链滴

高并发程序设计（6）—— 有锁和无锁（ThreadLocal 和 CAS）

作者：wanglei0622

原文链接：https://ld246.com/article/1488957154209

来源网站：链滴

<h1>锁</h1>

synchronized(同步锁)，ReentrantLock(重入锁)、ReadWritLock(读写锁)这些都是通过加锁的方式保证多线程之间共享资源的一致性，使用多线程会明显的提升统性能，但是多线程也会额外增加系统的开销，除了处理线程本身的任务外，还要维护多线程环境特的信息，和线程的调度和上下文切换。
死锁：使用重入锁的限时等待可以有效规避死锁。

<h2>提高锁的性能的几点建议</h2>

减小锁持有时间
减小锁粒度。

对于HashMap来说最重要的就是put和get两个方法，最自然到的就是对整个HashMap加锁，必然得到一个线程安全的对象，但是这样做加锁粒度太大，ConCurrntHashMap它内部进一步细分若干个小的HashMap,称之为段（segment）,默认情况下一个onCurruntHashMap分为16个段，如果需要put一个值，并不是将整个hashMap加锁，而根据hashcode得到该值应该存放在哪一个段中，然后对该段加锁，完成put操作，多线程环境中，多线程进行put操作，只要新增的值不存在同一个段中，就可能达到真正的并行。
减少锁粒度会有一个新问题，当系统需要获得全局锁时，消耗资源更多，如ConCurruntHashMap的size()方法。只有在获取全局信息方法调用不频繁时，种减小粒度的方法才有意义。

读写分离来替换独占锁。如果减少粒度是通过分割数据结构实，那么读写锁则是对系统功能点分割。
锁分离，将读写锁进一步延伸，读写锁根据读写操作的不同，行了有效的锁分离。

LinkedBlockingQueue的实现中，take和put函数分别实现从列中去数据和增加数据，虽然都对队列数据进行修改，但由于LinkedBlockingQueue是基于表的，因此两个操作分别作用在队列的头和尾，理论上并不冲突，如果使用独占锁，那么两个操作就能并行，影响高并发时的性能，因此JDK的实现中采用的是两把不同的锁ReentrantLock分离 take和put操作，实现真正意义的并发操作。

锁粗化，锁粗化和减少锁持有时间恰好相反，不同场合他们效不同，根据实际情况权衡。

<h2>Java虚拟机对锁优化</h2>

锁偏向：如果一个线程获得锁，那么就进入偏向模式，这个线再次请求锁时，无需再做同步操作，节省申请时间，在几乎没有锁竞争的场合，偏向锁有比较好的优效果。竞争激烈的场合，每次都是不同的线程来申请，偏向模式就失效了，还不如不用，可以关掉虚机的偏向锁。
轻量级锁：如果偏向锁失败，虚拟机不会立刻挂起线程，还会用一种轻量级锁的优化手段，如果轻量级锁也失败，当前线程的锁就膨胀为重量级锁。
自旋锁：锁膨胀后为避免线程真实的在系统层面挂起，系统会行一次赌注，假设线程在不久的将来会得到这把锁，因此，虚拟机会让当前线程循环几次，经过若干循环后，如果可以得到锁，就顺利进入临界区，否则就真的挂起。
锁消除：Java虚拟机在JIT编译时，通过对上下文扫描，去除可能存在共享资源竞争的锁，节省毫无意义的请求锁的时间，如使用JDK内置API的StringBuffer、Vector等，涉及逃逸分析技术，可以设置打开关闭。

<h1>无锁</h1>

ThreadLocal:如果加锁是100个人用一只笔，那ThreadLocal是100个人每人一只笔，通过增加资源解决竞争。适合用在共享对象对于竞争的处理会引起能问题时。如在多线程环境生成随机数的问题，并发调用日期解析的方法SimpleDateFormat.parse()因为他们都不是线程安全的方法。
CAS：比较交换，适合对共享数据修改时使用，JD并发包中有一个atomic包，里面提供一些直接使用CAS指令操作的线程安全数据类型，对并发控制，锁是一种悲观策略，假设每一次临界区的访问都会产生冲突，只运行一个线程进入，而无是一种乐观策略，假设对资源的访问没有冲突，只在提交操作时检查是否违反数据完整性，发下CAS操作不容易成功，一般要多次循环尝试，直到成功，乐观锁不能解决脏读的问题。/strong>

<h2>ThreadLocal</h2>

它是线程的局部变量，只有当前线程能访问，自然是线程安全。
变量的维护是在Thread内部，意味着只要线程不退，对象将一直引用，当线程退出时，Thread类会进行清理工作，因此如果我们使用线程池，意味着当的线程未必会退出，如果这样ThreadLocal将有内存泄露的可能，所以用完ThreadLocal保存的对象，应该及时清理手动设置null,或者remove保存的对象。

<h2>CAS</h2>

使用CAS会使程序看起来更复杂，但由于他非阻塞性，天生对锁免疫，而且线程间的互相影响也远远小于基于锁的方式，更重要的是使用无锁的方式完全没有使用竞争带来的系统开销，也没有线程间频繁调度带来的开销，因此他比锁拥有更优越的性能。
CAS算法包含三个参数CAS(V,E,N),v表示要更新的量，E表示预期的值，N表示新值，当V的值等于E时才会将V的值设置为N。现实中，还可能存在另一情况，我能能否修改对象的值，不仅取决于当前值，还和对象的过程变化有关，这时就要增加一个时戳或一个状态值，更新数据时同时更新状态值，设置对象值时，预期值和状态值都必须满足，写入才功。JDK提供的AtomicStampedReference就提供了这个功能。
