RSTM 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 李　明

指导教师 张 坤 龙

2007 年　**6** 月　**15** 日

摘　要

随着多核计算机的发展，并行程序设计越来越受到关注。然而，并行程序设计有很多地方不同于传统的串行程序设计，其中最重要的是基于共享内存的同步机制。由于传统的同步机制锁有许多应用上的缺点，研究人员提出了无锁的事务内存方法，该方法受到了越来越多人的重视。

事务内存把数据库中事务的概念引入到程序设计中，使得在程序中可以将一系列对内存的访问封装成一个原子操作，从而简化并行程序设计。

RSTM 系统是众多事务内存系统中比较完善的一个，它采用 C++语言设计和实现。为了能更深层次地理解事务内存，论文对 RSTM 系统的源代码进行了分析。论文首先介绍了 RSTM 系统的编程接口，然后描述了 RSTM 系统实现时采用的重要数据结构，接着给出了 RSTM 系统的主要算法，最后分析了 RSTM 系统自带的基准测试程序。

关键词：事务内存；RSTM；并行程序；源码分析

目　　录

第1章　　绪论

1.1　多核处理技术

多核处理技术，也称芯片多处理技术（CMP，Chip Multi-Processing），就是在一个处理器基板上集成多个功能相同的处理器核心。处理器实际性能是处理器在每个时钟周期内所能处理指令数的总量。由于每增加一个内核，处理器将增加一套执行单元和寄存器等资源，所以处理器性能也就相应地大幅度增加。正因为如此，全球芯片巨擎莫不投入巨资于多核处理器的研发上。那么，多核处理技术的主要优势究竟是什么呢?

1)　多核处理技术能有效提高性能

相比以前的由分立处理器构成的多处理技术，多核处理技术由于将多个 CPU 放在一块硅片上，当工程技术人员在两个 CPU 内核之间传送数据时，可以利用更短的距离和更快的总线速度的优势达到更高的性能。测试证明，CMP 性能超过 SMP 性能 50％以上。与超线程技术相比，多核处理器中每个核心拥有独立的执行单元和寄存器等资源，可以真正并发执行多项任务，其效率要比超线程技术高得多。

2)　多核处理技术能有效降低功耗

现代的 CMOS 微处理器，能量主要消耗在晶体管状态的转换上。每个晶体管需要的能量正比于晶体管的负载电容、转换频率和电压的平方三者的乘积。多核处理器可以用较低的频率达到单核处理器使用更高频率才能达到的性能，因此也就能有效降低功耗了。

3)　多核处理技术能有效节约成本

对于今天的芯片制造商来说，封装和测试占总成本的四分之一到一半。多核处理器共享相同的封装和 I/O（I/O 有时能占到芯片的四分之一面积），显然可以有效地降低成本。

4)　多核处理技术能有效增加功能

多核处理技术能有效增加单核处理器所难以具备的某些功能，最典型的当属虚拟技术。虚拟技术可以让一台物理计算机虚拟出若干个独立的系统，这些虚拟系统能使用同样的计算机资源独立工作。

多核处理技术是提高计算机性能的创新手段，但是要真正发挥出多核处理技术的性能优势是对软件程序员的严峻挑战，因为他们必须对平台中运行的系统软件进行重新编写，使系统软件能在多个执行内核间妥善分配任务并且并行运行。

1.2 并行程序设计

1.2.1 并行程序设计现状

在过去 30 年里，并行虽然一直被鼓吹为“下一件大事”或“未来之路”，但软件界不为所动。并行程序设计只与少数人相关，不像顺序程序设计那样广为人知并受到大多数软件开发商支持。造成这种现象的主要原因是并行计算机始终未成为通用计算机市场上的主流，并且顺序程序无需修改其性能就能随着单核处理器性能的提升而提升。

现在，情况发生了改变：新一代计算机全面支持并发。这将引发软件开发方式的巨变，对软件至少是主流软件的开发带来深远影响。计算机的能力无疑越来越强，但程序不再可能从硬件性能提升的大餐中免费获益，除非它们实现了并行。即便抛开多核处理技术的因素不谈，我们也有理由实现并行。例如，并行可以提高响应速度，就像目前的应用里必须让工作远离 GUI 线程，以便计算在后台进行时，屏幕能得到重绘。但实现并行是有难度的。不仅目前的语言和工具仍未为将应用转化为并行程序做好充分准备，而且在主流应用中也很难找到并行，尤其糟糕的是——并行要求程序员以人类难以适应的方式思维。不过，多核在未来不可回避，研究人员必须找到与之相适应的软件开发方式。

1.2.2 并行程序设计的难点

尽管顺序程序设计已经很难，但并行程序设计更困难。造成并行程序设计困难的原因有很多，下面列出其中的一些。

由于存在并发和不确定性，并行程序设计比顺序程序设计困难有着心理学上的原因。心理学研究表明，人把注意力发散开来处理多个任务的能力是有限的。例如，即使最细心的程序员，也很可能考虑不到简单半有序作业集里的交叉问题。

分析并行程序比分析顺序程序困难。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。已经证明，即使只有两个线程，将上下文和同步结合起来分析是一个不可判定问题。

在程序设计模型、程序设计语言、程序设计工具等方面，并行程序设计落后于顺序程序设计。在当前的任务并行程序设计模型中，需要使用抽象程度较低的同步机制。

1.2.3 同步机制

先看一个并行程序设计的例子。将加工后的数据送入缓冲区和从缓冲区取出数据打印输出必须依次进行。在数据送入缓冲区前不能打印输出，在缓冲区内的数据没有打印输出完毕时不能输入；否则，一批数据可能被重复打印或者一批数据还没有打印输出就被新送入的数据冲掉。因此，对“送入缓冲区”和“从缓冲区取出数据”两个操作必须加以约制，以保证它们依次执行，否则就会发生错误。产生这个问题的原因是两个进程都要访问缓冲区，也就是说它们有一个公共变量。

在并行程序设计中，各进程对公共变量的访问必须加以约制，这种约制称为同步。进程的同步是通过同步机制实现的。现已有多种同步机制，具有代表性的是锁、PV 操作和管程。锁将在第 1.2.4 节中介绍。

PV 操作是最早提出的同步操作。PV 操作是作用于信号量上的原语。所谓原语是指其执行是不会被打断的，即一个进程在执行 PV 操作时，不会强行地被打断而让处理器去执行另一个进程。PV 操作的定义是：执 行 P 操作 P(S)时，信号量 S 之值减 1，若结果不为负数，则 P(S)执行完毕；否则,执行 P 操作的进程暂时停止。等待释放。执行 V 操作 V(S)时，信号量 S 之值加 1，若结果不大于 0，则释放一个等待释放的进程。有了 PV 操作后，上面例子中的问题就即可解决。

1973 年霍尔提出的管程是另一种重要的同步机制。管程是指一组公共数据同与其有关的操作的集合。只有引用管程中的操作才能访问管程中的数据。一个进程引用管程中的操作时，只有在管程中的各操作均不处于活动状态时才被响应。当管程中的一个操作被引用后，它就成为活动状态。当管程中一个操作已执行完毕或在执行中处于等待状态时，它就不是活动状态。管程将公共数据同与其有关的操作集中在一起，使得并发程序设计易于理解，程序正确性也容易保证。因此，管程有助于同步机制从 PV 操作向前发展。它是并行程序设计趋于成熟的标志之一。

1.2.4 锁

在目前的任务并行程序设计模型中，有大量同步策略可以解决数据竞争问题，其中最简单的就是使用锁。每一个需要访问共享数据片的任务在访问数据前必须申请得到锁，然后执行计算；最后要释放锁，以便其他任务可以对这些数据执行别的操作。

不幸的是，尽管锁在一定程度上能避免数据竞争，但它也给现代软件开发带

来了严重问题。最主要的问题是，锁不具有可组合性。你不能保证由两部分以锁为基础、能正确运行的代码合并得到的程序依然正确。而现代软件开发的基础恰恰是将小程序库合并为更大程序的组装能力；因此，我们无法做到不考察组件的具体实现，就能在它们基础上组装大软件，这是个大问题。

除了不具有可组合性之外，锁还有可能造成死锁等问题。进程因争夺资源而无休止地相互等待称为死锁。例如，进程 P_1 占有了绘图机而申请行式打印机，进程 P_2 占有了行式打印机而申请绘图机。它们都因为申请不到资源而永远等待，这就是死锁。解决死锁问题有两种途径：一是预防死锁，设计各种资源调度算法，防止死锁发生；另一种途径是检测死锁，当死锁发生时能及时发现并进行排除。无论是那种途径，其代价都很高。

1.3 事务内存

为了解决并行程序设计困难的问题，研究人员提出了锁的替代物，既事务内存（Transactional Memory）。事务内存将数据库中事务处理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原子性的块，它们可以分离执行，但只有在每个原子块执行前和执行后，并行操作才能访问共享数据。目前，有很多研究人员看好事务内存的前途。

1.3.1 事务内存与数据库

数据库系统能够有效地适应各种并行环境，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：

a) 原子性(Atomicity)：事务是数据库的逻辑工作单位，诸操作要么都做，要么都不做。

b) 一致性(Consistency)：事务执行的结果必须是使数据库从一个一致状态变成另一个一致状态。

c) 隔离性(Isolation)：一个事务的执行不能被其他事务干扰，即并发执行的各个事务之间不能互相干扰。

d) 持久性(Durability)：一个事务一旦提交，对数据库中的数据的改变是永久性的。

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。

事务内存借鉴了数据库中解决同步问题的思想。1977 年，Lomet 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，但未给出实现。1993 年，Herlihy 和 Moss 给出了一个硬件事务内存系统。1995 年， Shavit 和 Touitou 给

出了一个软件事务内存系统。最近几年，事务内存成为研究热点，出现了一些（软件，硬件、混合）事务内存系统。

虽然事务内存借鉴了数据库中的思想，但事务内存和数据库有许多的不同之处：

(1) 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

(2) 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

(3) 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。

1.3.2 事务内存的编程接口

从程序员的角度来看，事务内存提供 atomic,abort,retry 等新的关键字，其中 atomic 用于将代码片断定义为事务，abort 用于事务的强行中止，而 retry 将使得事务的夭折后重做。下面举例来说明这三个关键字的用法。

```
atomic {
    account1-=100;
    account2+=100;
}
```

atomic表示事务是一个代码序列，被封装在一个原子块内。上面的代码中两个操作被封装在一个原子块内，它们必须原子性的执行，它们的执行结果可能有以下几种情况

a) 事务提交，可改变程序状态，account1与account2均被改写

b) 事务夭折，不改变程序状态

c) 事务不终止，结果无定义

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。这个过程由事务内存系统来实现，对用户是透明的。用户可以使用abort或者retry使事务显式夭折。

```
atomic {
    if (account1<100)
        abort;
```

```
        account1-=100;
        account2+=100;
    }
```

在上面的代码中，使用abort使事务显式夭折，其效果相当于事务未被执行。而同样的，还还可以使用retry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，如下面的代码所示：

```
    atomic {
        if (account1<100)
            retry;
        account1-=100;
        account2+=100;
    }
```

在这里，使用atomic实现互斥，使用retry实现条件同步。

上面提出的几种编程接口并未在RSTM系统中给与实现，但在这里给出这样一种设想，也许在将来的事务内存系统中将会见到类似于上面的代码，那样并行程序设计将会更加的简单易行。

1.3.3 事务内存系统

研究人员很早就注意到了锁引起的各种问题。为了解决这些问题，他们借鉴数据库领域的研究成果，提出了一种新的基于事务内存的同步机制。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。

事务内存根据实现方式可以分为：

1） 软件事务内存（Software Transactional Memory，STM）

2） 硬件事务内存（Hardware Transactional Memory，HTM）

3） 混合事务内存（Hybrid Transacional Memory，HyTM）

在这里，将主要对软件事务内存系统进行研究。

在事务内存系统中，首先需要解决的是冲突的检测与解决。在介绍这个问题之前，首先需要对冲突的含义给出明确的定义。

如果一个事务读过的共享对象被另外一个事务写，或者一个事务写过的共享对象被另外一个事务读或者写，那么就称这两个事务发生了一个冲突。事务内存系统的主要任务就是保证已提交的诸事务间不发生冲突。为了完成这一任务，事务内存系统在检测到两个事务发生冲突时选择其中一个夭折重做。这样从另一个事务的角度来说，就好像没有发生过冲突一样。如何检测和解决事务间的冲突，不同的事务内存系统采用不同的算法。

为了检测到冲突，需要知道一个共享对象上所有的读者和写者。由于写操作和其他所有的操作都是互斥的，所以只需要对每个共享对象记载它的一个写者即可。但是，由于两个读操作是相容的，所以一个共享对象上的读者数目是无法预测的。在目前已有的软件事务内存系统之中，都会修改共享对象的存储结构，以便将写者记载在其中。至于读者，有两种记载方式：第一种方式是在共享对象的存储结构内用链表记载读者；第二种方式是在事务的私有存储空间中用链表记载事务读过的所有共享对象。前一种方式的优点是读者可见，有利于尽早发现冲突，但是为了避免在链表上查找，共享对象每次被读都会增加一个记录，导致需要占据较大的共享内存空间。后一种方式的优缺点正好和前一种方式相反。

在本文下文将对冲突的检测和解决做出详细的说明，在这里不多做介绍。

1.4 课题的意义

RSTM（Rochester Software Transactional Memory）系统是用 C++语言编写的软件事务内存（STM）系统，它采用了非阻塞的数据结构，避免了采用锁所造成的死锁，护航，优先级倒置等问题。通过分析 RSTM 源代码，可以：

1. 更好的理解事务内存系统：RSTM 系统采用了大家都非常熟悉的 C++语言系统，不像其他 STM 系统一样要求读者掌握 C#，Java，Haskell 等语言，它把事务内存与 C++中的多线程编程结合一起，更方便了读者利用已有的知识去了解事务内存系统。
2. 加深对事务内存系统的认识：分析 RSTM 源码可以更直观的看到事务内存系统的实现方式，从而使对事务内存系统有更深层次的理解。
3. 写出更好的并行程序：通过分析和理解 RSTM 系统源码，不仅可以加大对事务内存系统的认识，也可以把 STM 系统和现有的编程系统联系到一起，使并行程序设计更加的简单易行。

1.5 论文的组织

在这篇论文中，将对 RSTM 系统源代码给出具体的分析，从而对软件事务内存系统有更好的理解。

为了使大家对本文的讲述过程有更清晰的了解，在这里，对本文的组织结构给出简单的介绍：

在本文的第二章，将简单的介绍 RSTM 系统的编程接口，通过简单的例子，学会 RSTM 系统 API 的使用方法，从而对 RSTM 编程有一个最初步的理解。

在本文的第三章，将对 RSTM 系统的主要数据结构给出详细的介绍。

在本文的第四章，将介绍 RSTM 系统的主要算法。通过第三章和第四章的介绍，读者可以对 RSTM 系统有更深的了解。为了把第三章和第四章介绍的理论知识和实际的编程结合起来，在第五章我们将给出一些基准测试，从具体的程序设计中理解 RSTM 系统的实现过程。

在本文的第六章，将对所做的工作做一个简单的总结。

第2章　　RSTM 程序设计

2.1　RSTM 的编程接口

RSTM 共有 9 个 API 接口函数供编程使用，它们分别如表 3-1 所示。

表 3-1 RSTM 的编程接口

接口函数	意义
Stm_init()	必须在每个线程的任何事务创建之前被调用
Stm_dest()	当一个线程结束时调用
open_RO()	打开一个对象进行读操作
open_RW()	打开一个对象进行写操作
shared()	获得一个打开对象的事务外壳
release()	关闭一个用 open_RO() 打开的对象
tx_alloc()	从事务内存的堆中获得内存
tx_free()	释放由 tx_alloc 申请的内存
BEGIN_TRANSACTION	开始一个事务
END_TRANSACTION	结束一个事务(必须与 BEGIN_TRANSACTION 在同一嵌套级)

2.1.1　创建共享对象

在使用 RSTM 系统进行数据读写之前，首先需要对一个对象进行封装，使它具有“事务内存外壳”，从而可以在事务中使用 open_RO 和 open_RW 函数来读写对象。创建一个共享的对象，仅仅需要对一个已存在的对象做如下的改变：

1) 继承自 Object<T>：这可以确保对象有 RSTM 程序设计所必须的附加元数据域
2) 用 Shared<T>替换 T ：这可以使对象拥有访问 RSTM API 的权限
3) 写一个 T. clone()方法：这可以让 RSTM 更容易复制对象

2.1.2　创建事务

在 RSTM 系统中，分别使用 stm_init 和 stm_dest 在每个线程的开始和结束时开启和结束使用事务内存系统，它们分别负责事务内存系统的初始化和清理工作。在使用 stm_init 开启事务内存系统后，可以使用 BEGIN_TRANSACTION 和 END_TRANSACTION 来创建一个事务，在这个事务中，可以使用 tx_alloc 和 tx_free 函数来从事务内存的堆中申请和释放内存，同样的，可以在事务中对对象进行读写操作。

2.2 程序设计举例

2.2.1 共享计数器

根据上面的 API 函数，就可以开始事务内存编程之旅。例如想创建一个共享的计数器，这个计数器可以同时被许多线程所访问。为了保证计数器对象可以使用 RSTM 系统 API，首先，需要对它加上一个“事务内存外壳”，即创建一个共享的计数器对象。

创建一个共享对象，根据上一节中介绍的知识，需要：

1. 定义对象 Counter，该对象继承自对象 Object<Counter>
2. 写一个 clone 函数，该函数被定义为虚函数，确保它在被继承时 clone 函数只执行一次；该函数返回一个 Counter *型值，完成对象的复制工作
3. 读写计数器的值

代码如下表所示：

```
class Counter : public Object<Counter>                                    (1)
{
    int value;
public:
    Counter(int startingValue) : value(startingValue) { }
    virtual Counter* clone()                                              (2)
    {
        return new (stm::tx_alloc(sizeof(Counter))) Counter(value);
    }
    void increment() { value++; }                                         (3)
    int getValue() { return value; }                                      (3)
};
```

这样就定义了一个共享的计数器对象，通过它，可以访问 RSTM API 函数，进行我事务内存编程。通过前面创建事务的知识，可以这么创建一个事务：

1. 使用 BEGIN_TRANSACTION 开始一个事务，为了对各事务进行区分，我们使用 counter_tx 来标记这个事务。
2. 调用共享对象的 open_RW 函数修改计数器的值。open_RW 返回一个 Counter*值，通过它我们可以调用 increment 函数修改计数器。
3. 使用 END_TRANSACTION 结束一个事务。

代码如下所示：

```
BEGIN_TRANSACTION(counter_tx);                    (1)
C->open_RW(counter_tx)->increment();              (2)
END_TRANSACTION(counter_tx);                      (3)
```

这样就完成了一个共享计数器的事务操作，首先创建了一个共享的计数器对象，然后对这个共享计数器进行事务的读写操作来完成计数器加 1 的任务，通过事务进行计数器的读写可以确保在多个线程同时访问计数器产生冲突的时候只能有一个进行成功读写。需要说明的是，由于在这里仅仅作为示例使用 RSTM 系统，并未给出所有代码，如前一节说明的那样，在每个线程利用事务来读写对象之前，首先必须先调用 stm_init 来初始化 RSTM 系统，同样的，在线程的结束前，使用 stm_init 来结束事务内存系统。

在介绍过共享计数器之后，下一节继续给出共享链表的代码，并根据它来进一步介绍 RSTM 系统。

2.2.2 共享链表

首先看下面这样一段代码：

```
class LLNode : public Object<LLNode>
{
public:
        int val;
        Shared<LLNode>* next;
        LLNode(int val = -1, Shared<LLNode>* next = 0) :
                        val(val), next(next) { }
        LLNode(LLNode* l) : val(l->val), next(l->next) { }
        virtual LLNode* clone()
        {
                return new LLNode(val, next);
        }
};
class LinkedList : public IntSet
{
        Shared<LLNode>* sentinel;
public:
        LinkedList();
        virtual bool lookup(int val) const;
        virtual void remove(int val);
        virtual bool isSane() const;
```

```
        virtual void print() const;,
        virtual bool extendedSanityCheck(verifier v, unsigned long param) const;
};
```

首先，定义了一个共享对象 LLNode，它继承于 Object<LLNode>，这样，就可以在程序中用 RSTM 系统的 API 来操作该对象。在 RSTM 系统中，通过把对象继承自 Object<T>来封装对象，需要注意的是，这里所封装的对象为程序中所直接操作的数据结构，例如这里的链表节点 LLNode。

在封装对象时，要给该对象写一个 clone 函数，这样，可以在 RSTM 系统中更好的复制该对象。同时，使用 Shared<LLNode>替换 LLNode 来定义 next，这样，next 就可以使用 RSTM 系统所定义的 API 函数，如 open_RO 等。

在定义了节点后，就可以使用该节点定义共享的链表。该链表继承于 IntSet，IntSet 是一个抽象类，它定义了一些添加，删除，查找的操作，通过继承自它，可以重载这些操作来为我们所用。在该链表类中，用 sentinel 来表示链表头，它是 Shared<LLNode>类型指针，指向链表的第一个节点的前面。如前面介绍的，通过把它定义为 Shared 类型，可以通过它操作 RSTM 的接口函数。

在该共享链表中，定义了几个函数用来插入，删除，查找等操作。在这里，将通过介绍 insert 函数来了解 RSTM 的事务操作。insert 函数的代码如下：

```
void LinkedList::insert(int val)
{
        BEGIN_TRANSACTION(insert_tx);
        const LLNode* prev = sentinel->open_RO(insert_tx);
        const LLNode* curr = prev->next->open_RO(insert_tx);
        while (curr) {
                if (curr->val >= val)
                        break;
                prev = curr;
                curr = prev->next->open_RO(insert_tx);
        }
       if (!curr->shared() || (curr->val > val)) {
                LLNode* newnode = new LLNode(val, curr->shared());
                prev->open_RW(insert_tx)->next = new Shared<LLNode>(newnode, insert_tx);
        }
        END_TRANSACTION(insert_tx);
}
```

insert 函数用来在共享链表中添加一个元素。

首先，使用 insert_tx 来标记该事务，并使用 BEGIN_TRANSACTION 来开始该事务。把该操作封装为事务，可以把操作定义为“原子性”，从而使操作在执行过程中不会受到因为资源冲突而产生的各种问题。

下面的代码为普通的插入操作的流程，所不同的是在对象读写的时候需要用

open_RO 或 open_RW 来获得该对象的“读版本”或“写版本”。

在最后，当找到插入点后，为新插入的对象申请一个新节点，同样的，该节点需要定义为 Shared 类型，以便我们以后对该节点进行的其它操作。

插入数据后，使用 END_TRANSACTION 来结束一个事务。

到此，已经利用两个例子介绍了 RSTM 的编程接口，相信读者对使用 RSTM 编程有了一个初步的认识，下一章中，我们将从它的数据结构展开具体的分析。

第3章　　RSTM 的数据结构

3.1　共享对象

前面提到过，在使用 RSTM 进行编程时，首先需要把对象封装为共享对象，以便它可以访问 RSTM 函数，进行事务操作。

通过前面的介绍可以知道，事务内存系统的关键是冲突的检测及解决。在两个事务同时访问一个共享内存时，便有可能发生冲突，例如读写冲突，写读冲突，写写冲突等，为了检测并解决，需要对对象的读者及写者进行记录，并在事务写对象失败后可以及早恢复至未修改以前的版本。介于此，需要对对象进行封装，以便通过它可以及早的发现并解决冲突。

RSTM 系统通过两个类来实现对象的封装，一个是 Shared 类，另一个是 Object 类。它们都是模板类，在下面将一一介绍。

3.1.1　Shared 类

首先看 Shared 类，它在 rstm/stm.h 中定义（关于 RSTM 文件作用的介绍请参照附录 2，下同）。

Shared 类共有两个变量， header 和 readers。它们都被定义为 volatile 属性（volatile 是 RSTM 系统用到的一个非常重要的修饰关键字，关于它的介绍请参照附录 1），表示这两个变量都有可能同时被多个线程访问。

其中，header 存放 Object 对象的地址，由于在 C++中有“字节对齐”机制，数据的存放均按字节为单位进行对齐，由于一个字为两个字节，这样，数据的起始地址总为偶数，也就是 header 的末位总为 0。因此，可以用 header 的末位记录其它一些有用的信息，在这里，用它记录对象是否被获取用于写操作。由于一个共享对象最多只能同时被一个写者所修改，因此仅需记录它是否被改写，至于改写它的是哪个，RSTM 系统在其它地方做了说明。

变量 readers 记录了对象读者的情况，它采用了映射机制，把 32 个线程对该对象的读取情况用位图的形式存储在 readers 中，但一个对象可能不仅仅有 32 个线程所访问，对于其它的情况，对每个事务用链表的形式记录了每个事务所读写的对象，这个问题将在下一节给与介绍，这里不多作论述。

到此，已经介绍完了 Shared 类的数据成员。通过对它们的介绍可以知道，Shared 类实现了一个“外壳”一样的功能，通过它记录了对象的可视读者，对象是否被修改，以及存放对象数据的 Object 对象的地址。

介绍过了 Shared 类的成员变量，下面将对它的成员函数进行简略的说明。

首先先介绍一下它的私有函数，这些函数用 private 关键字所修饰，为外界不能访问，但理解它们有助于分析该类其他的对外接口函数。

swapHeader 函数，该函数允许事务描述符（关于事务描述符的概念将在下一节介绍）在事务提交或终止时清除 header。它检验 header 的值，如果 header 中存放的是期望值，就把它置位新值，否则返回 false。

isCurrent 函数，该函数检验期望值是否是 header，或是当前对象被事务 tx 所修改之前的老版本。

installVisibleReader，该函数安装一个可视读者，readers 中通过映射的方式存储了前 32 个线程对对象的读情况，这里，把在 headers 中记录的读者称为可视读者（关于可视读者与非可视读者将在下一节介绍），该函数通过把 mask 指定的 readers 位置为 1 来把该线程置为该对象的可视读者。同理，removeVisibleReader 通过把相应的位置为 0 来移出可视读者。

abortVisibleReaders，RSTM 系统通过 readers 记录了对象的可视读者情况，这样，在检测到冲突时，可以终止这些读者来解决冲突，该函数就是把 readers 标志的事务的状态置为 ABORTED 来终止相应的事务来解决冲突。

lazyAcquire，该函数获得对象的 lazy 方式的写权限，（关于对象的读写方式将在下一节介绍）。该函数首先验证修改该对象的事务是否被终止，这样可以确保事务被其他事务终止后不在继续申请写权限。然后，通过竞争管理器裁决修改该对象的事务与该对象的可视读者之间的冲突，若该事务不被终止，则把该对象的 header 置为新 header，然后通过 abortVisibleReaders 终止所有可视读者。

介绍完 Shared 类的私有函数之后，现在来介绍它的公有函数，这些函数是它的对外接口函数，通过它们，可以在事务中访问或修改对象的值。在这些函数中 open_RO 与 open_RW 函数是该类的核心函数，将在下一章给出详细的介绍，本章先略去不讲。

new 和 delete 操作符重载，它们调用 RSTM 系统的内存申请及释放函数来申请和释放内存，通过运算符重载，可以在 Shared 对象中更方便的使用它们。

release 函数，前面介绍接口函数时介绍过它，它关闭用 open_RO 打开的对象，在事务使用 open_RO 打开对象用于读的时候，它把当前事务做为（可视）读者加以存储，以便可以及时的发现并解决冲突，在这里，当关闭该对象的时候，需要把该可视对象在 readers 或事务描述符的可视读者链表里加以清除，release 函数就完成这些清理工作。

在介绍过 Shared 类的基本数据及方法之后，下一节，将对 Object 类进行分析。

3.1.2 Object 类

同 Shared 类一样，首先看一下它的成员变量。它含有三个变量，next_obj_version，owner 和 st。

变量 next_obj_version 指向旧版本地址，每个共享对象同时只能被一个写者所修改，前面在 Shared 类中，通过在 header 末位记录了对象修改情况，在这里，next_obj_version 则指向修改前的版本，以便在修改失败时及时恢复数据。

owner 指向修改该对象的事务描述符，由于一个对象只能被一个事务所修改，通过记录修改者的事务描述符，可以了解到该事务的所有的读者和写者情况，这样，在该对象被其他对象访问时，可以查找该写者的信息以检测冲突。

而 st 指向该对象的 Shared 版本地址。

Object 类的成员函数不多，并且其含义并不难于理解，介于篇幅所限，本文不多做介绍，需要注意的是 clone 函数，该函数被定义为虚函数，通过在子类中具体化来加以使用，主要完成一些复制功能。

通过对 Shared 和 Object 的描述可以知道，在 Object 类中，st 指向 Shared 对象，而同时，Shared 类中，header 存放 Object 的地址，这样，通过 Shared 类和 Object 类，对对象进行了一次包装。在共享对象中，Shared 类和 Object 类是同时出现的，如图 3-1 所示。

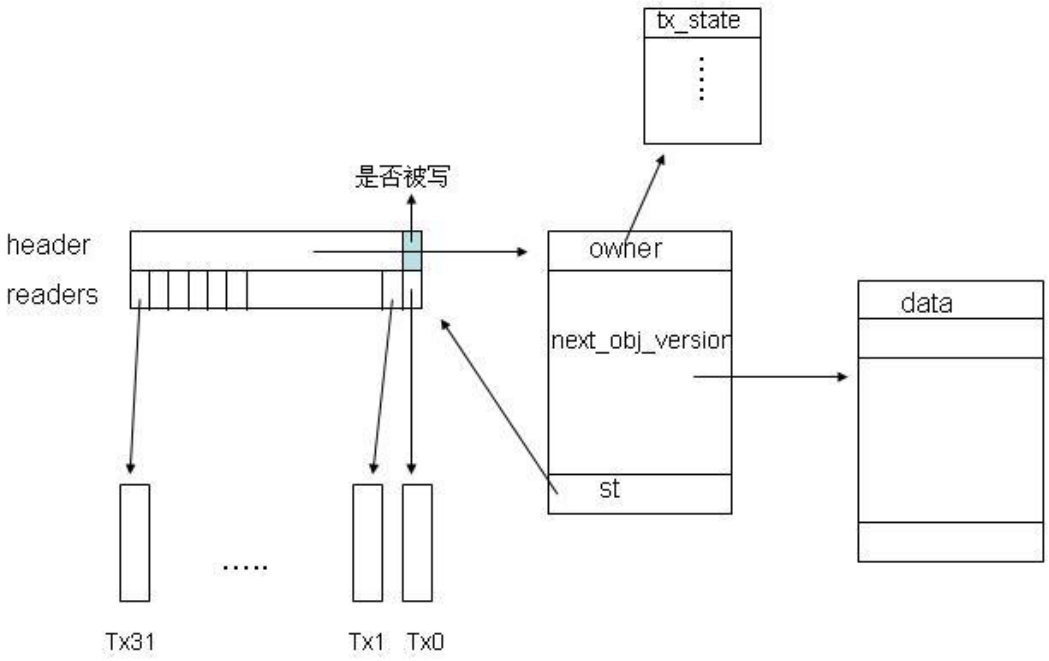

图 3-1 共享对象的封装

在图 3-1 的左侧，通过 Shared 类记录了对象的可视读者及该对象是否被修改，

而在右侧的 Object 类中，则通过 owner 记录了该对象的写者的信息，并通过 next_obj_version 记录修改前的对象信息，最后，通过 headers 和 st 来分别的指向对方，完成一次封装。

到此，我们对共享对象有了初步的认识。在完成了共享对象的介绍后，下一节，将对事务的描述符做一下分析，以便可以更好的理解事务的概念及 RSTM 的实现原理。

3.2 事务描述符

在介绍事务描述符之前，需要首先介绍一下在它里面经常使用的两个数据结构，WordList 和 SearchList。

3.2.1 WordList

在事务描述符 Descriptor 中，WordList 链表用来记录事务提交或终止时未回收的内存。在事务内存系统中，每个对象都可能同时被其他多个事务所访问，这样，在事务结束时，若事务 A 删除该对象回收内存，必然会导致其他访问该对象的事务发生错误，同时，若多个线程均对同一对象的内存进行回收，也可能出现错误，因此，在 RSTM 系统中，对象的内存回收由专门的内存回收装置来实现（见第四章）。在 Descriptor 中，使用两个链表 deleteOnCommit 和 deleteOnAbort 来记录事务提交或终止时未回收的内存情况，以后事务提交或终止时可以正确的回收这些内存。

WordList 结构图 3-2 所示。

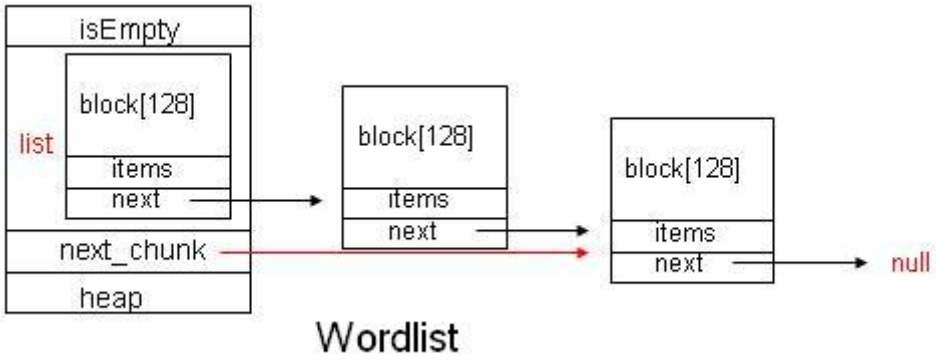

图 3-2 WordList 结构图

从图 3-2 可以看出，链表 WordList 共含有四个变量。

变量 is_empty 用于标示链表是否为空，list 为链表表头，next_chunk 指向下一个插入点的位置，方便插入操作，而 heap 则用专门的内存管理器来管理内存

的申请及释放。

在节点 wl_chunk_t 中，block 为一 void *型数组，WordList 用于存放事务提交或终止后未释放的内存，因此这里使用 block 来存放未释放内存的地址。items 表示 block 数组中非空元素的个数。next 则指向下一个块的地址。

WordList 有两个对外接口函数，reset 用于清空链表，insert 把一个内存地址插入到以 next_chunk 开始的第一个非空 block 位置。由于这些函数均很简单，在这里不多做说明。

3.2.2 SearchList

前面介绍过，一个共享对象的读者根据是否在 shared.readers 中记录，可以分为可视读者与非可视读者。

而一个事务修改的对象，则可以根据更新时机分为 lazy 型和 eager 型。

在事务修改一个对象是，有两种更新时机：直接更新和延迟更新。

直接更新：事务在事务执行过程中可以立即修改对象，其他事务不能并发地读取或者修改被更新的对象，对象的原始值被保存，以便在事务夭折时恢复

延期更新：事务要等到提交时才能修改对象，事务先建立对象的私有拷贝，然后修改这个私有拷贝，提交时根据私有拷贝修改对象（拷贝或者替代），夭折时丢弃私有拷贝。

在 RSTM 系统中，用 lazy 表示延迟更新，而用 eager 表示直接更新。

在每个事务中，都用专门的链表来存储事务的读对象和写对象的情况，这些链表有 invisibleReads，visibleReads，eagerWrites，lazyWrites。它们分别记录非可视读对象，可视读对象，eager 写对象，lazy 写对象的情况。由于它们都是使用 SearchList 链表加以存储的，所以在这里需要先简单的介绍一下 SearchList 的结构。

SearchList 结构图如下：

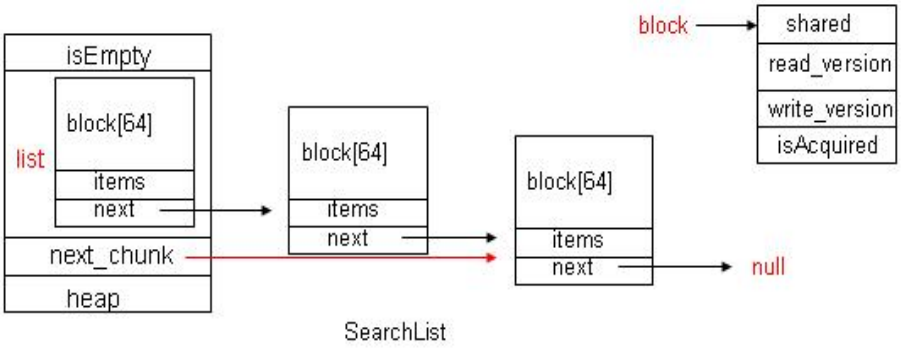

图 3-3 SearchList 结构图

SearchList 与 WordList 类似，is_empty 用于标示链表是否为空，next_chunk 指向下一个插入点的位置，用于插入操作，heap 用于内存的申请及释放。

在 block 块中，shared 用于记录对象的当前版本指针，read_version 和 write_version 分别记录对象的读版本和写版本的指针，isAcquried 标示对象是否被获取（被修改）。

SearchList 有三个对外接口函数，reset 用于清空链表，insert 把元素插入以 next_chunk 开始的第一个非空 block 位置，remove 删除 chunk 块中第 index 位置已 shared 为标示的元素。

介绍完 WordList 及 SearchList 之后，下一节，开始介绍事务描述符 Descriptor。

3.2.3 Descriptor

RSTM 系统用 Descriptor 记录事务的信息。RSTM 系统定义全局数组 desc_array 记录了每个线程的事务描述符，用于描述事务的状态。

类 Descriptor 在 rstm/descriptor.h 中定义，它的主要成员变量及描述如下图所示：

图 3-4 Descriptor 成员列表

通过上一节的介绍可以知道，在事务描述符 Descriptor 中，使用 SearchList 来记录事务读取或修改的对象。而使用 read_cnt 来记录事务读取对象的数目，使

用 acquire_rule 和 read_rule 来标示事务记录对象的方式。在 Descriptor 的成员函数中，有大量函数用于处理事务的读（写）对象。如 addVisReads 用来在事务的可视读对象列表中添加一个对象，removeVisReads 用于在可视读对象列表中删除一个对象，cleanupVisReads 用于清空可视对象列表。即其他类似函数。在这里不多做介绍，详情请参照表 3-1。

事务描述符中，WordList 来记录事务提交或终止时未回收的内存情况，使用 should_log 来标示是否使用内存回收机制，而用 reclaimer 来进行内存的回收。

在其他的成员变量中，tx_state 用来记录事务的状态，每个事务有三种状态，活跃，提交，终止，在发生冲突时，可以通过把相应的事务置为相应的状态来实现冲突的解决。在 RSTM 系统中，对于每个线程只能有一个事务，同样，每个事务必属于一个线程，在这里，记录线程 id 来标示该事务所属线程。其他的，如内存管理 heap，竞争管理 CM，冲突检测等，我们将在下面的章节加以介绍。

表 3-1 Descriptor 成员函数列表

函数	分析
verifySelf	检测事务的状态，若 tx_state 为 ABORTED，则终止事务，抛出
beginTransaction	开始一个事务，完成 Descriptor 的初始化操作
addVisReads	在事务的打开的可视的用于读的对象列表里添加一个对象
removeVisReads	在事务的打开的可视的用于读的对象列表里删除一个对象
cleanupVisReads	清空可视读对象列表
…	…
addEagerWirte	在事务的 EagerWrite 列表中添加一个对象
lookupLazyWrite	在事务的 LazyWirte 列表中查询一个对象
verifyInvisReads	确认非可视读对象是否有效
verifyLazyWrites	确认 LazyWrites 是否有效
addValidateInvisRead	添加一个有效的非可视读对象
acquireLazily	获取所有打开用于写的 lazy 对象
cleanupLazyWrites	清空 LazyWrite 列表
cleanupEagerWrites	清空 EagerWrite 列表
txAlloc txFree	内存的申请及释放，should_log 记录是否使用内存回收机制，若使用，则在事务申请释放内存的时候把它加入到待回收列表
CMOnBeginTx 等	待检测到冲突后调用相应的函数来解决冲突
logFlush	在事务提交或终止时回收内存
cleanup	在事务的最后做一些清理工作
commit	提交事务

在表 3-1 列出的成员函数中，将择其要者加以介绍，限于篇幅，其他的函数本文不多做介绍，有兴趣的读者可自行阅读代码加以理解。

verifySelf。该函数非常简单，但却在 RSTM 系统中经常出现。它首先检查事务状态，若事务状态为 ABORTED，则调用函数 STM_abort 来终止事务并抛出异常。在前面说过，由于事务发生资源冲突需要终止其中一个事务来解决冲突，将

tx_state 置为 volatile 类型。这样，在该事务进行一些操作时，事务状态随时可能被其他事务所改变，因此，需要在相应的时候检测事务状态，以免被终止的事务继续发生作用。

logFlush 函数在事务提交或终止时回收用 deleteOnCommit 或 deleteOnAbort 记录的内存。关于内存回收的情况将在第四章加以介绍。

cleanup 和 commit，将在第四章算法中介绍。

至此，已完成了对事务描述符 Descriptor 的分析。下一节，将介绍一下 RSTM 系统的全局变量。

3.3 全局变量

RSTM 系统共设置了 5 个全局变量，这些变量用来保存全局性的关于线程和事务一些共享信息，这些信息是被所有线程共享的。这些全局变量包括：存储各线程内存申请及释放信息的 heaps[MAX_THREADS](MAX_THREADS 为定义的常量，用于存储最大线程数)，存储记录各线程事务描述符的数组 desc_array[MAX_THREADS]，记录各线程线程 id 的 id 管理器 idManager，全局的冲突管理器 CONFLICTS，记录各线程时间戳的 globalEpoch，记录事务是否终止的 terminate_stm。下面将对这些全局变量一一分析：

3.3.1 idManager

线程 id 管理器（IdManager）（在 common/stm_common.h 中）共有两个变量，active_threads 用于标示全局的线程总数，tls_threadid_key 用于产生及存储每个线程的 id。该类共有三个接口函数，getThreadCount 返回全局线程总数；registerThread 注册当前线程，把线程数加一，产生并存储线程 id；getThreadId 获取当前线程 id。

通过对全局变量的分析可以知道，有大量信息是每个线程均需记录的，如线程 id，线程事务描述符，线程的内存管理器等，在这些信息中，线程 id 是最基本的信息，因为只有通过它，才可以获得当前线程的其他的全局信息。

3.3.2 globalEpoch

类 Epoch（在 common/epoch.h 中）记录各线程的时间戳信息，它在线程的内存回收时会用到。它仅含有一个变量，变量 trans_nums 用于记录每个线程的事务数，它含有以下几个对外的接口函数：

1) isStrictlyOlder ：判断 newer 的时间戳是否比 older 晚

2) copy_timestamp ： 把时间戳拷贝到 ts_ptr 开始的数组里
3) enter_transaction： 事务进入是把当前线程事务数加一
4) exit_transaction ： 事务退出时把当前线程事务数加一
5) zero_transaction ： 把数组 trans_nums 清 0

3.3.3 desc_array

全局数组 desc_array 用于记录每个线程的事务描述符，用于描述该线程事务的状态信息，在后面的模块结构一章中，将对它做详细说明。

3.3.4 heaps

关于内存管理的内容将在第四章做详细报告，本章暂不做论述。

3.3.5 CONFLICTS

关于冲突检测的内容将在第四章介绍，本章暂不做处理。

3.3.6 terminate_stm

用于标示事务内存系统是否终止。在 BEGIN_TRANSACTION 和 END_TRANSACTION 宏定义中可以看到，terminate_stm 为终止循环的一个条件，通过对它的设置可以在恰当的时候终止系统，此后，一切事务均被终止。

至此，已经完成了对 RSTM 主要数据结构的介绍，下一章，将介绍其主要算法。

第4章　　RSTM 的主要算法

在前面曾经说过，事务具有原子性，这样可以保证事务要么被完成，要么什么都不做。在 RSTM 系统中，事务的原子性是通过许多方法实现的。在这些方法中，非阻塞的同步原语是一个重要内容，在介绍 RSTM 的主要算法之前，首先对它加以介绍，以便可以更好的理解 RSTM 的算法。

4.1　非阻塞的同步原语

原子操作，顾名思义是“不可分割的”操作，在该操作执行的过程中不可被其他的过程所修改，该过程要么完全执行完成，要么所做的修改全部无效。事务的原子性决定了需要有一些“原子的”操作来完成变量的修改。RSTM 中常用的原子操作如下：

1） cas 操作

cas 操作原型为 unsigned long cas(volatile unsigned long* addr, unsigned long old, unsigned long _new)，该函数用汇编语言写成，它首先判断 addr 内容，如果*addr==old，则把*addr 改为_new，然后返回 addr 的内容。cas 操作的函数定义在 common/atomic_ops. h 文件中（下同），用伪代码表示为：

```
atomic{
    ret=*addr;
    if(*addr==old)
        *addr=_new;
    return ret;
}
```

2） tas 操作

tas 操作原型为 unsigned long tas(volatile unsigned long* addr)，它首先判断 addr 的内容，如果 addr 的内容为 0，则把它置为 1，返回 addr 的内容，用伪代码表示为：

```
atomic{
    ret=*addr;
    if(*addr==0)
        *addr=1;
    return ret;
}
```

3） bool_cas 操作

bool_cas 操作为 cas 操作的变形，它首先判断 addr 的内容，如果*addr 的值为 old，则把它置为_new,并返回 true，否则直接返回 false，返回值表示 addr 的内容是否被修改。

cas 和 bool_cas 操作是最重要的同步原语，在 Descriptor 和其他地方均有出现。

4） fai 操作

fai 操作源码如下：

```
static inline unsigned long fai(volatile unsigned long* ptr)
{
    unsigned long found = *ptr;
    unsigned long expected;
    do {
        expected = found;
    } while ((found = cas(ptr, expected, expected + 1)) != expected);
  return found;
}
```

fai 操作是 RSTM 系统中较常用的原子操作之一，它把 ptr 所指的内容加一，并返回 ptr 的值。如果 ptr 仅被一个线程访问，则改操作象普通串行程序一样运行；否则如果 ptr 的值被其他线程所修改，如在 found=*ptr 操作后，ptr 的内容被其他线程修改，则 while 循环中 ptr!=expected，故 found=cas()=*ptr!=expected，循环继续执行，found=*ptr,expected=found，直到满足循环结束条件。程序流程图如图 4-1。

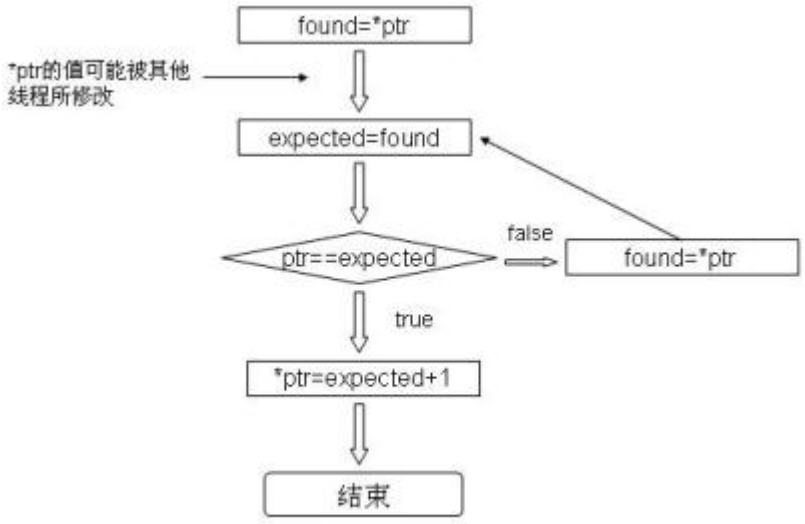

图 4-1　fai 操作流程图

用第二章介绍的伪代码表示就是：

```
atomic{
    *ptr++;
    return *ptr;
}
```

Common/atomic_ops. h 中还包含了其他的原子操作，本文不作一一描述，在

下文用到之处再为提醒。

4.2 共享对象的读写

在前面介绍共享对象时说过，open_RO 和 open_RW 是共享对象最主要的对外借口函数，通过它们，我们可以打开共享对象用于读写操作，在这里，将对它们深入分析。

4.2.1 open_RO

为了更好的理解共享对象的读写，需要对 RSTM 源码进行深入分析。下面，将开始分析 open_RO 函数的主要代码。在 RSTM 系统中，有大量代码被用来记录一些统计信息，除去这部分代码，可以看到，在 open_RO 函数的开始，有下面这样的代码：

```
tx->verifySelf();
const T* ret = (const T*)tx->lookupLazyWrite(this);
if (ret) {    return ret;       }
```

首先，它调用了 verifySeft 函数检查事务当前的状态，由于一个事务可能在任何状态被其他的事务所终止，为了保证执行 open_RO 的事务在活跃的状态中，需要在恰当的时候对事务的状态进行检测，在该函数中，verifySeft 函数被调用多次，本文不再一一叙述。

其次，它检查事务的 LazyWrite 列表。在事务的状态描述符中保存了多种列表，它们记录了事务打开用于读（写）的对象。LazyWrite 列表中记录了事务打开用于写操作的对象，它操作的是对象的副本，在对象写入时采用我们前面介绍过的延迟写入，只有在事务提交时才真正改变对象的原本。与此相反的是 EagerWrite，它采用了即时写入策略，在对象改变时立即更新对象。在此处，首先检查列表中是否有以前打开的对象，若有，因为 lazy 型写对象只有在事务提交或终止时才修改对象的原值，在此处，所保存的值并未发生改变，故可以直接返回该对象。

继续向下阅读 open_RO 源码，注意到下面是一个循环，代码反复执行，直到正确的返回对象。下一部分代码如下：

```
unsigned long snapshot = header;
T* curr_version = (T*)get_data_ptr(snapshot);
if (is_owned(snapshot)) {
        Descriptor* owner = curr_version->owner;
        unsigned owner_state = owner->tx_state;
        if (owner_state == ACTIVE) {
                if (owner == tx)    return (const T*)curr_version;
```

```
                if (!tx->CMShouldAbort(owner->getCM(), this))
                    continue;
                if (!bool_cas(&(owner->tx_state), ACTIVE, ABORTED))
                    continue;
                curr_version = curr_version->next_obj_version;
            }
            if (owner_state == ABORTED)
                curr_version = curr_version->next_obj_version;
            if (!swapHeader(snapshot, (unsigned long)curr_version)) {
                if (header != (unsigned long)curr_version)
                    continue;
            }
            snapshot = (unsigned long)curr_version;
        }
```

首先把 header 的内容赋给 snapshot，由于 header 是一个 volatile 变量，可能被其他线程所修改，把它赋给一个局部变量有助于简化分析。然后检测 snapshot 的低位，若为 1，则该对象被其他对象所写，且拥有该对象的事务状态为 ACTIVE，如果拥有该对象的事务为当前事务，则直接返回对象，否则把它提交给竞争管理器，如果本事务被终止，则重新循环，否则把拥有该对象的事务状态置为 ABORTED，把对象的老版本赋给 curr_version。如果拥有该对象的事务状态为 ABORTED，则直接把对象老版本赋给 curr_version。最后，把 curr_version 的值赋给 header。这样，就把该对象的旧版本赋给了 curr_version。在下面的代码中，首先判断将以可视还是不可视的方式保存对象，以可视列表存储作为分析：

```
            bool bookkeep = installVisibleReader(tx->id_mask);
            unsigned long snapshot2 = header;
            if (snapshot != snapshot2) {
                tx->addVisRead(this, curr_version);
                STM_abort(tx);
            }
```

这段代码把已获得的版本加入可视/非可视读者列表中。

到这里，已完成了 open_RO 的分析工作，回顾一下，首先检测该对象是否以前被存放在 LazyWrite 中，如果能在 LazyWrite 中找到，则直接返回，否则的话，检查该对象是否被获取以进行写操作，若被获取，则进行冲突检测，若该事务得以继续进行，则把对象的旧版本返回，若该对象以前为被写过，则直接返回该对象的当前版本。这样，open_RO 已找到合适的版本做为读操作。大致的流程图如图 4-2。

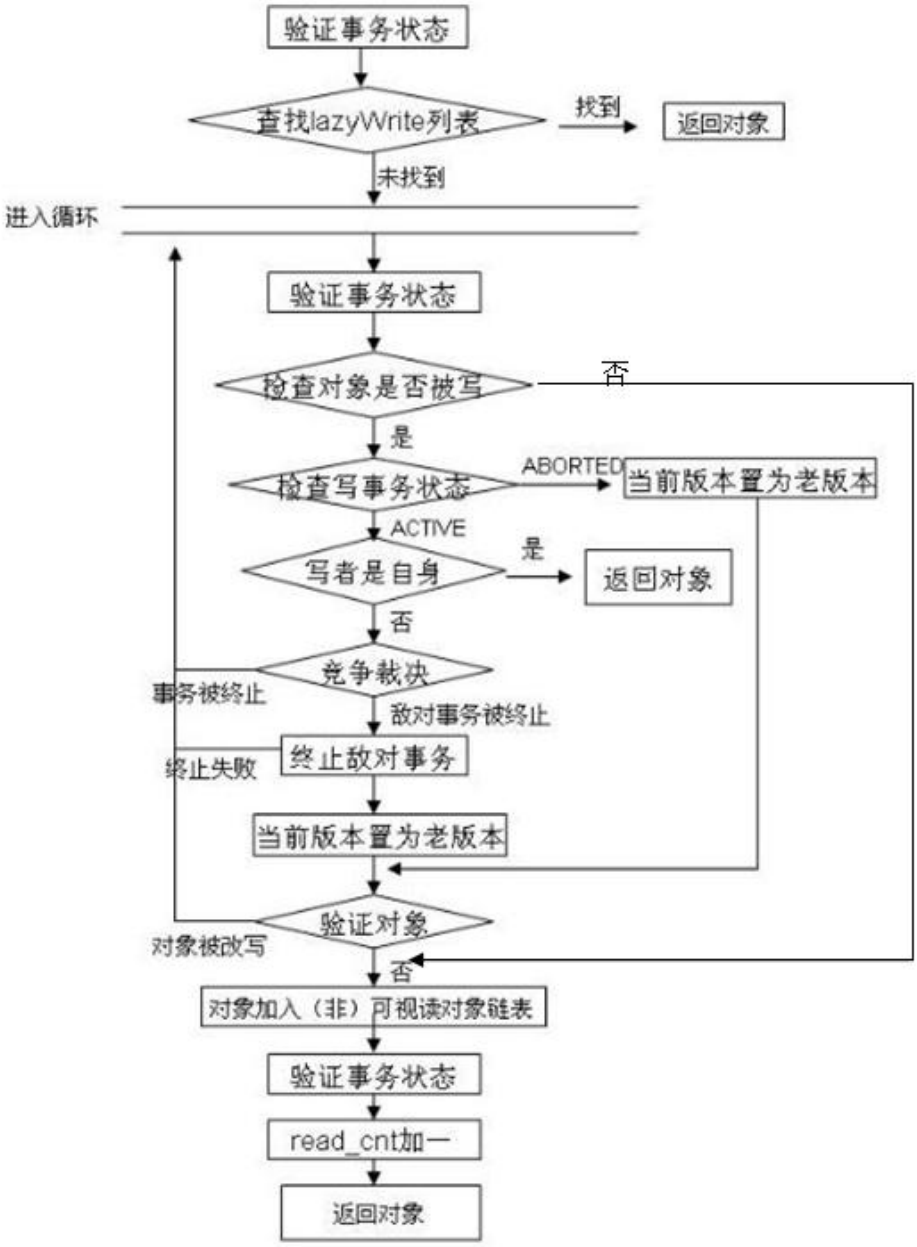

图 4-2　open_RO 流程图

4.2.2 **open_RW**

OpenRW 操作大致与 open_RO 相同，不同之处在当前版本未在 LazyWrite 中找到且未被其他事务所写的时候，首先建立一个新对象，并把当前对象作为老版本赋给新对象的 next_obj_version 指针，并根据相应的规则把新版本加入的 LazyWrite 或 EagerWrite 中。

程序流程图如图 4-3 所示，在该图中略去与 open_RO 相似部分。

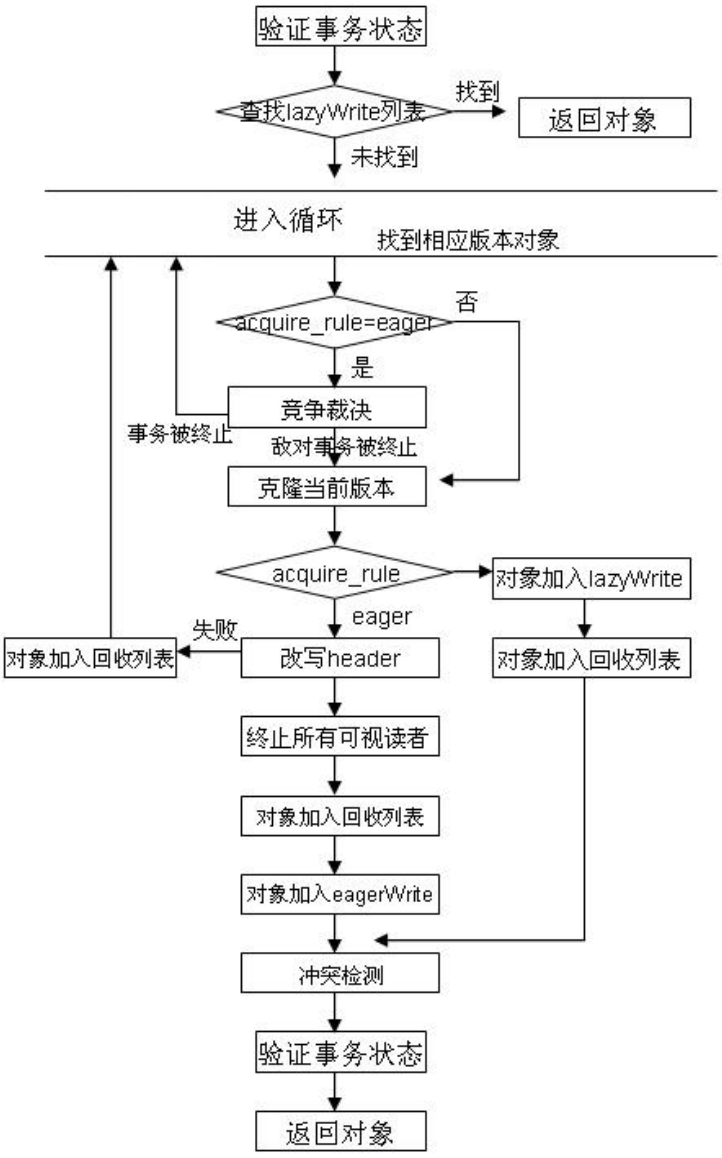

图 4-3　open_RW 程序流程图

到此，已经完成了对对象读写的介绍，相信大家到这里已经大致的了解了对象的封装及冲突的检测机制，下面，将进一步介绍事务的描述，事务的提交与终止。

4.3　事务的提交与夭折

4.3.1　BEGIN_TRANSACTION 与 END_TRANSACTION

前面曾经说过，在 RSTM 系统中，可以使用 BEGIN_TRANSACTION 和 END_TRANSACTION 来分别开始和结束一个事务。但它们究竟完成什么工作，

在上文中并未多做介绍。下面，将对它做详细的分析。

首先，先看一下 RSTM 系统中对它们的定义：

```
#define BEGIN_TRANSACTION(tx)                                    \
    Descriptor* tx = NULL;                                       \
    do {                                                         \
        try {                                                    \
            tx = begin_transaction();
/*--------------------------body of transaction goes here-----------------------------
-----------------------------body of transaction goes here ------------------------------*/
#define END_TRANSACTION(tx)                                       \
            assert(tx);                                          \
            tx->commit();                                         \
        } catch (Aborted) {                                      \
        } catch (...) {                                         \
            abort_transaction(tx);                                \
            tx->cleanup();                                       \
            throw;                                               \
        }                                                       \
    } while (!tx->cleanup() && !terminate_stm);                   \
    tx = NULL;
```

从上面的代码可以看出，BENGIN_TRANSACTION 和 END_TRANSACTION 均为宏定义，它们共同组成了一个 do-while 循环，循环终止的条件是 tx->cleanup() 返回非值或 terminate_stm 的值为 false，也就是说，在 事务未能正常提交或终止并且事务内存系统并未得到中止信号时，反复执行循环体代码，直到事务被正常提交。

在这里，begin_transaction 函数主要完成一些初始化操作，例如把各种链表清空，各种计数器清零，把事务状态置为 ACTIVE，以及事务开始时的冲突检测等等，而同样的，abort_transaction 完成事务终止时的一些操作，如把事务状态置为 ABORTED，以及冲突检测等等。函数 commit 及 cleanup 将在下面给以介绍。

从上面的分析可以看出，BEGIN_TRANSACTION 和 END_TRANSACTION 用宏定义了一个循环，循环保证了事务在被其他事务终止重做时总能正常提交，以保证任何事务都能完成。

下面，将继续介绍事务的提交与终止。

4.3.2 事务的提交

首先看事务的提交，事务的提交是通过 Descriptor 中的函数 commit 来实现的。commit 的主要代码如下：

```
void Descriptor::commit()
{
    if (tx_state == stm::ACTIVE) {
        CMOnTryCommitTx();
        if (acquire_rule == 0)
            acquireLazily();
        if (!local_CONFLICTS.tryCommit()) {
            verifyInvisReads();
            local_CONFLICTS.forceCommit();
        }
        bool_cas(&(tx_state), stm::ACTIVE, stm::COMMITTED);
    }
}
```

从上面的代码可以看出，commit 操作首先检验事务的状态是否为活跃状态，这样可以保证它所做的工作都是在活跃状态下完成的。不会出现在事务状态为提交或夭折时仍提交事务的情况。

在确定事务状态为 ACTIVE 后，commit 函数首先调用冲突检测函数检测事务提交时是否有冲突发生，然后检验事务写者类型是否为 lazy，若为 lazy，则更新那些所修改的对象。前面说过，lazy 类型为延迟更新，它在事务提交或终止时才更新对象，在这里，当事务提交时，使用 acquireLazily 函数更新对象。

然后它调用局部冲突检测器试图提交事务，若提交不成功时，校验所有非可视读对象，确保它们都是写对象或写对象修改前版本，这样，由于该事务所访问的都是写对象版本，事务提交不会对其他对象产生影响，故可强行提交事务。commit 函数通过 forceCommit 来强行提交事务。

最后，把事务状态置为 COMMITED，事务提交成功。

综上，可以看出，commit 函数首先检查事务的状态，以确保它在活跃状态下执行。然后检测事务提交时的冲突，并把 lazy 写者未更新的对象加以更新，最后调用局部冲突检测器提交事务，把事务状态置为 COMMITTED。

下面，将继续介绍事务的夭折。

4.3.3 事务的夭折

通过前面的介绍可以知道，事务有三种状态，提交，夭折，活跃。在事务为提交或未被其他事务因冲突裁决而终止时，它总是活跃的。在两个事务发生资源冲突时，RSTM 系统必须终止一个事务来解决冲突。这样，就导致一个事务必须夭折以保证与它敌对的事务得以继续执行，竞争的裁决是通过竞争管理器来实现的。关于竞争管理的内容将在下一节加以介绍。

在前面介绍对象的读写时知道，两个事务发生冲突时，通过把一个事务的状

态置为 ABORTED，并抛出异常来实现。在事务内存系统检测到异常后，通过循环加以重做，直到事务得以提交。

一个事务从产生开始，它便做了一系列的工作，首先需要记录写对象与读对象，以便检测冲突，其次需要记录未回收的内存以便事务终止或提交时回收内存。因此，在事务提交或被其他事务终止时，它需要清除这些记录，以便在开始新事务时不会受到影响。

为了更直观的看到事务提交或夭折时所做的清理工作，先来看一下它的代码。

```
bool Descriptor::cleanup()
{
    assert(tx_state == stm::COMMITTED || tx_state == stm::ABORTED);
    if (tx_state == stm::COMMITTED)     CMOnTxCommitted();
    else            CMOnTxAborted();
    cleanupLazyWrites(tx_state);
    cleanupEagerWrites(tx_state);
    cleanupVisReads();
    logFlush();
    should_log = false;
    globalEpoch.exit_transaction(id);
    return tx_state == stm::COMMITTED;
}
```

事务的清理工作是通过 cleanup 函数来实现的。

从上面的代码可以看出，cleanup 函数首先根据事务的状态调用相应的事务级冲突管理，然后做一些清理工作，以及内存的回收工作。

到此，已经介绍完了事务的提交与夭折，下一节，将介绍冲突的检测及解决，冲突的检测与解决是事务内存系统的关键，理解它有助于更好的理解事务内存系统。

4.4 冲突检测

4.4.1 冲突的定义

在介绍冲突检测以前，首先需要明确冲突的含义。在前面曾经说过，由于在并行程序设计中，多个并行的线程可能同时对一块内存数据进行访问，从而产生多个线程同时访问一个内存数据而产生读写错误的情况。我们把这种多个并行程

序访问同一块内存区域而产生读写错误的情况成为冲突。

根据读写对象的线程的性质，冲突可以分为读-写，写-读，写-写冲突。

读写冲突(R-W)：线程 a 读数据，线程 b 写数据，这样当线程 b 写数据后，线程 a 将无法读取到以前的数据，这种不可重复读的情况称为读写冲突。

写读冲突(W-R)：线程 a 写数据，线程 b 读数据，当线程 a 修改数据后，线程 b 将无法读取到正确的数据，这种读脏数据的情况称为写读冲突。

写写冲突(W-W)：线程 ab 均写数据，这样就会发生后写的数据覆盖前些的数据的情况，这种丢失修改的情况称为写写冲突。

4.4.2 读者与写者

为了保证程序得以正确顺利的执行，需要对并行程序中的冲突进行检测处理。在 RSTM 系统中通过线程描述符及共享对象，存储了对象的读写情况，以及事务读取或修改的对象的列表，这样，在以后事务中访问数据时，可以随时检测冲突并加以解决。

前面曾经介绍过，在对象的封装时，在 Shared 类中通过 readers 定义了对象的可视读者情况，它通过映射的方式记录了线程对对象的访问情况。除此之外，RSTM 系统还通过 header 的低位来标示该对象是否被写，由于一个对象只能同时被一个事务所修改，因此记录该对象是否被修改有助于在其他事务中访问该对象时及早检测到冲突并加以处理。同时，RSTM 系统还通过在 Object 类中存储老版本数据的形式，以便在检测到冲突而终止事务时可以及时恢复数据。

由于通过 readers 记录的对象数有限，因此在 Descriptor 中通过链表的形式记录了该事务所读取或修改的对象列表。关于记录对象的情况前面有所介绍，本节不再多做说明。

这样，通过记录事务所读取或修改的对象，可以及早的发现并解决冲突。

4.4.3 冲突的检测

在 RSTM 系统中，定义了类 GlobalConflictDetector 和类 LocalConflictDetector 进行冲突的检测。全局变量 CONFLICTS 为一个 GlobalConflictDetector 型变量，在每个线程描述符 Descriptor 中，保存了一个局部的冲突检测器 local_CONFLICTS，RSTM 系统通过检测 CONFLICTS 与 local_CONFLICTS 中计数器的值来判断冲突是否发生。

冲突检测有两种时机，一是事务级的，它在事务的开始及提交（或终止）时检测冲突，另一种是对象级的，它在访问对象时检测冲突。它们在 Descriptor 中均有实现。

事务级检测：CMOnBeginTx，CMOnTryCommitTx，CMOnTxCommitted，CMOnTxAborted。它们分别在事务的开始，提交或终止时加以调用，以检测冲突。

对象级检测：CMOnTryOpenRead，CMOnTryOpenWrite，CMOnOpenRead，CMOnOpenWrite，CMOnReOpen。它们分别在打开对象用于读写的时候加以检测。如 open_RO 及 open_RW 就经常调用此类函数加以检测冲突。

除此之外，Descriptor 中还有一类函数，CMShouldAbort，这类函数在检测到冲突时加以调用，它使用专门的冲突管理策略以解决冲突。冲突的解决一般通过终止其中一个事务的方法加以实现，但到底终止哪一个事务，则取决于相应的竞争管理策略。关于竞争管理的内容，将在下一节介绍。

4.5 竞争管理

在冲突检测器检测到冲突后，它会把冲突提交给用于专门解决冲突的竞争管理器来处理冲突。通常的冲突解决策略是终止其中一个事务来实现。但终止策略是什么，有专门的竞争管理器来加以实现。

4.5.1 ContentionManager

在 RSTM 系统中，类 ContentionManager 专门用于竞争管理，它在 CM/cm.h 中定义，它是一个抽象类，不能由它直接生成对象，必须由它派生出的子类来完成竞争的解决。类 ContentionManager 仅含有一个保护行变量 priority 来记录冲突的优先级，这样在发生冲突时可以通过比较它们的优先级来决定终止哪一个事务。

竞争管理器含有三类接口函数，它们全部为虚函数，一类为事务级事件触发的矛盾冲突，一类为对象级矛盾冲突，再一类为请求方法，它们为纯虚函数，必须在子类中具体化，它们用于在两个事务发生对象访问冲突后，由竞争管理器来决定终止两个事务的哪一个，这类函数为类的主要接口函数。

事务级方法：OnBeginTransaction，OnTryCommitTransaction，OnTransactionCommitted，OnTransactionAborted。同冲突检测的时机一样，在检测到冲突后，调用相应的方法来解决冲突。事务级方法通常在事务的开始，提交或终止时加以调用，以解决冲突。

对象级方法：OnTryOpenRead，OnTryOpenWrite，OnOpenRead，OnOpenWrite，OnReOpen，这类方法在打开对象用于读写时检测到冲突后加以调用以解决冲突。

竞争裁决：ShouldAbort。该函数是竞争管理器的主要函数。通过对它加以具体化后，以相应的策略来解决冲突。竞争裁决策略有多种不同的方法，它们在

CM 文件夹中的文件加以实现。它们之间的关系如图 4-4 所示。

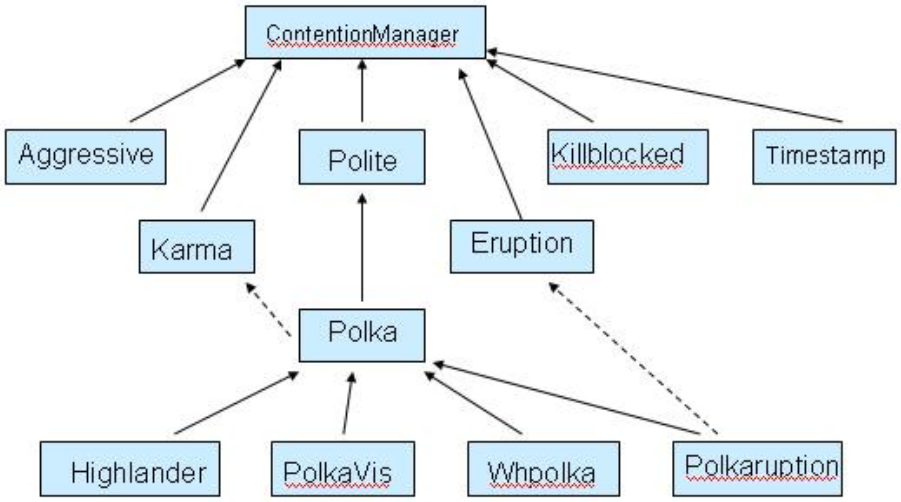

图 4-4　竞争管理策略关系图

从上图可以看出，类 ContentionManager 为所有类的父类，由它派生出一系列用于解决竞争的子类。在这里，将择其一二做为介绍，有兴趣的读者可以通过阅读 RSTM 源码来理解其他类的实现方法。

4.5.2　Aggressive

类 Aggressive 派生于类 ContentionManager，它通过实现父类的 ShouldAbort 函数来解决冲突。顾名思义，它仅仅在发生冲突后直接终止敌对事务来解决冲突。

4.5.3　Polka

类 Polka 继承于类 Polite，但它的实现中又体现了 Karma 类的方法，在这里，先介绍 Polite。

类 Polite 继承于类 ContentionManager，在试图打开对象 OnTryOpen 时，首先检查尝试次数 tries，若 tries<BACKOFF_MAX_RETRIES,则退避等待，否则则打开对象。在打开对象后 OnOpen，把尝试次数置为 0。

在发生冲突时，首先检查 tries，若 tries 大于最大重试次数，则终止敌对事务，否则终止本事务。

类 Polka 的实现方式与 Polite 类似，它多加入了优先级考虑来冲突的解决，当冲突发生时，它首先检测优先级 priority 与重试次数 tires 之和，若它们之和大于敌对事务优先级，则终止敌对事务，否则终止本事务。

到此，已经介绍完了冲突的检测和竞争的解决。下一节将介绍 RSTM 的内存管理策略。

4.6 共享内存的回收

4.6.1 内存的申请及释放

RSTM 系统用全局变量 heaps[MAX_THREADS]记录了为每个线程管理内存的内存管理器(heap)，mm 文件夹中的文件详细的介绍了关于内存的申请和释放的过程。

Stm_mm.h 文件中定义了一些宏信息，通过在编译时加入宏指令可以选择编译时管理内存的管理器。Stl_allocator.h 文件中定义了类似于 STL 的内存申请及释放方法，而 GCHeap.h 中定义了 GCHeap 的内存管理方法。在这里，将详细介绍 MallocHeap.h 头文件中叙述的方法，对其他两种方法本文不在一一详述，读者可以自己参考。

TxHeap 类中仅包含一个私有变量 id 用于标示使用该内存管理器的线程 id。该类含有两个对外接口方法，tx_alloc 用于申请内存，它直接调用 C 语言中的 malloc 方法进行内存申请，同样，tx_free 用于内存的释放，它直接调用 C 语言中的 free 方法释放内存。

在该类之外，定义函数 txalloc 和 txfree 用于内存的申请及释放。

前面提到，在全局变量 heaps[MAX_THREADS]中，为每个线程定义了一个内存管理器，为此，定义 getHeap 获取当前线程的内存管理器。

4.6.2 共享内存回收

在事务内存系统中，每个对象都可能同时被其他多个事务所访问，这样，在事务结束时，若事务 A 删除该对象回收内存，必然会导致其他访问该对象的事务发生错误，同时，若多个线程均对同一对象的内存进行回收，也可能出现错误，因此，在 RSTM 系统中，为避免这种对象已被释放后仍有其他线程访问的可能，内存的回收并未由对象本身来完成，而是由专门定义的内存回收系统来回收。RSTM 中类 Reclaimer 用于回收无用内存，它在 common/reclaimer.h 头文件中被定义。Reclaimer 结构如图 4-5 所示。

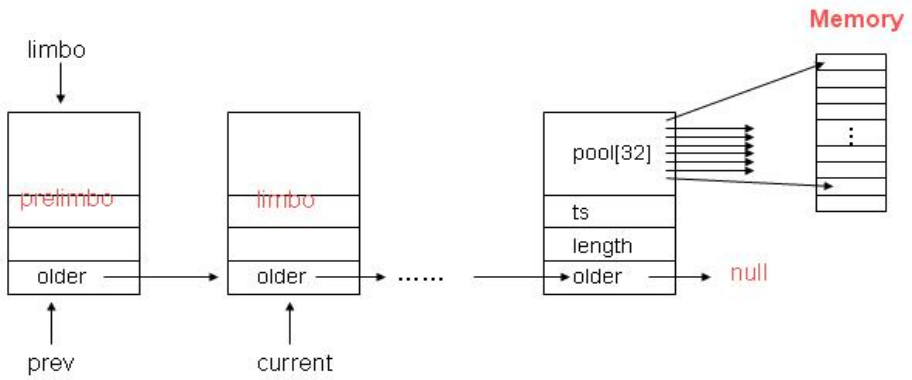

图 4-5　reclaimer 结构图

节点 limbo_t 用于存放指向废弃内存的指针的块，其中 pool 为存放待回收的内存首地址的数组，ts 为指向所有线程时间戳的指针，length 为 pool 中元素的个数或 ts 的多少，视具体情况而定，older 指向较老的待回收块的指针。

类 Reclaimer 有一个私有函数 transfer，用于回收内存。首先它把 limbo 置为 prelimbo，current=limbo->older,prev=limbo,然后进入循环，直到 current 移动到适当的位置，最后从 current 开始，依 older 方向逐个释放内存，直到链表结尾。

类 Reclaimer 的对外接口函数为 add(ptr)，它把 ptr 所指的待回收内存放入 prelimbo 块的 length 位置，然后执行 transfer 操作，释放以前存入的放弃内存。

在 Descriptor 中通过记录事务提交货终止时为回收的内存，可以在每个事务提交(COMMITED)或终止(ABORTED)时执行 cleanup 操作来清理内存。Cleanup 操作地用 logFlush 来完成内存的回收操作，它检验 WordList 链表中的内存块，通过执行 add 操作，把之前通过 heap 申请的内存都得以释放，使得 RSTM 系统能够正确高效的运行。

至此，已经介绍完了 RSTM 系统的基本实现。下面，将用一些测试例子来具体分析。

第5章　RSTM 的基准测试

前面已经介绍了 RSTM 系统主要的数据结构及模块结构，通过对 RSTM 系统初步的分析，我们已经了解了它的内存管理，冲突的检测及解决等主要问题，下面将用一个简单的例子来分析 RSTM 系统工作的过程，通过这个例子可以更好的理解前面所讲过内存管理、冲突检测、竞争管理等内容。

为了更好的理解 RSTM 系统的工作流程，将用一个简单的例子——前面介绍过的共享计数器来进行描述。

在 RSTM 源码中，Bench 文件夹中的内容为 RSTM 系统自带的示例程序，通过对它们的阅读及学习，可以更好的掌握 RSTM 的工作原理。下面将从外到内，由浅及深的介绍 Counter 的实现过程。

首先，浏览 BenchMain.cpp 这个文件，BenchMain 是 RSTM 系统所有示例程序的主函数，通过它可以了解 RSTM 主要的运作过程。抛却对命令行的解析、系统的初始化及对时间和线程统计信息不提，可以得到如下的有关 Counter 的代码：

```
stm_init(cm_type, stm_validation, use_static_cm);                    ①
B = new CounterBench();                                              ②
B->measure_speed(bm_duration, bm_threads, bm_verbosity);             ③
stm_dest();                                                          ④
```

通过前面对 RSTM 系统接口函数及 Counter 的简单介绍，可以看出，上面的代码中，①为事务系统的初始化操作，通过执行它，可以开始使用 RSTM 系统；同理，④与 stm_init 相对应，用于结束事务内存系统；②创建了一个对象，通过对该对象的封装，可以在 RSTM 的接口函数中访问该对象的信息；③为该过程的核心步骤，将在下面给出详细分析。下面，将对上面的代码做进一步的处理。

stm_init 函数在 rstm/stm.h 中给出了定义：

```
void stm::stm_init(string cm_type, string validation, bool use_static_cm)
{
    unsigned long id = stm::idManager.registerThread();
    ContentionManager* cm = cm_factory(cm_type);
    desc_array[id] = new Descriptor(id, cm, validation, use_static_cm);
}
```

通过上面的代码可以看出，stm_init 总共完成了三项初始化工作：注册当前线程，为当前线程分配一个 id；通过传入参数获取一个竞争管理器；为当前线程产生一个描述符，并把它加入到全局数组 desc_array[]中。

当程序执行到 stm_init 之后，系统为当前线程分配了一个 id，一个竞争管理器，一个描述符。继续跟踪程序向下执行，在 benchMain 中，创建了一个 CounterBench 类型的对象，在 bench/counter.h 中，可以看到如下有关 CounterBench 的信息：

```
class CounterBench : public Benchmark
```

类 CounterBench 继承自类 Benchmark，在 bench/benchmark.h 中可以看到，类 BenchMark 为一个抽象类，通过它可以对 RSTM 做一些测试及统计工作。

通过前面的介绍可以知道，对象 Counter 继承自 Object<Counter>，可以在 RSTM 的接口函数中访问该对象的一些信息，除此之外，还可以利用 Object 的 read_RO,read_RW 函数来完成对对象的读写操作。下面，来重点看 measure_speed 这个函数所执行的工作。

measure_speed 函数是类 BenchMark 的一个接口函数，在 measure_speed 中用到了 work_thread 函数，下面，先来看一下 work_thread 函数所实现的功能。work_thread 主要代码如下：

```
if (id != 0)stm_init(cm_type, stm_validation, use_static_cm);
barrier(args->id, args->threads);
do {
    b->random_transaction(args, &seed, vals[i], chance[i]);
} while (clock() < globalEndTime);
barrier(args->id, args->threads);
if (args->id != 0) stm_dest();
```

首先，当 id!=0 时，执行 stm_init 操作来完成事务的初始化操作，关于 id!=0 的问题，注意到在 BenchMain 中，同样执行了 stm_init 操作，由此我们可以推断出 main 函数线程 id 应为 0，这样，对于所有线程，stm_init 操作都执行过一次。

在上面的代码中，barrier 函数使用了两次，barrier 函数究竟完成什么功能呢？通过研究 barrier 函数的代码，可以知道 barrier 起到一面“墙”的作用，当某线程运行到 barrier 处时，它首先检查 fai(&count)==nthreads-1 是否成立，也就是说首先看当前到达此持的线程数是否为最大线程数-1，若成立，则继续向下执行，若不成立，则进入循环等待。这样，当第一个线程执行到此处时，条件不满足，它将在此处等待；当第二个线程执行到此处时，同样等待；如此，直到第 n 个线程到达，条件满足，所有线程同时放行。如图 5-1 所示：

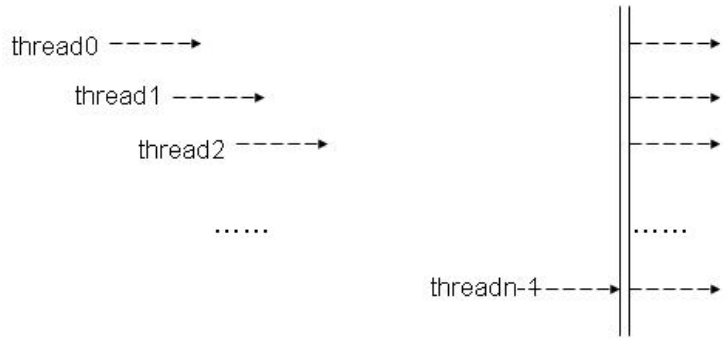

图 5-1 barrier 含义图

这样，通过两次执行 barrier 操作，可以使所有的线程均在两面“墙”之间执行。如此，RSTM 系统可以统计一些事务执行时的信息，在这里不做多介绍。

说到这里，可以看到，work_thread 函数主要功能在 random_transaction 中实现。关于 random_transaction 的分析将在下面介绍，现在，回到 measure_speed 那里，重新分析 measure_speed 的功能。

measure_speed 主要代码如下：

```
for (int i = 0; i < threads; i++) {                                    ①
        args[i].id = i;
        args[i].b = this;
        args[i].duration = duration;
        args[i].threads = threads;
        for (int op = 0; op < TXN_NUM_OPS; op++) {
                args[i].count[op] = 0;
        }
}

Green_light = true;
for (int j = 1; j < threads; j++)
        pthread_create(&tid[j], &attr, &work_thread, &args[j]);        ②
work_thread((void*)(&args[0]));                                        ③

if (bm_verify) {
        bool sanity = (args[0].b)->sanity_check();                     ④
        assert(sanity);
        if (verbosity > 0)
                cout << "Completed sanity check." << endl;
}
```

其中①为每个线程初始化执行参数，②为每个线程调用 work_thread 操作，③为本线程调用 work_thread 操作，④进行一些校验工作，在此处为校验计数器是否有效（大于 0）。在上面的代码中省去了关于时间及事务执行数量的统计信息，如此，可以知道，measure_speed 像它的名字所描述的那样，首先调用 work_thread 执行随机事务，然后对各种事务执行情况进行统计，输出显示。

逐步向下，对完成主要工作的 random_transaction 做重点分析。random_transaction 为类 BenchMark 的虚函数，它在类 CounterBench 中具体实现。

```
void random_transaction(args, seed,val, chance)
{
        BEGIN_TRANSACTION(counter_tx);
        C->open_RW(counter_tx)->increment();
        END_TRANSACTION(counter_tx);
}
```

从上面的代码可以看出，random_transaction 在这里进行了一个事务，它打开

计数器进行写操作，把计数器的值加一。BENGIN_TRANSACTION 和 END_TRANSACTION 在前面均有所介绍，这里不作赘述。

可以看出，函数的主要流程是，首先根据命令行参数生成一个相应的测试对象，对该对象调用相应的函数，完成测试信息。在测试时，通过随机的读写事务，测试事务的运行情况，然后加以输出。

至此，已大致介绍了 RSTM 系统的基本流程，对事务内存有了基本的认识。现在有关事务内存的系统还不是很完善，需要广大的爱好者们继续努力。

第6章　结论

6.1　本文所做的工作

本文首先介绍了 RSTM 系统的编程接口，通过两个简单的例子让我们了解了使用 RSTM 系统进行编程的一般步骤。然后通过分析 RSTM 系统的主要数据结构，了解了事务的概念，及对象的封装等知识。进一步分析 RSTM 系统的主要算法，使我们懂得了 RSTM 系统核心的冲突检测，竞争管理以及内存回收机制等内容。最后，我们通过一个例子，系统深入了介绍了 RSTM 系统编程的一般流程，让我们对 RSTM 系统有了一个整体认识。

本文通过对 RSTM 系统源码的分析，让我们进一步的理解了事务内存的基本原理，可以使我们对并行程序设计有更深的理解。

6.2　下一步的工作

但 RSTM 系统尚有许多不足之处，如：

1. 事务内存仍是一个新生的概念，它在很多地方都不是很完善，比起传统的加锁技术，在性能上尚不能占优势。
2. RSTM 系统虽然有了初步的发展，但它仍是一个初步的测试版本，在很多地方都充斥着大量测试代码，尚不能投入正式使用。
3. RSTM 系统在使用上仍不是很方便，希望有朝一日能出现类似于我们第一章使用过的伪代码的关键字，或用标准库的形式出现在各种编程语言中。

参考文献

[1] Olukotun K and Hammond L. The Future of Microprocessors. ACM Queue, 3(7):26–29, 2005.

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

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

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

[5] Lomet D B. Process structuring, synchronization, and recovery using atomic actions. In Proceedings of ACM Conferences on Language Design for Reliable Software, 1977.

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

[7] Nir Shavit and Dan Touitou. Software Transactional Memory. In Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, 1995.

[8] Herb Sutter and James Larus, Software and the Concurrency Revolution, acmqueue.com,2006

[9] Harris T, Herlihy M, Marlow S, Peyton-Jones S. Composable memory transactions. In Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming, 2005.

[10] Blundell C, Lewis E, and Martin M. Deconstructing Transactions: The Subtleties of Atomicity, in *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*. 2005.

[11] Scott M. Sequential Specification of Transactional Memory Semantics. In Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT), 2006.

[12] Herlihy M and Wing J. Linearizability: A Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems, 12(3):463–492, 1990.

[13] 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.

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

[15] Kung H and Robinson J. On Optimistic Methods of Concurrency Control.ACM

Transactions on Database Systems, 6(2):213—226, 1981.

[16] Scherer W III andScott M. Advanced Contention Management for Dynamic Software Transactional Memory. In Proceedings of the Twentyfourth Annual ACMSIGACT-SIGOPS Symposium on Principles of Distributed Computing, 2005.

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

[18] 肖志辉,多核改善计算机性能,网络媒体,2006

附录 1　volatile 关键字

volatile 本意为“易变的”，在 C++语言中它是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改，比如操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。在 RSTM 系统中，一个变量可能被多个线程读写，volatile 关键字使得被它修饰的变量可以被其他的线程所修改，从而可以在多个线程之间传递信息。

附录 2　RSTM 文件列表

RSTM 源代码共有 7 个文件夹，每个文件夹的功能及所包含的文件如下

表 7-1　RSTM 文件列表

bench：一些利用 RSTM 所写的示例程序及模拟并行程序运行的基准程序，在默认的组建方式中允许以 bench /obj/Bench_rstm －B <benchmark>的方式 8 个基准程序，其中<benchmark>如下：Counter ，HashTable，LFUCache，LinkedList，LinkedListRelease，RandomGraph，RBTree，RBTreeLarge。	
BenchMain.cpp	模拟并发的主程序，通过读入命令行参数来产生模拟结果并显示
Benchmark.h	定义了其他示例结构的父类及其它一些用于模拟并发的数据结构
Benchmark.cpp	模拟并发的主要程序
Counter.h	共享的计数器
Hash.h　Hash.cpp	哈希表
IntSet.h	定义其它数据结构的父类及一些常用操作
FLUCache.h	网络缓存仿真
FLUCache.cpp	
LinkedList.h	排序链表
LinkedList.cpp	
LinkedListRelease.h	早期发布的排序链表
LinkedListRelease.cpp	
RandomGraphList.h	随机图
RandomGraphList.cpp	
RBTree.h	红黑树
RBTree.cpp	
RBTreeLarge.h	4KB 节点的红黑树
RBTreeLarge.cpp	
cgl：一个用锁实现的符合 RSTM API 基准的粗粒度库	
Stm.h	定义了粗粒度 RSTM 库的对外接口
Stm.cpp	定义全局锁
Cm：冲突管理器，我们可以以 bench /obj/Bench_rstm －C <contention manager>方式选择冲突管理器，其中<contention manager>为如下几种：Aggressive，Polite，Karma，Killblocked，Eruption，Timestamp，Polka，Polkaruption ，Polkavis，Highlander，Whpolka。	
Cm.h	定义了冲突管理器的父类
Cm.cpp	通过参数产生所需类型的冲突管理器
Aggressive.h	总是中止竞争者
Polite.h	指数性的退避等待
Karma.h	打开更多对象的竞争者胜利
Killblocked.h	当竞争者似乎枯竭的时候中止
Eruption.h	donate Karma when you wait
Timestamp.h	最先的总是胜利
Polka.h	polite 和 karma 的组合（默认）

Polkaruption.h	polka 与 eruption 的组合
Polkavis.h	polka　维护可视读者的代码
Highlander.h	当 A 杀死 B，A 取代 B 的 karma
Whpolka.h	polka 对于读和写有不同的 karma
mm ：　事务安全的内存管理	
Stm_mm.h	根据定义宏的不同选择不同的内存管理方式
Stm_mm.cpp	定义了全局的内存空间
StlAllocator.h	类似于 STL 的内存管理模式
MallocHeap.h	内存的申请及释放
GCHeap.h	内存的申请及释放
Rstm：　rstm 实现，共享的描述符和运行时函数	
ConflictDetector.h	冲突检测
Descriptor.h Descriptor.cpp	事务描述符
Lowbit_manip.h	一些位运算
Seachlist.h	定义链表 Seachlist
Stm.h stm.cpp Stm_def.h	定义 Object 和 Shared 类
common：rstm 和 cgl 编码的公共部分	
Atomic_ops.h	一些原子操作
Conficts.h	冲突计数器
Epoch.h	时间戳
Hrtime.h Timing.h	时间计数
Reclaimer.h	内存回收
Stm_common.h	id 管理器
Wordlist.h	定义链表 Wordlist
XGetopt.h XGetopt.cpp	从命令行中解析参数
script:　少量用于实验运行的脚本	

致　　谢

在这里感谢我的老师张坤龙副教授给我的指导与教育，也感谢我的同学与朋友们对我的帮助与关怀。