SXM 系统源代码分析

学　　院 计算机科学与技术

专　　业 计算机科学与技术

年　　级 **2003 级**

姓　　名 何福星

指导教师 张坤龙

2007 年 6 月 15 日

摘　要

随着计算机体系结构的不断发展，多核计算机出现了，它对程序的并发性提出了很高的要求。为了解决人们在编写高效的并行程序时遇到的困难，软件事务内存被用于基于共享内存的并行程序设计，取得了很好的效果。

为了深入理解软件事务内存技术，论文分析了一个名为 SXM 的软件事务内存系统的源代码。论文取得的主要成果如下：1）分析了 SXM 系统中共享对象的表示方法、事务的表示方法、以及事务与共享对象间的关系。2）分析了 SXM 系统如何检测冲突，描述了它检测冲突的方法。3）分析了当冲突被检测到之后 SXM 系统如何解决冲突，描述了它的各种冲突管理策略。

关键字：软件事务内存；SXM；并发；非阻塞；多核计算机

目　　录

第一章　　绪论

1.1　事务内存的来源

1.1.1　并行程序设计

1965 年，Gordon Moore 提出摩尔定律：“芯片上的晶体管数目大约每过两年就翻一番”[1]。好消息是，摩尔定律在现在和将来的一段时间内仍然有效。不好的消息是，由于功耗和散热方面的原因，现在已经难以通过提高时钟频率来提升处理器的性能。这两个因素导致的结果是多核处理器应运而生[2]。下一代芯片将集成更多共享内存的 CPU，但同时单个 CPU 却不会跑得比上一年的型号更快。如果希望程序跑得更快，必须学习如何去编写并行程序。

而并行程序由于存在并发和不确定性，并发程序设计比顺序程序设计困难有着心理学上的原因。心理学研究表示，人把注意力分散开来处理多个任务的能力是有限的[3]。另外，在顺序程序分析中只需要考虑上下文，而在并行程序分析中还需要考虑同步。因此这类程序难于调试，同时程序中的错误也几乎不可能再现。所以虽然并行计算机已经出现了 40 多年，但是无需修改就能够充分利用多核处理器的并行应用却很少。

在目前的任务并行程序设计模型中，使用显式的同步原语，如 Lock/Unlock。而显式的同步原语使程序设计变的困难，例如链表的并行版本就比顺序版本复杂的多；另一方面，显式的同步原语也使得更高级的抽象原语难以组合，例如只使用插入和删除原语就无法组合出一个移动原语来在两个链表间移动一个元素，而使用同步原语就有可能导致死锁；免锁算法试图避免使用 Lock/Unlock 这样的同步原语，然而这种算法的设计却更加困难[4]。

1.1.2　基于锁的同步

在多核处理器程序设计中，需要解决的关键问题是同步多个线程对共享内存的访问。传统上，这一问题主要使用基于锁的同步机制来解决。但是锁有下列问题：1）粗粒度的锁虽然易于使用，但是性能低。例如，通过一个唯一的全局锁来同步所有对共享链表的操作，这时即使是两个无关的操作也要被迫顺序执行；2）细粒度的锁虽然性能高，但是难于使用，并且容易导致优先级反转、护航效应（convoying）、死锁等问题；3）基于锁的程序可组合性差。例如，从链表 A 移动一个元素 e 到链表 B 的移动操作 Move(B, A, e)可以利用删除操作 Delete(A, e)和插入操作 Insert(B, e)的组合来完成。这时，需要使用低性能的粗粒度的锁来避免其它线程看到中间状态。否则的话，Move(B, A, e_1)和 Move(A, B, e_2)两个操作并行执行时又有可能导致死锁[5]。

这是一个简单的编程任务：编写一个例程，将钱款从一个帐户转移到另一个

帐户。简单起见，两个帐户都储存在内存中：不需要与数据库进行任何交互。这个例程必须能够在并发程序中正确地工作，在该程序中很多线程将可能同时调用 transfer。任何一个线程都不允许观察到事务的中间状态：钱款已经从一个帐户转出，而还没有转入另一个帐户（反之亦然）[6]。

下面来看几个通常的解决方法当前在并发程序调度中占有统治地位的技术是 锁(lock) 与 条件变量(conditional variable) 。在面向对象语言中，每个对象都具有一个隐式锁，而锁定操作通过 同步方法(synchronized method) 完成，但其思想别无二致。因此一个帐户类可以定义为：

```
class Account
{
int balance;
synchronized void withdraw( int n )
{
balance = balance - n;
}
void deposit( int n )
{
withdraw( -n );
}
}
```

这里必须小心地将 withdraw 声明为 synchronized，从而当两个线程同时调用 withdraw 时，将不会出现某个取款操作丢失的错误。synchronized 的作用是将帐户锁定，执行 withdraw，然后释放锁。

现在，transfer 可能写成这样：

```
void transfer( Account from, Account to, int amount )
{
from.withdraw( amount );
to.deposit( amount );
}
```

对串行程序而言，这段代码没有问题，但在并发程序中，另一个线程可能观察到一个中间状态：钱款已经从 from 中转出，但还没有转入 to 中。尽管两个方法（withdraw 和 deposit）都是 synchronized，但这一点用处也没有。帐户 from 首先被 withdraw 方法锁定并解锁，而后被 deposit 方法锁定并解锁。在这两次调用之间，钱款（看起来）从这两个帐户中消失了。

在一个金融程序中，这将是不可接受的。该如何修复这个问题呢？通常的方

案是添加显式锁定代码，如下：

```
void transfer( Account from, Account to, int amount )
{
from.lock(); to.lock();
from.withdraw( amount );
to.deposit( amount );
from.unlock(); to.unlock();
}
```

但这个程序将极易出现致命的死锁。特别地，考虑以下（并不常见的）状况：另一个线程正在同样的两个帐户之间进行相反方向的转帐操作。此时每个线程都将取得一个锁并等待对方解锁，从而陷入永远的阻塞之中。

一旦认识到这一点——问题并不总是很明显的——那么通常的修复方式便是在锁上附加一种任意的全局顺序，而后以升序请求它们。锁定代码将变为：

```
if from < to
then { from.lock(); to.lock(); }
else { to.lock(); from.lock(); }
```

当全部所需的锁都能被预见到时，这种方法工作得不错。但问题并不总是这样简单的。例如，假设 from.withdraw 被实现为：当 from 中的资金不足时，就从 fromdeposit（译者注：一个内部对象）中转出钱款。（在转帐时）并不知道是否应该请求 fromdeposit 的锁，直到读取 from 中的数据之后才行。但此时（读取之后）若还想以“正确”的顺序来请求锁，为时已晚。更进一步说，fromdeposit 的存在是一个私有的问题，只能为 from 所知，而不能为 transfer 所知。即使 transfer 确实知晓 fromdeposit 的存在，锁定代码也必须处理三个锁——大概是通过将它们排序的方式。

当希望使用阻塞操作时，事态变得更复杂了。例如，假设当 from 中资金不足时，transfer 将发生阻塞。这一般是通过等待一个条件变量，同时释放 from 的锁来实现的。如果希望阻塞持续至 from 中有足够资金为止，同时将 from_deposit 纳入考虑范围，那么情况还会变得更加诡异。

长话短说，当前占统治地位的并发式编程技术——锁与条件变量——存在根本性的缺陷。以下是几点典型的困难，有一些已经在上文中提及：获取太少的锁，程序员很容易忘记去获取锁，最终导致两个线程同时修改同一个变量；获取太多的锁，程序员很容易获取太多锁，从而阻止了程序的并发（最好的情况），或者造成死锁（最糟的情况）；获取错误的锁，在基于锁的编程中，锁与其保护的数据之间的联系常常只存在于程序员的头脑中，而不是显式存在于程序中。其结果是，获取或保持错误的锁实在是太容易了；以错误的顺序获取锁，在基于锁的编

程中，程序员必须小心翼翼地以“正确”的顺序获取锁。避免死锁的工作总是令人厌倦，错误丛生，并且有时是极端困难的[7]。

但是，基于锁的编程其最根本的缺点是：锁与条件变量不支持模块化编程。说到“模块化编程”时，指的是通过胶合小程序构造大程序的过程。锁摧毁了其可能性。例如，无法在不修改 withdraw 和 deposit 的（正确）实现的情况下，使用它们来实现 transfer。取而代之地，必须暴露锁协议。阻塞与选择甚至更为反模块化[8]。例如，假设有一个当源帐户资金不足时将阻塞的 withdraw 版本，那么在不暴露阻塞条件的情况下，将不能直接使用该 withdraw 来从 A 或者 B 取出钱款（选择 A 或 B 取决于哪个帐户资金充足）——甚至即使暴露了条件，要实现这个功能也还是不容易。在其它文献中，对这一批判有详细的阐述。

1.1.3 数据库与事务内存

从上节可见，基于锁的同步程序设计是有很多缺点的，由此引出锁的两个主要备选方案。一个是无锁编程（Lock-Free Programming）。通过对处理器内存模式的深入认识，建立可共享但无显式锁定的数据结构是可能的。无锁编程难度很大且非常脆弱，因此新的无锁数据结构实现方案，仍在不断修改之中。第二个替代物就是事务内存（Transactional Memory），它将数据库中事务模式的核心思想移植到编程语言里来。程序员将自己的程序编写为一个个具有确定原子性的块，它们可以分离执行，因此只有在每个原子行为执行前和执行后，并发操作才能访问共享数据[9]。尽管很多人看好事务内存的前途，但它目前仍处于研究之中。

事务内存是从数据库系统受到的启发。数据库系统能有效的适应各种并行环境，其原因是数据库只能通过事务来访问。事务是用户定义的一个数据库操作序列，具有以下特性：原子性，一致性，隔离性，持久性。在数据库系统中，通过将一系列操作抽象为事务，避免了程序设计员考虑复杂的同步问题。由此引入的软件事务内存(Software Transactional Memory, STM) ，一种令人欢欣鼓舞的新方法。它被用于进行基于共享内存的并行多处理器的编程，它似乎能以一种当前技术所不能提供的方式来支持模块化程序。阅读完本文之后，希望你也能为 STM 而疯狂。它不是万灵药，但它是对并发这一令人畏惧的堡垒的一次漂亮而振奋人心的进攻。

而事务内存和数据库的区别也是很明显的，数据库中的数据存储在磁盘上，事务内存中的数据存储在主存中。假定一次磁盘存取时间是 5ms，一次主存存取时间是 50ns，CPU 的计算速度为 10GPIS，则在一次磁盘存取时间内 CPU 可执行 5 千万条指令，而在一次主存存取时间内 CPU 只可执行 500 条指令。数据库中的数据需要持久存储，而事务内存中的数据不需要持久存储；数据库系统是一个封闭的世界，所有对磁盘数据的访问都必须通过数据库系统来进行。但是对内存数据的访问不太可能只是来自于事务内存系统，事务内存系统必须和现有的程

序设计语言、程序库、操作系统等共存[10]。

1.1.4 事务内存的发展

外国对事务内存的研究已经有了较长的历史。早在 1977 年，Lomet 就提出了在程序设计语言中引入类似数据库事务的抽象的想法，但是没有给出实现。1993 年，Herlihy 和 Moss 实现了第一个硬件事务内存系统，这一系统通过增加几条新的指令和一个事务高速缓存来同步对有限数量的共享内存位置的访问。1995 年，在假设事务要访问的内存位置事先已经全部知道的前提下，Shavit 和 Touitou 实现了第一个软件事务内存系统[11]，该系统完全依靠软件来同步对共享内存的访问，不需要增加新的硬件。

随着多核处理器投入商用，对事务内存的研究也变得空前活跃起来。根据在 2007 年 5 月 31 所做的统计，在 Transactional Memory Online Bibliography 网站（http://www.cs.wisc.edu/trans-memory/biblio/）上列出的 159 篇参考文献中，有超过半数是在 2004-2006 年期间发表的，具体分布是 2004 年 17 篇，2005 年 25 篇，2006 年 40 篇。尤其值得提起的是，近几年中涌现了一批新的事务内存系统，包括软件事务内存系统 DSTM、WSTM、SXM、RSTM、TL、McRT-STM、BSTM，硬件事务内存系统 TCC、LTM/UTM、Bulk、LogTM、VTM、HSTM、HyTM、RTM 等等[12]。这些事务内存系统虽各具特色，但事务内存程序设计模型作为一种新的并行程序设计模型，在走向实用之前还有许多的问题需要解决。虽然事务内存系统还不够实用，但是已经有人在研究如何应用事务内存程序设计模型来开发软件了，例如 Carlstrom 等人构建了基于事务内存的类库，Ramadan 等人设计和实现了一个基于事务内存的操作系统原型[13]。

1.2 事务内存的概念

1.2.1 事务语义

事物的语义主要介绍 ACI 的特性，可线性化，操作语义，非事务存取。ACI 特性是指原子性，一致性和隔离性，未涵盖非事务存取。可线性化是正确共享对象的充分条件，它假定对对象方法的调用结果是在调用和返回之间的某个时刻达成的；可线性化可以看作是可串行化的特殊情形，但是它们之间也有着区别。可线性化是一个方法作用于一个对象，其特性包括局部性和非阻塞性，适用于单个共享内存对象的共享。可串行化是一个方法作用于多个对象，不具备局部性和非阻塞性，适用于数据库的共享，且可线性化和可串行化一样，也未涵盖非事务存取。操作语义中主要是单锁原子性，指所有事务共享一个全局锁，事务开始前加

锁，事务终止后解锁，这样在任何时刻都只能有一个事务在执行中；这种操作语义满足了隔离性的要求，但是它不能完全刻画原子性，事务因为冲突自动夭折后重新执行，它也无法应用于嵌套事务。非事务存取指当事务之内的代码和事务之外的代码共享数据时，就有可能产生冲突。

而在非事务存取上，当事务之内的代码和事务之外的代码共享数据时，就有可能产生冲突。在下面的例子中，线程 T1 想做一个私有化的操作，如果线程 T2 的操作不被封装成一个事务，就有可能出现数据竞争[14]。

```
Thread T1:
ListNode res;
atomic {
    Res=1Head;
    If (lHead!=null)
      Lhead=1head.next;
}
use res;
```

```
Thread T2:
atomic {
    ListNode n=1Head;
    while (n!=null) {
        n.val++;
        n=n.next
    }
}
```

1.2.2 事务内存程序设计

事务内存有多种实现方法。本论文里主要分析描述 CSharp 语言版本的程序设计与实现。下列为设计结构中起重要作用的四个编程接口[15]。

1.atomic：事务是一个代码序列，被封装在一个原子块内；事务的执行结果包括事务提交（可改变程序状态）事务夭折（不改变程序状态）事务不终止（结果不定义）；当两个事务产生冲突时，其中的一个会自动夭折，然后再重新执行。

2.bort：用户可以使用 abort 使事务显式夭折，其效果相当于事务未被执行。

3.retry：可以使事务显式夭折，这时事务所做的工作全被抛弃，事务重新开始执行。使用 atomic 实现互斥，使用 retry 实现条件同步。

4.orElse：安排事务执行顺序及是否执行。如例：T1,T2 是事物，“T1 orElse T2”，则会出现下列可能，分别为执行 T1；若 T1 终止，则 orElse 语句结束；若 T1 执行一个 retry 语句，则开始执行 T2；若 T2 终止，则 orElse 语句结束；若 T2 执行一个 retry 语句，则等待之后转为执行 T1。

但要注意的是：事务不是万能的！Blundell 等给出了一个例子，说明使用事务也能写出错误的程序[16]：

```
bool flagA = false;
bool flagB = false;
Thread 1:
atomic {
    while (!flagA);
```

```
Thread 2:
atomic {
    flagA = true;
```

```
        flagB = true;                          while (!flagB);
    }                                      }
```

上例线程 1 中，开始时 flagA 为假，所以 while 循环会一直执行，这时如果线程 B 开始执行，先把 flagA 设为真，因为原子操作还没结束，所以线程 1 这里还不能发现 flagA 已经被置为真了，必须等待线程 2 的 while 循环跳出。但因为 flagB 开始设为假，而线程 1 中的 flagB=ture 又得不到执行，所以线程 2 中的 while 循环会无限下去，与此同时，线程 1 也停留在 while 循环上，造成两个原子操作都得不到执行的情况，造成死锁。

1.3 课题的研究意义

1.3.1 STM 的研究意义

多核计算机的普及给传统的程序设计模型带来了巨大的挑战，由于需要显式地去解决线程间的同步问题，设计出可靠并且高效的多线程应用程序目前还是一件很困难的事情。事务内存编程模型通过将数据库系统中的事务处理技术移植到在程序设计语言中，有望能够使得程序设计员摆脱使用锁、信号量、管程等进行同步的负担，是目前公认的最有希望的新的多核程序设计模型。软件事务内存，通过将一系列操作抽象为事务，避免了程序设计员考虑复杂的同步问题。

在把一系列操作封装成事务后，在多核计算机上编写高效的并行程序将会展现新的发展空间，程序处理速度将进一步提升，多核时代将全面到来。

1.3.2 SXM 系统的研究意义

SXM 系统是着眼于事务内存实现的，向事务内存打开了一扇门。通过对 SXM 的学习，首先明白了基于 STM 基础上的利用，里面很多概念如非阻塞、高效等都在 SXM 里得到了很好的体现。SXM 里提供一个具体的利用事务内存的范例，通过学习里面的编程思想，会对 STM 思想有了进一步的认识。

国外的事务内存研究已成一定规模，而国内近几年才算兴起，所以跟先进水平还有一定差距。在事务内存编程方面，通过研究 SXM 系统，掌握主流的事务编程思想，有利于拓展这方面的视野，对其内部运行机制有一定的理解。

1.4 论文的结构

下文的第二章将会介绍 SXM 系统的运行和执行结果显示和系统的编程接口，为了更好的解释共享对象的概念，第二章还会先介绍 C#中反射机制；第三章里，着重介绍 SXM 里两个重要概念事务和共享对象及其两者之间的关系；第四章着主要为事务冲突检测；第五章继冲突检测之后，提出 SXM 系统里解决冲突的方法，即竞争管理策略；最后一章为对全文的总结。

第二章　　SXM 系统

2.1　SXM 系统的运行显示

下载 Visual Studio.net 2005 系统，按提示信息即可完成安装。此程序可用来打开修改和执行 SXM。把 SXM 目录下的可执行文件 SXM 打开，在 Studio 中进行进一步分析运行。

你的执行必须可以被共同语言运行时间所允许，否则SXM不会工作。这意味着你的执行必须在一本地软盘上，不是网络系统文件。从shell上运行SXM，使用或稍后使用Visual Studio. Net 2005建立SXM。

SXM -b main [-m mgr] [-t #threads] [-n #ms] [-e experiment#] [-f shefactory]

这是在 shell 里运行 SXM 的命令行，其中包括各输入参数。下面先介绍一下各参数所代表的意义。

·-b：规范的 benchmark 名 (在 9 个 benchmark 里选一个 如 SXM.List)

·-m: 规范的 contention manager 名.默认为:SXM.GreedyManager

·-m: 规范的事务对象包装名(factory name)。默认为：SXM.TMemFactory

·-t: 线程数。默认为：1

·-n: 运行毫秒数。默认为：5000（5 秒）。

·-e: 被 benchmark 解释的整数位。整默认为 1。

·-f: 规范的对象因素。默认为：SXM.TMemFactory

默认的设置可以在文件目录 Defaults.cs 下修改。

要从 Visual Studio 上运行 SXM，在 Solution Explorer 窗口下右键点击 SXM，进了命令窗口下，输入 Debugging->Command 命令行。如上所述，即为运行 SXM 显示结果的过程。

在全部为默认设置的情况下，本机的显示结果如下：

SXM: C# Software Transactional Memory

<c> Microsoft Corporation. All rights reserved

Benchmark:　　SXM.Nested

Manager:　　SXM.AggressiveManager

Factory:　　SXM.TMemFactory

Threads:　　2

Mix:　　100% modifier calls

Milliseconds:　　30000

Data Structure OK

Elapsed time: 00:00:30.2635168 seconds.

Transactions:

Committed:	1,442,769	
Commit ratio:	99.96%	1442769/1443283
Retries:	0	

Calls:

Insert:	480915
Remove:	480914
Contains:	0

如上所示，Benchmark,Manager,Factory,Threads,Milliseconds 都是可以选择的参数，在不同的参数选择下会有不一样的结果。在设置的一组参数下，系统会先检查数据结构，如果是对的就会显示 Data Structure OK. Elapsed time 是指对系统进行扫描和程序执行的时间，在你自己设定的时间（Milliseconds）左右，会有一定偏差。

Transactions 部分是事务执行部分：Committed 是提交的事务数；Commit ratio 是提交的事物占总事务数的比率，这个比率越好事物内存的效果越好。提交即是事务顺利达成，没被丢弃且等待数在一定范围内，这显示了此系统的效率和正确率；Retries 是重试次数。

Calls 部分是原子操作数目，包括了 Insert(插入操作)，Remove（移除操作），Contains（嵌套操作）。一般来说 Insert 数加上 Remove 数即为事务操作成功提交数目(Committed)。

2.2 SXM 系统的编程接口

SXM 是想要方便新的算法的实验和软件事务内存技术的实施。这里鼓励使用者使用和体验这一新元素，特别是 benchmarks,contention managers, 和 object factories, 和使它们可以为将来 SXM 的开发作出贡献。

请看下列代码，声明一个表里的元素

```
[Atomic]
public class Node
{
    protected int value;
    protected Node next;
    public Node(int value)
    {
        this.value = value;
```

```
}
    public virtual int Value
{
    get
    {
        return value;
    }
    set
    {
        this.value = value;
    }
}
public virtual Node Next
{
    get
    {
        return next;
    }
    set
    {
        this.next = value;
```

第一行中的 Atomic，代表了这个类中在 SXM.AtomicAttribute 下的属性。所有被并发事物共享的对象都必须在有此属性的类下。

同时，这个类本身有两个域：value（值域）和 next（指针，指向下一个元素），意义显而易见。这两个域必须被保护（代码中 protected）。通向这两个域的入口被 properties 控制，这些 properties 必须是公共的且有效（public and virtual）。

这个类提供了一个结点（int value）。原子对象通过两步被制造：第一步，在结点类中建立一个 factory：

```
IFactory factory = new XAction.MakeFactory(typeof(Node));
```

这个 factory 创建了截取所有权调用的事务代理，一但这个 factory 被创建，则单个对象就被创立了（关键代码如下）：

```
Node node = (Node)factory.Create(value)
```

对 factory.Create(…)的调用是调用同一数目和类型的 constructor 类型，各种事务外的方法调用都有着通常的效用，非同步方法调用。它们不保证线程的安全性，但是在建造 benchmark 和安全检查之前，对建立初始数据对象非常有用。

一种基于数据共享的操作方法对于对象列表设定定长的条件变量，而且返回对象类的一个值。比如，这是一个关于往表中插入元素的代码：

```
public override object Insert(object _v)
{
    int v = (int)_v;
    Node newNode = (Node)factory.Create(v);
    Node prevNode = this.root;
    Node currNode = prevNode.Next;
    while (currNode.Value < v)
    {
        prevNode = currNode;
        currNode = prevNode.Next;
    }
    if (currNode.Value == v)
    {
        return false;
    }
    else
    {
        newNode.Next = prevNode.Next;
        prevNode.Next = newNode;
        return true;
    }
}
```

这段代码表明了怎样利用 factories 和 properties 去建立一个事务的主体结构。为了准备被事务执行的这个方法，必须转向一个 XStart 的代表，一种正常强硬类型的指针：

```
XStart insertXStart = new XStart (Insert);
```

这个调用建立了一个新的 XStart 的代表，利用了条件变量并返回结果。这里可以像事务一样可以执行这个代表：

```
XAction.Run (insertXStart , value);
```

这里，这种方法是跟随一个冲突调用的，这个冲突在这个方法里被解决。执行原子对象的规定有：

·所有域都被保护且可见的；

·所有域必须或者是标量的或者是拥有原子属性的类的参数。

·用事务对象 factory 来建立一个新的原子对象

Ⅰ 用调用来建立一个 factory 的默认类型

IFactory factory = XAction.MakeFactory(type)

Ⅱ 用调用建立一个 type 的对象类型

Type x = factory.Create(…)

这里先来看看 orElse 处理法。SXM 提供了可选择的执行途径。假如你想从缓冲区 b1 里移除一个元素，但如果 b1 是空的，为了不引起阻塞你可以从缓冲区 b2 里移除一个元素。

让 Get1( )为从 b1 里试着移除一个元素的方法，如果缓冲区为空调用 Retry，同样的方法设个 Get2( )。建立一个 XStart 代理如下：

getXStart = XAction.OrElse(new XStart(Get1), new XStart(Get2));

这个合成的代理可以像其它任何的事务一样运行

int x = (int)XAction.Run(getXStart)

SXM 通过 XAction.Retry( )支持一种新的模形式的条件等待。当事务被修改的对象入口暴露时，这种调用会丢弃并发的事务，重新启动它。

比如，这段代码是表示如何执行一个缓冲区，从它本身的原子对象来启动它：

```
[Atomic]
    public class Buffer
    {
        protected int capacity;
        protected int size;
        protected AtomicArray data;

        public Buffer(int capacity)
        {
            this.data = new AtomicArray(capacity);
            this.size = 0;
            this.capacity = capacity;
        }
        // Size and Capacity properties omitted for brevity.
        // Indexer to access the buffer array.
        public virtual int this[int i]
        {
            get
```

```
            {
                return (int)this.data[i];
            }
            set
            {
                this.data[i] = value;
            }
        }
    }
```

像上个例子一样，被保护的 capacity 和 size 域只有公用的实际 Capacity 和 size 能访问。这些元素被放在 AtomicArray 类型的域里（你不能用一个通常的整数序列因为它不是原子性的）。元素被一个 C# 的索引访问，这是一个用类序列语法的方法。

这里是一段 benchmark 的代码：

```
public class Buffer: SXM.Benchmark
{
    // atomic shared buffer
    Buffer buffer;
    …
    /// <remarks>
    /// Put a value into the buffer.
    /// </remarks>
    public object Put(params object[] _v)
    {
        // value to be put in buffer
        int v = (int)_v[0];
        // check buffer size
        int size = buffer.Size;
        // cannot proceed if buffer is full
        if (size == buffer.Capacity - 1)
        {
            // try again later
            XAction.Retry();
        }
        // put item in buffer
```

```
        buffer[buffer.Size++] = v;
        return null;
```

这里的 Put( ) 首先测试看缓冲区是否已满。如果满了，调用 XAction.Retry( ) 稍后再试一次。不能保证当事务重启时能在缓冲区内找到位置，但是 SXM 的实时运行系统仅当其它事务修改这个对象时会返回值。否则，当缓冲区有空间时，这种方法会放入新的项目并把当前的空间向上累加。

2.3 C#的反射机制

在以 C#编写的 SXM 系统中，多次使用了 C#语言的反射机制来共享类与函数，在整个系统中有着重要作用。这节就是关于反射的简单介绍。

反射的定义：审查元数据并收集关于它的类型信息的能力。元数据（编译以后的最基本数据单元）就是一大堆的表，当编译程序集或者模块时，编译器会创建一个类定义表，一个字段定义表，和一个方法定义表等。System.Reflection 命名空间包含的几个类，允许你反射（解析）这些元数据表的代码。和反射相关的命名空间有（就是通过这几个命名空间访问反射信息）：

System.Reflection.MemberInfo
System.Reflection.EventInfo
System.Reflection.FieldInfo
System.Reflection.MethodBase
System.Reflection.ConstructorInfo
System.Reflection.MethodInfo
System.Reflection.PropertyInfo
System.Type
System.Reflection.Assembly

C#中反射的作用明显：

1．可以使用反射动态地创建类型的实例，将类型绑定到现有对象，或从现有对象中获取类型

2．应用程序需要在运行时从某个特定的程序集中载入一个特定的类型，以便实现某个任务时可以用到反射。

3．反射主要应用与类库，这些类库需要知道一个类型的定义，以便提供更多的功能。

下面看一个简单的利用反射获取类型信息的例子：

```
using system;
using sytem.reflection;
class reflecting
{
```

```
static void Main(string[]args)
{
reflecting reflect=new reflecting();//定义一个新的自身类
//调用一个 reflecting.exe 程序集
assembly myAssembly =assembly.loadfrom(“reflecting.exe”)
reflect.getreflectioninfo(myAssembly);//获取反射信息
}
//定义一个获取反射内容的方法
void getreflectioninfo(assembly myassembly)
{
type[] typearr=myassemby.Gettypes();//获取类型
foreach (type type in typearr)//针对每个类型获取详细信息
{
//获取类型的结构信息
constructorinfo[] myconstructors=type.GetConstructors;
//获取类型的字段信息
fieldinfo[] myfields=type.GetFiedls()
//获取方法信息
MethodInfo   myMethodInfo=type.GetMethods();
//获取属性信息
propertyInfo[] myproperties=type.GetProperties
//获取事件信息
EventInfo[] Myevents=type.GetEvents;
}
}
}
```

反射类型的成员就是反射层次模型中最下面的一层数据。可以通过 type 对象的 GetMembers 方法取得一个类型的成员。如果使用的是不带参数的 GetMembers，它只返回该类型的公共定义的静态变量和实例成员，也可以通过使用带参数的 GetMembers 通过参数设置来返回指定的类型成员。具体参数参考 msdn 中 system.reflection.bindingflags 枚举类型的详细说明。

例如：

```
//设置需要返回的类型的成员内容
bindingFlags
bf=bingdingFlags.DeclaredOnly|bingdingFlags.Nonpublic|BingdingFlags.Public;
```

```
foreach (MemberInfo mi int t.getmembers(bf))
{
writeline(mi.membertype)//输出指定的类型成员
}
```

如果你想要获得一个类型继承的所有接口集合，可以调用 Type 的 FindInterfaces GetInterface 或者 GetInterfaces。所有这些方法只能返回该类型直接继承的接口，它们不会返回从一个接口继承下来的接口。要想返回接口的基础接口必须再次调用上述方法。

用反射来调用类型或者触发方法，或者访问一个字段或者属性时 CLR 需要做更多的工作：校验参数，检查权限等等，所以速度是非常慢的。所以尽量不要使用反射进行编程，对于打算编写一个动态构造类型（晚绑定）的应用程序，可以采取以下的几种方式进行代替：

1．通过类的继承关系。让该类型从一个编译时可知的基础类型派生出来，在运行时生成该类 型的一个实例，将对其的引用放到其基础类型的一个变量中，然后调用该基础类型的虚方法。

2．通过接口实现。在运行时，构建该类型的一个实例，将对其的引用放到其接口类型的一个变量中，然后调用该接口定义的虚方法。

3．通过委托实现。让该类型实现一个方法，其名称和原型都与一个在编译时就已知的委托相符。在运行时先构造该类型的实例，然后在用该方法的对象及名称构造出该委托的实例，接着通过委托调用你想要的方法。这个方法相对与前面两个方法所作的工作要多一些，效率更低一些。

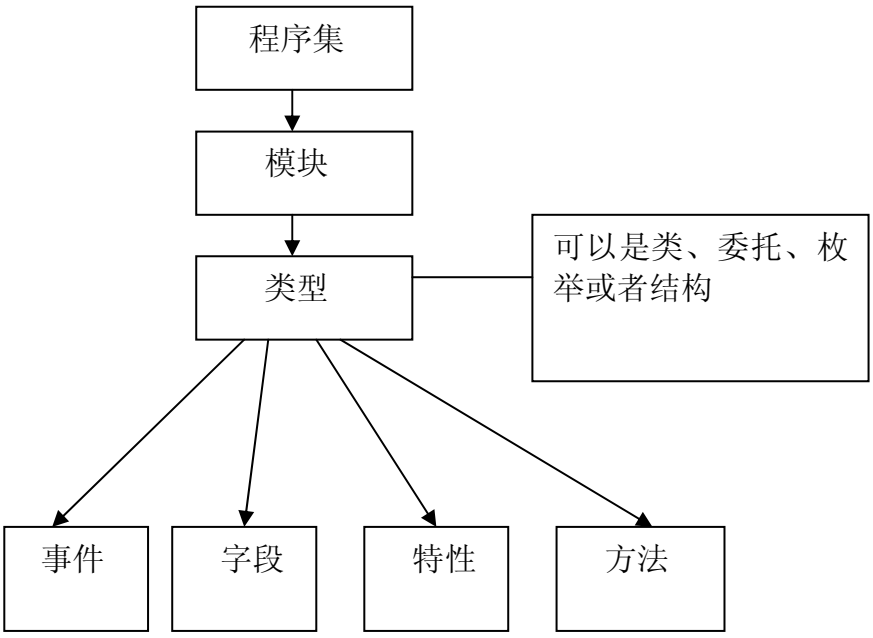

图 2-1　反射的层次结构

第三章　　事务和共享对象

3.1　事务

事务内存是一种共享内存，可以通过事务来对它进行访问。事务是一个有穷指令序列，具有原子性和隔离性。事务的原子性，是指一个事务的诸指令要么全部被执行（事务提交），要么一个也不被执行（事务夭折）。事务的隔离性，是指一个事务看不到另一个事务的中间状态，多个事务的执行互不影响，仿佛它们不是被并行执行而是被顺行执行的。事务由事务内存系统来管理。事务内存系统负责同步各事务对事务内存的访问，使得事务内存一方面像粗粒度锁一样容易使用，另一方面又具有和细粒度锁相当的性能。在基于事务内存的并行程序设计模型中，程序员只需将访问共享内存的代码段标识为事务，同步的细节则由事务内存系统自动去处理。

根据从数据库事务系统获得的灵感，事务是一个代码序列，被封装在一个原子块内。在平面化的事务里，如果子事务夭折，则父事务也夭折；子事务提交，但父事务夭折，子事务的操作结果不出现在父事务之外；子事务提交，父事务提交，子事务的操作结果出现在父事务之外。由此可见，平面化的事务易于实现，但其组合性差。为了弥补这一点，出现了嵌套事务，允许事务进行嵌套和回溯访问。

SXM 系统中的 XState 首先定义了一个状态类，里面预定义了一个事务所要具备的元素。包括其默认等待时间，优先级，用于判定的布尔函数等。接着创立了一个新事务的信息，定义了提交、丢弃、祖先、验证和冲突的布尔函数，以监控事务的状态及对事务状态的判断。在做出判断以后，返回事务状态值。例如，在判断事务是否与其它的有冲突时，如果此事务是一个父类，则返回错误值；接着检查事务处于默认的三种状态中的哪种，如果是活动的则返回 true，其它的则返回 false。其中还有处于非正常状态时抛出的异常。

而在 XAction 里则主要是运行事务和及时更新各项统计环境变量值。首先定义了一个名为 XAction 的抽象类，其中包括运行结果出统计值的有关变量（Total，Commits，Retries）及其初始化为 0，线程何时应该结束的的判断量的定义，由此引出 Retry 函数的建立。接着为主函数 Run，访问共享对象的代理（在 Factory 中包装），参数包括了 XState 类型的变量及对象代理列表。事务在运行中会遇到以下二种可能性：优先级最大的先处于活动状态并正常提交，然后推出新的事务进行执行，更新 Total 及 Commit 的统计值；被迫中止的事务，会捕捉其异常，选择直接丢弃或重试，更新 Total 及 Retries 的统计值。此更新在整个运行过程中处于自动状态，在每个事务运行完毕或中止后都会运行，如下所示。

```
If (current == null)   // top-level
```

```
        {
            UpdateStatistic(ref commits, myCommits);
            UpdateStatistic(ref total, myTotal);
            UpdateStatistic(ref retries, myRetries);
        }
```

3.2 共享对象

对象粒度以面向对象语言中的一个对象作为粒度。大多数的STM系统采用对象粒度，将元数据存储在对象之中。字粒度（块粒度）以一个内存字（内存块）作为粒度。大多数的HTM系统采用字粒度，将元数据和内存字一起存放在高速缓存行中。在STM系统采用字粒度则需要另外建立一个表来记录映射关系。字粒度的优点是粒度更细，对程序员透明。为了方便起见，在没有必要区分时，以后总是将一个共享的存储单元称为“对象”。

SXM 系统里对共享对象的实现主要依赖于事务对象的包装（Factories），早先的软件事务内存系统在同步的包装下关闭共享的数据，只能外在地打开，一个单调且可能出错的过程。为了避免阻塞或是输出错误的结果，在对象外部加了一层能够记录和识别操作的“包装”。这个“包装”可以是一个表加计数器，来记录可能对此对象进行操作的事务，并且记录当前对象上的操作及其施加操作的事务，这时有别的事务想对此对象操作，竞争管理系统就会读到包装上记录的信息，来判断是否可以进行此操作，会不会有冲突等。被包装后的对象即为共享对象。SXM 对一个对象的包装集中在 Factory 代码中，SXM 目前支持两种 factory：TMemFactory 用非常短而严格的部分来模拟硬件事务内存；OFreeFactory 用拷贝（copy）和比较交换(compare-and-swap)的调用结合来提供非阻塞的同步并发。这里鼓励使用者写出自己的 factory 并实验。

TMemFactory设立了如下的同步机制，定义了一个写者，多个读者，因为两个写者就会引起冲突，而有可能多个读者可以同步进行，这也是代理对象可以用来实现避免冲突的机制之一。这也属于对共享对象的包装。

```
public class SynchState
        {
            private const int READERS = 8;
            private const String FORMAT = "Unexpected transaction state: {0}";
            XState writer;
            IContentionManager manager;
            int readCount;
            XState[] readers;
            IRecoverable myObject;
```

```
}
```

TMemFactory 和 OFreeFactory 都支持 Benchmark 里的所有数据结构，例如 nested，当冲突对象被子事务访问而没有被父母事务访问时，它可以在不丢弃自己双亲的情况下丢弃子事务。

如果一个事务创建了一个对象，并修改它，丢弃它，然而这个对象将继续以原始状态存在，但所有的修改会被丢弃。大部分这样的对象会被保留而不显现，但有可能被捕捉到异常。

而在 factory 里对共享对象的访问有一个重要方面就是对设置的公共类的访问，类作为共享对象，怎么被以后的类及函数利用，会很大影响到整个系统的效率速度。在 C#语言里，反射机制很好的解决了这个问题，能够方便地访问使用前面定义的类。

```
const string usage ="usage: SXM.Driver -b benchmark [-m manager ][-f factory] [-t
                        #threads] [-n #millis] [-e experiment#]";
    static void Main(string[] args)
        {
            int numThreads = Default.THREADS;
            int runtime    = Default.TIME;
            int experiment = Default.EXPERIMENT;
            string managerName = Default.MANAGER;
            Type managerType = Type.GetType(managerName);
            string factoryName = Default.FACTORY;
            Type factoryType = Type.GetType(factoryName);
            string benchmarkName = "UNKNOWN";
            Type benchmarkType = null;
```

上段为 Driver 里的一段代码，这段代码就充分表现了 C#中反射机制对对象共享的作用，属于类的类型（Gettypes）。从第一行可知，程序运行时可进行设置，选择想用的参数，以 manager 举例，Type.GetType(managerName)中给 Type manager 赋值，用户输入的 managerName 为一个字符串，在七个选择项中选择，比如 GreedyManager，如果在 C 语言中，就没法识别，只当做字符串来处理，就会引起错误。而在反射机制下，系统得到用户输入的字符串后，就会全文扫描，寻找与用户输入字符串相匹配的类，此例中就会搜在 Manager 下定义的 GreedyManager 类，就能成功赋值给 managerType。这样 managerType 就成功被定义成一种 manager 的类，在 Driver 中启动此类进行相关操作。此反射机制在本系统中有过多次使用，在提高系统性能方法有着很大的作用。

3.3 事务与共享对象间关系

SXM 在对事务对象的访问上，采取了对对象进行包装来决定访问的情况，这也是避免冲突的关键之一。包装，顾名思义，就是不直接去访问资源而是改而访问资源的包装。在一个时间，会有多个事务对它同时进行读写操作，有三种读写冲突在这都会引起，为了避免阻塞或是输出错误的结果，在对象外部加了一层能够记录和识别操作的“包装”。这个“包装”可以是一个表加计数器，来记录可能对此对象进行操作的事务，并且记录当前对象上的操作及其施加操作的事务，这时有别的事务想对此对象操作，竞争管理系统就会读到包装上记录的信息，来判断是否可以进行此操作，会不会有冲突等。

事务对共享对象的访问读写是通过两个函数执行的，OpenRead()和 OpenWrite()。在打开读和打开写的函数里同时包括了访问对象的“包装”，即共享对象包括的读写记录和信息，看是否会发生冲突再决定是否进行操作，或等待及丢弃。

SXM 系统中的包装信息可以说很好的解决在对共享对象读写时各事务竞争冲突的问题，它们在 Factory 目录下，分为 IFactory，OFreeFactory，TMemFactory。

IFactory中的代码

```
public interface IFactory
{
  object Create(params object[] args);
 }
```

把IFactory定义为一个公共接口，OFreeFactory和TMemFactory为它的子类，通过IFactory接口实现完成其功能。

```
public class TMemFactory: IFactory
{
  private const int READERS = 8;
  Type type;
  Type proxyType;
 }
```

拿 TMemFactory 为例，接口 IFactory 可以直接使用此类。上段代码中，类里默认了 8 个读者，如果更多，下面会自动增加。注意到，系统中实际要访问的对象为 Type 类型，这里又加了一个 proxyType 类型，即为 Type 的代理，在读写者要用到对象对其实方向的是对象代理。拿 List 来说，比如有一个结点 Node，会新加一个 new Node，事务在 new Node 上操作，以下为 List.cs 中的一句代码：即上面所说的创建新结点。

```
Node newNode = (Node)factory.Create(v)
```

新结点会增加维护一个域，记录会对它进行读写的事务，和当前的操作，

这时别的事务想要读写它时，它会比较是否会与当前正在进行的操作引起冲突，而接受或者拒绝，而遵循的规则在 manager 里任选一种。

第四章　　冲突检测

4.1　冲突检测的概念

现在的冲突主要有三种类型：W－W，丢失修改；W－R，读脏数据；R－W，不可重复读。而引起这些冲突就是读者和写者之间的活动冲突，所以需要记录读者和写者，想办法避免它们引起此三类冲突。这解决两个事务之间的冲突的办法是使其中一个夭折。记录的方式主要有三种：1.由对象记录。由于写具有排它性，只需要记录一个写者。目前大多数的TM系统中对象都不记录其读者；2.由事务记录。事务记录自己的读集和写集。这些集合可以是私有的（不可见的），也可以是公开的（可见的）；3.由事务内存系统记录。记录事务及其打开的对象。

而冲突的检测，很重要的一点就是抓好检测的时机，在事务内存系统中，一般有三种比较合适的时机：1.在打开时检测，这时事务企图读或写一个对象；2.在确认时检测，这时事务要检查它先前读或写的对象，看看它们有没有被别的事务修改。在事务执行期间，确认次数不限；3.在提交时检测，这时事务要做最后一次确认。

冲突检测的方法分为早检测和晚检测。早检测即为在打开或者确认时检测。早检测能减少并发执行的机会。例如，T1 和 T2 共享 O1，T2 和 T3 共享 O2，若 T2 夭折，则 T1，T2 都可以提交，但是可能的执行结果是 T1，T2 都夭折。晚检测为在提交时检测。晚检测在事务夭折时将丢弃事务所有的计算结果。晚检测有利于短事务。

在并发程序中，当一个事务写另一个事务读或者写的对象时，就会发生冲突，系统检测到冲突之后，就会采取行动来解决冲突，这就是并发控制。在悲观并发控制中，上述动作在同一点发生，也就是系统在事务打算存取一个对象时先检测有无冲突，若有就解决之。这种并发控制方法相当于使用锁来保证事务对对象的独占。在乐观并发控制中，要到事务提交时才检测和解决冲突。这种方法允许多个事务并发存取对象，适合于冲突较少的情形。目前，大多数 STM 和 HTM 系统使用的是乐观并发控制。

在 SXM 系统中的冲突检测就是在实际读写对象之前的，在 OpenRead 和 OpenWrite 中，就会访问共享对象的包装，其中就有了是否会有冲突的判断。可以说，SXM 系统中在对象打开前就进行冲突检测。然后再进行比较，选择执行及丢弃某个事务。

4.2　共享对象的读写

当一个类的成员函数要读谢共享对象时，成员函数可能修改共享对象，这会导致三种冲突之中的一种或几种，这三种冲突是：读-写冲突，写-读冲突，写-写冲突。为了解决这个问题，在FACTORY中创建了一个类SynchState，这个类

以下几个变量：

```
XState writer;
IContentionManager manager;
XState[] readers;
```

Writer 代表共享对象的写者，reader 代表一个读者数组，manager 代表了冲突时的解决冲突的策略。

由于一个对象的写者只可能存在一个，而不可能有两个同时存在，所以写者只需要用一个变量就可以表示。而读者则有可能存在多个而不会产生冲突。

一个成员函数中的事务要读一个共享对象前需要监测是否会产生冲突，这个 SynchState 中的 OpenRead()函数实现的。OpenRead()首先验证事务 me 是否已经结束，如果这个事务没有结束，就对监测这个事务 me 是否与写者属于同一个方法，因为一个方法的事务是存在于一个事务链中，如果方法已经对共享对象有读的权限，就不需要再 OpenRead，这个由函数 me.Ancestor(writer)完成。当 me 与 writer 不属于同一个方法时，需要对 writer 的状态进行一个比较，如果 writer 的状态是 XStates.ACTIVE，这就监测到写读冲突，把 writer 作为冲突对象返回。如果这样没监测到冲突，就把读者加入到读者数组之中，这里可能会超出数组长度，采用的办法是数组长度增加一倍，然后把新的读者存入数组。由此可见，打开一个对共享对象的读方法只能产生一种冲突写读冲突，如果要打开一个共享对象的写方法就有可能产生读写和写写冲突。在 OpenWrite 中，实现了对这两种冲突的监测，首先监测事务 me 与读者数组的冲突，这里用一个循环监测数组中的每一个读者，调用函数 me.Conflict(readers[i])，如果产生冲突，返回事务 readers[i]。接着检测 me 和 writer 之间的冲突，如果 writer 的状态为 XStates.ACTIVE，返回 writer 对象。

4.3 冲突检测的方法

一个普通对象通过 Factory 的包装将成为一个共享对象，这个共享对象可以避免三种冲突，满足事务的特性。Factory 产生新的对象能自动冲突监测。SXM 实现了两种 Factory，一种叫 TMemFactory，另一种叫 OFreeFactory ，下面以 TMemFactory 为例分析 Factory 怎样来包装一个对象的。首先利用了反射机制把原对象的原有属性构造出来。这样就为把原对象的构造函数更改成一个新构造函数，这个新构造函数的产生是由函数：MakeConstructor(TypeBuilder typeBuilder, ConstructorInfo baseCtor)获得，新构造函数出来把原构造函数的一些属性构造出来后，并添加 了一些新属性，一个就是类 SynchState，把这个类构造出来，这个类就是为监测冲突产生的共享对象状态。还有一个新属性是解决冲突的管理策略：Managers。有了这两个新属性就能检测冲突并能用 Managers 解决冲突。

在新对象的构造函数产生后，把对象的所有方法分成两类：读对象的方法和写对象的方法。对所有读对象的方法调用一个函数 InterceptReader(MethodInfo method)，对所有写对象的方法调用才 Writer。在 InterceptReader 将会使用 SynchState 的方法 OpenRead()，并根据 OpenRead() 返回的冲突对象，调用 Managers 的接口：ResolveConflict()，用来解决冲突。同样 InterceptWriter 会调用 OpenWrite()，并解决冲突。

就这样一个普通对象经过包装后，既保持了原有对象的属性，又能实现事务内存的性质。

第五章　竞争管理

5.1　竞争管理接口

竞争管理的部分在 manager 目录下,其中 IContentionManager 即为其中的接口。其中并没有做具体的实现，具体竞争解决在其六个策略中实现。

```
public interface IContentionManager
{
    void ResolveConflict(XState me, XState writer);
    int Priority
    {
        get;
    }
};
```

此代码段先把 IContention 定义为公共类的接口，创建函数 ResolveConflict 定义两个参数自己和写者，设立一个优先级。后面六个冲突策略都是按照这个方法对于两个事务（即参数）给予解决方法，其中一个等待，丢弃，或两个都是。

5.2　竞争管理策略

检测到冲突后到底夭折哪一个事务取决于竞争解决策略。竞争解决策略通常由事务内存系统中的竞争管理器负责实施。竞争解决策略影响系统的性能，如图 5-1，选择不同的策略、在不同的线程数下，各种策略的实用性和效率区别很大。但它必须应该能够保证系统能向前前进。已经提出了多种竞争解决策略。研究表明这些策略各有所长，但没有一个能够全面地压倒另一个。这样就产生了一个问题：是由程序员来指定竞争解决策略，还是由事务内存系统自动选择竞争解决策略。

SXM 系统在 Managers 中，明确列出了其中的冲突管理策略，程序员可以自行选择：

→AggressiveManager

→BackoffManager

→GreedyManager

→PassiverManager

→PriorityManager

→WaitManager

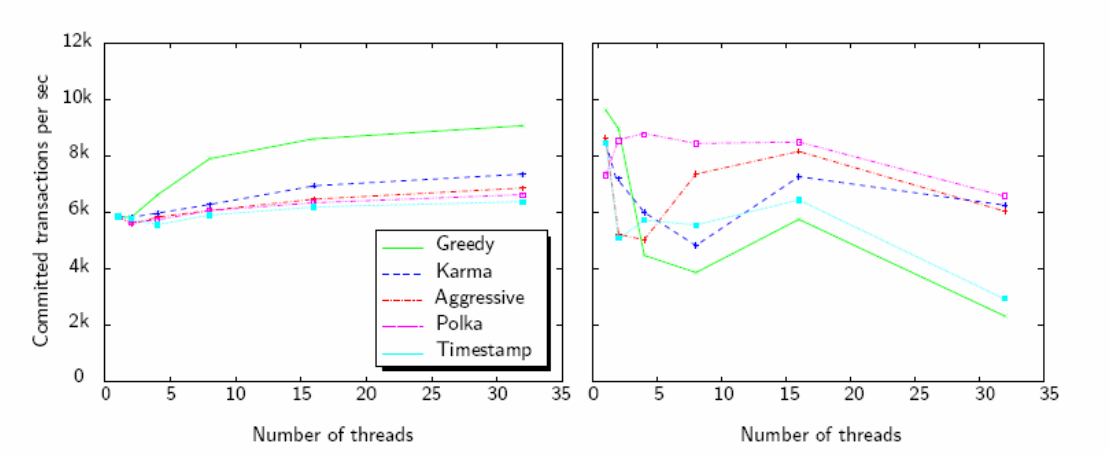

图 5-1　竞争策略比较图

由图 5-1 可见，SXM 系统里提供的几种非阻塞竞争管理策略的效率都较高。例如其中的 GreedyManager.cs，贪婪管理法，企图维护一个最大的事务运行集。由下面的代码可见：如果先前的事务是活动的而且有高的优先级，就等待；如果先前的事务是活动的而且优先级较低，则丢弃；如果先前的事务正在等待且有高优先级，就采用它的优先级且丢弃它；如果先前的事务正在等待且有低的优先级就丢弃它。其中，当前事务如果进入等待，如果其它事务出现 XState 中函数 BlockWhileConflictAndNotWaiting 的状态，则当前事务不再等了。从中可以看出，具有越高代号的事务如果具有越高的优先级则会少很多麻烦。

```
public void ResolveConflict(XState me, XState other)
        {
            if (other.Waiting || other.priority < me.priority)
            {
                other.Abort();
            } else
            {
                me.Waiting = true;
                me.BlockWhileConflictAndNotWaiting(other);
                me.Waiting = false;
             }
            }
```

如果两个事务访问同一个数据对象或一个访问正在被另一个修改的对象，这两个事务会引起冲突。事务并发在 SXM 是乐观的：只有当没有别的事务存在冲突访问执行时，另一个事务完成时才会提交。

如果事务 A 发现马上要与事务 B 冲突，就有个选择：可以暂停，给事务 B 机会完成，或者继续进行，逼迫事务 B 放弃。对于这个选择，就可以问当地的

冲突管理内核 contention manager 看做哪个选择。

SXM 系统里包括了 contention manager 的各协议策略，从很简单的策略如带潜力的回溯到优先表。可以看出，对于 contention manager 算法的选择会充分影响到事务的吞吐量。SXM 提供了几种可选择的 contention manager 策略（执行方法），你可以自由选择你想要的。

SXM 里的 manager 部分主要负责了六种冲突解决的方法，根据执行者的意愿任意选择一种解决的方法。以 AggressiveManager.cs 为例，有如下代码段：

```
#region IContentionManager methods
        /// <summary>
        public void ResolveConflict(XState me, XState writer)
        {
            writer.Abort();
        }
        public int Priority
        {
            get {
                return 0;
            }
        }
    }
        #endregion
```

注意到公共类中的函数 ResolveConflict 中引用的两个参数（me and writer）都是 XState 类型的，从 XState 中继承了类的基本特性。而以下代码是 XState 处理冲突的类结构。先从这分析对事务三种活动的跟踪处理。

```
public bool Conflict(XState other)
        {
            // don't conflict with ancestors
            if (Ancestor(other))
            {
                return false;
            }
            switch ((XStates)other.State)
            {
                case XStates.ACTIVE:
                    return true;
```

```
                case XStates.ABORTED:
                    return false;
                case XStates.COMMITTED:
                    return false;
                default:
                    throw new PanicException("Unexpected XState: {0}", other);
            }
        }
```

此段主要是检测事务的状态类型，先看是否与其祖先有冲突，如果有则返回错误。其后就是看事务状态：如果事务是活动的，返回正常：如果事务是被丢弃或者已提交，则返回错误。系统默认错误时丢出Panic异常（Unexpected XState）。

```
        public void BlockWhileConflictAndNotWaiting(XState other)
        {
            XState root = other.Root;
            Monitor.Enter(root);
            try
            {
                while (Conflict(other) && !other.Waiting)
                {
                    Monitor.Wait(root, TIMEOUT);
                    if (XAction.stop)
                    {
                        this.Abort();
                        return;
```

就像公共类函数名称一样，此段的内容是当有冲突时阻塞而且不等待。此函数的参数other为XState类型，如果other与其它事务有冲突且没有事务等待时，root将一直等待直到超时，即为阻塞状态。如果事务因此停止，就丢弃它。

```
        public void BlockWhileConflict(XState other)
        {
            XState root = other.Root;
            Monitor.Enter(root);
            try
            {
                while (Conflict(other))
                {
```

```
                Monitor.Wait(root, TIMEOUT);
                if (XAction.stop)
                {
                    this.Abort();
                    return;
```

此段代码与上段形成比较，设置函数BlockWhileConflict，当有冲突时阻塞。

```
    public void BlockWhileActive()
    {
        XState root = this.Root;
        Monitor.Enter(root);
        try
        {
            while (this.State != XStates.ABORTED)
            {
                Monitor.Wait(root, TIMEOUT);
                if (XAction.stop)
                {
                    this.Abort();
                    return;
```

这段代码设置公共类函数 BlockWhileActiver( )，当此事务状态不是因为另一个事务而被丢弃时，将会在指定时间内等待着被调用，而如果活动状态因为另一个事务的竞争而变停止则丢弃此事务。

由此可见，manager 的 Aggessive 方法就是通过把自己这个事务和另一个写者设为 XState 类型，当两方有冲突时，直接丢弃写者。

第六章　总结

在本文中，首先介绍了多核计算机的出现，引起并发程序设计的难题。锁的不适用，非阻塞的要求，从数据库事务的概念引出了事务内存及其相关概念。它将数据库中事务模式的核心思想移植到编程语言里来。程序员将自己的程序编写为一个个具有确定原子性的块，它们可以分离执行，因此只有在每个原子行为执行前和执行后，并发操作才能访问共享数据。它可以解决并发程序设计难的问题，避免了锁的同步困难，有良好的发展前景。

随着多核处理器投入商用，对事务内存的研究也变得空前活跃起来。近几年中涌现的一批新的事务内存系统，就包括了软件事务内存系统 SXM。本文在 Visual Studio.net 2005 下运行 SXM 系统，介绍了参数的设置及运行的结果显示，显示数据的意义主要是为了看出 CPU 事务处理的效率。

SXM 系统中，先要了解事务是一个有穷指令序列，具有原子性和隔离性。事务的原子性，是指一个事务的诸指令要么全部被执行（事务提交），要么一个也不被执行（事务夭折）。事务的隔离性，是指一个事务看不到另一个事务的中间状态，多个事务的执行互不影响，仿佛它们不是被并行执行而是被顺行执行的。事务由事务内存系统来管理。事务内存系统负责同步各事务对事务内存的访问。XStates 和 XAction 定义了事务及其变量，事务要访问的共享对象则由 Factory 负责包装，使记录可能对此对象进行操作的事务，并且记录当前对象上的操作及其施加操作的事务，这时有别的事务想对此对象操作，竞争管理系统就会读到包装上记录的信息，进行相关的操作。而事务对共享对象的读写主要是依靠 OpenRead 和 OpenWrite 两个函数。

多核计算机中，多个事务要对共享对象同时进行读写，必然会引起冲突。SXM 的一个重要内容就是对冲突的管理，以达到减少冲突的目的。于是先有了冲突检测，在 SXM 系统中的冲突检测就是在实际读写对象之前的。一个普通对象通过 Factory 的包装将成为一个共享对象，这个共享对象可以避免三种冲突，满足事务的特性。Factory 产生新的对象能自动冲突监测。SXM 实现了两种 Factory，一种叫 TMemFactory，另一种叫 OFreeFactory。TMemFactory 把原对象的构造函数更改成一个新构造函数并添加了一些新属性，一个就是类 SynchState，把这个类构造出来，这个类就是为监测冲突产生的共享对象状态。在 OpenRead 和 OpenWrite 中，就会访问共享对象的包装，其中就有了是否会有冲突的判断。可以说，SXM 系统中在对象打开前就进行冲突检测。然后再进行比较，选择执行及丢弃某个事务。还有一个新属性为 Managers。有了这两个新属性就能检测冲突并能用 Managers 解决冲突。

Managers 就是解决冲突的管理策略，SXM 系统提供了六种竞争管理策略和竞争管理的接口 IContentionManager。使用哪个策略可由用户自己指定，选择不同

的策略、在不同的线程数下，各种策略的实用性和效率区别很大，但它必能够保证系统能向前前进。如 Manager 的 Aggessive 方法就是通过把自己这个事务和另一个写者设为 XState 类型，当两方有冲突时，直接丢弃写者。

通过研究 SXM 系统，明白了基于 STM 基础上的利用，里面很多概念如非阻塞、高效等都在 SXM 里得到了很好的体现。SXM 里提供一个具体的利用事务内存的范例，通过学习里面的编程思想，对 STM 思想有了进一步的认识。

参考文献

[1]Herlihy.M and Moss.J. Transactional memory: architectural support for lock-free data structures. In Proceedings of ISCA '93, 1993.

[2]Shavit.N and Touitou.D. Software Transactional Memory. In Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, 1995.

[3]Guerraoui.R, Herlihy.M, and Pochon.B. Polymorphic Contention Management. In Proceedings of the Nineteenth International Symposium on Distributed Computing, 2005.

[4]Diomidis Spinellis [著]，赵学良[译]，代码阅读方法与实践，清华大学出版社，2004 年 3 月.

[5]Valois.J. Lock-Free Linked List Using Compare-and-Swap. In Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, Pages 214-222, August 1995.

[6]Kung.H and Robinson.J. On Optimistic Methods of Concurrency Control.ACM Transactions on Database Systems, 6(2)：213—226, 1981.

[7]Herlihy.M. Wait-Free Synchronization. ACM Transactions on Programming Languages and Systems, 13(1)：124—149, 1991.

[8]Herb Sutter. The Free Lunch Is Over：A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal. Vol 30. No 3. March 2005.

[9]Ramalingam.G. Context-sensitive synchronization-sensitive analysis is undecidable. ACM TOPLS 22(2)：416–430, 2000.

[10]Moss.J and Hosking.A. Nested Transactional Memory：Model and Preliminary Architecture Sketches. in Proceedings of the OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL), 2005.

[11]Michael Greenwald, computer & information Science Department. Two-Handed Emulation: How to build non-blocking implementations of complex data-structures using DCAS.

[12]Ｔ.Harris, S.Marlow, S.Jones, and M.Herlily. Composable memory transaction. Technical report, Microsoft Research Cambridge, December 2004.

[13]W.Weihl. Specification and Implementation of Atomic Data Types. PHD thesis, MIT, 1984.

[14]L.Hammond, B.Nayfeh, and K.Olukotun. A single-chip multiprocessor. Computer, 30(9): 79-85, 1997.

[15]C.papadimitriou. The serializability of concurrent database updates. Journal of the ACM, 26(4):641-653, 1979.

[16]P.Schwarz and A.Spector. Synchronizing shared abstract types. ACM Transaction on Computer Systems, 2(3):223-250, 1984.

附录

SXM：这个用 C＃编写的程序分为以下几个部分：

1. Attributes 定义了三个特性。AtomicAttribute 定义了操作的原子性及一个公共类；ReaderAttribute 定义属性了读属性也建了公共类；WriteAttribute 定义了一个写者公共类。
2. Exceptions 包括四种呼叫异常。Aborted 是指当事务被发现处于中止状态时丢弃；Graceful 指发现竞争时中止此事务；Panic 指当内部系统出现错误时丢弃事务；Retry 时出现事务阻塞时丢弃。
3. Benchmarks 是执行时的选择性特性之一。①Factory 试着建立几个几个非法事物类，包括四个代码块 Members，Constructors，Benchmark overrides，Atomic classes。Members 中 factory 定义为 IFactory(后面会有介绍)类变量；Constructors 代码块中建立了 Factory 类，主要用来捕捉异常判断 Passed 或是 Failed；Benchmark overrides 中又定义了 Run，Validate，Report 三个函数子类；Atomic classes 中定义了 5 个错误类。②Buffer 建立了有　　固定大小的缓冲区，是对 Retry 的被动测试。先有一个 fields 的代码块，定义了这个缓区的初始大小，三个私有变量（putCalls getCalls delta）及两个私有 XStart 子类；Constructors 代码块包括了放入及移出缓冲区的规则和限制；methods 代码块主要负责 2 个功能，一是把一个 value 放入缓冲区，如果满了就重试，二是从缓冲区拿出一个 value，如果缓冲区是空的就重试；Benchmark overrides 代码块负责检查缓冲区内的各元素是否被排列好且占了正确的位置大小，这种方法被每个并行的线程所使用，在每个操作完成后自动进行更新统计列表；Atomic classes 代码块包括关于缓冲区的一些主要项目，定义了受保护的变量 capacity（缓冲区现在的容量），size（item 占的地方），AtomicArray buffer。③IntSetbenchmark 在 fields 代码块里设定受保护成员 insertCalls，removeCalls，containsCalls，delta 的设定，给事务对象建立代理 factory，再设定 XStart 的三个保护成员 insertXStart，removeXStart，containsXStart，以便在随时通过反射机能调用。在 abstract methods 代码块中，用公共类对这些项目数值进行操作,增加检查移除排列等；Benchmark overrides 代码块中主要分以下几个内容，先是检查每个数集是否排列好并分到合适的空间，而后显示出 Calls，Insert，Remove，Contains 的值，然后把这种方法应用到全部的线程，其中包括了捕捉异常，最后在每次操作结束自动地刷新显示的 Calls，Insert，Remove，Contains 的值。④List：一种简单的线性表数集的 benchmark。⑤Nested：包括一个对聚集事物类的测试，用了 Red-BLACK 树中的特性，做一个无偿的聚集。具体过程是先建立 RB 树的根，把 sentinelNode 作为声明一个叶结点的途径并归为 RBNode 的一个类，再使树被每一个线程共享。建立 sentinel node，其中

sentinel node 是成功的关键，再就是一些具体操作。⑥OrElse：⑦RBTree：利用 RB 树特性的 benchmark。⑧SkipList：利用数据结构中的跳表特性，规定了跳表中可存在结点的最大等级及其决定元素，可以做为 NIL 结的跳表头，可以产生随机结点等级的随机产生器和当前最大的表等级。

4. Factory：对事务对象的包装。①IFatory：一种事物对象包装接口。②OFreeFactory:非阻塞对象包装，基于 DSTM。③TMemFactory：建立事物对象的包装类，用了反射机能，保护每个线程必须是安全的。TMemFactory 看似用了硬件事物机制，因为 HW 事物在此不被支持。
5. Managers：负责冲突管理机制，有改下 7 种管理策略。①Aggressive：管理机制总会丢弃冲突的事务,不容易产生死锁,不过在实际中不常用。②Backoff：一种有效的回退方法，关键看事务被怎么放置。③Greedy：贪婪管理法。试着维持着最大依赖集，如果有优先权的事务在等待低优先权的，就丢弃它，否则等它提交、丢弃或等待。④IContention 如果事务 A 将要引起与事务 B 的冲突，Contention 管理系统将会调控到底是 A 等 B，丢弃 B 还是两个都执行。⑤Passive：被动策略。内存管理从不丢弃冲突的事务，跟锁是等效的，还是会有死锁，也不常用于实际利用。⑥Priority：优先管理法，如果优先事务有比较迟的时间片，就丢弃它，或者等待。⑦Wait：等待管理法。如果先前的事务是处于活跃状态的，就等待；如果它是等待态的，就丢弃它。
6. AssemblyInfo：用其中的各属性控制一些关于聚集的主要信息，通过修改这样关于集合的信息来改变属性性。
7. AtomicArray：提供原子性的似队列类，当你需要一个原子类中队列域时可以使用它。当打开时，当前整个队列都会拷贝保存下来，可以对于大的队列做一些改进。
8. Benchmark：一个简单的抽象类，可以建立统一的基准。定义了类中的 experiment 常项，和三个抽象的公共函数 Run 运行，Validate 提交，Report 报告，分别用来表示每个线程运行方法，检查和显示统计数据。
9. Default：里面定义了 SXM 程序运行时的一些默认设置值。如 THREADS=1，TIME=5000，EXPERIMENT=1，MANAGER=Greedy，FACTORY=TMemFactory。
10. Driver：运行 SXM benchmarks 的主程序。主要功能是驱动作用，启动其它各段程序，从 shell 里读程序，开始和停止线程，和显示运行结果。
11. XAction：最主要的事务类，负责事务的运行和跟踪事务的各环境变量。
12. XState：设置不断变化的对象，以跟踪事务的状态。
13. XStates：定义可能的事务状态。分三种，Aborted(丢弃)，Active（运行），Committed（提交。

致谢

首先我要感谢我的导师张坤龙老师在我做毕业设计阶段对我孜孜不倦的教导，在我迷茫时指导我，在我失败时鼓励我。

我还要感谢我的同学对我的帮助，尤其是胡正军和李明同学在我毕业设计期间对于我的课题给了我很大帮助。

最后，还有我的父母和所有支持我的朋友们对我的信任和鼓励！