软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2016,27(4):1009−1025 [doi: 10.13328/j.cnki.jos.005021]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于共享内存的多核时代数据结构研究*

周 维[1], 周可人[1], 栾钟治[2], 姚绍文[1], 钱德沛[2]

[1](云南大学 软件学院,云南 昆明 650091)

[2](北京航空航天大学 计算机学院,北京 100191)

通信作者: 周维, E-mail: zwei@ynu.edu.cn

摘 要: 随着计算机硬件技术的发展,如今我们已经迈入了多核 CPU 时代.然而,作为软件核心的数据结构仍然是按照单核 CPU 和顺序型准则来设计的.在基于共享内存的多核时代,大量并发运行的线程会交替地修改数据,产生不可预期的结果,因而我们面临着严峻挑战.针对基于共享内存多核时代数据结构的相关研究进行综述.首先,对比了并发与并行的区别,归纳了基于演进条件(progress condition)的多核数据结构分类,对近年来学术界对各种类型并发数据结构的研究进行综述.在此基础上,剖析了并发数据结构设计和实现的关键技术,并从并发数据结构的开发流程、正确性验证等方面进行了归纳阐述.最后,基于这些讨论,对多核架构下并发数据结构未来的研究趋势和应用前景进行了展望.

关键词: 多核处理器;并发数据结构;演进条件

中图法分类号: TP311

中文引用格式: 周维,周可人,栾钟治,姚绍文,钱德沛.基于共享内存的多核时代数据结构研究.软件学报,2016,27(4): 1009−1025. http://www.jos.org.cn/1000-9825/5021.htm

英文引用格式: Zhou W, Zhou KR, Luan ZZ, Yao SW, Qian DP. Study on multi-core data structure in shared-memory. Ruan Jian Xue Bao/Journal of Software, 2016,27(4):1009−1025 (in Chinese). http://www.jos.org.cn/1000-9825/5021.htm

Study on Multi-Core Data Structure in Shared-Memory

ZHOU Wei[1], ZHOU Ke-Ren[1], LUAN Zhong-Zhi[2], YAO Shao-Wen[2], QIAN De-Pei[2]

[1](National Pilot School of Software, Yunnan University, Kunming 650091, China)

[2](School of Computer Science and Engineering, BeiHang University, Beijing 100191, China)

Abstract: The development of computer hardware technology has led to an era of multi-core CPU. However, data structures, as the core of the software, are traditionally designed in line with single-core CPU and ordered sequence principle. Operating on the shared-memory multicore, a large number of concurrent running threads alternately modify the data structure, which brings big challenges. This paper surveys researches on multi-core data structure in shared-memory. First, the paper compares the differences between the concurrent and parallel data structures, and investigates the multicore structure classification characteristics based on progress condition. Then it reviews academic research on various types of concurrent data structures in recent years. Based on the findings, this paper summarizes the key technologies of concurrent data structure, and explains the design and development process as well as correctness verification of concurrent data structures. Finally, it discusses research prospects.

Key words: multi-core CPU; concurrent data structures; progress condition

随着计算机技术的发展,研发人员意识到通过不断增加主频来提升 CPU 性能的时代已经结束,近年来,CPU 架构更加注重低功耗和多核心.芯片设计工程师将两个或多个内核封装到单一处理器中,片上多核处理器已经

* 基金项目: 国家自然科学基金(61133004, 61363021, 61540061)
Foundation item: National Natural Science Foundation of China (61133004, 61363021, 61540061)
收稿时间: 2014-09-25; 修改时间: 2015-10-08; 采用时间: 2015-12-22; jos 在线出版时间: 2016-01-05
CNKI 网络优先出版: 2016-01-06 15:15:57, http://www.cnki.net/kcms/detail/11.2560.TP.20160106.1515.002.html

© 中国科学院软件研究所 http://www.jos.org.cn

成为处理器发展的趋势.多核带来的性能提升及其在商业上的成功使得多核心架构从最初的 2 核、4 核,跨越到 8 核、16 核,不久的将来,普通 PC 机或许将迎来超过 100 核的处理器.那么,这波多核运动要发展到什么程度呢?没有人能给出一个明确的答案.可以肯定的是,我们已经迈入了一个多核时代,这已成为计算机设计和应用中不可回避的问题[1].

多核时代给人们带来了机遇,同时也带来了巨大的挑战.在一个单独封装的处理器内拥有多个内核,它们共享同一块内存,并发执行多个线程时通过共享内存中的数据结构来进行通信和同步[2].随着内核数量的增多,程序对共享资源访问的冲突也在加剧,这成为一个亟待解决的问题[3,4].虽然在过去的几十年中,研究者们已经在多核处理器的机器上运行软件程序,然而这些软件程序通常是数值计算、大规模矩阵运算等科学研究性质的并行程序,或者是未针对多核处理器优化的顺序型程序.伴随着商用多核(multicore)系统成本的降低,未来个人电脑、平板电脑、手机等设备势必将具备更多核心,我们越来越需要对原有的顺序型程序进行改造和优化,使其能够更充分地利用多核资源[5,6].目前面临的瓶颈是多核软件的发展没有跟上硬件的发展.

数据结构是软件设计的核心,对整个软件的性能有着重要影响.往往在数据结构上的一些细微改动就能够很大程度地提升整个软件的运行效率.然而,目前大部分程序员、科研工作者对多核时代数据结构并发编程的理解还不够深刻,只有少部分人具备设计并发程序的能力[7].对于大多数人而言,设计多核并发数据结构的难点在于:在一个进程内大量并发运行的线程会交错地执行指令,从而产生不可预期的结果.这就要求我们对多核时代的数据结构有新的认识,用新的方法和工具集来开展研究和设计[8].

总体而言,多核架构下基于共享内存的并发数据结构仍然有很多问题亟待解决.本文针对多核时代数据结构的相关研究进行综述.剖析了国内外各类并发数据结构的研究现状,总结了并发数据结构设计和实现的关键技术.最后,基于这些讨论,对多核架构下并发数据结构未来的研究趋势和应用前景进行了展望.

本文在第 1 节分析数据结构并发的重要性及其与并行的区别.第 2 节归纳提出基于演进条件(progress condition)的并发数据结构分类.第 3 节对近年来学术界对各类并发数据结构(如:堆栈、链表、哈希、树型结构等)的研究现状进行综述.在此基础上,在第 4 节~第 6 节总结并发数据结构的关键技术、开发流程和理论正确性验证方法.最后在第 7 节对多核架构下并发数据结构未来的研究趋势和应用前景进行展望.

1 数据结构并发的重要性及其与并行的区别

1.1 并发与并行的区别

多核系统在一段时间片段内,运行的线程会交错地执行指令.因此,并发(concurrency)和并行(parallelism)是多核编程中经常会碰到的概念,也容易混淆,首先需要区分并发与并行.

并发:用来描述存在至少两个线程正在工作(并不一定在同时执行)的状态,是一种更加通用的、可以在单核处理器上通过时间切片实现的状态.

并行:用来描述至少两个线程同时执行的状态.在同一时间点上,多个线程可同时并行执行.

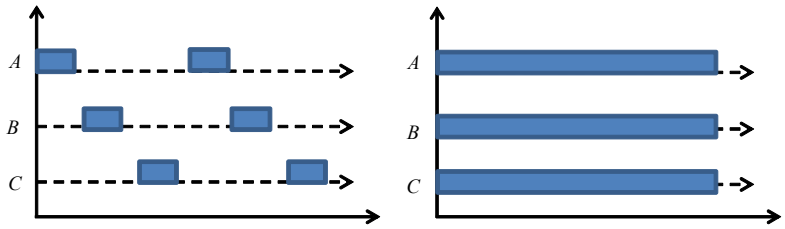
如图 1(a)所示,并发从逻辑上看是多个线程同时发生,但在物理上则是由操作系统调度完成.CPU 某一时刻依然只执行某个线程的任务.所有的并发处理都有排队等候、唤醒、执行至少 3 个这样的步骤.当多个线程同时工作时,从宏观上看它们是并发的,从微观上看它们却又是顺序被处理的,只不过资源不会被阻塞(一般是通过时间片轮转).而并行则是物理上多个线程同时发生.如图 1(b)所示,同一个时刻,有 3 个线程同时运行,无论从宏观还是微观来看,3 个线程都是同时工作的,也即是并行执行的.

通常而言,并发编程会涉及到较多系统资源竞争,比如同时读写共享内容、操作系统层面的线程调度等.而并行编程则较少出现严重的资源竞争现象.如 GPU 平台的编程,同一时刻可以存在大量同时执行的线程,因而带来高吞吐量.但是,由于 GPU 架构中计算能力较强,逻辑控制单元较弱,GPU 不适合开发逻辑较为复杂的并发程序.

在多核出现前的单核处理器时代,经过几十年的发展,人们已经在编译器优化、处理器优化、多线程编程等方面有深入的研究,开发了对编程人员而言非常抽象的各种基础库,编程人员无需知道底层硬件设备、编译

© 中国科学院软件研究所 http://www.jos.org.cn

器、OS 调度等实现细节.进入多核时代后,由于计算机 CPU 架构的变化,相应的 OS 调度、编程模型等的调整才刚刚开始,一个高度抽象的基础库还没有形成,因此,多核时代的并发编程更加困难.



(a) 并发,单核,逻辑上同时 (b) 并行,多核,物理上同时

Fig.1 The difference between concurrency and parallelism

图 1 并发和并行的区别

1.2 数据结构并发的重要性

为了更深刻地理解数据结构并发的重要性,首先让我们来看看著名的阿姆达尔定律(Amdahl's law)[9].设 n 为可并发执行线程数,p 为可并发执行部分所占整个程序的比例,那么最后可以得到这个程序最优情况下能够达到的加速比 S.

$$S=\frac{1}{1-p+\frac{p}{n}}.$$

通过上述公式可以推出 S 并不是和 n 线性相关的.举一个直观的例子:假设目前可并发执行的线程数为 10,那么理想情况下可以达到的加速比是 10.然而,根据上面的阿姆达尔定律,当并发执行的程序比重占到 90%时,最后只能得到 5.2 倍的性能提升,仅为理想值的一半左右.因此,阿姆达尔定律揭示我们,对一个程序进行并发改造的空间是有限的,主要的影响因素是 CPU 核心数目和程序必须顺序执行的部分占整个程序的比例.在上述例子中,剩下不可并发执行的 10%部分对于整个程序并行效率的提升相当重要,而这部分在程序中通常体现为需要内部调整、具有特殊性质的数据结构.因此,只有提升了这些关键数据结构的并发能力,才有可能给整个应用带来很大的性能提升.

2 基于演进条件(progress condition)的多核数据结构分类

演进条件是指并发数据结构活性能够被保障的条件.它是用来评价并发数据结构的重要指标,意味着并发数据结构算法最终在怎样的情况下完成[5].演进条件在一定程度上反映了并发数据结构的性能.根据演进条件的差异性可以将其主要分为 4 个大类别:阻塞(blocking)、无干扰(obstruction-free)、无锁(lock-free,简称 LF)和无等待(wait-free,简称 WF),这 4 个类别又可以被细分为如图 2 所示的 7 个不同小类.分析已有的演进条件,归纳它们之间的定义及关系如下.

- 阻塞:即这种方法在其执行过程中不能正常运行,直到其他(占有锁的)线程释放.阻塞是大家所熟知的,基本上,所有加锁的算法都可以说是阻塞的.使用阻塞算法时,某个线程所引起的意外延迟会阻止其他线程继续运行.在极端情况下,会造成死锁;
- 无饥饿(starvation-free):那些希望进入互斥区域的线程最终都能够进入互斥区域(即使之前在互斥区中的线程意外停止了).无饥饿有的时候也被称为无闭锁.
- 无干扰:如果一种方法满足无干扰性质,那么这种方法从任意一点开始它的执行都是隔离的.当一个线程调用该无干扰方法时,其他任何线程调用的各种方法都不会对这种无干扰方法的运行结果造成实质性影响,但是该方法的运行时间可能由于其他线程占用资源而受到影响,但是绝对不会出现死锁.
- 无锁:如果一种方法是无锁的,那么它保证有一些调用能够在有限步内完成.
- 无等待:假如一种方法是无等待的,那么无论其他操作如何,这种方法的每一次调用都可以在有限的步骤

© 中国科学院软件研究所 http://www.jos.org.cn

内结束.无等待是一种非阻塞的演进条件,意味着一个线程的任意意外延迟(比如说一个线程持有锁)都不会阻塞其他线程的继续执行.

- 有界无等待(wait-free bounded,简称 WFB):如果一种方法是有界无等待的,那么这种方法保证每次调用都能够在有限并且有界的步骤内完成.这个界限可能依赖于线程的数量.
- 集居数无关无等待(wait-free population oblivious,简称 WFPO):一种无等待的方法,如果其性能和活动线程数目无关,那么被称为集居数无关无等待的.

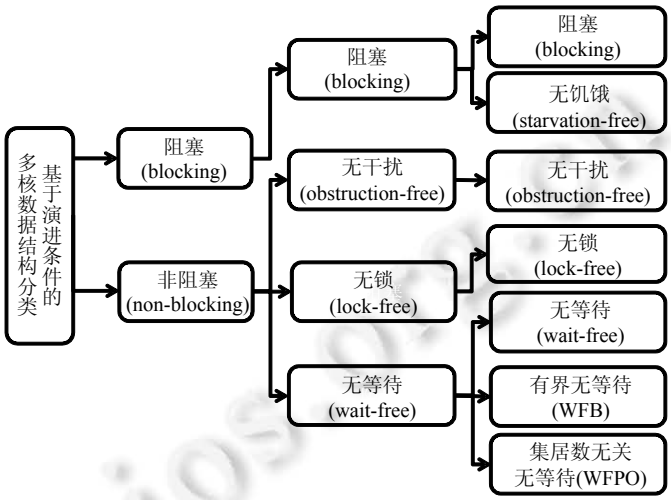


Fig.2 Multi-Core data structure categories based on progress condition

图 2 基于演进条件的多核数据结构分类

多核时代的数据结构基本上都可以按照上述 7 类演进条件来进行划分,从演进条件的活性保障程度来说,存在如下关系:集居数无关无等待>有界无等待>无等待>无锁>无干扰>无饥饿>阻塞.演进程度越高,意味着线程的活性能够得到更多的保障,多线程在执行时受到的干扰和约束越少,因此执行的效率更高.演进程度高的涵盖演进程度低的.对于阻塞和非阻塞两大类来说,若一个数据结构满足演进程度高的条件,那么它也一定满足演进程度低的演进保证.举个例子:我们可以说“无等待意味着无锁”(但是不能反过来说).这意味着一种方法如果满足无等待,那么这种方法就有着与一种无锁方法同样的演进保证.非阻塞类的演进条件涵盖关系可以用维恩图的形式来表达.

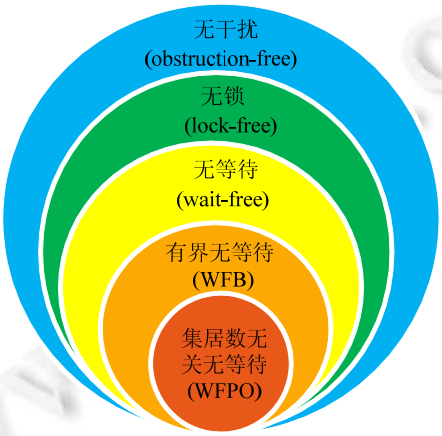


Fig.3 The Venn diagram of non-blocking progress condition

图 3 非阻塞类演进条件的维恩图表示

© 中国科学院软件研究所 http://www.jos.org.cn

在设计并实现并发数据结构时,其能够达到的演进条件取决于应用的需求和底层软硬件平台的支持.一般来讲,演进程度高会具有比演进程度低更好的效率.

3 各类数据结构并发研究现状

近 30 年来,科研人员对各种数据结构并发操作的研究步伐一直没有停歇过,研究的重点也逐步从最初简单的数据结构(如:Stack,Queue,Hash-Table,Skiplist 等)转向复杂树形结构.在这一节中,我们将对这些主要数据结构的研究情况进行分析.

3.1 并发堆栈

堆栈(stack)是较为简单的数据结构.在前人深入研究的基础上,Michael 和 Scott 在文献[10]中总结了多个基于锁的并发堆栈实现,其基本原理是利用全局锁对堆栈访问进行控制,然而,这种设计使得堆栈的栈顶成为并发的瓶颈.Treiber 在文献[11]中首次提出了演进条件达到无锁级别的并发堆栈,它采用单链表来表示堆栈,使用 CAS(compare-and-swap)原子操作来修改堆栈的栈顶.实验结果表明,相对于基于锁的实现,这种无锁实现在性能上具有很大优势.

如图 4 所示,当多个线程(线程 *A* 和线程 *B*)进行栈顶访问发生资源冲突时,使用 CAS 原子操作来修改堆栈的栈顶.同一个时刻只有一个线程成功(线程 *A*),另外一个线程 *B* 失败后则进入下一轮 CAS 竞争.这样在对资源的竞争中避免使用锁,取得了比较高的并发性.除此之外,文献[12,13]中提出一种称为“消除(elimination)”的技术来提高堆栈的并发性,其核心思想是允许成对的彼此相反的操作在不经过中央调度的情况下立即完成.因此,当一个 pop 操作遇到一个并发执行的 push 操作时,允许 pop 操作立即获得 push 操作的值,并且这两个操作都立即返回.

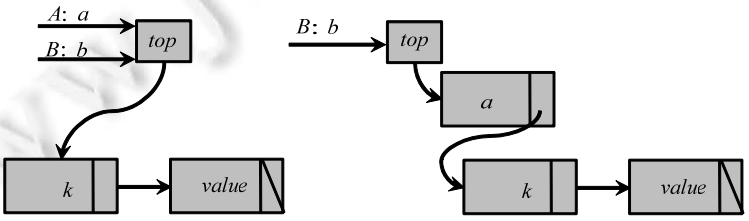


Fig.4 The concurrent stack implementation

图 4 并发堆栈设计

3.2 并发队列

最早的无锁并发队列出现在文献[14]中,其核心思想是通过一个有限长度的数组来实现无锁功能,两个独立的线程分别在队列的入口和出口负责读操作和写操作.在此基础上,文献[15]提出了一个基于链表的无锁队列.文献[16]提出一个数组容量无限、基于无锁数组的队列.文献[10]对无阻塞队列的实现作了详细总结,并提出一种基于 CAS 原子操作且演进条件为无锁的队列,通过 CAS 来对队列的头尾进行多线程的资源互斥和并发控制.上述工作[10,14−16]都基于单端队列,双端队列的并发设计则比普通队列难很多,因此目前实现的大多是基于锁的阻塞双端队列.文献[17]指出,即便使用 CAS,也很难设计出无锁的双端队列.目前已知的实现是 Herlihy 在文献[18]中提出一种基于 CAS、满足演进条件为无干扰的双端队列.

3.3 并发链表

最早的并发链表是一种基于全局锁的实现,每一个获得全局锁的线程进入链表进行操作,其他线程则在外等候,因而全局锁的粒度较粗.一种称为“手连手锁(hand-over-hand locking)”的细粒度锁在文献[19,20]中被提出,在这种方法中,链表中的每个节点都有一个锁的标志位,线程在遍历链表时,只有当获得下一个节点的锁之后才会释放当前节点的锁.相对于全局锁,手连手锁的特点是缩小了锁影响的节点范围,一个链表中可以同时存在多个线程进行操作,提高了链表的并发性.但是,在相互临近节点的插入和删除操作还是会相互干扰,影响了结构

© 中国科学院软件研究所 http://www.jos.org.cn

的并发性.Valois[21]首次提出了基于 CAS 的无锁链表,但是 Valois 的设计过于复杂而难以实现.在此基础上,Harris[22]提出了一种更易于实现、演进条件为无锁的链表,通过为节点设置一个删除标志位,只允许用原子操作来修改这个标志位.在这种实现中,节点被设置成删除后并不是马上就物理删除,而是先存在于链表中,等待内存垃圾回收时再统一删除.

3.4 并发哈希表

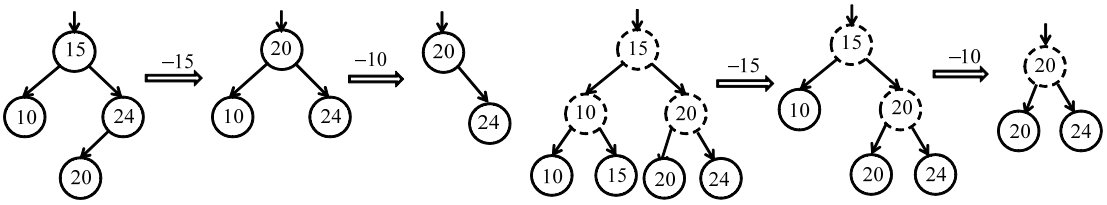
通过为哈希表的每个存储单位分配一个读/写锁,固定长度哈希表的并发操作相对容易实现[23].然而,随着插入元素的增多,只有可扩展的哈希表才能保障良好的性能,但设计并发的可扩展哈希表难度较大.文献[24–26]提出了一种适用于分布式数据库的、基于两层锁的可扩展哈希表.在此基础上,Lea 在文献[27]中提出了可扩展哈希表的算法,在测试中表现出了良好的性能.该算法不对每一个存储单元加锁,而是建立少量更高层的锁.在哈希表进行调整时允许进行并发查找,但是不允许并发的插入和删除操作.上述工作[24–27]都是基于锁的可扩展并发哈希表来实现,这种方法不可避免地存在阻塞同步的缺陷,并且当哈希表的容量进行扩充时需要重新调整元素的分布,这使得阻塞情况更加严峻.Shalev 和 Shavit[28]首次提出一种演进条件为无锁的可扩展哈希表,其核心思想是将哈希表中的每个元素放到一个无锁的单向链表中.由于单向链表本身是无锁实现的,因此对其中每个元素(也就是哈希表中的每个元素)的操作(插入、删除)也就是无锁的.单向无锁链表的引入使得对元素的查询具有 $O(n)$的时间复杂度,这有悖于哈希表中 $O(1)$的属性.为了解决这个问题,Shalev 和 Shavit 在算法中维护了一个可变长的指示数组,数组中存储了指向相应单向链表节点的指针.通过这种方式,使得查询操作不再是顺序查找的过程,而是能够通过数组中的指针快速定位到相应的链表节点的过程.

并发技术也被引入一些新近提出的 Hash 算法中以提高效率.Cuckoo hashing 是 Pagh 等人[29]于 2001 年提出的一种解决 hash 冲突的方法,其基本思想是使用两个 hash 函数来处理碰撞,通过多步探测,以增加查找与插入时间为代价减小 hash 碰撞,提高了 hash table 的利用率.原始的 Cuckoo hashing 并不支持并发读写,Fan 等人[30]于 2013 年提出 Concurrent Cuckoo hashing,该技术采用版本记数器和乐观并发(optimistic)思想,支持多个 reader 和一个 writer 的并发访问.并在 memcache 系统改进中取得了很好的测试效果.Hopscotch hashing 是 Herlihy 等人[31]在 2008 年提出的一种开放寻址法 hash table,它结合了 linear 探测、链式 hash 和 Cuckoo hash 的优点.Hopscotch hash table 的每一项都包含一个 hop-information(即一个 H-bit 的 bitmap),它描述了临近 H-bit 位中元素的占用情况.Hopscotch hashing 设计之初就考虑到了并发,为该结构占用的每一个内存 Bucket 分配一个 lock,从而保证了并发的安全性.

3.5 并发树形结构

传统的并发数据结构(如 Linked-list,Stack,Queue,Hash-Table,Skiplist 等),由于结构相对简单且研究时间较长,都已经做到无锁实现.对于更高程度的无等待并发,由于研究难度较大,目前进展缓慢.因此,近年来科研工作者将研究重心转向并发树型结构的研究[32–35],其中“二叉树”最具有代表性.Fraser 提出了一种使用 MCAS(multi-word compare-and-swap)原语实现的并发二叉树结构[36],由于 MCAS 原语并非在所有硬件平台上都予以支持,因而该结构不具有实用性.文献[37]中提出了一种使用 STM(software transactional memory)实现的并发红黑树结构,虽然这种实现非常方便,不需要程序员进行额外的同步处理.然而,STM 实现普遍导致了巨大的时间开销,一个简单的细粒度锁二叉树结构,或者是非阻塞二叉树结构可以在性能上轻易地超过它.文献[38]提出了一种基于细粒度锁的并发平衡二叉树数据结构.该二叉树结构利用了乐观并发的思想,利用版本号控制技术来判断操作是否冲突:如果冲突,那么算法从根节点开始重新遍历;否则,操作完成.Ellen 等人[39]首次提出了利用 CAS 原子操作的并发二叉树结构,该结构属于“外部树”结构,即所有键值存储在叶子节点,并且删除时将父节点和叶子节点一起删除,而不对其他内部节点作任何处理.文献[40]中将待删除节点的后继节点覆盖到待删除节点,如果后继节点只有一个孩子,那么将后继节点删除,否则,保留后继节点(采取逻辑删除的思想).上面提到的并发树[38]本质上都是基于外部树,其结构如图 5 所示.

© 中国科学院软件研究所 http://www.jos.org.cn



(a) 删除内部树节点 (b) 删除外部树节点

Fig.5 The external tree diagram

图 5 外部树示意图

如图 5 所示,所有外部树的信息都存储于叶节点,中间节点起到路由作用.因此,相对于普通内部树,外部树会导致一定的额外存储空间开销,但是这样做的好处是在插入、删除时避免了树型结构的大范围调整,有利于多核多线程的并发操作.如图 5(a)所示,在内部树中删除节点 15 时,需要将底层节点 20 调整到顶部,影响范围比较大,而外部树则避免了这种情况,减小了因为节点调整影响的范围.

Natarajan 等人在文献[41]中利用 CAS 和 SETB 操作设计了一种高效的并发二叉树数据结构,大幅度提升了二叉树的性能.Drachsler 等人[42]针对二叉树中的查找操作进行优化(这里提到的查找操作指的是查询、插入和删除都需要涉及的定位节点操作),主要利用了逻辑有序(logical ordering)的思想,即存储每个节点逻辑上的前后节点,将插入和删除操作的查找行为和查询操作的查找行为分离,从而提升查找操作的效率.他们也提到了关于在提升查找操作的性能、额外的存储空间以及调整 Logical order 的额外时间之间存在一个权衡的问题.文献[43]结合了前人的技术要点,构造了一个高效的二叉树结构.该二叉树结构具有以下几个特点:(1) 采用内部树以及线索树结构,并且通过一些技巧加以优化,使得操作的复杂度从 $O(cH(n))$降低到了 $O(H(n)+c)$.(2) 仅使用 CAS 原语,因此跨平台可用.在二叉树研究的基础上,Brown 等人将研究的范围扩展到多叉树(k-ary tree),并且着重讨论了多核情况下多叉树中的范围查询问题,实验结果显示,基于 CAS 的实现在小范围查询时取得了显著效果[44,45].

对于需要平衡的树形结构来说,插入、删除操作都会带来树形结构的调整,因此,前人还针对并发二叉树的平衡性策略进行了相关研究.文献[38]在每一次互斥操作之后进行上锁调整平衡,而文献[46]则采用一个额外的线程专门进行调整操作.有的时候,可以通过弱化平衡操作的实效性来减小数据结构设计的难度,如文献[47]通过将树的平衡操作分离实现了一个平衡二叉树,其核心思想是在进行插入、删除操作时并不立即进行平衡调整,而是等到一定时间后再进行平衡调整.

目前所有已知实现的树型结构,能够达到的最高演进条件是无锁级别.

4 并发数据结构关键技术

在上一节中,我们对现有数据结构的并发研究现状进行了分析.可以看出,每一种并发数据结构的设计都具有一定的难度,需要选用特定的技术来精心设计,我们将其中一些关键技术进行分析总结如下.

4.1 通用构建(universal construction)

通用构建是对不同数据结构提供一套在多核环境下的统一构建方法.有两种不同的实现:(1) 将数据结构的所有并发操作都放入队列中来生成对应的顺序化操作[48].(2) 对数据结构建立多个副本,并发操作在不同的副本上得到执行,更新受到影响的共享结构部分,完成不同副本之间的一致性保证[49,50].由于其普适性的设计思想,通用构建方法相对于采用特殊设计技术(如:基于锁的实现)的执行效率较低,除此之外,在一些复杂结构(如树型结构)中很难采用通用构建方法.

4.2 基于锁的实现(lock-based)

基于锁的实现是目前多核并发编程中采用最为广泛的技术.根据锁的粒度的不同,可分为粗粒度锁和细粒度锁.

© 中国科学院软件研究所 http://www.jos.org.cn

粗粒度锁主要解决线程之间同步和互斥问题,在 Java 编程中体现为管程(moniter)的概念.管程是一种程序结构,结构内的多个子程序(对象或模块)形成的多个工作线程互斥访问共享资源.通过使用管程,可以针对管程对象的每一种方法自动获取锁资源,释放锁资源.与那些通过修改数据结构实现互斥访问的并发程序设计相比,管程很大程度上简化了程序设计.管程实现了在一个时间点,最多只有一个线程在执行管程的某个子程序.采用粗粒度的锁可以简化加锁行为,减少编程难度,但是粗粒度锁对并发性影响更大.在第 3.3 节中提到的全局锁就是一种粗粒度锁.

相对于粗粒度锁在线程层面实现并发,细粒度锁可以在数据结构层面实现并发.几乎所有数据结构(如:堆栈[10]、队列[17]、链表[19,20]、哈希表[24−26]、树形结构[10])等都可以采用细粒度锁来实现多核数据结构并发.这种技术在简单数据结构(例如:堆栈队列、哈希表、跳表)中实现的效果很好,然而在树形结构,特别是需要调整平衡的树形结构中效率却不高.其根本原因在于,基于细粒度锁的实现需要对树形结构中附近区域内的多个节点进行加锁操作,同时只允许一个线程进入共享区进行操作来保障共享数据的安全.对于需要平衡的树形结构来说,插入、删除操作都带来树形结构的调整,这就意味着加锁的范围会很大,降低了数据结构的并发性.Leo[47]通过将树的平衡操作分离实现了一个平衡二叉树,其核心思想是在进行插入、删除操作时并不立即进行平衡调整,而是等到一定时间后再进行平衡调整.基于这样的思想,Nurmi 放松了平衡的条件,实现了 chromatic 树[51],这是一种采用面向叶子(leaf-oriented)技术的红黑树.在此基础上,Boyar 通过修改 chromatic 树的平衡条件提高了其性能[52],并首次给出了完整的正确性证明.

手连手锁是一种特殊的细粒度锁,在这种方法中,每个节点都维护一个锁.线程在遍历时,只有当获得下一个节点的锁之后才会释放当前节点的锁.相对于全局锁,手连手锁的特点是缩小了锁影响的节点范围.文献[47]提出了一种基于手连手锁的并发平衡二叉树数据结构.在文献[53]中提出了一种基于手连手锁的红黑树结构.

综合近几年来二叉树研究的情况,在多核时代的树形数据结构设计中,基于细粒度锁的实现会对树中一定范围的节点进行锁定,这在一定程度上会对并发的更新操作产生影响.

此外,由于基于锁的技术能够有效保护共享数据,一些特殊设计的锁(如:自旋锁(spinlock)、读写锁(rwlock))在Linux操作系统中得到了广泛的使用.传统上,当发生访问资源冲突时,可以有两种选择:一种是死等,另一种是挂起当前进程,调度其他进程执行.spin lock 就是一种死等的机制,当前的执行线程会不断地重新尝试直到获取锁进入临界区.因此,自旋锁在同一时刻只能被最多一个内核任务持有,所以,一个时刻只有一个线程允许存在于临界区中.在多核时代,自旋锁总是只允许一个 CPU 核访问共享资源,其余 CPU 核均是忙等待,无法发挥多核的优势.考虑到这样的情况,Linux 中还提供了读写锁,该方法允许多个 CPU 核读者同时访问,绝对限制只有一个核可以作为写者.当读者访问资源时,写者必须忙等待,反之亦然.然而,随着计算机硬件的快速发展,获得各种锁的开销相对于 CPU 的速度在成倍地增加.在大部分非 x86 架构上获取锁使用了内存栅(memory barrier),这会导致使用锁的时候处理器流水线停滞或刷新,因此使用锁的开销相对于CPU速度而言就越来越大,因此,一种能够提供高并发性的同步机制——Read-Copy-Update(RCU)[54]即被提出,它允许 reader 和 writer 并发地访问共享数据,支持一个 writer 和多个 reader 之间的并发.通过维护多个版本的数据,RCU 保证了 reader 读取到的数据的一致性,还保证在 reader 完成读取前,被访问的数据不会被释放.

综合近几年来二叉树研究的情况,在多核时代的树形数据结构设计中,基于细粒度锁的实现会对树中一定范围的节点进行锁定,这在一定程度上会对并发的更新操作产生影响.

如图 6 所示,对于读者,在访问被 RCU 保护的共享数据时可以直接访问,不用关心这个数据是否被锁住;而对于写者,在进行更新操作时,首选需要获得该数据的一个拷贝(如图 6 中步骤 2 所示),然后对该拷贝进行更新,在没有任何读者来读取该数据时,将这个原始数据指针修改为指向该更新拷贝(如图 6 中步骤 3 所示),于是就完成了修改操作.RCU 的好处是读操作是无锁的,所以不存在同步问题,也不需要内存栅栏,对于由于锁机制导致的死锁和内存延迟等问题都有较好的改善.写操作的同步开销相对较大,因为需要等待合适的更新数据的时机,以及增加了原始数据释放、修改指针等操作,如果当前存在多个写者,依旧需要使用锁机制来同步其他写操作.RCU 在 Linux 内核 2.5.43 版本开始使用,截至 2014 年,已经使用了超过 9 000 个 RCU API.

© 中国科学院软件研究所 http://www.jos.org.cn

在 2015 年召开的 SOSP 会议上,Alexander 等人[55]提出了 RCU 的改进版 RLU(read-log-update)技术.相对于 RCU 只允许多个读、一个写机制来说,RLU 的最大优势是允许多个读、多个写同时进行,提高了并发性.

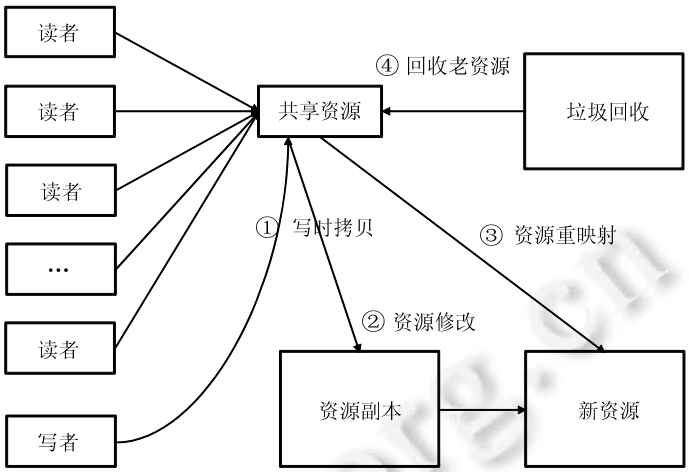


Fig.6 The RCU runing mechanism

图 6 RCU 运行机制

4.3 基于OCC实现-基于乐观并发控制的实现(optimistic concurrency control based)

乐观并发控制(optimistic concurrency control,简称 OCC)是一种并发控制的方法,首先出现在数据库系统[56]的相关工作中.其核心思想是让每个线程分离执行,使得每个线程执行时并不受到其他线程的干扰,当线程需要对数据结构作修改时,再对比前后是否保持一致性状态,如果确认状态一致,那么说明在这段时间内没有任何其他线程对数据结构做过修改,该线程提交修改,并且更新版本号;否则,线程需要回滚重试.

在多核、多线程数据结构设计中,OCC 最早于 2003 年被引入软件事务内存(software transactional memory,简称 STM),被用来设计并发树[57]和并发平衡树[58].后来,OCC 被用来对并发二叉树中查询操作进行优化[38].值得注意的是,演进条件“无干扰”本身就具有乐观并发性,即旧值和新值比较时,只有不存在冲突时才能避免重试.无干扰的演进条件屏蔽了线程操作的中间状态,而通过数据结构的前后版本对比来确定前后是否保持一致.显然,这种演进条件比无锁和无等待要弱得多,因为它并不能保证在某一时刻至少一个线程是处在演进状态的(可以把演进看作是数据结构的一种不断改变的状态).然而,这种演进条件也有一定的好处,其实现思想简单,极大地简化了设计并发数据结构的过程,程序员不用花时间去思考如何验证数据结构的正确性.乐观并发控制的思想并不一定用在非阻塞技术中,如文献[38]中提出了一种基于细粒度锁的乐观平衡二叉树,文献[59]中提出了一种易于证明的乐观并发跳表.

4.4 软件事务内存(software transactional memory)

事务性内存可分为两类:软件事务内存(STM)和硬件事务内存(HTM),硬件事务内存的概念很早于 1986 由 Herlihy[60]提出,然而直到最近的 Haswell 处理器上,才开始提供硬件事务内存接口[61].借鉴硬件事务内存的思想,1995 年,麻省理工大学的 Shavit 教授在其论文中首次提出软件事务内存的概念[62],并因为在该领域的一系列杰出贡献获得了 2012 年分布式领域著名的 Dijkstra 奖.软件事务内存的核心思想是假设多用户并发的事务在处理时不会彼此互相影响,各事务能够在不产生锁的情况下处理各自影响的那部分数据.在提交数据更新之前,每个事务会先检查在该事务读取数据后有没有其他事务又修改了该数据.如果其他事务有更新的话,正在提交的事务会进行回滚.在数据冲突较少的环境中,偶尔事务回滚的成本会低于读取数据时锁定数据的成本,因此可以获得比其他并发控制方法更高的吞吐量.

软件事物内存是并发实现技术中锁的一种替代机制,它的出现引起了研究人员和开发人员的高度关注,分

© 中国科学院软件研究所 http://www.jos.org.cn

别从编程语言、编译器、指令执行等各个软件层面进行了深入研究[57,58,63−70].目前主流语言都在不同程度上支持软件事务内存[71],其中,Clojure 将软件事务内存集成在其核心语言中.相对于其他并发实现技术,软件事物内存对于程序开发人员的好处在于,简化了并发设计中对于资源共享、冲突部分的软件设计.相对于传统意义上广泛采用的加锁技术,STM 的概念要简单得多,因为每一个事务可以理解为隔离的,就像只有单个线程进行操作一般,而死锁等特殊情况都由系统来控制完全加以避免,程序员无需担心这些问题.软件事物内存简化了无阻塞编程的难度,然而,相对于成熟的基于锁的技术,软件事物内存需要记录每次操作日志,因而带来不小的开销,在高并发情况下,不断的事物回滚减弱了多核数据结构执行的效率,扩大了事物执行的顺序开销.因此,目前软事物内存尚未成熟,远未达到实际系统中大规模采用的程度[37,72].

4.5 基于标志位和原子操作的实现(flag and atomic operations)

资源竞争是并发编程中存在的最核心问题,通常可采用互斥的方式来解决.操作系统分别从软件和硬件层面提供了支持,软件层面可以采用信号量和锁的机制,硬件层面则提供了原子操作的手段.从 CPU 指令执行的角度来看,软件层面的信号量和锁通常涉及多个 CPU 执行周期,而硬件层面的原子操作则能够由一条指令完成,因此原子操作执行速度最快,这使它成为高速并发数据结构设计的首选.原子操作具有不可分割性,在执行完毕之前不会被任何其他任务或事件中断,也不会被线程调度机制打断.在并发编程中,经常采用的原子操作包括:

(1) TAS(test-and-set);

(2) FAS(fetch-and-store);

(3) LL/SC(load-link/store-conditional);

(4) CAS(compare-and-swap);

(5) MCAS(multi-word compare-and-swap).

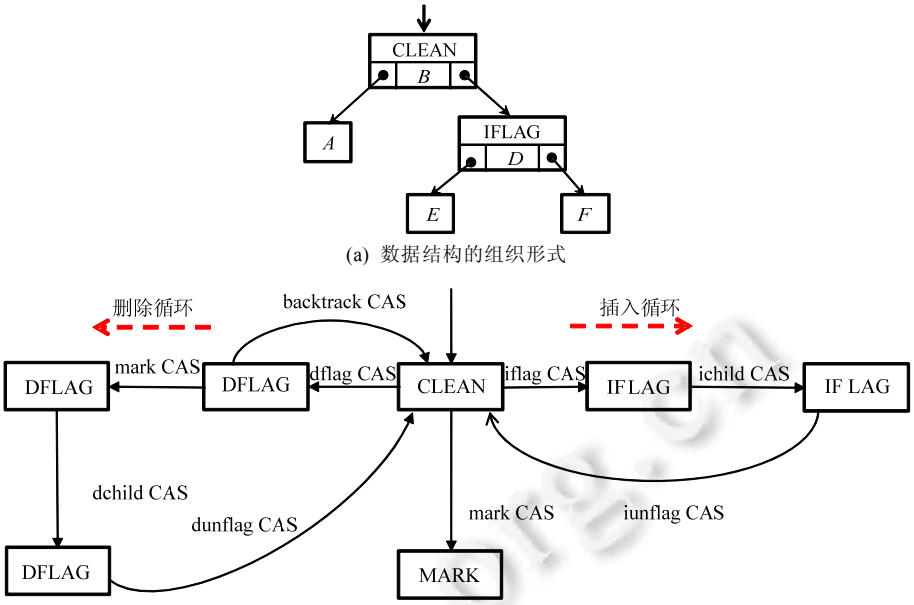
需要指出的是,MCAS 可以同时对多个内存单元的数据进行 compare-and-swap 原子操作,因此在并发数据结构设计中更具灵活性,但是 MCAS 的最大缺点是可移植性差,很多硬件并不提供支持.原子操作使得并发线程对资源竞争的相应速度达到最大化,同时也保证了对并发线程的隔离.

标志位是近年来多核数据结构非阻塞算法中最为核心的辅助技术,主要用于让线程对竞争区域的操作可见,即明确其他线程对自身线程操作区域的动作.因此,标志位的好处在于使得其他线程不必等待占有临界区的线程操作,而是能够感知临界区的当前操作,从而做出相关动作.对标志位的修改只能用原子操作来实现,这样才能做到线程之间的绝对隔离.

文献[22]中最早提出了一种用标志位和 CAS 操作实现的无锁链表,Michael[23]在前人的基础上,改善了内存管理机制,并且大量优化了并发效率.Lea 在此基础上实现了 ConcurrentSkipListMap,并集成到 java.util.concurrent 中.该并发跳表被认为是最快的并发跳表,经常被用来和并发平衡二叉树、乐观并发跳表等实现作效率对比.2010 年,Ellen 等人[39]将标志位和原子操作引入平衡二叉树的设计中,极大地降低了由于插入、删除操作造成的平衡开销,首次实现了无阻塞的平衡二叉树,取得很大的突破.从第 3.5 节的分析中我们也可以看出,标志位和原子操作是近几年并发树型结构采用的主要技术.值得注意的是,对于原子操作的选取需要慎重考虑底层硬件的支持,否则整个结构的可移植性和实用性都不高[36].

应用标志位和原子操作的示意图如图 7 所示,每个节点都维护一个标志位,初始状态为 CLEAN,当需要进行插入或删除操作时,多线程采用 CAS 原子操作进行竞争,只有一个线程能够进入插入循环圈(如图 7 所示,在 CLEAN 右边)或删除循环圈(如图 7 所示,在 CLEAN 左边)进行活动,这样使得进行并发操作的多个线程得到隔离,保证了并发操作的安全性.以插入循环圈为例,只有一个线程进入,其他线程在外面等候,通过 iflagCAS 操作将该节点标志位从 CLEAN 设置为 IFLAG,然后,通过 iflagCAS 操作将孩子节点的标志位从 CLEAN 设置为 IFLAG,等到插入完成后,再利用 iunflagCAS 操作将该节点和孩子节点的标志位一同恢复到 CLEAN.

© 中国科学院软件研究所 http://www.jos.org.cn



(a) 数据结构的组织形式

(b) 基于 CAS 的插入和删除操作

Fig.7 The diagram of the flag and atomic operations

图 7 标志位和原子操作的应用示意图

4.6 线程监控数组

Kogan 在 2011 年[73]提出了首个满足无等待演进条件且高效的队列结构,该结构的基本思想是将所有线程的操作记录到线程数组中,使得后来的线程能够帮助之前的线程完成操作,因而每一步操作都可以在固定步骤内完成.这种技术参考了文献[74]中提出的快速和慢速通路方法(fast-path-slow-path methodology).文献[74]又提出了用相同的方法来构建无等待 linked list[75],然而,其证明过程相对复杂.

目前,已有多核数据结构都是特定构建的,支持的功能以及采用的关键技术都存在很大差别,并没有形成一套系统化的规范准则.上述 6 种核心技术各有特色,在多核数据结构设计中需要慎重选取.在对它们进行分析后得到如下对照表(见表 1).

Table 1 The key techniques of concurrenct programming

表 1 并发编程核心技术对照表

核心技术	已实现结构的最大演进条件	对开发人员的友好程度	运行效率	实际应用情况
通用构建	无等待	低	低	低
基于锁的实现	阻塞	中	中	高
基于乐观并发控制	无锁	低	高	低
软件事务内存	无干扰	高	低	低
标志位、原子操作	无锁	低	高	低
线程监控数组	无等待	中	高	低

如表 1 所示,不同核心技术在已实现结构中达到的最大演进条件各不相同,线程监控数组技术和通用构建实现的演进条件最高,达到 Wait free 级别.但是,基于线程监控数组的方案很难实现,实际应用很低.通用构建在几种情况下得分都比较落后,因此不适合作为实际选取的开发技术.基于锁的方案虽然是一种阻塞实现,且运行效率为中等,但是由于从单核时代就已经大量运用,程序员较为熟悉,因此实际应用最广泛.标志位、原子操作和乐观并发控制具有运行效率高等特点,但是对程序员友好程度低,开发难度较大,实际研究和应用刚刚开始,这也意味着它们是目前最具研究潜力的两项技术.

© 中国科学院软件研究所 http://www.jos.org.cn

5 并发数据结构开发

5.1 并发数据结构开发流程

设计并实现并发数据结构是一个复杂的过程,需要涉及很多概念和步骤,整个过程可以总结如图 8 所示.

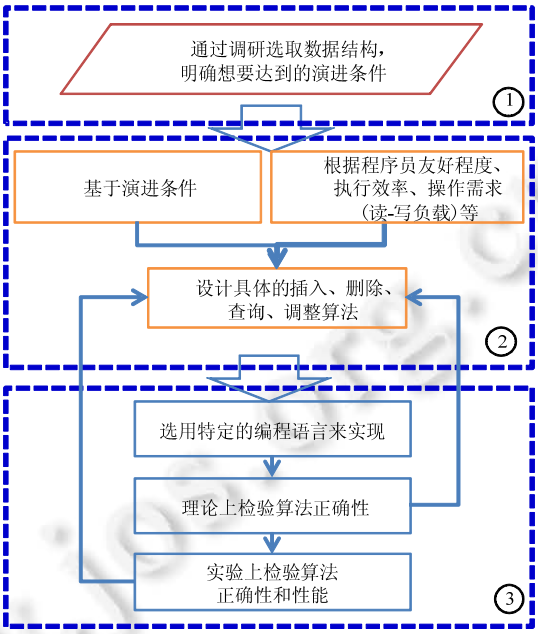


Fig.8 The developing flow diagram of concurrent data structures

图 8 并发数据结构开发流程图

如图 8 所示,并发数据结构的设计和实现工作可以总结为 3 个主要步骤:(1) 分析;(2) 设计;(3) 实现和验证.

(1) 分析

首先通过实际需求调研,选择合适的数据结构.然后查看国内外文献,以及搜索开源代码来确定这些数据结构是否已有成熟的实现.对于已有开源实现的,分析其工作原理、性能和已经达到的演进条件,可作为下一步设计的参考对比.对于还没有实现的,则需要明确演进条件,作为下一步设计的基础.就目前的软硬件条件来说,对多核数据结构的演进条件(意味着能达到的性能)的要求主要集中在无锁层面,对于更高的演进条件(集居数无关无等待、有界无等待和无等待),由于实际完成难度太大,因此建议慎重考虑.

(2) 设计

根据演进条件、对程序员的友好程度、结构执行的效率、应用场景中对于数据结构中执行不同操作的需求(如:侧重读访问或是读写均衡)等因素来选择实现技术手段.例如:如果想要实现一个阻塞的、执行效率高的数据结构,则基于锁的实现是一个很好的选择.如果想要实现一个 obstruction-free 的、程序员开发容易的数据结构则可以考虑软件事物内存这种方式来实现.如果应用需求中读操作很多,而插入、删除比重很小,则可以考虑利用 OCC 技术来实现.最后,再综合考虑数据结构具体的组织形式和相应的插入、删除、查询、调整等算法.合理的技术路线选择能够在很大程度上指导和简化之后的算法设计和实现.

(3) 实现和验证

选用合适的语言来实现,一般建议选择自己最熟悉的语言,这样可以减少开发难度.此外,选择的开发语言必须能够支撑步骤(2)中选择的实现技术手段.例如:在步骤(2)中确定要选择软件事务内存,那么所选用的开发语言需要支持.如果在步骤(2)中是选择基于锁的实现,因为不同开发语言对锁的实现其实差异很大,所以也需要认真评估.选定开发语言后,接下来就是按照之前的设计,具体实现这个数据结构.然后分别从理论和实验角度

© 中国科学院软件研究所 http://www.jos.org.cn

验证数据结构的正确性,并通过实验来验证相应结构的性能.

5.2 Java中现有并发数据结构实现情况

由于并发数据结构存在设计和实现上的难度,因此真正已发布的、稳定的并发库其实并不太多.Java 中的 java.util.concurrent 是极少数得到广泛应用的软件包,现已在并发编程中成为很常用的工具类.在 Java 5 之前,Java 中如果需要处理并发,那么通常需要使用 wait()、notify()和 synchronized 手工实现并发处理的代码,加上需要考虑性能、死锁和资源管理等因素,开发的负担会很重.从 Java 5 开始,Java 推出了 java.util.concurrent 包,以简化并发的完成.这个包有几个小的、已标准化的可扩展框架和一些提供有用功能的类,没有这些类,这些功能会很难实现或实现起来很繁杂.java.util.concurrent 主要包括如下几个部分:Executors、Queues、Timing、Synchronizers、Concurrent Collections,其中,Concurrent Collections 包括了一些常用的并发数据结构.

截止 2015 年发布的 Java 8,java.util.concurrent 提供的数据结构有 ConcurrentLinkedQueue,ConcurrentLinkedDeque,ConcurrentHashMap,ConcurrentSkipListMap,ConcurrentSkipListSet,CopyOnWriteArrayList 和 CopyOnWriteArraySet.由此可见,目前 Java 8 版本中只提供一些最基本的并发数据结构,对于一些复杂的结构,则需要编程人员自己实现.

6 多核 CPU 数据结构正确性理论分析方法

设计出多核CPU数据结构后,就需要评估设计的正确性,通常有两种手段可选:实验的方法和理论分析的方法.实验的方法通常比较直观,理论分析的方法则逻辑性要求更强,表达起来也更困难.

通过对前人在多核 CPU 数据结构正确性分析方法进行对比研究,总结如下.

结构不变性

结构不变性主要用来说明并发操作并不会改变原始数据结构的属性.每种数据结构都有其自身特点的数据组织形式(如:链表和树的结构就不同),这些性质在数据结构被创建时就已成立,并且任何操作都不能改变这些性质.因此,当多线程并发地对多核 CPU 数据结构进行操作时,无论经过多少次插入、删除、查找,最终得到的还是同种性质的数据结构(如:初期是二叉树,经过 N 次并发操作后还应该是二叉树.不允许出现初期是二叉树,经过N次并发操作后变成m叉树).在具体分析说明时涉及到两个关键步骤:(1) 列举出这种结构的不变性质;(2) 对每一种操作都分析其不会改变原来数据结构的性质.举个例子:如果现在来分析并发跳表的结构不变性,则需要首先列举出跳表的结构的不变性质:(1) 哨兵节点不能改变.(2) 每一层的节点都按照关键字排序,并且不会出现重复的关键字.(3) 下层节点的节点数一定不少于上层节点.然后,再分别对查找、插入、删除、置标志位等操作进行分析,说明这些操作并不会改变其跳表的属性.

安全性(safety)

安全性意味着对并发数据结构的操作不会出错[5].对安全性的一个重要评价是可线性化,其基本思想是每一个并发的经历(history)都等价于一个顺序的经历.通常来说,用来说明并发数据结构的可线性化性质的方法就是指出该算法的可线性化点(linearizability point),然后指明在该线性化点的并发操作是如何映射为不同线程的等价顺序操作的.

活性(liveness)

当考虑并发数据结构的活性时,我们期待对这个数据结构的操作最终都能够完成[8].活性意味着并发数据结构算法最终在怎样的情况下完成.通常可以用演进条件来表示,如无死锁、无饥渴、无锁、无等待等.如果保证算法调用不会因为互相等待而无法获取资源,那么称其为无死锁.如果保证算法调用最终都能取得请求的资源,那么称其为无饥渴.如果保证算法某次调用能在有限的步骤内完成,那么这种方法是无锁的.如果算法每次调用总是在有限步骤内完成,那么这种方法是无等待的.

7 结束语及研究展望

随着商用多核(multicore)和众核(many-core)系统越来越普及,软件中数据结构对多核支持的需求也就越来

© 中国科学院软件研究所 http://www.jos.org.cn

越迫切.因此,如何在多核时代提升数据结构的性能成为研究的热点.结合已有相关研究成果及其发展趋势,我们认为,以下有关研究将是研究者近年来所关注的热点问题.

基于硬件事务内存的并发数据结构研究

近年来,CPU 对硬件事务内存的支持不断提高.这为在此之上的并发算法研究提供了便利的条件.Intel 公司在 x86 架构中设计了 Transactional Synchronization Extensions(TSX)扩展指令集,加入 Hardware Transactional Memory(HTM)的支持.2013 年 6 月,Intel 推出了基于 Haswell 微架构的处理器.这成为主流设计中首次引入 Transactional Memory 的处理器.IBM 在 2013 年 8 月的 Hot Chips 会议上推出了 Power 8.它是基于 Power 架构的超标量体系结构对称多处理器家族,Power 8 加入了 Hardware Transactional Memory 的支持.

支持无等待的高效可实用的数据结构研究

按照多核数据结构演进条件的定义,演进条件越高,意味着活性越高,也在一定程度上表明其性能越好.在目前已经实现的数据结构中,只有采用线程监控数组技术实现了高效、实用的无等待的队列,虽然采用通用构建也可以得到无等待的数据结构,但是这些实现效率都不高[76,77].因此,对于很多并不具有无等待性质的数据结构来说,总是有部分比例的线程在多核执行时存在等待的时间.在这个问题上,任何高效、实用的数据结构(如:哈希、链表、树等)研究的突破都将产生较大的实际影响.

基于标志位和原子操作的无锁带调整复杂树型结构研究

标志位和原子操作是近几年来并发树型结构研究的重点,已在平衡二叉树上得到实现.然而,对于一些更加复杂的树形结构,如红黑树、空间树、前缀树、后缀树等,由于其结构组织本身具有特殊性或者存在特殊的调整策略,目前还少有文献涉及,因此这方面具有比较大的研究空间.

分布式系统的多核数据结构研究

现在大型系统都是分布式架构的,文献[78]对分布式系统中不同机器的内存建立了一个统一的映射,并在上面建立 B+树.这样,对于用户来说,看到的是一棵统一的 B+树,然而,B+树上不同节点是映射到不同机器的内存中的.文献[78]中 B+树的多核实现是基于锁的传统实现.那么,如何发挥多核优势,研究并设计出适应分布式环境的无锁数据结构也将是研究者关注的热点.

总之,多核的出现给我们带来机遇的同时也带来了巨大的挑战.基于共享内存的多核数据结构已成为研究者关注的热点问题.为了有效发挥硬件多核的优势,多核数据结构及其相关研究还存在许多挑战性问题,需要不断地去探讨和研究.

References:

[1] Gao L, Wang R, Qian DP. Deterministic replay for parallel programs in multi-core processors. Ruan Jian Xue Bao/Journal of Software, 2013,24(6):1390−1402 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4392.htm [doi: 10.3724/SP.J.1001.2013.04392]

[2] Yuan QB, Zhao JB, Chen MY, Sun NH. Performance bottleneck analysis and solution of shared memory operating system on a multi-core platform. Journal of Computer Research and Development, 2011,48(12):2268−2276 (in Chinese with English abstract).

[3] Anderson JH, Kim YJ. An improved lower bound for the time complexity of mutual exclusion. Distributed Computing, 2002,15(4):221−253. [doi: 10.1007/s00446-002-0084-2]

[4] Arora NS, Blumofe RD, Plaxton CG. Thread scheduling for multiprogrammed multiprocessors. Theory of Computing Systems, 2001,34(2):115−144. [doi: 10.1007/s00224-001-0004-z]

[5] Herlihy M, Shavit N. The Art of Multiprocessor Programming (Revised Reprint). Elsevier, 2012.

[6] Fich FE, Hendler D, Shavit N. Linear lower bounds on real-world implementations of concurrent objects. In: Proc. of the 46th Annual IEEE Symposium on Foundations of Computer Science, 2005 FOCS. IEEE, 2005. 165−173. [doi: 10.1109/SFCS.2005.47]

[7] Moir M, Shavit N. Concurrent data structures. Lecture Notes in Computer Science, 2001,605(12):703−724.

[8] Shavit N. Data structures in the multicore age. Communications of the ACM, 2011,54(3):76−84. [doi: 10.1145/1897852.1897873]

[9] Amdahl GM. Validity of the single processor approach to achieving large scale computing capabilities. In: Proc. of the Spring Joint Computer Conf. ACM, 1967. 483−485. [doi: 10.1145/1465482.1465560]

[10] Michael MM, Scott ML. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. Journal of Parallel and Distributed Computing, 1998,51(1):1−26. [doi: 10.1006/jpdc.1998.1446]

© 中国科学院软件研究所 http://www.jos.org.cn

[11] Treiber RK. Systems Programming: Coping with parallelism. Int'l Business Machines Incorporated, Thomas J. Watson Research Center, 1986.
[12] Shavit N, Touitou D. Elimination trees and the construction of pools and stacks. Theory of Computing Systems, 1997,30(6): 645−670. [doi: 10.1007/s002240000072]
[13] Hendler D, Shavit N, Yerushalmi L. A scalable lock-free stack algorithm. In: Proc. of the 16th Annual ACM Symp. on Parallelism in Algorithms and Architectures. ACM, 2004. 206−215. [doi: 10.1145/1007912.1007944]
[14] Lamport L. Specifying concurrent program modules. ACM Trans. on Programming Languages and Systems (TOPLAS), 1983,5(2): 190−222. [doi: 10.1145/69624.357207]
[15] Hendler D, Shavit N. Work dealing. In: Proc. of the 14th Annual ACM Symp. on Parallel Algorithms and Architectures, ACM, 2002. 164−172. [doi: 10.1145/564870.564900]
[16] Herlihy MP, Wing JM. Linearizability: A correctness condition for concurrent objects. ACM Trans. on Programming Languages and Systems (TOPLAS), 1990,12(3):463−492. [doi: 10.1145/78969.78972]
[17] Martin P, Moir M, Steele G. DCAS-Based concurrent deques supporting bulk allocation. Theory of Computing Systems, 2002, 35(3):59−73.
[18] Herlihy M, Luchangco V, Moir M. Obstruction-Free synchronization: Double-Ended queues as an example. In: Proc. of the 23rd Int'l Conf. on Distributed Computing Systems. IEEE, 2003. 522−529. [doi: 10.1109/ICDCS.2003.1203503]
[19] Bayer R, Schkolnick M. Concurrency of operations on B-trees. Acta informatica, 1977,9(1):1−21. [doi: 10.1007/BF00263762]
[20] Lea D. Concurrent Programming in Java: Design Principles and Patterns. Addison-Wesley Professional, 2000.
[21] Valois JD. Lock-Free linked lists using compare-and-swap. In: Proc. of the 14th Annual ACM Symp. on Principles of Distributed Computing. ACM, 1995. 214−222. [doi: 10.1145/224964.224988]
[22] Harris TL. A pragmatic implementation of non-blocking linked-lists. In: Distributed Computing. Springer-Verlag, 2001. 300−314. [doi: 10.1007/3-540-45414-4_21]
[23] Michael MM. High performance dynamic lock-free hash tables and list-based sets. In: Proc. of the 14th Annual ACM Symp. on Parallel Algorithms and Architectures. ACM, 2002. 73−82. [doi: 10.1145/564870.564881]
[24] Ellis CS. Concurrency in linear hashing. ACM Trans. on Database Systems (TODS), 1987,12(2):195−217. [doi: 10.1145/22952. 22954]
[25] Kung H, Lehman PL. Concurrent manipulation of binary search trees. ACM Trans. on Database Systems (TODS), 1980,5(3): 354−382. [doi: 10.1145/320613.320619]
[26] Hsu M, Yang WP. Concurrent operations in extendible hashing. In: Proc. of the VLDB. Springer-Verlag, 1986. 25−28.
[27] Lea D. Concurrency JSR-166 Interest Site. 2014. http://gee.cs.oswego.edu/dl/concurrency-interest/index.html
[28] Shalev O, Shavit N. Split-Ordered lists: Lock-Free extensible hash tables. Journal of the ACM (JACM), 2006,53(3):379−405. [doi: 10.1145/1147954.1147958]
[29] Pagh R, Rodler FF. Cuckoo hashing. Journal of Algorithms, 2004,51(2):122−144.
[30] Fan B, Andersen DG, Kaminsky M. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. NSDI, 2003,13:385−398.
[31] Herlihy M, Shavit N, Tzafrir M. Hopscotch hashing. In: Distributed Computing. Berlin, Heidelberg: Springer-Verlag, 2008. 350−364. [doi: 10.1007/978-3-540-87779-0_24]
[32] Afek Y, Kaplan H, Korenfeld B, Morrison A, Tarjan RE. CBTree: A practical concurrent self-adjusting search tree. In: Distributed Computing. Springer-Verlag, 2012. 1−15. [doi: 10.1007/978-3-642-33651-5_1]
[33] Brown T, Ellen F, Ruppert E. A general technique for non-blocking trees. In: Proc. of the 19th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. ACM, 2014. 329−342. [doi: 10.1145/2555243.2555267]
[34] Prokopec A, Bronson NG, Bagwell P, Odersky M. Concurrent tries with efficient non-blocking snapshots. In: Proc. of the ACM Sigplan Notices. ACM, 2012. 151−160. [doi: 10.1145/2145816.2145836]
[35] Prokopec A, Bagwell P, Odersky M. Lock-Free resizeable concurrent tries. In: Languages and Compilers for Parallel Computing. Springer-Verlag, 2013. 156−170. [doi: 10.1007/978-3-642-36036-7_11]
[36] Fraser K. Practical lock-freedom [Ph.D. Thesis]. Cambridge: University of Cambridge, 2004.
[37] Crain T, Gramoli V, Raynal M. A speculation-friendly binary search tree. ACM Sigplan Notices, 2012,47(8):161−170. [doi: 10. 1145/2370036.2145837]
[38] Bronson NG, Casper J, Chafi H, Olukotun K. A practical concurrent binary search tree. In: Proc. of the ACM Sigplan Notices. ACM, 2010. 257−268. [doi: 10.1145/1693453.1693488]
[39] Ellen F, Fatourou P, Ruppert E, Van Breugel F. Non-Blocking binary search trees. In: Proc. of the 29th ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing. ACM, 2010. 131−140. [doi: 10.1145/1835698.1835736]

© 中国科学院软件研究所 http://www.jos.org.cn

[40] Howley SV, Jones J. A non-blocking internal binary search tree. In: Proc. of the 24th ACM Symp. on Parallelism in Algorithms and Architectures. ACM, 2012. 161−171. [doi: 10.1145/2312005.2312036]

[41] Natarajan A, Mittal N. Fast concurrent lock-free binary search trees. In: Proc. of the 19th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. ACM, 2014. 317−328. [doi: 10.1145/2555243.2555256]

[42] Drachsler D, Vechev M, Yahav E. Practical concurrent binary search trees via logical ordering. In: Proc. of the 19th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, ACM, 2014. 343−356. [doi: 10.1145/2555243.2555269]

[43] Chatterjee B, Nguyen N, Tsigas P. Efficient lock-free binary search trees. arXiv preprint arXiv:14043272, 2014.

[44] Brown T, Helga J. Non-Blocking *k*-ary search trees. In: Principles of Distributed Systems. Springer-Verlag, 2011. 207−221. [doi: 10.1007/978-3-642-25873-2_15]

[45] Brown T, Avni H. Range queries in non-blocking *k*-ary search trees. In: Principles of Distributed Systems. Springer-Verlag, 2012. 31−45. [doi: 10.1007/978-3-642-35476-2_3]

[46] Crain T, Gramoli V, Raynal M. A contention-friendly binary search tree. In: Euro-Par 2013 Parallel Processing. Springer-Verlag, 2013. 229−240. [doi: 10.1007/978-3-642-40047-6_25]

[47] Leo JG. A dichromatic framework for balanced trees. In: Proc. of the Annual IEEE Symp. on Foundations of Computer Science. IEEE Computer Society, 1978. 8−21. [doi: 10.1109/SFCS.1978.3]

[48] Herlihy M. Wait-Free synchronization. ACM Trans. on Programming Languages and Systems (TOPLAS), 1991,13(1):124−149. [doi: 10.1145/114005.102808]

[49] Herlihy M. A methodology for implementing highly concurrent data objects. ACM Trans. on Programming Languages and Systems (TOPLAS), 1993,15(5):745−770. [doi: 10.1145/161468.161469]

[50] Herlihy M. A methodology for implementing highly concurrent data structures. In: Proc. of the ACM SIGPLAN Notices. ACM, 1990. 197−206. [doi: 10.1145/99163.99185]

[51] Nurmi O, Soisalon-Soininen E. Chromatic binary search trees. Acta Informatica, 1996,33(6):547−557. [doi: 10.1007/s002360050057]

[52] Boyar J, Fagerberg R, Larsen KS. Amortization results for chromatic search trees, with an application to priority queues. Journal of Computer and System Sciences, 1997,55(3):504−521. [doi: 10.1006/jcss.1997.1511]

[53] Besa J, Eterovic Y. A concurrent red-black tree. Journal of Parallel and Distributed Computing, 2013,73(4):434−449. [doi: 10.1016/j.jpdc.2012.12.010]

[54] McKenney PE, Slingwine JD. Read-Copy update: Using execution history to solve concurrency problems. In: Parallel and Distributed Computing and Systems. 1998. 509−518.

[55] Matveev A, Shavit N, Felber P, Marlier P. Read-Log-Update: A lightweight synchronization mechanism for concurrent programming. In: Proc. of the 25th Symp. on Operating Systems Principles. New York: ACM, 2015. 168−183. [doi: 10.1145/2815400.2815406]

[56] Kung HT, Robinson JT. On optimistic methods for concurrency control. ACM Trans. on Database Systems (TODS), 1981,6(2):213−226. [doi: 10.1145/319566.319567]

[57] Herlihy M, Luchangco V, Moir M, Scherer Iii WN. Software transactional memory for dynamic-sized data structures. In: Proc. of the 22nd Annual Symp. on Principles of Distributed Computing. ACM, 2003. 92−101. [doi: 10.1145/872035.872048]

[58] Ballard L. Conflict avoidance: Data structures in transactional memory [Ph.D. Thesis]. Brown University Undergraduate, 2006.

[59] Herlihy M, Lev Y, Luchangco V, Shavit N. A provably correct scalable concurrent skip list. In: Proc. of the Conf. on Principles of Distributed Systems (OPODIS), 2006. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.170.719&rep=rep1&type=pdf

[60] Herlihy M, Moss JEB. Transactional memory: Architectural support for lock-free data structures. Journal of the ACM, 1993,21(2):289−300. http://hpl.americas.hp.net/techreports/Compaq-DEC/CRL-92-7.pdf

[61] Wikipedia. Transactional synchronization extensions. 2014. http://en.wikipedia.org/w/index.php?title=Transactional_Synchronization_Extensions&oldid=606067145

[62] Shavit N, Touitou D. Software transactional memory. Distributed Computing, 1997,10(2):99−116. [doi: 10.1007/s004460050028]

[63] Adl-Tabatabai AR, Kozyrakis C, Saha B. Unlocking concurrency. Queue, 2006,4(10):24−33. [doi: 10.1145/1189276.1189288]

[64] Adl-Tabatabai AR, Lewis BT, Menon V, Murphy BR, Saha B, Shpeisman T. Compiler and runtime support for efficient software transactional memory. In: Proc. of the ACM SIGPLAN Notices. ACM, 2006. 26−37. [doi: 10.1145/1133981.1133985]

[65] Ansari M, Kotselidis C, Watson I, Kirkham C, Luján M, Jarvis K. Lee-TM: A non-trivial benchmark suite for transactional memory. In: Algorithms and Architectures for Parallel Processing. Springer-Verlag, 2008. 196−207. [doi: 10.1007/978-3-540-69501-1_21]

[66] Minh CC, Chung J, Kozyrakis C, Olukotun K. STAMP: Stanford transactional applications for multi-processing. In: Proc. of the 2008 IEEE Int'l Symp. on Workload Characterization. IEEE, 2008. 35−46. [doi: 10.1109/IISWC.2008.4636089]

© 中国科学院软件研究所 http://www.jos.org.cn

[67] Dalessandro L, Marathe VJ, Spear MF, Scott ML. Capabilities and limitations of library-based software transactional memory in C++. In: Proc. of the 2nd ACM SIGPLAN Workshop on Transactional Computing. 2007. 49. https://anon.cs.rochester.edu/u/scott/papers/2007_TRANSACT_RSTM2.pdf

[68] Dice D, Shalev O, Shavit N. Transactional locking II. In: Distributed Computing. Springer-Verlag, 2006. 194−208. [doi: 10.1007/11864219_14]

[69] Dice D, Shavit N. What really makes transactions faster. In: Proc. of the TRANSACT Workshop. 2006. http://www.cs.tau.ac.il/~shanir/nir-pubs-web/Papers/TRANSACT06.pdf

[70] Eddon G, Herlihy M. Language support and compiler optimizations for STM and transactional boosting. In: Distributed Computing and Internet Technology. Springer-Verlag, 2007. 209−224. [doi: 10.1007/978-3-540-77115-9_22]

[71] Wikipedia. Software transactional memory. 2014. http://en.wikipedia.org/w/index.php?title=Software_transactional_memory&oldid=613666501

[72] Cascaval C, Blundell C, Michael M, Cain HW, Wu P, Chiras S, Chatterjee S. Software transactional memory: Why is it only a research toy. Queue, 2008,6(5):40−46. [doi: 10.1145/1454456.1454466]

[73] Kogan A, Petrank E. Wait-Free queues with multiple enqueuers and dequeuers. ACM SIGPLAN Notices, 2011,46(8):223−234. [doi: 10.1145/2038037.1941585]

[74] Kogan A, Petrank E. A methodology for creating fast wait-free data structures. In: Proc. of the ACM SIGPLAN Notices. ACM, 2012. 141−150. [doi: 10.1145/2145816.2145835]

[75] Timnat S, Braginsky A, Kogan A, Petrank E. Wait-Free linked-lists. In: Principles of Distributed Systems. Springer-Verlag, 2012. 330−344. [doi: 10.1007/978-3-642-35476-2_23]

[76] Fatourou P, Kallimanis ND. A highly-efficient wait-free universal construction. In: Proc. of the 23rd Annual ACM Symp. on Parallelism in Algorithms and Architectures. ACM, 2011. 325−334. [doi: 10.1145/1989493.1989549]

[77] Chuong P, Ellen F, Ramachandran V. A universal construction for wait-free transaction friendly data structures. In: Proc. of the 22nd ACM Symp. on Parallelism in Algorithms and Architectures. ACM, 2010. 335−344. [doi: 10.1145/1810479.1810538]

[78] Aguilera MK, Golab W, Shah MA. A practical scalable distributed b-tree. Proc. of the VLDB Endowment, 2008,1(1):598−609. [doi: 10.14778/1453856.1453922]

附中文参考文献:

[1] 高岚,王锐,钱德沛.多核处理器并行程序的确定性重放研究.软件学报,2013,24(6):1390−1402. http://www.jos.org.cn/1000-9825/4392.htm [doi: 10.3724/SP.J.1001.2013.04392]

[2] 袁清波,赵健博,陈明宇,孙凝晖.多核平台共享内存操作系统性能瓶颈分析及解决.计算机研究与发展,2011,48(12):2268−2276.

周维(1974－),男,湖南常德人,博士,副教授,主要研究领域为分布式计算,云计算.

姚绍文(1966－),男,博士,教授,博士生导师,主要研究领域为分布式计算,工作流.

周可人(1992－),男,学士,主要研究领域为并发编程,并行算法优化.

钱德沛(1952－),男,教授,博士生导师,CCF会士,主要研究领域为计算机系统结构,分布式计算,高性能计算,并行计算.

栾钟治(1971－),男,博士,副教授,CCF 高级会员,主要研究领域为分布式计算,高性能计算,并行计算,计算机系统结构.

© 中国科学院软件研究所 http://www.jos.org.cn