30－34，40 事务模型 数据库 ⑧ 维普资讯 http://www.cqvip.com

计算机科学1993 Vol.20 №.3

可扩充的事务模型

周源源 唐世渭 杨冬青 关 涛

TP311.13

（北京大学计算机科学技术系 北京大学视觉听觉信息处理实验室 北京100871）

摘 要

Recently, a number of extensions to the traditional model have been proposed to support new infomation-intensive applications such as CAD/CAM, AI, and GIS, etc. Long Lived Transactions, Nest Transactions, Split Transactions and Recoverable Communicating Actions are four kinds of the new models. After summarizing several kinds of new models, in this paper we present an extensible transaction model, which has some flexibility and supports many different transaction models.

一、引 言

传统的事务模型（层次、网状、关系数据库中的事务模型）是面向传统的应用领域的（如商务数据处理），具有四个特点[5]：（1)原子性；（2）持久性；（3）一致性；(4)隔离性。然而在新的应用领域中，一个事务可能要存取很多的对象，在一个事务活动期间，还可能有冗长的计算，并且又可能是交互式的，在执行过程中会停下来等待用户的输入。因此，在这些应用中，它并不象传统的事务模型那样对某一些性质要求严格。根据这些领域的特点，新的事务模型还具有如下的新特性[6,7]：

（1）长期性(long-lived)。指一个事务的执行可以长达几个月甚至几年之久。

（2）合作性(cooperativity)。指往往由多个事务合作完成一个共同的任务。这就意味着事务之间可能要进行通信，可能需要事务的修改对它事务可见，还可能在提交或滚回时要进行必要的制约。

分析传统的事务模型和新的事务模型的特点，我们可以归纳出在事务模型中必需要考虑的一些主要问题，对这些问题进一步原子化并用相应的基本过程加以实现之后，也许通过这些基本过程的不同组合就能实现不同的事务模型。ACTA形式框架的理论为这种想法提供了可能。

二、ACTA形式框架

ACTA形式框架的理论是麻塞诸塞大学的Panayiotis K. Chrysanthis等提出的，其目的是为了寻找一种有效的方法来描述各种不同的事务模型并对不同的事务模型的并发与恢复特性进行推导[1]。ACTA形式框架理论从事务的行为和事务间的相互作用出发，给出了一个描述事务系统行为的形式框架(图1)。在这个形式框架中，事务系统存在着事务与事务和事务与对象这两种作用。图中的提交依赖指的是一个事务要等另一个事务提交后才能提交。滚回依赖指的是在一个事务滚回时另一个事务也必须滚回。对象代理指的是一个事务把其存取集中的某些对象转交给另一个事务，其中若只转交对对象的控制权限，则称这种代理为对象控制权限代理，若不仅转交控制权限而且还转交对象的当前值，则称这种代理为对象当前值代理。一个事务的View集由所有该事务潜在地能

维普资讯 http://www.cqvip.com

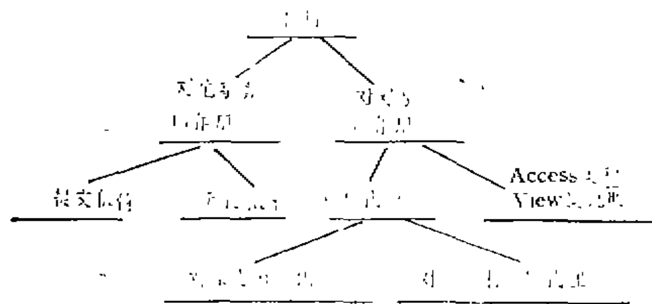

图1 ACTA形式框架结构

存取的对象组成，Access集由所有已被该事务存取过的对象组成。

在ACTA形式框架理论中还引入了一系列的符号表示，并用这些符号对嵌套、切分、RCA等事务模型进行了描述。具体细节参见[1]。

三、事务的基本行为

事务的行为有很多，但我们仅考虑应用领域中由所含关系构成的子集。

一般来讲，事务的行为可以有以下九种：

(1)事务建立 (transaction begin)。指的是事务的开始。主要任务是在内部分配属于该事务的有关数据结构，确定系列的标识信息如事务号、事务建立时间等等，并记录相应的日志信息。为了支持某一些特定的场合，事务建立还可以带一些参数。

(2) 事务提交 (transaction commit)。是指事务在完成其全部处理后，释放在事务建立时分配给该事务的数据结构，并作一些必要的后处理，最后记录日志信息等一系列活动。事务提交后，该事务在数据库中不复存在，当然该事务与它事务的所有关系也同样不复存在。

(3)修改写回(transaction check-out)。指的是把事务新插入的对象和事务修改过的对象真正写回数据库中。在大多数数据库管理系统中一般把修改写回和事务提交合并在一起，但是，由于在这里论述的是事务的基本行为，况且在某些情况中（如设有检查点的事务），事务修改写回和事务提交并不必然联系在一起，所以我们把它们看成事务的两种不同的行为。

(4)事务滚回 (transaction rollback 或 abort)。是指在某些故障发生时，撤消该事务对数据库产生的所有影响，使数据库恢复到某一个一致性的状态，就象该事务没有执行过一样。其主要任务就是利用恢复机制对事务所做的处理（修改、插入、删除）进行undo，并撤消该事务。若另外有事务与该滚回事务有滚回的约束关系时，还要根据约束的语义，进行相应处理。

上述四种行为是最基本的，一般的传统事务模型中的事务也基本上只具有这四种。下面五种行为是基于新事务模型，主要围绕事务与事务之间、事务与对象之间的关系进行考虑的。

(5)滚回依赖 (abort dependency)。如果事务A滚回依赖于事务B（符号化为A ⟶ B），则如果事务B滚回了就意味着事务A也必须滚回，但它并不包含如果事务B提交了，事务A就必须提交不得滚回之意；也不包含如果事务A滚回了、事务B也必须滚回的语义[1]。

(6)提交依赖 (commit dependency)。如果事务A对事务B有提交依赖关系（符号化为A⤳B），则事务A在事务B提交或者滚回之前不能提交[1]。但这并不意味着，如果事务B滚回了，事务A也必须滚回。

提交依赖和滚回依赖都是在提交的先后次序上做了一个规定。若事务A与事务B之间有A⟶B或者A⤳B，这都意味着若A、B都能提交的话，必须事务B先提交而A后提交。通过对事务间这种依赖关系的说明，就能防止事务过早地提交，从而避免数据库出现不一致。

我们用嵌套事务模型对这两种关系做一

维普资讯 http://www.cqvip.com

个说明。假设父事务P生成了子事务C，因为子事务可以访问父事务的数据，那么如果父事务滚回了，子事务已无存在的必要，并且继续执行下去会导致数据库的不一致性，因此父事务P与子事务C之间必须有一种滚回依赖关系：C⟶P。反过来，如果子事务没有提交，父事务P先提交也会导致数据库的不一致性，因此父事务P与子事务C之间还必须有一种提交依赖关系：P⌒→C。

(7)对象开放(object opening)。指的是事务A向事务B开放全部或部分对象，允许事务B访问事务A[1]。在诸如CASE/CAD等合作型事务处理中，在保证DB一致性的前提下，可以通过对象开放，实现多个事务合作共同完成一个任务。如在嵌套事务模型中，创立子事务时，父事务P向子事务C开放全部对象，使父事务P的修改对子事务C可见。

(8)对象当前值代理(object delegation of state)。指的是被代理事务A把被代理对象的修改后的值(可能还未写入数据库中)转移给代理事务B[1]。这显然也包含把A对这些对象的控制权限以及日志记录也转交给事务B，以后完全由B来决定是否对这些对象进行undo或者写入库中。

(9)对象的控制权限代理(object delegation of status)。指的是被代理事务A放弃对其全部或部分存取对象的控制权限，并把它转交给代理事务B[1]。这就表明被代理事务A对代理对象的修改必须进行undo以后才把对这些对象的控制权限代理给事务B。这样，以后B存取的对象当前值就不应该是A中修改过的内容。

对象的当前值代理和对象的控制权限代理合称为对象代理。事务通过对象代理放弃对某些对象的占有而将这些对象转交给另一个事务。代理的概念有效地提高了两个事务间对存取对象的可视性从而能够有选择地使某些中间结果以及协调信息对其它任务是可见的。一个代理的典型例子就是嵌套事务中，子事务提交后它所占有的对象均为父事务继承。即子事务把它的全部存取集代理给父事务，这种代理是对象当前值代理。

四、四种新的事务模型

Herin在"复杂事务"一文中，曾对当前扩充的新的事务模型做了一个总结。下面，我们就结合上述九种事务行为对常用的四种复杂事务模型作一个简介。

• 长事务模型。是针对应用领域的长期性要求提出来的。对于一个运行长达几个月甚至一年的事务来说，一个小小的故障就滚回整个事务是一种很不明智的作法。对此，长事务模型中通常采用设保存点(savepoint)的办法。在每个保存点处，把事务前一个阶段所作的修改写回库中。这样，在发生故障时就无须滚回整个事务，只要滚回到最近一个保存点即可。当然用户自己也可以指定要求事务全部滚回或者滚回到某一个保存点。

长事务模型其实在关系数据库中已有了体现。这种模型中的事务只有建立、提交、修改、写回、滚回等行为，并没有体现事务之间的合作关系。

• 嵌套事务模型。一般用于分布式环境中，是为了解决事务的部分故障问题而设计的。象长事务模型一样，利用嵌套事务，也能使故障局部化。其具体作法是：一个事务能生成子事务，子事务还能生成子子事务，由各个子事务完成对对象存取的串行化[1,6]。子事务相对于父事务及兄弟事务来说，能独立地提交，子事务的滚回也不会导致父事务及兄弟事务的滚回。因此，一个小小的故障就局部于一个很小的子事务中，不影响整个事务。此外，各个事务同时处理各自的任务，能充分发挥并行的优势。但是，嵌套事务模型只能是一种树状的层次结构，而不能是非树状的层次结构。

嵌套事务模型中的事务除了有四种最基本的行为外，还存在着事务与事务之间的关系，这在前面讲述事务的基本行为时，已经以它为例子进行过说明。在父事务P与子事务C之间具有滚回依赖约束和提交依赖约

维普资讯 http://www.cqvip.com

求：C⟶P；P⤳C。另外，在建立子事务时，父事务向子事务开放其全部存取集。在子事务提交时，子事务要把它的全部存取对象通过当前值代理转交给父事务P。

• 切分事务模型（Split Transactions）。是解决事务内部部分故障的另一种办法。假定一个CASE事务中前一个阶段对F模块进行了修改，在后一个阶段只对G模块进行修改，并不对F进行修改。若后一个阶段因故障而要滚回此事务时，显然对F的值进行恢复有些多余，降低了效率。另一方面，在后一个阶段，这个事务对F不进行任何处理，却占有对F的存取权限，影响了并行度。而采用切分事务模型，事务T在进入后一个阶段时可以分裂成两个事务T_1和T_2，T把后一个阶段不再需要存取的对象如F代理给事务T_1，通过T_1的提交把这些数据的修改永久地写入库中，同时释放加在这些数据上的存取权限，由T_2接着完成后一个阶段的任务，使故障局部于T_2。

根据T_1与T_2的相互关系，又把切分事务模型分为独立的切分事务模型和连续的切分事务模型。后者T_2必须等待T_1提交后才可提交，前者T_1与T_2的提交的前后次序是独立的[1,4]。

独立的切分事务模型中的事务行为比较简单，除了四种基本行为外，仅多了一种代理行为，即事务T把以后继续存取的数据集合当前值代理给事务T_2。由于T_1和T_2是相互独立的，并且T分裂成T_1和T_2后，T不复存在，或者说以事务T_2的形式继续执行下去，因此，整个模型中也就不存在什么依赖关系了。

连续的切分事务模型具有较复杂的语义。例如事务T_2滚回依赖于事务T_1（$T_2 \longrightarrow T_1$）。此外，除了在独立的切分事务模型中所具有的那种代理行为外，连续的切分事务模型中还有另外一种代理关系，即当事务T_1提交时，事务T_1必须把事务T_2可以进行存取的对象代理给事务C。

• 可恢复的通信事务模型RCA（Recoverable Communicating Action）。也主要用于复杂的分布式操作系统中，是为实现复杂的事务关系而设计的。其实现通过两个事务之间的信息传递来完成。称为Sender的事务把它存取集中有关的对象传递给Receiver事务（这是一种对象当前值代理）。为了维护数据库的一致性，Sender与Receiver之间必须具有：Receiver⟶Sender，Sender⤳Receiver，也就是要求Sender和Receiver若都能提交，那它们只能同时提交[1]。

其它的事务模型诸如合作事务模型（Cooperative Transactions）、事务组模型（Transaction Group）等等都是从不同的侧面对传统的事务模型进行了扩充。这里不再一一介绍了。

五、Nstar的可扩充事务模型

上面谈到的这四种模型都对传统的事务模型进行了扩充，在一定程度上满足了应用领域的要求，但正如Panayiotis[1]所说的那样，“不管这些新扩充的事务模型在支持它们初始目标系统时是多么的成功，但也只是反映了在竞争、合作环境中可能出现的相互关系集合的几个方面。因此，它们仅仅只能体现复杂信息系统中所发生的相互关系的一个子集。”我们正在着手开发可扩充的面向对象数据库Nstar中的事务模型，在一定程度上避免了这个问题。

Nstar的事务模型是在两个层次上实现的。底层提供一组通用的有关事务的调用，完成一些基本的、比较单一的功能；上层主要由一些Nstar函数组成，这些Nstar函数利用底层的Nstar调用来编写。扩充时主要是替换Nstar函数层。DBA可以根据实际的需要，利用底层的Nstar调用，编写出新的Nstar函数，替换原来的函数，从而达到扩充的目的，此外，Nstar事务管理还提供一个简单的支持长事务处理的事务模型，因此，如果用户不需要什么扩充，也可使用该模型。

从实现可扩充的途径来看，Nstar事务管

维普资讯 http://www.cqvip.com

理中所采用的方法堪称可扩充的DBMS工具箱法。高级用户能从工具箱中挑选合适的模块安装自己的并发控制和恢复模块，从而建立适当剪裁的事务管理系统[7]。这种方式特别适合于某些特别功能的场合（如CASE应用领域），生成的DBMS精干且有效[7]。

那么怎么来保证利用底层的Nstar调用构成Nstar函数以实现各种事务模型呢?我们从前述的九种行为出发，分别映射到相应的Nstar调用（用C++实现），其对应关系请见表1。

例如，若用户想使用嵌套事务模型，则可定义如下Nstar函数：

表 1　Nstar 调 用

事务行为	Nstar 调用	说　　明
事务建立	int dbntransbeg (int ifver, int ifwait, char*verno)	ifver 表明该事务是否建立版本 ifwait表明该事务在申请不到控制权限时是否等待 verno表明该事务在何版本下运行 返回事务标识号（事务号）
事务提交	int dbncommit (int tid)	tid 是提交事务的事务号返回成功与否（下均同）
修改写回	int dbncheckout (int tid)	同上
滚回依赖	int dbnrolldep(int tid1,int tid2)	tid1是滚回依赖的事务的事务号 tid2是被滚回依赖的事务的事务号
提交依赖	int dbncomdep(int tid1,int tid2)	tid1是提交依赖的事务的事务号 tid2是被提交依赖的事务的事务号
对象开放	int dbnobjopen (int tid1, int tid2, oid objoid=0)	objoid缺省时为事务tid1向事务tid2 开放其全部存取集
对象当前值代理	int dbnobjmov(int tid1,int tid2, oid objoid=0)	objoid缺省时为事务tid1 把它的全部存取集代理给事务tid2
对象控制权限代理	int dbnlockmov (int tid1, int tid2, oid objoid=0)	同上

(1) Int DBTRANSACTION (int, int, char*) ;

(2) Int DBCOMMIT ( ) ;

(3) Int DBROLLBACK ( ) ;

这三个Nstar函数可按如下方式实现：

```
int transnum, /*事务嵌套层数*/
int tid[100], /*嵌套的事务标志符*/
Int DBTRANSACTION (int ifver, int
    ifwait, char * verno)
{
  tid [transnum]=dbntransbeg (ifver,
      ifwait, verno) ;
  if (transnum>0) {
      dbn comdep (tid [transnum-1], tid
      [transnum]) ;
      dbnrolldep(tid[transnum], tid[trans-
      num-1]) ,
      dbnobjopen (tid[transnum-1],tid[tr-
      ansnum]) ,
  }
transnum++;
}
Int DBCOMMIT ( )
{
transnum--;
if (transnum==0) dbnunlock (tid[0], s-
  LOCK) ,
dbncheckout (tid[transnum]) ,
if (transnum==0) {
  dbnunlock (tid[0]) ;
  }
  else{
    dbnlockmov (tid[transnum], tid[tra-
        nsnum-1]) ,
  }
  dbncommit (tid[transnum]) ,
}
Int DBROLLBACK ( )
{
    transnum--;
    dbnunlock(tid[transnum]) ;
    dbnrollback (tid[transnum]) ,
}
```

上面给出的只是嵌套事务模型Nstar函数的大致框架，并不是严格的程序清单。此外，用户还可按上述方法扩充切分事务模型，可恢复的通信模型等事务模型。这些都显示了Nstar的事务模型的灵活性和可扩充性。（参考文献转P.40）

维普资讯 http://www.cqvip.com

SAG规范基于OSI参考模型并采用OSI的寻址和命名结构。只要不太大地破坏其格式和协议，SAG规范可以适合于任何提供端到端、全双工及虚拟线路类型连接的通信网络，TCP/IP和DECnet的peer-to-peer通信就可以提供SAG规范所需要的服务。DRDA把SNA LU6.2用于客户/服务器通信，DRDA与LU6.2关系非常密切，还不能把LU6.2的命名、动词及协议明显地映射成另一网络环境的类似机构。

■数据转换。在DRDA中，由消息的接收者来执行发送者的数据格式与其自身平台上的格式之间的转换。这种方式使数据转换和信息的最终损耗最少。但不幸的是，这一机制要求双方对于n种编码格式，至少要实现n-1种数据转换子例程集合。在SAG规范中，被传送的值是以一种定义良好的、独立于平台的标准的形式来表示的，发送方把自己的数据转换成标准格式，而接收方把标准格式转换成自己平台上使用的格式。虽然要经过两次转换，但每个平台上只需一个数据转换子例程集合。所以DRDA的机制在相似平台的环境中很有效，而在异种平台的网络环境中，则SAG的规范更现实些。

SAG规范与DRDA在技术上的差别还不止这些，以上列出的是几个主要的环节。但这已足以看出SAG规范更适合于实现基于多厂商异种平台之间透明的数据查询。

五、未来

由于标准的不完善，造成各厂商按标准实现的各种版本互不兼容，彼此无法沟通而产生了数据库互操作的问题。作为一种“权宜”之计，人们开发了用于界面转换的数据库信关。而为了从根本上解决问题，SAG和IBM又都在完善标准。在发展数据库互操作性规范方面进行了道路不同的努力。随着正式的或事实上的“完善”标准的出现，符合这些新标准的产品是否真的可以实现互操作了呢？回答是“不完全肯定”，随着标准的“完善”，只能说实现互操作的基础更好了，起点更高了。由于商业竞争的需要，各个厂商为保持其地位，必将在自己的产品中加入独有的特征，从而产生新的不兼容性，也为互操作造成新的障碍，于是又必须建立新的数据库信关来满足对异种数据库的数据共享的要求。这可能是一种永不完结的循环。那么是否数据库的互操作性就无法真正实现呢？人们已发现，关系型数据库系统由于其内在的不可克服的缺陷，而导致了互操作的不可实现，出路很可能是新一代的数据库技术——面向对象的数据库技术。对此研究者们正开始这方面新的探索。

感谢 完成本研究的过程中，特别得到了导师刘锦德教授的悉心指教，特致以衷心的感谢。（参考文献共34篇略）

（接第34页）

参考文献

[1] Panayiotis K. G, et al., ACTA: A Framework for Specifying and Reasoning about Structure and Behavior, SIGMOD'90

[2] Alt R. K. et al., Transaction Management in Engineering Database, IEEE Data Base Week 1983

[3] Calton Pu, et al., Split-Transactions for Open-Ended Activities, Proceedings of 14th VLDB Conference 1989

[4] Hector Garcia, et al., SAGAS, ACM SIGMOD'87

[5] Jorge F. G, et al., Transaction Management in an Object-Oriented Database System, SIGMOD'88

[6] Pu. C., Replication and Nested Transactions in the Eden Distributed System, VLDB'86

[7] 施伯乐等，《数据库理论新领域》