数据库系统导论

卷 II

〔美〕C. J. 戴特 著

科学出版社

目　录

第一章 恢 复

1.1 引 言

世界上没有任何事物能在百分之百的时间里永远正确无误地工作. 这个简单的结论虽然平凡,但对于一般计算机系统的设计,尤其是对于数据库系统的设计,意义却十分重大. 数据库系统中不但要有各种各样的检验和控制以减少出错的可能性,更重要的是,还得有相当庞大的一组过程来实现出错后的恢复. 因为即使有种种检验和控制措施,出错仍然是难免的. 例如,在系统R中,将近百分之十的程序代码是用于恢复的,而这百分之十的恢复程序却很难编写[1.4]. 在信息管理系统IMS中,这个比例数甚至更大. 因此,本章将较为详细地讨论数据库系统中的恢复问题.

在数据库系统中,恢复的基本含意是恢复数据库本身. 也就是说,在某种故障使数据库当前的状态已经不再正确,或至少是可疑时,把数据库回复到已知为正确的某一状态(后面将会看到,恢复还隐含着处理一些讯息). 造成这类故障的原因是很多的. 例如,在应用中,或在操作系统中,或数据库系统本身的程序设计错误,设备、通道或中央处理器的硬件错误,操作员错误(如装错磁带),电源波动,机房起火,甚至故意破坏,等等,可以列出一张无穷尽的清单. 但不管什么原因造成破坏,恢复的基本原理却十分简单,可以用一个词来概括,即**冗余**. 这就是说,保护数据库的方法在于保证其中的任一部分信息可以根据冗余地存贮在系统别处的其它信息来重建. 提纲挈领地说(略去许多细节),要做的事如下:

1. 周期性地(如一天一次)把整个数据库拷贝(或叫**转贮**)

到存档的存贮器上去（典型的是磁带）。

2．每次修改数据库时，在一个叫做**运行日志**的特殊的数据集中写下一个记录，这个记录中包含被修改项目的旧值和新值（当然，对于插入，没有旧值；对于删除，没有新值）。

3．如果出现故障，有两种可能性：

a）数据库本身被破坏（例如，磁盘的磁头碰撞）。在这种情况下，要装入最新的存档拷贝，然后利用运行日志**重做**这个存档拷贝之后所作的一切修改，就可重新建立起数据库。

b）数据库没有破坏，但其内容已不可靠（例如，一个程序在完成一系列逻辑上相关的更新中，突然在某一点不正常终止）。在这种情况下，要利用运行日志**撤消**所有“不可靠”的修改而把数据库恢复到某一正确状态。这时无需存档拷贝。

我们再强调一下，上面给出的要点是非常不完整的。实际上，正如我们在开头所说的，虽然恢复这个题目的基本原理是十分简单的，但它的细节却相当复杂。造成这种情况的某些原因我们将在本章的后面说明。然而，我们并不准备对恢复这个课题作详尽无遗的讨论，也不要求我们所描写的方法必须被任何系统严格地采用；我们所涉及的只是关于恢复的一般问题以及解决这些问题的**概念化**的方法。我们将忽略许多实施方面的问题，包括优化的各种可能方案。对某些特定系统恢复技术的描述见1.7节及本章的参考文献。

在开始详细解释前面所概述的恢复过程之前，我们先解释一下**双份**的可能性问题。前面我们说过，一切恢复最后都可归结为基于冗余。这说明，恢复的一个显而易见的方法是使“数据库双份制”，也即使数据库保持两个相同的拷贝，并且把所有的更新同时加到这两个拷贝上去。双份制确有其优点，这表现在既提高了可靠性，又改善了性能（后面这一点是令人惊奇的，但可见〔1.11〕的说明），因此它被某些系统所采用（如ENCOMPASS〔1.10〕）。但是这个方法也有下述缺点。首先，它需要两倍的存贮空间。其次，两个拷贝应该尽可能有独立的故障模式（例如，分

配给它们以不同的通道），以减少一个故障使两个拷贝同时遭到破坏的机会[1]。但遗憾的是，这种独立性是不能完全达到的。例如，两个拷贝总得依赖于同一中央处理器（假定系统中只有一个中央处理器的话）。第三，仅使用这个方法是不够的，因为还需要**撤消修改**，这就意味着需要一个既能给出新值又能给出旧值的运行日志。第四，运行日志可能用于与恢复完全无关的事物，诸如检查跟踪、性能分析等（虽然某些系统为此另外建立一个运行日志）。

我们还要说明一个问题再结束这一节。实践中发现，对数据库的某些部分提供可恢复性的代价——特别是维护运行日志的开销，常常抵消了其潜在的好处。一个具体的例子是一个程序可能对于单个事务在数据库中建立一个暂时的私人文件，短暂地使用它，然后又去掉它（对事务的讨论见下一节）。对这样的私人文件就没有必要去恢复它。因此，人们希望系统应该允许用户（或更可能是数据库管理员）在整个数据库中一个文件一个文件地规定哪些是需要恢复的，哪些是不需要恢复的。但为了简单，本章的大多数情况我们均假定整个数据库都是需要完全能恢复的，并且不再讨论不可恢复的数据。

1.2 事　务

数据库系统的基本目的是执行各个**事务**。一个事务就是一个**工作单元**，包括执行由应用所规定的一个操作序列，这个序列以特设的 BEGIN TRANSACTION 操作开始，以 COMMIT 操作或 ROLLBACK 操作结束*。COMMIT 的意思是“提交”，用来标志成功的结束（该工作单元已胜利完成）；ROLLBACK 的意

1）当然，即使恢复并不基于全双份方式，对于数据库和恢复信息来说，也希望有尽可能独立的故障模式。

* 在有些系统中，COMMIT和ROLLBACK分别叫做END TRANSACTION 和 ABORT TRANSACTION. ——译者注

思是“滚回”，用来标志失败的结束（该工作单元由于出现某种意外情况，例如未找到所需记录而不能顺利地完成）。显然，这里的结束指事务的结束，而不一定是程序的结束。一个程序的执行可以对应于一个接一个的若干个事务序列（见图1.1）。然而，在实践中，一个程序的执行一般只表示一个事务，而不是几个事务。

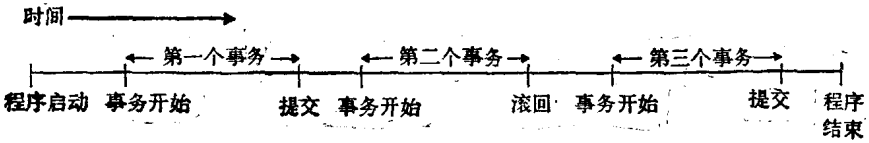

图1.1 事务和程序的执行(示例)

事务不能嵌套，就是说，只有在当前并无事务正在进行时，才能执行BEGIN TRANSACTION。反之，仅当一个事务正在进行时才能执行COMMIT和ROLLBACK。所有可恢复的操作必须在事务中执行。所谓可恢复的操作是这样一种操作，这种操作在出错的情况下可能必须撤消或者重做（换句话说，对于可恢复的操作，在运行日志中必须有相应的一个登记项）。数据库更新和讯息的输入/输出传送均为可恢复的操作。

注：为了方便，假定事务总是以明确写出的BEGIN TRANSACTION和COMMIT/ROLLBACK为界线的（也即在源程序中，凡是需要的地方，就要明显地出现BEGIN TRANSACTION和COMMIT/BOLLBACK语句）。而实际上，这些操作可能经常是隐含的。例如，程序启动就隐含一个BEGIN TRANSACTION，程序正常结束则隐含一个COMMIT，程序不正常结束隐含一个ROLLBACK。在UDL〔1.7〕中就是这样。事实上，UDL根本不包括BEGIN TRANSACTION语句——程序的启动自动开始程序的第一个事务，随后的COMMIT和ROLLBACK操作不但终止当前事务，而且也启动下一个事务[1]

1）写作本书时尚无商用的UDL系统。

(更确切地说，在程序启动之后或 COMMIT 或 ROLLBACK 之后的第一个可恢复操作被执行时，出现一个隐含的 BEGIN TRANSACTION 语句)。本章中所有程序的示例都将根据 UDL 的PL/I 文本，因此不出现显式的 BEGIN TRANSACTION 语句。但是读者应该知道，在适当的点上，BEGIN TRANSACTION（以及 COMMIT 和 ROLLBACK）还是起作用的。

图 1.2 给出了处理一个银行事务的源程序的例子。这个事务将一笔金额从一个帐户转给另一个帐户。事务由终端键入的如下输入讯息所调用：

TRANSFER $100 FROM 4732166 TO 9940103

```
TRANSFER: PROC;

   GET (FROM, TO, AMOUNT);   /* input message */
   FIND UNIQUE (ACCOUNT WHERE ACCOUNT# = FROM);
   /* now decrement the FROM balance */
   ASSIGN (BALANCE - AMOUNT) TO BALANCE;
   IF BALANCE < 0
   THEN
      DO;
         PUT ('INSUFFICIENT FUNDS');   /* output message */
         /* undo the update and terminate the transaction */
         ROLLBACK;
      END;
   ELSE
      DO;
         FIND UNIQUE (ACCOUNT WHERE ACCOUNT# = TO);
         /* now increment the TO balance */
         ASSIGN (BALANCE + AMOUNT) TO BALANCE;
         PUT ('TRANSFER COMPLETE');    /* output message */
         /* commit the update and terminate the transaction */
         COMMIT;
      END;

END /* TRANSFER */ ;
```

图1.2 TRANSFER——转帐事务(源程序)

这个讯息规定了程序名（TRANSFER)，两个帐号以及金额(值得注意的是，对于键入这个讯息的用户——可能是一个银行职员——来说，它很象是让系统完成某一任务的命令)。然后，名为 TRANSFER 的转帐程序就被调用并利用这个输入讯息(如何利用下面将说明)。在这个示例的程序中，为了说明需要

ROLLBACK 语句，故意从 FROM 的余额中先减去款项，再检查它，这当然是不切实际的。为了简化，程序中还删去了 PL/I 的 DECLARE 语句。此外，假定 AMOUNT 是正的。

从终端用户的观点看，事务是**原子**，说明这一点是很重要的。例如，在转帐的例子中，对数据库必须作两个不同的更新，这对于银行职员是不感兴趣的。对他来说，“将 x 美元从帐号 A 转到帐号 B”是一个原子操作，这个操作要么成功，要么失败。如果它成功，那当然好；如果它失败，则一切照旧（就象它根本没有被启动过一样）。特别是，数据库不允许一方面从 FROM 帐号中减去一笔金额，而另一方面 TO 帐号却没有增加金额。

由此可见，事务是一种两极命题：用户必须保证每个事务要么完整地被执行，要么完全不执行（就效果而言）。而且，用户还必须保证，如果事务被执行，则它确实只执行一次（也是就效果而言）。1.1节中提过，有时事务必须重做，即执行一次以上。但其净效果仍同执行一次一样。换句话说，可以把事务处理器系统看作是可靠的：事务既不会丢失，也不会部分地执行，也不会执行一次以上。对输入讯息的验收则提供下述保证：保证事务确实只运行一次，保证由事务所造成的数据库更新确实只进行一次，也保证由事务所产生的输出讯息确实只送出一次。系统中负责提供这种可靠性的部分就叫**恢复管理程序**。

现在我们再稍微详细些来讨论一下图 1.2 的转帐程序[1]。程序从获得输入讯息（GET）开始。输入讯息中有程序所需要的参数（FROM，TO 和 AMOUNT）。然后它找出 FROM 这个帐号，从它的余额中减去规定的金额（注：UDL 的 ASSIGN 语句确实要更新数据库中的这个记录，而不是仅仅更新所找到的记录在主存中的拷贝〔1.7〕）。完成这个更新以后，程序对余额进行测

1）更确切地说，图1.2表示的是整个一类相似的事务，而不只是一个特定的事务，各别的这类事务对应于该程序的特定的执行。这类似于我们所熟悉的过程（静态的源程序）和进程（该源程序的动态执行）的区别。在不拘形式的讨论中常常混淆这种区别。

试，看它是否变为负值；若是，则表明 FROM 帐号无足够存款，这笔转帐不能顺利完成。因此，事务向终端送回“INSUFFICIENT FUNDS”讯息(PUT)，并发出 ROLLBACK。ROLLBACK 操作表示不成功的事务结束，它的作用是撤消该事务所作的一切更新（也即所有被更新的记录恢复到该事务开始时所具有的值）。ROLLBACK 以后，控制返回程序，在我们这个例子里，也就是经过最后一个 END 语句来结束程序。

另一方面，如果帐号 FROM 有足够的存款，则程序往下执行，找出帐户 TO，并相应地更新该帐户的余额。接着它向终端发出讯息“TRANSFER COMPLETE”(PUT)，并发出一个 COMMIT 命令(表示事务成功结束，常称之为**提交点**)。在这个点上，该事务所进行的全部更新被提交。也就是说，COMMIT保证更新生效并使更新成为永恒(在COMMIT以前，最好把所有的更新都当作是暂时的。因为在提交点之前，它们可能滚回)。COMMIT 以后，控制返回程序，在我们这个例子里，也就是结束程序。

讯　息

我们前面已经几次提到，恢复除了与数据库有各种蒂莲之外，同讯息也有各种各样的蒂莲。现在我们讨论转帐这个例子中有关讯息处理的这部分内容。转帐事务不但更新数据库，还向终端用户发送讯息（INSUFFICIENT FUNDS 或 TRANSFER COMPLETE)。如果事务到达其**预期的**结束点（显式的 COMMIT 或 ROLLBACK语句)，则显然最好在终端上显示出这两个讯息中相应的一个。但如果事务失败，也即由于诸如溢出这类错误而使事务没有到达其预期的终止点，则系统将自动把它滚回（见下节的说明)。在这种情况下，其效果就如同这个事务根本没有被启动过一样，它对数据库所作的更新将被撤消，而上述的任一个讯息都不会显示出来（也许会代之以显示由系统所生成的出错讯息，例如，“OVERFLOW OCCURRED AT STATEMENT”之类)。由此可见，**在（预期的）事务结束之前，一般不应该发**

送输出讯息。换句话说，PUT 语句的作用不是直接发送输出讯息，而只是把输出讯息放在一个挂起的队列中，在事务到达预期的终点时(也就是应用程序发出 COMMIT 或 ROLLBACK 时)，队列中的所有讯息才能被送出；而当事务因诸如溢出而失败时，可抛弃这些讯息[1]。由“讯息”触发引起一些不可挽回的外部动作这一情况可作为显著的例子，说明的确需要这样一个规则。比如，根据现金出纳终端上的指示付款，则要么付出现金并更新数据库中的帐户，要么什么也不发生。

系统中负责处理讯息和讯息队列的部分叫**数据通讯管理程序**(DATA COMMUNICATIONS MANAGER，可缩写为 DC MANAGER)。由它接收原始输入讯息(在转帐例子中给出 FROM，TO 和 AMOUNT)。在收到这个讯息以后，数据通讯管理程序要完成以下两件工作：

1. 在运行日志中写下一个记录，包括输入讯息的内容和其它细节。

2. 把讯息放到输入队列中去。

然后，才可以用 GET 操作从输入队列中检索输入讯息的**拷贝**。反之，则如前所述，用 PUT 把输出讯息放到输出队列中去(输入和输出队列可在主存，也可在辅存，也可同时在主存和辅存。为了便于理解，可以认为它们是在辅存中。每个队列可包含任意数量的讯息)。

对队列有影响的其它操作是 COMMIT 和 ROLLBACK。COMMIT 和(显式的，预期的) ROLLBACK 使数据通讯管理程序完成以下工作：

1. 在运行日志中为输出队列中的讯息写下一个记录。

---

1) 应该指出，许多系统的安排实际上和本节所介绍的不同。应用程序发出的 ROLLBACK，在典型情况下并不被当作预期的终点。因此，这样一个显式 ROLLBACK 的作用是抛弃输出讯息而不是发送它。这种状况是不好的。它意味着(以转帐事务为例)，“存款不足”讯息必须在 ROLLBACK 之后，也就是由下一个事务发出；而这又意味着它是否真被送出并无保证(因为下一个事务可能由于其自身不能控制的原因而失败，见后面的说明)。

2．安排这些讯息的实际传送。

3．从输入队列中除去输入讯息（因为输入讯息现在已经被处理过了）。

由于溢出这类原因造成的事务失败，将使数据通讯管理程序删掉输出讯息（至于运行日志中对输入讯息和输出讯息的记录，则是为了支持事务的重做，而不是为了撤消，见1.4节）。

顺便说一下，就象数据库管理系统提供数据独立性一样，数据通讯管理程序也应该提供讯息独立性。换句话说，对应用而言，讯息就应该象是一些逻辑记录，它们在显示屏上或外部文件上的格式细节是无关紧要的，而且应该由程序之外的映象过程处理。例如，打印航空公司机票的程序无需操心机票的确切编排格式，它所关心的只是机票包含的信息。IMS的**讯息格式服务程序**可以作为提供这种独立性的一个系统实例。在转帐例子的源程序代码中，事实上我们已假定有这样一个系统。

关于数据通讯的其它背景材料，请参阅〔1.3〕。

事务结构

转帐这个例子的逻辑结构是典型的事务。也就是说，我们可以一般地假定所有事务均有同样的简单形式，即

接收输入讯息；

完成数据库处理；

送出一个或多个输出讯息。

注：一个输入讯息可能引出多个输出讯息。当然，这些讯息的形成可能与完成数据库处理的步骤重叠，虽然我们已经看到，不达到事务的终点，这些讯息是不会真正传送出来的。

在终端用户和程序之间可以进行更为复杂的对话，即每个方向上有多次通讯。这有两种处理方法：把它们细分为若干简单事务的序列，使每个简单事务具有上述结构；或者把它们当作一个大的事务，在这个大事务中多次重复输入—处理—输出循环，然后发出COMMIT或ROLLBACK。但是这两个方法都不完全令