China-pub.com
下载

第四部分　事 务 管 理

这一部分由两章组成。两章的内容分别是恢复和并发，其内容彼此交错，共同构成了事务管理的主要内容。为便于教学，所以一般还是组织为两个章节。

恢复和并发，或者叫恢复和并发控制，都是关于数据保护的，即保证数据不丢失或毁坏。尤其关注以下问题：

- 系统在执行程序的过程中会出现故障，因此会使数据库处于一个未知状态；
- 两个程序在同时执行（即“并发”）时，会相互交错干扰，因此会造成不正确的结果，这种错误有可能发生在数据库的内部，也可能发生外部的现实世界。

第14章是有关恢复的内容，第15章是关于并发的讨论。注意：这两章的部分内容以不同的形式出现在《数据库系统导论：卷II》（Addison-Wesley,1983）一书。

第14章　恢　　复

14.1　引言

恢复和并发控制作为事务管理的两个重要组成部分，彼此交错。为了更清楚地阐述这两个概念，特将之分为两个章节，但本章中某些地方引用了下一章的内容。

数据库系统中的恢复（recovery）主要指恢复数据库本身，即在故障引起数据库当前状态不一致后将数据库恢复到某个正确状态或一致状态。㊀恢复的原理很简单，可以用一个词来概括，即冗余（redundancy）。换句话说，确定数据库是否可恢复的方法就是确定其包含的每一条信息是否都可利用冗余地存储在系统别处的信息重构。冗余是物理级的，我们通常认为逻辑级没有冗余，原因可参看本书的第三部分。

恢复的思想，实际上整个事务处理的思想，与系统是关系型还是层次、网状或其他模型无关，因此有关事务处理的理论研究都基于关系型系统。本章只介绍有关恢复的最重要和最基本的思想，若要对恢复作进一步的了解，可参看参考文献（尤其是[14.12]）以及练习的答案。

本章各节的内容如下：14.2节和14.3节给出了事务的基本定义以及事务恢复基本思想（将数据库从单个事务故障中恢复）；14.4节将事务恢复的思想扩展到系统恢复（将系统从引起多个并发事务同时发生故障的系统崩溃中恢复）；14.5节将事务恢复的思想继续扩展到介质故障恢复（在数据库物理上受到损害后恢复，这样的损害如磁盘的磁头碰撞等）；14.6节介绍一个相当重要的概念——两阶段提交（two-phase commit）；14.7节描述SQL中的相关语

㊀ 这时的一致性是指“满足所有已知的完整性约束”。由此可见，一致性并不一定意味着正确，正确状态一定是一致的，而一致状态可能是不正确的，即其并不能精确反映现实世界事件的真实状态。“一致性”可定义为“系统所关心的正确程度”。

China-pub.com 下载

句；14.8节进行了小结，并给出结论性的评论。

注意：我们假设数据库环境是很“大”的，即共享、多用户的。“小”的，即非共享、单用户的DBMS通常不提供或只提供很少的一部分恢复支持，这种环境下恢复是用户的责任。这就意味着用户必须进行周期性的数据库备份，故障发生时必须手工恢复。

14.2 事务

首先我们给出事务的基本定义。事务是一个逻辑工作单元（logical unit of work）。举个例子，假设零件变量P包含一个附加属性TOTQTY，表示指定零件的发货总量。换句话说，任何一个指定零件的TOTQTY的值都等于该零件的所有发货量即所有QTY值的总和，用第8章的术语来说，这是一个数据库约束。现考虑图14-1所描述的伪过程，该过程用于向数据库增加一条新的发货记录，表示供应商S5提供了发货数量为1000的零件P1（INSERT语句插入这条新的记录，UPDATE更新相应零件P1的TOTQTY值）。

```
    BEGIN TRANSACTION ;

    INSERT INTO SP
           RELATION { TUPLE { S#  S#  ( 'S5' ),
                              P#  P#  ( 'P1' ),
                              QTY QTY ( 1000 ) } } ;
    IF any error occurred THEN GO TO UNDO ; END IF ;

    UPDATE P WHERE P# = P# ( 'P1' )
           TOTQTY := TOTQTY + QTY ( 1000 ) ;
    IF any error occurred THEN GO TO UNDO ; END IF ;

    COMMIT ;
    GO TO FINISH ;

UNDO :
    ROLLBACK ;

FINISH :
    RETURN ;
```

图14-1 事务例子（伪代码）

上述例子本意是一个原子操作——增加一条新的发货记录，但事实上对数据库进行了两个更新操作——INSERT和UPDATE。而且在这两个操作之间数据库甚至是不一致的，它临时违背了零件P1的TOTQTY值应该等于所有QTY值总和的约束。由此可见，一个逻辑工作单元（即一个事务）不一定只是一个简单的数据库操作，而可能是这样的几个操作的序列，该操作序列将数据库从一个一致状态转换到另一个一致状态，中间无须保证一致性。

显然，上例的两个更新操作中一个执行而另一个未执行的情况是不允许发生的，否则会使数据库处于不一致的状态。理想的办法是可靠地保证这两个操作都被执行，很不幸不可能提供这样的保证——错误不可避免，且可能在最坏的情况下发生。如系统崩溃可能发生在INSERT与UPDATE之间；UPDATE时发生运算溢出，等等。支持事务管理（transaction management）的系统提供了另一种相当可靠的保证方式。它保证如果事务执行了几个更新操作，并在事务结束前发生了故障，这些更新操作将被撤消。也就是说，事务或者完全执行，或者全部取消（就像根本没执行过）。尽管操作序列本质上不是原子的，但从外部的角度看就像是原子的。

提供原子性保证的系统组成部分是事务管理器（transaction manager），亦称为事务处理监控器（transation processing monitor或TP monitor），COMMIT(提交)和ROLLBACK(回滚)操作是其中的关键。

China-pub.com
下载

- COMMIT操作表明事务成功地结束：它告诉事务管理器一个逻辑工作单元已成功完成，数据库又处于或应该又处于一个一致性状态，该工作单元的所有更新操作现在可被提交或永久保留。
- ROLLBACK操作表明事务不成功地结束：它告诉事务管理器出故障了，数据库可能处于不一致的状态，该逻辑工作单元已做的所有更新操作必须回滚或撤消。

因此，对上例，在两个更新操作成功执行完后，可发出 COMMIT命令提交数据库所发生的变化并使结果永久保存。如果发生了错误，也就是说，若任何一个更新操作产生了错误，必须发出ROLLBACK命令撤消已发生的变化。注：即使我们发出了 COMMIT命令，原则上系统应检查数据库的完整性约束，检测出数据库不一致的情况，并强行 ROLLBACK。但实际上我们假定系统并不知道相关的约束，因此由用户发出 ROLLBACK命令就非常必要。商用DBMS在COMMIT时并不做太多的完整性检查。

顺便提一下，实际应用中我们不仅更新数据库，而且要向用户返回提示信息。如上例，当事务成功提交时，可返回消息“发货量增加”；否则，返回“错误——发货量未增加”。相应地，消息处理也隐含着恢复处理的要求，可参看 [14.12]。

注：现在，大家可能想知道更新操作是怎样撤消的。实际上系统必须维护存储在磁盘上的日志（log或journal），日志记录了所有更新操作的具体细节，尤其是被更新对象的前后映像。因此，如果有必要撤消某个更新，系统可利用相应的日志记录恢复被更新对象的原先的值。

日志由两部分组成，活动（active）或称联机部分，以及归档（archive）或称脱机部分。联机部分用于通常的系统操作，记录系统执行更新时的具体细节，通常保存在磁盘上。当联机部分写满了，将其转移到脱机部分，由于其是顺序处理的，因此可保存在磁带上。

系统还必须保证单个语句自身的原子性，这在关系系统中尤其重要，因为关系系统的语句是基于集合的，通常一次操作基于多个元组。这样的语句完全可能在执行过程中发生故障使数据库处于不一致的状态（即某些元组更新，而另一些没有）。如果错误发生，数据库必须保证所有的元组都未改变。而且正如第 8章和第9章提到的，如果语句引起了内部另外的操作，也应有类似的处理，典型的如指定 CASCADE参照处理的外码DELETE规则。

14.3 事务恢复

一个事务以BEGIN TRANSACTION语句的成功执行开始，以 COMMIT或ROLLBACK语句的成功执行结束。 COMMIT建立了一个提交点（commit point），在商用数据库产品中亦称为同步点（syncpoint）。提交点标志着逻辑工作单元的结束，亦标志着数据库处于或应处于一致状态。相反，ROLLBACK将数据库回滚到 BEGIN TRANSACTION的状态，实际上就是回滚到前一个提交点（这里“前一个提交点”的表述是正确的，即使该事务是程序的第一个事务，可将程序的第一个BEGIN TRANSACTION默认为建立了初始的“提交点”）。

注意：这里的术语“数据库”实际指事务所存取的那部分数据库，其他事务可能与该事务并行地执行，对自己的那部分数据库进行更新，因此，整个“数据库”在提交点不可能处于完全一致的状态。我们将不考虑并发事务，这种简化并不影响讨论。

当设置提交点时：

1) 执行程序中从上一个提交点以来的所有更新操作都被提交，即结果永久保存。在提交

China-pub.com 下载

点之前所有的更新操作都应看作是不确定（tentative）的，不确定意味着它们接下来可能被取消（即回滚）。一旦提交，将保证任一个更新操作都不被撤消（这就是“提交”的定义）。

2) 所有的数据库定位（database positioning）将消失，所有的元组锁（tuple lock）都将被释放。这里“数据库定位”指在任意时间点，一个执行程序将具有对某些特定元组的寻址能力（如通过第4章中SQL的游标），在提交点该寻址能力将消失。“元组锁”将在下一章中解释。注意：有些系统提供可选项使程序实际在从一个事务转向另一事务时仍保持对元组的寻址能力（因此也保持元组锁）。14.7节将有更进一步的讨论。

上述第2点（不包括可能保持一定的寻址能力以及元组锁的说明）对以ROLLBACK结束而不是以COMMIT结束的事务也适用，而第1点显然不适用。

注意COMMIT和ROLLBACK只是结束了事务，而不是程序。通常一个程序的简单执行过程由接连运行的事务序列组成，如图14-2所示。

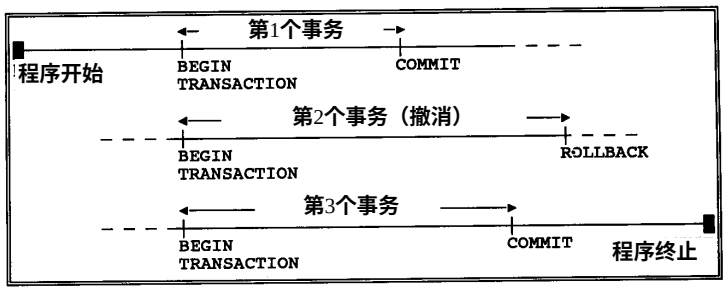

图14-2 程序的执行实际是一事务序列

再看一下图14-1所示的例子。在该例中对错误进行显式的检查，如果检测到错误，则显式地执行ROLLBACK。但是，系统不可能假设应用程序将对所有可能的错误都进行显式的检查，因此，如果事务没有到达预期的终点（显式的COMMIT或ROLLBACK），系统将隐式地执行ROLLBACK。

现在我们可以看出，事务不仅是工作单元，而且是恢复单元。因为一旦事务成功提交，系统将保证其更新永久作用到数据库上，即使系统在紧接下来的时刻就崩溃了。举个例子，在COMMIT刚被确认，而更新在物理上写入数据库之前，系统是可能崩溃的，数据可能仍在主存，并在系统崩溃时全部丢失。即使这种情况发生，系统的重启过程应将这些更新重新作用到数据库上，通过检查日志中的相关入口点，系统可找到那些已经被写入的值。这说明在COMMIT过程完成之前，日志必须物理地写出，此即日志先写原则（write-ahead log rule）。因此，重启过程将对那些成功提交但在系统崩溃前物理上未对数据库更新的事务进行恢复，由此可见事务实际上也是恢复单元。注意：下一章将提到事务也是并发（concurrency）单元，进一步讲，因为事务应该是将数据库从一个一致性状态转变为另一个一致性状态，也可认为是数据库完整性（integrity）单元。

ACID性质

事务有四个重要性质——原子性（atomicity）、一致性（consistency）、隔离性（isolation）以及持久性（durability），通常称作“ACID性质”。

- 原子性：事务是原子的，要么都做，要么都不做。

China-pub.com
下载

- 一致性：事务保证了数据库的一致性。事务将数据库从一个一致性状态转变为另一个一致性状态，但在事务内无须保证一致性。
- 隔离性：事务相互隔离。也就是说，即使通常多个事务并发执行，任一事务的更新操作直到其成功提交对其他事务都是不可见的。另一种说法为，对任意两个不同的事务 *T1*和 *T2*，*T1*可在*T2*提交后看到其更新，或 *T2*可在*T1*提交后看到其更新，但是两者不可能同时发生，第15章将做进一步的讨论。
- 持久性：一旦事务成功提交，即使系统崩溃，其对数据库的更新也将永久有效。

14.4 系统恢复

系统必须不仅能从单个事务由于溢出发生的局部故障恢复，而且能从断电引起的全局（global）故障恢复。顾名思义，局部故障只影响发生故障的事务，这已在 14.2节和14.3节进行了讨论；而全局故障影响正在运行的所有事务，具有系统范围的含义。本节及下一节将对全局故障的恢复进行简单的讨论，全局故障涉及两类：

- 系统故障（如断电）：影响正在运行的所有事务，但不破坏数据库，有时也称作软故障；
- 介质故障（如磁盘的磁头碰撞）：将破坏数据库或部分数据库，并影响正存取这部分数据的所有事务，有时也称作硬故障。

发生系统故障时，主存内容，尤其是数据库缓冲区中的内容都被丢失。此时正在运行的事务的状态都不得而知，这样的事务将不可能成功结束，因此必须在系统重启时撤消（ undo），即回滚。而且，在重启时重做（ redo）那些在系统崩溃前成功结束但未将更新从缓冲区写入物理数据库的事务也是非常必要的。

那么系统在重启时怎么知道哪些事务该撤消还是该重做呢？在一定的预定时间间隔（通常在预定数量的登记项写入日志时）内，系统自动设置检查点（ take a checkpoint）。设置检查点涉及到（a）将数据库缓冲区的内容强制写入（“ force-writing ”）物理数据库；（b）将一特殊的检查点记录（checkpoint record）写入物理日志。检查点记录包含设置检查点时正在运行的事务列表，该信息如何被使用，可参看图 14-3。

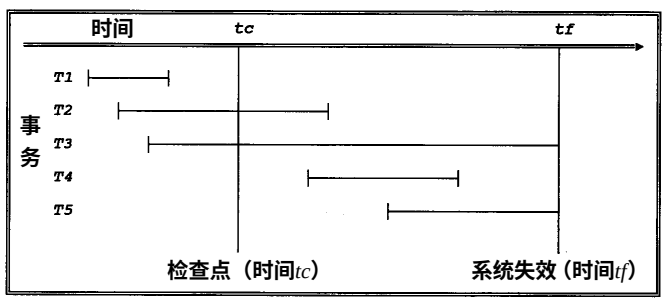

图14-3 五类事务

- 系统故障在*tf*时发生；
- *tf*之前最近的检查点在*tc*时设置；
- *T1*类的事务在*tc*前成功结束；
- *T2*类的事务在*tc*前开始，在*tc*后*tf*前成功结束；

China-pub.com 下载

- *T3*类的事务在*tc*前开始，但直到*tf*时尚未结束；
- *T4*类的事务在*tc*后开始，在*tf*前成功结束；
- *T5*类的事务在*tc*后开始，但直到*tf*时尚未结束。

显然系统重启时，*T3*和*T5*类的事务必须撤消，*T2*和*T4*类的事务必须重做。但重启过程并不涉及*T1*类的事务，因为它们的更新在*tc*设置检查点时已强制写入数据库中。对于那些在*tf*之前未成功完成（即已ROLLBACK）的事务，重启过程也不涉及。

因此，在重启时系统首先按照如下的步骤标识*T2*~*T5*类的事务：

1) 首先设置两个事务列表：UNDO列表和REDO列表。UNDO列表设置为最近一个检查点记录所包含的所有事务的列表，REDO列表设置为空。
2) 从检查点记录开始，对日志进行正向扫描；
3) 如果遇到事务*T*的BEGIN TRANSACTION日志登记项，则将*T*加入UNDO列表；
4) 如果遇到事务*T*的COMMIT日志登记项，则将*T*从UNDO列表移到REDO列表；
5) 当日志扫描结束时，UNDO列表和REDO列表分别标识了*T3*和*T5*类的事务以及*T2*和*T4*类的事务。

现在系统将反向扫描日志，撤消UNDO列表中的事务；接着再正向扫描，重做REDO列表中的事务[⊖]。注意：通过撤消操作将数据库恢复到一致性状态有时称作反向恢复（backward recovery）；通过重做操作将数据库恢复到一致性状态有时称作正向恢复（forward recovery）。

最后，当所有的恢复工作都完成时，系统就可以接受新的处理请求了。

14.5 介质恢复

注意：介质故障恢复与事务故障和系统故障的恢复有些不同，介绍它是为了完备恢复的内容。

介质故障将破坏部分数据库，如磁盘的磁头碰撞，磁盘控制器故障。介质故障恢复需利用转储的后备副本重载数据库，然后利用日志（联机和脱机日志）对转储后备副本以来完成的所有事务进行重做处理。没有必要对发生故障时正在运行的事务进行撤消处理，因为这些事务的所有更新已“撤消”（实际上是丢失）了。

要能执行介质故障恢复系统必须具有转储/重建（dump/restore）或卸载/重载（unload/reload）实用例程。转储即制作后备副本，这些副本可保存在磁带或其他用于存档的介质上，不必保存在直接存取介质上。重建即从指定的后备副本重构数据库。

14.6 两阶段提交

两阶段提交(two-phase commit)是提交/回滚概念的又一重要内容，当一事务与几个独立的“资源管理器”（每个管理器管理自己的可恢复资源集合，并维护自己的恢复日志）相互作用时，两阶段提交尤为重要。举个例子，若一事务运行在IBM大型机上，该事务对IMS数据库和DB2数据库进行更新（这样的事务是合法的）。如果该事务成功结束，其对IMS和DB2数据的

⊖ 这里对系统恢复过程的描述简化了很多。尤其是其表明系统先执行“UNDO”操作再执行“REDO”操作。事实上，早期的系统是如此工作的，但因为效率原因现在的系统通常采用其他的方法（参看[4.17]和[4.19]）。

China-pub.com
下载

所有更新操作都必须被提交；如果失败，所有的更新操作必须回滚。换句话说，不可能出现对IMS数据的更新操作被提交同时对DB2数据的更新操作被回滚的情形，反之亦然，否则事务就不是原子的。

由此可见对该事务，系统要求IMS执行COMMIT而要求DB2执行ROLLBACK是没有意义的，即使系统向两者发出相同的指示，系统也可能在两者COMMIT间或ROLLBACK间崩溃，造成不可预料的结果。因此，事务需发出一个系统范围（system-wide）的COMMIT（或ROLLBACK）。该“全局”的COMMIT或ROLLBACK由一称作协调者（coordinator）的系统部件控制，协调者保证两个资源管理器（即例中的IMS和DB2）对它们各自的更新操作所作的提交或回滚是一致的，即使系统在事务运行过程发生了故障。正是两阶段提交协议使协调者提供了这样的保证。

下面具体介绍两阶段提交协议的工作原理。假设事务已成功完成数据处理过程，它将发出系统范围的COMMIT指示，协调者在收到COMMIT请求后将进入如下两个阶段的处理过程：

1) 首先，协调者指示所有的资源管理器做好准备，这意味着每个参与者，即资源管理器，必须将事务对本地资源的所有操作的日志登记选项强制写到物理日志（即非挥发性存储设备；这样，资源管理器将具有其代表事务在本地工作的永久记录，在必要时可用于提交和回滚）。假设已成功地强制写出日志，资源管理器将向协调者发出“准备好”的响应，否则发出“未准备好”的响应。
2) 当协调者收到来自所有参与者的响应时，它将在自己的日志中登记其关于事务的决定。如果所有的响应都是“准备好”，其决定就是“提交”；如果有一个响应是“未准备好”，其决定就是“回滚”。接着协调者将向所有的参与者通知其所做决定，每个参与者将根据指示对事务进行本地的提交或回滚。注意：每个参与者必须在第二阶段完成协调者的指示，这就是协议；正是协调者日志中的事务决定的记录项标识了从阶段1到阶段2的转变。

如果系统在整个处理过程中出现了故障，重启过程将在协调者日志中查找事务决定的记录项。如果找到该记录，两阶段提交过程将从其被中止的那一点继续执行；如果未找到，系统将假设事务决定是“回滚”，并相应地完成回滚。注意：如果协调者和参与者运行于不同的计算机（如分布式系统）上，则协调者发生故障时，参与者可能需等待很长一段时间才能收到协调者的决定。只要它一直在等待，则事务由参与者执行的那部分更新操作对其他事务是不可见的，即将被加锁。

数据通信管理器（DC管理器——参看第2章）也被看作资源管理器，这就是说，消息就像数据库一样也被看作可恢复的资源，DC管理器需能参与两阶段提交过程。有关两阶段提交的更权威的论述可参看[14.12]。

14.7 SQL对事务的支持

SQL对事务的支持以及对基于事务的恢复的支持都遵循前面的概念。SQL支持通常的COMMIT和ROLLBACK语句（两者都具有可选的关键字WORK）；这些语句强制每个打开的游标CLOSE，这就引起了所有数据库定位的丢失。注意：有些SQL实现能在COMMIT时防止自动地CLOSE和数据库定位的丢失，但对ROLLBACK不支持。DB2支持在游标声明时使用

China-pub.com 下载

WITH HOLD选项，COMMIT并不关闭这样的游标，而是使其保持打开、定位状态，这样下一个FETCH操作将按顺序将游标指向下一个元组。因此，原先在下一个 OPEN时所需要的复杂的重定位代码就不再需要了，这一特性目前已包括在 SQL3中(见附录B)。

SQL对事务的支持与本章前面的概念的区别之一在于， SQL不包括显式的 BEGIN TRANSACTION语句。当程序执行了一个“ 初启事务 ”(transaction-initiaing）语句并且此时程序中还没有事务时，事务隐式地开始了（对显式的 BEGIN TRANSACTION语句的支持将在将来的标准中增加，目前已包含在 SQL3中)。哪些语句是“ 初启事务 ”型的讨论超出了本书的范围，知道前面讨论的所有可执行语句是“ 初启事务 ”型的就足够了，显然 COMMIT和ROLLBACK不包括在内。SET TRANSACTION这一语句用于设置下一事务的特性（只有当程序中无事务在执行时才能执行该语句，而且其自身不是“ 初启事务 ”型的语句)，下面讨论其中的两个性质：存取模式（access mode）和隔离级别（isolation level)，语法如下：

```
SET TRANSACTION <option commalist>
```

<option commalist>可以指定存取模式或隔离级别或两者都指定。

- 存取模式可以是READ ONLY或READ WRITE。如果两者都未指定，则存取模式默认为 READ WRITE（当然这是在隔离级别未指定为 READ UNCOMMITED的情况下，因为此时的缺省为 READ READ）；如果指定了 READ WRITE，隔离级别不能为 READ UNCOMMITED。
- 隔离级别的格式为 ISOLATION LEVEL<*isolation*>，这里 <*isolation*>为READ UNCOMMITED、READ COMMITED、REPEATABLE READ或SERIALIZABLE。其解释可参看第15章。

14.8 小结

本章对事务管理做了简单的介绍，事务是一逻辑工作单元，也是一恢复单元（同时也是并发单元和完整性单元)。事务具有ACID性质：原子性、一致性、隔离性和持久性。事务管理监控事务的执行，以保证事务具有上述的重要性质。事实上系统的目的就是保证事务的可靠执行。

事务以BEGIN TRANSACTION开始，以COMMIT或ROLLBACK结束。COMMIT建立了一个提交点（更新永久驻留)，ROLLBACK将数据库回滚到上一提交点（更新被撤消)。如果事务未到达预期的终点，系统将强制 ROLLBACK（事务恢复)。为了能撤消（或重做）对数据库的更新操作，系统必须维护恢复日志，而且事务的日志记录必须在该事务的 COMMIT操作完成之前物理地写回到日志（日志先写原则)。

系统还必须保证崩溃时事务的 ACID性质，因此系统必须（ a）重做崩溃前成功完成的事务的所有工作；（b）撤消崩溃前已启动但仍未成功完成的事务的所有工作。系统恢复作为系统重启过程的一部分，通过检查最近设置的检查点记录决定需重做和撤消的事务，检查点记录在一定的预定的时间间隔写入日志。

系统还提供介质恢复，通过先从转储的后备副本恢复数据库，再利用日志重做转储以来所做的工作。系统要支持介质恢复必须具备转储 /重建实用例程。

系统如果要使事务与多个不同的资源管理器（如两个不同的 DBMS，或一个DBMS和一个DC管理器）进行交互，同时又要维护事务的原子性，就必须使用两阶段提交协议。这两个阶

China-pub.com
下载

段是：(a) 准备阶段，协调者指示所有的参与者作好提交准备；(b) 提交阶段，在收到所有参与者的响应后，协调者作出最终决定并指示参与者提交（或回滚）。

本章还介绍了SQL对恢复的支持，SQL提供显式的COMMIT和ROLLBACK语句，但不提供显式的BEGIN TRANSACTION语句。同时它还支持SET TRANSACTION语句，这使得用户可指定下一事务的存取模式和隔离级别。

对本章的讨论都默认在应用程序环境，但是所有的概念也适用于最终用户环境。举个例子，SQL产品通常允许用户从终端交互地输入 SQL语句，每个交互式语句将作为一个事务，系统在执行了该SQL语句后，将自动代表用户发出 COMMIT命令。但是，有些系统允许用户禁止自动COMMIT功能，这样就可执行一系列的SQL语句（以显式的COMMIT和ROLLBACK结束）作为一个单一事务。这种方式可能使数据库的一部分被锁住，其他用户将无法访问，而且可能造成死锁，这些将在第15章讨论。

练习

14.1 系统不允许事务只提交对某些数据库或变量所做的更新，而不同时提交对其他数据库或变量所做的更新。为什么？

14.2 事务不能嵌套。为什么？

14.3 说明日志先写规则的内容，为什么该规则是必要的？

14.4 在下列情况下，恢复的含义分别是什么？

a. COMMIT时将缓冲区的内容强制写入数据库。

b. 在COMMIT之前缓冲区的内容不物理地写入数据库。

14.5 说明一下两阶段提交协议的内容，并讨论在每个阶段协调者和参与者分别发生故障时的含义。

14.6 利用供应商和零件数据库，写一个SQL应用程序，按零件编号顺序查询并显示所有的零件，每十个记录重新开始一个新的事务，并且要求将第十个记录删除。假设从零件表到供应表的DELETE外码规则为CASCADE（也就是说，该练习中可忽略对供应表的操作）。注：可使用SQL游标机制。

参考文献和简介

14.1 Philip A.Bernstein: “Transaction Processing Monitors,” *CACM 33*, No. 11(November 1990).

引用：“TP系统是一产品集，包括像处理器、内存、磁盘、通讯控制器这样的硬件以及像操作系统、数据库管理系统、计算机网络、 TP监控器这样的软件。这些产品的许多集成工作是由TP监控器承担的。”这篇文章是对TP监控器的结构以及功能的非正式的介绍。

14.2 Philip A.Bernstein, Vassos Hadzilacos, and Nathan Goodman: *Concurrency Control and Recovery in Database Systems*. Reading, Mass. :Addison-Wesley(1987).

这是一本教科书，正如其书名，其内容不仅包括了恢复而且包括了整个事务管理，比本章讲解得更透彻。

14.3 A.Bilris *et al*.: “ASSET: A System for Supporting Extended Transactions,” Proc.1994

China-pub.com 下载

ACM SIGMOD Int.Conf. on Management of Data,Minneapolis, Minn.(May 1994).

本章以及下一章的基本的事务概念对于新的应用来说太严格了，尤其是对交互程度相当高的应用，因此提出许多“扩展事务模型”来表示这样的应用（参看 [14.15]）。但是，在本书写作的时候，没有一个模型明显地优于其他的任何一个模型，因此数据库厂商不愿意在其产品中纳入任何一个模型。

ASSET独树一帜，其并未提出新的事务模型，而是提供了一个原语操作集合，包括通常的COMMIT等操作以及一些新的特殊操作，用以“对应用定制专用事务模型”。这篇论文还说明了 ASSET是如何指定“嵌套事务、分割事务、长事务以及其他扩展事务模型”。

14.4 L.A.Bjork:“Recovery Scenario for a DB/DC System,”Proc. ACM National Conf., Atlanta, Ga.(August 1973).

这篇论文以及其姊妹篇[14.7]很有可能是最早对恢复进行理论探讨的文章。

14.5 R.A.Crus:“Data Recovery in IBM DATABASE 2,”*IBM Sys.J.23*, No.2(1984).

详细描述了DB2的恢复机制，是对恢复技术进行总体介绍的一篇很不错的文章。尤其是，这篇文章介绍了DB2在有些事务处于恢复阶段时发生系统故障后的恢复处理过程，要保证将正在回滚的事务所做的未提交的更新撤消需要特别考虑（在某种意义上，这类问题是丢失更新的反问题）。

14.6 C.J.Date:“Distributed Database:A Closer Lock,”in C.J.Date and Hugh Darwen. *Relational Database Writing 1989-1991*. Reading, Mass. : Addison-Wesley(1992).

本章的14.6节已描述了基本的两阶段协议，基于基本的协议仍可进行改进。例如，如果参与者*P*在第一阶段响应协调者 *C*，通知*C*其在本事务中并无更新操作（即只读），则*C*在第二阶段将忽略*P*。更进一步，如果所有的参与者都在第一阶段响应内容皆为只读，则第二阶段将整个忽略。

还有其他的可能改进方案，论文中涉及了一些，主要包括：假设提交和假设回滚协议（基本协议的改进版）；过程树模型（当参与者在事务的某一部分充当协调者时）；在参与者对协调者的确认过程中通讯故障发生时的情形。注：虽然这样的讨论一般都基于分布式系统环境，大部分概念实际上适用于广泛的应用环境，可参看第 20章。

14.7 C.T.Davies, Jr.:“Recovery Semantics for a DB/DC System.”Proc. ACM National Conf., Atlanta., Ga.(August 1973).

参看[14.4]的注释。

14.8 C.T.Davies, Jr.:“Data Processing Spheres of Control,”*IBM Sys. J.17*, No.2(1978).

控制范围是对事务管理规则的最初研究和形式化探讨，是对外界视为原子的工作的抽象。不像现今大多数系统所支持的事务概念，控制范围可彼此嵌套，并可嵌套到任意深度（参看练习14.2的答案）。

14.9 Hector Garcia-Molina and Kenneth Salem:“Sagas,”Proc.1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco,Calif.(May 1987).

本章所讨论的事务都默认为在很短的时间内执行完毕（毫秒甚至微秒）。如果一个事务持续较长的时间（小时、天甚至星期），那么（a）如果其必须回滚，则需撤消大量的工作；（b）即使其最终成功结束，在相当长的时间内其将占用大量的系统资源（如数据

China-pub.com
下载

库数据等），这样将阻止其他用户的使用（参看第 15章）。不幸的是，许多“现实世界”的事务持续时间都比较长，尤其在像硬件和软件工程这样的新的应用领域。

Sagas是对上述问题的解决方案，saga是一个短事务（通常意义上）序列，并且系统保证要么该序列中的所有事务都成功执行，要么执行特定的补偿事务消除在整个未完成的saga中成功执行了的事务的影响，使得saga就像从未执行过一样。举个例子，在银行系统可能存在这样的事务，“向帐户A增加100美元”，补偿事务显然为“从帐户A中减去100美元”。对COMMIT语句的扩展允许用户在必须消除已完成的事务的影响的情况下通知系统运行补偿事务。注：理想地认为补偿事务从不回滚。

14.10 James Gray: “Notes on Data Base Operating Systems,” in R.Bayer,R.M.Graham,and G.Seegmuller (eds.),*Operating Systems:An Advanced Course* (Springer Verlag *Lecture Notes in Computer Science 60*). New York, N.Y.: Springer Verlag(1978). Also available as IBM Research Report RJ 2188(February 1978).

这是最早的也必定是最易获得的有关事务管理的资料。其对两阶段提交协议最早进行了概要描述，显然其讨论不及最新的参考文献[14.12]那样深入广泛，但仍推荐给读者。

14.11 Jim Gray: “The Transaction Concept: Virtues and Limitations,” Proc.7th Int. Conf. on Very Large Data Bases,Cannes,France(September 1981).

简明扼要地讲述了事务相关的概念、问题，包括大量的实现。其中一个特殊的问题是：通常意义下的事务是不能彼此嵌套的（见练习 14.2的答案）。然而，有没有方法允许事务划分成更小的“子事务”呢？答案是肯定的，在事务执行时，可设立多个中间保存点，以便必要时回滚到前一个保存点而不是回滚到事务的初始点。实际上，上述的保存点功能已在几个系统中实现，如Ingres（不是原型，而是商业产品），System R（虽然不是DB2）。这个概念看起来有点类似事务的概念，因为其反映了现实世界，易于理解。但是设立保存点与执行COMMIT操作并不相同，事务所做的更新直到事务（成功）结束才为其他事务可见。

注意：参考文献[14.9]中的“sagas”在某些方面说明了与保存点相同的问题，但这个概念是在Gray最先写了这篇论文后才提出的。

14.12 Jim Gray and Andreas Reuter: *Transaction Processing: Concepts and Techniques*. San Mateo,Calif.: Morgan Kaufmann(1993).

这部著作堪称计算机科学著作中的经典，乃鸿篇巨著，读来较为费时，但作者对各主题阐述得清晰明确，即使最枯燥的部分也令人爱不释手。在前言中，作者表明他们的目的是“帮助解决实际问题”，这本书“切合实际，详细描述了各种基本的事务情形”，并且包含“大量的代码段、基本算法以及数据结构”，但并不是“一本百科全书，面面俱到”。尽管正如其最后声明的，这本书并不是包罗万象，但仍是一部堪作标准的著作。强力推荐。

14.13 Jim Gray *et al*.: “The Recovery Manager of the System R Data Manager,” *ACM Comp.Surv.13*,No.2(June 1981).

参考文献[14.13]和[14.18]都与System R（是数据库领域的某些方面的先锋）的恢复机制相关。[14.13]给出了整个恢复子系统的全貌；[14.18]详细描述了一种特别的处理机制——影子页机制。

China-pub.com 下载

14.14 Theo Härder and Andreas Reuter: “Principles of Transaction-Oriented Database Recovery,” *ACM Comp. Surv.15*, No.4(December 1983).

ACID特性最早在这篇文章中提出，文章对恢复原理作了仔细的阐述，清晰易懂，同时还给出了用统一方式描述大量的恢复模式以及日志技术的术语框架，并依据这一框架对已存在的大量系统进行了分类和描述。

文章还包括一些有趣的经验性数据，包括在一个典型的大系统中三类故障（局部、系统、介质）的发生频率和可接受的恢复次数。如下表所示：

故障类型	发生频率	恢复时间
局部	每分钟10~100	与事务执行时间相同
系统	每周几次	几分钟
介质	每年一两次	1~2小时

14.15 Henry F.Korth: “The Double Life of the Abstraction Concept: Fundamental Principle and Evolving System Concept” (invited talk), Proc.21st Int.Conf. on Very Large Data Bases, Zurich,Switzerland(September 1995).

简单概要地描述了为支持新的应用需求事务概念的新发展。

14.16 Henry F.Korth,Eliezer Levy,and Abraham Silberschatz: “A Formal Approach to Recovery by Compensating Transactions,” Proc. 16th Int. Conf. on Very Large Data Bases, Brisbane, Australia (August 1990).

形式化地定义了补偿事务的概念，用于 sagas[14.9]以及其他类似方法中的已提交事务以及未提交的事务的撤消。

14.17 David Lomet and Mark R.Tuttle: “Redo Recovery after System Crashes,” Proc.21st Int. Conf. on Very Large Data Bases,Zurich,Switzerland(September 1995).

对重做恢复（即正向恢复）进行了精确、仔细的分析。“虽然重做恢复仅为恢复的一种形式，但其作为整个恢复过程的重要组成部分，必须处理最困难的问题，因而是非常重要的。”（注意，与本章14.4节中给出的算法相对照，[14.19]中的ARIES“建议将恢复过程理解为先做重做恢复，后做撤消恢复”）。作者认为他们的分析将有助于更好地理解已存在的实现方式以及提出对恢复系统的改进方案。

14.18 Raymond A.Lorie: “Physical Integrity in a Large Segmented Database,” *ACM TODS 2*, No.1(March 1977).

正如[14.13]中所说的，这篇论文描述了 System R恢复子系统的影子页机制（注：顺便提一下，术语“集成”与第 8章中的完整性概念没有任何关系）。基本思想很简单：当一（未提交的）更新第一次写入数据库时，系统并不替换已存在的那一页，而是在磁盘的其他地方存储新的一页。旧页即为新页的“影子”。提交更新必须更新指针的指向，使其指向新的一页，并废除影子页；回滚更新必须将影子页复位，并废除新页。

虽然概念很简单，但影子页机制有一严重的缺陷，就是破坏了数据原有的物理聚集。因此该机制并未应用于DB2[14.5]，虽然其曾应用于SQL/DS[4.13]中。

14.19 C.Mohan,Don Haderle,Bruce Lindsay,Hamid Pirahesh,and Peter Schwartz: “ARIES:A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks

China-pub.com
下载

Using Write Ahead Logging, " *ACM TODS 17*, No.1(March 1992).

ARIES表示“恢复和隔离应用语义的算法”。ARIES已经在几个商业和实验系统中得以实现，只是程度不同而已，其中包括 DB2。以下引自论文：“事务管理问题的解决方案可用几个指标进行判断：对一页以及跨页并发支持的程度；结果逻辑的复杂性；用于存储数据和日志的非挥发性存储介质和内存的空间负载；在重启 /恢复和正常处理阶段所需的同步和异步 I/O的次数；支持的功能类型（如部分事务回滚，等等）；重启 /恢复阶段处理的项数；重启 /恢复阶段并发处理的程度；死锁引起的事务回滚的范围；对存储数据的限制（如要求所有的记录有唯一码，限制对象的最大数目为页面大小，等等）；对新的锁模式的支持能力，这种锁模式允许基于交换及其它特性并发执行不同事务对同一数据所做的增加 /减少操作；等等。[ARIES]对这些指标处理得不错。”（对原文稍做修改）。

自从ARIES最先提出后，出现了大量的改进版和专用版：ARIES/CS（用于客户/服务器系统），ARIES/IM（用于索引管理），ARIES/NT（用于嵌套事务），等等。

部分练习答案

14.1 这违背了事务的原子性。如果事务能提交一些而不是全部更新，那么未提交的更新有可能将回滚，而已提交的更新又不能回滚，这样，事务将不再满足所有操作“要么都做，要么都不做”。

14.2 这违背了事务的原子性。若事务 *B*嵌在事务*A*内，并且发生了如下的事件（为简化起见，假设操作为“更新一个元组”）：

```
BEGIN TRANSACTION (transactionA) ;
   ...
   BEGIN TRANSACTION (transactionB) ;
   transactionB updates tuplet ;
   COMMIT (transactionB) ;
ROLLBACK (transactionA) ;
```

如果此时将元组*t*恢复为事务*A*发生前的值，则事务 *B*的COMMIT实际上就不是真正的COMMIT。相反，如果保证了 *B*的COMMIT的真实性，元组 *t*就不能恢复为事务 *A*发生前的值，从而*A*的ROLLBACK就不是真正的ROLLBACK。

事务不能嵌套，也就是说，当且仅当当前没有事务在运行时，程序才能执行 BEGIN TRANSACTION操作。

实际上，许多作者（最初由 Davies在参考文献[14.7]提出）认为，通过放弃内层事务的持久性（即 ACID中的性质“ D ”）可允许事务嵌套。也就是说，内层事务的 COMMIT将提交该事务的更新操作，但只限于外层事务的范围。如果外层事务以回滚操作终止，内层事务也必须回滚。上例中，*B*的COMMIT只对*A*是COMMIT操作，而不是对外部的世界，因此实际上其可能也将回滚。

对上述观点做进一步的简单说明。嵌套事务可看作 [14.11]中保存点（savepoint）的推广。保存点允许事务组织为一个动作序列（ a sequence of actions），从而在任何时候都可回滚到序列中最近一个动作的初始点；而嵌套则允许事务递归地组织为一个动作层次结构（a hierarchy of actions），参看图14-4，即为：

China-pub.com 下载

- 对BEGIN TRANSACTION扩展，使其支持子事务，即如果一个事务已在运行，则新的BEGIN TRANSACTION表示启动一个子事务；
- 如果事务为子事务，COMMIT提交只限于父事务范围内；
- ROLLBACK撤消所做操作，但是只恢复到特定事务的初始点，包括子事务、子子事务等等，但不包括父事务。

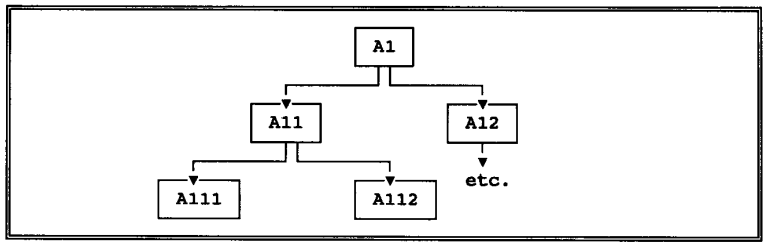

图14-4 典型的嵌套事务

从纯语义的角度看，在像SQL这样缺少显式BEGIN TRANSACTION的语言中嵌套事务很难实现，因为必须有某种显式的方式指出内层事务的初始点，用以标记内层事务失败时应回滚的终点。

14.4 a. 系统故障恢复时无须“重做”操作；

b. 无须“撤消”操作，进而用于“撤消”的日志记录也不再必要。

14.6 该习题代表了一类典型的应用，下面的解决方案也是很典型的。

```
EXEC SQL DECLARE CP CURSOR FOR
         SELECT P.P#, P.PNAME, P.COLOR, P.WEIGHT, P.CITY
         FROM   P
         WHERE  P.P# > previous_P#
         ORDER  BY P# ;
previous_P# := '      ' ;
eof := false ;
DO WHILE ( eof = false ) ;
EXEC SQL OPEN CP ;
   DO count := 1 TO 10 ;
      EXEC SQL FETCH CP INTO :P#, ... ;
      IF SQLSTATE = '02000' THEN
         DO ;
            EXEC SQL CLOSE CP ;
            EXEC SQL COMMIT ;
            eof := true ;
         END DO ;
      ELSE print P#, ... ;
      END IF ;
   END DO ;
   EXEC SQL DELETE FROM P WHERE P.P# = :P# ;
   EXEC SQL CLOSE CP ;
   EXEC SQL COMMIT ;
   previous_P# := P# ;
END DO ;
```

注意在每个事务结束时原来的定位将会丢失（即使未显式地关闭游标CP，COMMIT也将会自动执行关闭操作）。前面的代码并不高效，因为每个新事务为了重新定位都要对零件表进行再次查询。如果在P#列上有索引，查询效率将有所改善，事实上，因为{P#}是主码，优化器将选择索引作为该表的存取路径。