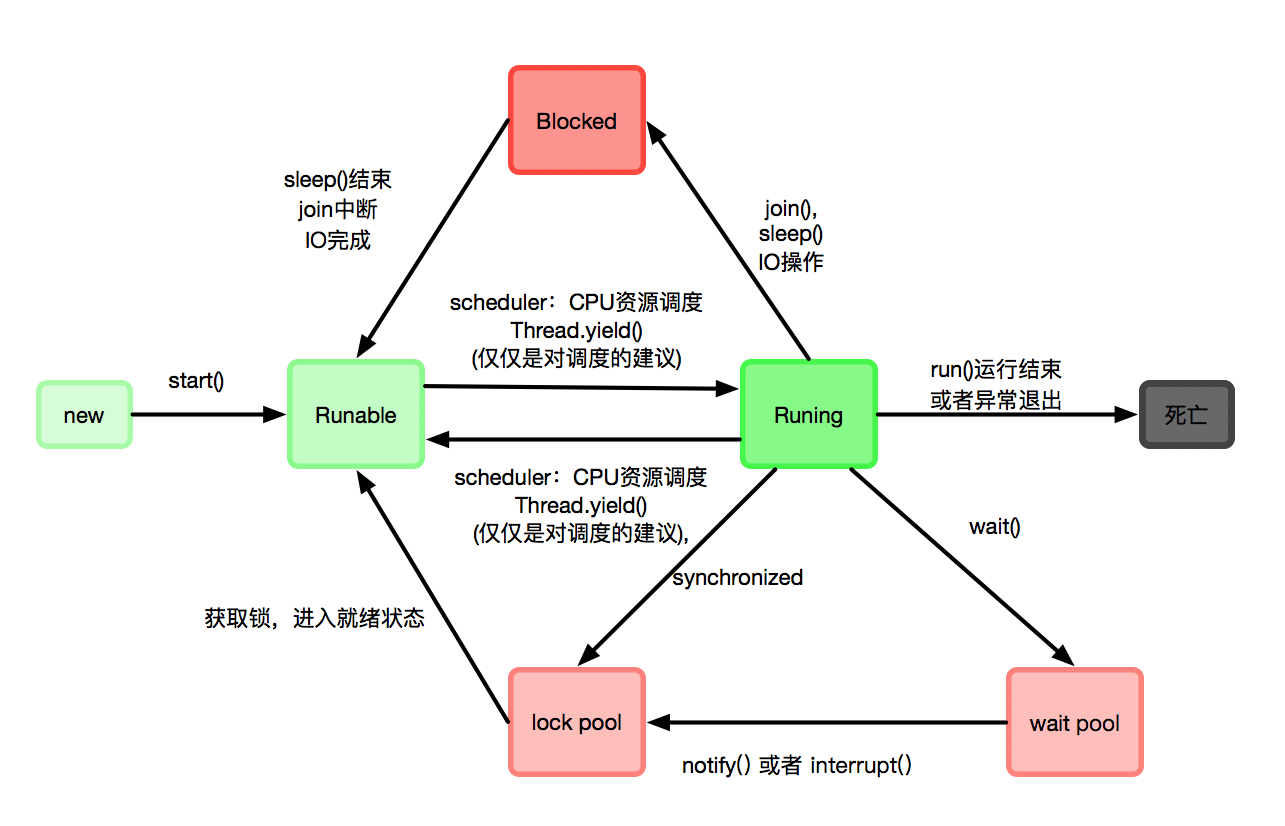
final的可见性

被final修饰的字段在构造器中一旦初始化完成，并且构造器没有把this的引用传递出去，那在其他线程中就能看到final的值，this的引用逃逸是一件很危险的事情，其他线程有可能通过这个引用访问到“初始化了一半”的对象。

This的引用逃逸了解吗？

线程的生命周期

线程的生命周期：一个线程从创建到消亡的过程。（包括同步时的状态变化）



**1.创建状态（new）**：

　　当用new操作符创建一个新的线程对象时，该线程处于创建状态。

　　处于创建状态的线程只是一个空的线程对象，系统不为它分配资源。

**2.可运行状态（就绪状态Runable）：**

执行线程的start()方法将为线程分配必须的系统资源，安排其运行，并调用线程体——run()方法，这样就使得该线程处于可运行状态（Runnable）。

　　这一状态并不是运行中状态（Running），因为线程也许实际上并未真正运行

3. **运行状态（Running）**：就绪状态的线程获取了CPU，执行程序代码

1. **阻塞状态（Blocked）**：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：

（一）、**等待阻塞**：运行的线程执行wait()方法，JVM会把该线程放入等待池中，待调用notify()/notifyAll()唤醒指定的线程或者所有线程，接着会进入锁池，竞争锁，再次获得对象锁才会进入运行状态

（二）、**同步阻塞**：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池中。

（三）、**其他阻塞**：运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

1. **死亡状态（Dead）**：线程执行完了或者因异常退出了run()方法，该线程结束生命周期

speep（）和 wait()的区别

|  |  |
| --- | --- |
| Sleep | Wait |
| 只是让出了CPU，不会释放同步锁，也不会释放同步资源 | 当前线程会放弃同步锁，进入等待队列，等待锁对象调用notify/notifyall唤醒指定的先，或者所有线程 |
| Sleep是线程的方法，可以在任何地方使用 | Wait是Object的方法，只能在同步方法或同步块中使用。永远在条件循环里调用wait，因为有可能错误使用notify或notifyall，导致被锁住的线程被唤醒，则会放在while里面，是防止出于waiting的对象被别的原因调用了唤醒方法，但是while里面的条件并没有满足（也可能当时满足了，但是由于别的线程操作后，又不满足了），就需要再次调用wait将其挂起。 |

Wait和notify知识点：

1.调用sleep（）的时候并没有释放锁，调用yiel也属于这种情况，理解这一点很重要，另一方面在方法里遇到对wait（）的调用，线程的执行被挂起，但对象上的锁被释放。   
2.wait和notify都是Object类的函数，而不属于Thread的一部分。但是只能在同步控制方法或者同步控制块里调用wait（），notify（），notifyAll（），这非常重要。如果在非同步控制方法里调用这些方法，程序能通过编译，但是运行的时候，将得到IllegalMonitorStateException异常，并伴随着一些含糊的信息，比如“当前线程不是拥有者”。消息的意思是，调用wait（），notify（），notifyAll（）的任务在调用这些方法前必须拥有（获取）对象的锁。并且notify（）和notifyAll（）方法将唤醒“正在等待的任务”，这是否意味着在程序中任何地方，任务处于wait（）状态的任务都将被任何对notifyAll（）的调用唤醒呢？事实上，当notifyAll（）因某个特定锁而被调用时，只有等待这个锁的任务才会被唤醒。   
3.第二点提到了必须在同步控制方法或者同步控制块里调用wait（），notify（），notifyAll（），那到底哪个对象应该被synchronized呢？答案是，那个 你希望上锁的对象（调用wait的对象）就应该被synchronized。   
4.永远在条件循环里调用wait，而不是在if语句里，这非常重要。在你知道wait应该永远在被synchronized下调用和在那个多线程共享的锁对象上调用，下一个一定要记住的问题就是，你应该永远在 while循环，而不是if语句中调用wait。原因是：举个例子：在我们的例子里，即“如果缓冲区队列是满的话，那么生产者线程应该等待”，你可能直觉就会写一个if语句。但if语句存在一些微妙的小问题。如果在条件没被满足的情况下，有可能错误的调用了notify或者notifyAll方法，那么你的线程你也有可能就被错误地唤醒了，如果使用if的话，那么就有可能出现在缓冲区为满的时候生产者继续生成数据或者在缓冲区为空的时候消费者 开始消耗数据，如果使用while，会再次检查条件，再次对锁对象进行wait调用，即再次等待。所以如果你不在线程被唤 醒后再次使用while循环检查唤醒条件是否被满足，你的程序就有可能会出错——例如在缓冲区为满的时候生产者继续生成数据，或者缓冲区为空的时候消费者开始消耗数据。所以记住，永远在while循环而不是if语句中使用wait！   
5.lock condition的await signal方法和 synchronize 锁对象 wait notify的一样的用法，singalAll比notifyAll要更加的安全。

wait和notify函数的规格代码模板

1. *// The standard idiom for calling the wait method in Java*
2. synchronized (sharedObject) {
3. while (condition) {
4. sharedObject.wait();
5. *// (Releases lock, and reacquires on wakeup)*
6. }
7. *// do action based upon condition e.g. take or put into queue*
8. }
9. synchronized (sharedObject) {
10. sharedObject.notify;
11. }

面题，可以手写用wait和notify解决生产者消费者问题

线程池的调优策略？

为什么[ConcurrentHashMap 1.8为什么要使用CAS+Synchronized取代Segment+ReentrantLock](https://www.cnblogs.com/yangfeiORfeiyang/p/9694383.html)

<https://juejin.im/post/5bc87409f265da0ad701da35>

一个线程接着一个线程执行 有什么方法？

<https://blog.csdn.net/qq_35571554/article/details/82834486>

public static void main(String[] args) throws InterruptedException {  
  
 final Thread t1 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 add();  
 }  
 });  
 final Thread t2 = new Thread(new Runnable() {  
  
 @Override  
 public void run() {  
 try {  
 //引用t1线程，等待t1线程执行完  
 t1.start();  
 t1.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 add();  
 }  
 });  
 Thread t3 = new Thread(new Runnable() {  
  
 @Override  
 public void run() {  
 try {  
 //引用t2线程，等待t2线程执行完,一定要在此处t2.start()  
 //在主线程中调用t2.start()执行顺序不确定，因为若t2.join()先于t2.start()运行  
 //导致结果不确定  
 t2.start();  
 t2.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 add();  
 }  
 });  
 t3.start();  
// t2.start();  
// t1.start();  
 }

使用thread.join方法， 首先join() 是一个synchronized方法， 里面调用了wait()，

当t1调用join方法时，相当于在外部线程中写了一个同步同步代码块，

t1.start()

sychronized(t1) {

while(true)

t1.wait() // 当前线程等待

}

}

add()

当t1线程执行完之后，也就是执行了t1.notifyall(), 也就相当于唤醒当前线程，然后当前线程继续执行add()。

不可重入锁实现

所谓不可重入锁，即若当前线程执行某个方法已经获取了该锁，那么在方法中尝试再次获取锁时，就会获取不到被阻塞。我们尝试设计一个不可重入锁：

public class Lock{

private boolean isLocked = false;

public synchronized void lock() throws InterruptedException{

while(isLocked){

wait();

}

isLocked = true;

}

public synchronized void unlock(){

isLocked = false;

notify();

}