

第五章

事务处理、并发控制与故障恢复技术

5. 1 事务处理

5.1 事务处理

5.1.1 事务

5.1.2 事务的性质

5.1.3 事务活动

5.1.4 有关事务的语句

5.1.5 事务的组成

5.1.1 事务

口 事务 (transaction)

- 由某个用户所执行的一个不能被打断的对数据库的操作序列被称为‘**事务**’
 - ‘**事务**’ 是应用程序访问数据库的基本逻辑工作单位
 - ‘**事务**’ 通常由一组对数据库的访问操作组成，在执行过程中按照预定的次序顺序执行
 - 一个‘**事务**’的执行过程是串行的，它将数据库从一个旧的一致性状态转换到一个新的的一致性状态。在‘**事务**’的执行过程中，数据库中的数据可能有不一致的现象，但在‘**事务**’执行结束时，系统将保证数据库中数据的一致性

5.1.1 事务

- **[例]**银行的转帐业务：根据输入的两个银行存款帐号A和B，以及转帐金额X，将帐号A的金额减去X，帐号B的金额增加X。其处理过程如下（其中 $READ(A)$ 表示将帐号A的金额读入内存变量A， $WRITE(A)$ 表示将内存变量A的值作为帐号A的金额写入数据库）：

```
READ(A);  
IF (A >= X)  
THEN BEGIN  
    A := A - X;  
    WRITE(A);  
    READ(B);  
    B := B + X;  
    WRITE(B);  
END
```

■ 该事务的DB访问操作为：

```
READ(A);  
WRITE(A);  
READ(B);  
WRITE(B);
```

对该事务而言，数据库中数据的一致性就是指：帐号A和帐号B的总金额之和不变

5.1.2 事务的性质

事务具有四个特性，简称为事务的 **ACID** 特性，
也被称为事务的执行过程必须满足的四条准则

1. 原子性 (**Atomicity**)
2. 一致性 (**Consistency**)
3. 隔离性 (**Isolation**)
4. 持久性 (**Durability**)

5.1.2 事务的性质

1. 原子性 (Atomicity)

- 在一个事务中，所有的数据库访问操作构成一个不可分割的操作序列，这些操作要么全部执行结束，要么一个都不要执行。（例：银行转帐）
- 数据库管理系统会自动维护用户事务执行的原子性
 - 事务管理子系统
 - 事务日志

5.1.2 事务的性质

2. 一致性 (Consistency)

- 一个事务的成功执行总是将数据库从一个一致的状态转换到另一个一致的状态
 - 数据库的‘状态’：指数据库中所有数据对象的当前取值情况
 - 数据库的一致的状态可以理解为数据库中所有数据的正确性，它要求数据库中的数据必须满足：
 - ❖ 在数据库中显式定义的各种完整性约束
 - ❖ 用户心目中的隐式数据约束

5.1.2 事务的性质

2. 一致性（续）

- 事务执行的‘一致性’原则基于这样一个假设：
 - 在一个事务开始执行之前数据库处于一个一致的状态，如果没有‘其它事务的干扰和系统故障’，那么当该事务执行结束时数据库仍然处于一致的状态
- 事务的‘一致性’特性由两方面完成
 - DBMS中的‘数据完整性保护’子系统
 - 编写事务的应用程序员

5.1.2 事务的性质

3. 隔离性 (Isolation)

- 一个事务的执行与并发执行的其它事务之间是相互独立的，互不干扰，这被称为事务执行的‘隔离性’
- 事务执行的‘隔离性’要求：
 - 多个事务并发执行的最终结果，应该与它们的某种串行执行的最终结果相等，这被称为并发事务的可串行化

5.1.2 事务的性质

3. 隔离性（续）

- 事务执行的‘隔离性’是由**DBMS**的并发控制子系统来实现的。
 - 在数据库系统中，当多个事务并发执行时，每个事务都可以看成只有自己在执行，只有当它与其它事务发生封锁冲突时，才会感觉到其它事务的存在
- 例如：**12306**联网售票系统

5.1.2 事务的性质

4. 持久性 (Durability)

- 一个事务一旦完成其全部操作后，它对数据库的所有更新应永久地反映在数据库中，即使以后系统发生故障也应该能够通过故障恢复来保留这个事务的执行结果
 - 事务的‘持久性’是由DBMS的恢复管理子系统实现的
- 数据库管理系统通过其‘事务管理’子系统（含‘并发控制’子系统）、‘恢复管理’子系统、‘数据完整性保护’子系统来实现事务的原子性(A)、一致性(C)、隔离性(I)和持久性(D)

5.1.3 事务活动

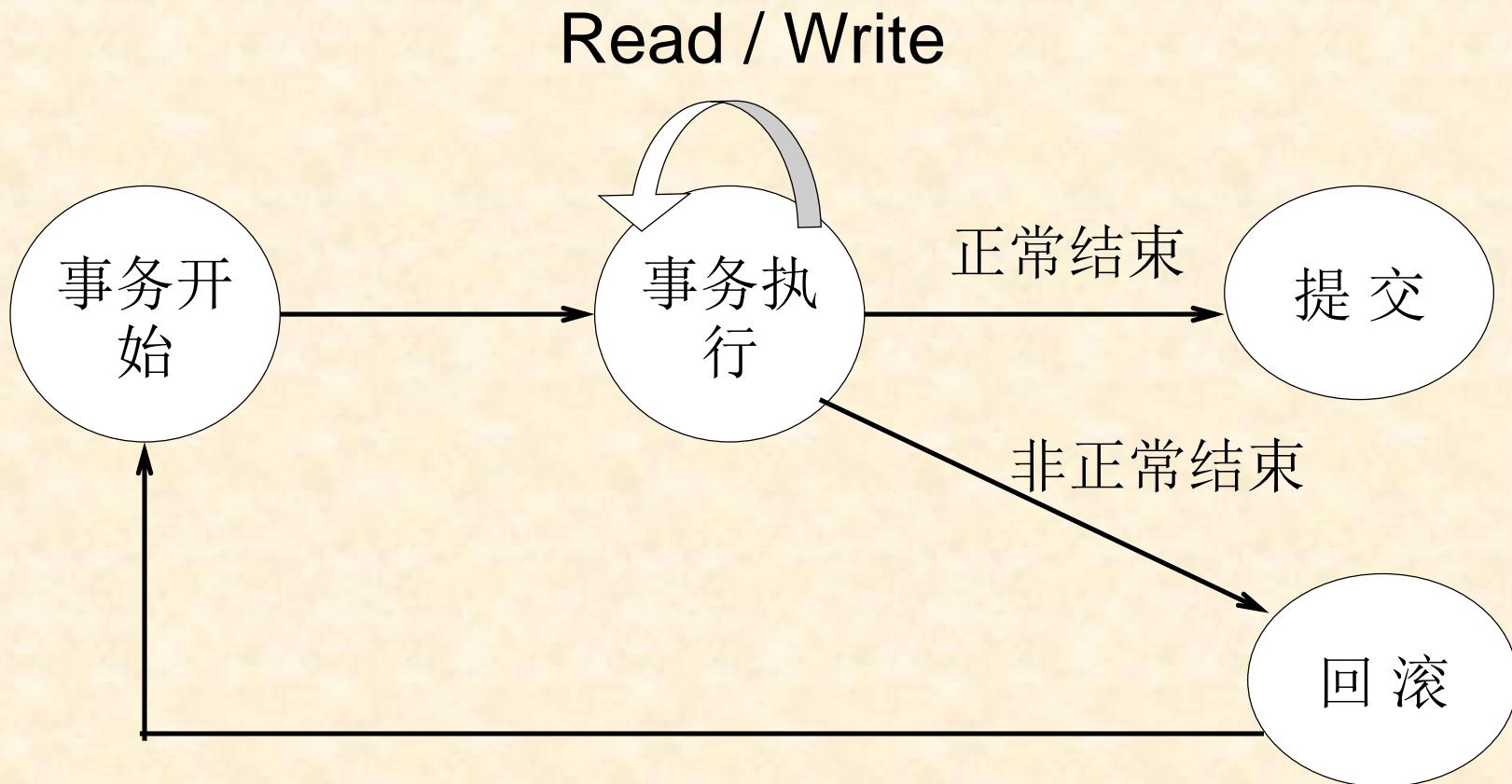
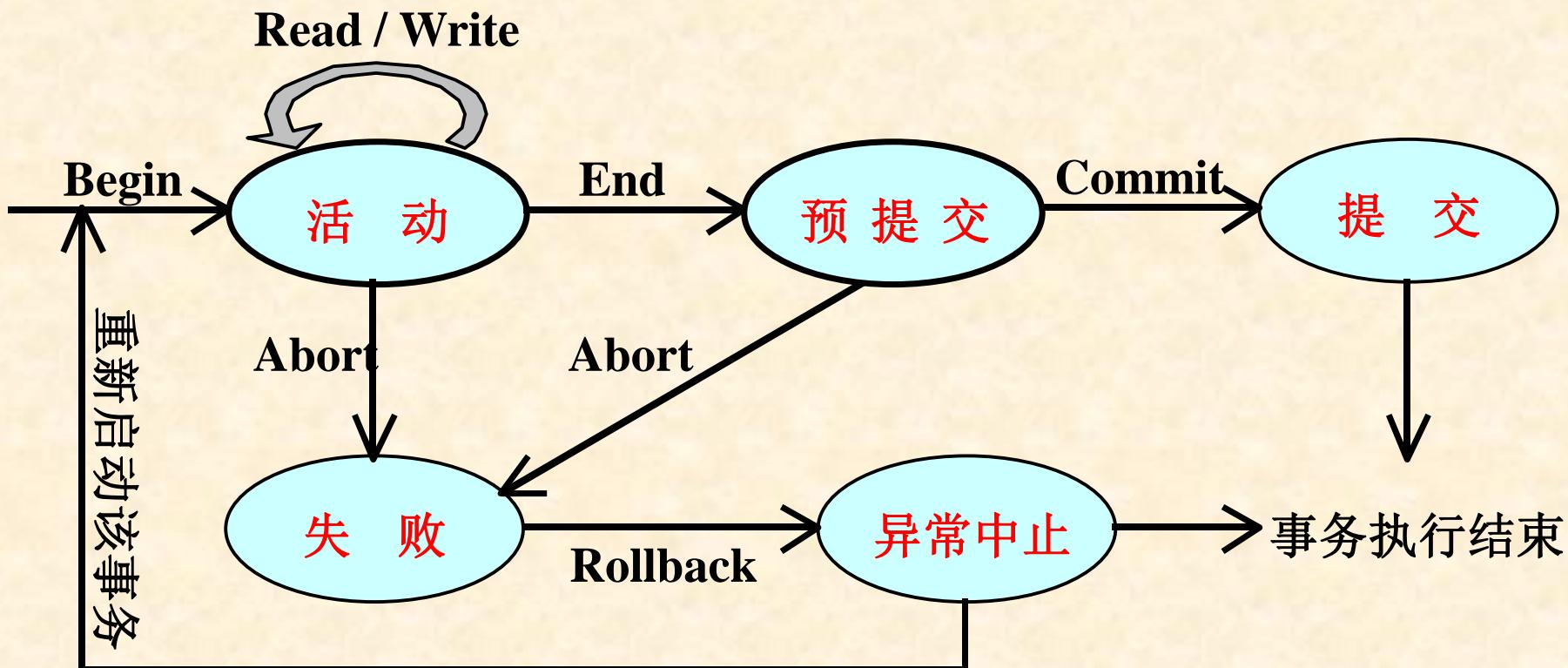


图5.1 事务活动过程图

5.1.3 事务活动

- 为了精确地描述一个事务的工作过程，我们建立了一个抽象的事务模型，以表示事务的状态变迁情况



事务的状态变迁图

5.1.3 事务活动

□ ‘活动’状态

- 事务在开始执行后，立即进入‘活动’状态。在‘活动’状态中，事务将执行对数据库的访问操作
- 在**DBMS**的事务管理子系统看来，用户事务对数据库的访问操作就是对数据库中数据的读/写操作

5.1.3 事务活动

□ (cont.)

- ‘读’ 操作
 - 将数据读入用户事务的私有工作区间
 - 如果该数据当前不在**DBMS**的系统缓冲区中，那么**DBMS**首先将该数据从磁盘读入系统缓冲区，然后再将其拷贝到用户事务的私有工作区
- ‘写’ 操作
 - 将修改后的数据‘写入’数据库
 - 这里的‘写’操作并不是立即将数据永久性地写入磁盘，很可能暂时存放在**DBMS**的系统缓冲区中

5.1.3 事务活动

□ ‘预提交’状态

➤ 当事务的最后一条访问语句执行结束之后，事务进入‘**预提交**’状态。此时事务对于数据库的访问操作虽然已经执行结束，但其对于数据的修改结果可能还在内存的系统缓冲区中，必须将其真正写入数据库的磁盘

5.1.3 事务活动

□ (cont.)

- 事务在‘**预提交**’阶段，必须确保将当前事务的所有修改操作的执行结果被真正写入到数据库的磁盘中去
 - 在所有‘写’磁盘操作执行结束后，事务就进入‘**提交**’状态
- 在‘**预提交**’阶段，虽然事务本身的操作命令已经执行结束，但是在写磁盘的过程中仍然会发生系统故障，从而导致当前事务的执行失败
 - 在‘**预提交**’失败后，当前事务也将被放弃(**abort**)，事务转而进入‘**失败**’状态

5.1.3 事务活动

□ ‘失败’状态

- 处于‘活动’状态的事务在顺利到达并执行完最后一条语句之前就中止执行，或者在‘预提交’状态下因发生系统故障而中止执行时，我们称事务进入‘失败’状态
 - 事务从‘活动’状态转变为‘失败’状态的原因
 - ❖ 应用程序(或用户)主动放弃(**abort**)当前事务
 - ❖ 因并发控制的原因而被放弃的事务，如：
 - 封锁申请的超时等待
 - 死锁
 - ❖ 发生系统故障
 - 事务从‘预提交’状态转变为‘失败’状态的原因
 - ❖ 发生系统故障

5.1.3 事务活动

□ ‘异常中止’状态

➤ 处于‘失败’状态的事务，很可能已对磁盘中的数据进行了一部分修改。为了保证事务的原子性，系统应该撤消(**undo**操作)该事务对数据库已作的修改。在撤消操作完成以后，事务将被打上一个放弃的标志(**aborted**)，转而进入‘异常中止’状态

- 回退(**rollback**)

- 对事务的撤消操作也称为事务的‘回退’或‘回滚’
- 事务的‘回退’由**DBMS**的‘恢复子系统’实现

- 在事务进入‘异常中止’状态后，系统有两种选择：
 - 作为一个新的事务，重新启动
 - 取消事务

5.1.3 事务活动

□ ‘提交’状态

- 事务进入‘预提交’状态后，‘并发控制子系统’将检查该事务与并发执行的其它事务之间是否发生干扰现象
- 在检查通过以后，系统执行提交(**commit**)操作，把对数据库的修改全部写到磁盘上，并通知系统，事务已成功地结束
- 为事务打上一个提交标志(**committed**)，事务就进入‘提交’状态

□ 不论是‘提交’状态，还是‘异常中止’状态，都意味着一个事务的执行结束

5.1.4 有关事务的语句

□ ‘事务’ 除了由一组对于数据库的访问操作构成以外，通常还应该包括少量的事务控制语句，用于定义一个事务的开始和结束(包括正常结束和非正常结束)。因此，与事务有关的控制语句主要有三条：

- 事务的开始 (**begin transaction**)
- 事务的结束
 - 正常结束
 - ❖ 提交事务 (**commit transaction**)
 - 非正常结束
 - ❖ 回退事务 (**rollback transaction**)

5.1.4 有关事务的语句

□ Begin transaction

- 现有的关系数据库管理系统并没有提供一个用于定义从什么时候开始一个事务的控制语句，事务的启动是隐式的
- 可以通过三种方式来启动一个新的事务：
 - ① 数据定义命令(**DDL**)
 - ② 将系统设为自动提交方式（打开自动提交标志）
 - ③ 数据操纵命令(**DML**)

□ 事务的启动方式

① 数据定义命令(**DDL**)

- 每一条数据定义命令都将被作为一个单独的事务来执行
- 在此之前的用户事务将被自动提交

② 将系统设为自动提交方式（打开自动提交标志）

- 每一条数据库访问命令都将被作为一个单独的事务来执行，并根据执行结果自动提交或回退

③ 数据操纵命令(**DML**)

- 在当前用户的前一个事务执行结束之后，在用户提交的下一条数据库访问操作之前，数据库管理系统将自动启动一个新的事务

5.1.4 有关事务的语句

□ Commit transaction

- 提交当前事务，事务在执行过程中对于数据库的所有修改操作都将永久地反应到数据库中，并且不可被取消
- 事务的提交操作也可能失败，其原因包括：
 - 发生系统故障
 - 在提交阶段执行的数据完整性检查
- 在事务提交失败后，用户可以通过回退(**Rollback**)操作来取消当前事务
 - 由系统自动提交的事务，如果提交失败，系统将自动执行事务的回退操作

5.1.4 有关事务的语句

□ Rollback transaction

- 取消在该事务执行过程中的所有操作，回滚该事务至事务的起点，以便重新执行或放弃（**abort**）该事务
- 检查点(**savepoint**)
 - 在事务的执行过程中，系统可以为该事务设置若干个检查点
 - 用户事务可以使用 Rollback 命令将当前事务回退到前面的某个检查点sp，放弃“在检查点sp之后，回退操作之前”执行的对数据库的所有访问操作，并继续执行当前事务
 - 不带检查点的回退操作将结束并放弃整个事务

5.1.4 有关事务的语句

□ 除了上述三条事务控制语句外，数据库管理系统通常还会提供下述几条与事务有关的控制命令(**DCL**)

- ① 设置事务的自动提交命令

SET AUTO COMMIT ON|OFF

- ② 设置事务的类型

SET TRANSACTION READONLY|READWRITE

- ③ 设置事务的隔离级别

SET TRANSACTION ISOLATION LEVEL

READUNCOMMITTED|READCOMMITTED|

READREPEATABLE|SERIALIZABLE

5.1.4 有关事务的语句

□ **SET TRANSACTION READONLY | READWRITE**

➤ 定义在这之后启动的所有事务在执行过程中对数据库的访问方式

- **READONLY:** 只读型事务

- ❖ 在事务的运行过程中只能执行对数据库的‘读’操作，而不能执行‘更新’类型的操作
- ❖ 直到定义新的事务类型

- **READWRITE:** 读/写型事务

- ❖ 在事务的运行过程中可以执行对数据库的‘读/写’操作
- ❖ 这是事务的缺省类型定义

5.1.4 有关事务的语句

SET TRANSACTION ISOLATION LEVEL

READUNCOMMITTED

| **READCOMMITTED**

| **READREPEATABLE**

| **SERIALIZABLE**

- 定义当前用户的事务与其它并发执行事务之间的隔离级别
- 事务的隔离级别与所采用的封锁策略紧密相关。选择不同的隔离级别，系统所采用的封锁策略则不同
- 有四种不同的隔离级别可供选择

5.1.4 有关事务的语句

1) **READUNCOMMITTED**: 未提交读

- 在该方式下，当前事务不需要申请任何类型的封锁，因而可能会‘读’到未提交的修改结果
- 禁止一个事务以该方式去执行对数据的‘写’操作，以避免与其它并发事务的‘写’冲突现象

2) **READCOMMITTED**: 提交读

- 在‘读’数据对象**A**之前需要先申请对数据对象**A**的‘共享性’封锁，在‘读’操作执行结束之后立即释放该封锁
- 以避免读取到其它并发事务未提交的修改结果

5.1.4 有关事务的语句

3) **READREPEATABLE**: 可重复读

- 在‘读’数据对象**A**之前需要先申请对数据对象**A**的‘共享性’封锁，并将该封锁维持到当前事务的结束
- 可以避免其它的并发事务对当前事务正在使用的数据对象的修改

4) **SERIALIZABLE**: 可序列化(可串行化)

- 并发事务以一种可串行化的调度策略实现其并发执行，以避免它们相互之间的干扰现象

5.1.4 有关事务的语句

- 不管采用何种隔离级别，在事务‘写’数据对象A之前需要先申请对数据对象A的‘排它性’封锁，并将该封锁维持到当前事务的结束。(事务的隔离级别与封锁策略之间的关系)

5.1.5 事务的组成

□ [例]转帐事务($A=10000$, $B=20000$, 转帐金额 5000)

```
READ(A, t);
t := t - 5000;
WRITE(A, t);
READ(B, t);
t := t + 5000;
WRITE(B, t);
COMMIT T1;
```



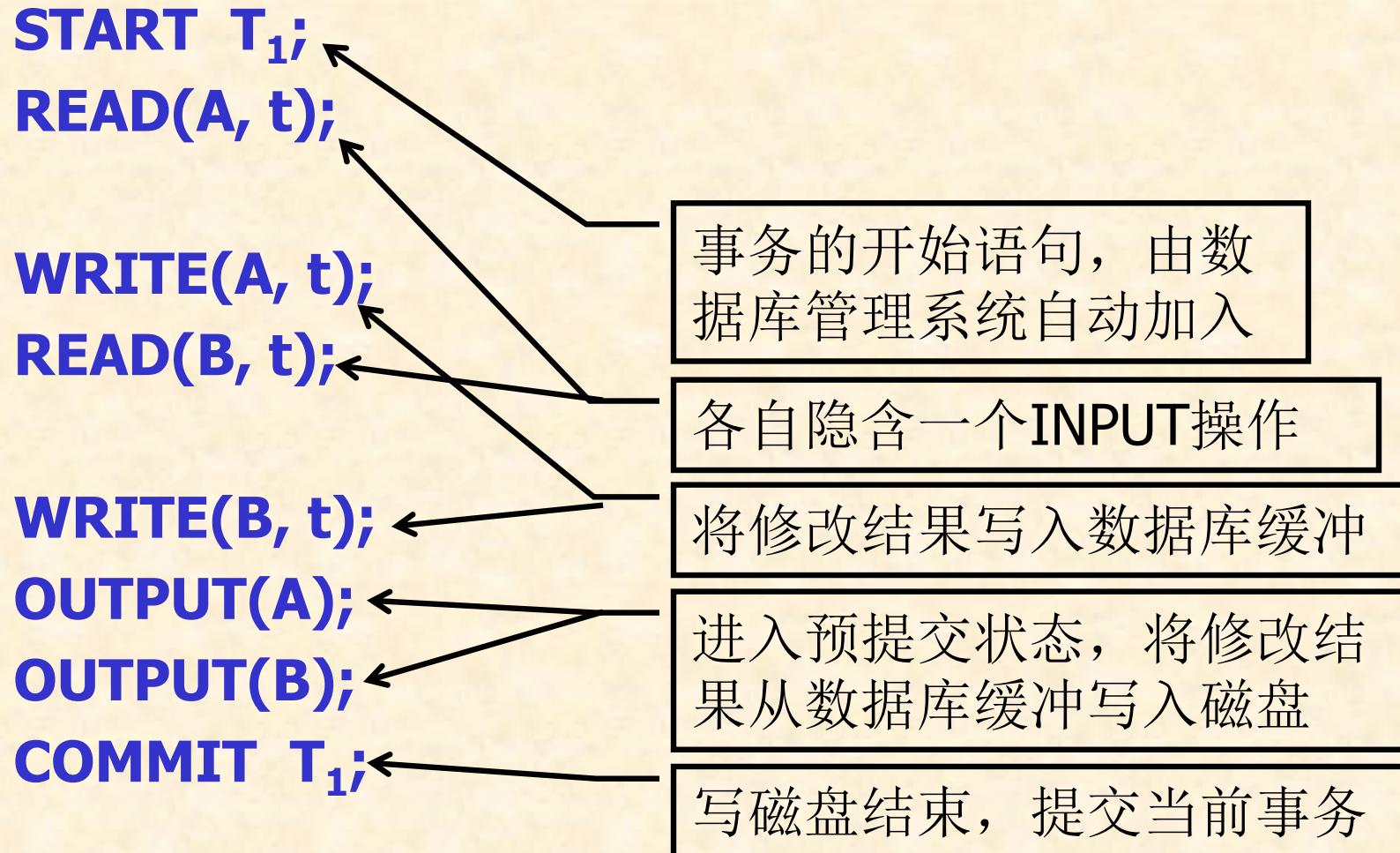
用户事务

```
START T1;
READ(A, t);
t := t - 5000;
WRITE(A, t);
READ(B, t);
t := t + 5000;
WRITE(B, t);
OUTPUT(A);
OUTPUT(B);
COMMIT T1;
```

数据库日志

5.1.5 事务的组成

□ [例]转帐事务($A=10000$, $B=20000$, 转帐金额 5000)



□ 事务

- 原子性、一致性、隔离性、持久性

□ 三种不一致现象

- 丢失修改、脏读、不可重复读

□ 事务的组成

- **Read (Input) 、 Write、 Output**

第五章 事务处理、并发控制与故障恢复技术

5.2 并发控制技术

5.2 并发控制技术

5.2.1 事务的并发执行

5.2.2 封锁

5.2.3 封锁协议

5.2.4 两阶段封锁协议

5.2.5 封锁粒度

5.2.6 活锁与死锁

5.2.1 事务的并发执行

□ 并发性

- 数据库是一个多用户共享系统
- 每个用户（或用户程序）都是以‘事务’为单位访问数据库
- 用于实现多个用户事务的并发执行的技术被称为**并发控制**(Concurrent Control)技术

5.2.1 事务的并发执行

□ 多个事务的执行方式

➤ 串行执行

- 以事务为单位，多个事务依次顺序执行
- 前一个事务对数据库的访问操作执行结束后，再去处理下一个事务对数据库的访问操作

➤ 并发执行

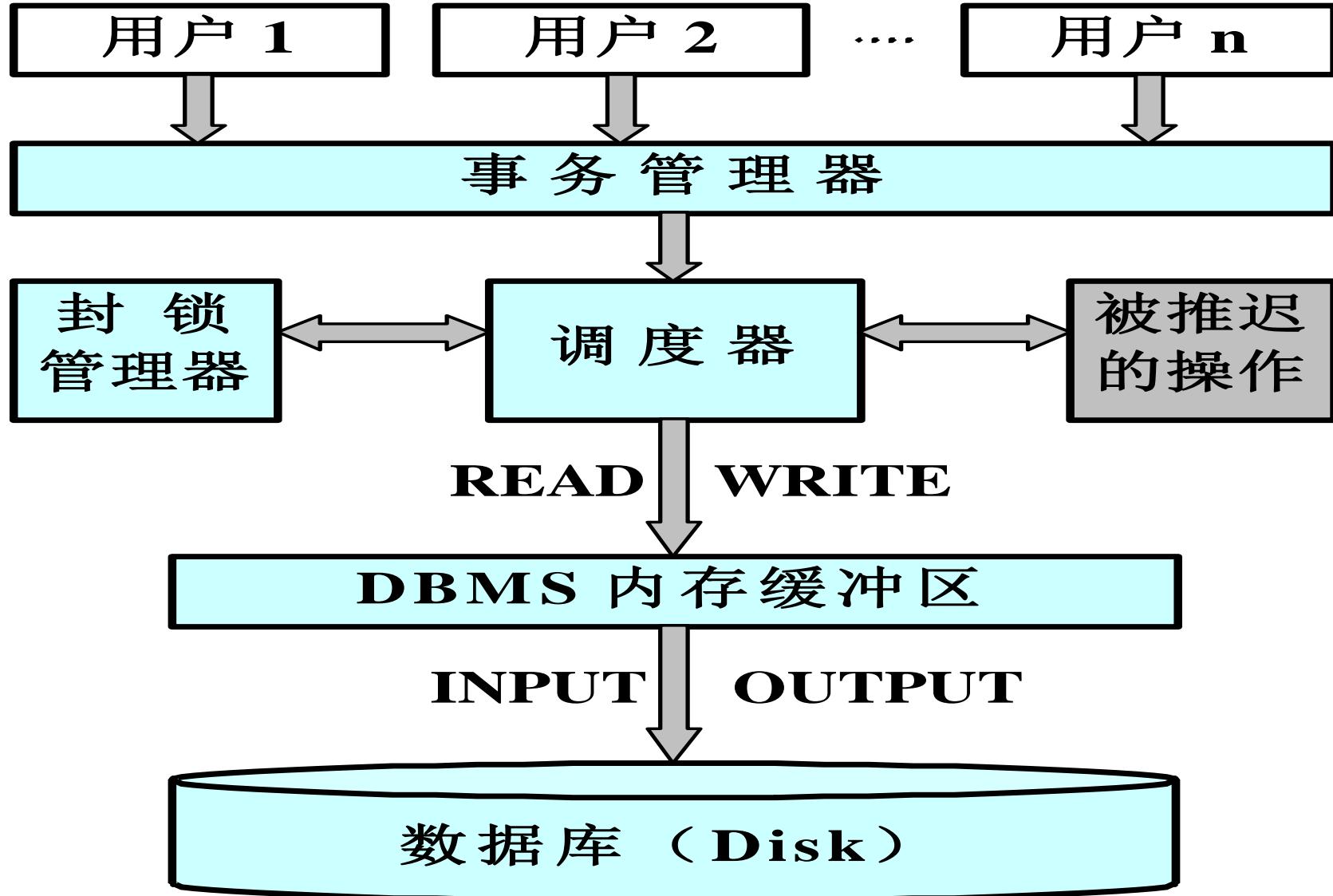
- 按一定调度策略交替执行

➤ 并发执行的可串行化(Serializability)

- 如果一组事务并发执行的结果等价于它们之间的某种串行执行的结果，则称其为可串行化调度

□ 并发控制的目标：实现并发事务的可串行化调度

5.2.1 事务的并发执行



用户事务并发执行示意图

5.2.1 事务的并发执行

□ 几个概念

- 调度 (**schedule**)
- 串行调度 (**serial schedule**)
- 可串行化调度 (**Serializable schedule**)
- 事务及其调度的表示方法
- 冲突可串行化
 - 冲突 (**Conflict**)
 - 优先图
 - 冲突可串行性判断



5.2.1 事务的并发执行

□ 调度

- 一个或多个事务中的数据库访问操作，按照这些操作被执行的时间排序所形成的一个操作序列

5.2.1 事务的并发执行

□ 数据库访问操作

- 用户事务对于数据库的访问操作包括**READ**和**WRITE**，这是事务对数据库缓冲区中的数据的‘读’和‘写’操作。(在这里我们忽略了对于磁盘的**INPUT**和**OUTPUT**操作)
- 驻留在内存缓冲区中的同一个数据对象**A**，有可能同时被若干个事务‘读’或‘写’。我们在考虑一个‘调度’时，最关心的是这一组事务以一种什么样的顺序来‘读/写’内存缓冲区中的数据
- 同一个事务的一组‘读/写’操作在‘调度’中的执行顺序是固定不变的，但是并发事务的不同交叉执行方式就构成了不同的‘调度’

5.2.1 事务的并发执行

□ 串行调度

- 如果一个调度的操作组成方式如下：首先是一个事务的所有操作，然后是另一个事务的所有操作，依此类推，则我们称该调度是串行的，或称为‘串行调度’
- 每一个事务中的所有操作，都是按照其预定的顺序依次执行的

5.2.1 事务的并发执行

□ 串行调度

- 任何一个串行调度均具有以下性质
 - 1) 在该调度中任意取两个事务X和Y，如果X的某个操作在Y的某个操作之前，则X的所有操作都在Y的所有操作之前
 - ❖ 不存在不同事务之间的交叉执行现象
 - 2) 并发事务的串行调度执行方式不会破坏数据库状态的一致性

5.2.1 事务的并发执行

□ 例1：银行转账

- 事务T₁：从账号A转 10000 至账号B
- 事务T₂：从账号A转 10% 的款项至账号B

□ 设帐号A与帐号B的初值都是20000，数据库状态的一致性要求是： $A + B = 40000$

□ 事务T₁与事务T₂的串行调度如下：

- 调度1：T₁, T₂ (图2 (图2 
- 调度2：T₂, T₁ (图3 (图3 

□ 在调度1或调度2执行结束后，数据库的状态是不同的，但都满足状态的一致性要求

调度1 (初值: A=B=20000)



	T_1	T_2	Temp	A	B
1	Read(A)			20000	20000
2	$A := A - 10000$				
3	Write(A)			10000	
4	Read(B)				
5	$B := B + 10000$				
6	Write(B)				30000
7		Read(A)			
8		Temp := A * 0.1	1000		
9		$A := A - Temp$			
10		Write(A)		9000	
11		Read(B)			
12		$B := B + Temp$			
13		Write(B)			31000

图2 串行执行之一 (结果: A=9000, B=31000)

调度2 (初值: A=B=20000)



	T ₁	T ₂	Temp	A	B
1		Read(A)		20000	20000
2		Temp:=A*0.1	2000		
3		A:=A-Temp			
4		Write(A)		18000	
5		Read(B)			
6		B:=B+Temp			
7		Write(B)			22000
8	Read(A)				
9	A:=A-10000				
10	Write(A)			8000	
11	Read(B)				
12	B:=B+10000				
13	Write(B)				32000

图3 串行执行之二 (结果: A=8000, B=32000)

5.2.1 事务的并发执行

□ 可串行化调度

- 每个串行调度都将保持数据库状态的一致性。但也存在其它的调度（非串行调度）能够保证数据库状态的一致性
- 如果一个调度对数据库状态的影响和某个串行调度相同，则我们称该调度是可串行化的，或称为‘可串行化调度’

5.2.1 事务的并发执行

□ 并发调度的例子：

- 调度3（图4 ）：并发事务的可串行化调度
- 调度4（图5 ）：不正确的并发调度

□ 从最后的执行结果上来看

- a) 调度3等价于串行调度1，因而是一个可串行化调度
- b) 调度4与调度1和调度2都不等价，因而是一个不正确的并发调度

调度3 (初值: A=B=20000)



	T ₁	T ₂	x	y	Temp	A	B
1	Read(A, x)		20000	-		20000	20000
2	x:=x-10000		10000				
3	Write(A, x)					10000	
4		Read(A, y)		10000			
5		Temp:=y*0.1			1000		
6		y:=y-Temp		9000			
7		Write(A, y)				9000	
8	Read(B, x)		20000				
9	x:=x+10000		30000				
10	Write(B, x)						30000
11		Read(B, y)		30000			
12		y:=y+Temp		31000			
13		Write(B, y)					31000

图4 并发执行可串行化 (结果: A=9000, B=31000)

调度4 (初值: A=B=20000)



	T_1	T_2	x	y	Temp	A	B
1	Read(A, x)		20000	-		20000	20000
2	$x := x - 10000$		10000				
3		Read(A, y)		20000			
4		Temp := y * 0.1			2000		
5		$y := y - Temp$		18000			
6		Write(A, y)				18000	
7		Read(B, y)		20000			
8	Write(A, x)					10000	
9	Read(B, x)		20000				
10	$x := x + 10000$		30000				
11	Write(B, x)						30000
12		$y := y + Temp$		22000			
13		Write(B, y)					22000

图5 不正确的并发执行 (结果: A=10000, B=22000)

5.2.1 事务的并发执行

□ 事务及调度的表示方法

- 在事务的执行过程中，我们只关心它对数据库中的哪些数据对象进行了读/写操作，并不在意每次读/写的值是多少，也不关心在用户事务中对这些数据对象的值做了怎样的处理
- 另外，如果两个事务对同一个数据对象进行‘写’操作，则我们假设它们‘写’后的值总是不相等的
- 事务及调度的记法
 - 事务用符号 T_1, T_2, \dots 表示
 - 用 $r_i(X)$ 表示事务 T_i 读数据库对象 X
 - 用 $w_i(X)$ 表示事务 T_i 写数据库对象 X

5.2.1 事务的并发执行

□ 事务及调度的表示方法 (续)

- 例1中的两个事务可以分别表示为：
 - 事务 T_1 : $r_1(A); w_1(A); r_1(B); w_1(B);$
 - 事务 T_2 : $r_2(A); w_2(A); r_2(B); w_2(B);$
 - 调度1(图2)可以表示为：
 $r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B);$
 - 该串行调度可以简化表示为： T_1, T_2
-
- 调度2(图3)可以表示为：
 $r_2(A); w_2(A); r_2(B); w_2(B); r_1(A); w_1(A); r_1(B); w_1(B);$
 - 该串行调度可以简化表示为： T_2, T_1

5.2.1 事务的并发执行

□ 事务及调度的表示方法（续）

- 事务 T_1 : $r_1(A);w_1(A);r_1(B);w_1(B);$
- 事务 T_2 : $r_2(A);w_2(A);r_2(B);w_2(B);$

➤ 调度3(图4)可以表示为:

$r_1(A);w_1(A);r_2(A);w_2(A);r_1(B);w_1(B);r_2(B);w_2(B);$

➤ 调度4(图5)可以表示为:

$r_1(A);r_2(A);w_2(A);r_2(B);w_1(A);r_1(B);w_1(B);w_2(B);$

	T ₁	T ₂	x	y	Temp	A	B
1	Read(A, x)		20000	-		20000	20000
2	x:=x-10000		10000				
3	Write(A, x)					10000	
4		Read(A, y)		10000			
5		Temp:=y*0.1			1000		
6		y:=y-Temp		9000			
7		Write(A, y)				9000	
8	Read(B, x)		20000				
9	x:=x+10000		30000				
10	Write(B, x)						30000
11		Read(B, y)		30000			
12		y:=y+Temp		31000			
13		Write(B, y)					31000

➤ 调度3(图4)可以表示为:

r₁(A);w₁(A);r₂(A);w₂(A);r₁(B);w₁(B);r₂(B);w₂(B);

	T ₁	T ₂	x	y	Temp	A	B
1	Read(A, x)		20000	-		20000	20000
2	x := x - 10000		10000				
3		Read(A, y)		20000			
4		Temp := y * 0.1			2000		
5		y := y - Temp		18000			
6		Write(A, y)				18000	
7		Read(B, y)		20000			
8	Write(A, x)					10000	
9	Read(B, x)		20000				
10	x := x + 10000		30000				
11	Write(B, x)						30000
12		y := y + Temp		22000			
13		Write(B, y)					22000

➤ 调度4(图5)可以表示为:

r₁(A); r₂(A); w₂(A); r₂(B); w₁(A); r₁(B); w₁(B); w₂(B);

5.2.1 事务的并发执行

□ 冲突可串行化

➤ 冲突

- 是指调度中的一对连续操作(**op1; op2**)，它们满足如下的条件：
 - ❖ 如果交换它们两者的执行顺序，那么涉及的事务中至少有一个的行为会改变
- 符合上述条件的一对连续的操作(**op1; op2**)被称为‘冲突’

□ 冲突的判断

- 假设 T_i 和 T_j 是两个不同的事务(即 $i \neq j$)，则：
 - $r_i(X);r_j(Y)$ 不是冲突(即使 $X=Y$ 也不会是冲突)
 - 如果 $X \neq Y$ ，则 $r_i(X);w_j(Y)$ 不会是冲突
 - 如果 $X \neq Y$ ，则 $w_i(X);r_j(Y)$ 不会是冲突
 - 如果 $X \neq Y$ ，则 $w_i(X);w_j(Y)$ 不会是冲突
- 同一事务的任意两个相邻的操作都是冲突，如：
 $(r_i(X);w_i(Y))$ 、 $(w_i(X);r_i(Y))$ 、 $(r_i(X);r_i(Y))$ 等
- 不同事务对同一数据对象的‘写’冲突：
 $w_i(X);w_j(X)$
- 不同事务对同一数据对象的‘读-写’冲突：
 $r_i(X);w_j(X) \quad w_i(X);r_j(X)$

5.2.1 事务的并发执行

□ 概括起来讲

- 同一事务中的任意两个操作不可以交换它们的执行顺序(是冲突)
- 不同事务中的任何两个操作在执行顺序上是可以交换的，除非：
 - 1) 它们涉及同一个数据对象； 并且
 - 2) 至少有一个是‘写’操作

□ 对于初始给定的一个调度，如果通过一组‘非冲突’操作的交换，能够将该调度转换为一个串行调度，则我们认为最初的调度就是一个可串行化调度

5.2.1 事务的并发执行

□ 冲突等价

- 如果通过一系列相邻操作的非冲突交换能够将一个调度转换为另一个调度，则我们称这两个调度是冲突等价的

□ 冲突可串行化

- 如果一个调度 **S** 冲突等价于一个串行调度，则我们称调度 **S** 是“冲突可串行化”的

□ ‘冲突可串行化’是‘可串行化’的一个充分条件，而非必要条件

- a) ‘冲突可串行化’调度必定也是一个‘可串行化’调度
- b) 一个‘可串行化’调度并不一定是‘冲突可串行化’的

5.2.1 事务的并发执行

口 例：将调度3通过一系列非冲突交换可以转换成一个串行调度（调度1）（事务 T_2 中的操作被标上下划线）

➤ 步骤1 (原调度3)

$r_1(A); w_1(A); \underline{r_2(A)}; \underline{w_2(A)}; r_1(B); w_1(B); \underline{r_2(B)}; \underline{w_2(B)}$;

➤ 步骤2：

$r_1(A); w_1(A); \underline{r_2(A)}; r_1(B); \underline{w_2(A)}; w_1(B); \underline{r_2(B)}; \underline{w_2(B)}$;

➤ 步骤3：

$r_1(A); w_1(A); r_1(B); \underline{r_2(A)}; \underline{w_2(A)}; w_1(B); \underline{r_2(B)}; \underline{w_2(B)}$;

➤ 步骤4：

$r_1(A); w_1(A); r_1(B); \underline{r_2(A)}; w_1(B); \underline{w_2(A)}; \underline{r_2(B)}; \underline{w_2(B)}$;

➤ 步骤5 (串行调度1)

$\underline{r_1(A)}; w_1(A); r_1(B); w_1(B); \underline{r_2(A)}; \underline{w_2(A)}; \underline{r_2(B)}; \underline{w_2(B)}$;

T_1

T_2

5.2.1 事务的并发执行

□ 优先图

- 思想：冲突操作反映了给定事务在冲突等价的串行调度（如果存在的话）中的执行顺序
- 定义：已知在调度 S 中存在两个事务 T_1 和 T_2 ，如果有 T_1 的一个动作 A_1 和 T_2 的一个动作 A_2 满足：
 - ❖ 在调度 S 中 A_1 在 A_2 前
 - ❖ A_1 和 A_2 涉及数据库中的同一数据对象，且至少有一个是‘写’动作
- 则我们称 T_1 优先于 T_2 ，记为 $T_1 <_s T_2$
- 在上述情况下， A_1 和 A_2 是两个不能交换的动作，如果存在一个冲突等价于 S 的串行调度，则在该串行调度中 T_1 必在 T_2 之前

5.2.1 事务的并发执行

□ 优先图

【定义】以调度S中的事务作为优先图中的结点（事务 T_i 可简记为*i*），如果 $T_i <_s T_j$ ，则从结点*i*到结点*j*引一条有向边。依此类推构成的一个有向图被称为调度S的事务优先图。

□ 例: $r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

- 其优先图是: 

$1 \rightarrow 2 \rightarrow 3$

□ 例: $r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

- 其优先图是: $1 \rightarrow 2 \rightarrow 3$ 


5.2.1 事务的并发执行

□ 冲突可串行化的判断规则

- 构造调度 \mathbf{S} 的事务优先图，如果该图是无环的，则调度 \mathbf{S} 是冲突可串行化的。如果有环，则调度 \mathbf{S} 不是冲突可串行化的

□ 例: $r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

- 其优先图是: $1 \rightarrow 2 \rightarrow 3$ (无环)，因此该调度是冲突可串行化的

□ 例: $r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

- 其优先图是: $1 \rightarrow 2 \rightarrow 3$ (存在环状结构)，因此该

调度不是冲突可串行化的

[结论1]如果在优先图中存在环，则该调度不是冲突可串行化的。

证明（反证法）：设有一个由 T_1, T_2, \dots, T_n 等n个事务构成的调度H，其优先图中存在一个涉及这n个事务的环状结构： $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$ 。假设存在一个与调度H冲突等价的串行调度S

- ∴ 在优先图中 T_1 优先于 T_2 的前提条件是存在 T_1 的一个动作 A_1 和 T_2 的一个动作 A_2 ，在调度H中 A_1 先于 A_2 ，且 A_1 与 A_2 不可交换（冲突）
- ∴ 在调度S中， A_1 也应先于 A_2 。即在串行调度S中 T_1 应该先于 T_2 执行。依此类推： T_1 先于 T_2 ， T_2 先于 T_3 ， \dots ， T_{n-1} 先于 T_n ， T_n 先于 T_1
- ∴ 得到一对矛盾的关系： T_1 先于 T_n ， T_n 先于 T_1
- ∴ 假设不成立，不可能存在一个与调度H冲突等价的串行调度，即不是冲突可串行化的

[结论2]如果调度S的优先图中无环，则调度S是冲突可串行化的。

证明(归纳法)设由n个事务 T_1, T_2, \dots, T_n 构成一个调度 H_n

1. 当n=1时，调度 H_1 只有一个事务组成，因此 H_1 是一个串行调度，因而也是冲突可串行化的
2. 假设n=(k-1)时结论成立，即：如果存在一个由(k-1)个事务 T_1, T_2, \dots, T_{k-1} 所构成的调度 H_{k-1} ，其事务优先图是无环的，则调度 H_{k-1} 是冲突可串行化的，即调度 H_{k-1} 冲突等价于它们的某个串行调度 S_{k-1}
3. 当n=k时，如果由k个事务 T_1, T_2, \dots, T_k 所构成的调度 H_k 的事务优先图G是无环的，则有如下结论：

∴优先图**G**是一个有向无环图

∴在图**G**中至少存在一个结点 i (事务 T_i 所对应的结点), 不存在指向该结点的有向边

∴在调度 H_k 中不存在这样一个动作 A :

- a) 是 T_i 之外的某个事务 T_j 的动作
- b) 动作 A 位于 T_i 的某个动作 A_i 之前
- c) 动作 A 与动作 A_i 是冲突

∴可以通过非冲突动作的交换将事务 T_i 的所有动作交换到调度的最前面, 并保持它们的原有顺序不变。即将调度 H_k 转换成下述的调度 S_k :

(T_i 的所有动作) (其它 $k-1$ 个事务的动作)
且调度 H_k 与调度 S_k 是冲突等价的

考虑调度 S_k 的后半部分：

结论2的证明(续)

(T_i 的所有动作) (其它 $k-1$ 个事务的动作)

H_{k-1}

由 T_i 之外的其它($k-1$)个事务中的动作所构成的调度 H_{k-1}

- ∴ 调度 H_{k-1} 中的所有动作均保持了它们在调度 H_k 中的排列顺序
- ∴ 从优先图 G 中去删除结点 i 以及与其相关的所有有向边，就构成了调度 H_{k-1} 的事务优先图 G' (即：图 G' 是图 G 的一个子图)
- ∴ 图 G 是有向无环图
- ∴ 图 G' 也是一个有向无环图

根据步骤2的假设：如果由 $(k-1)$ 个事务所构成的调度的事务优先图是无环的，则该调度是冲突可串行化的

- \because 上述的调度 H_{k-1} 是冲突可串行化的，即调度 H_{k-1} 冲突等价于这 $(k-1)$ 个事务的某个串行调度 S_{k-1}
- \because 调度 S_k 等于：(事务 T_i 的所有动作) + 调度 H_{k-1}
调度 H_{k-1} 冲突等价于某个串行调度 S_{k-1}
- \therefore 调度 S_k 冲突等价于一个串行调度 (T_i, S_{k-1})
- \because 调度 H_k 冲突等价于调度 S_k
调度 S_k 冲突等价于串行调度 (T_i, S_{k-1})
- \therefore 调度 H_k 冲突等价于串行调度 (T_i, S_{k-1})
- \therefore 调度 H_k 是冲突可串行化的。 证毕

5.2.1 事务的并发执行

□ 三种数据不一致现象

- 在事务的并发执行过程中，如果采用了不合理的调度方法，就会出现并发执行错误，破坏数据库中数据的一致性。
- 常见的并发执行错误有三种：
 1. 丢失修改 (**lost update**)
 2. 脏读 (**dirty read**)
 3. 不可重复读 (**non-repeatable read**)

5.2.1 事务的并发执行

□ 例：飞机售票业务

- 假设某个航班的剩余机票的张数保存在数据库对象X（初始值为5）中，则出售一张该航班机票的事务的执行流程是：

Read(X, t); /* t是该事务使用的内存变量 */

t := t - 1;

Write(X, t);

□丢失修改

步骤	甲售票点	变量 t1	乙售票点	变量 t2	剩余 机票X
1	Read(X, t1);	5			5
2	t1:=t1 - 1;	4			
3			Read(X, t2);	5	
4			t1:=t2 - 1;	4	
5	Write(X, t1);				4
6			Write(X, t2);		4

➤ 现象：

一个事务的修改结果破坏了另一个事务的修改结果

➤ 原因：

对多个事务并发修改同一个数据对象的情况未加控制

□ 脏读

步骤	甲售票点	变量 t1	乙售票点	变量 t2	剩余 机票X
1	Read(X, t1) ;	5			5
2	t1:=t1 - 1;	4			
3	Write(X, t1) ;	4			4
4			Read(X, t2) ;	4	
5	放弃当前的售票操作， 取消对X的修改				5

➤ 现象：

读到了错误的数据(即与数据库中的情况不相符的数据)

➤ 原因：

一个事务读取了另一个事务未提交的修改结果

□不可重复读

步骤	甲售票点	变量 t1	乙售票点	变量 t2	剩余 机票X
1	Read(X, t1);	5			5
2			Read(X, t2);	5	
3			t2:=t2 - 1;	4	
4			Write(X, t2);	4	4
5	Read(X, t1);	4			

➤ 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



□ Example 10.3.3 Blind Writes

$T_1: W_1(A, 50); W_1(B, 50); C_1;$

$T_2: W_2(A, 80); W_2(B, 20); C_2;$

$H_3: W_1(A, 50); W_2(A, 80); W_2(B, 20); W_1(B, 50); C_1; C_2;$

schedule	H_3	$T_1 \& T_2$	$T_2 \& T_1$
value of A	80	80	50
value of B	50	20	50

5.2.1 事务的并发执行

- 出现上述三种数据不一致现象的原因在于：
 - 多个并发执行的事务反复交叉使用同一个数据库（数据对象），而数据库管理系统又没有提供必要的控制手段
- 合理组织调度多个用户的并发操作，避免产生数据不一致现象的工作也被称为‘并发控制’
- 常用的并发控制技术是：**封锁**

5.2 并发控制技术

5.2.1 事务的并发执行

5.2.2 封锁

5.2.3 封锁协议

5.2.4 两阶段封锁协议

5.2.5 封锁粒度

5.2.6 活锁与死锁

5.2.2 封锁

□ 主要内容

- 封锁 (**lock**)
- 封锁类型
 - 常用的两种类型封锁
 - 锁相容矩阵
 - 申请封锁和释放封锁的处理过程
- 使用封锁技术实现并发控制的方法

5.2.2 封锁

□ 封锁 (lock)

➤ 使用封锁技术的前提

- 在一个事务访问数据库中的数据时，必须先获得被访问的数据对象上的封锁，以保证数据访问操作的正确性和一致性。

➤ 封锁的作用

- 在一段时间内禁止其它事务在被封锁的数据对象上执行某些类型的操作 (由封锁的类型决定)
 - 同时也表明：持有该封锁的事务在被封锁的数据对象上将要执行什么类型的操作 (由系统所采用的封锁协议来决定)
- ‘封锁’是多用户环境中最常采用的一种并发控制技术

5.2.2 封锁

□ 封锁类型

- 常用的封锁类型有两种
 - 排它锁
 - ❖ **eXclusive lock**, 又简称为: **X锁**
 - 共享锁
 - ❖ **Sharing lock**, 又简称为: **S锁**

□ 排它锁（X锁）

➤ 特性

- 只有当数据对象A没有被其它事务封锁时，事务T才能在数据对象A上施加‘X锁’
- 如果事务T对数据对象A施加了‘X锁’，则其它任何事务都不能在数据对象A上再施加任何类型的封锁

➤ 作用

- 如果一个事务T申请在数据对象A上施加‘X锁’并得到满足，则：事务T自身可以对数据对象A作读、写操作，而其它事务则被禁止访问数据对象A
 - 这样可以让事务T独占该数据对象A，从而保证了事务T对数据对象A的访问操作的正确性和一致性
 - 缺点：降低了整个系统的并行性
- ‘X锁’必须维持到事务T的执行结束

□ 共享锁（S锁）

➤ 特性

- 如果数据对象A没有被其它事务封锁，或者其它事务仅仅以‘S锁’的方式来封锁数据对象A时，事务T才能在数据对象A上施加‘S锁’

➤ 作用

- 如果一个事务T申请在数据对象A上施加‘S锁’并得到满足，则：事务T可以对‘读’数据对象A，但不能‘写’数据对象A
 - 不同事务所申请的‘S锁’可以共存于同一个数据对象A上，从而保证了多个事务可以同时‘读’数据对象A，有利于提高整个系统的并发性
 - 在持有封锁的事务释放数据对象A上的所有‘S锁’之前，任何事务都不能‘写’数据对象A
- ‘S锁’不必维持到事务T的执行结束(依封锁协议而定)

- ‘排它锁’与‘共享锁’的相互关系可以用如下图所示的‘锁相容矩阵’来表示

		其它事务已持有的锁		
		X锁	S锁	-
当前事务 申请的锁	X锁	No	No	Yes
	S锁	No	Yes	Yes

图5.8 封锁的相容矩阵

- 合适(well formed)事务

- 如果一个事务在访问数据库中的数据对象A之前按照要求申请对A的封锁，在操作结束后释放A上的封锁，这种事务被称为合适事务

- ‘合适事务’是保证并发事务的正确执行的基本条件

5.2.2 封锁

□ 封锁的申请与释放

➤ 封锁管理器的数据结构

- 数组**LOCK(A)**

- ❖ 记录被施加在数据对象**A**上的封锁类型，其值是：

- Read_locked** (共享锁)

- Write_locked** (排它锁)

- Unlocked** (无封锁)

- 数组**no_of_reads(A)**

- ❖ 记录被施加在数据对象**A**上的共享锁的个数

申请对数据对象A的共享锁(S锁): **read_lock(A)**

```
B: if LOCK(A) = 'Unlocked' {
    LOCK(A) := 'Read_locked';
    no_of_reads(A) := 1;
} else {
    if LOCK(A) = 'Read_locked'
        no_of_reads(A) := no_of_reads(A) + 1;
    else {
        wait ( until LOCK(A) != 'Write_locked'
            and the lock manager wakes up
            the transaction);
        go to B;
    }
}
```

申请对数据对象A的排它锁(X锁): **write_lock(A)**

B:

```
if LOCK(A) = 'Unlocked'  
{  
    LOCK(A) := 'Write_locked';  
}  
else  
{  
    wait ( until LOCK(A) = 'Unlocked' and the lock  
          manager wakes up the transaction);  
    go to B;  
}
```

释放对数据对象A的封锁: **unlock(A)**

```
if LOCK(A) = 'Write_locked' {  
    LOCK(A) := 'Unlocked';  
    wake up one of the waiting transaction, if any  
}  
  
else if LOCK(A) = 'Read_locked' {  
    no_of_reads(A) := no_of_reads(A) - 1;  
    if no_of_reads(A) = 0 {  
        LOCK(A) := 'Unlocked'  
        wake up one of the waiting transaction, if any  
    }  
}
```

5.2.2 封锁

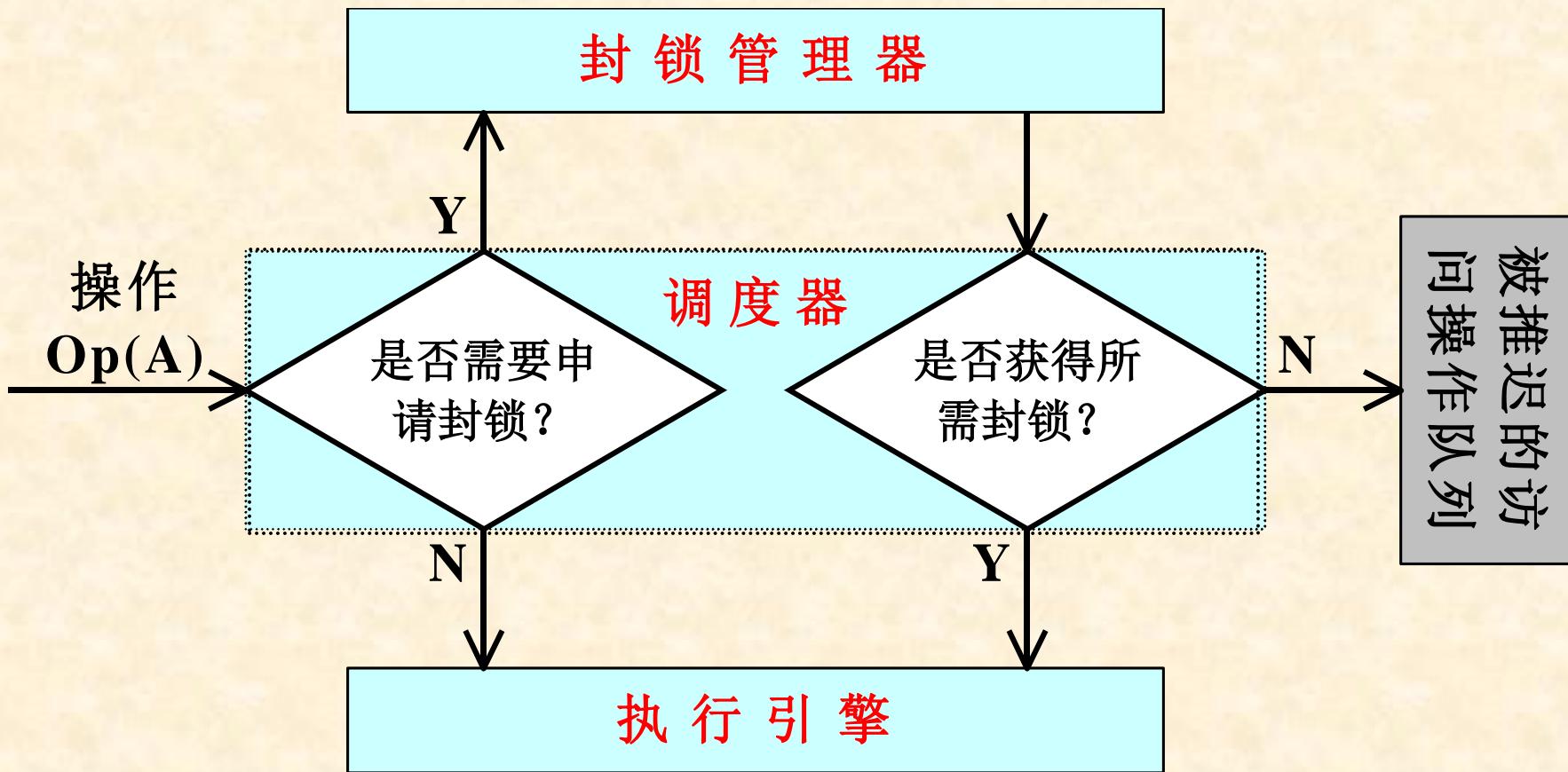
□ 使用封锁的并发控制技术

- 在DBMS的‘封锁管理器’中维护着一张‘锁表’，以记录当前：
 - 封锁的持有情况
 - ❖ 有哪些‘事务’在哪些‘数据对象’上持有什么类型的‘封锁’
 - 封锁的申请情况
 - ❖ 有哪些‘事务’正在申请哪些‘数据对象’上的什么类型的‘封锁’
- DBMS对于用户事务的数据访问操作Op(A)的处理过程如下：

事务的数据访问操作的处理过程

1. 将访问操作Op(A)发送给并发控制子系统的‘调度器’
2. ‘调度器’根据系统采用的封锁协议来决定：
 - 是否需要为该操作申请封锁
 - 申请何种类型的封锁并将封锁申请发送给‘封锁管理器’
3. ‘封锁管理器’根据锁表中记载的情况来决定是否能够立即满足该封锁申请，并将申请结果返回给‘调度器’
4. 如果封锁申请得不到满足，则‘调度器’将访问操作Op(A)放入‘被推迟的访问操作’队列，否则将该操作发送给系统的执行引擎执行

事务的数据访问操作的处理过程（图）



5.2 并发控制技术

5.2.1 事务的并发执行

5.2.2 封锁

5.2.3 封锁协议

5.2.4 两阶段封锁协议

5.2.5 封锁粒度

5.2.6 活锁与死锁

5.2.3 封锁协议

- 采用‘封锁’技术为并发事务的正确执行提供了可能性。但是要真正确保事务并发执行的正确性，还必须按照规定的方法来使用‘封锁’技术，即规定事务使用‘封锁’的方式。包括：
 - 何时申请封锁？
 - 申请何种类型的封锁？(**S锁/X锁**)
 - 何时释放所持有的封锁？

5.2.3 封锁协议

□ 封锁协议 (**locking protocol**)

➤ 不同的封锁方式构成了不同的封锁规则，我们称之为‘封锁协议’

- 三级封锁协议

- ❖ 不同级别的封锁协议可以防止不同的并发错误

- 两阶段封锁协议

□ 一级封锁协议

- 事务T在‘写’数据对象A之前，必须先申请并获得A上的‘X锁’，并维持到事务T的执行结束(包括Commit与Rollback)才释放被加在A上的‘X锁’

□ 二级封锁协议

- ‘一级封锁协议’的要求；并且
- 事务T在‘读’数据对象A之前，必须先申请并获得A上的‘S锁’，在‘读’操作完成后即可释放A上的‘S锁’
 - 这里没有规定释放‘S锁’的时间

□ 三级封锁协议

- ‘一级封锁协议’的要求；并且
- 事务T在‘读’数据对象A之前，必须先申请并获得A上的‘S锁’，并维持到事务T的执行结束(包括Commit与Rollback)才释放被加在A上的‘S锁’

‘一级封锁协议’可防止‘丢失修改’现象



步骤	甲售票点	乙售票点	剩余机票X
1	write_lock(X);		5
2	Read(X,t1);		
3		write_lock(X);	
4	t1:=t1 - 1;	wait	
5	Write(X,t1);	wait	4
6	Commit;	wait	
7	unlock(X);	wait	
8		Read(X,t2);	
9		t1:=t2 - 1;	
10		Write(X,t2);	3
11		Commit;	
12		unlock(X);	

‘二级封锁协议’可防止‘丢失修改、脏读’现象

步骤	甲售票点	变量 t1	乙售票点	变量 t2	剩余 机票X
1	write_lock(X) ;				5
2	Read(X, t1) ;	5			
3	t1:=t1 - 1;	4			
4	Write(X, t1) ;				4
5			read_lock(X) ;		
6			wait		
7	Rollback		wait		5
8	unlock(X) ;		wait		
9			Read(X, t1) ;	5	
				

‘三级封锁协议’可防止‘丢失修改、脏读、不可重复读’现象

步骤	甲售票点	变量 t1	乙售票点	变量 t2	剩余 机票X
1	<code>read_lock(X);</code>				5
2	<code>Read(X,t1);</code>	5			
3			<code>write_lock(X);</code>		
4		<code>wait</code>		
5	<code>Read(X,t1);</code>	5	<code>wait</code>		
6	<code>Commit;</code>		<code>wait</code>		
7	<code>unlock(X);</code>		<code>wait</code>		
8			<code>Read(X,t2);</code>	5	
9				

口‘三级封锁协议’与三种数据不一致现象的关系

		可能出现的数据不一致现象		
		丢失修改	脏读	不可重复读
封锁协议	一级封锁协议	No	Yes	Yes
	二级封锁协议	No	No	Yes
	三级封锁协议	No	No	No

□ 根据对于封锁协议的介绍我们可以得知，多个事务的并发运行之所以会破坏数据库状态的一致性，其原因是：

- 事务没有为被访问的数据对象申请封锁
- 事务没有在合适的时候释放所持有的锁

□ ‘**三级封锁协议**’ 正是为解决上述问题而定义的封锁规则

□ 调度

- 串行调度
- 可串行化调度
- 冲突可串行化

□ 封锁

- 排它锁、共享锁

□ 三级封锁协议

5.2 并发控制技术

5.2.1 事务的并发执行

5.2.2 封锁

5.2.3 封锁协议

5.2.4 两阶段封锁协议

5.2.5 封锁粒度

5.2.6 活锁与死锁

5.2.4 两阶段封锁协议

□ 按照‘三级封锁协议’的规定，事务T在其执行过程中所申请的所有‘锁’必须在事务T结束后才能释放。这就意味着，在一个事务执行过程中，必须把锁的申请与释放分为两个阶段：

- 第一个阶段：申请并获得封锁
 - 在此阶段中，事务可以申请其整个执行过程中所需要的锁，此阶段也可称为‘扩展阶段’
- 第二阶段：释放所有申请获得的锁
 - 此阶段也可称为‘收缩阶段’
 - 事务一旦开始释放封锁，那么就不能再申请任何封锁

□ 此种设置封锁的方法称为‘两阶段封锁协议’

- **two-phase locking protocol**, 简称**2PL**协议

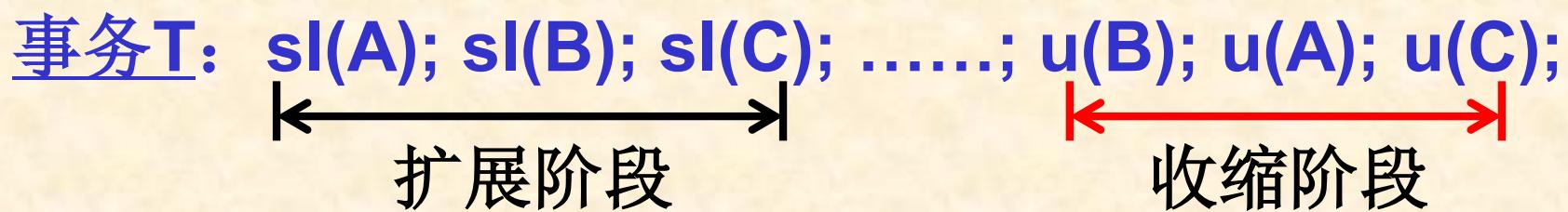
5.2.4 两阶段封锁协议

□ 两阶段封锁事务

- 在一个事务T中，如果所有的封锁请求都先于所有的解锁请求，则该事务被称为‘两阶段封锁事务’，简称‘2PL事务’
- 或者说：采用两阶段封锁协议的事务

5.2.4 两阶段封锁协议

- 用 **sl(A)** 表示申请对 A 的 ‘S锁’，用 **xl(A)** 表示申请对 A 的 ‘X锁’，用 **u(A)** 表示释放 A 上的封锁
- 2PL 事务的例子(图5.12)



- 不遵守2PL协议的例子(图5.13)

事务T': **sl(A); u(A); sl(B); xl(C);; u(C); u(B)**

5.2.4 两阶段封锁协议

口 基于前述的‘三级封锁协议’和‘两阶段封锁协议’的要求，我们归纳出有关封锁技术的使用规定

- 假设系统采用‘X锁’和‘S锁’两种类型锁，有关封锁的申请与释放操作表示如下：
 - **sl_i(A)**: 事务T_i申请数据对象A上的一个‘S锁’
 - **xl_i(A)**: 事务T_i申请数据对象A上的一个‘X锁’
 - ❖ 也可以用I_i(A)表示事务T_i申请数据对象A上的锁
 - **u_i(A)**: 事务T_i释放自己在数据对象A上所持有的锁

□ 在每一个事务 T_i 中

➤ 第1点：采用如下的封锁协议：

- 读动作 $r_i(A)$ 之前必须有 $sl_i(A)$ 或 $xl_i(A)$ ，而且在两者之间没有 $u_i(A)$
- 写动作 $w_i(A)$ 之前必须有 $xl_i(A)$ ，而且在两者之间没有 $u_i(A)$
- 每一个 $sl_i(A)$ 或 $xl_i(A)$ 之后必须有一个 $u_i(A)$

➤ 第2点：必须遵循‘两阶段封锁协议’

- 在任何 $sl_i(A)$ 或 $xl_i(A)$ 之前不能有 $u_i(B)$
 - ❖ A和B可以是同一个数据对象

- 如果事务 T_i 在执行过程中重复申请同一个数据对象 A 上的锁，‘封锁管理器’的处理方法如下：
 - 1) 如果 T_i 已经持有数据对象 A 上的锁，则对 $sl_i(A)$ 不作任何处理
 - 2) 如果 T_i 已经持有数据对象 A 上的 ‘S 锁’，在处理 $xl_i(A)$ 请求时，仅仅将 T_i 所持有的 ‘S 锁’ 改为 ‘X 锁’
 - 仅当 只有 事务 T_i 持有数据对象 A 上的 ‘S 锁’ 时，封锁管理器才能将 T_i 所持有的 ‘S 锁’ 改为 ‘X 锁’
 - 3) 如果 T_i 已经持有数据对象 A 上的 ‘X 锁’，则对 $xl_i(A)$ 不作任何处理

➤ 上述的处理过程确保

- 1) 一个事务在一个数据对象上只能持有一把‘锁’
- 2) 如果事务 T_i 在执行过程中在同一个数据对象 A 上执行了若干次锁申请操作：

$l_{i1}(A); l_{i2}(A); \dots; l_{in}(A)$

并且在 $l_{i1}(A)$ 和 $l_{in}(A)$ 之间没有 $u_i(A)$ ，则在 $l_{in}(A)$ 之后必须有且仅有一个 $u_i(A)$

- 1) 封锁的重复申请过程只能是：

$sl_i(A); \dots; xl_i(A);$

□ 第3点：保证事务调度的合法性

- 对任意两个不同的事务 T_i 和 T_j ，其调度必须满足：
 - 如果 $x_{l_i}(A)$ 出现在调度中，那么后面不能再有 $s_{l_j}(A)$ 或 $x_{l_j}(A)$ ，除非中间插入了 $u_i(A)$
 - 如果 $s_{l_i}(A)$ 出现在调度中，那么后面不能再有 $x_{l_j}(A)$ ，除非中间插入了 $u_i(A)$

- 如果一个调度以及它的每个事务都满足上述三个方面的要求，我们称该调度是一个‘合法调度’
- 我们可以将前述的‘调度器’，和‘封锁管理器’，合并为一个‘封锁调度器’，DBMS正是通过‘封锁调度器’来产生若干个并发运行事务之间的一个‘合法调度’的

封锁的使用规定（续）

	T_1	T_2	x	y	Temp	A	B
1	Read(A,x)		20000	-		20000	20000
2	$x:=x-10000$		10000				
3		Read(A,y)		20000			
4		Temp:=y*0.1			2000		
5		$y:=y-Temp$		18000			
6		Write(A,y)				18000	
7		Read(B,y)		20000			
8	Write(A,x)					10000	
9	Read(B,x)		20000				
10	$x:=x+10000$		30000				
11	Write(B,x)						30000
12		$y:=y+Temp$		22000			
13		Write(B,y)					22000

图5 不正确的并发执行 (结果: A=10000,B=22000)

□ 图5所示的两个转帐事务 T_1 和 T_2 的执行流程如下(只保留了对于数据库的访问操作):

$r_1(A); r_2(A); w_2(A); r_2(B); w_1(A); r_1(B); w_1(B); w_2(B);$

□ 该操作执行流程被发送给‘封锁调度器’时，‘封锁调度器’将首先根据前面介绍的‘封锁协议’和‘2PL事务’的要求插入相应的封锁申请和释放操作，其流程如下:

$xI_1(A); r_1(A); xI_2(A); r_2(A); w_2(A); xI_2(B); r_2(B); w_1(A);$

$xI_1(B); r_1(B); w_1(B); u_1(A); u_1(B); w_2(B); u_2(A); u_2(B);$

$xl_1(A); r_1(A); xl_2(A); r_2(A); w_2(A); xl_2(B); r_2(B); w_1(A);$
 $xl_1(B); r_1(B); w_1(B); u_1(A); u_1(B); w_2(B); u_2(A); u_2(B);$

- 在上述（预想）执行流程中，'封锁调度器'在处理事务 T_2 的 $xl_2(A)$ 请求时，因封锁申请得不到满足而将事务 T_2 及其所有的后续操作请求推迟，真正的操作执行流程是（下面的 $xl(A)$ 是申请并获得A上的‘X锁’的时间）

$xl_1(A); r_1(A); w_1(A); xl_1(B); r_1(B); w_1(B); u_1(A); u_1(B);$
 $xl_2(A); r_2(A); w_2(A); xl_2(B); r_2(B); w_2(B); u_2(A); u_2(B);$

- 最终产生的是 T_1 和 T_2 之间的一个串行调度

5.2.4 两阶段封锁协议

定理: 由2PL事务所构成的任意合法调度S都是冲突可串行化的

证明:

1. 当调度S仅由一个事务组成时，调度S是冲突可串行化的
2. 假设：由($n-1$)个2PL事务所构成的任意一个合法调度都是冲突可串行化的
3. 设调度S涉及n个2PL事务： T_1, T_2, \dots, T_n ，并且 T_i 是调度S中第一个有解锁动作的事务，则我们可以得到以下结论：
 - * 可以将 T_i 的所有动作不经过任何冲突而移动到调度S的最前面

□ 设在 T_i 中有一个动作 $w_i(A)$ (或 $r_i(A)$)，如果调度 S 在该动作的前面有一个与之冲突的动作 $w_j(A)(i \neq j)$ ，那么调度 S 的情况必是：

...; $w_j(A)$; ...; $u_j(A)$; ...; $l_i(A)$; ...; $w_i(A)$; ...

$\because T_i$ 是调度 S 中第一个有解锁动作的事务

\therefore 在 $u_j(A)$ 之前必存在 T_i 中的一个解锁动作(如 $u_i(B)$)，则

调度 S 变为： ...; $w_j(A)$; ...; $u_i(B)$; ...; $u_j(A)$; ...;

$l_i(A)$; ...; $w_i(A)$; ...

在上述的调度 S 中，仅考虑与事务 T_i 有关的动作序列

□ 在上述的调度 S 中，仅考虑与事务 T_i 有关的动作序列：

...; $u_i(B)$; ...; $l_i(A)$; ...; $w_i(A)$; ...

这不符合 2PL 事务的定义，与 T_i 是 2PL 事务相矛盾。

∴ 在 T_i 的每个动作 $w_i(A)$ (或 $r_i(A)$)之前，都不存在与其产生冲突且属于其它事务的动作

∴ 结论 * 成立

- 根据得到的结论 *，我们可以将调度S转换为另一个冲突等价的调度S'：
 $(T_i\text{的所有动作}) (\text{其它}(n-1)\text{个2PL事务的动作})$
- 并且维持其后半部分动作在调度S中的原有顺序不变；
- 其中 T_i 的封锁/解锁动作在转换后可以恢复到调度S'中去。

□ 调度S':

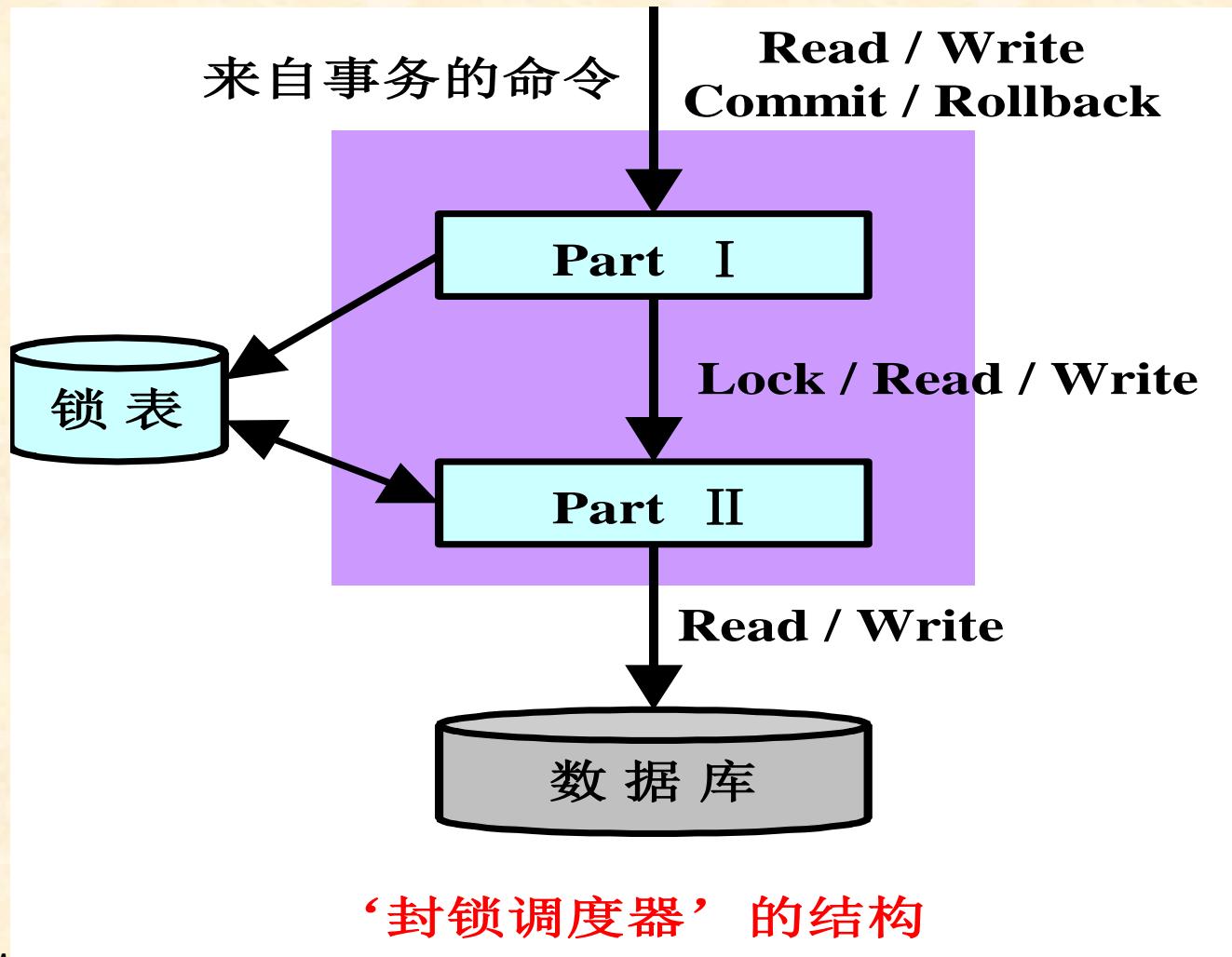
(T_i 的所有动作) (其它 $(n-1)$ 个 2PL 事务的动作)

- ∴ 调度 S 是一个合法调度
- ∴ 调度 S' 的后半部分是其它 $(n-1)$ 个 2PL 事务的一个合法调度
- ∴ 根据步骤 2 的归纳假设的前提可得：调度 S' 的后半部分是其它 $(n-1)$ 个 2PL 事务的一个‘冲突可串行化’的调度，即冲突等价于这 $(n-1)$ 个 2PL 事务的某个串行调度。
- ∴ 调度 S' 冲突等价于这 n 个 2PL 事务的某个串行调度
- ∴ 调度 S' 是冲突可串行化的
- ∴ 调度 S 也是冲突可串行化的

证毕

‘封锁调度器’的实现

□ 用于构造‘两阶段封锁事务’并实现它们的合法调度的‘封锁调度器’的结构如下图所示



- a) Part I : 接受事务产生的操作请求，并决定是否需要为该请求申请适当的封锁
 - 即：在操作请求命令前插入适当的锁申请动作（Lock）

- b) Part II：接受Part I 传来的封锁或数据库访问请求
- 如果‘请求’是数据库访问操作，则该请求将被传送到数据库中执行
 - 如果‘请求’是一个封锁动作，它将查看‘锁表’以决定锁是否能被授予
 - 如果是，则修改‘锁表’，并将刚刚授予的锁包括进去
 - 如果不是，那么‘锁表’中必须加入一项以表明该锁已经被申请。在这以后，Part II 将推迟事务 T(发出该访问请求的事务)的进一步动作，直到锁申请得到满足的时候

- c) 当事务T结束时，‘事务管理器’将通知Part I，于是Part I 释放事务T持有的所有封锁
 - 如果有事务正在等待这些锁中的任何一个，则Part I 将通知Part II
 - d) 当Part II 被告知某个数据库元素A上的锁可以获得时，它决定接下来能获得A上的锁的一个或多个事务。Part II 将接着执行这些事务被推迟的请求
- 在上述执行情况下，每个事务都满足2PL要求

5.2 并发控制技术

5.2.1 事务的并发执行

5.2.2 封锁

5.2.3 封锁协议

5.2.4 两阶段封锁协议

5.2.5 封锁粒度

5.2.6 活锁与死锁

5.2.5 封锁粒度

□ 主要内容

- 封锁粒度

- 封锁粒度、系统并发度、并发控制的开销三者之间的关系

- 多粒度封锁

- 意向锁

5.2.5 封锁粒度

□ 封锁粒度 (Granularity)

- 一把锁可以封锁的数据对象的大小
- 锁的封锁对象可以是数据库中的逻辑数据单元，也可以是物理数据单元

□ 以关系数据库系统为例，可以采用的封锁粒度有：

- 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

5.2.5 封锁粒度

- ‘封锁粒度’ 与 ‘系统并发度’ 的关系
- ‘封锁粒度’ 与 ‘并发控制的开销’ 的关系

封锁粒度	系统并发度	并发控制的开销
大	低	小
小	高	大

5.2.5 封锁粒度

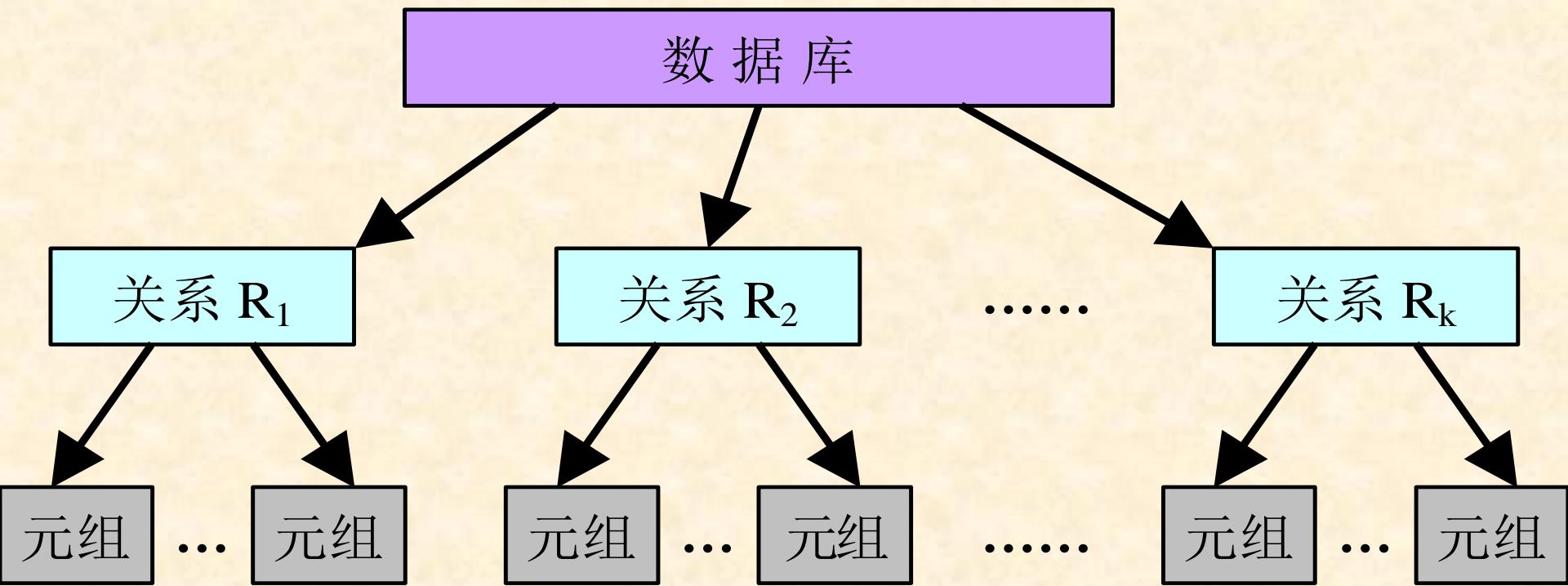
□ 多粒度封锁 (Multiple Granularity Locking)

- 如果在一个系统中同时支持多种封锁粒度供事务选择使用，这种封锁方法被称为‘多粒度封锁’
- 通过选择合适的‘封锁粒度’来达到如下目的
 - 通过加大‘封锁粒度’来减少‘锁’的数量，降低并发控制的开销
 - 通过降低‘封锁粒度’来缩小一把锁可以封锁的数据范围，减少封锁冲突现象，提高系统并发度
- 可以按照封锁粒度的大小构造出一棵‘多粒度树’，以树中的每个结点作为封锁对象，可以构成一个‘多粒度封锁协议’

5.2.5 封锁粒度

□ 多粒度树

➤ 例：以关系数据库为例



三个层次的多粒度树

5.2.5 封锁粒度

□ 多粒度封锁协议

1. 可以对‘多粒度树’中的每个结点独立加锁(显式封锁)
2. 对一个结点加锁意味着该结点的所有后裔结点也被加以同样类型的锁(隐式封锁)

- ‘隐式封锁’与‘显式封锁’的作用是一样的
- 如果希望对‘多粒度树’中的某个结点加锁，则需要在该结点的所有先驱结点和后裔结点中都进行锁的相容性检查，这样的检查方法效率很低。为了解决此问题，提出了一种新的封锁类型：
 - ❖ 意向锁

5.2.5 封锁粒度

□ 意向锁 (Intention Lock)

- ‘意向锁’ 的使用规定
 - 如果对一个结点加 ‘意向锁’，则说明该结点的下层结点正在被加锁
 - 对任一结点加锁时，必须先对它的上层结点加 ‘意向锁’
- 三种常见的 ‘意向锁’
 - 意向共享锁 (IS锁)
 - 意向排它锁 (IX锁)
 - 共享意向排它锁 (SIX锁)
- 可以联合使用上述三种意向锁和原有的 ‘S锁’ 、‘X锁’ 共同实现基于封锁技术的并发控制

5.2.5 封锁粒度

□ 意向共享锁 (IS锁)

- 如果对结点N加‘IS锁’，表示准备在结点N的某些后裔结点上加‘S锁’

□ 意向排它锁 (IX锁)

- 如果对结点N加‘IX锁’，表示准备在结点N的某些后裔结点上加‘X锁’

□ 共享意向排它锁 (SIX锁)

- 如果对结点N加‘SIX锁’，表示对结点N本身加‘S锁’，并准备在N的某些后裔结点上加‘X锁’

5.2.5 封锁粒度

□ 带有意向锁的锁相容矩阵

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

Yes: 表示相容的请求 **No:** 表示不相容的请求

5.2.5 封锁粒度

□ 带有意向锁的锁相容矩阵

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

Yes: 表示相容的请求

No: 表示不相容的请求

5.2.5 封锁粒度

□ 多粒度封锁协议

- 如果要对一个结点加锁，必须先对它的上层结点加意向锁
 - 申请封锁的顺序：自上而下
 - 释放封锁的顺序：由底向上

□ 具有‘意向锁’的多粒度封锁方法的优点

- 减少了‘加锁’和‘解锁’的开销
- 提高了系统的并发度

5.2 并发控制技术

5.2.1 事务的并发执行

5.2.2 封锁

5.2.3 封锁协议

5.2.4 两阶段封锁协议

5.2.5 封锁粒度

5.2.6 活锁与死锁

5.2.6 活锁与死锁

□ 死锁 (deadlock)

- 每个事务都可能拥有一部分‘锁’，并因申请其它事务所持有的‘锁’而等待，因此产生的循环等待现象被称为‘死锁’

5.2.6 活锁与死锁

□ 死锁的例子

- 假设在数据库中有两个数据对象：A和B，并存在下述两个事务：
 - 事务T₁：根据读到的A的值来修改B上的值，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上的值，其数据库访问的操作流程如下：r₂(B); r₂(A); w₂(A);
 - 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);

死锁的例子（一）

	事务T ₁	事务T ₂	A的封锁状态	B的封锁状态
1	sl₁(A);		S(T₁)	
2	r₁(A);			
3	sl₁(B);			S(T₁)
4	r₁(B);			
5		sl₂(B);		S(T₁, T₂)
6		r₂(B);		
7	xl₁(B);			
8	Wait...	sl₂(A);	S(T₁, T₂)	
9	Wait...	r₂(A);		
10	Wait...	xl₂(A);		
11	Wait...	Wait...		
12	Wait...	Wait...		

死锁的例子（二）

	事务T ₁	事务T ₂	A的封锁状态	B的封锁状态
1	sl₁(A);		S(T₁)	
2	r₁(A);			
3		sl₂(B);		S(T₂)
4		r₂(B);		
5	xl₁(B);			
6	Wait...	xl₂(A);		
7	Wait...	Wait...		
8	Wait...	Wait...		

5.2.6 活锁与死锁

□ ‘死锁’问题的处理办法

1. 预防法
2. 解除法

1. 预防法

- 采用一定的锁申请操作方式以避免死锁现象的发生
 - a) 顺序申请法
 - b) 二次申请法

5.2.6 活锁与死锁

□ 解除法

- 允许产生死锁，当系统通过‘死锁检测’程序发现出现死锁现象时，可以调用解锁程序来解除死锁

a) 超时死锁检测法

- ❖ 事务的执行时间超时
- ❖ 锁申请的等待时间超时

b) 等待图法

c) 时间戳死锁检测法

□ 时间戳死锁检测法

- 每个事务都具有一个用于死锁检测的时间戳，该时间戳反映当前事务的新老程度（即已运行时间的长短）
- 如果事务**T**必须等待另一个事务**U**所持有的锁，那么有两种死锁检测策略：
 - 等待-死亡方案
 - ❖ 如果**T**比**U**老，那么允许**T**等待**U**持有的锁
 - ❖ 如果**U**比**T**老，那么事务**T**死亡（被回滚）
 - 伤害-等待方案
 - ❖ 如果**T**比**U**老，它将伤害**U**，**U**必须被回滚
 - ❖ 如果**U**比**T**老，那么**T**等待**U**持有的锁
- 上述两种策略都可以避免死锁的发生，并且牺牲的往往是较年轻的事务

5.2.6 活锁与死锁

□ 活锁 (livelock)

- 有部分事务因封锁申请得不到满足而处于长期等待状态，但其它的事务仍然可以继续运行下去，这种情况被称为“活锁”
- 解决方法
 - 先来先服务

□ 活锁的例子

- 假设有两个有关某个航班的剩余票额（数据对象A）的应用例子（事务）：
 - 事务T₁（卖票）： x1₁(A) ; r₁(A) ; w₁(A) ; u₁(A) ;
 - 事务T₂（剩余票额查询）： s1₂(A) ; r₂(A) ; u₂(A) ;
- 假设先启动第一个事务T₂，然后再启动一个事务T₁
 - 在T₂结束之前，T₁处于等待状态
 - 在第一个T₂结束之前，另外一个用户又启动了一个事务T₂'（暂时称其为T₂'）。由于‘S锁’是可以共存于同一个数据对象上的，因此T₂'也能获得A上的‘S锁’而运行
 - 如果在已经运行的事务T₂结束之前，不断有新启动的事务T₂'加进来，则事务T₁将处于一种永久等待的状态。此种现象就称为‘活锁’

避免出现‘活锁’问题的 **read_lock(A)**

```
B: if LOCK(A) = 'Unlocked' {  
    LOCK(A) := 'Read_locked';  
    no_of_reads(A) := 1;  
} else {  
    if ( LOCK(A) = 'Read_locked' AND  
        A的封锁等待队列不为空 )  
        no_of_reads(A) := no_of_reads(A) + 1;  
    else {  
        /* 将自身加入到等待者队列的最后面 */  
        wait ( until LOCK(A) != 'Write_locked'  
            and the lock manager wakes up  
            the transaction);  
        go to B;  
    }  
}
```

避免出现‘活锁’问题的 **write_lock(A)**

B:

```
if LOCK(A) = 'Unlocked'  
{  
    LOCK(A) := 'Write_locked';  
}  
else  
{  
    /* 将自身加入到等待者队列的最后面 */  
    wait ( until LOCK(A) = 'Unlocked' and the lock  
           manager wakes up the transaction);  
    go to B;  
}
```

避免出现‘活锁’问题的 unlock(A)

```
if LOCK(A) = ‘Write_locked’ {  
    LOCK(A) := ‘Unlocked’;  
/* 唤醒等待队列中的第一个事务T，如果事务T申请的是‘S  
锁’，那么唤醒在T后面紧接着的所有正在申请‘S锁’的事  
务 */  
}  
else if LOCK(A) = ‘Read_locked’  
{  
    no_of_reads(A) := no_of_reads(A) - 1;  
    if no_of_reads(A) = 0  
    {  
        LOCK(A) := ‘Unlocked’;  
/* 唤醒等待队列中的第一个事务T，如果事务T申请的是‘S锁’，  
那么唤醒在T后面紧接着的所有正在申请‘S锁’的事务 */  
    }    }
```

第五章 事务处理、并发控制与故障恢复技术

5.3 数据库恢复技术

5.3 数据库恢复技术

5.3.1 概述

5.3.2 数据库故障分类

- 小型故障
- 中型故障
- 大型故障

5.3.3 数据库故障恢复三大技术

- 数据转储
- 日志
- 事务的撤消与重做

5.3.4 恢复策略

5.3.5 数据库镜像

5.3.1 概述

□ 数据库恢复

➤ 在数据库遭受破坏后及时进行恢复的功能

➤ 方法

- 利用数据冗余原理，将数据库中的数据在不同的存储介质上进行冗余存储，当数据库本身受到破坏时，可以利用这些冗余信息进行恢复

➤ 常用措施

- 数据转储
- 日志
- 数据库镜像

5.3.2 数据库故障分类

➤ 共分三类、六种情况

□ 小型故障

➤ 事务内部故障

- 故障的影响范围在一个事务之内，不影响整个系统的正常运行

□ 中型故障

➤ 系统故障

➤ 外部影响

- 可导致整个系统停止工作，但磁盘数据不受影响。在系统重启时，可通过当前的日志文件进行恢复

5.3.2 数据库故障分类

□ 大型故障

- 磁盘故障
- 计算机病毒
- 黑客入侵
 - 可导致内存及磁盘数据的严重破坏，需要对数据库做彻底的恢复

5.3.3 数据库故障恢复三大技术 - 转储

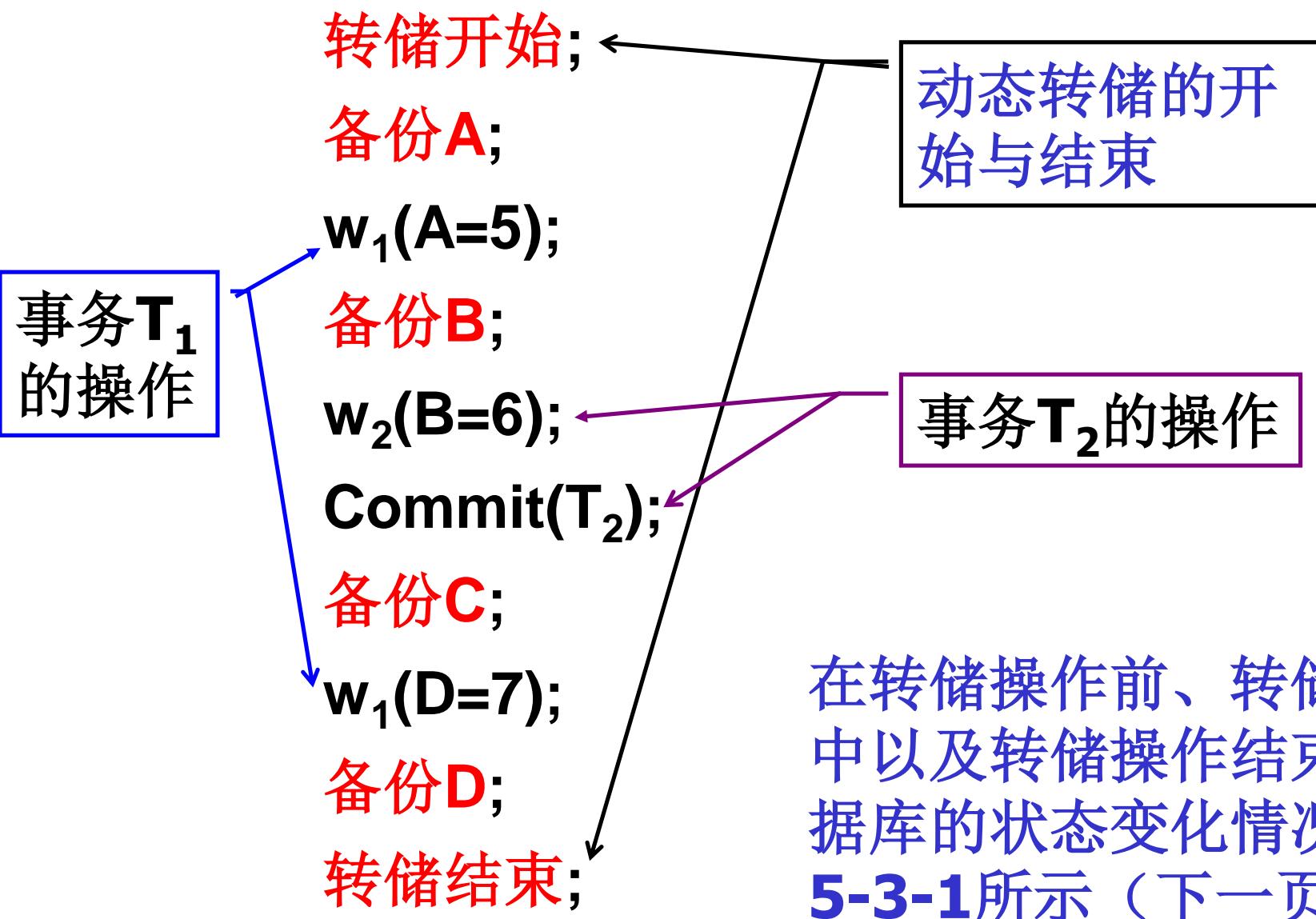
□ 数据转储 (dump)

- 定期地将数据库中的内容复制到其它存储设备中去的过程
- 后备（后援）副本
 - 经转储而得到的备份数据
- 转储的分类
 - 静态转储 **vs** 动态转储
 - 海量转储 **vs** 增量转储

□ 动态转储的例子

- 假设以A, B, C, D的顺序备份数据库中的值，且其初始值分别为1, 2, 3, 4
- 在转储过程中，有两个事务正在运行：
 - T_1 : $w_1(A=5); w_1(D=7);$
 - T_2 : $w_2(B=6);$

□ 在转储过程中，对于数据库的访问操作的执行顺序如下：



在转储操作前、转储过程中以及转储操作结束后数据库的状态变化情况如图 5-3-1 所示（下一页）

	T_1	T_2	转储	后备副本	A	B	C	D	日志
1			Start		1	2	3	4	<Start dump>
2			备份A	$A=1$	1	2	3	4	
3	$w_1(A=5);$				5	2	3	4	< $T_1, A, 1, 5$ >
4			备份B	$B=2$	5	2	3	4	
5		$w_2(B=6);$			5	6	3	4	< $T_2, B, 2, 6$ >
6		Commit;			5	6	3	4	<Commit T_2 >
7			备份C	$C=3$	5	6	3	4	
8	$w_1(D=7);$				5	6	3	7	< $T_1, D, 4, 7$ >
9			备份D	$D=7$	5	6	3	7	
10			End		5	6	3	7	<End dump>

图5-3-1 日志的例子（动态转储）

- 可以看到，上述的转储过程所得到的后备副本并不能保证数据库中数据的一致性。如果使用该副本进行数据库的故障恢复，需要结合当时记载的数据库日志信息
- 在日志中需要记载：
 - ① 转储的开始点和结束点
 - ② 在转储的执行过程中，事务的更新操作情况：
 <事务标识，更新对象，更新前的值，更新后的值>
 - ③ 在转储的执行过程中完成的事务的结束状态
 - Commit / Abort

5.3.3 数据库故障恢复三大技术 – 日志

□ 日志 (log)

- 是由数据库系统创建和维护的，用于自动记载数据库中修改型操作的数据更新情况的文件

□ 日志的内容

- 每个更新操作的事务标识、更新对象、更新前的值 和/或 更新后的值
- 每个事务的开始、结束等执行情况
- 其它信息

□ 日志的组成

- 日志是‘日志记录’的一个序列，每个‘日志记录’记载有关某个事务已执行操作的情况。由于日志主要用于**DB**的故障恢复，所以这里记载的主要是事务的更新操作的执行情况
- 由于事务通常是并发执行的，所以多个事务的日志记录通常是交错在一起的

日志的作用

1. 确保事务执行的原子性
2. 实现增量转储
3. 实现故障恢复
 - 在发生系统故障时，为了修复故障所产生的影响，某些事务的操作将会被重做（**redo**），而另一些事务的操作将会被撤消（**undo**）
 - 为了区分哪些事务将被重做，哪些事务将被撤消，在日志中需要记载每个事务的结束标志：
 - 提交（**commit**）：将被重执
 - 放弃（**abort**）： 将被撤消
 - 在日志中尚未有结束标志的事务，在进行故障恢复时将等同于被放弃的事务看待

□ 日志的书写原则

- 按照操作执行的先后次序，遵循先写日志，后修改数据库的原则

□ 几种不同的日志

➤ **undo**日志

- 用于被放弃事务（包括在发生故障时尚未结束的事务）的撤消工作

➤ **redo**日志

- 用于已提交事务的重做工作

□ 不同类型日志的更新记录是不一样的，也可以将上述两种类型的日志合并为一种日志：

➤ **undo/redo**日志

□ undo日志的记录格式

1. 开始一个事务: <Start T>
2. 提交事务T: <Commit T>
3. 放弃事务T: <Abort T>
4. 更新记录: <T, X, V>
 - 事务T修改了数据库元素X的值, 而X的旧值是V

□ undo日志的记载规则

- U₁:** 如果事务T修改了数据库元素X, 则更新日志<T,X,V>必须在X的新值写到磁盘前写到磁盘
- U₂:** 如果事务T提交, 则日志记录<Commit T>必须在事务T改变的所有DB元素已写到磁盘后再写到磁盘

undo日志的例子

- 引入一个新的操作：Flush Log
 - 将保存在内存中的日志记录全部写到日志文件的磁盘中去
- 同时引入下述几个符号
 - D-A：数据库元素A在计算机磁盘中的值
 - M-A：数据库元素A在内存缓冲中的值
 - t：表示内存变量
- 假设有一个事务T，需要将数据库中A和B两个数据对象上的值分别乘以2，其执行流程和数据库中记载的日志信息如图5-3-2所示（假设A，B的初始值都是8）

图5-3-2 undo日志的例子

	T	t	M-A	M-B	D-A	D-B	undo日志
1					8	8	<Start T>
2	Read(A,t)	8	8		8	8	
3	$t := t^2$	16	8		8	8	
4	Write(A,t)	16	16		8	8	<T, A, 8>
5	Read(B,t)	8	16	8	8	8	
6	$t := t^2$	16	16	8	8	8	
7	Write(B,t)	16	16	16	8	8	<T, B, 8>
8	Flush Log						
9	Output(A)		16	16	16	8	
10	Output(B)		16	16	16	16	
11							<Commit T>
12	Flush Log						

图5-3-2 undo日志的例子

	T	t	M-A	M-B	D-A	D-B	undo日志
1					8	8	<Start T>
2	Read(A,t)	8	8		8	8	
3	$t := t^2$	16	8		8	8	
4	Write(A,t)	16	16		8	8	<T, A, 8>
5	Read(B,t)	8	16	8	8	8	
6	$t := t^2$	16	16	8	8	8	
7	Write(B,t)	16	16	16	8	8	<T, B, 8>

图5-3-2 undo日志的例子

	T	t	M-A	M-B	D-A	D-B	undo日志
1					8	8	<Start T>
2	Read(A,t)	8	8		8	8	
3	$t := t^2$	16	8		8	8	
4	Write(A,t)	16	16		8	8	<T, A, 8>
5	Read(B,t)	8	16	8	8	8	
6	$t := t^2$	16	16	8	8	8	
7	Write(B,t)	16	16	16	8	8	<T, B, 8>
8	Flush Log						
9	Output(A)						
10	Output(B)						

why?

图5-3-2 undo日志的例子

	T	t	M-A	M-B	D-A	D-B	undo日志
1					8	8	<Start T>
2	Read(A,t)	8	8		8	8	
3	$t := t^2$	16	8		8	8	
4	Write(A,t)	16	16		8	8	<T, A, 8>
5	Read(B,t)	8	16	8	8	8	
6	$t := t^2$	16					
7	Write(B,t)	16					
8	Flush Log						<p>U₁: 如果事务T修改数据库元素X，则更新日志<T,X,V>必须在X的新值写到磁盘前写到磁盘</p>
9	Output(A)						
10	Output(B)						<p>U₂: 如果事务T提交，则日志记录<Commit T>必须在事务T改变的所有DB元素已写到磁盘后再写到磁盘</p>

图5-3-2 undo日志的例子

	T	t	M-A	M-B	D-A	D-B	undo日志
1					8	8	<Start T>
2	Read(A,t)	8	8		8	8	
3	$t := t^2$	16	8		8	8	
4	Write(A,t)	16	16		8	8	<T, A, 8>
5	Read(B,t)	8	16	8	8	8	
6	$t := t^2$	16	16	8	8	8	
7	Write(B,t)	16	16	16	8	8	<T, B, 8>
8	Flush Log						
9	Output(A)		16	16	16	8	
10	Output(B)		16	16	16	16	
11							<Commit T>
12	Flush Log	← why?					

图5-3-2 undo日志的例子

	T	t	M-A	M-B	D-A	D-B	undo日志
1					8	8	<Start T>
2	Read(A,t)	8	8		8	8	
3	$t := t^2$	16	8		8	8	
4	Write(A,t)	16	16		8	8	<T, A, 8>
5	Read(B,t)	8	16	8	8	8	
6	$t := t^2$	16	16	8	8	8	
7	Write(B,t)	16	16	16	8	8	<T, B, 8>
8	Flush Log						
9	Output(A)		16	16	16	8	
10	Output(B)		16	16	16	16	
11							<Commit T>
12	Flush Log						

确保事务T被提交!

使用undo日志的恢复过程

1. 将所有事务划分为两个集合
 - 已提交事务：有 $\langle \text{Start T} \rangle$ 和 $\langle \text{Commit T} \rangle$
 - 未提交事务：有 $\langle \text{Start T} \rangle$ 但没有 $\langle \text{Commit T} \rangle$
2. 从undo日志的尾部开始向后（日志头部）扫描整个日志，对每一条更新记录 $\langle T, X, V \rangle$ 作如下处理：
 - 如果 $\langle \text{Commit T} \rangle$ 已被扫描到，则继续扫描下一条日志记录（基于规则 U_2 ）
 - 否则，由恢复管理器将数据库中X的值改为V（基于规则 U_1 ）
3. 在日志的尾部为每个未结束事务写入一条日志记录 $\langle \text{Abort T} \rangle$ ，并刷新日志（Flush Log）

- 在图5-3-2中，根据系统故障发生的不同时机，对于事务T的恢复处理过程可能会不一样
- 假设系统故障发生在：
 - 1) 第12步(**Flush Log**)之后
 - 2) 第11步(**Commit T**)与第12步(**Flush Log**)之间
 - 日志记录<**Commit T**>已写到磁盘
 - 日志记录<**Commit T**>尚未写到磁盘
 - 3) 第10步(**Output(B)**)与第11步(**Commit T**)之间
 - 4) 第8步(**Flush Log**)与第10步(**Output(B)**)之间
 - 5) 第8步(**Flush Log**)之前

情况1：故障发生在第12步之后

	T	t	M-A	M-B	D-A	D-B	undo日志
1					8	8	<Start T>
2	Read(A,t)		8	8			
3	$t := t^*2$		16				
4	Write(A,t)						>
5	Read(B,						
6	$t := t^*2$						
7	Write(B,t,						, B, 8>
8	Flush Log						
9	Output(A)		16	16	16	8	
10	Output(B)		16	16	16	16	
11							<Commit T>
12	Flush Log						

系统重启后，事务T将被作为一个已提交的事务，在故障恢复过程中，其日志记录不需要作任何处理！

情况2：故障发生在第11步与第12步之间



无论是否执行过（多少）**OUTPUT**操作，都将根据日志文件磁盘中记载的更新日志记录来恢复他们在数据库磁盘中的原有值！

	D-B	undo日志
5	8	<Start T>
6	8	
7	8	
8	8	<T, A, 8>
9	8	
10	8	
	系统崩溃	

手写注释说明：

- 第5步：D-B = 8, undo日志 = <Start T>
- 第6步：D-B = 8, undo日志 = <Start T>
- 第7步：
 - t := t * 2
 - Write(B, t)
 D-B = 16, undo日志 = <T, B, 8>
- 第8步：Flush Log, D-B = 16, undo日志 = <T, B, 8>
- 第9步：Output(A), D-B = 16, undo日志 = 16, 16
- 第10步：Output(B), D-B = 16, undo日志 = 16, 16
- 崩溃原因：由于系统崩溃，导致未完成的事务T1被回滚，从而导致D-B与undo日志不一致。

情况4：故障发生在第8步与第10步之间

	T	t	M-A	M-B	D-A	D-B	undo日志
1					8	8	<Start T>
2					8	8	
3					8	8	
4					8	8	<T, A, 8>
5			6	8	8	8	
6	$t :=$	16	6	8	8	8	
7	Write(B,t)	16	16	16	8	8	<T, B, 8>
8	Flush Log						
	系统崩溃，不知有没有执行过Output(A)和Output(B)						

情况5：故障发生在第8步之前

	T	D-B	undo日志
1			<Start T>?
2			
3			
4			
5		8	<T, A, 8>?
6	t := t*2	16	8
7	Write(B,t)	16	16
		16	8
		8	8
		8	<T, B, 8>?
	在第1步到第8步之间的任何时间点上出现系统崩溃		

思考题：如果在恢复过程中再次出现系统崩溃，如何进行系统的故障恢复？

□ 检查点 (checkpoint)

- 上述恢复过程需要扫描整个undo日志文件，为了降低数据库故障恢复的开销，可以定期地在日志文件中插入检查点
- 在日志中插入检查点的处理过程包括：
 1. 系统停止接受‘启动新事务的请求’
 2. 等到所有当前活跃的事务被提交或中止，并且在日志中写入了<Commit T>或<Abort T>记录
 3. 将日志记录刷新到磁盘
 4. 写入日志记录<CKPT>，并再次刷新日志
 5. 重新开始接受新的事务
- 在故障恢复时，只要逆向扫描到第一条<CKPT>记录（最后一个被记入的检查点）就可以结束故障恢复工作

设置检查点的例子

- 日志的开始情况如图5-3-3所示

undo日志
<Start T ₁ >
<T ₁ , A, 5>
<Start T ₂ >
<T ₂ , B, 10>

图5-3-3

- 假设现在需要插入一个检查点，则插入检查点后的日志可能如图5-3-4所示

undo日志
<Start T ₁ >
<T ₁ , A, 5>
<Start T ₂ >
<T ₂ , B, 10>
.....
<T ₂ , C, 15>
<T ₁ , D, 20>
<Commit T ₁ >
<Abort T ₂ >
<CKPT>
<Start T ₃ >
<T ₃ , E, 25>

图5-3-4

□ 非静止检查点

- 在设置检查点的过程中，允许新的事务进入系统

□ 设置‘非静止检查点’的步骤包括

1. 写入日志记录<**Start CKPT(T_1, \dots, T_k)**>，并刷新日志；其中： T_1, \dots, T_k 是当前所有活跃事务的标识符
2. 等待 T_1, \dots, T_k 中的每一个事务的提交或中止，但允许开始执行其它新的事务
3. 当 T_1, \dots, T_k 都已经完成时，写入日志记录<**End CKPT**>并刷新日志

设置非静止检查点的例子

图5-3-5

undo日志
<Start T ₁ >
<T ₁ , A, 5>
<Start T ₂ >
<T ₂ , B, 10>
<Start CKPT(T ₁ , T ₂)>
<T ₂ , C, 15>
<Start T ₃ >
<T ₁ , D, 20>
<Commit T ₁ >
<T ₃ , E, 25>
<Commit T ₂ >
<End CKPT>
<T ₃ , F, 30>

□ 带有非静止检查点日志的恢复

- 从日志尾部向后（日志头部）扫描日志文件进行故障恢复
- 可能会出现以下的两种情况：
 1. 先遇到<End CKPT>记录
 2. 先遇到<Start CKPT(T_1, \dots, T_k)>记录

带有非静止检查点undo日志的恢复

1. 先遇到<End CKPT>记录

- 继续向后扫描，直到出现与之相对应的<Start CKPT (...)>记录就可以结束故障恢复工作，在这之后的日志记录是没有用处的，可以被抛弃

2. 先遇到<Start CKPT(T_1, \dots, T_k)>记录，此种情况下的故障恢复工作需要撤消两类事务的操作：

- 在<Start CKPT(T_1, \dots, T_k)>记录之后启动的事务
 - 在扫描到<Start CKPT(T_1, \dots, T_k)>记录时，这类事务的操作已经被撤消
- T_1, \dots, T_k 中在系统崩溃前尚未完成的事务
 - 继续向后扫描日志，直至其中未完成事务的访问操作被全部撤消

带有非静止检查点的故障恢复（图5-3-6）

undo日志
.....
<Start T ₁ >
<T ₁ , A, 5>
<Start T ₂ >
<T ₂ , B, 10>
<Start CKPT(T ₁ , T ₂)>
<T ₂ , C, 15>
<Start T ₃ >
<T ₁ , D, 20>
<Commit T ₁ >
<T ₃ , E, 25>

到这里undo操作
就可以结束！

- 对于已提交事务(T₁)的日志记录不做任何处理
- undo所有未提交事务(T₂, T₃)的操作

□ undo日志的不足

- 在将事务改变的所有数据写到磁盘前不能提交该事务
 - 上述要求将会导致：在事务的提交过程中需要执行许多‘写’磁盘操作，从而增加了事务提交的时间开销
-
- 能否将数据库的修改结果暂时保存在内存中，在需要的时候再写入磁盘，以减少磁盘I/O的次数？

□ redo日志的记录格式

➤ redo日志的记录格式与undo日志一样，唯一的区别是：在更新记录 $\langle T, X, V \rangle$ 中记载的是更新后的值

□ redo日志的记载规则

R₁: 在修改磁盘上的任何数据库元素X之前，要保证所有与X的这一修改有关的日志记录（包括更新记录 $\langle T, X, V \rangle$ 和提交记录 $\langle Commit\ T \rangle$ ）都必须出现在磁盘上

图5-3-7 redo日志的例子

	T	t	M-A	M-B	D-A	D-B	redo日志
1					8	8	<Start T>
2	Read(A,t)	8	8		8	8	
3	$t := t^2$	16	8		8	8	
4	Write(A,t)	16	16		8	8	<T, A, 16>
5	Read(B,t)	8	16	8	8	8	
6	$t := t^2$	16	16	8	8	8	
7	Write(B,t)	16	16	16	8	8	<T, B, 16>
8							<Commit T>
9	Flush Log						
10	Output(A)		16	16	16	8	
11	Output(B)		16	16	16	16	

□事实上，Output(A)和Output(B)事件可能发生得更晚

□ 使用redo日志的恢复过程

1. 确定所有已提交的事务
2. 从日志文件的首部开始扫描日志，对遇到的每一条更新记录 $\langle T, X, V \rangle$ ：
 - 如果T是未提交事务，则继续扫描日志
 - 如果T是已提交的事务，则为数据库元素X写入新值V
3. 对每个未完成的事务T，在日志的尾部写入结束标志 $\langle \text{Abort } T \rangle$ 并刷新日志

□ 以图5-3-7为例来看使用redo日志的恢复过程。如果系统故障发生在：

1) 第9步之后

- 由于与事务T有关的日志记录已经全部写到磁盘，因此恢复子系统认为T是一个已提交的事务。在恢复过程中会根据更新记录将A和B的新值写入数据库的磁盘中(有可能是冗余的写操作)

2) 第8步和第9步之间

- 如果 $\langle\text{Commit T}\rangle$ 已到达磁盘，则恢复过程同情况 1)
- 如果 $\langle\text{Commit T}\rangle$ 尚未到达磁盘，则恢复过程同情况 3)

3) 第8步之前

- T被看成一个未完成的事务，由于T尚未修改任何数据库元素，因此在恢复过程中将对T的更新记录 $\langle T, X, V \rangle$ 不做任何处理，只是在日志的尾部写入一条记录 $\langle\text{Abort T}\rangle$

虽然事务T的修改结果还没有来得及被写回（**OUTPUT**）数据库的磁盘，但由于其更新日志和提交日志都已经被写入到日志文件的磁盘（**Flush Log**），因此，在故障恢复过程中，一定可以从日志文件中找到事务T的所有修改结果，并被重新写入数据库的磁盘（**REDO**）

后

B	D-A	D-B	redo日志
	8	8	<Start T>
	8	8	
	8	8	
	8	8	<T,A,16>
	8	8	
	8	8	
	8	8	<T,B,16>
			<Commit T>

9

Flush Log

系统崩溃

需要根据事务T的提交日志
(**Commit T**) 是否已经被写入日志文件的磁盘来决定其故障恢复的处理方法:

- a) 如果已经被写入日志文件的磁盘，那么同前面的情况1) 处理
- b) 否则，将按照后面的情况3) 来处理

		A		D-B		redo日志	
7	8			8		<Start T>	
				8			
				8			
				8		<T,A,16>	
				8			
				8			
				8			
				8		<T,B,16>	
				8			
				8			
						<Commit T>?	

系统崩溃

在日志文件中肯定没有事务**T**的结束标志，在故障恢复过程中，事务**T**将被作为一个未结束的事务，其所有的日志都不会被处理。

在故障发生前，事务**T**可能已经修改了某些数据库元素，但这些修改仅仅发生在内存缓冲区中，并没有修改其在数据库磁盘中的值。因此不影响整个数据库的数据一致性。

前

B	D-A	D-B	redo日志
	8	8	<Start T>
	8	8	
	8	8	
	8	8	<T,A,16>
	8	8	
	8	8	
	8	8	<T,B,16>

有时候出现系统崩溃

□ redo日志与undo日志的主要区别

- 1) 恢复的目的不一样
- 2) 提交记录<Commit T>写入日志的时间不一样
 - undo日志：在事务T的所有数据库磁盘修改操作(Output)结束后才能在日志中写入提交记录<Commit T>
 - redo日志：在提交记录<Commit T>被写入磁盘后才能将事务T更新后的值写入数据库的磁盘中
- 3) 在更新记录<T,X,V>中保存的值V不一样

□ redo日志的检查点

- 在redo日志中插入检查点时，由于已提交事务所做的修改被写入数据库磁盘的时间可能比事务提交的时间要晚得多，因此在插入检查点时，不仅仅需要考虑当前有哪些事务是活跃的，还要确保当前已提交事务的所有修改被写入到数据库的磁盘中去
- 为了做到这一点，系统必须知道：
 1. 有哪些内存缓冲区被修改过，但还没有将修改结果写入磁盘？
 2. 每一个内存缓冲区都被哪些事务修改过？每个事务修改后的结果是什么？

□ 在redo日志中插入(非静止)检查点的步骤

1. 写入日志记录<**Start CKPT(T_1, \dots, T_k)**>, 并刷新日志;
 - 其中: T_1, \dots, T_k 是当前所有活跃事务的标识符
 - 同时获得当时所有已提交事务的标识符集合**S**
2. 将集合**S**中的事务已经写到内存缓冲区但还没有写到数据库磁盘的数据库元素写入磁盘;
3. 写入日志记录<**End CKPT**>并刷新日志
 - 不必等待事务 T_1, \dots, T_k 或新开始事务的结束

图5-3-8 带有非静止检查点的redo日志

redo日志
<Start T ₁ >
<T ₁ , A, 5>
<Start T ₂ >
<Commit T ₁ >
<T ₂ , B, 10>
<Start CKPT(T ₂)>
<T ₂ , C, 15>
<Start T ₃ >
<T ₃ , D, 20>
<End CKPT>
<Commit T ₂ >
<Commit T ₃ >

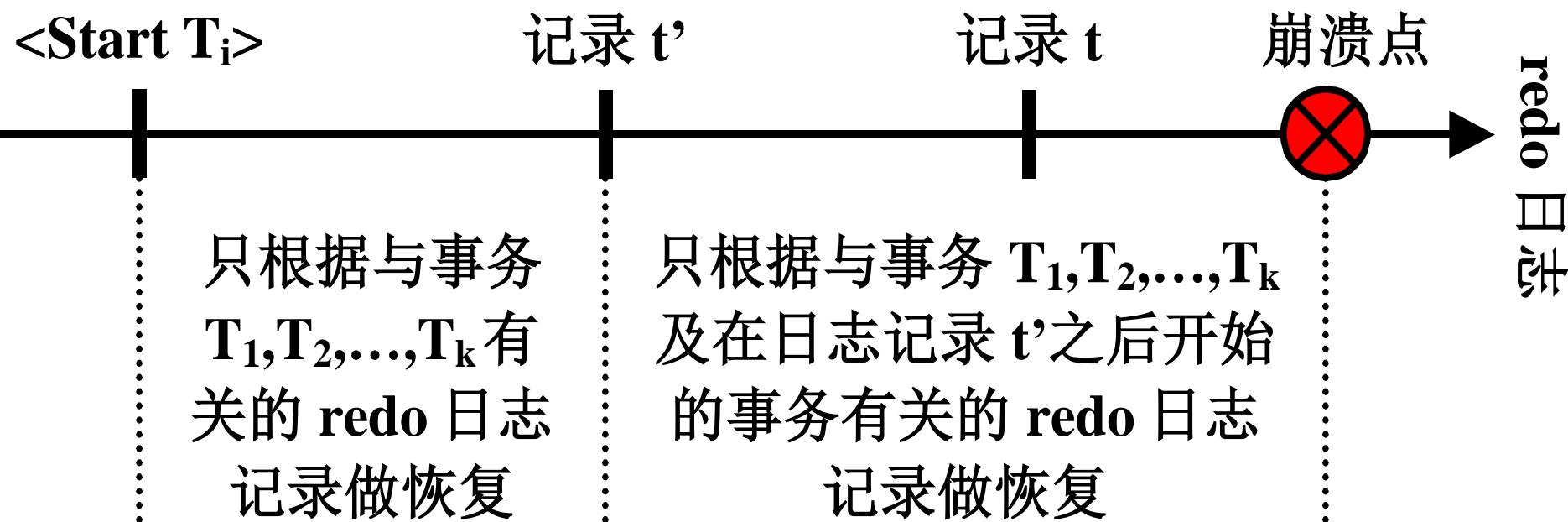
使用带检查点的redo日志的恢复

□ 寻找最后一个被记入日志的检查点记录：

1. <End CKPT> 记录
2. <Start CKPT(T_1, \dots, T_k)> 记录（设为记录t）
 - 继续逆向扫描日志文件，直至最后被记入的<End CKPT> 记录
 - 就如同没有日志记录t，像情况1一样进行恢复

□ 使用带检查点的redo日志的恢复

- 找到最后一个被记入日志的 $\langle \text{End CKPT} \rangle$ (记为记录t)，假设与之相对应的检查点记录是 $\langle \text{Start CKPT}(T_1, \dots, T_k) \rangle$ (记为记录t')，并找到最早出现的 $\langle \text{Start } T_i \rangle$ (记为记录 t_i)
- 故障恢复方法如下：针对事务 T_1, \dots, T_k 以及在 t' 之后开始的那些事务，重做其中已经被提交的事务



例：使用带有检查点的redo日志进行恢复

redo日志
<Start T ₁ >
<T ₁ , A, 5>
<Start T ₂ >
<Commit T ₁ >
<T ₂ , B, 10>
<Start CKPT(T ₂)>
<T ₂ , C, 15>
<Start T ₃ >
<T ₃ , D, 20>
<End CKPT>
<Commit T ₂ >
系统崩溃

从这里开始redo操作

在<Start CKPT(T₂)>
之前已经结束的事务
(T₁)不必处理

重做在<Start CKPT(T₂)>
之后，崩溃发生之前已经
被提交的事务(T₂)

最后为尚未结束事务(T₃)
加入结束标志<Abort T₃>

□ undo与redo日志的不足

- Undo日志要求数据在事务结束后立即写到磁盘，可能增加需要执行磁盘I/O的次数
- Redo日志要求事务提交和日志记录刷新之前将所有修改过的数据保留在内存缓冲区中，可能增加事务需要的平均缓冲区的数量
- 如果被访问的数据对象X不是完整的磁盘块，那么在undo日志与redo日志之间可能产生相互矛盾的请求

□ 可以通过一种被称为undo/redo日志的日志类型来解决上述矛盾

□ undo/redo日志的记录格式

- 与undo日志或redo日志的格式基本一样，区别在于更新记录的格式： $\langle T, X, v, w \rangle$
 - 不仅记录更新前的值v，同时也要记录更新后的newValue
 - 因此该种类型的日志既能用于未结束事务的撤销(undo)，也能用于已提交事务的重做(redo)

□ undo/redo日志的记载规则

UR_1 ：在由于某个事务T所做的改变而修改磁盘上的数据库元素X之前，更新记录 $\langle T, X, v, w \rangle$ 必须出现在磁盘上。

- 其中，v是X被更新前的值，w是X被更新后的值

图5-3-9 undo/redo日志的例子

	T	t	M-A	M-B	D-A	D-B	undo/redo日志
1					8	8	<Start T>
2	Read(A,t)	8	8		8	8	
3	$t := t^2$	16	8		8	8	
4	Write(A,t)	16	16		8	8	<T, A, 8, 16>
5	Read(B,t)	8	16	8	8	8	
6	$t := t^2$	16	16	8	8	8	
7	Write(B,t)	16	16	16	8	8	<T, B, 8, 16>
8	Flush Log						
9	Output(A)		16	16	16	8	
10	Commit						<Commit T>
11	Output(B)		16	16	16	16	

图5-3-9 undo/redo日志的例子

	T	t	M-A	M-B	D-A	D-B	undo/redo日志
1					8	8	<Start T>
2	Read(A,t)	8	8		8	8	
3	$t := t^2$	16	8		8	8	
4	Write(A,t)	16	16		8	8	-T A 8 16-
5	Read(B,t)	8					
6	$t := t^2$	16					
7	Write(B,t)	16	16	16	8	8	<T, B, 8, 16>
8	Flush Log						
9	Output(A)						
10	Commit						
11	Output(B)						

确保事务T的所有更新日志都已经被写入日志文件的磁盘！

事务提交(Commit)和写数据
库磁盘(Output)的操作顺序是
随机的！

□ 第10步也可以出现在第9步之前或第11步之后。

图5-3-9 undo/redo日志的例子

	T	t	M-A	M-B	D-A	D-B	undo/redo日志
1					8	8	<Start T>
2	Read(A,t)	8	8		8	8	
3	$t := t^2$	16	8		8	8	

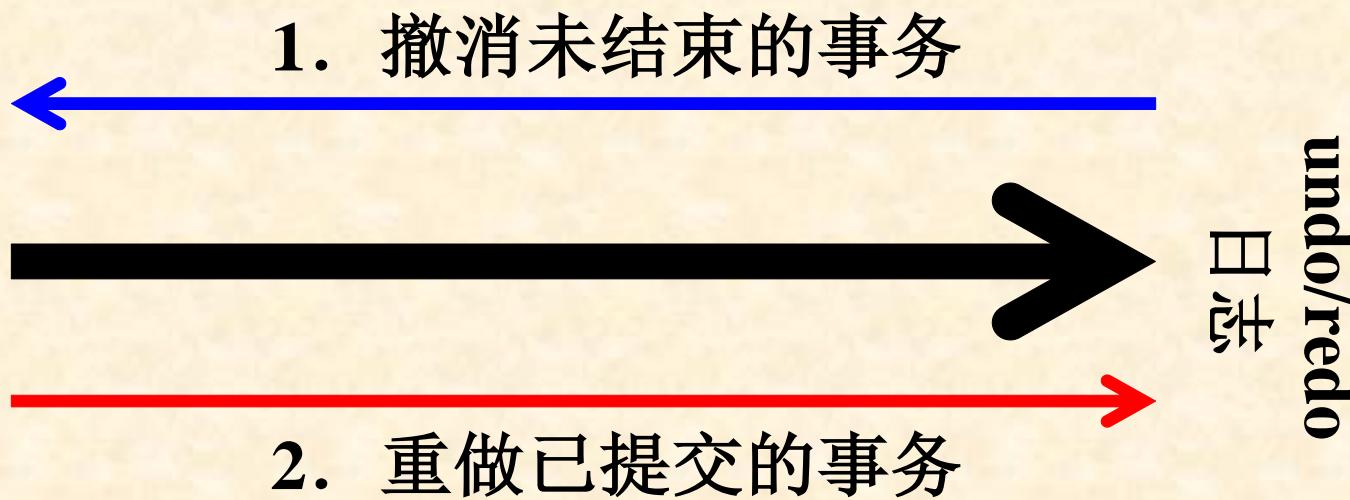
实际上，只要确信更新记录<T, A, 8, 16>已经出现在日志文件的磁盘中

7	Write(B,t)	16	16	16	8	8	<T, B, 8, 16>
8	Flush Log						
9	Output(A)						
10	Commit						
11	Output(B)						

我们还可以在更早的时候执行 Output (A) 操作，而不管当前事务T是否已经被提交

□ 使用undo/redo日志的故障恢复过程

- 根据<Commit T>是否已经出现在磁盘中来决定事务T是否已经被提交
- 具体的恢复过程如下：
 1. 按照从后往前的顺序，撤消(undo)所有未提交的事务
 2. 按照从前往后的顺序，重做(redo)所有已提交的事务



- 为了确保在日志中写入<Commit T>记录的事务T确实被提交，为undo/redo日志新增加了一条记载规则：
 - **UR₂**: 在每一条 <Commit T> 后面必须紧跟一条 Flush Log 操作
- 在undo/redo日志中插入检查点
 1. 写入日志记录<Start CKPT(T₁, ..., T_k)>, 并刷新日志
 - ❖ 其中: T₁, ..., T_k是当前所有活跃事务的标识符
 2. 将所有被修改过的缓冲区写到数据库的磁盘中去
 3. 写入日志记录<End CKPT>并刷新日志

带有检查点的undo/redo日志的例子

	undo/redo日志
1	<Start T ₁ >
2	<T ₁ , A, 4, 5>
3	<Start T ₂ >
4	<Commit T ₁ >
5	<T ₂ , B, 9, 10>
6	<Start CKPT(T ₂)>
7	<T ₂ , C, 14, 15>
8	<Start T ₃ >
9	<T ₃ , D, 19, 20>
10	<End CKPT>
11	<Commit T ₂ >
12	<Commit T ₃ >

口思考题：在下述情况下如何进行数据库的故障恢复？

(哪些事务将被重做，哪些事务将被撤消，从哪一步开始，到哪一步结束)

1. 在第12步之后发生故障
2. 在第11步与第12步之间发生故障
3. 在第10步与第11步之间发生故障
4. 在第9步与第10步之间发生故障

undo/redo日志

1	<Start T ₁ >
2	<T ₁ , A, 4, 5>
3	<Start T ₂ >
4	<Commit T ₁ >
5	<T ₂ , B, 9, 10>
6	<Start CKPT(T ₂)>
7	<T ₂ , C, 14, 15>
8	<Start T ₃ >
9	<T ₃ , D, 19, 20>
10	<End CKPT>
11	<Commit T ₂ >
12	<Commit T ₃ >
	发生故障

1. 在第12步之后发生故障，如何进行数据库的故障恢复？

① 逆向扫描，撤销(undo)所有未提交和未结束的事务()，直至事务T₂的起点

② 正向扫描，重做(redo)所有已提交事务(T₂&T₃)，直至日志的终点

undo/redo日志

1	<Start T ₁ >
2	<T ₁ , A, 4, 5>
3	<Start T ₂ >
4	<Commit T ₁ >
5	<T ₂ , B, 9, 10>
6	<Start CKPT(T ₂)>
7	<T ₂ , C, 14, 15>
8	<Start T ₃ >
9	<T ₃ , D, 19, 20>
10	<End CKPT>
11	<Commit T ₂ >
	<Abort T ₃ >

2. 在第11步与第12步之间发生故障，如何进行数据库的故障恢复？

① 逆向扫描，撤销(undo)所有未提交和未结束的事务(T₃)，直至事务T₂的起点

② 正向扫描，重做(redo)所有已提交事务(T₂)，直至日志的终点

③ 故障恢复结束后，将在日志文件的尾部插入T₃的结束标志

undo/redo日志

1	<Start T ₁ >
2	<T ₁ , A, 4, 5>
3	<Start T ₂ >
4	<Commit T ₁ >
5	<T ₂ , B, 9, 10>
6	<Start CKPT(T ₂)>
7	<T ₂ , C, 14, 15>
8	<Start T ₃ >
9	<T ₃ , D, 19, 20>
10	<End CKPT>
	<Abort T ₂ >
	<Abort T ₃ >

3. 在第10步与第11步之间发生故障，如何进行数据库的故障恢复？

① 逆向扫描，撤销(undo)所有未提交和未结束的事务(T₂&T₃)，直至事务T₂的起点

② 正向扫描，重做(redo)所有已提交事务()，直至日志的终点

③ 故障恢复结束后，将在日志文件的尾部插入T₂和T₃的结束标志

undo/redo日志

1	<Start T ₁ >
2	<T ₁ , A, 4, 5>
3	<Start T ₂ >
4	<Commit T ₁ >
5	<T ₂ , B, 9, 10>
6	<Start CKPT(T ₂)>
7	<T ₂ , C, 14, 15>
8	<Start T ₃ >
9	<T ₃ , D, 19, 20>
	<Abort T ₂ >
	<Abort T ₃ >
	<End CKPT>

4. 在第9步与第10步之间发生故障，如何进行数

① 逆向扫描，撤销所有未提交和未结束的事务($T_2 \& T_3$)，直至日志的起点或前一个Start CKPT

② 正向扫描，重做所有已提交事务(T_1)

③ 故障恢复结束后，在日志文件的尾部插入 T_2 和 T_3 的结束标志

④ 最后再插入<End CKPT>日志记录

5.3.3 数据库故障恢复三大技术

1. 事务的撤消（undo）

- 反向扫描日志文件，查找应该撤消的事务
- 查找这些事务的更新操作
- 对更新操作做逆操作
 - 插入操作：删除被插入的元组
 - 删除操作：将被删除的元组重新插入
 - 修改操作：用修改前的值替代修改后的值
- 如此反向扫描直到日志文件的头部

5.3.3 数据库故障恢复三大技术

2. 事务的重做 (redo)

- 正向扫描日志文件，查找应该重做的事务
- 查找这些事务的更新操作
- 对更新操作作重做处理
 - 插入操作：重新插入新元组
 - 删除操作：重新删除旧的元组
 - 修改操作：用修改后的值替代修改前的值
- 如此正向扫描直到日志文件的尾部

5.3.4 恢复策略

□ 小型故障的恢复

➤ 利用未结束事务的undo操作进行恢复

□ 中型故障的恢复

➤ 有两类事务需要恢复

a) 非正常中止的事务

❖ 执行undo操作

b) 已完成提交的事务

❖ 其更新操作的修改结果还留在内存缓冲区
中，尚未来得及写入磁盘，由于故障使内
存缓冲区中的数据被丢失

❖ 执行redo操作

5.3.4 恢复策略

□ 大型故障的恢复

- 先利用后备副本进行数据库恢复，再利用日志进行数据库的恢复。具体步骤如下
 1. 将后备副本中的数据拷贝到数据库中
 2. 检查日志文件：确定哪些事务已经执行结束，哪些尚未结束
 3. 按照日志的记载顺序
 - a) 逆向：对尚未结束的事务作撤销处理(**undo**)
 - b) 正向：对已经结束的事务作重做处理(**redo**)

5.3.5 数据库镜像

□ 数据库镜像

- 将整个数据库中的数据（或主要数据）实时复制到另一个磁盘中
 - 也称为‘磁盘镜像’
 - 系统自动保证主数据库服务器中的数据与数据库镜像中的数据的一致性
 - 在数据库中的数据遭受破坏后，可以利用数据库镜像中的数据进行修补

【例】银行的转帐业务：从银行存款帐号A（当前金额10000元）转5000元到银行存款帐号B（当前金额20000元）

	转帐事务	内存变量X	帐号A	帐号B
		—	10000	20000
1	READ(A) to X;	10000	10000	20000
	IF (X ≥ 5000) THEN BEGIN	10000	10000	20000
	X := X - 5000;	5000	10000	20000
2	WRITE(X) to A;	5000	5000	20000
3	READ(B) to X;	20000	5000	20000
	X := X + 5000;	25000	5000	20000
4	WRITE(X) to B;	25000	5000	25000
	END	—	5000	25000





构造一个调度的事务优先图

□ 调度S:

- $r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

□ 第一步：从调度S中找出所有与数据对象A有关的操作

$r_2(A); w_2(A); r_3(A); w_3(A);$

- 根据冲突 ($w_2(A);r_3(A)$) 可知：存在从事务2到事务3的一条有向边

□ 第二步：从调度S中找出所有与数据对象B有关的操作

$r_1(B); w_1(B); r_2(B); w_2(B)$

- 根据冲突 ($w_1(B);r_2(B)$) 可知：存在从事务1到事务2的一条有向边



构造一个调度的事务优先图

□ 调度S:

- $r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

□ 第一步：从调度S中找出所有与数据对象A有关的操作

$r_2(A); w_2(A); r_3(A); w_3(A);$

- 根据冲突 ($w_2(A);r_3(A)$) 可知：存在从事务2到事务3的一条有向边

□ 第二步：从调度S中找出所有与数据对象B有关的操作

$r_1(B); r_2(B); w_1(B); w_2(B)$

- 根据冲突 ($r_2(B);w_1(B)$) 可知：存在从事务2到事务1的一条有向边

- 根据冲突 ($w_1(B);w_2(B)$) 可知：存在从事务1到事务2的一条有向边

没有并发控制而产生的‘丢失修改’现象

步骤	甲售票点	变量 t1	乙售票点	变量 t2	剩余 机票X
1	Read(X, t1);	5			5
2	t1:=t1 - 1;	4			
3			Read(X, t2);	5	
4			t1:=t2 - 1;	4	
5	Write(X, t1);				4
6			Write(X, t2);		4

使用一级封锁协议构成新的操作序列

步骤	甲售票点	变量 t1	乙售票点	变量 t2	剩余 机票X
	write_lock(X);				
1	Read(X, t1) ;	5			5
2	t1:=t1 - 1;	4			
			write_lock(X);		
3			Read(X, t2) ;	5	
4			t1:=t2 - 1;	4	
5	Write(X, t1) ;				4
	Commit;				
	unlock(X);				
6			Write(X, t2) ;		4
			Commit;		
			unlock(X);		



实际的调度结果，可以避免出现‘丢失修改’现象

步骤	甲售票点	变量 t1	乙售票点	变量 t2	剩余 机票X
	write_lock(X);				
1	Read(X, t1) ;	5			5
2	t1:=t1 - 1;	4			
			write_lock(X);		
3	Write(X, t1) ; Commit;		wait		4
	unlock(X);		wait		
4			Read(X, t2) ;	4	
5			t1:=t2 - 1;	3	
6			Write(X, t2) ; Commit;	3	
			unlock(X);		

	Read 操作		Write 操作	
	锁类型	封锁时间	锁类型	封锁时间
READUNCOMMITTED 未提交读	No Lock	—	不允许执行 Write 操作	
READCOMMITTED 提交读	共享锁	读操作		
READREPEATABLE 可重复读	共享锁	事务	排它锁	事务
SERIALIZABLE 可串行化	共享锁	事务		

事务的隔离级别与封锁策略之间的关系



总结

- 事务
- 事务的性质
- 事务的活动状态
- 与事务有关的控制语句
 - 事务的开始
 - 事务的结束（COMMIT/ROLLBACK）
 - 事务的运行方式
 - SET AUTO COMMIT ON|OFF
 - SET TRANSACTION READONLY|READWRITE
 - SET TRANSACTION ISOLATION LEVEL

- 并发控制
- 事务及其调度的表示方法
- 事务的并发运行方式
 - 串行调度
 - 可串行化调度
 - 冲突可串行化
 - 冲突
 - 冲突等价
 - 优先图
 - 冲突可串行化的判断方法

□ 三种数据不一致性

□ 封锁 (lock)

- 常用的两种类型封锁: **S锁 / X锁**
- 锁相容矩阵
- 申请封锁和释放封锁的处理过程

□ 封锁协议

- 三级封锁协议
- 两阶段封锁协议
- 合法调度

□ 多粒度封锁

- 封锁粒度
- 意向锁及其锁相容矩阵
- 多粒度封锁协议

□ 死锁及其常用的预防方法

□ 活锁及其预防方法

□ 数据库恢复技术

- 常见故障及其恢复方法
- 数据转储：静态/动态，海量/增量
- 日志：组成内容，作用，书写原则
- **Undo**日志：纪录格式，记载规则

