第7章 事务管理

事务是DBMS的执行单位，事务将数据库从当前一致状态变为下一个一致状态，从而保证数据库中的数据始终是完整的、正确的。
事务为何能这样呢？因为事务必须满足**ACID性质**
保证事务始终满足ACID性质的**技术措施**称为**事务管理（transaction management）**，包括两大方面：
数据库恢复（database recovery）：当数据库系统发生故障时的技术措施
并发控制（concurrency control）：当多个事务并发执行时的技术措施

7.1 数据库恢复

一、恢复的基本技术

1.故障

数据库系统如同其它任何（计算机）系统一样，不可能一直正常运作，总有可能在某个时候因为这样或那样的原因而导致数据库发生**故障（failure）**，这种情况一旦发生，就难于保证事务满足ACID性质；而对用户来说，数据库中的数据变得不可靠了、甚至丢失了，这是一个巨大损失！

故障是**不可避免**的
计算机硬件故障
软件错误（系统软件和应用软件）
操作员的失误
恶意的破坏

DBMS的**恢复子系统（recovery subsystem）**能把发生故障的数据库恢复到一致状态
恢复技术是衡量DBMS性能优劣的重要指标之一

2.三类故障

事务故障（transaction failure）：指事务因不可预知的原因而夭折，这些原因可能是：
事务因运行时出错（run-time error）而无法继续执行下去；
操作员命令DBMS撤销事务；
系统调度时强行中止事务的运行，etc。
特征：发生在事务提交完成前。
系统故障（system failure）：指掉电或系统崩溃（system crash）而导致数据库系统无法继续正常运行下去，此时必须重新启动，从而导致一切事务均无条件中止运行。
特征：内存数据全部丢失，但外存上的数据库未遭破坏。
介质故障（disk/media failure）：指数据库存储介质（通常是磁盘）发生故障而导致不可读/写盘或盘中数据丢失。
特征：外存上的数据库已遭破坏，一切已提交的事务对数据库的影响全部丢失。此类故障发生的可能性小，但破坏性大！

3.恢复操作技术

恢复操作的基本原理
利用冗余数据（redundant data），将数据库中的数据在不同的存储介质上进行重复存储，当数据库本身受到破坏时，可利用冗余数据来重建数据库中已遭破坏或不正确的数据，进行恢复。
恢复的实现技术
复杂：大型数据库产品中恢复子系统的代码占全部代码10%以上。
恢复机制涉及的关键问题

如何建立冗余数据？ 如何利用冗余数据实施数据库恢复？

常用技术

数据的后备副本（backup）

运行记录/日志（log）

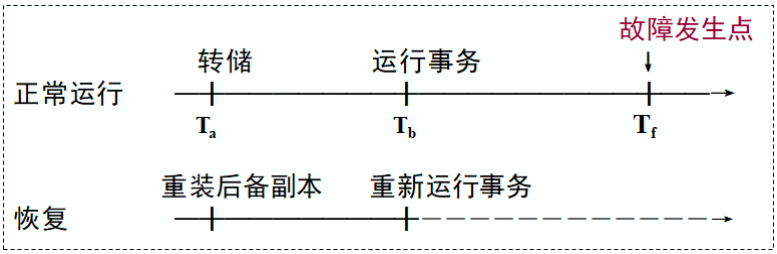
多副本

4.三种恢复技术

①恢复技术1：单纯以后备副本为基础的恢复技术

后备副本 （backup）：周期性地把数据库内容复制到磁带上（称转储dump）；副本也称存档转储（archival dump）

方法：一旦发生故障，利用后备副本重装数据库进行恢复



注：建立后备时，数据库须保持一致状态（通常需暂停系统的运行）；如果数据量很大，后备工作量很大（故后备周期不可能太频繁）。

特点：

实现技术简单。

不会持久影响数据库系统的运行性能，但备份与重装时工作量很大！

不能将数据库恢复到**最近一致状态**（会丢失更新）。

较长周期（如每个星期）的**海量转储**（Mass Dump, MD）+ 较短周期（如每天）的**增量转储**（Incremental Dump, ID）可减少转储工作量、少丢失更新，但技术复杂。

结论：这是从文件系统继承来的恢复技术，现代（大型）数据库系统一般不采用此技术。

②恢复技术2：以后备副本和运行记录为基础的恢复技术

运行记录/日志（log）：从数据库建立并开始运行起，记录全部提交事务对数据库的所有更新，包括以下内容：

事务及其状态：

事务标识（TID）

事务结束方式：提交 / 撤销

前像 & 后像

前像（Before Image, BI）：每次更新时，数据块的旧值。【插入时，BI为空】

后像（After Image, AI）：每次更新时，数据块的新值。【删除时，AI为空】

方法：

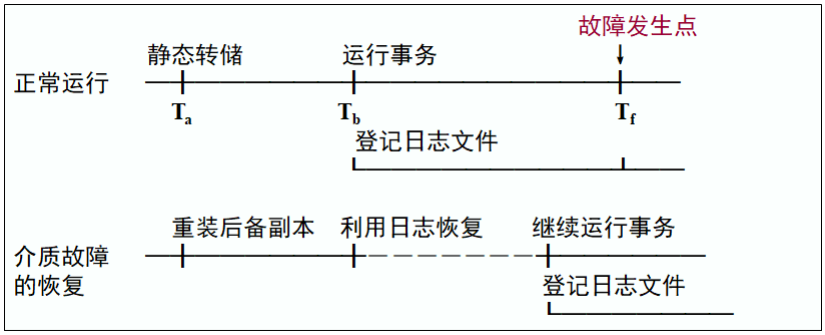
有了BI，如果需要，可使数据库恢复到更新前的状态（可撤销更新）—— 称**撤销**（undo）；

有了AI，如果需要，可使数据库恢复到更新后的状态（可重做更新）—— 称**重做**（redo）。

故当发生故障时，可通过以下方法来完全恢复数据库：

若**数据库未遭破坏**，则从最近一致状态开始，对未提交事务进行undo—**向后恢复（**backward recovery），对已提交事务进行redo—向前恢复（forward recovery）

若**数据库遭破坏**，则先重装最近的后备副本，再对自该后备副本以来的所有已提交事务进行redo。



特点：

实现技术较为复杂。

建立日志的过程持久地影响数据库系统的运行性能，也花费较大的存储空间（但存储设备愈来愈廉价）。

能将数据库恢复到最近一致状态（从不丢失更新）。

结论：恢复技术2 （backup + log）是最典型、常用的恢复技术，绝大多数商用DBMS均采用这种技术。

③恢复技术3：基于多副本的恢复技术

独立故障模式（independent failure modes）：不致于因同一差错而导致多个副本同时发生故障（因为支持环境是独立的）。

方法：系统中保持多个具有独立故障模式的数据库副本，互为备份、互为恢复的依据。

例如：镜像（mirroring）技术，包括：mirrored disks / mirrored files，采用“同时写，任选读”——对性能影响不大，有时可能反而提高性能。

特点：

成本高（用户要购置双倍/多倍硬件）；

恢复基本上由系统自动进行；

总能保持DB的一致状态；

仍然可能冒这样的（不太可能发生，但还是有可能发生）危险：“全军覆灭”→“无可救药”。

结论：

绝大多数商用DBMS不支持这种技术，如Oracle不支持镜像文件功能（控制文件及日志文件例外）；

较适合运用于分布式数据库环境：如：DB Server 双工 / 镜像。

二、日志结构与机制

1.日志记录的基本内容

活动事务表（ATL——active transaction list）：记录正在执行、尚未提交的事务标识（TID——transaction identifier）。

提交事务表（CTL ——committed transaction list）：记录已提交事务的标识。

当一个事务被提交结束时，需要做：① TID→CTL； ② Remove TID from ATL

前像（BI）文件：记录所有被更新的数据块之旧值（BI），用于进行undo操作。

后像（AI）文件：记录所有被更新的数据块之新值（AI），用于进行redo操作。

2.Oracle中日志及相关机制

ATL, (CTL), BI文件→**回滚段文件（Rollback Segment File）**

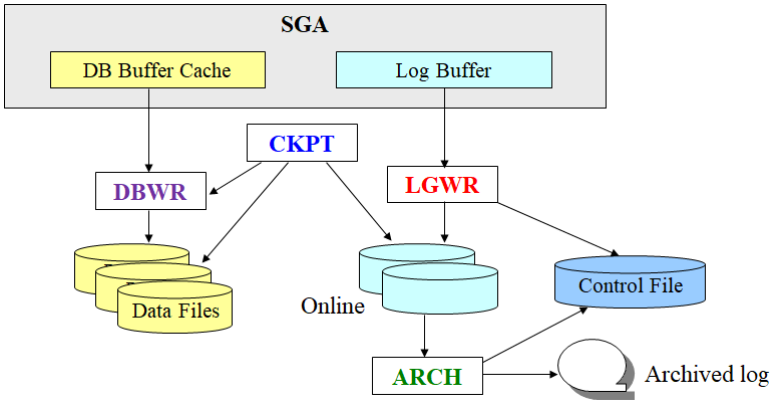
AI文件称**重做日志文件（Redo Log File）**

每个Oracle重做日志是一组文件。系统中总是保持至少两个文件组，以循环方式写入日志项（即一个填满就用另一个），这些日志文件称**在线日志（Online Redo Log）**。当一个在线日志文件被填满后，可以将其复制到离线存储设备（磁带）上，称为**归档日志（Archived Redo Log）**。

Oracle在线日志文件可以被镜像！

3.Oracle的几个后台进程

DBWR进程—负责数据库写入　ARCH进程—负责形成归档日志
LGWR进程—负责形成在线日志　CKPT进程—负责产生检查点



三、更新事务的执行与恢复

1.规则

为保证数据库是可恢复的，在系统具有日志机制后，更新事务的执行应遵守下列两条规则：
提交规则（Commit Rule）：后像AI必须在事务提交前写入非易失存储器（即数据库或日志文件）。
　　通常是立即写入DB Buffer Cache和log文件，以便当发生故障时，能通过redo而恢复。
先记后写规则（Log Ahead Rule）：如果后像AI在事务提交前写入数据库，则必须首先把前像BI记入日志。
　　以便一旦事务在提交前瞬间失败时，能通过undo而恢复。
　　为什么要先记后写?
　　写数据库和写日志文件是两个操作，两个操作之间可能发生故障
　　如果先写了数据库修改，但在日志文件中没有登记下这个修改，则以后就无法恢复这个修改（因此必须把老内容"留底"）
　　如果先写日志，但没有修改数据库，按日志文件恢复时只不过是多执行一次没意义的undo操作，并不会影响数据库的正确性

【后像（AI）写入数据库的时间不同】

2.方案1：后像在事务提交前完全写入数据库

(1) TID→ATL
(2) BI→Log　　/* 先记后写规则 /
*(3) AI→DB, Log　　/ 提交规则 /　/ AI直接写入DB */*
(4) TID→CTL
(5) 从ATL删除TID
当事务执行过程中发生故障时，根据ATL和CTL中是否有该事务的TID，可采取不同的恢复措施。如下表所示：

ATL	CTL	事务所处状态	恢复措施
有	-	(1)已完成，但(4)尚未完成	(1) 若 BI已写入日志则undo，否则无需undo；(2) 从ATL删除TID
有	**有**	(4)执行完	从ATL删除TID
-	**有**	(5)执行完	无需处理

3.方案2：后像在事务提交后才写入数据库

(1) TID→ATL
(2) AI→Log /* 提交规则 / / AI先写入日志文件 /
(3) TID→CTL
(4) AI→DB / AI后写入DB */
(5) 从ATL删除TID
后像在事务提交前未写入数据库，根据先记后写原则，不必记录前像（BI）。 不同情况下的恢复措施如下表所示：

ATL	CTL	事务所处状态	恢复措施
有	-	(1)已完成，但(3)尚未完成	从ATL删除TID
有	**有**	(3)已完成，但(5)尚未完成	(1) redo；(2) 从ATL删除TID
-	**有**	(5)执行完	无需处理

4.方案3：后像在事务提交前后写入数据库

(1) TID→ATL
(2) AI, BI→Log /* 满足两条规则 / / AI先写入日志 /
(3) AI→DB / AI部分写入DB /
(4) TID→CTL
(5) AI→DB / AI全部写入DB/
(6) 从ATL删除TID
不同情况下的恢复措施如下表所示：

ATL	CTL	事务所处状态	恢复措施
有	-	(1)已完成，但(4)尚未完成	(1) 若 BI已写入日志则undo，否则无需undo；(2) 从ATL删除TID
有	**有**	(4)已完成，但(6)尚未完成	(1) redo；(2) 从ATL删除TID
-	**有**	(6)执行完	无需处理

5.三种方案之间的比较

方案1：后像在事务提交前完全写入数据库
方案2：后像在事务提交后才写入数据库
方案3：后像在事务提交前后写入数据库

方案	redo	undo	AI	BI	实际中使用情况
1	—	√	—	√	未被采用
2	√	—	√	—	目前流行
3	√	√	√	√	现已不用

四、各类故障的恢复策略

在Oracle中，以下两种故障的恢复统称为**实例恢复（Instance Recovery）**：

1.事务故障的恢复

如前所述，事务故障必然发生在事务提交之前，故恢复措施/步骤：

如果需要，则进行undo操作（因为BI已写入log）；

从ATL中删除故障事务的TID，释放其所占的资源。

以上步骤均由系统自动实施，无需DBA介入。

2.系统故障的恢复

首先要使系统恢复正常运行；其次在故障发生时，可能有已提交事务的更新丢失了，可能有未提交的事务夭折了，故恢复措施/步骤：

重新启动OS和DBMS；（由DBA实施）

利用日志中的前像（BI）对未提交的事务进行undo操作（称向后恢复），利用日志中的后像（AI）对已提交的事务进行redo操作（称向前恢复）。（由系统自动实施）

> 在发生系统故障时，未提交的事务是有限的，故undo工作量较少；但已提交的事务是大量的，故redo工作量很大。而且，到底从哪一个时间点开始redo呢？——太前了很浪费，太后了会丢失更新。
> DBMS中一般都设置检查点（checkpoint，CP）机制：每隔一段时间（如一分钟）产生一个CP，在CP上，DBMS强制将已提交事务尚未完成的数据库更新（即在内存DB Buffer Cache中已修改的数据块的后像AI）写入数据库中；在日志的提交事务表（CTL）中记下CP。•例如，Oracle中CKPT进程就承担此工作。
> 在上一个CP之后、最近一个CP之前提交的事务就不必在向前恢复时进行redo了——大大减少了向前恢复的工作量！

3.介质故障的恢复

在Oracle中，称**介质恢复（media recovery）**

由于介质故障的特点是DB已不可用了，故必须首先利用**后备副本（backup）**来恢复DB，然后，再利用日志（log）进行向前恢复。故恢复措施/步骤：

修复系统，必要时更换磁盘；（由系统管理员或DBA实施）

重启系统OS和DBMS；（由系统管理员或DBA实施）

加载最近后备（backup）；（由DBA实施）

利用日志中的后像（AI）重做（redo）自该backup后的所有已提交事务的更新操作。（由系统自动实施）

小结

> 为了确保事务满足ACID性质，DBMS必须对事务故障、系统故障和介质故障进行恢复
> 恢复中最经常使用的技术：数据库的后备副本；日志
> 恢复的基本原理：利用存储在后备副本、日志文件与/或数据库镜像中的冗余数据来将数据库恢复到（最近）一致状态
> 常用恢复措施
> 事务故障的恢复：UNDO
> 系统故障的恢复：UNDO + REDO
> 介质故障的恢复：重装最近Backup + REDO
> 提高恢复效率的关键技术 检查点（CP）技术：
> 可以提高系统故障的恢复效率；
> 在一定程度上提高利用动态转储进行介质故障恢复的效率

7.2 并发控制

一、并发控制概述

1.并发访问

数据库是一个多用户共享的系统，**多个事务的执行方式**：

串行访问（serial access）：DBMS一次只接纳一个事务，事务串行地被执行，即一个事务结束后另一个事务才开始。**不能充分利用系统资源，不能发挥DB共享资源的特点**

并发访问（concurrent access）：DBMS同时接纳多个事务，事务在时间上重叠地执行

交叉并发方式（interleaved concurrency）：并行事务的并行操作轮流交叉运行，是单处理机系统中的并发方式。能够减少处理机的空闲时间，提高系统的效率。

同时并发方式（simultaneous concurrency）：多处理机系统中，每个处理机可以运行一个事务，多个处理机可以同时运行多个事务，实现多个事务真正的并行运行。最理想的、更复杂的并发方式，但受制于硬件环境。

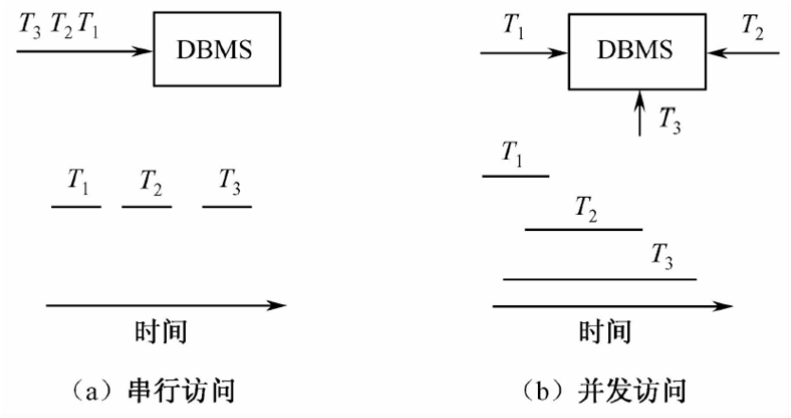


图 7-12　串行访问和并发访问

数据库系统中，事务必然是并发访问的。

性能方面考虑：

多个用户往往同时存取数据库，实际上，即往往同时有多个事务向DBMS提出申请，如果DBMS串行地提供服务，则必然会“**排队等待**”，这样极大地影响系统性能（**吞吐率**）。特别是“**短事务**”，更感觉“响应时间太慢”。

资源利用率方面考虑：

一个事务在执行过程中，不同执行阶段需不同的资源（CPU、I/O、通信...）。如果事务串行执行，必然有“**资源闲置**”，影响资源利用率；如果并行执行，既充分利用了系统资源，又可缩短响应时间。

2.并发所引起的不一致问题

①丢失更新（lost update）

源于：写-写冲突（write-write conflict）

> 指事务1与事务2从数据库中读取同一数据进行更新，且并发地写入数据库；事务2的提交结果破坏了事务1的提交结果，导致事务1写入的数据更新被丢失。
> 现象：一个事务的更新操作破坏了另一个事务的更新结果
> 原因：对多个事务并发更新同一个数据对象的情况未加控制

②读脏数据（dirty read）

源于：读-写冲突（read-write conflict）

事务1修改某一数据后将其写回数据库；事务2读取同一数据后，事务1由于某种原因被撤消，其修改过的数据恢复原值。事务2读到的数据就与数据库中的数据不一致，是不正确的数据，又称为“脏”数据。
现象：读到了错误的数据（即与数据库中不相符的数据）
原因：一个事务读取了另一个事务未提交完成的更新结果

③读值不可复现（unrepeatable read）

源于：读-写冲突

指事务1读取数据后，事务2执行更新操作，致使事务1无法再现前一次读取结果。
三类读值不可复现，事务1读取某一数据后：
事务2对其做了修改，当事务1再次读该数据时，得到与前一次不同的值。
事务2删除了其中部分数据，当事务1再次读取该数据时，发现某些数据神密地消失了。
事务2插入了一些记录，当事务1再次按相同条件读取该数据时，发现多了一些记录。
后两类读值不可复现有时也称发生了幻影行（phantom rows）现象
现象：在一个事务的执行过程中，前后两次读同一个数据对象所获得的值出现了不一致
原因：在两次‘读’操作之间插入了另一个事务的‘写’操作

3.并发控制

并发控制（concurrency control）：控制事务的并发访问，以避免访问冲突所引起的数据不一致，保证数据库始终处于一致状态。

DBMS必须配备并发控制机制。
并发控制机制是衡量一个DBMS性能的重要标志之一。

4.并发控制的正确性准则

调度（schedule）：在某一时刻，DBMS对并发访问的一组事务 {T1,T2,...,Tn}的所有操作步骤的顺序的一个安排S，可形式地表示成： S = ... Ri(x) ... Wj(x) ... Rk(y) ... **其中，i, j, k∈{1, 2, ..., n} 表示事务编号；事务中有读、写操作；Ri(x)表示事务Ti对数据x的一个读操作，Wj(x)表示事务Tj对数据x的一个写操作， Rk(y)表示事务Tk对数据y的一个读操作。**可见对同一个事务集 {T1,T2,...,Tn}，可有多种调度S。

等价的调度（equivalent schedule）：给定对同一个事务集的两个调度S1和S2，如果从DB的任何初始状态开始， S1和S2所有从DB中**读出的数据都是一样的**，且留给DB的**最终状态也是一样的**，则称S1和S2是等价的调度。这种等价称**目标等价**（view equivalence——a schedule equivalence that is based purely on the read and write operations）。

如前所述，有两种操作对是冲突的，分别是：
读-写冲突，表示为：Ri(x)和Wj(x)
写-写冲突，表示为：Wi(x)和Wj(x)
另四种操作对是不冲突的，分别是：
Ri(x)和Rj(x)，Ri(x)和Rj(y)，Ri(x)和Wj(y)，Wi(x)和Wj(y)
不冲突的操作之间可以相互调换次序，不会影响执行结果。
冲突等价（conflict equivalence）：通过调换（不同事务间的）不冲突操作的次序而得到的新调度，称为**冲突等价的调度。**这种等价称**冲突等价**（a schedule equivalence that is based on a series of swaps of nonconflicting instructions）。
可见**目标等价更普遍：冲突等价的调度→目标等价的调度**

串行调度（serial schedule）：导致事务**串行访问**的调度。
并行调度（concurrent schedule）：导致事务**并行访问**的调度。
可串行化调度（serializable schedule）：如果一个**（并行）调度**与某个**串行调度**是**等价的**，则称它为可串行化调度。
上述现象称为"可串行化（serializability）"。同样有：冲突可串行化（conflict serializability）vs. 目标可串行化（view serializability ）
冲突可串行化→目标可串行化
由于串行调度导致事务的串行访问（事务间不会有相互影响），总能保持数据库的一致性，而可串行化调度与串行调度等价，因此，**可串行化调度也总能保持数据库的一致性。**
"可串行化" 是并行事务正确性的唯一准则！

- 例1：设有三个事务：
- T_1：$R_1(y)$ 和 T_2：$R_2(x), W_2(y)$ 和 T_3：$W_3(x)$
  则对 $\{T_1, T_2, T_3\}$ 的一个并行调度：

$$Sc = R_2(x)\ W_3(x)\ R_1(y)\ W_2(y)$$
$$\longleftrightarrow R_2(x)\ R_1(y)\ W_3(x)\ W_2(y)$$
$$\longleftrightarrow R_1(y)\ R_2(x)\ W_2(y)\ W_3(x) = T_1T_2T_3 = Ss$$

（串行调度）

故并行调度Sc是（冲突）可串行化的调度。

- 例2：设有三个事务：
- T_1：$R_1(x), W_1(x)$ 和 T_2：$W_2(x)$ 和 T_3：$W_3(x)$
- 则对 $\{T_1, T_2, T_3\}$的一个并行调度：

$Sc = R_1(x)\ W_2(x)\ W_1(x)\ W_3(x)$ 无法调换（均是冲突操作）

- 但根据“目标等价”的定义【Sc和Ss所有从DB中读出的数据都是一样的，且留给DB的最终状态也是一样的】，并行调度Sc与下列串行调度Ss是“等价”的：

$Ss = R_1(x)\ W_1(x)\ W_2(x)\ W_3(x) = T_1T_2T_3$ （串行调度）

由以上两例说明：

冲突等价/冲突可串行化调度 $\rightleftarrows$ 目标等价/目标可串行化调度

结论1：“可串行化调度”能保持DB的一致状态，因此，一般DBMS都以“可串行化调度”作为并发控制的正确性准则

结论2：“目标可串行化调度”比“冲突可串行化调度”更普遍。但由于前者的测试问题是NP完全的，而后者可通过简单算法来判断【例如，对前趋图（precedence graph）运用拓扑排序法（topological sorting）】，故一般总是以“冲突可串行化”作为并发控制的正确性准则【即使明知会漏掉一些可串行化调度】。

结论3：然而，在实际系统中，不可能总是“通过检测是否可串行化”来控制并发访问、保证并发控制的正确性，而是通过让事务遵守“加锁协议”（locking protocol）的方法来确保并发控制遵循正确性准则。□【因为：事务是随机到达和退出的，没有一个固定不变的事务集等待着DBMS去调度，或者说事务集一直在动态变化，故频繁“检测”是不现实的！】

结论4：一旦引入“加锁”机制，就会面临死锁（dead lock）和活锁（live lock）问题，因此产生了如何检测以及如何预防死锁的问题。

二、加锁协议

在事务并发调度时，要求遵守“加锁协议”，即：在执行任何事务的任一操作（R/W）前对操作对象（数据对象）进行加锁，并遵守一定的协议，就可保证并发调度是（冲突）可串行化的，从而达到并发控制的目标。

加锁的作用：

确保在一段时间内禁止其它事务对被加锁的数据对象执行某些类型的操作。**（由加锁的类型来决定）**

表明持有该锁的事务对被加锁的数据对象将要执行什么类型的操作。**（由加锁协议来决定）**

有多种“加锁协议”：

使用X锁的加锁协议

使用（S, X）锁的加锁协议

使用（S, U, X）锁的加锁协议

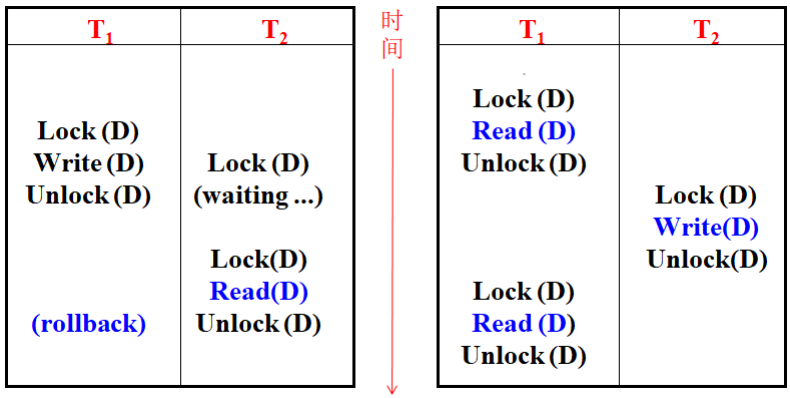
1.使用X锁的加锁协议

X锁：排它锁（eXclusive Lock），事务对数据对象实施独占式的读、写操作【缺点：系统的并发度很低】。可以用一个相容矩阵（compatibility matrix）来定义X锁：

	其他事务已拥有的锁	
	无	X
当前事务的锁请求 X	Yes	No

协议：① 任何事务的任何操作（R/W）在能够执行之前，必须先获得此操作对象的X锁（称：加X锁），操作完成后要释放X锁。② 每个X锁必须保持到该事务结束（end of transaction, EOT）时才能释放。【说明：协议②同样适合于其他任何加锁协议。不再重述！】

如果仅有协议①，会产生问题：读脏数据、读值不可复现、连锁回滚【教材Page 153-154中解释以及图7-20和图7-21】



定义：合式事务（well-formed transaction）

一个事务如果遵守"先加锁，后操作"的原则，则此事务称为合式事务。

合式事务是保证并发事务正确执行的基本条件。

定义：两段事务和两段封锁（2PL）协议

在一个事务中，如果所有加锁动作都在所有释放锁动作之前，则称这样的事务称为两段事务（two-phase transaction）—— 隐含了X锁的加锁协议①

上述加锁限制称为两段封锁协议（Two-Phase Locking Protocol，2PL协议）

事务T：$Lock(A)\ Lock(C)\ Lock(B)\ Unlock(C)\ Unlock(A)\ Unlock(B)$

锁增长（growing）阶段　　锁收缩（shrinking）阶段

定理：如果所有事务都是合式、两段事务，则它们的任何调度都是可串行化的。

证明见教材（反证法）：思路——前趋图中是否有回路。

注：①有的教科书上将"合式"作为事务的基本特性，则上述定理修改为：如果所有事务都遵守两段封锁协议，则它们的任何调度都是可串行化的。

2PL协议是充分条件，并不是调度可串行化的必要条件。

②上述结论同样适用于下文所有加锁协议。

使用X锁的加锁协议的特点：

简单，只使用一种锁

并发性较差，即使是"读-读"也要用X锁来"排它"

2.使用（S, X）锁的加锁协议

X锁：排它锁（eXclusive Lock），用于写操作。

S锁：共享锁（Sharing Lock），用于读操作。

可以用一个相容矩阵来定义：

		其它事务已持有的锁		
		X	S	无
当前事务的锁请求	X	No	No	Yes
	S	No	Yes	Yes

提高并发度（允许读-读并发），但可能发生"活锁"现象

活锁（live lock）：如果不断有事务请求对某个数据对象的S锁，该数据对象始终被S锁占有，而X锁请求迟迟不能获准，这种现象称为活锁。【对系统的性能有不良影响！】

活锁预防：在加锁协议中运用先来先服务（First Come, First Serve）原则。

3.使用（S, U, X）锁的加锁协议

X锁：排它锁（eXclusive Lock），用于写操作。
S锁：共享锁（Sharing Lock），用于读操作。
U锁：共享更新锁（Sharing Update Lock, SU），被当前事务用来"预置"对某个数据对象将实施更新的权力，并防止其它事务对该数据对象实施写操作或加X锁；在最后更新写入前，系统将U升级为X而排它。【提高了事务并发度！】

		其它事务已持有的锁			
		X	U	S	无
当前事务的锁请求	X	No	No	No	Yes
	U	No	No	Yes	Yes
	S	No	Yes	Yes	Yes

允许写前-读并发。但仍可能发生"活锁"（S一直占有，U久久不能升级为X）
——规定"先来先服务"原则！

4.使用加锁的并发控制技术

加锁由DBMS统一管理。
DBMS的加锁管理器中维护着一张**锁表**，其中记录各个数据对象当前的加锁情况：
锁的**持有情况**：有哪些'事务'在哪些'数据对象'上已持有什么类型的'封锁'?
锁的**申请情况**：有哪些'事务'正在申请哪些'数据对象'上的什么类型的'封锁'?

事务如果需要对某个数据对象进行操作，则必须向DBMS申请对该数据对象的加锁——**DBMS根据加锁协议以及"锁表"中信息，同意其申请或令其等待。**
锁表是DBMS的公共资源，且访问频繁，通常置于公共内存区。如果系统故障，锁表内容也随之故障，无保留价值。

三、多粒度封锁与意向锁（选学）

1.封锁粒度（locking granularity）

封锁的数据对象的大小，可以是数据库中的逻辑数据单元，或物理数据单元。以关系数据库为例，可采用的封锁粒度：

逻辑数据单元：属性值（集合），元组，关系； 索引项，索引文件；整个数据库
物理数据单元：页，块

2.单粒度封锁（single-granularity locking, SGL）

固定一种粒度，如"关系"（即"表"）。
SGL简单。但是，若封锁粒度太大，则并发度低；若封锁粒度太小，锁太多（对大操作），维护开销大。

3.多粒度封锁（multiple-granularity locking, MGL）

可有多种粒度，根据需要选用。
MGL复杂。但灵活性大。**一般大型DBMS都支持MGL。**

4.Oracle封锁

Oracle支持Table/Row两级封锁，因此有下列Locks：
[Table] Exclusive
[Table] Share
[Table] Share Update
Row Share

Row Exclusive
Share Row Exclusive

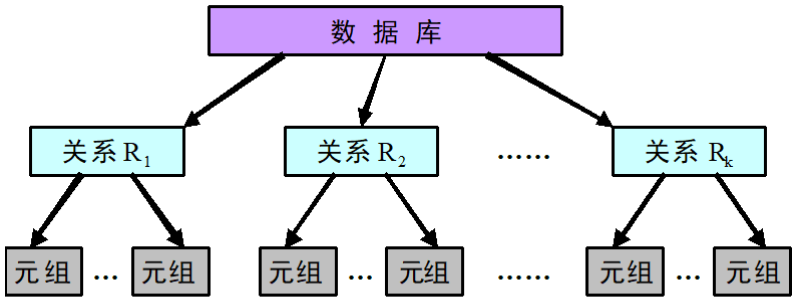
5.多粒度封锁（MGL）的不同目标

通过增大'封锁粒度'来减少'锁'的数量，从而降低并发控制的开销。
通过减小'封锁粒度'来缩小被封锁的数据范围，从而减少封锁冲突现象，提高系统并发度。

6.多粒度封锁（MGL）协议的实现

按不同的封锁粒度构造一棵多粒度树，以树中每个结点作为封锁对象。

7.多粒度树（以关系数据库为例）



三个层次的多粒度树

8.多粒度封锁带来的问题及解决方法

一个数据对象可能以两种方式被封锁：
显式封锁（explicit locking)
　　直接对"多粒度树"中的某个结点（即：数据对象，如：一个关系）实施加锁 。
隐式封锁（implicit locking)
　　对"多粒度树"中的某个结点（如：一个关系）加锁，意味着对该结点的所有子孙结点（如：该关系中的所有元组）也（隐式地）加了同样类型的锁 。
　　有了隐式封锁，"锁冲突"检测就复杂多了！
　　于是，引入了一种新的锁：意向锁（intent lock）。

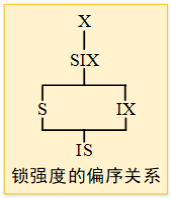
9.意向锁（intention lock)

IBM System R中首先采用
IS锁——意向共享锁（intent share lock）：一个数据对象加了IS锁，表示它的某些子孙拟加或已加了S锁。
IX锁——意向排它锁（intent exclusive lock）：一个数据对象加了IX锁，表示它的某些子孙拟加或已加了X锁。
SIX锁——共享意向排它锁：SIX = S + IX，表示对结点本身加S锁，并且它的某些子孙拟加或已加了X锁。

意向锁的使用规则：
　若要对一个数据对象加某种锁，则必须先对该数据对象的所有上级数据对象加相应的意向锁。
　若对一个数据对象加了某种意向锁，则说明该数据对象的下层数据对象正在被加某种锁。
　次序：自上而下申请锁，自下而上释放锁。

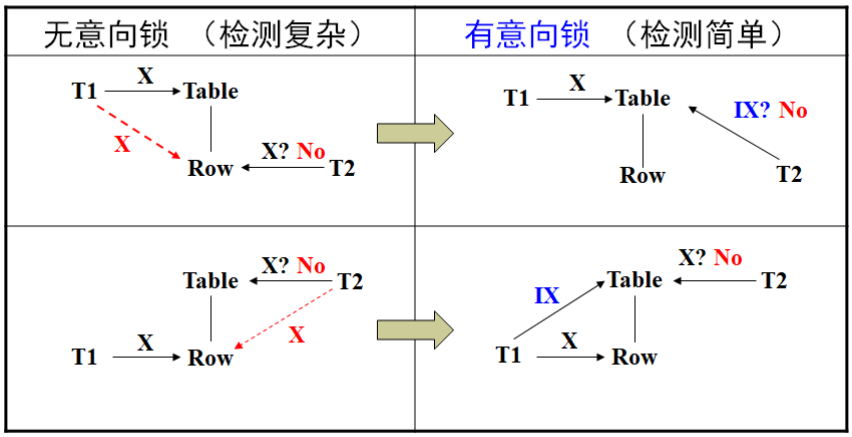
■ 带有意向锁的锁相容矩阵：

		其它事务已持有的锁					
		无	S	X	IS	IX	SIX
当前事务申请的锁	S	Yes	Yes	No	Yes	No	No
	X	Yes	No	No	No	No	No
	IS	Yes	Yes	No	Yes	Yes	Yes
	IX	Yes	No	No	Yes	Yes	No
	SIX	Yes	No	No	Yes	No	No



Yes：表示相容的锁请求　　No：表示不相容的锁请求

■ 例：简化“锁冲突”检测



10.Oracle中，锁如何获得/释放?

■ 有一条扩充的SQL命令，事务可用来显式地获得某种封锁。

```
LOCK TABLE <表标识> [, <表标识> …]
IN ( EXCLUSIVE                    ) MODE [NOWAIT];
   ( SHARE                        )
   ( SHARE UPDATE                 )
   ( ROW EXCLUSIVE                )
   ( ROW SHARE                    )
   ( SHARE ROW EXCLUSIVE          )
```

X锁

```
获得
    显式方式：LOCK TABLE语句
    隐式方式：执行如下SQL语句：
                INSERT INTO …;
                UPDATE …;
                DELETE FROM …;
释放，四种方式：
    COMMIT/ROLLBACK（即事务结束时）；
    DDL操作；
    程序终止运行；
    LOGOFF（即终止会话时）
```

S锁

```
获得：显式方式： LOCK TABLE语句
释放：同X锁的释放
```

SU锁

```
获得：
    显式方式： LOCK TABLE语句
    隐式方式：执行  SELECT … FOR UPDATE语句
释放：
    同X锁的释放
```

四、死锁的检测、处理和预防（选学）

1.死锁（deadlock）

有了封锁就有可能会发生死锁：两个事务对锁形成“循环等待”就产生了死锁。

死锁的例子：
假设在数据库中有两个数据对象A和B，并有两个事务：
事务T1：根据读到的A的值来修改B的值，其数据库访问的操作流程如下：R1(A);R1(B);W1(B)
事务T2：根据读到的B的值来修改A的值，其数据库访问的操作流程如下：R2(B);R2(A);W2(A)
如果采用如下的调度来并发执行事务T1和T2，将产生死锁现象：
R1(A);R1(B);R2(B);W1(B);R2(A);W2(A)□

T1和T2的调度：**R1(A);R1(B);R2(B);W1(B);R2(A);W2(A)**

时刻	事务T_1	事务T_2	A的加锁状态	B的加锁状态
1	$Slock_1(A)$		$S(T_1)$	
2	$R_1(A)$			
3	$Slock_1(B)$			$S(T_1)$
4	$R_1(B)$			
5		$Slock_2(B)$		$S(T_1, T_2)$
6		$R_2(B)$		
7	$Xlock_1(B)$			
8	Wait…	$Slock_2(A)$	$S(T_1, T_2)$	
9	Wait…	$R_2(A)$		
10	Wait…	$Xlock_2(A)$		
11	Wait…	Wait…		
12	Wait…	Wait…		

2.死锁的检测和处理

检测方法一：超时法

规定一个“等待时限”，如果一个事务因等待获得锁而超过该“等待时限”，则认为发生了死锁。

优点：简单。

缺点：久等：早已发生死锁，非要等到时限到才能发现；误判：尚未发生死锁，再等一会即可推进，但时限已到，错误地认为是死锁了。

检测方法二：等待图法

使用一个称“等待图”（wait-for graph）的有向图（OS中学过），周期性地检测图中是否有回路，若有，则发生了死锁。

优点：总能检测出来，并且从不会“误判”。

缺点：久等：当检测周期较长时。检测代价大：当检测周期较短时，需频繁检测。

处理方法：选牺牲品让路

在循环等待的事务中，选一个事务作为牺牲品（victim）让路：

撤消该牺牲品，并在屏幕上通知用户，事后由用户重新提交；

暂时撤消（即挂起），稍后由DBMS重启该事务再试。

选择牺牲品的几个可选原则：

最迟交付事务；

获得锁较少事务；

撤消代价最小事务。

3.死锁的预防

OS中防止进程死锁的思路：防止循环等待

一次申请所有的锁；

规定一个封锁次序。

但是，上述策略在数据库系统中不太实际。

数据库系统中较实际的方法：

选择性地“回退再试”or 称“卷回重执”（Rollback and Retry）事务。

```
卷回重执（Rollback and Retry）
每个事务被DBMS接纳时被赋予一个时戳（time stamp, TS）。
     若ts(T1) < ts(T2)，则T1是“年老”事务，T2是“年幼”事务。
设T2已持有某对象的锁，当T1申请该对象的锁而发生冲突时可采用两种策略
等待-死亡（wait-die）策略  【老者等，幼者亡】
    if  ts(T1) < ts(T2)  then  T1 waits;              // T1老，则等待
            老       幼    else { Rollback T1;       // T1幼，则死亡
                                  Restart T1 with the same ts(T1); // 重执T1
                                };【幼者总会变老，而不会总死亡】
击伤-等待（wound-wait）策略 【幼者等，老者抢】
    if  ts(T1) > ts(T2)  then  T1 waits;              // T1幼，则等待
            幼       老    else { Rollback T2;       // T1老，则击伤幼者T2
                                  Restart T2 with the same ts(T2); // 重执T2
                                };【幼者总会变老，而不会总被击伤】
以上两种策略的共性：总是将年轻事务作为牺牲品，卷回而重执。
——显然比一冲突就卷回这种“一触即退”的方案要好！
```