深入JVM锁机制1–synchronized

原创 2011年07月28日 16:24:41 标签：vm / cache / 汇编 / locking / list / 数据结构 30022

目前在Java中存在两种锁机制：synchronized和Lock，Lock接口及其实现类是JDK5增加的内容，其作者是大名鼎鼎的并发专家Doug Lea。本文比较synchronized与Lock孰优孰劣，只是介绍二者的实现原理。

数据同步需要依赖锁，那锁的同步又依赖谁？synchronized给出的答案是在软件层面依赖JVM，而Lock给出的方案是在硬件层面依赖特殊的CPU指令，大家可能会进一步追问：JVM底层又是如何实现synchronized的？

本文所指说的JVM是指Hotspot的6版本，下面首先介绍synchronized的实现：

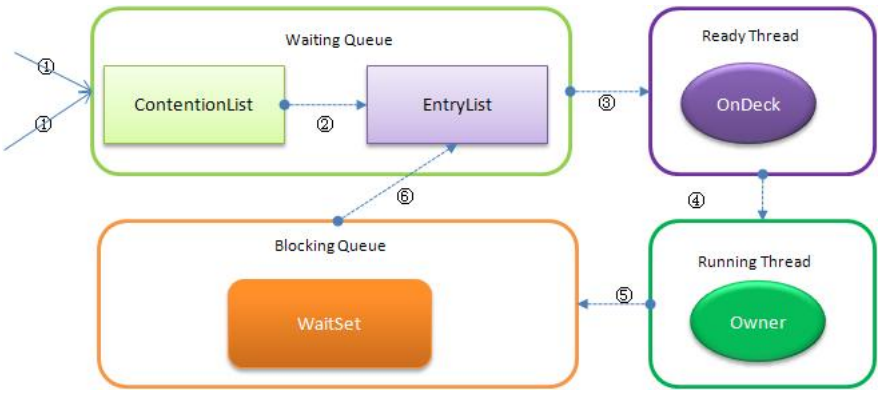
synrhronized关键字简洁、清晰、准确，因此即使有了Lock接口，使用的还是非常广泛。其应用层的语义是可以把任何一个非null对象作为"锁"，当synchronized作用在方法上时，锁住的便是对象实例（this）；当作用在静态方法时锁住的便是对象对应的Class实例，因为Class数据存在于永久带，因此静态方法锁相当于该类的一个全局锁；当synchronized作用在一个对象实例时，锁住的便是对应的代码块。在HotSpot JVM实现中，锁有个专门的名字：对象监视器。

1. 线程状态及状态转换

当多个线程同时请求某个对象监视器时，对象监视器会设置几种状态用来区分请求的线程：

- Contention List：所有请求锁的线程将被首先放置到该竞争队列
- Entry List：Contention List中那些有资格成为候选人的线程被移到Entry List
- Wait Set：那些调用wait方法被阻塞的线程被放置到Wait Set
- OnDeck：任何时刻最多只能有一个线程正在竞争锁，该线程称为OnDeck
- Owner：获得锁的线程称为Owner
- !Owner：释放锁的线程

下图反映了个状态转换关系：



新请求锁的线程将首先被加入到ConetentionList中，当某个拥有锁的线程（Owner状态）调用unlock之后，如果发现EntryList为空则从ContentionList中移动线程到EntryList，下面说明下ContentionList和EntryList的实现方式：

1.1 ContentionList虚拟队列

ContentionList并不是一个真正的Queue，而只是一个虚拟队列，原因在于ContentionList是由Node及其next指针逻辑构成，并不存在一个Queue的数据结构。ContentionList是一个后进先出（LIFO）的队列，每次新加入Node时都会在队头进行，通过CAS改变第一个节点的的指针为新增节点，同时设置新增节点的next指向后续节点，而取得操作则发生在队尾。显然，该结构其实是个Lock-Free的队列。

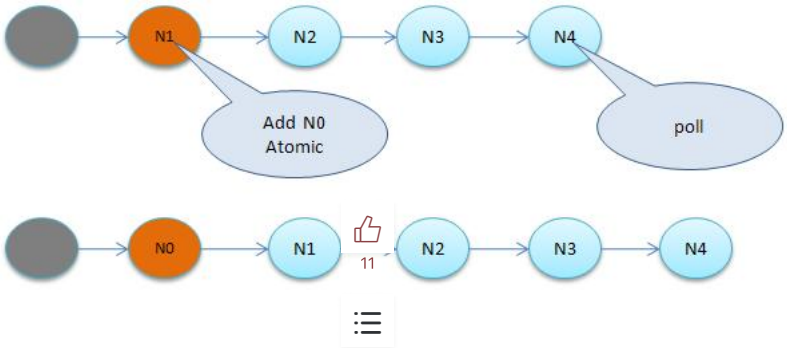
因为只有Owner线程才能从队尾取元素，也即线程出列操作无争用，当然也就避免了CAS的ABA问题。

纯粹的码农

原创 32 粉丝 510 喜欢 5

等级：博客 5 访问量：58

积分：4073 排名：9311



1.2 EntryList

EntryList与ContentionList逻辑上同属等待队列，ContentionList会被线程并发访问，为了降低对ContentionList队尾的争用，而建立EntryList。Owner线程在unlock时会从ContentionList中迁移线程到EntryList，并会指定EntryList中的某个线程（一般为Head）为Ready（OnDeck）线程。Owner线程并不是把锁传递给OnDeck线程，只是把竞争锁的权利交给OnDeck，OnDeck线程需要重新竞争锁。这样做虽然牺牲了一定的公平性，但极大的提高了整体吞吐量，在Hotspot中把OnDeck的选择行为称之为“竞争切换”。

OnDeck线程获得锁后即变为owner线程，无法获得锁则会依然留在EntryList中，考虑到公平性，在EntryList中的位置不发生变化（依然在队头）。如果Owner线程被wait方法阻塞，则转移到WaitSet队列；如果在某个时刻被notify/notifyAll唤醒，则再次转移到EntryList。

2. 自旋锁

那些处于ContetionList、EntryList、WaitSet中的线程均处于阻塞状态，阻塞操作由操作系统完成（在Linxu下通过pthread_mutex_lock函数）。线程被阻塞后便进入内核（Linux）调度状态，这个会导致系统在用户态与内核态之间来回切换，严重影响锁的性能

缓解上述问题的办法便是自旋，其原理是：当发生争用时，若Owner线程能在很短的时间内释放锁，则那些正在争用线程可以稍微等一等（自旋），在Owner线程释放锁后，争用线程可能会立即得到锁，从而避免了系统阻塞。但Owner运行的时间可能会超出了临界值，争用线程自旋一段时间后还是无法获得锁，这时争用线程则会停止自旋进入阻塞状态（后退）。基本思路就是自旋，不成功再阻塞，尽量降低阻塞的可能性，这对那些执行时间很短的代码块来说有非常重要的性能提高。自旋锁有个更贴切的名字：自旋-指数后退锁，也即复合锁。很显然，自旋在多处理器上才有意义。

还有个问题是，线程自旋时做些啥？其实啥都不做，可以执行几次for循环，可以执行几条空的汇编指令，目的是占着CPU不放，等待获取锁的机会。所以说，自旋是把双刃剑，如果旋的时间过长会影响整体性能，时间过短又达不到延迟阻塞的目的。显然，自旋的周期选择显得非常重要，但这与操作系统、硬件体系、系统的负载等诸多场景相关，很难选择，如果选择不当，不但性能得不到提高，可能还会下降，因此大家普遍认为自旋锁不具有扩展性。

对自旋锁周期的选择上，HotSpot认为最佳时间应是一个线程上下文切换的时间，但目前并没有做到。经过调查，目前只是通过汇编暂停了几个CPU周期，除了自旋周期选择，HotSpot还进行许多其他的自旋优化策略，具体如下：

- 如果平均负载小于CPUs则一直自旋
- 如果有超过(CPUs/2)个线程正在自旋，则后来线程直接阻塞
- 如果正在自旋的线程发现Owner发生了变化则延迟自旋时间（自旋计数）或进入阻塞
- 如果CPU处于节电模式则停止自旋
- 自旋时间的最坏情况是CPU的存储延迟（CPU A存储了一个数据，到CPU B得知这个数据直接的时间差）
- 自旋时会适当放弃线程优先级之间的差异

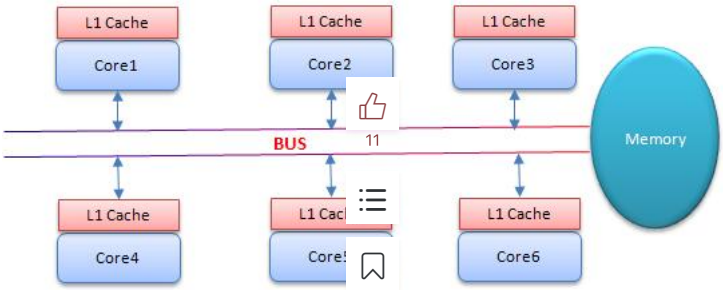
那synchronized实现何时使用了自旋锁？答案是在线程进入ContentionList时，也即第一步操作前。线程在进入等待队列时首先进行自旋尝试获得锁，如果不成功再进入等待队列。这对那些已经在等待队列中的线程来说，稍微显得不公平。还有一个不公平的地方是自旋线程可能会抢占了Ready线程的锁。自旋锁由每个监视对象维护，每个监视对象一个。

3. 偏向锁

在JVM1.6中引入了偏向锁，偏向锁主要解决无竞争下的锁性能问题，首先我们看下无竞争下锁存在什么问题：现在几乎所有的锁都是可重入的，也即已经获得锁的线程可以多次锁住/解锁监视对象，按照之前的HotSpot设计，每次加锁/解锁都会涉及到一些CAS操作（比如对等待队列的CAS操作），CAS操作会延迟本地调用，因此偏向锁的想法是一旦线程第一次获得了监视对象，之后让监视对象“偏向”这个线程，之后的多次调用则可以避免CAS操作，说白了就是置个变量，如果发现为true则无需再走各种加锁/解锁流程。但还有很多概念需要解释、很多引入的问题需要解决：

3.1 CAS及SMP架构

CAS为什么会引入本地延迟？这要从SMP（对称多处理器）架构说起，下图大概表明了SMP的结构：



其意思是所有的CPU会共享一条系统[illegible]（BUS），靠此总线连接主存。每个核都有自己的一级缓存，各核相对于BUS对称分布，因此这种结构称为“对称多处理器”。

而CAS的全称为Compare-And-Swap，是一条CPU的原子指令，其作用是让CPU比较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。

Core1和Core2可能会同时把主存中某个位置的值Load到自己的L1 Cache中，当Core1在自己的L1 Cache中修改这个位置的值时，会通过总线，使Core2中L1 Cache对应的值“失效”，而Core2一旦发现自己L1 Cache中的值失效（称为Cache命中缺失）则会通过总线从内存中加载该地址最新的值，大家通过总线的来回通信称为“Cache一致性流量”，因为总线被设计为固定的“通信能力”，如果Cache一致性流量过大，总线将成为瓶颈。而当Core1和Core2中的值再次一致时，称为“Cache一致性”，从这个层面来说，锁设计的终极目标便是减少Cache一致性流量。

而CAS恰好会导致Cache一致性流量，如果有很多线程都共享同一个对象，当某个Core CAS成功时必然会引起总线风暴，这就是所谓的本地延迟，本质上偏向锁就是为了消除CAS，降低Cache一致性流量。

Cache一致性：
上面提到Cache一致性，其实是有协议支持的，现在通用的协议是MESI（最早由Intel开始支持），具体参考：http://en.wikipedia.org/wiki/MESI_protocol，以后会仔细讲解这部分。
Cache一致性流量的例外情况：
其实也不是所有的CAS都会导致总线风暴，这跟Cache一致性协议有关，具体参考：http://blogs.oracle.com/dave/entry/biased_locking_in_hotspot
NUMA(Non Uniform Memory Access Achitecture）架构：
与SMP对应还有非对称多处理器架构，现在主要应用在一些高端处理器上，主要特点是没有总线，没有公用主存，每个Core有自己的内存，针对这种结构此处不做讨论。

3.2 偏向解除

偏向锁引入的一个重要问题是，在多争用的场景下，如果另外一个线程争用偏向对象，拥有者需要释放偏向锁，而释放的过程会带来一些性能开销，但总体说来偏向锁带来的好处还是大于CAS代价的。

4. 总结

关于锁，JVM中还引入了一些其他技术比如锁膨胀等，这些与自旋锁、偏向锁相比影响不是很大，这里就不做介绍。
通过上面的介绍可以看出，synchronized的底层实现主要依靠Lock-Free的队列，基本思路是自旋后阻塞，竞争切换后继续竞争锁，稍微牺牲了公平性，但获得了高吞吐量。下面会继续介绍JVM锁中的Lock（深入JVM锁2-Lock）。


ZDreamBoy 2018-01-21 17:51 #13楼 回复
越看越糊涂了，好多文章都说monitor是会在请求获取所得线程中进行与对象关联，可看了这篇文章我又蒙了。。。