多线程八股文

Part 1 Java并发---理论基础

一 为什么需要多线程？

众所周知，CPU、内存、I/O 设备的速度是有极大差异的，为了合理利用 CPU 的高性能，平衡这三者的速度差异，计算机体系结构、操作系统、编译程序都做出了贡献，主要体现为:

* CPU 增加了缓存，以均衡与内存的速度差异；--- 导致 **可见性** 问题
* 操作系统增加了进程、线程，以分时复用 CPU，进而均衡 CPU 与 I/O 设备的速度差异；-- 导致 **原子性** 问题
* 编译程序优化指令执行次序，使得缓存能够得到更加合理地利用。- 导致 **有序性** 问题

🡪含义：

* 可见性：一个线程对共享变量的修改，另外一个线程能够立刻看到。
* 原子性：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。
* 有序性：即程序执行的顺序按照代码的先后顺序执行。

🡪小结：

可见性---CPU缓存引起

原子性---时分复用引起

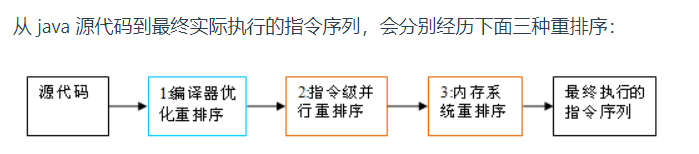
有序性---**重排序**引起



(了解)解析:重排序:

在执行程序时为了提高性能，编译器和处理器常常会对指令做重排序。重排序分三种类型：

* **编译器优化的重排序**。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。
* **指令级并行的重排序**。现代处理器采用了指令级并行技术（Instruction-Level Parallelism， ILP）来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。
* **内存系统的重排序**。由于处理器使用缓存和读 / 写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。



上述的 1 属于编译器重排序，2 和 3 属于处理器重排序。

这些重排序都可能会导致多线程程序出现内存可见性问题。对于编译器，JMM 的编译器重排序规则会禁止特定类型的编译器重排序（不是所有的编译器重排序都要禁止）。对于处理器重排序，JMM 的处理器重排序规则会要求 java 编译器在生成指令序列时，插入特定类型的内存屏障（memory barriers，intel 称之为 memory fence）指令，通过内存屏障指令来禁止特定类型的处理器重排序（不是所有的处理器重排序都要禁止）。

二 JAVA是怎么解决并发问题的: JMM(Java内存模型)

理解的第一个维度：核心知识点

JMM本质上可以理解为，**Java 内存模型规范了 JVM 如何提供按需禁用缓存和编译优化的方法。**具体来说，这些方法包括：

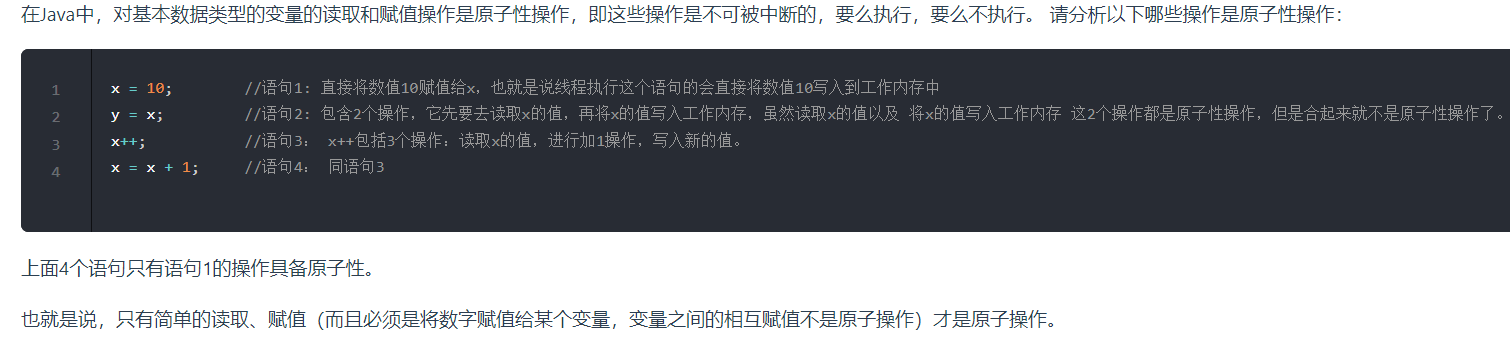
* volatile、synchronized 和 final 三个关键字
* Happens-Before 规则

理解的第二个维度: 可见性,有序性,原子性

1. **原子性**---Java内存模型（JMM）, synchronized, Lock

在Java中，对基本数据类型的变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行，要么不执行。

例子:



所以, Java内存模型只保证了基本读取和赋值是原子性操作，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。

1. 可见性--- volatile 关键字, (synchronized, Lock)

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。

另外，**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。

1. 有序性---Happens-Before (volatile, synchronized, Lock)

在Java里面，可以通过volatile关键字来保证**一定的“有序性”**（具体原理在下一节讲述）。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。当然**JMM是通过Happens-Before 规则来保证有序性的**。

Happens-Before规则: JVM 还规定了**先行发生原则**，让一个操作无需控制就能先于另一个操作完成。

1. 单一线程原则---在一个线程内，在程序前面的操作先行发生于后面的操作。
2. 管程锁定规则---一个 unlock 操作先行发生于后面对同一个锁的 lock 操作。
3. 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变量的读操作。
4. 线程启动规则--- Thread 对象的 start() 方法调用先行发生于此线程的每一个动作。
5. 线程加入规则--- Thread 对象的结束先行发生于 join() 方法返回。(join方法相当于线程按顺序执行)
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，可以通过 interrupted() 方法检测到是否有中断发生。
7. 对象终结规则---一个对象的初始化完成(构造函数执行结束)先行发生于它的 finalize() 方法的开始。
8. 传递性---如果操作 A 先行发生于操作 B，操作 B 先行发生于操作 C，那么操作 A 先行发生于操作 C。

======================================================================================================================

三 关键字:volatile, synchronized 和final

Synchronized

（一）Synchronized的使用:

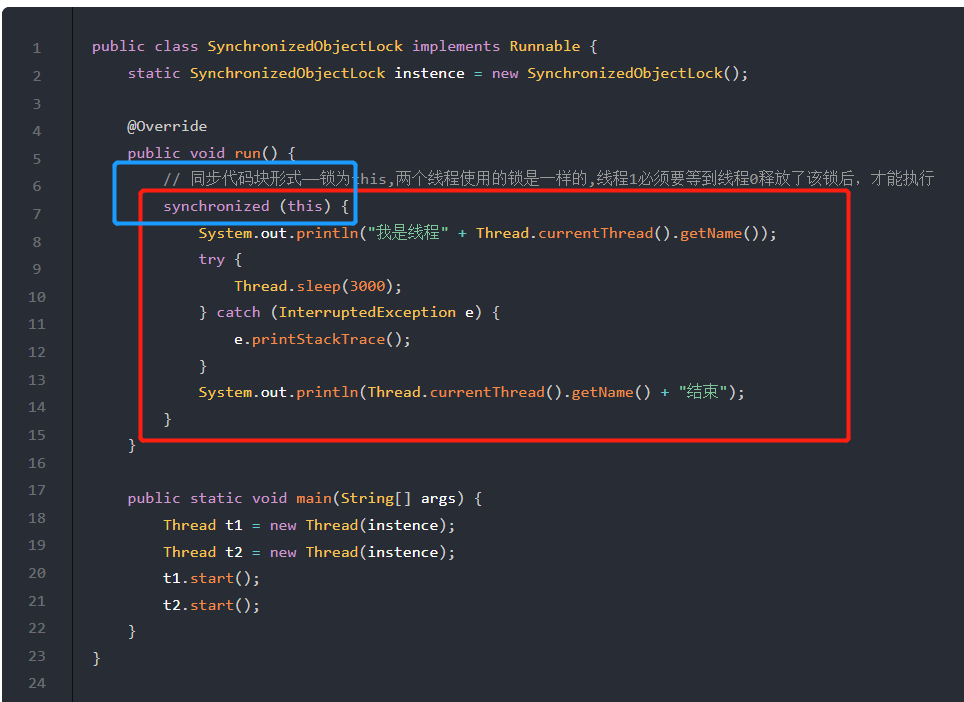
在应用Sychronized关键字时需要把握如下注意点：

* 一把锁只能同时被一个线程获取，没有获得锁的线程只能等待；
* 每个实例都对应有自己的一把锁(this),不同实例之间互不影响；例外：锁对象是\*.class以及synchronized修饰的是static方法的时候，所有对象公用同一把锁
* synchronized修饰的方法，无论方法正常执行完毕还是抛出异常，都会释放锁

1. 对象锁:

包括**方法锁**(默认锁对象为this,当前实例对象)和**同步代码块锁**(自己指定锁对象)

(1)代码块形式: 手动指定锁定对象，也可是是this,也可以是自定义的锁





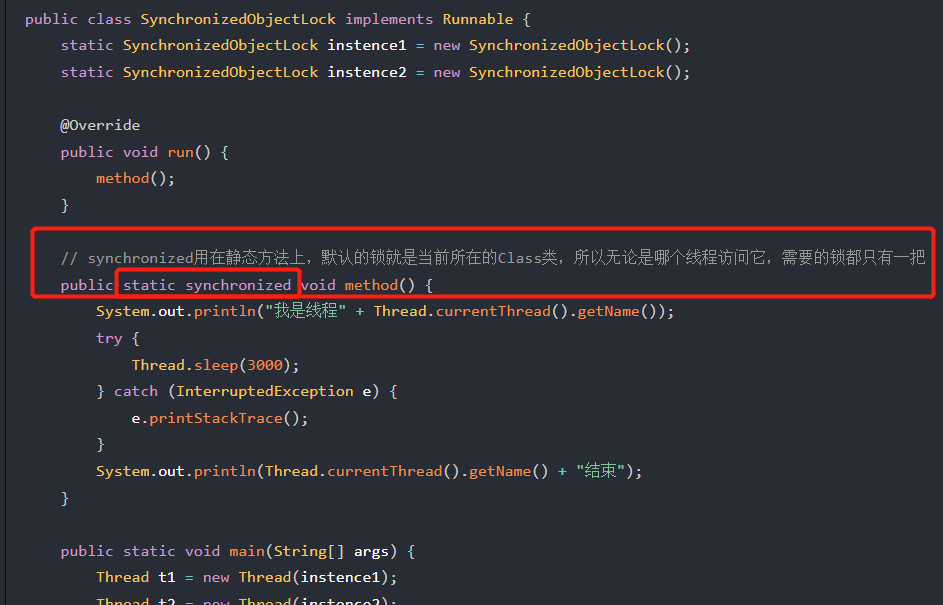
(2)方法锁形式: synchronized修饰普通方法，锁对象默认为this



1. 类锁

指synchronize修饰静态的方法或指定锁对象为Class对象

1. synchronized修饰静态方法



Synchronized用在普通方法上,默认的锁是this,当前实例

而synchronized用在静态方法上,默认的锁是当前所在的Class类,所以无论是哪个线程访问它,需要的锁都只有一把

1. synchronized指定锁对象为Class对象



（二）Synchronized原理分析:

1. 加锁和释放锁的原理:

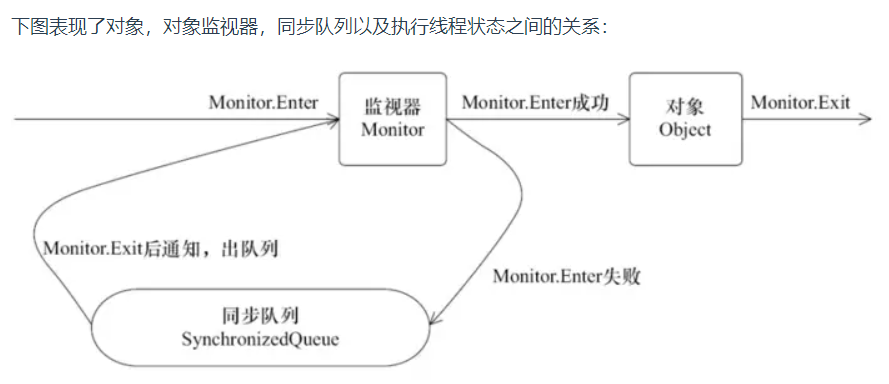
Synchronized内置锁是一种对象锁，作用粒度是对象，具体实现是通过内部对象**Monitor（监视器锁、管程对象）**来实现的，而Monitor的实现依赖底层操作系统的**Mutex Lock（互斥锁）**实现的（所以会从用户态切换到内核态）。互斥锁是重量级锁，性能较低。

**任何一个对象都有一个Monitor与之关联**，当对象的Monitor被持有后，该对象处于被锁定状态。**Monitor对象存在于每个java对象的对象头mark word中**（存储的指针的指向），Synchronized锁就是通过这种方式来获取锁的，也是为什么java中任意对象可以作为锁的原因，同时notify、notifyAll、wait方法会使用到Monitor锁对象，所以必须要在同步代码块中使用。

**Monitorenter**和**Monitorexit**指令，会让对象在执行，使 其锁计数器加1或者减1。**每一个对象在同一时间只与一个monitor(锁)相关联，而一个monitor在同一时间只能被一个线程获得**，一个对象在尝试获得与这个对象相关联的Monitor锁的所有权的时候，monitorenter指令会发生如下3中情况之一：

* monitor计数器为0，意味着目前还没有被获得，那这个线程就会立刻获得然后把锁计数器+1，一旦+1，别的线程再想获取，就需要等待
* 如果这个monitor已经拿到了这个锁的所有权，又重入了这把锁，那锁计数器就会累加，变成2，并且随着重入的次数，会一直累加
* 这把锁已经被别的线程获取了，等待锁释放

monitorexit指令：释放对于monitor的所有权，释放过程很简单，就是将monitor的计数器减1，如果减完以后，计数器不是0，则代表刚才是重入进来的，当前线程还继续持有这把锁的所有权，如果计数器变成0，则代表当前线程不再拥有该monitor的所有权，即释放锁。



该图可以看出，任意线程对Object的访问，首先要获得Object的监视器，如果获取失败，该线程就进入同步状态，线程状态变为BLOCKED，当Object的监视器占有者释放后，在同步队列中得线程就会有机会重新获取该监视器

1. 可重入原理：加锁次数计数器

上面的demo中在执行完同步代码块之后紧接着再会去执行一个静态同步方法，而这个方法锁的对象依然就这个类对象，那么这个正在执行的线程还需要获取该锁吗? 答案是不必的，从上图中就可以看出来，执行静态同步方法的时候就只有一条monitorexit指令，并没有monitorenter获取锁的指令。这就是锁的重入性，**即在同一锁程中，线程不需要再次获取同一把锁。**

**Synchronized先天具有重入性。每个对象拥有一个计数器，当线程获取该对象锁后，计数器就会加一，释放锁后就会将计数器减一。**

1. 保证可见性的原理：内存模型和happens-before规则

Synchronized的happens-before规则，即监视器锁规则：对同一个监视器的解锁，happens-before于对该监视器的加锁。

（三）JVM中锁的优化

简单来说在JVM中monitorenter和monitorexit字节码依赖于底层的操作系统的**Mutex Lock**来实现的，但是**由于使用Mutex Lock需要将当前线程挂起并从用户态切换到内核态来执行**，这种切换的代价是非常昂贵的；然而在现实中的大部分情况下，**同步方法是运行在单线程环境(无锁竞争环境)如果每次都调用Mutex Lock那么将严重的影响程序的性能。**不过在**jdk1.6**中对锁的实现引入了大量的优化，如**锁粗化(Lock Coarsening)**、**锁消除(Lock Elimination)**、**轻量级锁(Lightweight Locking)**、**偏向锁(Biased Locking)**、**适应性自旋(Adaptive Spinning)**等技术来减少锁操作的开销。

* **锁粗化(Lock Coarsening)**：也就是减少不必要的紧连在一起的unlock，lock操作，将多个连续的锁扩展成一个范围更大的锁。
* **锁消除(Lock Elimination)**：通过运行时JIT编译器的逃逸分析来消除一些没有在当前同步块以外被其他线程共享的数据的锁保护，通过逃逸分析也可以在线程本的Stack上进行对象空间的分配(同时还可以减少Heap上的垃圾收集开销)。
* **偏向锁(Biased Locking)**：是为了在无锁竞争的情况下避免在锁获取过程中执行不必要的CAS原子指令，因为CAS原子指令虽然相对于重量级锁来说开销比较小但还是存在非常可观的本的延迟。
* **轻量级锁(Lightweight Locking)**：这种锁实现的背后基于这样一种假设，即在真实的情况下我们程序中的大部分同步代码一般都处于无锁竞争状态(即单线程执行环境)，在无锁竞争的情况下完全可以避免调用操作系统层面的重量级互斥锁，取而代之的是在monitorenter和monitorexit中只需要依靠一条CAS原子指令就可以完成锁的获取及释放。当存在锁竞争的情况下，执行CAS指令失败的线程将调用操作系统互斥锁进入到阻塞状态，当锁被释放的时候被唤醒。
* **适应性自旋(Adaptive Spinning)**：当线程在获取轻量级锁的过程中执行CAS操作失败时，在进入与monitor相关联的操作系统重量级锁(mutex semaphore)前会进入忙等待(Spinning)然后再次尝试，当尝试一定的次数后如果仍然没有成功则调用与该monitor关联的semaphore(即互斥锁)进入到阻塞状态。

1. 锁的类型：

在Java SE 1.6里Synchronied同步锁，一共有四种状态：**无锁**、**偏向锁**、**轻量级锁**、**重量级锁**，它会随着竞争情况逐渐升级。**锁可以升级但是不可以降级**，目的是为了提供获取锁和释放锁的效率。



1. 自旋锁与自适应自旋锁

自旋锁是什么？？

---> 在没有加入锁优化时，Synchronized是一个非常“胖大”的家伙。在多线程竞争锁时，当一个线程获取锁时，它会阻塞所有正在竞争的线程，这样对性能带来了极大的影响。**在挂起线程和恢复线程的操作都需要转入内核态中完成**，这些操作对系统的并发性能带来很大的压力。在很多情况下，共享数据的锁定状态只会持续很短的一段时间，为了这段时间去挂起和回复阻塞线程并不值得。在如今多处理器环境下，完全可以让另一个没有获取到锁的线程在门外等待一会(**自旋**)，但不放弃CPU的执行时间。等待持有锁的线程是否很快就会释放锁。**为了让线程等待，我们只需要让线程执行一个忙循环(自旋)，**这便是自旋锁由来的原因。

​自旋锁早在**JDK1.4** 中就引入了，只是当时**默认是关闭的**。在**JDK 1.6**后**默认为开启状态**。自旋锁本质上与阻塞并不相同，自旋等待的时间必须要有一定的限度，如果自选超过了限定的次数仍然没有成功获取到锁，就应该使用**传统的方式去挂起线程**了，在JDK定义中，**自旋锁默认的自旋次数为10次，**用户可以使用参数-XX:PreBlockSpin来更改。

可是现在又出现了一个问题：如果线程锁在线程自旋刚结束就释放掉了锁，那么是不是有点得不偿失。所以这时候我们需要更加聪明的锁来实现更加灵活的自旋。来提高并发的性能。(这里则需要自适应自旋锁！)

自适应自旋锁是什么？？？

在**JDK 1.6**中引入了自适应自旋锁。这就意味着**自旋的时间不再固定**了，而是由**前一次在同一个锁上的自旋时间**及**锁的拥有者的状态**来决定的。如果在同一个锁对象上，自旋等待刚刚成功获取过锁，并且持有锁的线程正在运行中，那么JVM会认为该锁自旋获取到锁的可能性很大，会自动增加等待时间。相反，如果对于某个锁，自旋很少成功获取锁。那再以后要获取这个锁时将可能省略掉自旋过程，以避免浪费处理器资源。有了自适应自旋，JVM对程序的锁的状态预测会越来越准确，JVM也会越来越聪明。

1. 锁消除

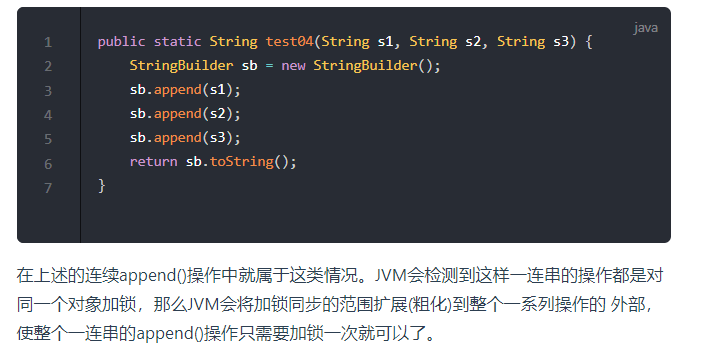
锁消除是指虚拟机即时编译器在运行时，对一些代码上要求同步，但是**被检测到不可能存在共享数据竞争的锁进行消除**。锁消除的主要判定依据来源于逃逸分析的数据支持。意思就是：JVM会判断在一段程序中的同步明显不会逃逸出去从而被其他线程访问到，那JVM就把它们当作**栈上数据**对待，**认为这些数据时线程独有的，不需要加同步**。此时就会进行锁消除。

例：字符串拼接时：String---> 线程安全的StringBuffer.append()---->检测到不可逃逸，锁消除，改用 线程不安全的StringBuilder.append()

1. 锁的粗化

我们都知道在加同步锁时，**尽可能的将同步块的作用范围限制到尽量小的范围**(只在共享数据的实际作用域中才进行同步，这样是为了**使得需要同步的操作数量尽可能变小**。**在存在锁同步竞争中，也可以使得等待锁的线程尽早的拿到锁**)。

大部分上述情况是完美正确的，但是如果存在连串的一系列操作都对同一个对象反复加锁和解锁，甚至加锁操作时出现在循环体中的，那即使没有线程竞争，频繁的进行互斥同步操作也会导致不必要的性能操作。



1. 轻量级锁

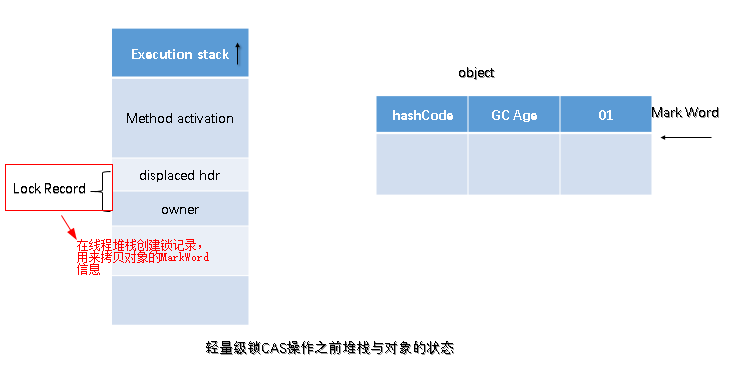
在**JDK 1.6**之后引入的轻量级锁，需要注意的是轻量级锁并**不是替代重量级锁**的，而是**对在大多数情况下同步块并不会有竞争出现提出的一种优化**。它**可以减少重量级锁对线程的阻塞带来的线程开销**。从而提高并发性能。

​如果要理解轻量级锁，那么必须先要了解HotSpot虚拟机中对象头的内存布局。

在对象头中(Object Header)存在两部分。**第一部分用于存储对象自身的运行时数据，HashCode、GC Age、锁标记位、是否为偏向锁等。一般为32位或者64位**(视操作系统位数定)。官方称之为**Mark Word**，它**是实现轻量级锁和偏向锁的关键**。 **另外一部分存储的是指向方法区对象类型数据的指针(Klass Point)**，如果对象是数组的话，还会有一个额外的部分用于存储数据的长度。

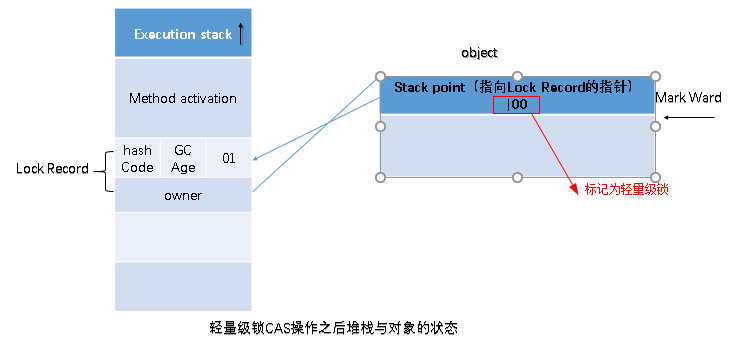
（1）轻量级锁加锁

在线程执行同步块之前，JVM会先在当前线程的栈帧中创建一个名为锁记录(Lock Record)的空间，用于存储锁对象目前的Mark Word的拷贝(JVM会将对象头中的Mark Word拷贝到锁记录中，官方称为Displaced Mark Ward)这个时候线程堆栈与对象头的状态如图：



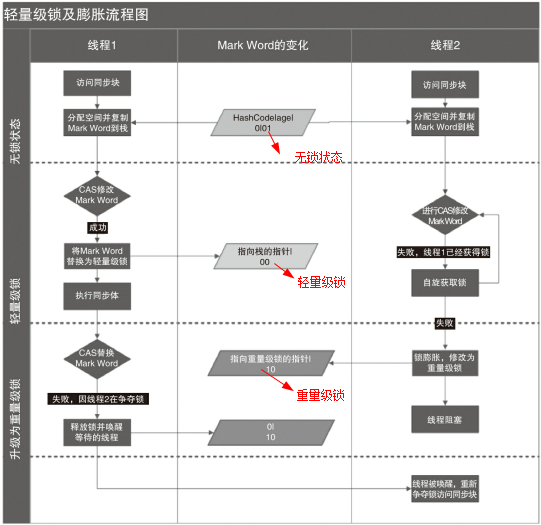
如上图所示：如果当前对象没有被锁定，那么锁标志位位01状态，JVM在执行当前线程时，首先会在当前线程栈帧中创建锁记录Lock Record的空间用于存储锁对象目前的Mark Word的拷贝。

然后，虚拟机使用CAS操作将标记字段Mark Word拷贝到锁记录中，并且将Mark Word更新为指向Lock Record的指针。如果更新成功了，那么这个线程就有用了该对象的锁，并且对象Mark Word的锁标志位更新为(Mark Word中最后的2bit)00，即表示此对象处于轻量级锁定状态，如图：



如果这个更新操作失败，JVM会检查当前的Mark Word中是否存在指向当前线程的栈帧的指针，如果有，说明该锁已经被获取，可以直接调用。如果没有，则说明该锁被其他线程抢占了，如果有两条以上的线程竞争同一个锁，那轻量级锁就不再有效，直接膨胀位重量级锁，没有获得锁的线程会被阻塞。此时，锁的标志位为10.Mark Word中存储的时指向重量级锁的指针。

​ 轻量级解锁时，会使用原子的CAS操作将Displaced Mark Word替换回到对象头中，如果成功，则表示没有发生竞争关系。如果失败，表示当前锁存在竞争关系。锁就会膨胀成重量级锁。两个线程同时争夺锁，导致锁膨胀的流程图如下：



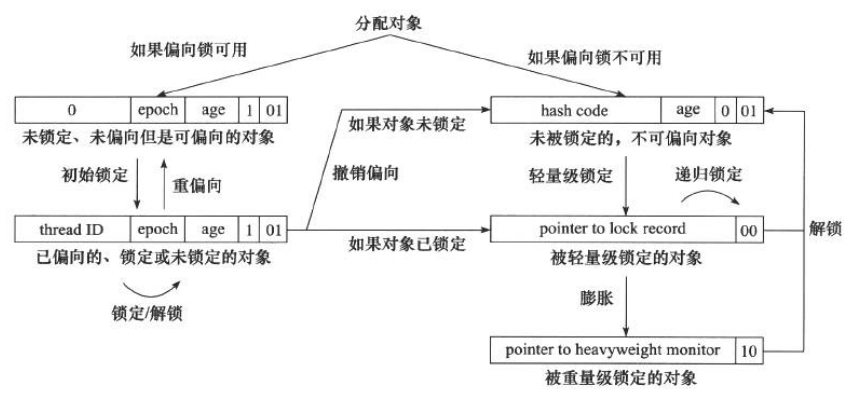
1. 偏向锁---对轻量级锁的优化

背景： 在大多实际环境下，锁不仅不存在多线程竞争，而且总是由同一个线程多次获取，那么在同一个线程反复获取所释放锁中，其中并还没有锁的竞争，那么这样看上去，多次的获取锁和释放锁带来了很多不必要的性能开销和上下文切换。

解决方法：偏向锁

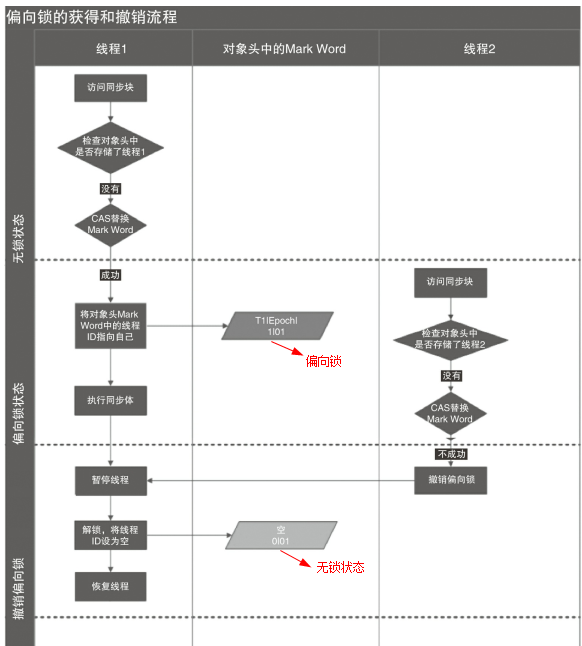
具体原理：

当一个线程访问同步块并获取锁时，会在对象头和栈帧中的锁记录里存储锁偏向的线程ID，以后该线程在进入和推出同步块时不需要进行CAS操作来加锁和解锁。只需要简单的测试一下对象头的Mark Word里是否存储着指向当前线程的偏向锁。如果成功，表示线程已经获取到了锁。

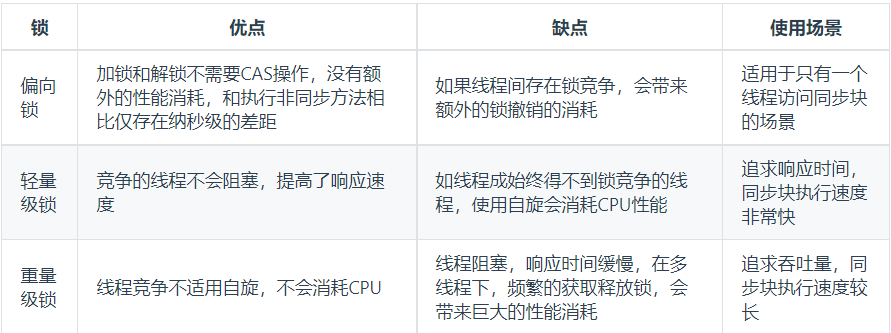


（2）偏向锁的撤销

偏向锁使用了一种**等待竞争出现才会释放锁**的机制。所以当其他线程尝试获取偏向锁时，持有偏向锁的线程才会释放锁。但是偏向锁的撤销需要等到**全局安全点(就是当前线程没有正在执行的字节码)**。它会**首先暂停拥有偏向锁的线程**，**然后检查持有偏向锁的线程是否活着**。如果线程不处于活动状态，直接将对象头设置为无锁状态。如果线程活着，JVM会遍历栈帧中的锁记录，栈帧中的锁记录和对象头**要么偏向于其他线程**，**要么恢复到无锁状态，或者标记对象不适合作为偏向锁**。



1. 锁的优缺点对比



1. Synchronized与Lock对比

|  |  |  |
| --- | --- | --- |
|  | Synchronized | Lock |
| 效率 | 相比Lock效率低。   1. Synchronized锁的释放情况少，只有代码执行完毕或者异常结束才会释放锁； 2. 试图获取锁的时候**不能设定超时**，**不能中断**一个正在使用锁的线程 | 相比Synchronized效率更高    Lock可以设置超时和中断 |
| 灵活性 | 不如Lock灵活   1. 加锁和释放的时机单一 2. 每个锁仅有一个单一的条件(是否获取锁) | 比Synchronized灵活  如，读写锁 |
| 知道是否成功获得锁 | 不能 | Lock可以得到状态 |

1. Lock类要点

4个常用方法：

* lock(): 加锁
* unlock(): 解锁
* tryLock(): 尝试获取锁，返回一个boolean值
* tryLock(long,TimeUtil): 尝试获取锁，可以设置超时

1. synchronized其他知识点：
2. synchronized是通过软件(JVM)实现的，简单易用，即使在JDK5之后有了Lock，仍然被广泛的使用。
3. 使用Synchronized有哪些要注意的？

* **锁对象不能为空**，因为锁的信息都保存在对象头里
* **作用域不宜过大**，影响程序执行的速度，控制范围过大，编写代码也容易出错
* **避免死锁**
* **在能选择的情况下，既不要用Lock也不要用synchronized关键字，用java.util.concurrent包中的各种各样的类**，如果不用该包下的类，在满足业务的情况下，可以使用synchronized关键，因为代码量少，避免出错

1. Synchronized是**非公平锁**

synchronized实际上是非公平的，新来的线程有可能立即获得监视器，而在等待区中等候已久的线程可能再次等待，这样有利于提高性能，但是也可能会导致饥饿现象。

===========================================================

Volatile关键字

1. Volatile的作用：

* 防重排序
* 实现可见性
* 保证原子性：单词读写

（1）防重排序：

例子：并发环境下单例实现方式，可以采用双重检查加锁（DCL）的方式实现



现在我们分析一下为什么要在**变量singleton之间加上volatile关键字**（singleton静态变量存储预定义的对象，但未存入对象）。要理解这个问题，先要了解对象的构造过程，实例化一个对象其实可以分为三个步骤：

* 分配内存空间。
* 初始化对象。
* 将内存空间的地址赋值给对应的引用。

但是由于操作系统可以对指令进行重排序，所以上面的过程也可能会变成如下过程：

* 分配内存空间。
* 将内存空间的地址赋值给对应的引用。
* 初始化对象

如果是这个流程，多线程环境下就可能将一个未初始化的对象引用暴露出来，从而导致不可预料的结果。因此，为了防止这个过程的重排序，我们需要将变量设置为volatile类型的变量。

（2）实现可见性

**可见性问题主要指一个线程修改了共享变量值，而另一个线程却看不到。**

引起可见性问题的主要原因是每个线程拥有自己的一个高速缓存区——线程工作内存。volatile关键字能有效的解决这个问题。

（3）保证原子性：单次读写

**volatile不能保证完全的原子性，只能保证单次的读/写操作具有原子性。**

问题1： i++为什么不能保证原子性?（i++本质上是读、写两次操作）

问题2： 共享的long和double变量的为什么要用volatile?

因为long和double两种数据类型的操作可分为高32位和低32位两部分，因此普通的long或double类型读/写可能不是原子的。因此，鼓励大家将共享的long和double变量设置为volatile类型，这样能保证任何情况下对long和double的**单次读/写操作**都具有原子性。

1. Volatile的实现原理：
2. volatile可见性实现

**volatile 变量的内存可见性是基于内存屏障(Memory Barrier)实现**：

 内存屏障，又称内存栅栏，是一个 CPU 指令。

 在程序运行时，为了提高执行性能，编译器和处理器会对指令进行重排序，JMM 为了保证在不同的编译器和 CPU 上有相同的结果**，通过插入特定类型的内存屏障来禁止+ 特定类型的编译器重排序和处理器重排序**，**插入一条内存屏障会告诉编译器和 CPU：不管什么指令都不能和这条 Memory Barrier 指令重排序**。

扩展知识：lock 前缀的指令在多核处理器下会引发两件事情:

* 将当前处理器缓存行的数据写回到系统内存。
* 写回内存的操作会使在其他 CPU 里缓存了该内存地址的数据无效。

为了提高处理速度，处理器不直接和内存进行通信，而是先将系统内存的数据读到内部缓存(L1，L2 或其他)后再进行操作，但操作完不知道何时会写到内存。

如果对声明了 volatile 的变量进行写操作，JVM 就会向处理器发送一条 lock 前缀的指令，将这个变量所在缓存行的数据写回到系统内存。

为了保证各个处理器的缓存是一致的，实现了**缓存一致性协议(MESI)**，每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器对这个数据进行修改操作的时候，会重新从系统内存中把数据读到处理器缓存里。

所有多核处理器下还会完成：当处理器发现本地缓存失效后，就会从内存中重读该变量数据，即可以获取当前最新值。

volatile 变量通过这样的机制就使得每个线程都能获得该变量的最新值。

1. volatile有序性实现

* volatile 的 happens-before 关系： happens-before 规则中有一条是 volatile 变量规则：**对一个 volatile 域的写，happens-before 于任意后续对这个 volatile 域的读。**
* volatile禁止重排序

为了性能优化，JMM 在不改变正确语义的前提下，会允许编译器和处理器对指令序列进行重排序。JMM 提供了内存屏障阻止这种重排序。

--Java 编译器会在生成指令系列时在适当的位置会插入内存屏障指令来禁止特定类型的处理器重排序。

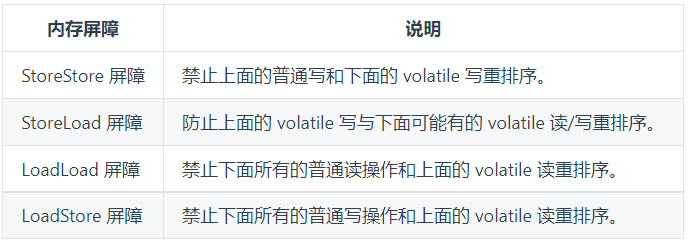
--JMM 会针对编译器制定 volatile 重排序规则表。



对于编译器来说，发现一个最优布置来最小化插入屏障的总数几乎是不可能的，为此，JMM 采取了保守的策略。

* 在每个 volatile 写操作的前面插入一个 StoreStore 屏障。
* 在每个 volatile 写操作的后面插入一个 StoreLoad 屏障。
* 在每个 volatile 读操作的后面插入一个 LoadLoad 屏障。
* 在每个 volatile 读操作的后面插入一个 LoadStore 屏障。

volatile 写是在前面和后面分别插入内存屏障，而 volatile 读操作是在后面插入两个内存屏障。



Volatile的应用场景：

使用 volatile 必须具备的条件

* 对变量的写操作不依赖于当前值。
* 该变量没有包含在具有其他变量的不变式中。
* 只有在状态真正独立于程序内其他内容时才能使用 volatile。

模式1：**状态标志**

也许实现 volatile 变量的规范使用仅仅是使用一个布尔状态标志，用于指示发生了一个重要的一次性事件，例如完成初始化或请求停机。

模式2：**一次性安全发布**(one-time safe publication)

模式3：**独立观察**(independent observation)---定期发布观察结果

模式4：**volatile bean 模式**

在 volatile bean 模式中，JavaBean 的所有数据成员都是 volatile 类型的，并且 getter 和 setter 方法必须非常普通 —— 除了获取或设置相应的属性外，不能包含任何逻辑。此外，对于对象引用的数据成员，引用的对象必须是有效不可变的。(这将禁止具有数组值的属性，因为当数组引用被声明为 volatile 时，只有引用而不是数组本身具有 volatile 语义)。对于任何 volatile 变量，不变式或约束都不能包含 JavaBean 属性。

模式5：**开销较低的读－写锁策略--**安全的计数器使用 synchronized 确保增量操作是原子的，并使用 volatile 保证当前结果的可见性。

模式6：**双重检查**---单例模式的一种实现方式，但很多人会忽略volatile关键字的作用

===========================================================final关键字

1. final关键字的基础用法：

* 修饰类---该类不能被继承
* 修饰方法---该方法不能被重写，但能被重载
* 修饰参数列表中的参数---无法在方法中更改参数引用所指向的对象
* 修饰变量---初始化之后，不能修改变量值

注意：

1. final类中的所有方法都隐式为final，因为无法覆盖他们，所以在final类中给任何方法添加final关键字是没有任何意义的。
2. private 方法是隐式的final
3. final方法是可以被重载的
4. 对于final修饰参数的特性，主要用来向匿名内部类传递数据
5. Final修饰变量时可以分为：static final 和 blank final

对于static final，一个既是static又是final 的字段只占据一段不能改变的存储空间，它必须在定义的时候进行赋值，否则编译器将不予通过。

对于black final， Java允许生成空白final，也就是说被声明为final但又没有给出定值的字段,但是必须在该字段被使用之前被赋值，这给予我们两种选择：

* 在定义处进行赋值(这不叫空白final)
* 在构造器中进行赋值，保证了该值在被使用前赋值。

这增强了final的灵活性。

1. 所有的final修饰的字段都是编译期常量吗?---不是所有的final修饰的字段都是编译器常量，只是被修饰的值在被初始化后无法被更改。
2. final域重排序规则(多线程情况下)

case1：final域为基本类型

（1）写final域重排序规则：

写final域的重排序规则禁止对final域的写重排序到构造函数之外，这个规则的实现主要包含了两个方面：

* JMM禁止编译器把final域的写重排序到构造函数之外；
* 编译器会在final域写之后，构造函数return之前，插入一个storestore屏障。这个屏障可以禁止处理器把final域的写重排序到构造函数之外。

因此，**写final域的重排序规则可以确保：在对象引用为任意线程可见之前，对象的final域已经被正确初始化过了，而普通域就不具有这个保障**。

（2）读final域重排序规则

读final域重排序规则为：在一个线程中，初次读对象引用和初次读该对象包含的final域，JMM会禁止这两个操作的重排序。(注意，这个规则仅仅是针对处理器)，

处理器会在读final域操作的前面插入一个LoadLoad屏障。实际上，读对象的引用和读该对象的final域存在间接依赖性，一般处理器不会重排序这两个操作。但是有一些处理器会重排序，因此，这条禁止重排序规则就是针对这些处理器而设定的。

**读final域的重排序规则可以确保：在读一个对象的final域之前，一定会先读这个包含这个final域的对象的引用。**

Case2：final域为引用类型

1. 对final修饰的对象的成员域写操作

针对引用数据类型，final域写针对编译器和处理器重排序增加了这样的约束：**在构造函数内对一个final修饰的对象的成员域的写入，与随后在构造函数之外把这个被构造的对象的引用赋给一个引用变量，这两个操作是不能被重排序的。**

1. 对final修饰的对象的成员域读操作

关于final重排序的总结：

按照final修饰的数据类型分类：

* 基本数据类型:
* final域写：**禁止final域写与构造方法重排序，即禁止final域写重排序到构造方法之外**，从而保证该对象对所有线程可见时，该对象的final域全部已经初始化过。
* final域读：**禁止初次读对象的引用与读该对象包含的final域的重排序。**
* 引用数据类型：
* 额外增加约束：**禁止在构造函数对一个final修饰的对象的成员域的写入与随后将这个被构造的对象的引用赋值给引用变量 重排序**

1. Final的实现原理：

写final域会要求编译器在**final域写之后**，**构造函数返回前**插入一个**StoreStore屏障**。

读final域的重排序规则会要求编译器**在读final域的操作前**插入一个**LoadLoad屏障**。

1. 为什么final引用不能从构造函数中“溢出”

上面对final域写重排序规则可以确保我们在使用一个对象引用的时候该对象的final域已经在构造函数被初始化过了。但是这里其实是有一个前提条件的，也就是：在构造函数，不能让这个被构造的对象被其他线程可见，也就是说该对象引用不能在构造函数中“溢出”

1. 使用 final 的限制条件和局限性

当声明一个 final 成员时，必须在构造函数退出前设置它的值。

或者，将指向对象的成员声明为 final 只能将该引用设为不可变的，而非所指的对象。

如果一个对象将会在多个线程中访问并且你并没有将其成员声明为 final，则必须提供其他方式保证线程安全。" 其他方式 " 可以包括声明成员为 volatile，使用 synchronized 或者显式 Lock 控制所有该成员的访问。

======================================================================================================================

四、线程安全---不是一个非真即假的命题

线程安全含义：一个类在可以被多个线程安全调用时就是线程安全的。

线程安全不是一个非真即假的命题，可以将共享数据按照安全程度的强弱顺序分成以下五类:

* 不可变
* 绝对线程安全
* 相对线程安全
* 线程兼容
* 线程对立

1. 不可变

不可变(Immutable)的对象一定是线程安全的，不需要再采取任何的线程安全保障措施。

多线程环境下，应当尽量使对象成为不可变，来满足线程安全。

不可变的类型:

* final 关键字修饰的基本数据类型
* String
* 枚举类型
* Number 部分子类，如 Long 和 Double 等数**值包装类型**，BigInteger 和 BigDecimal 等**大数据类型**。**但同为 Number 的原子类 AtomicInteger 和 AtomicLong 则是可变的**。
* 对于集合类型，可以使用 Collections.unmodifiableXXX() 方法来获取一个不可变的集合。

1. 绝对线程安全---不管运行时环境如何，调用者都不需要任何额外的同步措施。
2. 相对线程安全

**相对线程安全需要保证对这个对象单独的操作是线程安全的**，在调用的时候不需要做额外的保障措施。**但是对于一些特定顺序的连续调用，就可能需要在调用端使用额外的同步手段来保证调用的正确性**。

在 Java 语言中，大部分的线程安全类都属于这种类型，例如 Vector、HashTable、Collections 的 synchronizedCollection() 方法包装的集合等。

1. 线程兼容

**线程兼容是指对象本身并不是线程安全的，但是可以通过在调用端正确地使用同步手段来保证对象在并发环境中可以安全地使用**，我们平常说一个类不是线程安全的，绝大多数时候指的是这一种情况。

Java API 中大部分的类都是属于线程兼容的，如与前面的 Vector 和 HashTable 相对应的集合类 ArrayList 和 HashMap 等。

1. 线程对立

线程对立是指无论调用端是否采取了同步措施，都无法在多线程环境中并发使用的代码。由于 Java 语言天生就具备多线程特性，线程对立这种排斥多线程的代码是很少出现的，而且**通常都是有害的，应当尽量避免。**

======================================================================================================================

五、线程安全的实现方法（3种）

* 互斥同步
* 非阻塞同步
* 无同步方案

1. 互斥同步：

互斥同步最主要的问题就是线程阻塞和唤醒所带来的性能问题，因此这种同步也称为**阻塞同步**。互斥同步属于一种**悲观并发策略**，总是认为只要不去做正确的同步措施，那就肯定会出现问题。无论共享数据是否真的会出现竞争，它都要进行加锁。主要是应用**synchronized** 和 **ReentrantLock**

1. 非阻塞同步
2. CAS

随着硬件指令集的发展，我们可以使用基于冲突检测的**乐观并发策略**: 先进行操作，如果没有其它线程争用共享数据，那操作就成功了，否则采取补偿措施(不断地重试，直到成功为止)。这种乐观的并发策略的许多实现都不需要将线程阻塞，因此这种同步操作称为非阻塞同步。

**乐观锁需要操作和冲突检测这两个步骤具备原子性**，这里就不能再使用互斥同步来保证了，只能靠硬件来完成。硬件支持的原子性操作最典型的是: **比较并交换(Compare-and-Swap，CAS)**。

CAS 指令需要有 3 个操作数，分别是内存地址 V、旧的预期值 A 和新值 B。当执行操作时，只有当 V 的值等于 A，才将 V 的值更新为 B。

1. AtomicInteger

J.U.C 包里面的整数原子类 AtomicInteger，其中的 compareAndSet() 和 getAndIncrement() 等方法都使用了 Unsafe 类的 CAS 操作。

1. ABA

如果一个变量初次读取的时候是 A 值，它的值被改成了 B，后来又被改回为 A，那 CAS 操作就会误认为它从来没有被改变过。

J.U.C 包提供了一个带有标记的原子引用类 AtomicStampedReference 来解决这个问题，它可以通过控制变量值的版本来保证 CAS 的正确性。大部分情况下 ABA 问题不会影响程序并发的正确性，如果需要解决 ABA 问题，改用传统的互斥同步可能会比原子类更高效。

1. 无同步方案

要保证线程安全，并不是一定就要进行同步。如果一个方法本来就不涉及共享数据，那它自然就无须任何同步措施去保证正确性。

1. 栈封闭

多个线程访问同一个方法的局部变量时，不会出现线程安全问题，因为局部变量存储在虚拟机栈中，属于线程私有的。

1. 线程本地存储

如果一段代码中所需要的数据必须与其他代码共享，那就看看这些共享数据的代码是否能保证在同一个线程中执行。如果能保证，我们就可以把共享数据的可见范围限制在同一个线程之内，这样，无须同步也能保证线程之间不出现数据争用的问题。

符合这种特点的应用并不少见，大部分使用消费队列的架构模式(如“生产者-消费者”模式)都会将产品的消费过程尽量在一个线程中消费完。其中最重要的一个应用实例就是**经典 Web 交互模型中的“一个请求对应一个服务器线程”(Thread-per-Request)的处理方式**，这种处理方式的广泛应用使得很多 Web 服务端应用都可以使用线程本地存储来解决线程安全问题。

可以使用 java.lang.ThreadLocal 类来实现线程本地存储功能。

ThreadLocal 从理论上讲并不是用来解决多线程并发问题的，因为根本不存在多线程竞争。

在一些场景 (尤其是使用线程池) 下，由于 ThreadLocal.ThreadLocalMap 的底层数据结构导致 ThreadLocal 有内存泄漏的情况，应该尽可能在每次使用 ThreadLocal 后手动调用 remove()，以避免出现 ThreadLocal 经典的内存泄漏甚至是造成自身业务混乱的风险。

1. 可重入代码

这种代码也叫做**纯代码**(Pure Code)，可以在代码执行的任何时刻中断它，转而去执行另外一段代码(包括递归调用它本身)，而在控制权返回后，原来的程序不会出现任何错误。

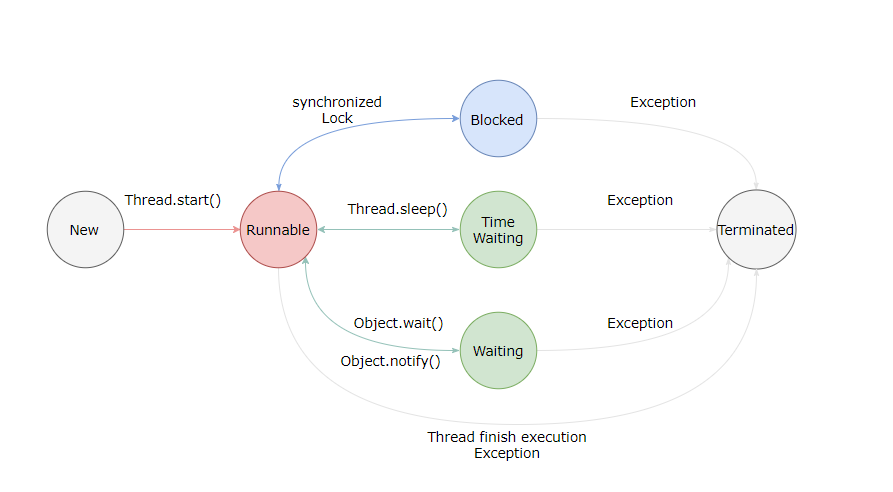
可重入代码有一些共同的特征，例如不依赖存储在堆上的数据和公用的系统资源、用到的状态量都由参数中传入、不调用非可重入的方法等。

======================================================================================================================

===========================================================

Part 2 java并发---线程基础

1. 线程状态转换（六种状态）



* 新建(New)

创建后尚未启动。

* 可运行(Runnable)

可能正在运行，也可能正在等待 CPU 时间片。

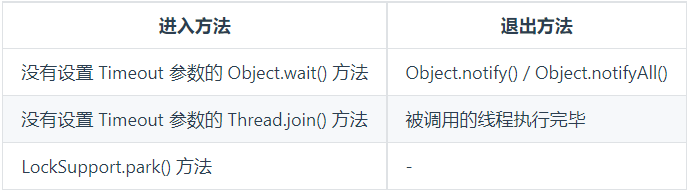
包含了操作系统线程状态中的 Running 和 Ready。

* 阻塞(Blocking)

等待获取一个排它锁，如果其线程释放了锁就会结束此状态。

* 无限等待(Waiting)

等待其它线程显式地唤醒，否则不会被分配 CPU 时间片。



* 计时等待(Timed Waiting)

无需等待其它线程显式地唤醒，在一定时间之后会被系统自动唤醒。

调用 Thread.sleep() 方法使线程进入限期等待状态时，常常用“**使一个线程睡眠**”进行描述。

调用 Object.wait() 方法使线程进入限期等待或者无限期等待时，常常用“**挂起一个线程**”进行描述。

睡眠和挂起是用来描述行为，而阻塞和等待用来描述状态。

阻塞和等待的区别在于，阻塞是被动的，它是在等待获取一个排它锁。而等待是主动的，通过调用 Thread.sleep() 和 Object.wait() 等方法进入。



* 死亡(Terminated)

可以是线程结束任务之后自己结束，或者产生了异常而结束。

===========================================================

1. 线程的使用方式：

有三种使用线程的方法（如果硬要说可以说四种，因为使用线程池也算一种）:

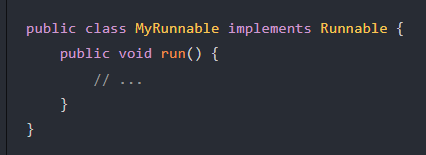
* 实现 Runnable 接口
* 实现 Callable 接口
* 继承Thread类
* 使用线程池

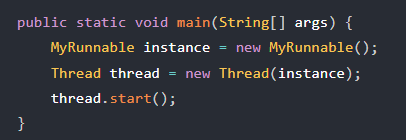
一定要注意：**实现 Runnable 和 Callable 接口的类只能当做一个可以在线程中运行的任务，不是真正意义上的线程**，因此**最后还需要通过 Thread 来调用**。可以说任务是通过线程驱动从而执行的。

（1）实现Runnable接口

需要实现 run() 方法。

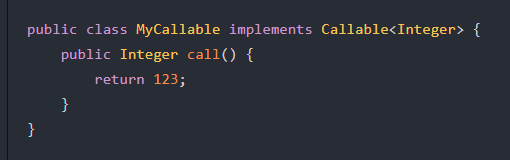
通过 Thread 调用 start() 方法来启动线程。

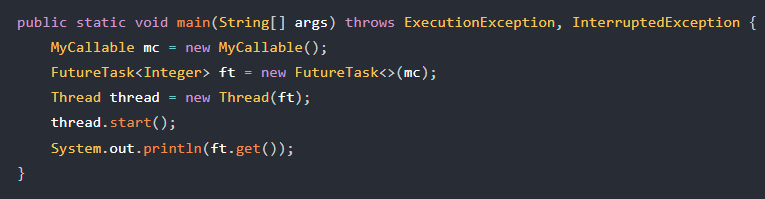




（2）实现Callable接口

与 Runnable 相比，Callable 可以有返回值，返回值通过 FutureTask 进行封装。

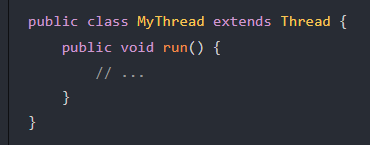


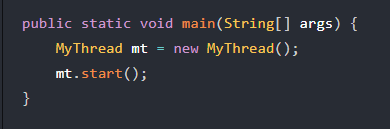


（3）继承Thread类

同样也是需要实现 run() 方法，因为 **Thread 类也实现了 Runnable 接口**。

当调用 start() 方法启动一个线程时，虚拟机会将该线程放入就绪队列中等待被调度，当一个线程被调度时会执行该线程的 run() 方法。





实现接口 VS 继承 Thread ？？？？？

实现接口会更好一些，因为：

* **Java 不支持多重继承，因此继承了 Thread 类就无法继承其它类，但是可以实现多个接口；**
* 类可能只要求可执行就行，继承整个 Thread 类开销过大。

===========================================================

1. 基础线程机制
2. **Executor**

**Executor 管理多个异步任务的执行，而无需程序员显式地管理线程的生命周期**。

这里的异步是指多个任务的执行互不干扰，不需要进行同步操作。

主要有三种 Executor:

* CachedThreadPool: 一个任务创建一个线程；
* FixedThreadPool: 所有任务只能使用固定大小的线程；
* SingleThreadExecutor: 相当于大小为 1 的 FixedThreadPool。

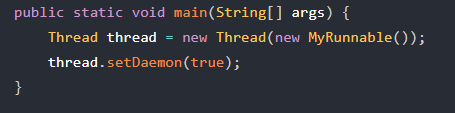
1. Daemon

守护线程是程序运行时在后台提供服务的线程，**不属于**程序中不可或缺的部分。

**当所有非守护线程结束时，程序也就终止，同时会杀死所有守护线程。**

main() 属于非守护线程。

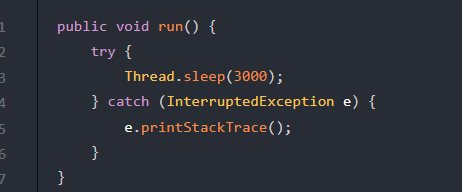
使用 setDaemon() 方法将一个线程设置为守护线程。



1. sleep（）

Thread.sleep(millisec) 方法会休眠当前正在执行的线程，millisec 单位为毫秒。

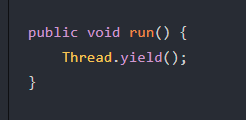
sleep() 可能会抛出 InterruptedException，因为异常不能跨线程传播回 main() 中，因此必须在本地进行处理。线程中抛出的其它异常也同样需要在本地进行处理。



1. yield（）

对**静态方法 Thread.yield()** 的调用声明了当前线程已经完成了生命周期中最重要的部分，可以切换给其它线程来执行。

该方法只是对线程调度器的一个建议，而且也只是建议具有相同优先级的其它线程可以运行。



===========================================================

1. 线程中断

一个线程执行完毕之后会自动结束，如果在运行过程中发生异常也会提前结束。

1. InterruptedException

通过调用一个线程的 **interrupt()** 来中断该线程

如果该线程处于**阻塞**、**限期等待**或者**无限期等待**状态，那么就会抛出 I**nterruptedException**，从而**提前结束该线程**。**但是不能中断 I/O 阻塞和 synchronized 锁阻塞。**

1. Interrupted（）

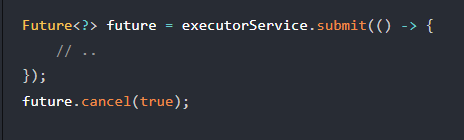
如果一个线程的 run() 方法执行一个**无限循环**，并且**没有执行 sleep() 等会抛出 InterruptedException 的操作**，那么调用线程的 interrupt() 方法就**无法使线程提前结束**。

但是**调用 interrupt() 方法会设置线程的中断标记**，此时调用 interrupted() 方法会返回 **true**。因此**可以在循环体中使用 interrupted() 方法来判断线程是否处于中断状态，从而提前结束线程。**

1. Executor的中断操作

调用 Executor 的 **shutdown() 方法**会等待线程都执行完毕之后再关闭，但是如果调用的是 **shutdownNow() 方法**，**则相当于调用每个线程的 interrupt() 方法。**

如果只想中断 Executor 中的一个线程，可以通过使用 **submit()** 方法来提交一个线程，它会返回一个 Future<?> 对象，通过调用该对象的 **cancel(true)** 方法就可以中断线程。



=========================================================

1. 线程互斥同步---synchronized 和 ReentrantLock

Java 提供了两种锁机制来控制多个线程对共享资源的互斥访问，第一个是 JVM 实现的 synchronized，而另一个是 JDK 实现的 ReentrantLock。

1. synchronized

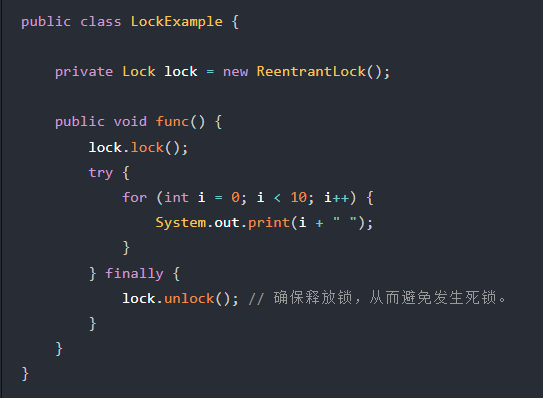
* 同步一个代码块
* 同步一个方法
* 同步一个类
* 同步一个静态方法

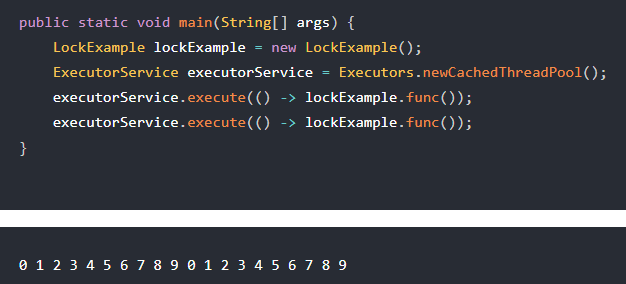
同步一个代码块、一个方法-----**作用于同一个对象**---如果调用两个对象上的同步代码块，就不会进行同步

同步一个类、一个静态方法-----**作用于整个类**---也就是说两个线程调用同一个类的不同对象上的这种同步语句，也会进行同步

1. ReentrantLock

ReentrantLock 是 java.util.concurrent(J.U.C)包中的锁。





1. Synchronized与ReentrantLock的对比

|  |  |  |
| --- | --- | --- |
|  | Synchronized | ReentrantLock |
| **锁的实现** | JVM实现 | JDK实现 |
| **性能**  （新版本 Java 对 synchronized 进行了很多优化，例如自旋锁等） | 性能与ReentrantLock大致相同 | 性能与synchronized大致相同 |
| **等待可中断**  （当持有锁的线程长期不释放锁的时候，正在等待的线程可以选择放弃等待，改为处理其他事情。） | Synchronized不能中断 | ReentrantLock可中断 |
| **公平锁**  （**公平锁是指多个线程在等待同一个锁时，必须按照申请锁的时间顺序来依次获得锁。**） | 非公平 | 默认非公平，但可以是公平 |
| **锁绑定多个条件** |  | 一个 ReentrantLock 可以同时绑定多个Condition对象。 |
| 使用场景 | **除非需要使用 ReentrantLock 的高级功能，否则优先使用 synchronized**。这是因为 synchronized 是 JVM 实现的一种锁机制，JVM 原生地支持它，而 **ReentrantLock 不是所有的 JDK 版本都支持**。并且**使用 synchronized 不用担心没有释放锁而导致死锁问题，因为 JVM 会确保锁的释放。** | |

===========================================================

1. 线程之间的协作

当多个线程可以一起工作去解决某个问题时，如果某些部分必须在其它部分之前完成，那么就需要对线程进行协调。

1. join（）

在线程中调用另一个线程的 join() 方法，会将当前线程挂起，而不是忙等待，直到目标线程结束。

1. wait()、 notify()、 notifyAll()

调用 wait() 使得线程等待某个条件满足，线程在等待时会被挂起，当其他线程的运行使得这个条件满足时，其它线程会调用 notify() 或者 notifyAll() 来唤醒挂起的线程。

**它们都属于 Object 的一部分，而不属于 Thread。**

**只能用在同步方法或者同步控制块中使用，否则会在运行时抛出 IllegalMonitorStateExeception。**

使用 wait() 挂起期间，线程会释放锁。这是因为，如果没有释放锁，那么其它线程就无法进入对象的同步方法或者同步控制块中，那么就无法执行 notify() 或者 notifyAll() 来唤醒挂起的线程，造成死锁。

wait() 和 sleep() 的区别：

* wait() 是 Object 的方法，而 sleep() 是 Thread 的静态方法；
* wait() 会释放锁，sleep() 不会。

1. await() 、signal()、 signalAll()

java.util.concurrent 类库中提供了 Condition 类来实现线程之间的协调，可以在 Condition 上调用 await() 方法使线程等待，其它线程调用 signal() 或 signalAll() 方法唤醒等待的线程。相比于 wait() 这种等待方式，await() 可以指定等待的条件，因此更加灵活。

=================================================================================================================================================================================

Part 3 JUC 类

1. CAS
2. 什么是CAS？

CAS的全称为Compare-And-Swap，就是对比交换。是一条CPU的原子指令，其作用是让CPU先进行比较两个值是否相等，然后原子地更新某个位置的值，经过调查发现，其实现方式是基于硬件平台的汇编指令，就是说CAS是靠硬件实现的，JVM只是封装了汇编调用，那些AtomicInteger类便是使用了这些封装后的接口。

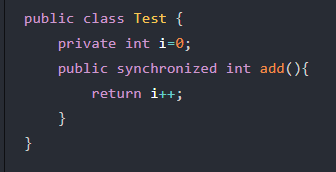
简单解释：CAS操作需要输入两个数值，一个旧值(期望操作前的值)和一个新值，在操作期间先比较下在旧值有没有发生变化，如果没有发生变化，才交换成新值，发生了变化则不交换。

**CAS操作是原子性的，所以多线程并发使用CAS更新数据时，可以不使用锁**。JDK中大量使用了CAS来更新数据而防止加锁(synchronized 重量级锁)来保持原子更新。

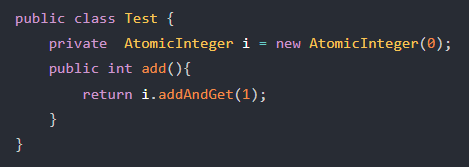
**CAS 方式为乐观锁，synchronized 为悲观锁。因此使用 CAS 解决并发问题通常情况下性能更优。**

1. 怎样使用CAS

如果不使用CAS，在高并发下，多线程同时修改一个变量的值我们需要synchronized加锁(可能有人说可以用Lock加锁，Lock底层的AQS也是基于CAS进行获取锁的)。



java中为我们提供了AtomicInteger 原子类(底层基于CAS进行更新数据的)，不需要加锁就在多线程并发场景下实现数据的一致性。



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

1. UnSafe类

Java原子类是通过UnSafe类实现的，UnSafe类在J.U.C中CAS操作有很广泛的应用

Unsafe是位于sun.misc包下的一个类，主要提供一些用于执行低级别、不安全操作的方法，如直接访问系统内存资源、自主管理内存资源等，这些方法在提升Java运行效率、增强Java语言底层资源操作能力方面起到了很大的作用。但**由于Unsafe类使Java语言拥有了类似C语言指针一样操作内存空间的能力，这无疑也增加了程序发生相关指针问题的风险。**在程序中过度、不正确使用Unsafe类会使得程序出错的概率变大，使得Java这种安全的语言变得不再“安全”，因此**对Unsafe的使用一定要慎重**。

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

1. 原子类

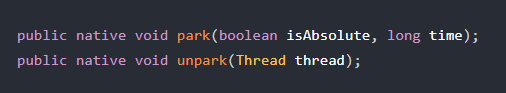
JDK中提供了13个原子操作类。

AtomicInteger 底层用的是**volatile的变量**和**CAS**来进行更改数据的。

* volatile保证线程的可见性，多线程并发时，一个线程修改数据，可以保证其它线程立马看到修改后的值
* CAS 保证数据更新的原子性。

四、LockSupport

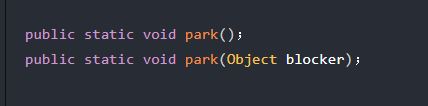
LockSupport用来创建锁和其他同步类的基本线程阻塞原语。简而言之，当调用LockSupport.park时，表示当前线程将会等待，直至获得许可，当调用LockSupport.unpark时，必须把等待获得许可的线程作为参数进行传递，好让此线程继续运行。



* park函数，阻塞线程，并且该线程在下列情况发生之前都会被阻塞: ① 调用unpark函数，释放该线程的许可。② 该线程被中断。③ 设置的时间到了。并且，当time为绝对时间时，isAbsolute为true，否则，isAbsolute为false。当time为0时，表示无限等待，直到unpark发生。
* unpark函数，释放线程的许可，即激活调用park后阻塞的线程。这个函数不是安全的，调用这个函数时要确保线程依旧存活。

1. park函数详解：

Park函数---两个重载版本



两个函数的区别在于park()函数没有没有blocker，即没有设置线程的parkBlocker字段。

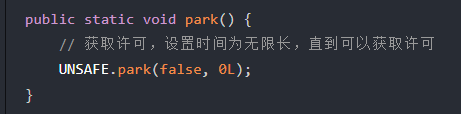
1. park(Object)型函数，即有参重载版本



为什么要在此park函数中要调用两次setBlocker函数呢?

原因其实很简单，调用park函数时，当前线程首先设置好parkBlocker字段，然后再调用Unsafe的park函数，此后，当前线程就已经阻塞了，等待该线程的unpark函数被调用，所以后面的一个setBlocker函数无法运行，unpark函数被调用，该线程获得许可后，就可以继续运行了，也就运行第二个setBlocker，把该线程的parkBlocker字段设置为null，这样就完成了整个park函数的逻辑。如果没有第二个setBlocker，那么之后没有调用park(Object blocker)，而直接调用getBlocker函数，得到的还是前一个park(Object blocker)设置的blocker，显然是不符合逻辑的。总之，必须要保证在park(Object blocker)整个函数执行完后，该线程的parkBlocker字段又恢复为null。所以，park(Object)型函数里必须要调用setBlocker函数两次。

（2）park()无参重载版本

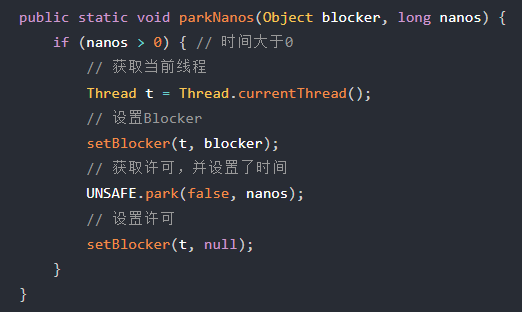


调用了park函数后，会禁用当前线程，除非许可可用。在以下三种情况之一发生之前，当前线程都将处于休眠状态，即下列情况发生时，当前线程会获取许可，可以继续运行。

* 其他某个线程将当前线程作为目标调用 unpark。
* 其他某个线程中断当前线程。
* 该调用不合逻辑地(即毫无理由地)返回。

1. parkUntil函数详解：

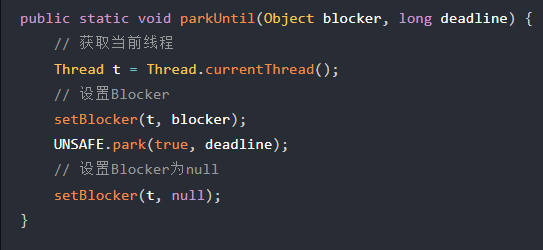
此函数表示在许可可用前禁用当前线程，并最多等待指定的等待时间。



该函数也是调用了两次setBlocker函数，nanos参数表示相对时间，表示等待多长时间。

1. parkUntil函数

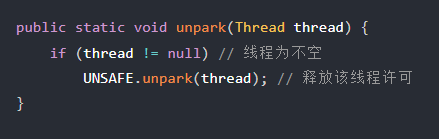
此函数表示在指定的时限前禁用当前线程，除非许可可用



该函数也调用了两次setBlocker函数，deadline参数表示绝对时间，表示指定的时间。

1. unpark函数

此函数表示如果给定线程的许可尚不可用，则使其可用。如果线程在 park 上受阻塞，则它将解除其阻塞状态。否则，保证下一次调用 park 不会受阻塞。如果给定线程尚未启动，则无法保证此操作有任何效果。



说明: 释放许可，指定线程可以继续运行。

**注意：park和unpark的顺序不当不会引起阻塞。先调用park，然后再调用unpark。但先调用unpark，再调用park时，仍能够正确实现同步，不会造成由wait/notify调用顺序不当所引起的阻塞。因此park/unpark相比wait/notify更加的灵活**

Thread.sleep()和Object.wait()的区别？？

* Thread.sleep()不会释放占有的锁，Object.wait()会释放占有的锁；
* Thread.sleep()必须传入时间，Object.wait()可传可不传，不传表示一直阻塞下去；
* Thread.sleep()到时间了会自动唤醒，然后继续执行；
* Object.wait()不带时间的，需要另一个线程使用Object.notify()唤醒；
* Object.wait()带时间的，假如没有被notify，到时间了会自动唤醒，这时又分好两种情况，一是立即获取到了锁，线程自然会继续执行；二是没有立即获取锁，线程进入同步队列等待获取锁；

其实，他们俩最大的区别就是Thread.sleep()不会释放锁资源，Object.wait()会释放锁资源。

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

Object.wait()和Condition.await()的区别？？

--Object.wait()和Condition.await()的原理是基本一致的，不同的是Condition.await()底层是调用LockSupport.park()来实现阻塞当前线程的。

实际上，它在阻塞当前线程之前还干了两件事，一是把当前线程添加到条件队列中，二是“完全”释放锁，也就是让state状态变量变为0，然后才是调用LockSupport.park()阻塞当前线程。

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

如果在wait()之前执行了notify()会怎样?

--如果当前的线程不是此对象锁的所有者，却调用该对象的notify()或wait()方法时抛出IllegalMonitorStateException异常；

如果当前线程是此对象锁的所有者，wait()将一直阻塞，因为后续将没有其它notify()唤醒它。

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

Thread.sleep()和LockSupport.park()的区别

LockSupport.park()还有几个兄弟方法——parkNanos()、parkUtil()等，我们这里说的park()方法统称这一类方法。

* 从功能上来说，Thread.sleep()和LockSupport.park()方法类似，都是阻塞当前线程的执行，且都不会释放当前线程占有的锁资源；
* Thread.sleep()没法从外部唤醒，只能自己醒过来；
* LockSupport.park()方法可以被另一个线程调用LockSupport.unpark()方法唤醒；
* Thread.sleep()方法声明上抛出了InterruptedException中断异常，所以调用者需要捕获这个异常或者再抛出；
* LockSupport.park()方法不需要捕获中断异常；
* Thread.sleep()本身就是一个native方法；
* LockSupport.park()底层是调用的Unsafe的native方法；

Object.wait()和LockSupport.park()的区别

二者都会阻塞当前线程的运行，他们有什么区别呢?

* Object.wait()方法需要在synchronized块中执行；
* LockSupport.park()可以在任意地方执行；
* Object.wait()方法声明抛出了中断异常，调用者需要捕获或者再抛出；
* LockSupport.park()不需要捕获中断异常；
* Object.wait()不带超时的，需要另一个线程执行notify()来唤醒，但不一定继续执行后续内容；
* LockSupport.park()不带超时的，需要另一个线程执行unpark()来唤醒，一定会继续执行后续内容；

park()/unpark()底层的原理是“二元信号量”，你可以把它相像成只有一个许可证的Semaphore，只不过这个信号量在重复执行unpark()的时候也不会再增加许可证，最多只有一个许可证。

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

如果在park()之前执行了unpark()会怎样?

--线程不会被阻塞，直接跳过park()，继续执行后续内容

LockSupport.park()会释放锁资源吗?

--不会，它只负责阻塞当前线程，释放锁资源实际上是在Condition的await()方法中实现的。

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

五、锁核心类AQS（AbstractQueuedSynchronizer）

1. AQS是什么？

**AQS是一个用来构建锁和同步器的框架**，使用AQS能简单且高效地构造出应用广泛的大量的同步器，比如我们提到的ReentrantLock，Semaphore，其他的诸如ReentrantReadWriteLock，SynchronousQueue，FutureTask等等皆是基于AQS的。

当然，我们自己也能利用AQS非常轻松容易地构造出符合我们自己需求的同步器。

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

1. AQS的核心思想：

AQS核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制AQS是用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中。

CLH(Craig,Landin,and Hagersten)队列是一个虚拟的双向队列(虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系)。AQS是将每条请求共享资源的线程封装成一个CLH锁队列的一个结点(Node)来实现锁的分配。

AQS使用一个int成员变量**state**来表示同步状态，通过内置的FIFO队列来完成获取资源线程的排队工作。AQS使用CAS对该同步状态进行原子操作实现对其值的修改。





1. AQS框架借助于两个类：**Unsafe(提供CAS操作)**和**LockSupport(提供park/unpark操作)**
2. AQS对资源的共享方式

AQS定义两种资源共享方式

* Exclusive(独占)：只有一个线程能执行，如ReentrantLock。

又可分为公平锁和非公平锁：

* 公平锁：按照线程在队列中的排队顺序，先到者先拿到锁
* 非公平锁：当线程要获取锁时，无视队列顺序直接去抢锁，谁抢到就是谁的
* Share(共享)：多个线程可同时执行，如Semaphore、CountDownLatch、 CyclicBarrier )。ReentrantReadWriteLock可以看成是组合式，允许多个线程同时对某一资源进行读。

1. AQS底层使用了什么设计模式？？？

模板方法模式， 共享锁和独占锁在一个接口类中。

同步器的设计是基于模板方法模式的。

使用者继承AbstractQueuedSynchronizer并重写指定的方法。AQS使用了模板方法模式，自定义同步器时需要重写下面几个AQS提供的模板方法（AQS的5个核心方法）：

* isHeldExclusively()//该线程是否正在独占资源。只有用到condition才需要去实现它。
* tryAcquire(int)//独占方式。尝试获取资源，成功则返回true，失败则返回false。
* tryRelease(int)//独占方式。尝试释放资源，成功则返回true，失败则返回false。
* tryAcquireShared(int)//共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
* tryReleaseShared(int)//共享方式。尝试释放资源，成功则返回true，失败则返回false。

默认情况下，每个方法都抛出 UnsupportedOperationException。 这些方法的实现必须是内部线程安全的，并且通常应该简短而不是阻塞。AQS类中的其他方法都是final ，所以无法被其他类使用，只有这几个方法可以被其他类使用。

以ReentrantLock为例，state初始化为0，表示未锁定状态。A线程lock()时，会调用tryAcquire()独占该锁并将state+1。此后，其他线程再tryAcquire()时就会失败，直到A线程unlock()到state=0(即释放锁)为止，其它线程才有机会获取该锁。当然，释放锁之前，A线程自己是可以重复获取此锁的(state会累加)，这就是可重入的概念。但要注意，获取多少次就要释放多么次，这样才能保证state是能回到零态的。

AbstractQueuedSynchronizer数据结构

AbstractQueuedSynchronizer类底层的数据结构是使用CLH(Craig,Landin,and Hagersten)队列是一个虚拟的双向队列(虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系)。

AQS是将每条请求共享资源的线程封装成一个CLH锁队列的一个结点(Node)来实现锁的分配。其中Sync queue，即同步队列，是双向链表，包括head结点和tail结点，head结点主要用作后续的调度。

**而Condition queue不是必须的，其是一个单向链表，只有当使用Condition时，才会存在此单向链表。并且可能会有多个Condition queue。**

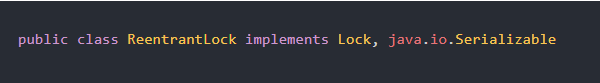


六、可重入锁ReentrantLock

可重入锁ReentrantLock的底层是通过AbstractQueuedSynchronizer（AQS）实现

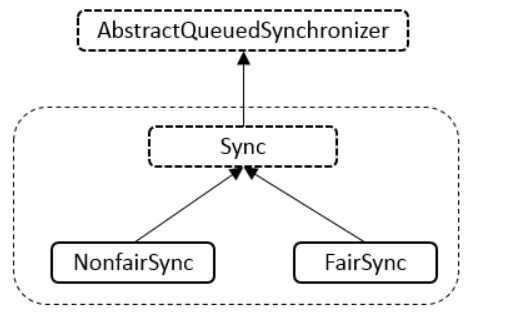
1. ReentrantLock的继承关系

ReentrantLock实现了Lock接口，Lock接口中定义了lock与unlock相关操作，并且还存在newCondition方法，表示生成一个条件。



1. ReentrantLock类的内部类

ReentrantLock总共有三个内部类，**Sync**、**NonfairSync**、**FairSync**，关系如下：



说明: ReentrantLock类内部总共存在Sync、NonfairSync、FairSync三个类，NonfairSync与FairSync类继承自Sync类，Sync类继承自AbstractQueuedSynchronizer抽象类。

* Sync类---一个静态抽象类



* NonfairSync类---NonfairSync类继承了Sync类，表示采用非公平策略获取锁，其实现了Sync类中抽象的lock方法

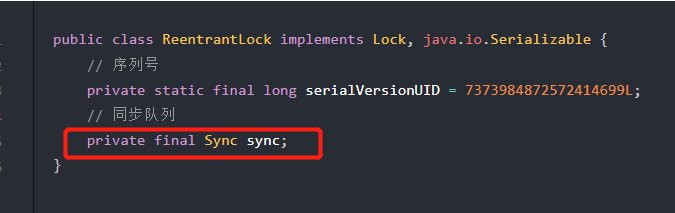
说明: 从lock方法的源码可知，每一次都尝试获取锁，而并不会按照公平等待的原则进行等待，让等待时间最久的线程获得锁。

* FairSync类---FairSync类也继承了Sync类，表示采用公平策略获取锁，其实现了Sync类中的抽象lock方法

说明：当资源空闲时，它总是会先判断sync队列(AbstractQueuedSynchronizer中的数据结构)是否有等待时间更长的线程，如果存在，则将该线程加入到等待队列的尾部，实现了公平获取原则。 只要资源被其他线程占用，该线程就会添加到sync queue中的尾部，而不会先尝试获取资源。这也是和Nonfair最大的区别，Nonfair每一次都会尝试去获取资源，如果此时该资源恰好被释放，则会被当前线程获取，这就造成了不公平的现象，当获取不成功，再加入队列尾部。

1. ReentrantLock类的属性：

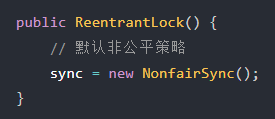
ReentrantLock类的**sync**非常重要，对ReentrantLock类的操作大部分都直接转化为对Sync和AbstractQueuedSynchronizer类的操作。



1. ReentrantLock类的构造函数---有参、无参

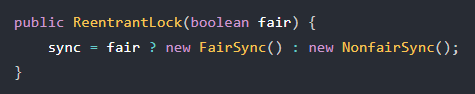
* ReentrantLock()型构造函数

**默认**是采用的**非公平策略**获取锁



* ReentrantLock(boolean)型构造函数

可以传递参数确定采用公平策略或者是非公平策略，**参数为true表示公平策略**，否则，采用非公平策略



1. 核心函数分析：

对ReentrantLock的操作--->可以转化成对Sync对象的操作--->因为Sync继承AQS--->所以基本上可以转化为对AQS的操作。

1. 公平锁和非公平锁的体现：以非公平锁为例：

未完待续！

七、ConcurrentHashMap

1. 为什么HashTable慢

Hashtable之所以效率低下主要是因为其实现**使用了synchronized关键字对put等操作进行加锁，而synchronized关键字加锁是对整个对象进行加锁**，也就是说在进行put等修改Hash表的操作时，锁住了整个Hash表，从而使得其表现的效率低下。

ConcurrentHashMap - **JDK 1.7**

1. 锁实现：在JDK1.5~1.7版本，Java使用了**分段锁机制**实现ConcurrentHashMap.

分析：

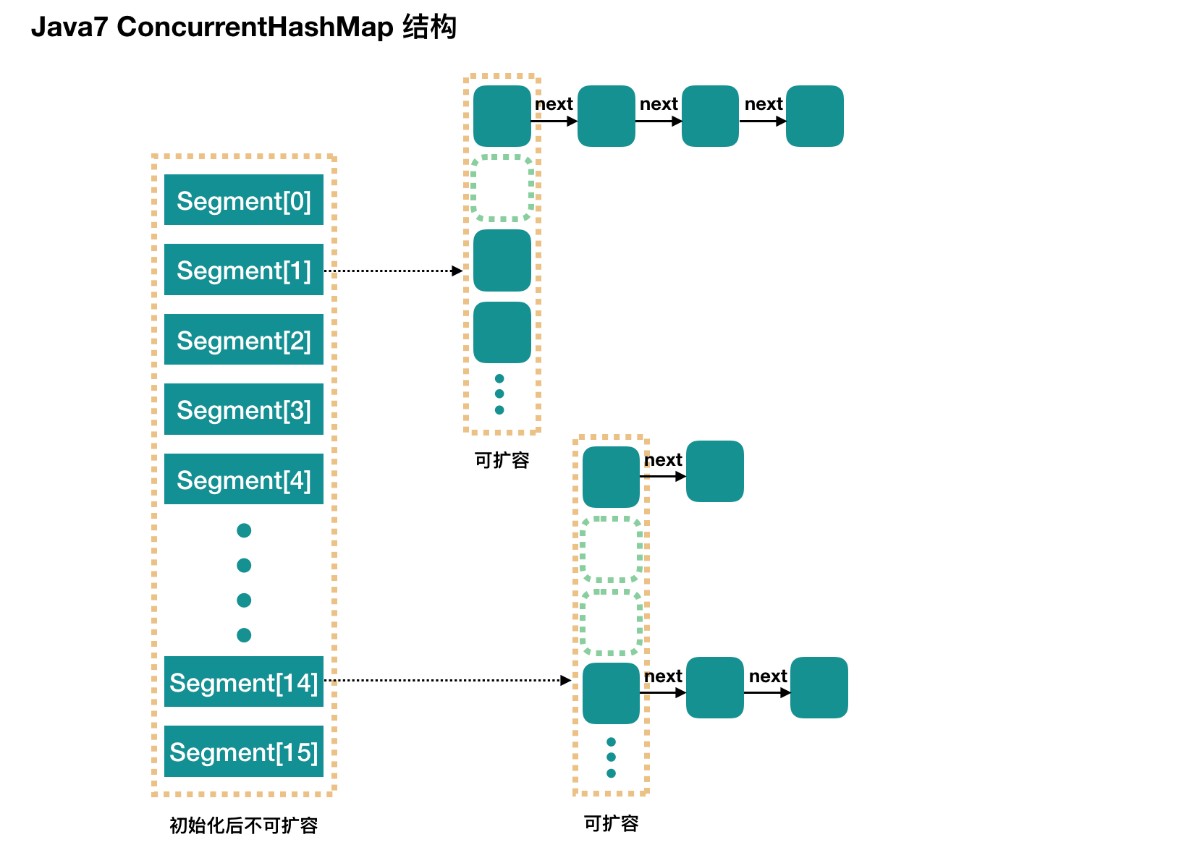
（1）ConcurrentHashMap在对象中保存了一个Segment数组，即将整个Hash表划分为多个分段；

（2）而每个Segment元素，即每个分段则类似于一个Hashtable；

（3）这样，在执行put操作时首先根据hash算法定位到元素属于哪个Segment，然后对该Segment加锁即可。

1. 因此，ConcurrentHashMap在多线程并发编程中可是实现多线程put操作。
2. 底层实现：**Segment 内部是由 数组+链表 组成的**。

ConcurrentHashMap 是一个 Segment 数组，Segment 通过继承 ReentrantLock 来进行加锁，所以每次需要加锁的操作锁住的是一个 segment，这样只要保证每个 Segment 是线程安全的，也就实现了全局的线程安全。



参数：

* concurrencyLevel: 并行级别、并发数、Segment 数

**默认是 16**，也就是说 ConcurrentHashMap 有 16 个 Segments

所以理论上，这个时候，最多可以同时支持 16 个线程并发写，只要它们的操作分别分布在不同的 Segment 上。

**这个值可以在初始化的时候设置为其他值，但是一旦初始化以后，它是不可以扩容的。**

1. 初始化：

* initialCapacity: 初始容量，这个值指的是**整个 ConcurrentHashMap 的初始容量**，实际操作的时候需要**平均分给每个 Segment**。
* loadFactor: 负载因子， **Segment 数组不可以扩容**，所以**这个负载因子是给每个 Segment 内部使用的**。

初始化完成，我们得到了一个 Segment 数组。我们就当是用 new ConcurrentHashMap() 无参构造函数进行初始化的，那么初始化完成后：

* Segment 数组长度为 16，不可以扩容
* Segment[i] 的默认大小为 2，负载因子是 0.75，得出初始阈值为 1.5，也就是以后插入第一个元素不会触发扩容，插入第二个会进行第一次扩容
* 这里初始化了 segment[0]，其他位置还是 null，至于为什么要初始化 segment[0]，后面的代码会介绍
* 当前 segmentShift 的值为 32 - 4 = 28，segmentMask 为 16 - 1 = 15，姑且把它们简单翻译为**移位数**和**掩码**，这两个值马上就会用到

1. 初始化槽: ensureSegment

ConcurrentHashMap 初始化的时候会初始化第一个槽 segment[0]，对于其他槽来说，在插入第一个值的时候进行初始化。

1. 获取写入锁: scanAndLockForPut

在往某个 segment 中 put 的时候，首先会调用 node = tryLock() ? null : scanAndLockForPut(key, hash, value)，也就是说先进行一次 tryLock() 快速获取该 segment 的独占锁，如果失败，那么进入到 scanAndLockForPut 这个方法来获取锁。

这个方法有两个出口，一个是 tryLock() 成功了，循环终止，另一个就是重试次数超过了 MAX\_SCAN\_RETRIES，进到 lock() 方法，此方法会阻塞等待，直到成功拿到独占锁。

1. 扩容: rehash

重复一下，**segment 数组不能扩容**，

**扩容是 segment 数组某个位置内部的数组 HashEntry<K,V>[] 进行扩容**，

**扩容后，容量为原来的 2 倍**。

1. get过程分析

相对于 put 来说，get 就很简单了：

* 计算 hash 值，找到 segment 数组中的具体位置，或我们前面用的“槽”
* 槽中也是一个数组，根据 hash 找到数组中具体的位置
* 到这里是链表了，顺着链表进行查找即可

1. get过程导致的并发问题分析

get 过程中是没有加锁的，那自然我们就需要去考虑并发问题。

添加节点的操作 put 和删除节点的操作 remove 都是要加 segment 上的独占锁的，所以它们之间自然不会有问题

我们需要考虑的问题就是 **get 的时候在同一个 segment 中发生了 put 或 remove 操作**

* put 操作的线程安全性。
* 初始化槽，这个我们之前就说过了，使用了 CAS 来初始化 Segment 中的数组。
* 添加节点到链表的操作是插入到表头的，所以，如果这个时候 get 操作在链表遍历的过程已经到了中间，是不会影响的。当然，另一个并发问题就是 get 操作在 put 之后，需要保证刚刚插入表头的节点被读取，这个依赖于 setEntryAt 方法中使用的 UNSAFE.putOrderedObject。
* 扩容。扩容是新创建了数组，然后进行迁移数据，最后面将 newTable 设置给属性 table。所以，如果 get 操作此时也在进行，那么也没关系，如果 get 先行，那么就是在旧的 table 上做查询操作；而 put 先行，那么 put 操作的可见性保证就是 table 使用了 volatile 关键字。
* remove 操作的线程安全性。
* remove 操作我们没有分析源码，所以这里说的读者感兴趣的话还是需要到源码中去求实一下的。
* get 操作需要遍历链表，但是 remove 操作会"破坏"链表。
* 如果 remove 破坏的节点 get 操作已经过去了，那么这里不存在任何问题。
* 如果 remove 先破坏了一个节点，分两种情况考虑。 1、如果此节点是头节点，那么需要将头节点的 next 设置为数组该位置的元素，table 虽然使用了 volatile 修饰，但是 volatile 并不能提供数组内部操作的可见性保证，所以源码中使用了 UNSAFE 来操作数组，请看方法 setEntryAt。2、如果要删除的节点不是头节点，它会将要删除节点的后继节点接到前驱节点中，这里的并发保证就是 next 属性是 volatile 的。

ConcurrentHashMap - **JDK 1.8**

在JDK1.7之前，ConcurrentHashMap是通过分段锁机制来实现的，所以其最大并发度受Segment的个数限制。

因此，在JDK1.8中，ConcurrentHashMap的实现原理摒弃了这种设计，而是选择了与HashMap类似的**数组+链表+红黑树**的方式实现，而**加锁则采用CAS和synchronized实现**。

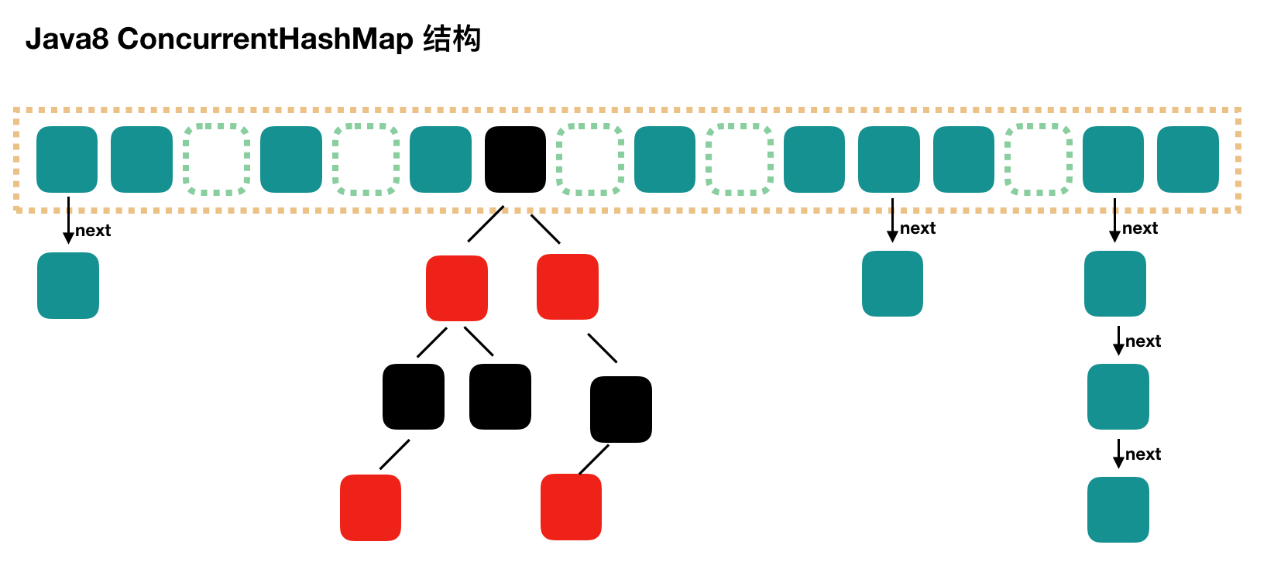
1. 锁实现

**采用CAS和synchronized实现**

1. 底层实现

与HashMap类似的**数组+链表+红黑树**的方式实现

1. 数据结构---结构基本与java8的hashmap一样，但要保证线程安全，源码更复杂点



1. 初始化

这个初始化方法有点意思，通过提供初始容量，计算了 sizeCtl，**sizeCtl = 【 (1.5 \* initialCapacity + 1)**，然后向上取最近的 2 的 n 次方】。如 initialCapacity 为 10，那么得到 sizeCtl 为 16，如果 initialCapacity 为 11，得到 sizeCtl 为 32。

1. put过程分析
2. 初始化数组：initTable

主要就是初始化一个合适大小的数组，然后会设置 sizeCtl。

初始化方法中的并发问题是通过对 sizeCtl 进行一个 CAS 操作来控制的。

1. 链表转红黑树：treeifyBin

treeifyBin 不一定就会进行红黑树转换，也可能是仅仅做数组扩容。

1. 扩容：tryPresize

如果说 Java8 ConcurrentHashMap 的源码不简单，那么说的就是扩容操作和迁移操作。

这里的扩容也是做翻倍扩容的，扩容后数组容量为原来的 2 倍。

1. 数据迁移：transfer

将原来的 tab 数组的元素迁移到新的 nextTab 数组中

1. get过程分析

* 计算 hash 值
* 根据 hash 值找到数组对应位置: (n - 1) & h
* 根据该位置处结点性质进行相应查找
* 如果该位置为 null，那么直接返回 null 就可以了
* 如果该位置处的节点刚好就是我们需要的，返回该节点的值即可
* 如果该位置节点的 hash 值小于 0，说明正在扩容，或者是红黑树，后面我们再介绍 find 方法
* 如果以上 3 条都不满足，那就是链表，进行遍历比对即可

对比总结

* HashTable : 使用了synchronized关键字对put等操作进行加锁;
* ConcurrentHashMap JDK1.7: 使用分段锁机制实现;
* ConcurrentHashMap JDK1.8: 则使用数组+链表+红黑树数据结构和CAS原子操作实现;