一种新的无锁算法

# 1、背景

在分布式环境中，经常会出现多台服务器同时竞争同一个资源（如数据库某条纪录更新权）的情况，遇到这样的问题，通常采用的办法是使用乐观锁。也就是只要当前知道的数据是最新的，那么更新该条数据就应该不会存在副作用。

乐观锁虽然比悲观锁的性能开销虽然要少，但终究还是锁，特别当锁失败，重新获取锁的代价特别高的情况，乐观锁比悲观锁的性能好不到哪里去。

下面是我的一些单机实测结果。

实验项目：

把一个全局静态变量0变成有规律的数字666666，100条线程分别设置各个位数上面的6，方法为直接在原数之上加上 6 \* (10 ^ n) ，n为位数。为了体现效果，故意在执行的过程中增加了100ms的延迟。

使用悲观锁（对象锁）耗时大概为715ms，使用乐观锁（CAS）耗时大概为682ms

而不使用锁的情况下，耗时大概为179ms。可以明显看得出一旦对线程加锁，所带来的性能开销是巨大的。

下面我将介绍一种新算法，这种算法不仅不需要用锁，还能在发生冲突的时候，检测出冲突，并自动进行消解，从而保证最终的数据一致性。使用该算法运行上面的例子，不仅能得到与加锁的情况下完全相同的结果，而且实际耗时为186ms，几乎与不加锁状态一样。

# 2、基于版本树的数据归并算法

算法的灵感来源于区块链和版本管理工具。算法实现的关键，基于下面2个假设：

1、数据的添加是原子性的

2、大部分数据冲突都是可以消解的，而且一部分数据的更新失败不影响最终的数据一致性

基本点1相对来说，比较容易得到满足，对于计算机而言，只要在内存或硬盘中开辟一个独立的数据存储区域，就可以实现新增数据的原子性写入。基本点2是算法成败的关键，大多数应用场景都可以满足

下面将具体介绍算法实现的原理及核心概念。

## 2-1 建立版本树

首先，算法会尝试建立一棵版本树，在这棵共享的版本树上，分布着同一份共享数据的各个不同副本，各个线程就是围绕这棵共享树上的各个结点来进行操作。下面以上面的例子来说明版本树的构建过程。

为了让每个数据副本能进行有效的溯源，以备进行最终的冲突消解，每个数据副本都有属于自己的唯一版本号（类似于git的做法），所有基于该数据副本所进行的更新操作，都将会以版本链的形式，串连起来。

这样做的好处是，尽管有些线程拿到的并不是最新的数据，并在旧的数据上进行了操作，也不会影响其它线程在自己的链路上更新自己的数据。

所有的线程都维护着一条自己的版本链，并在自己的版本链上不停地进行更新操作。这样的结果，就是会生成一棵包含各个线程版本的版本树。最终树上的所有叶子节点，就是每条线程对于共享数据最后的更新状态。

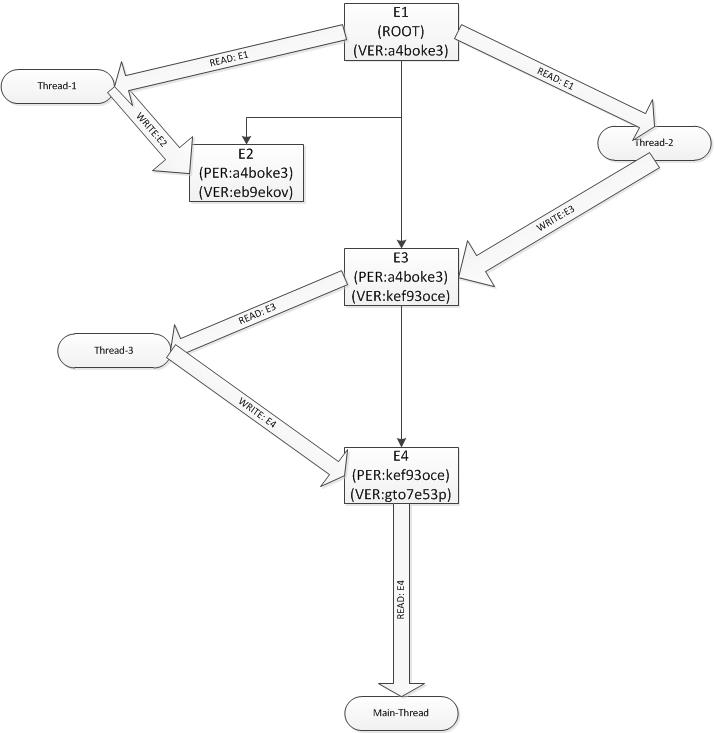


图1 版本树建立

## 2-2 归并结果

归并结果的动作，是所有并发线程都成功结束运行后，才进行的过程。这个过程就是实现冲突消解的核心。首先，我们假定所有线程并不一定全都是冲突的。而且即便部分线程产生了数据冲突，也不会影响大局。

归并版本树，首要的动作是找到主链。这里借用了区块链的概念。所谓的主链就是长度最长的链，长度最长证明该计算结果已经合并了多条线程的并发执行结果，理应是最理想的结果。因此，我们选择取用主链的最后一条数据作为数据合并的基准点。

数据合并的过程就是把版本树中的所有的叶结点进行冲突消解。一如前文所述，版本树的所有叶子结点实际上是各条线程结合自己当前所知的版本数据所作出的最终修改。因此，算法最后只需要合并整个版本树中的所有叶子结点就可以了。关于这部分的数学原理将在后面详述。

合并叶子结点的过程并不是简单的更新主链上最后一条数据，而是把在原来的主链的基础上，继续生长出一连串的合并结点，这样做的目的是为了维持主链最长。为了不影响版本树的正常生长，归并过程必须是同步的。也就是说，归并过程应该由一个全局唯一的线程来统一进行。

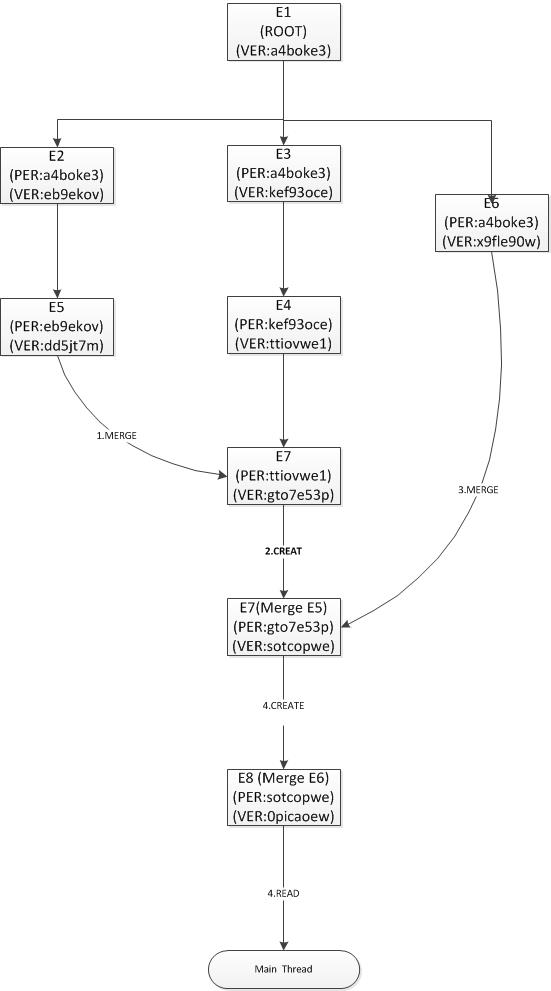


图2 版本树最终归并过程

## 2-3 算法的最终一致性

与区块链类似，每个线程（类似区块链中的客户端）当需要使用到同步数据时，都必须从版本树中拉取最新的数据，虽然不能保证每次拉取到的数据都是“最新”的，但从概率上来看，因为整棵树会一直会往最长的主链方向生长，所以最终大家看到的数据版本会是一模一样的。

## 2-4 算法背后的数学原理及相关证明

算法的核心思想就是“求同存异”。与其它同步算法不一样，新算法在运行的过程中不要求数据达到强一致性，只是希望数据在产生不一致的情况下，能在最后的合并步骤中得到修复，从而达到最终一致性。但是，算法能否保证数据一定能得到正确的修复呢？

下面将通过严谨的数学工具来证明。

为了更好地描述问题，下面将使用数学语言来定义一些必要的概念。

**定义1 : 递归函数F对于数据X来说，是顺序无关的，是指对于递归函数来说，F<N>(X) = G(X,N)。F<N>(X)表示对X执行了N次递归之后的函数值，而G(X,N)则表示一个以X和N为自变量的非递归函数。**

该定义用通俗的话来说，就是对于一个递归函数而言，如果对数据X执行N步递归计算，所得到的结果，可以用一个以X,N为变元的二元的非递归函数来表示，则这样的递归函数是顺序无关的。

举例，F(n + 1) = F(n) + C对任意整数q来说，(C代表任意函数)都是顺序无关的。

证明 :使用数学归纳法。

F(1) = q + C，F(2) = F(F(1)) = (q + C) + C = q + 2C ,

从上面的推导中，隐约可以推测G(q, N) = q + N \* C。

下面来证明F<N> (X) = G(X,N) ：

N = 1,2的时候已经得到证明，现在假设G(q, k )= q + k \* C，现在要推导出G(q, k + 1) = q + (k + 1) \* C。

由F<k+1>(X)=F<k>(D)=F(G(q,k+1))=(q + k \* C) + C = q + (k + 1) \* C = G(q,k+1)

推测成立。所以G(X , N)= q + N \* C。

至此F(n + 1)满足定义条件，所以是顺序无关的。

一般来说，一元一次线性递归函数F(n + 1)= a \* F(n) + C 都是顺序无关的。读者可以自行以上述的方法来证明，并能求出G(X,N)的表达式为G(X,N)= an \* x ，这里不再赘述。

**定义2: 若递归函数是顺序无关的，且满足F<a+b>(X) = M (X,F<a>(X),F<b>(X))，则称函数是可归并的。**

上例中F的就是可归并。

证明 : 由F是顺序无关的得F<a+b> (X)=G(X,a+b) = x + (a + b ) \* C = (x + a \* C) + (x + b \* C) – x = G(X,a) + G(X,b) – x = F<a>(X) + F<b>(X) – x。

即得F<a+b>(X) = M (X,F<a>(X),F<b>(X))= (x + a \* C) + (x + b \* C) – x

所以F是可归并。

可以证明前面一元一次线性递归函数也是可归并的，但归并函数M比较复杂，读者可自行推导。

下面开始来证明版本树归并无锁算法（下简称VT(VersionTree)算法或VT）对于可归并的递归函数是一个能达到数据最终一致性的并发算法。

要证明算法的有效性，需要先证明下述引理。

**引理1：若递归函数F是可归并的，则对于任一个F<N>(X)都可以分拆成F<N>(X)=M(X, F<N1>(X), F<N2>(X)… F<Nk>(X))其中N = 。**

证明：使用数学归纳法。

由前面的定义知，F<N>(X)=M1(X,F<N1>(X),F<N-N1>(X))。

又由于F<N-N1>(X)=M2(X, F<N2>(X),F<N-N1-N2>(X))，F<N-N1>(X)的值只与X, F<N2>(X),F<N-N1-N2>(X)有关。

所以可以重新定义F<N>(X)=M’1(X,F<N1>(X), F<N2>(X),F<N-N1-N2>(X))

现假设经过层层迭推，已有F<N>(X)=Mk(X,F<N1>(X), F<N2>(X)…,F<Nk>(X),F<N’>(X))，其中N’ > Nk+1

这时，继续拆解F<N’>(X)=Mk+1(X,F <Nk+1>(X), F<N’-Nk+1>(X))

以前面的情况类似，F<N’>(X)只与X，F <Nk+1>(X), F<N’-Nk+1>(X)有关。

所以F<N’>(X)=M’ k+1(X,F<N1>(X), F<N2>(X)…,F<Nk+1>(X),F<N’>(X))

定理得证。

上面的引理说明了，可归并的递归函数可以通过函数归并的办法，把递归过程中的产生的中间结果归并到一起。

下面介绍另一个与离散数学相关的引理2。（在离散数学相关书中应该会有介绍）

引理2：对于一棵n叉树而言，设每个节点的深度为d，则每个叶节点的深度之和S必定大于等于树的所有节点之和N。

证明：使用反证法。

假设S < N，则必然存在一个非叶结点N并不在树的所有叶子结点所经过的路径上。

那么必然在树中存在一条环路，使某个经过N的下级结点N’的叶子结点L可以在不经过N的情况下，可以经过它的下级结点N’。这与树的结构定义冲突。

现又假设下级结点N’也不被任何叶子结点经过，则N’要么是叶子结点，要么在N’之下还有一条环路使所有的叶子结点，可以在不经过N’的情况下绕到N’的下级结点。但是这两个假设者均与题设矛盾。

相似的推理可以迭推进行，最终必然导致矛盾的产生。所以引理2成立。

**VT算法数据一致性定理:如果存在一个可归并递归函数F，则通过VT算法进行并行化执行后，必然能得到合法的执行结果F<s> ∈ S , S 为递归函数F的有效值域。**

证明：如果把算法运行初的共享数据看作是变量X，而每条线程执行的动作看作是求解变量函数F(\*)的值，这里的\*表示线程在执行初期看到的数据副本，这时的\*可能是X本身，也可能是经过几代“先辈”运算过的F<M>(X)，则整个版本树上挂着的k个叶子结点，就是经过若干步运算得到F<N1>(X), F<N2>(X) ,…F<Nk>(X)。

由引理2知N1 + N2 + … Nk > N

又因为F是一个可归并的递归函数，由引理1 知必然有F<S> = M(X, F<N1>(X), F<N2>(X) ,F<N3>(X) ,F<N4>(X)…F<N4>(X))，其中S = N1 + N2 + … Nk，即S >= N。其中M是VT算法最后执行的归并算法。证毕。

如果VT算法的期望输出结果是F<N>，这个时候，可以采取剪枝的办法使S=N，也就是放弃一些计算结果，虽然并不一定保证S=N，但至少起码算法输出的结果，在递归函数F的期望序列之内。

**如果不强制要求S=N，VT算法甚至可以得到比强一致性加锁算法更好的结果。**这是因为强一致性加锁算法抱持着的是一个悲观的态度，就是一旦数据出现不一致了，所做的修改都是不可取的。

实际上，上面的理论证明在一定条件下，**这种悲观的态度是错的！！！**即使数据出现了不一致，只要算法在可控的范围内进行，并且能在后面进行有效的修正，最终结果依然还会是正确的！！！而且，通过有效的容错，线程之间还能形成一种有效的协作，充分利用算法运行过程中的中间结果。

以上就是VT算法的全部魔法。

## 2-5 算法的效率

正如前文所述，当并发任务是可归并的，而且任务竞争资源的代价较高时，VT算法有明显的优势。下面的定理说明了这个问题

**定理2: 当MAX(O(M), O(F**<m> **))<< O(F**<N>**)时，N= ,m = MAX(n1,n2….nk) , 其中O代表函数的时间复杂度，M是归并算法，VT算法优于有锁算法。**

证明：

VT算法的时间损耗主要在两个部分，线程并发执行时间和归并算法执行时间O(M)，

线程并发执行期间，根据木桶原理，这部分时间大于等于最慢的线程的执行时间，因为m=MAX(n1,n2….nk) , 所以T = F<m> ，所以算法的总时间复杂度为O(VT) = O(M) + O(T)，若O(F<N>) >> MAX(O(M) + O(T)) ，则O(F<N>) >> 2MAX(O(M) + O(T))>O(VT) ,所以VT算法的时间复杂度远少于有锁算法。

举例：若 k = n，M的时间复杂度为C1,F的时间复杂度为C2，且0<<C1<<C2,则VT算法最优.这是因为O(C) = k – 1 \* C1 + C2, O(F<N>)=n \* C2。O(F<N>) – O(C) ≈ (n – 1) \* C2 >> 0 。

一般并发执行任务，都会要求任务是无状态的，所以几乎不需要做任何归并操作，C1 ≈ 0，所以O(C)的速度只取决于**F**<m>的速度，理论上m越小，**F**<m>越快，并发运行的速度越快。但像前面的提到的递归函数，是有状态的。对于有状态的数据操作，就不得不考虑归并操作所带来的损耗。像前面的的例子，做一次数据归并操作的时间需要做两次加减法操作，而原来的递归操作只做了一次加法操作，所以当k = n时，O(C) = (n – 1) \* 2 + 1 = 2 \* n – 1 ，结果比原来的F<N> = n还要多。

**由此看出设计出好的归并函数，是整个算法成败的关键。**例如当k < <n ，而且S=N时，归并函数可以设计成M=SUM(**F**<m>) – (NL - 1) \* X，其中NL为叶子结点计数，**F**<m>为各线程的计算结果。这样，整个时间损耗只有k次加和，一次乘法和一次减法操作。因为k << n，所以程序的性能得到大幅提升。

# 3 小结

与业界常用的独占锁（悲观锁），乐观锁相比，本文介绍了一个真正的无锁算法，并通过真实的实验数据，验证了无锁算法的性能。同时，本文也通过严密的数学论证，证明了算法的数据最终一致性，并论证了在什么条件下，基于版本树的无锁算法可以实现比串行加锁算法更高的性能。

事实上，在实际的工程应用中，很多并发任务都是满足VT算法运行的前置条件（可归并性）。过去由于在任务中往往会使用大量的共享资源，以致于很多并发任务会因为争抢资源，损耗宝贵的CPU资源，并最终导致整体性能下降。本文提出的VT算法，将有效解决这一点，因为这是一种不争抢资源的前提下，共用资源的一种有效手段。

而且VT算法还可以借助第三方高速存储中间件（如Redis,MQ消息队列）等，实现分布式服务之间的无锁高并发，而这恰恰是过去针对单机的锁占用算法所不具备的能力。