数 据 库 原 理

Theory of Database

李静

信息科学与技术学院

问题背景

- 例：飞机订票系统有两个数据表Sale和Flight，分别记录各售票点的售票数及全部航班的剩余票数：

  Sale(agentNo, flightNo, date, saledNumber)

  Flight(flightNo, date, remainNumber)

- 现有A0010售票点欲出售F007航班2021年5月18日机票2张。

问题背景

- 可编制如下程序：

查询F007航班2021年5月18日剩余票数A; } (1)

if *(A<2)*

拒绝操作，并通知票源不足；

else

更新A0010售票点的售票数；

更新F007航班的剩余票数；

(2)

如何用SQL语句分别实现语句(1)和语句(2)？

```
SELECT remainNumber
FROM Flight
WHERE flightNo='F007' AND date='2021-05-18'
```

问题背景

- 语句(2)是在当F007航班2021年5月18日的剩余票数大于请求票数时更新Sale和Flight表。该更新包括两个update操作：

```
UPDATE Sale
SET saledNumber=saledNumber+2
WHERE  agentNo='A0010' AND flightNo='F007
      AND date='2021-05-18'
UPDATE Flight
SET remainNumber=remainNumber–2
WHERE flightNo='F007' AND date='2021-05-18'
```

如果第一个UPDATE语句执行成功，而第二个UPDATE语句执行失败，会发生什么问题呢？？？

问题背景

- 假设F007航班共有200个座位，2021年5月18日机票已售出198张(其中被A0010售出20张)，余票2张。
- 当第1个UPDATE语句执行成功时，即A0010已售票数更新为22。当系统发生故障重新提供服务时，如果又有售票点请求出售F005航班2021年5月18日机票2张，由于F005的剩余票数未更新(仍为2)，因此满足其要求又出售了2张。

结果多卖了2张票！

出现上述问题的原因是什么？

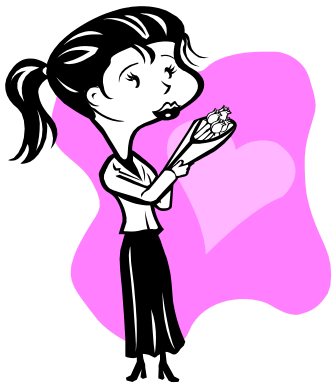
出现故障后，系统重新提供服务时数据库状态与现实世界状态出现了不一致。

问题背景

- 为解决上述问题，数据库管理系统引入了事务概念，它将这些有内在联系的操作当作一个逻辑单元看待，并采取相应策略保证一个逻辑单元内的全部操作要么都执行成功，要么都不执行。
- 对数据库用户而言，只需将具有完整逻辑意义的一组操作正确地定义在一个事务之内 即可。

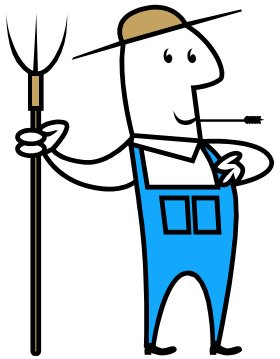
第7章 数据库保护

- 7.1 事务
- 7.2 并发控制
- 7.3 数据库备份与恢复



7.1 事务

- 7.1.1 事务的基本概念
- 7.1.2 事务的特征
- 7.1.3 SQL事务处理模型



7.1.1 事务

- 事务是用户定义的数据操作系列，这些操作作为一个完整的工作单元，一个事务内的所有语句被作为一个整体，要么全部执行，要么全部不执行。
- 例如：

对于一个转帐活动：A帐户转帐给B帐户n元钱，这个活动包含两个动作：

- 第一个动作：A帐户 - n
- 第二个动作：B帐户 + n

事务结束语句

- 事务结束的两种类型：
  - 事务提交(commit)：将成功完成事务的执行结果(即更新)永久化，并释放事务占有的全部资源。
  - 事务回滚(rollback)：中止当前事务、撤销其对数据库所做的更新，并释放事务占有的全部资源。

7.1.2 事务的特征

- **原子性（Atomicity）：**

事务是数据库的逻辑工作单位，事务中的操作要么都做，要么都不做。

- **一致性（Consistency）：**

事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。

- **隔离性（Isolation）：**

数据库中一个事务的执行不能被其它事务干扰。

- **持久性（ Durability ）：**

事务一旦提交，对数据库数据的改变是永久的。

- 保证事务的ACID特性是事务处理的重要任务。
- 事务的ACID特性可能遭到破坏的因素有两种：
  - 多个事务并行运行时，不同事务的操作有交叉情况；
  - 事务在运行过程中被强迫停止。

7.1.3 SQL事务处理模型

- **隐式事务：**

每一条数据操作语句都自动成为一个事务。

- **显式事务：**

有显式的开始和结束标记的事务。

两种处理方式：

ISO事务处理模型

T-SQL事务处理模型

ISO事务处理模型

- 明尾暗头：事务的开头是隐含的，事务的结束有明确标记

A．事务结束符

COMMIT：事务成功结束符

ROLLBACK：事务失败结束符

B．事务提交方式

自动提交：每条SQL语句为一个事务

指定位置提交：在事务结束符或程序正常结束处提交

C．事务起始/终止位置

程序的首条SQL语句或事务结束符后的语句。

在程序正常结束处或COMMIT语句处成功终止；

在程序出错处或ROLLBACK处失败终止。

示例

```
UPDATE 支付表
    SET 帐户总额 = 帐户总额 - n
        WHERE 帐户名 = ‘A’
UPDATE 支付表
    SET 帐户总额 = 帐户总额 + n
        WHERE 帐户名 = ‘B’
COMMIT
```

T-SQL事务处理模型

- 每个事务都有显式的开始和结束标记。
- 事务的开始标记是：

  BEGIN TRANSACTION | TRAN
- 事务的结束标记为：

  COMMIT ［TRANSACTION｜TRAN］

  ROLLBACK ［TRANSACTION｜TRAN］

示例

- Transact-SQL事务处理模型描述：

```
BEGIN TRANSACTION
UPDATE 支付表
   SET 帐户总额 = 帐户总额 - n
    WHERE 帐户名 = ‘A’
UPDATE 支付表
   SET 帐户总额 = 帐户总额 + n
    WHERE 帐户名 = ‘B’
COMMIT
```

7.2 并发控制

- 7.2.1 并发控制概述
- 7.2.2 并发控制措施
- 7.2.3 封锁协议
- 7.2.4 死锁
- 7.2.5 并发调度的可串行性
- 7.2.6 两段锁协议

7.2.1 并发控制概述

- 多用户数据库系统的存在：

  允许多个用户同时使用的数据库系统.

  ■银行数据库系统

  ■选课系统

  特点：同一时刻并发运行的事务数可达数百个。

  问题：有可能产生相互干扰。

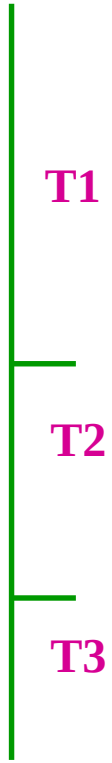
7.2.1 并发控制概述

- 数据库管理系统允许多个事务并发执行
  - 优点
    - 增加系统吞吐量(throughput)。吞吐量是指单位时间系统完成事务的数量。当一事务需等待磁盘I/O时，CPU可去处理其它正在等待CPU的事务。这样，可减少CPU和磁盘空闲时间，增加给定时间内完成事务的数量。
    - 减少平均响应时间(average response time)。事务响应时间是指事务从提交给系统到最后完成所需要的时间。缩短短事务的等待时间。
  - 缺点
    - 若不对事务的并发执行加以控制，则可能破坏数据库的一致性。

不同的多事务执行方式

串行执行

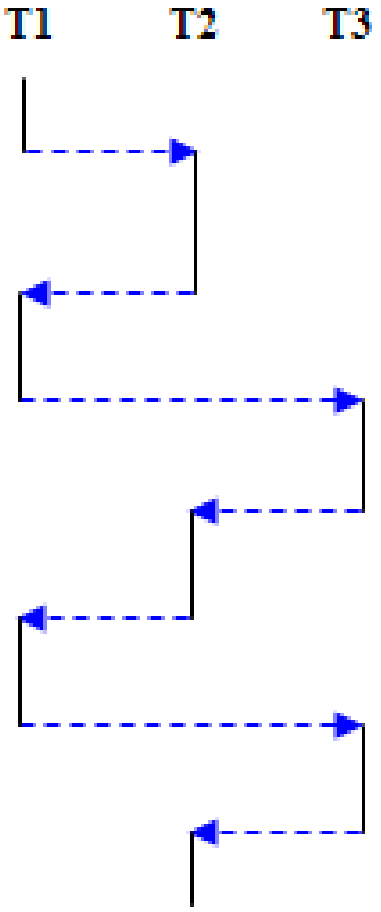
- 每个时刻只有一个事务运行，其他事务必须等到这个事务结束以后方能运行。
- 问题：不能充分利用系统资源，发挥数据库共享资源的特点。



不同的多事务执行方式

交叉并行执行

- 在单处理机系统中，事务的并行执行是这些并行事务的并行操作轮流交叉运行。
- 单处理机系统中的并行事务并没有真正地并行运行，但能够减少处理机的空闲时间，提高系统的效率。



不同的多事务执行方式

- 同时并发方式
  - 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。
- 本章主要讨论单处理机环境下的并发控制技术。

并发事务的相互干扰示例

例： 飞机订票系统中的一个活动序列 ：

① 甲售票点(甲事务)读出某航班的机票余额A，设A=16；

② 乙售票点(乙事务)读出同一航班的机票余额A，也为16；

③ 甲售票点卖出一张票，修改余额A←A-1，所以A为15，写回数据库；

④ 乙售票点也卖出一张票，修改余额A←A-1，所以A为15，写回数据库 。

结果明明卖出两张机票，数据库中机票余额只减少1 。

原因：第4步中乙事务修改A并写回后覆盖了甲事务的修改。

事务并发执行带来的问题

- 多个事务同时存取同一数据的情况。
- 可能会存取和存储不正确的数据。
- 破坏事务一致性和数据库的一致性。

并发控制是衡量DBMS性能的重要标志之一。

并发操作带来的数据不一致性

- 并发操作带来的数据不一致情况：
  - 丢失更新（Lost Update）
  - 读“脏”数据（Dirty Read）
  - 不可重复读（Non-repeatable Read）
- 记号
  - **R(x)**：读数据x
  - **W(x)**：写数据x

1. 丢失更新

T_1	T_2
① R(A)=16	
②	R(A)=16
③ A←A-1	
W(A)=15	
④	A←A-1
	W(A)=15

两个事务T1和T2读入同一数据并修改，T2的提交结果破坏了T1提交的结果，导致T1的修改被丢失。

2. 不可重复读

T_1	T_2
① R(A)=50	
R(B)=100	
求和=150	
②	R(B)=100
	B←B*2
	(B)=200
③ R(A)=50	
R(B)=200	
和=250	
(验算不对)	

- T1读取B=100进行运算。
- T2读取同一数据B，对其进行修改后将B=200写回数据库。
- T1为了对读取值校对重读B，B已为200，与第一次读取值不一致。

不可重复读的三种情况

(1) 事务T_1读取某一数据后，事务T_2对其做了修改，当事务T_1再次读该数据时，得到与前一次不同的值 。

(2) 事务T1按一定条件从数据库中读取了某些数据记录后，事务T2删除了其中部分记录，当T1再次按相同条件读取数据时，发现某些记录消失了 。

(3) 事务T1按一定条件从数据库中读取某些数据记录后，事务T2插入了一些记录，当T1再次按相同条件读取数据时，发现多了一些记录。

后两种不可重复读有时也称为幽灵/幻影现象。

3. 读“脏”数据

T_1	T_2
① R(C)=100	
C←C*2	
W(C)=200	
②	R(C)=200
③ROLLBACK	
C恢复为100	

- T1将C修改为200，T2读到C为200。
- T1由于某种原因撤销，其修改作废，C恢复原值100。
- 这时T2读到的C为200，与数据库内容不一致，就是“脏”数据 。

并发控制机制的任务

- 数据不一致性原因：

  并发操作破坏了事务的隔离性。

- 并发控制就是要用正确的方式调度并发操作，使一个用户事务的执行不受其他事务的干扰，从而避免造成数据的不一致性。

7.2.2 并发控制措施

- 控制目标：

  事务运行过程中尽可能隔离事务外操作对本事务数据环境的影响。

- 并发控制的主要技术：

  加锁(Locking)、时间戳、乐观控制法

什么是加锁

- **加锁：**

事务T在对某个数据对象（例如表、记录等）操作之前，先向系统发出请求，对其加锁。加锁后事务T就对该数据对象有了一定的控制，在事务T释放锁前，其它事务不能更新此数据对象。

基本封锁类型

- 一个事务对某个数据对象加锁后究竟拥有什么样的控制权由封锁的类型决定。
- 基本封锁类型
  - 排它锁（Exclusive Locks，简记为X锁）
  - 共享锁（Share Locks，简记为S锁）

排它锁

- 排它锁又称为写锁（ X锁）。
- 若事务T对数据对象加上X锁，则允许事务T读取和修改数据，其它任何事务都不能再对此数据加任何类型的锁，直到事务T释放X锁。
- 保证其他事务在事务T释放X锁之前不能再读取和修改数据。

共享锁

- 共享锁又称为读锁（ S锁） 。
- 若事务T对数据对象加上S锁，则其它事务只能再对数据加S锁，而不能加X锁，直到事务T释放数据上的S锁。
- 保证其他事务可以读数据 ，但是在事务T释放S锁前不能对数据做任何修改 。

锁的相容矩阵

T_1 \ T_2	X	S	无锁
X	N	N	Y
S	N	Y	Y
无锁	Y	Y	Y

使用封锁机制解决丢失修改问题

例：

T_1	T_2
① Xlock A	
② R(A)=16	
	Xlock A
③ A←A-1	等待
W(A)=15	等待
Commit	等待
Unlock A	等待
④	获得Xlock A
	R(A)=15
	A←A-1
⑤	W(A)=14
	Commit
	Unlock A

- 事务T1在读A进行修改之前先对A加X锁
- 当T2再请求对A加X锁时被拒绝
- T2只能等待T1释放A上的锁后T2获得对A的X锁
- 这时T2读到的A已经是T1更新过的值15
- T2按此新的A值进行运算，并将结果值A=14送回到磁盘。避免了丢失T1的更新。

解决不可重复读问题

- T1读A，B前，先对A,B加S锁。
- 其他事务只能再对A，B加S锁，而不能加X锁，即其他事务只能读A，B，而不能修改。
- 当T2为修改B而申请B的X锁时被拒绝只能等T1释放B上的锁。
- T1为验算再读A，B，这时读出的B仍是100，求和结果仍为150，即可重复读。
- T1结束才释放A，B上的S锁。T2才获得对B的X锁 。

T_1	T_2
① Slock A　Slock B	
R(A)=50　R(B)=100	
求和=150	
②	Xlock B
	等待
③ R(A)=50　(B)=100	等待
求和=150	等待
Commit	等待
Unlock A　Unlock B	等待
④	获得XlockB
	R(B)=100
	B←B*2
⑤	W(B)=200
	Commit
	Unlock B

例 使用封锁机制解决读“脏”数据问题

T_1	T_2
① Xlock C	
R(C)=100	
C←C*2	
W(C)=200	
②	Slock C
	等待
③ ROLLBACK	等待
(C恢复为100)	等待
Unlock C	等待
④	获得Slock C
	R(C)=100
⑤	Commit C
	Unlock C

- T1在对C进行修改前，先对C加X锁，修改后写回磁盘，
- T2请求C上加S锁，因T1已在C上加了X锁，T2只能等待，
- T1因某种原因被撤销，C恢复为原值100，
- T1释放C上的X锁后T2获得C上的S锁，读C=100。避免了T2读“脏”数据。

7.2.3 封锁协议

- **封锁协议（加锁协议）：**

  在运用X锁和S锁对数据对象进行加锁时，还需要约定一些规则，如：何时申请X锁或S锁、持锁时间、何时释放锁等。

- 对封锁方式规定不同的规则，就形成了各种不同级别的封锁协议。
- 不同级别的封锁协议达到的系统一致性级别不同。

一级封锁协议

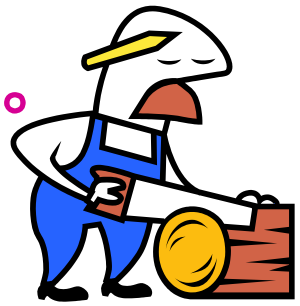
- 规定：

  对事务T要修改的数据加X锁，直到事务结束（包括正常结束和非正常结束）时才释放。

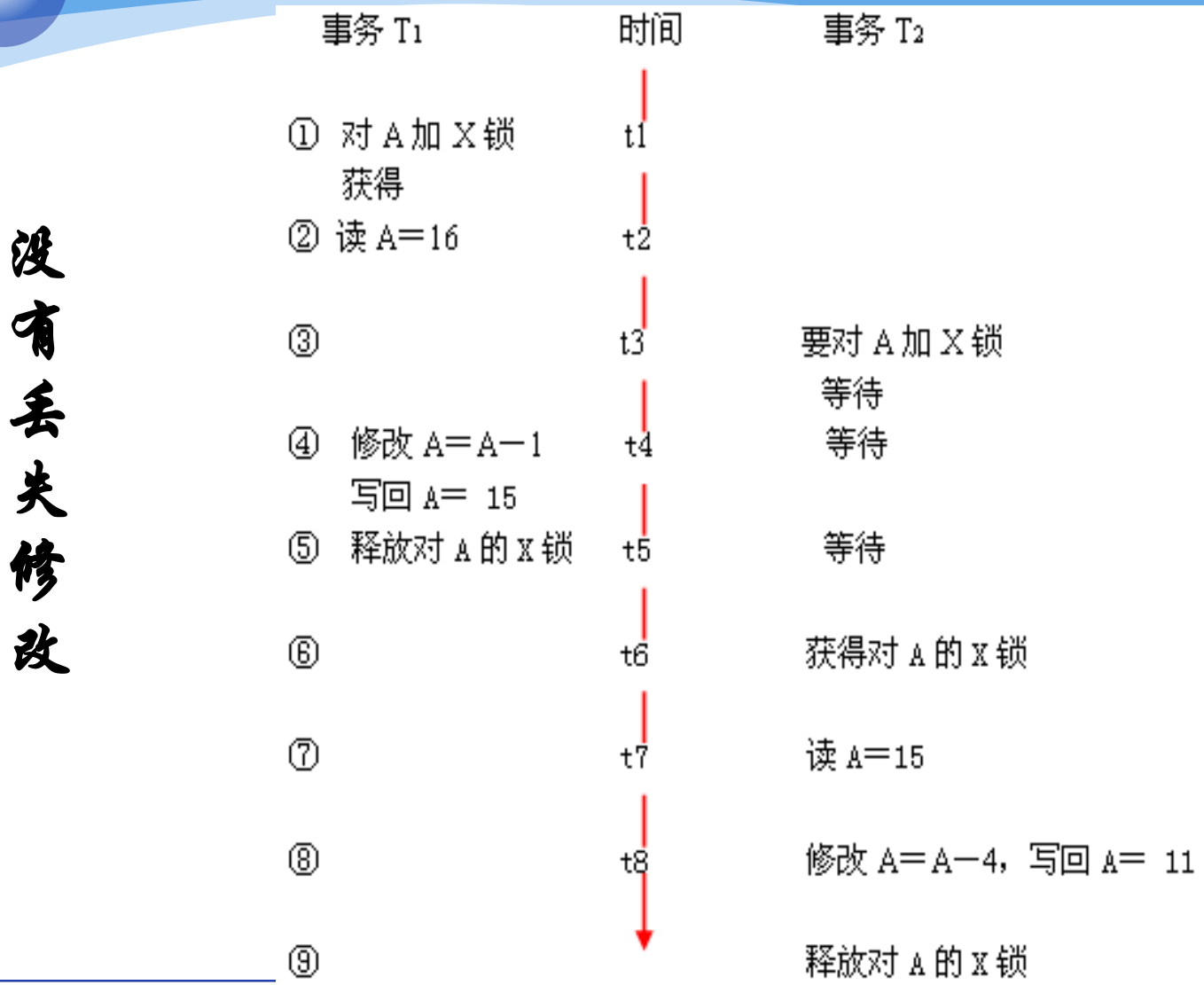
- 效果：

  可以防止丢失修改

  不能防止可重复读和不读“脏”数据。



一级封锁协议示例



没有丢失修改

二级封锁协议

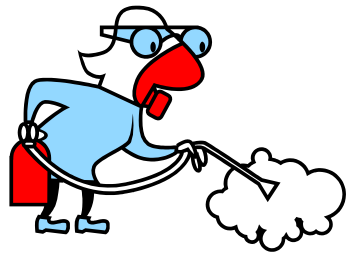
- 规定：

  一级封锁协议加上对事务T对要读取的数据加S锁，读完后即释放S锁。

- 效果：

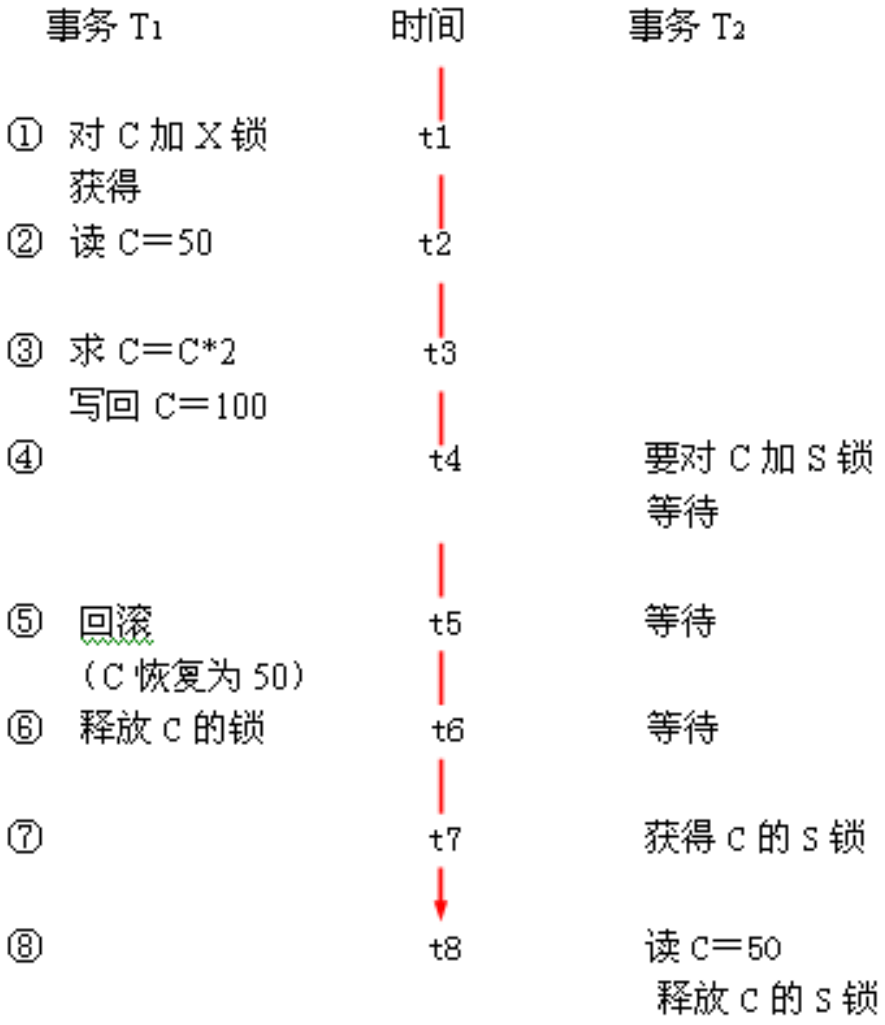
  可以防止丢失修改、防止读“脏”数据。

  不能防止可重复读数据。



二级封锁协议示例

没有读脏数据



三级封锁协议

- 规定

  一级封锁协议加上事务T对要读取的数据加S锁，并直到事务结束才释放。

- 效果：

  可以防止丢失修改、防止读“脏”数据、防止不可重复读。

三级封锁协议示例

可重复读

事务 T_1	时间	事务 T_2
① 对 A、B 分别加 S 锁 获得	t1	
② 读 A=50，B= 100 求 A+B= 150	t2	
③	t3	要对 B 加 X 锁 等待
④ 读 A=50，B= 100 求 A+B= 150	t4	等待
⑤ 将和值写回到数据库中	t5	等待
⑥ 释放 A 的锁 释放 B 的锁	t6	等待
⑦	t7	获得 B 的 X 锁
⑧	t8	读 B=100 修改 B=B*2，写回 B=200
⑨	t9	释放对 B 的 X 锁

不同级别的封锁协议总结

封锁协议	X锁（对写数据）	S锁（对只读数据）	不丢失修改（写）	不读脏数据（读）	可重复读（读）
一级	事务全程加锁	不加	√		
二级	事务全程加锁	事务开始加锁，读完即释放锁	√	√	
三级	事务全程加锁	事务全程加锁	√	√	√

7.2.4 活锁和死锁

- 活锁：

T_1	T_2	T_3	T_4
lock R	.	.	.
.	lock R	.	.
.	等待	Lock R	.
Unlock	等待	.	Lock R
.	等待	Lock R	等待
.	等待	.	等待
.	等待	Unlock	等待
.	等待	.	Lock R
.	等待	.	.

活锁（续）

- 避免活锁：

  采用先来先服务的策略。

  - 当多个事务请求封锁同一数据对象时。
  - 按请求封锁的先后次序对这些事务排队。
  - 该数据对象上的锁一旦释放，首先批准申请队列中第一个事务获得锁。

死锁

T_1	T_2
lock R_1	•
•	Lock R_2
•	•
Lock R_2.	•
等待	•
等待	Lock R_1
等待	等待
等待	等待
	•

解决方法：

1. 死锁的预防
2. 死锁的诊断与解除

1. 死锁的预防

- 产生死锁的原因：

  两个或多个事务都已封锁了数据对象，又都请求对已被其他事务封锁的数据对象加锁，从而出现死等待。

- 预防死锁的发生就是要破坏产生死锁的条件。

- 预防死锁的方法

  一次封锁法、顺序封锁法。

(1) 一次封锁法

- 要求每个事务必须一次将所有要使用的数据全部加锁，否则就不能继续执行。
- 存在的问题：
  - 降低系统并发度；
  - 难于事先精确确定封锁对象。

(2) 顺序封锁法

- 顺序封锁法是预先对数据对象规定一个封锁顺序，所有事务都按这个顺序实行封锁。
- 存在的问题
  - 维护成本大：

    数据库系统中封锁的数据对象多，且在不断地变化。
  - 难以实现：

    很难事先确定每一个事务要封锁哪些对象。

死锁的预防（续）

- 结论
  - 在操作系统中广为采用的预防死锁的策略并不很适合数据库的特点。
  - **DBMS在解决死锁的问题上更普遍采用的是诊断并解除死锁的方法。**

2. 死锁的诊断与解除

- 死锁的诊断方式：
  - 超时法
  - 事务等待图法

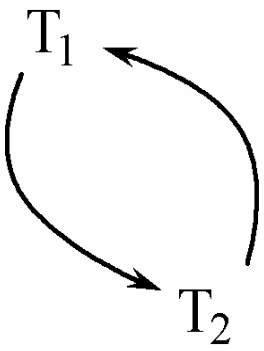
（1）超时法

- 如果一个事务的等待时间超过了规定的时限，就认为发生了死锁。
- 优点：实现简单。
- 缺点：
  - 时限设置过短，有可能误判死锁；
  - 时限设置太长，死锁发生后不能及时发现。

(2) 等待图法

- 用事务等待图动态反映所有事务的等待情况：
  - 事务等待图是一个有向图，
  - 每个结点表示正运行的事务，
  - 每条边表示事务等待的情况，
  - 若T_1等待T_2，则从T_1指向T_2划一条有向边。
  - 图中存在回路，则表示系统中出现了死锁。

等待图法（续）



(a)

- 图(a)中，事务T1等待T2，T2等待T1，产生了死锁。
- 图(b)中，事务T1等待T2，T2等待T3，T3等待T4，T4又等待T1，产生了死锁；
- 图(b)中，事务T3可能还等待T2，在大回路中又有小回路。

死锁的诊断与解除（续）

- 诊断死锁：

并发控制子系统周期性地生成事务等待图，检测事务。如果发现图中存在回路，则表示系统中出现了死锁。

- 解除死锁：
  - 选择一个处理死锁代价最小的事务，将其撤消；
  - 释放此事务持有的所有锁，使其它事务能继续运行。

7.2.5 并发调度的可串行性

- 可串行化调度：
  - 多个事务的并发执行是正确的，当且仅当其结果与按某一次序串行地执行这些事务时的结果相同。
- 可串行性：
  - 并发事务正确调度的准则。
  - 一个给定的并发调度，当且仅当它是可串行化的，才认为是正确调度 。

可串行化调度（续）

[例] 现在有两个事务，分别包含下列操作：

- 事务T1：读B；A=B+1；写回A.
- 事务T2：读A；B=A+1；写回B.

设A,B初始值均为2。

串行调度：
- T1,T2 结果：A= 3 B=4。
- T2,T1 结果：A= 4 B=3。

不可串行化调度

- 执行结果与两种串行调度的结果都不同。
- 是错误的调度 。

T_1	T_2
Slock B	
Y=R(B)=2	
	Slock A
	X=R(A)=2
Unlock B	
	Unlock A
Xlock A	
A=Y+1=3	
W(A)	
	Xlock B
	B=X+1=3
	W(B)
Unlock A	
	Unlock B

可串行化调度

- 执行结果与串行调度T1,T2的执行结果相同.
- 是正确的调度 .

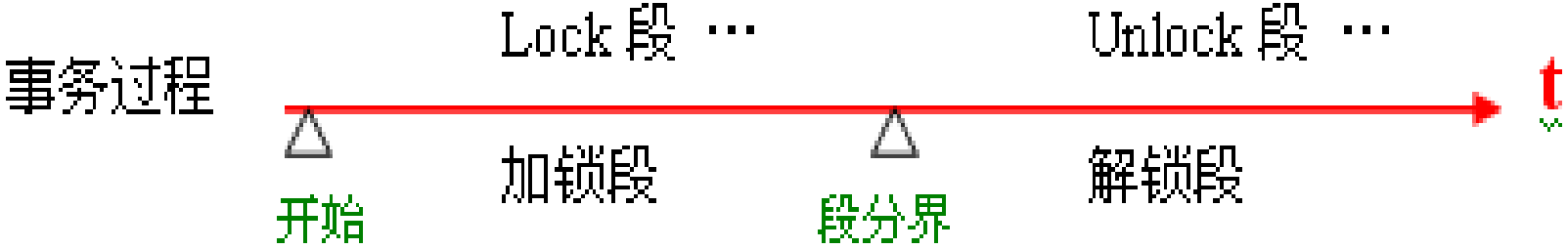
T_1	T_2
Slock B	
Y=R(B)=2	
Unlock B	
Xlock A	
	Slock A
A=Y+1=3	等待
W(A)	等待
Unlock A	等待
	X=R(A)=3
	Unlock A
	Xlock B
	B=X+1=4
	W(B)
	Unlock B

7.2.6两段锁协议

- 两段锁协议：

指所有事务必须分两个阶段对数据项加锁和解锁。

- 第一阶段是获得封锁，也称为扩展阶段：
  - 事务可以申请获得任何数据项上的任何类型的锁，但是不能释放任何锁 。
- 第二阶段是释放封锁，也称为收缩阶段：
  - 事务可以释放任何数据项上的任何类型的锁，但是不能再申请任何锁 。



两段锁协议（续）

例如：

事务T1:

Slock A Slock B Xlock C Unlock B Unlock A Unlock C；

|←---扩展阶段------→| |←----- 收缩阶段 ------→|

事务T1遵守两段锁协议。

事务T2:

Slock A Unlock A Slock B Xlock C Unlock C Unlock B；事务T2不遵守两段锁协议。

调度范例

- 调度遵守两段锁协议的，

  因此是可串行化调度。

事务T_1	事务T_2
Slock(A)	
R(A=260)	
	Slock(C)
	R(C=300)
Xlock(A) W(A=160)	
	Xlock(C)
	W(C=250)
	Slock(A)
Slock(B) R(B=1000)	等待
Xlock(B) W(B=1100)	等待
Unlock(A)	等待
	R(A=160)
	Xlock(A)
Unlock(B)	
	W(A=210)
	Unlock(C)

两段锁协议（续）

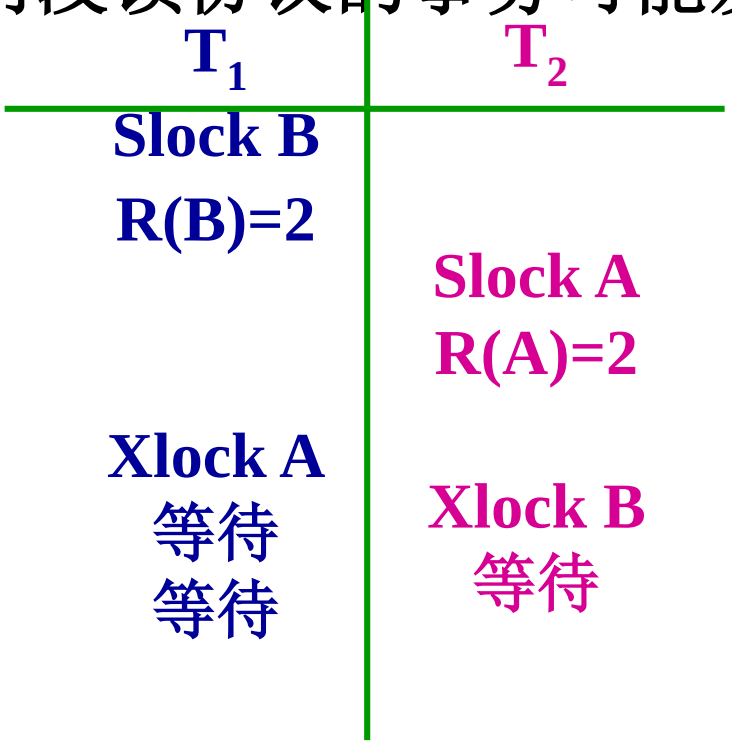
- 事务遵守两段锁协议是可串行化调度。
- 可串行化调度不一定都符合两段锁协议 。

两段锁协议（续）

- 两段锁协议与防止死锁的一次封锁法：
  - 一次封锁法要求每个事务必须一次将所有要使用的数据全部加锁，因此一次封锁法遵守两段锁协议。
  - 两段锁协议并不要求事务必须一次将所有使用的数据全部加锁，因此遵守两段锁协议的事务可能发生死锁。

两段锁协议（续）

[例] 遵守两段锁协议的事务可能发生死锁



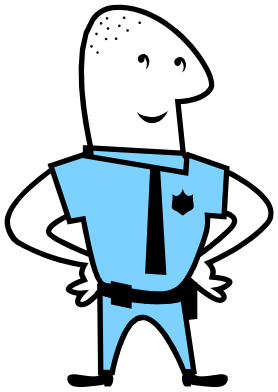
复 习

- 事务、ACID特性、并发带来的问题。
- 数据库的并发控制通常使用封锁机制：
  - 封锁种类（排它锁，共享锁）
  - 锁的相容矩阵。
- 封锁协议
- 两种状况：
  - 活锁： 先来先服务
  - 死锁：
  - 并发调度的可串行性
  - 两段锁协议
  - 封锁粒度

预防方法：一次封锁法、顺序封锁法.
死锁的诊断与解除：超时法、等待图法.

7.3 数据库备份与恢复

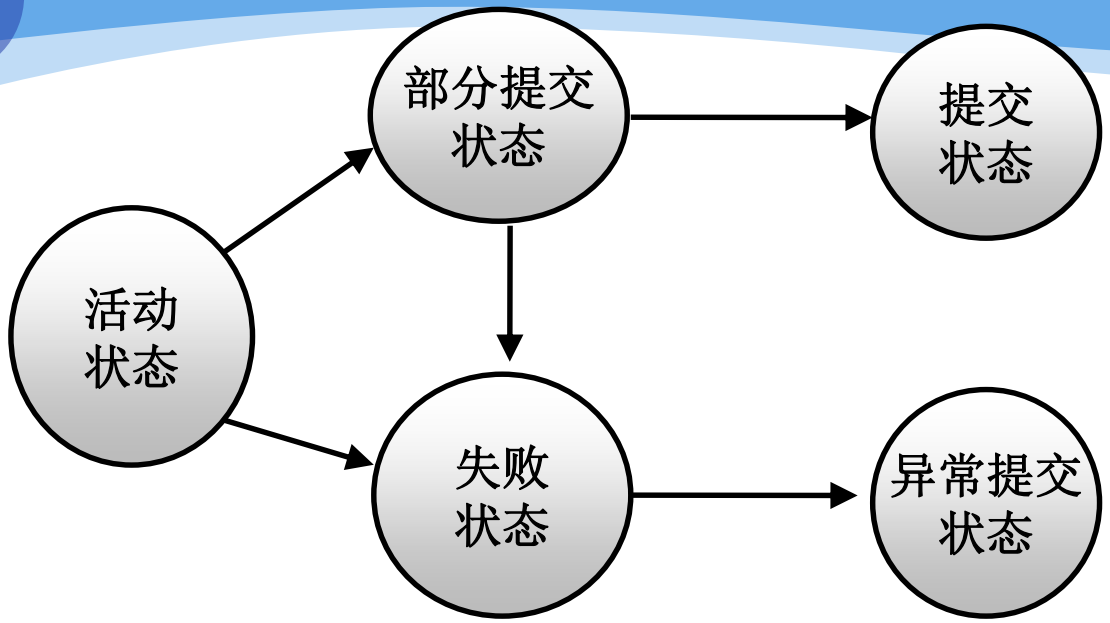
- 7.3.1 数据库故障的种类
- 7.3.2 数据库备份
- 7.3.3 数据库恢复



- 故障是不可避免的
  - 系统故障：计算机软、硬件故障；
  - 人为故障：操作员的失误、恶意的破坏等。
- 数据库的恢复

把数据库从错误状态恢复到某一已知的正确状态(亦称为一致状态或完整状态)。

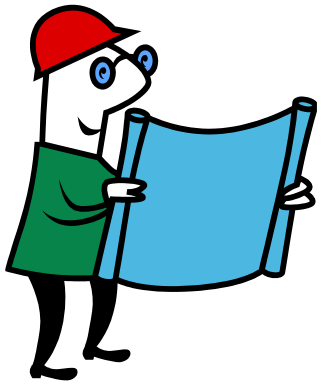
事务的状态及其转换



- 事务开始运行进入活动状态，
- 事务执行完所有语句数据没有存入数据库中
  或未执行完所有语句称为部分提交状态；
- 完成所有操作并永久影响数据库，进入提交状态表示正常完成任务；
- 如果没有正常完成任务最后提交异常，结束事务。

7.3.1 数据库故障的种类

- 事务内部的故障
- 系统故障
- 其他故障



1. 事务内部的故障

- 事务内部故障指非预期的，不能由应用程序处理

  例如：运算溢出，违反了某些完整性限制等。

- 事务故障的恢复：

  事务故障意味着事务没有达到预期的终点，数据库可能处于不正确状态，因此，要在不影响其它事务运行的情况下，撤销该事务已经做的对数据库的修改，使得该事务好像根本没有运行一样。

- 事务故障的恢复操作：

  撤销事务（UNDO）。

2、系统故障

- 系统故障

  称为软故障，是指造成系统停止运转的任何事件，使得系统要重新启动。 例如：突然断电。

- 带来的问题：
  - 整个系统的正常运行突然被破坏；
  - 所有正在运行的事务都非正常终止；
  - 数据库缓冲区的信息全部丢失；不破坏数据库。

系统故障的恢复

- 发生系统故障时，事务未完成
  - 恢复策略：强行撤消（UNDO）所有未完成事务。
- 发生系统故障时，事务已提交，但缓冲区中的信息尚未完全写回到磁盘上。
  - 恢复策略：重做（REDO）所有已提交的事务。

3. 其他故障

- 介质故障
- 计算机病毒

介质故障

- 介质故障

  称为硬故障，指外存故障。

  - 磁盘损坏；
  - 磁头碰撞；
  - 操作系统的某种潜在错误；
  - 瞬时强磁场干扰。

- 会破坏数据库或部分数据库，并影响正在存取这部分数据的事务。这类故障发生的可能性小，但破坏性很大。

介质故障的恢复

- 1、装入数据库发生介质故障前某个时刻的数据副本；
- 2、重做自此时始（副本备份时刻）的所有成功事务，将这些事务已提交的结果重新记入数据库。

计算机病毒

- 计算机病毒
  - 一种人为的故障或破坏，一种计算机程序；
  - 可以繁殖和传播。
- 危害
  - 破坏、盗窃系统中的数据；
  - 破坏系统文件。
- 应对策略：
  - 查杀病毒，注意安全保护，安装防火墙和实时监控软件；
  - 根据数据备份文件和日志文件恢复数据库。

故障小结

- 各类故障，对数据库的影响有两种：
  - 数据库本身被破坏。
  - 数据库没有被破坏，但数据可能不正确，这是由于事务的运行被非正常终止造成的。

7.3.2数据库备份

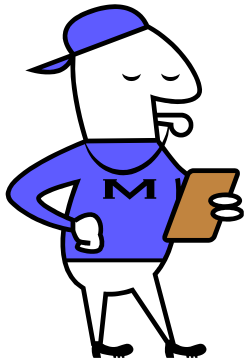
恢复操作的基本原理：冗余

利用存储在系统其它地方的冗余数据来重建数据库中已被破坏或不正确的那部分数据。

- 数据的恢复涉及两个关键问题：
  - 如何建立冗余数据；
    - 备份
  - 如何利用这些冗余数据实施数据库恢复。

备份的内容

- 备份数据（数据转储）
  - 表（结构），包含系统表、用户定义的表。
  - 数据库用户（包括用户和用户操作权）。
  - 用户定义的数据库对象和数据库中的全部数据
- 备份日志（登记日志文件）



备份频率

- 要考虑两个因素：
- 出现故障时，允许丢失的数据量的大小。
- 数据库的事务类型（读多还是写多）以及事故发生的频率（经常发生还是不经常发生）。
- 通常情况下，数据库可以每周备份一次，事务日志可以每日备份一次。
- 对于一些重要的联机事务处理数据库，可以每日备份，事务日志则每隔数小时备份一次。

备份数据（数据转储）

数据转储：

转储是指DBA将整个数据库复制到磁带或另一个磁盘上保存起来的过程，备用的数据称为后备副本或后援副本。

如何使用：

- 数据库遭到破坏后可以将后备副本重新装入；
- 重装后备副本只能将数据库恢复到转储时的状态。

备份方法：

- 静态转储与动态转储
- 海量转储与增量转储

1．静态转储与动态转储

静态转储

- 在系统中无运行事务时进行的转储操作，转储期间不允许对数据库的任何存取、修改活动。得到的是数据一致性的副本 。
- 优点：实现简单。
- 缺点：降低了数据库的可用性。

动态专储

- 转储操作与用户事务并发进行，允许对数据库进行存取或修改。
- 优点：不会影响事务的运行。
- 缺点：不能保证副本中的数据正确有效。

动态转储

例：在转储期间的某个时刻*Tc*，系统把数据A=100转储到磁带上，而在下一时刻*Td*，某一事务将A改为200。转储结束后，后备副本上的A已是过时的数据了。

- 利用动态转储得到的副本进行故障恢复
  - 需要把动态转储期间各事务对数据库的修改活动登记下来，建立日志文件；
  - 后备副本加上日志文件才能把数据库恢复到某一时刻的正确状态。

2．海量转储与增量转储

- 海量转储: 每次转储全部数据库。
- 增量转储: 只转储上次转储后更新过的数据。
- 海量转储与增量转储比较：
  - 从恢复角度看，使用海量转储得到的后备副本进行恢复往往更方便；
  - 但如果数据库很大，事务处理又十分频繁，则增量转储方式更实用更有效。

转储方法分类

		转储状态	
		动态转储	静态转储
转储方式	海量转储	动态海量转储	静态海量转储
	增量转储	动态增量转储	静态增量转储

登记日志文件

一、日志文件的格式和内容

二、日志文件的作用

三、登记日志文件

一、日志文件的格式和内容

- 日志文件(log)：

  用来记录事务对数据库的更新操作的文件.

- 日志文件的格式
  - 以记录为单位的日志文件;
  - 以数据块为单位的日志文件.

以记录为单位的日志文件

- 以记录为单位的日志文件内容
  - 事务标识；
  - 事务的开始标记；
  - 事务的结束标记；
  - 事务的所有更新操作。
    - 类型、对象；
    - 更新前旧值；
    - 更新后新值；

日志记录

以数据块为单位的日志文件

- 以数据块为单位的日志文件的内容
  - 事务标识；
  - 被更新的数据块：{ 更新前 / 更新后 }

二、日志文件的作用

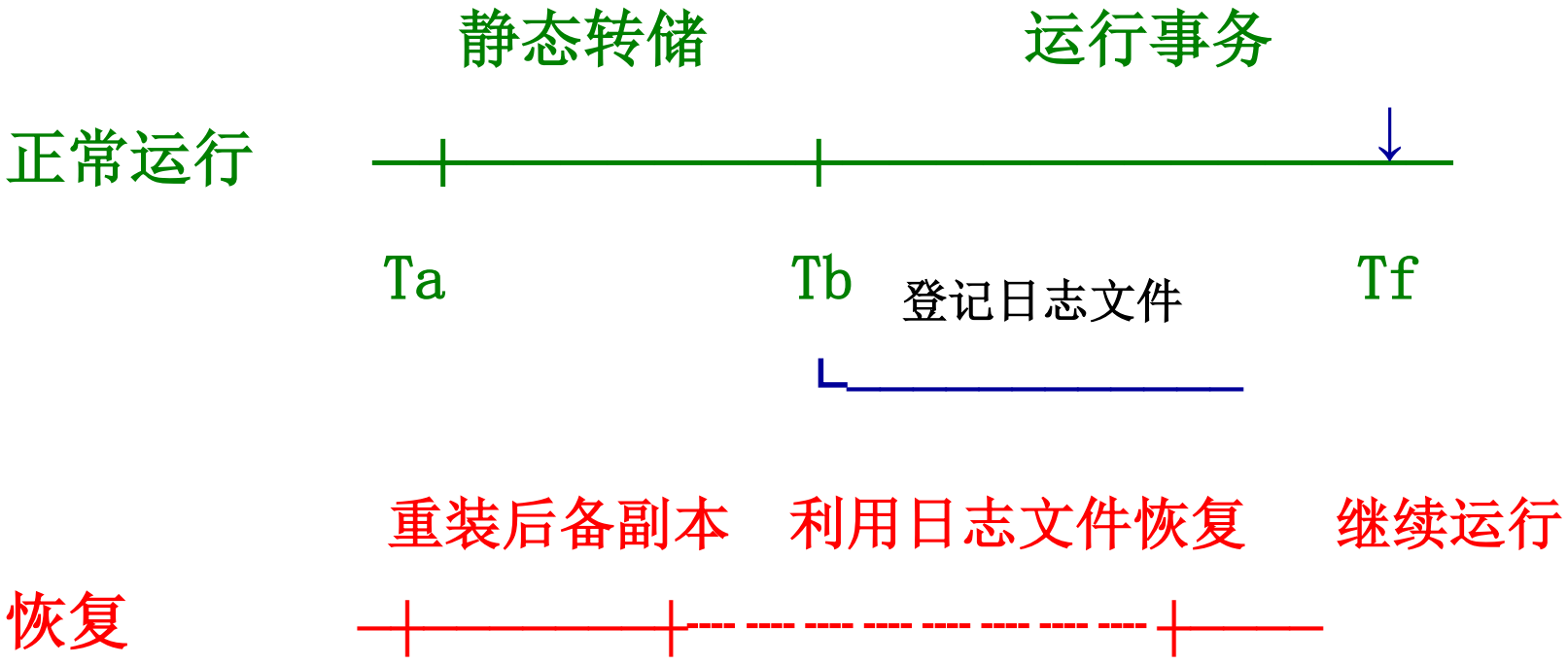
- 日志文件的三个作用：

  1、进行事务故障恢复；

  2、进行系统故障恢复；

  3、协助后备副本进行介质故障恢复。

利用静态转储副本和日志文件进行恢复



三、登记日志文件

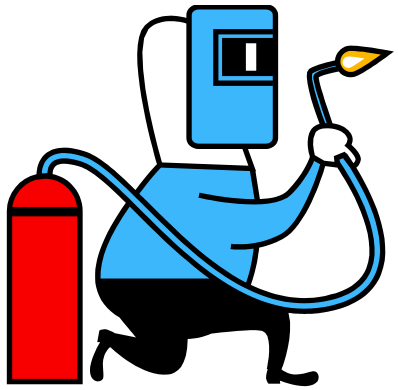
基本原则

- 登记的次序严格按并行事务执行的时间次序；
- 必须先写日志文件，后写数据库：
  - 写日志文件操作：将修改的记录写到日志文件。
  - 写数据库操作：把对数据的修改写到数据库中。

7.3.3数据库恢复

恢复数据库是指将数据库从错误描述状态恢复到正确的描述状态（最近的正确时刻）的过程。

- 恢复策略
- 恢复技术



恢复策略

1 事务故障的恢复

2 系统故障的恢复

3 介质故障的恢复

1　事务故障的恢复

- **事务故障：**

  事务在运行至正常终止点前被终止。

- **恢复方法：**
  - 利用日志文件撤消此事务已对数据库进行的修改。
- 事务故障的恢复由系统自动完成，对用户是透明的，不需要用户干预。

事务故障的恢复步骤

1. 反向扫描文件日志，查找该事务的更新操作。

2. 对该事务的更新操作执行逆操作。

3. 继续反向扫描日志文件，查找其他更新操作，同样处理。

4. 直至读到此事务的开始标记，事务故障恢复就完成了。

2　系统故障的恢复

- 系统故障造成数据库不一致状态的原因：
  - 未完成事务对数据库的更新已写入数据库；
  - 已提交事务的更新还留在缓冲区没来得及写入数据库。
- 恢复方法：
  - 1. 撤销故障发生时未完成的事务；
  - 2. 重做已完成的事务。
- 系统故障的恢复由系统在重新启动时自动完成。

系统故障的恢复步骤

1. 正向扫描日志文件：
   - 重做队列: 在故障发生前已经提交的事务；
   - 撤销队列: 故障发生时尚未完成的事务；
2. 对撤销队列事务进行撤销处理：
   - 反向扫描日志文件，对每个撤销事务执行逆操作。
3. 对重做队列事务进行重做处理：
   - 正向扫描日志文件，对每个重做事务重新执行。

3　介质故障的恢复

1. 重装数据库；

2. 重做已完成的事务。

介质故障恢复的步骤

1、装入最新的后备数据库副本，使数据库恢复到最近一次转储时的一致性状态。

- 利用静态转储的数据库副本，装入数据库后，系统处于一致性状态
- 利用动态转储的数据库副本，还须同时装入转储时刻的日志文件副本，才能将数据库恢复到一致性状态。

2. 装入有关的日志文件副本，重做已完成的事务。

- 首先扫描日志文件，找出故障发生时已提交的事务的标识，将其记入重做队列。
- 然后正向扫描日志文件，对重做队列中的所有事务进行重做处理。

介质故障的恢复

介质故障的恢复需要DBA介入。

- DBA的工作：
  - 重装最近转储的数据库副本和有关的各日志文件副本；
  - 执行系统提供的恢复命令。
- 具体的恢复操作仍由DBMS完成。

数据库恢复技术

1、具有检查点的恢复技术

2、数据库镜像

1、 具有检查点的恢复技术

一、问题的提出

二、检查点技术

三、利用检查点的恢复策略

一、问题的提出

- 两个问题
  - 搜索整个日志将耗费大量的时间；
  - 事务重做处理：重新执行，浪费了大量时间。

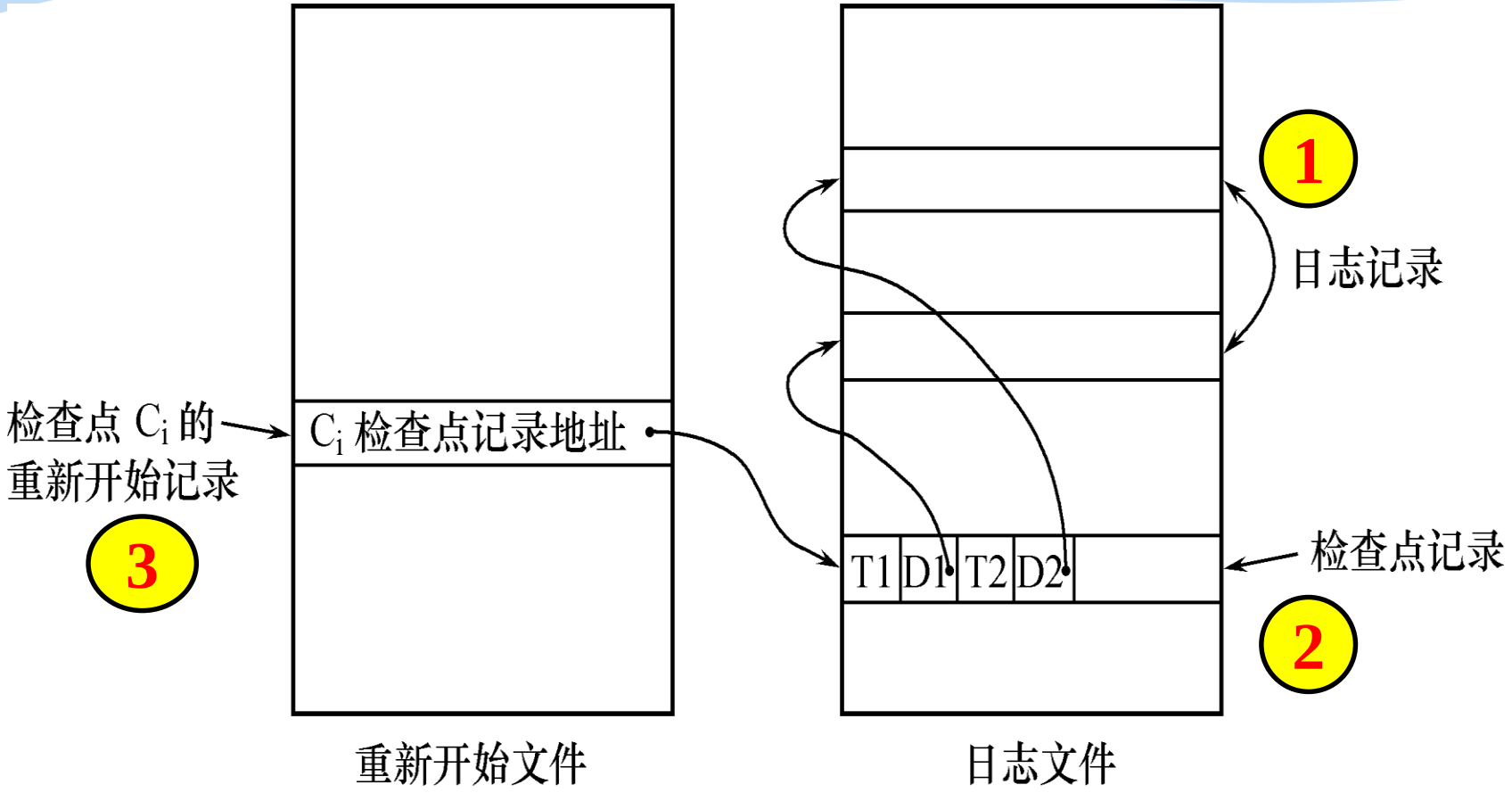
解决方案

- 具有检查点的恢复技术：
  - 在日志文件中增加检查点记录；
  - 增加重新开始文件；
  - 恢复子系统在登录日志文件期间动态地维护日志。

二、检查点技术

- 检查点记录的内容
  - 1. 建立检查点时刻所有正在执行的事务清单;
  - 2. 这些事务最近一个日志记录的地址.
- 重新开始文件的内容
  - 记录各个检查点记录在日志文件中的地址.

具有检查点的日志文件和重新开始文件



动态维护日志文件的方法

- 方法:

  周期性建立检查点，保存数据库状态。

- 具体步骤是:
  - 1.将当前日志缓冲区中的所有日志记录写入日志文件上;
  - 2.在日志文件中写入一个检查点记录;
  - 3.将当前数据缓冲区的所有数据记录写入数据库中;
  - 4.把检查点记录的地址写入一个重新开始文件.

建立检查点

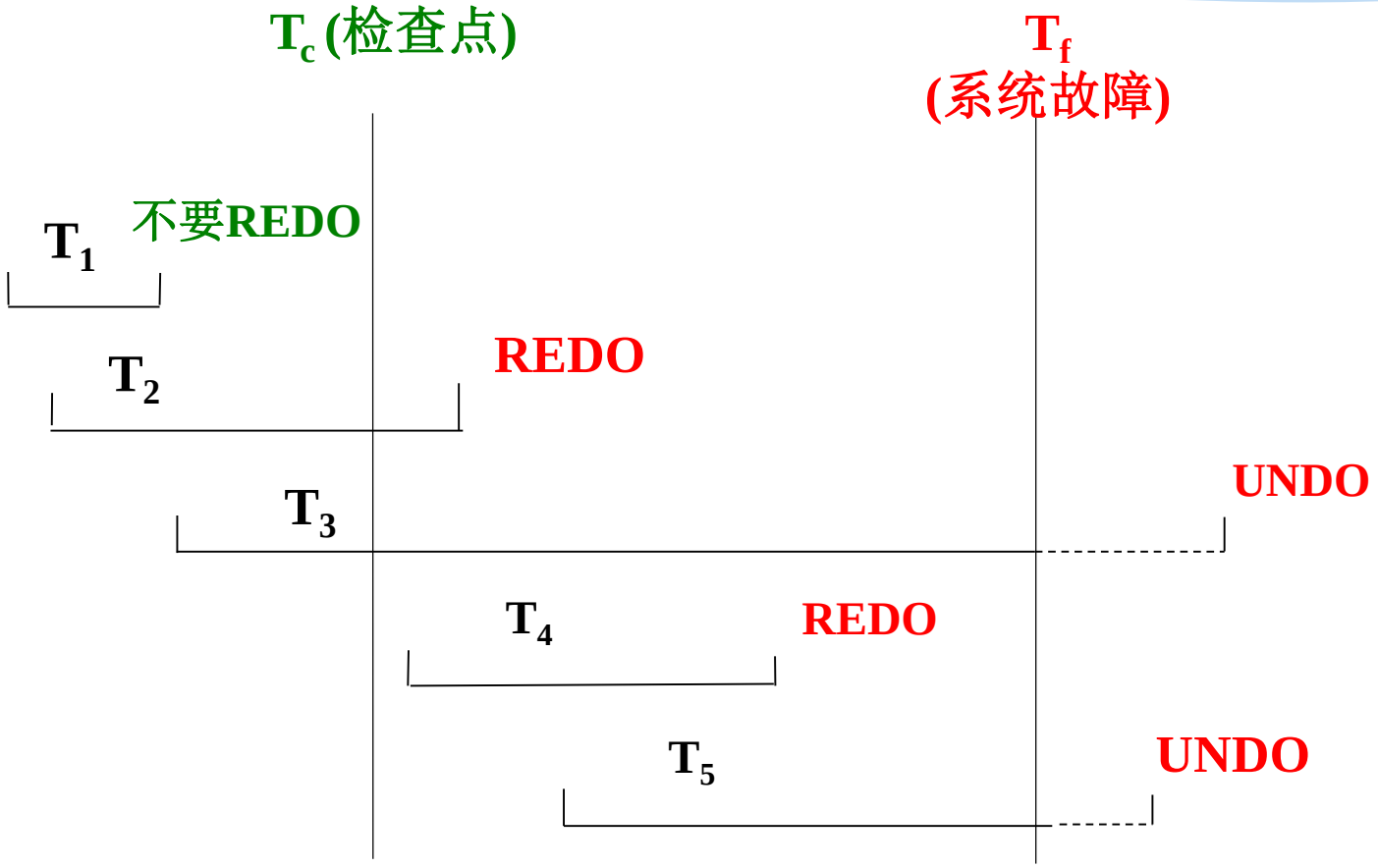
- 恢复子系统可以定期或不定期地建立检查点,保存数据库状态 .
  - 定期
    - 按照预定的一个时间间隔，如每隔一小时建立一个检查点
  - 不定期
    - 按照某种规则，如日志文件已写满一半建立一个检查点.

三、利用检查点的恢复策略

- 使用检查点方法可以改善恢复效率。
  - 当事务T在一个检查点之前提交：

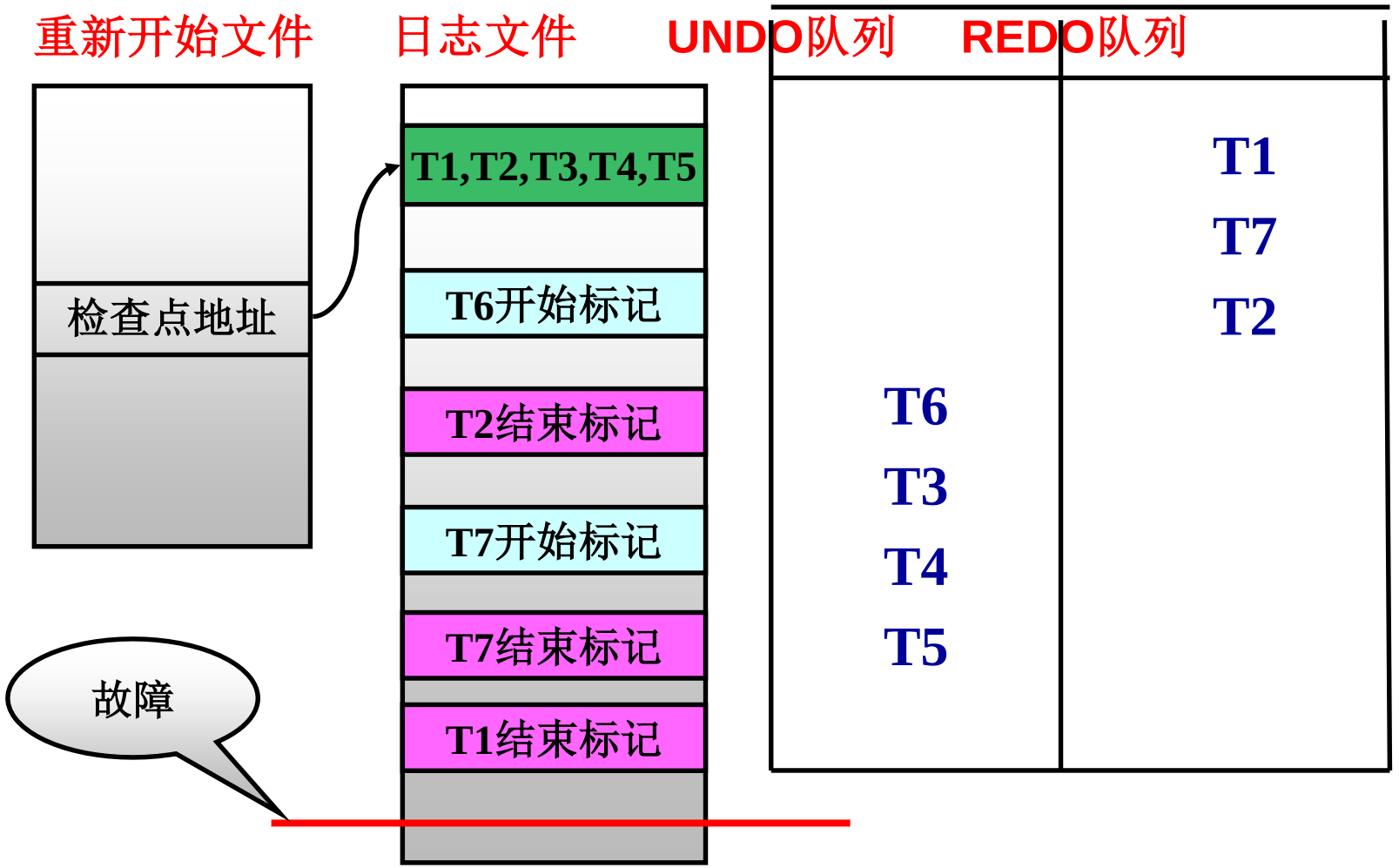
    事务 T 所做的修改已写入数据库，写入时间是在这个检查点建立之前。这样，检查点之后的故障，进行故障恢复处理时，没有必要对事务T执行重做操作。

利用检查点的恢复策略（续）



系统出现故障时，恢复子系统将根据事务的不同状态采取不同的恢复策略。

利用检查点的恢复步骤


重新开始文件
日志文件
UNDO队列
REDO队列
检查点地址
T1,T2,T3,T4,T5
T6开始标记
T2结束标记
T7开始标记
T7结束标记
T1结束标记
故障
T1
T7
T2
T6
T3
T4
T5

2　数据库镜像

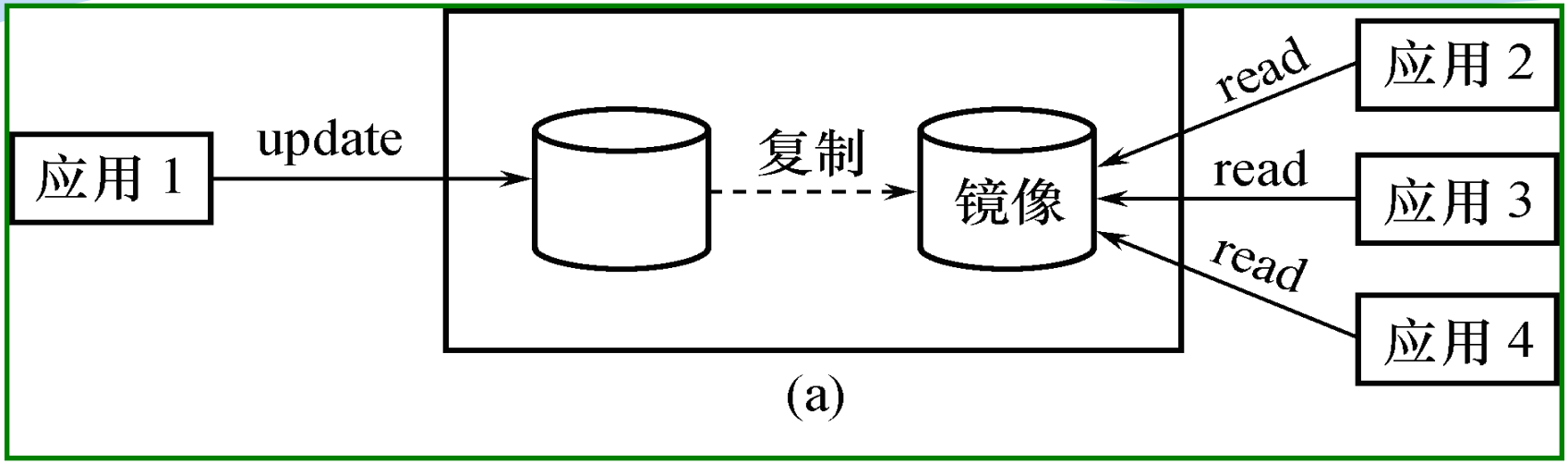
- 问题：
  - 介质故障是对系统影响严重，影响数据库的可用性。
  - 介质故障恢复比较费时；
  - 为预防介质故障，DBA必须周期性地转储数据库。
- 解决方案：
  - 数据库镜像（Mirror）：

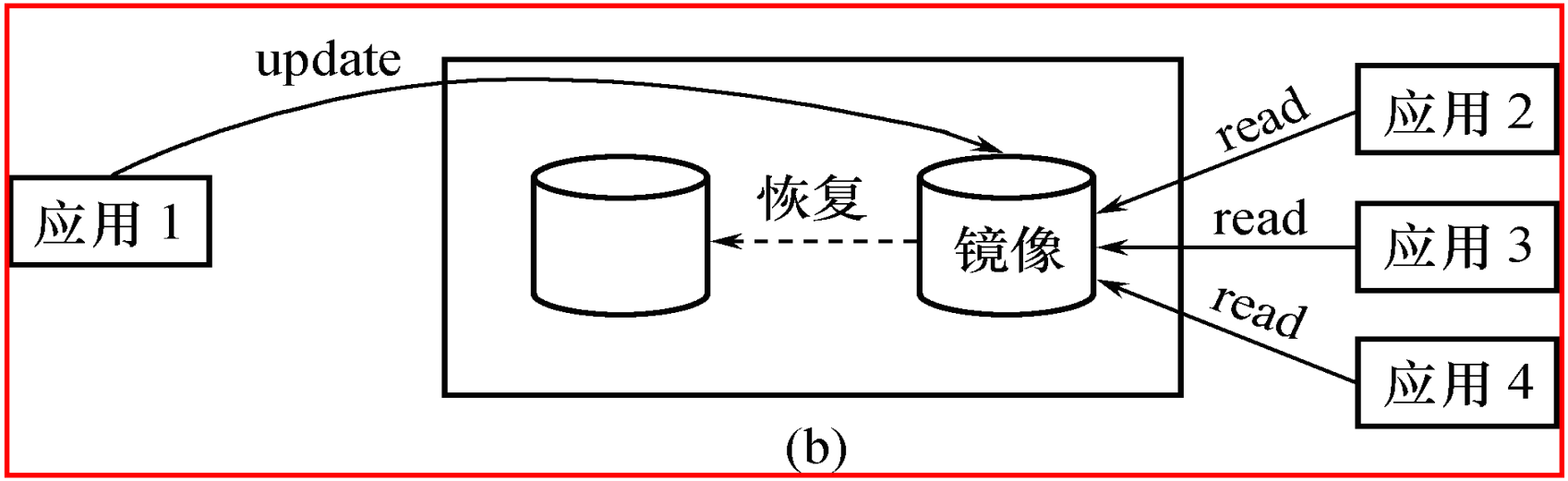
DBMS自动把整个数据库或关键数据复制到另一个磁盘上。

DBMS自动保证镜像数据与主数据库的一致性。

当主数据库更新时，DBMS自动把更新后的数据复制过去。

数据库镜像的用途


应用 1
update
复制
镜像
read
read
read
应用 2
应用 3
应用 4
(a)


update
应用 1
恢复
镜像
read
read
read
应用 2
应用 3
应用 4
(b)

数据库镜像的用途

- 没有出现故障时：

  数据库镜像可用于并发操作。

- 出现介质故障时：
  - 可由镜像磁盘继续提供使用；
  - 同时DBMS自动利用镜像磁盘数据进行数据库的恢复；
  - 不需要关闭系统和重装数据库副本。
- 频繁地复制数据自然会降低系统运行效率。

  只对关键数据和日志文件镜像，而不是对整个数据库进行镜像。

小结

- 保证数据一致性是对数据库的最基本的要求。
- 事务是数据库的逻辑工作单位：ACID特性。
- 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：数据转储和登记日志文件。
- 恢复的基本原理：利用存储在后备副本、日志文件和数据库镜像中的冗余数据来重建数据库。
- 提高恢复效率的技术：检查点技术、镜像技术。

作业

📖 P93