



数据库原理及应用讲义

第5章 数据库管理

主讲：陈安龙

chenanlong@uestc.edu.cn

电子科技大学

复杂工程问题——数据库管理

一、数据库管理的工程问题探讨

- 1、DBMS有哪些主要管理功能？
- 2、在特定数据库应用处理中，为什么需要事务机制？
- 3、在SQL中如何编写事务程序？
- 4、并发事务调度解决什么问题？
- 5、不同级别的加锁协议，可以分别解决哪些数据不一致问题？
- 6、并发事务出现死锁的四个必要条件是什么，如何打破死锁？
- 7、如何针对角色或用户进行数据库对象访问的权限管理？

二、数据库应用编程实践案例问题探讨

针对成绩管理系统的编程中，探讨该数据库程序实施过程。

- 1、在 PostgreSQL 数据库中，如何设计实现成绩管理系统数据库角色？
- 2、成绩管理系统数据库的教务人员的权限最小集包括哪些权限？
- 3、如何验证用户的数据库访问权限？
- 4、如何备份和恢复成绩管理数据库？

挑战性问题——数据库管理

一、数据库管理问题探讨

- 1、在事务编程中，如何避免死锁的产生？
- 2、事务中，回退点的设置考虑因素有哪些？
- 3、在 DBMS 中，如何设置显式事务和隐含事务模式？
- 4、在 DBMS 中，事务隔离级别应如何设置？

二、工程案例的数据库编程优化探讨

针对成绩管理系统，探讨数据库管理问题。

- 1、在成绩管理系统中，创建教务管理人员账户，并为其赋予相应的权限。
- 2、在成绩管理系统中，如何确保教师用户只能修改自己承担课程的信息，教材数据库设计是否能够满足该要求，如果不能，你考虑增加哪些数据表？
- 3、在成绩管理系统，可能有上万的学生，如果为每个学生建立一个用户，是否是最好的选择，有没有更好的解决方法？
- 4、在成绩管理系统，哪些操作可能会产生死锁，轻描述一个可能的产生死锁的应用流程？

第5章 数据库管理

学习目的和要求

- 了解数据库管理的内容
- 理解数据事务概念及特点
- 掌握数据库安全管理技术
- 理解并掌握数据库备份与恢复技术
- 理解数据库并发执行的问题
- 掌握数据库并发控制技术

一、什么是数据库管理

数据库管理(Database Management)是指为保证数据库系统的正常运行和服务质量必须进行的系统管理工作。

二、为什么需要数据库管理

- 数据库系统随规模增大,系统会变得异常复杂
- 多用户数据库应用带来数据库访问复杂性
- 数据安全和数据隐私对机构和用户都非常重要
- 数据库系统随数据量增加和使用时间增长其性能会降低
- 系统遭遇意外事件, 数据库损坏或数据丢失

三、数据库管理目标

- 保障数据库系统正常稳定运行
- 充分发挥数据库系统的软硬件处理能力
- 确保数据库系统安全和用户数据隐私性
- 有效管理数据库用户及其角色权限
- 解决数据库系统性能优化、系统故障与数据损坏等问题
- 最大程度地发挥数据库对其所属机构的作用

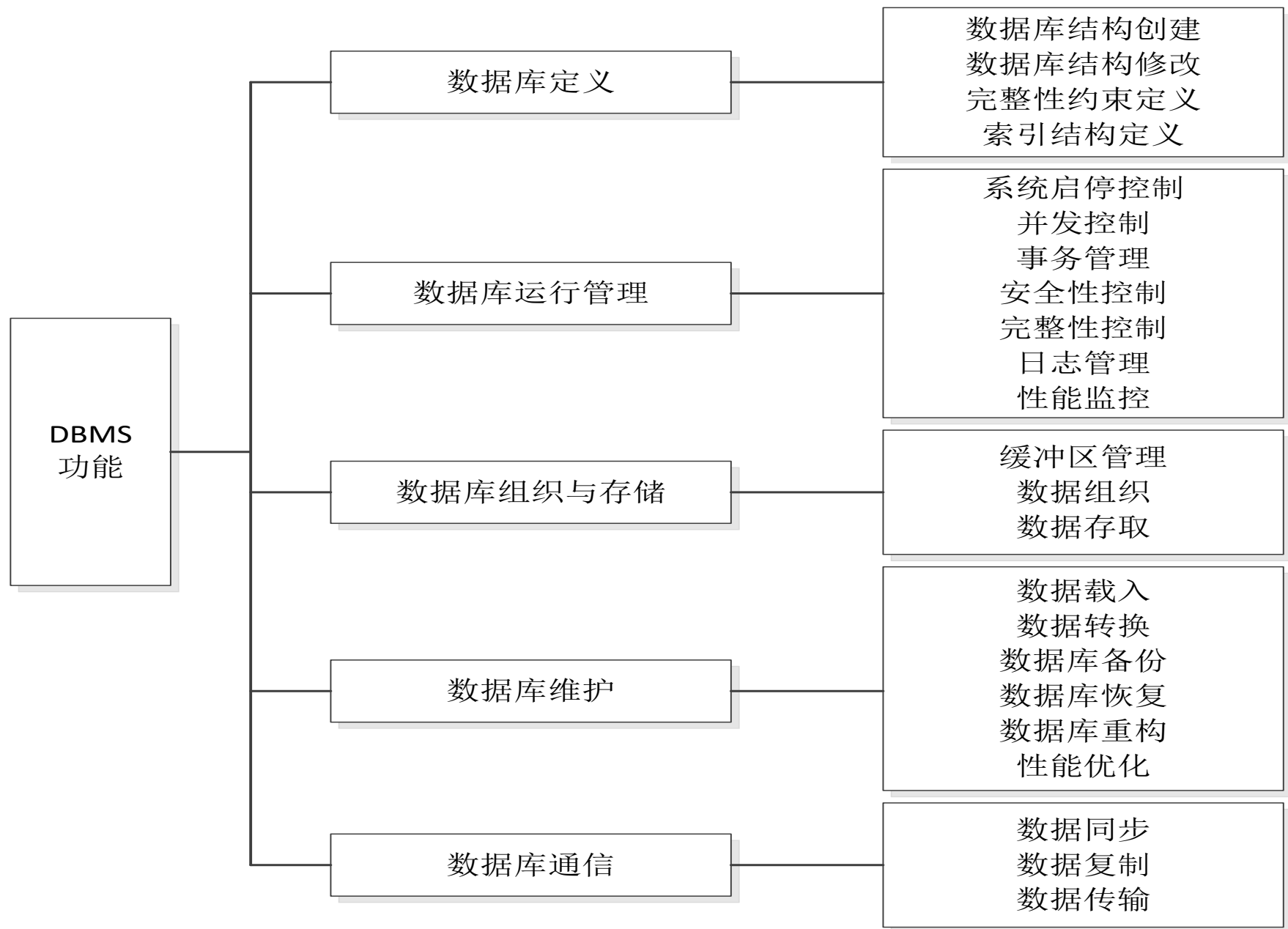
四、数据库管理内容

- ① DBMS系统运行管理
- ② 数据库性能监控
- ③ 数据库索引管理
- ④ 数据库查询优化
- ⑤ 数据库事务并发控制
- ⑥ 数据库角色管理
- ⑦ 数据库用户管理
- ⑧ 数据库对象权限管理
- ⑨ 数据安全的管理
- ⑩ 数据库备份
- ⑪ 数据库恢复

五、数据库管理员（DBA）职责

- ① 负责数据库系统开发与运维
- ② 负责数据库用户与权限管理
- ③ 负责数据库备份与数据库恢复管理
- ④ 负责数据库性能调优管理

六、DBMS管理功能结构



对数据库中存储的大量数据，有下面几个问题：

- ① 如何使数据资源只被相关人员合理使用？
- ② 如何恢复被破坏的数据？
- ③ 如何协调多用户的工作来保证数据的一致性？
- ④ 如何自动地发现用户的失误？

作为一个完善的DBMS，应该提供统一的数据管理功能来保证数据的安全可靠和正确有效！

数据管理也叫数据控制，主要包括：

数据库恢复

并发控制

数据的安全性

数据的完整性

本章首先讨论数据库恢复技术。

问题：系统软、硬件故障对系统数据造成破坏时，该如何处理？

例：银行转帐

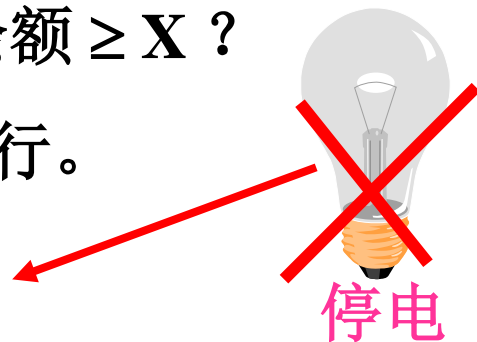
设从帐号A中拨一笔款X到帐号B，正常的执行过程是：

☆ 查看帐号A上是否有足够的款数，即余额 $\geq X$ ？

若余额 $< X$ ，则给出提示信息，中止执行。

若余额 $\geq X$ ，则执行下面三步：

- ⌚ 在A中记上一笔支出，从余额中减去 X；
- ⌚ 把值X传到B上；
- ⌚ 在B中记上一笔收入，在余额上加X，结束。



若在执行了第二步后突然断电或线路传输错误，则导致帐面不平

§ 1 事务的基本概念

1、事务（transaction）

主要是更新操作

一个数据库操作序列，是数据库应用程序的基本逻辑单元。这些操作要么都做，要么都不做，是一个不可分割的执行单位。

事务标记: **BEGIN TRANSACTION**

事务开始

⋮

COMMIT 或 ROLLBACK

事务提交：
事务完成了其包含的所有活动，正常结束

事务回滚（中止）：
撤消已做的所有操作，回到事务开始时的状态

DBMS中的事务控制:

(1) 隐式的事务控制: 默认情况下, DBMS一般将一个数据库操作 (如一条SQL语句) 当作一个事务来控制执行。

说明: 事实上, 有时一条SQL语句的工作也有事务特点, 例如一条删除多行数据的SQL语句。

(2) 显式的事务控制: 对涉及多步操作的 (一般含多条SQL语句)、有事务特点的工作, 则需要人为地、显式地将这些操作 “**界定**” 组合成一个事务交DBMS控制执行。

事务定义示例

■ 事务的显式定义

BEGIN TRANSACTION

SQL 语句1

SQL 语句2

...

COMMIT

END TRANSACTION

BEGIN TRANSACTION

SQL 语句1

SQL 语句2

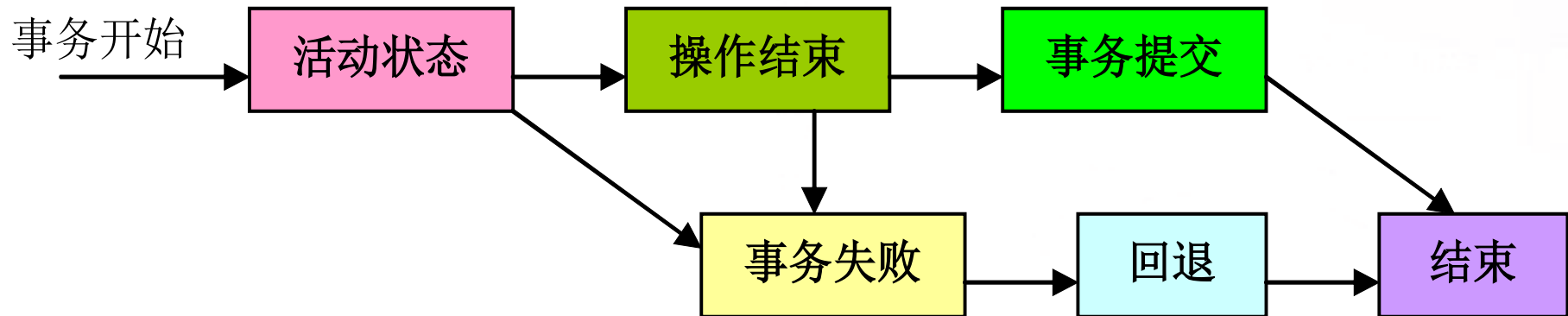
...

ROLLBACK

END TRANSACTION

■ 没有显式定义事务时，**DBMS**缺省为自动划分事务

事务状态：根据事务状态变迁，事务状态有：活动状态、操作结束、事务提交、事务结束、事务失败、回退。



事务状态变迁图

事务的两种结局：①提交而结束，其对DB的更新才能被其它事务访问；②事务失败，事务所作的DB更新必须回退。

故障事务处理的依据：事务是已提交，还是未提交。

2、事务应具有的性质

(1) 原子性 (Atomicity) : 事务执行时的不可分割性，
即事务所包含的活动要么都做，要么都不做

若事务因故障而中止，则要设法消除该事务所产生的影响，使数据库恢复到该事务执行前的状态。

(2) 一致性 (Consistency) : 事务对数据库的作用应
使数据库从一个一致状态到另一个一致状态

例如：一个帐号的收支之差应等于余额。

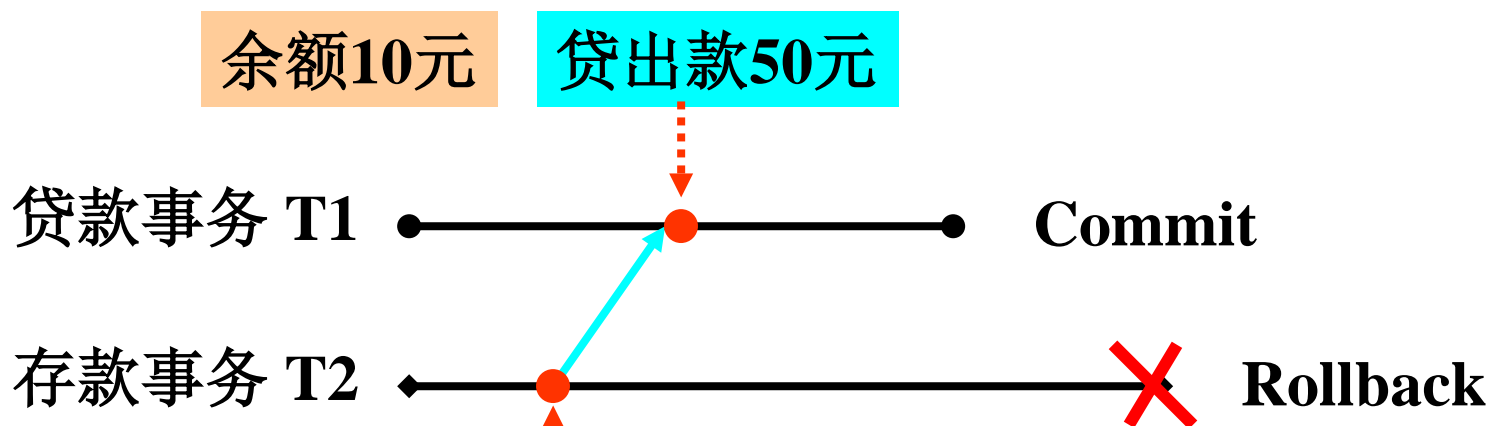
飞机订票系统，事务执行前后，座位与订出座位等信息必须一致。

2、事务应具有的性质

(3) 隔离性 (Isolation) :

多事务并发执行，应象各事务独立执行一样，不能相互干扰。一个正在执行的事务其中间结果不能为其它事务所访问。

例如：有两个事务，在同一帐号上存款和贷款：



**T2中止，T1也必须中止，
造成链式事务中止
(cascading aborts)**

2、事务应具有的性质

能恢复

(4) 持久性 (Durability) : 一旦事务提交, 不论执行何种操作或发生何种故障, 都不应对该事务的执行结果有任何影响。

(5) 可串行性 (Serializability) : 并发控制正确性的标准用户程序在逻辑上是正确的, 它在串行执行时没有问题; 当多个事务并发执行时, 可以等价于一个串行执行序列。

3、事务管理任务

事务管理的任务就是要保证事务满足上述性质。使事务不具有上述性质的因素可能是:

- (1) 事务在运行过程中被强行终止;
- (2) 多个事务并行运行时, 不同事务的操作交叉执行。

因此事物管理又分为两个方面：

恢复： 保证事务在故障时满足上述性质的技术。

并发控制： 保证事务在并发执行时满足上述性质的技术。

§ 2 故障的种类

数据库系统中可能发生各种各样的故障，分为以下几类。

1、事务内部的故障

☆ 可以通过事务程序本身发现并处理的故障

如银行转帐事务程序在余额小于转帐额时的情形

⌚ 非预期的故障（不能由应用程序处理）

如运算溢出、被零除、发生死锁时被选中撤消等

通常，所说的事务故障仅指非预期故障。事务故障意味着事务没有达到预期的终点（**COMMIT**或者显式的**ROLLBACK**），因此数据库可能处于不一致状态，恢复程序应在不影响其他事务的情况下，撤消故障事务的所有修改，使得故障事务就象没有运行一样。这类操作称为事务撤消（**UNDO**）。

2、系统范围内的故障：软故障

造成系统停止的任何事件，如CPU故障、操作系统故障、程序代码错误、断电等，使得系统必须重新启动。

这类故障的特征是：影响所有正在运行的事务，但不破坏数据库。它们可引起缓冲区内容丢失，并使所有正在运行的事务不能到达预期终点。

系统故障发生时，**可能使数据库处于不一致状态**：

一方面，有些**非正常终止事务的结果可能已写入数据库**，在系统下次启动时，恢复程序必须回滚这些非正常终止的事务，撤销这些事务对数据库的影响。

另一方面，**有些已完成事务的结果可能部分或全部留在缓冲区，而尚未写回磁盘上的数据库中**。在系统下次启动时，恢复程序必须重做（**REDO**）所有已提交的事务，将数据库真正恢复到一致状态。

3、介质故障：硬故障

如磁盘损坏、磁头碰撞、强磁场干扰等。

这类故障发生概率很小，但破坏性极大，将破坏部分甚至整个数据库的内容，并影响使用相应数据的所有事务。

4、计算机病毒

§ 3 恢复的实现技术

数据库故障对数据库的影响

- * 数据库本身被破坏；
- * 数据库没有破坏，正在运行的事务被非正常终止，可能造成数据库数据不正确。

数据库恢复的基本原理-----冗余

数据库任何一部分被破坏或数据不正确时，可根据存储在系统别处的数据来重建。

数据库恢复的机制（两步）

- * 建立冗余数据
常用技术：数据转储、登记日志文件
- * 利用冗余数据实施数据库恢复

§ 3 恢复的实现技术

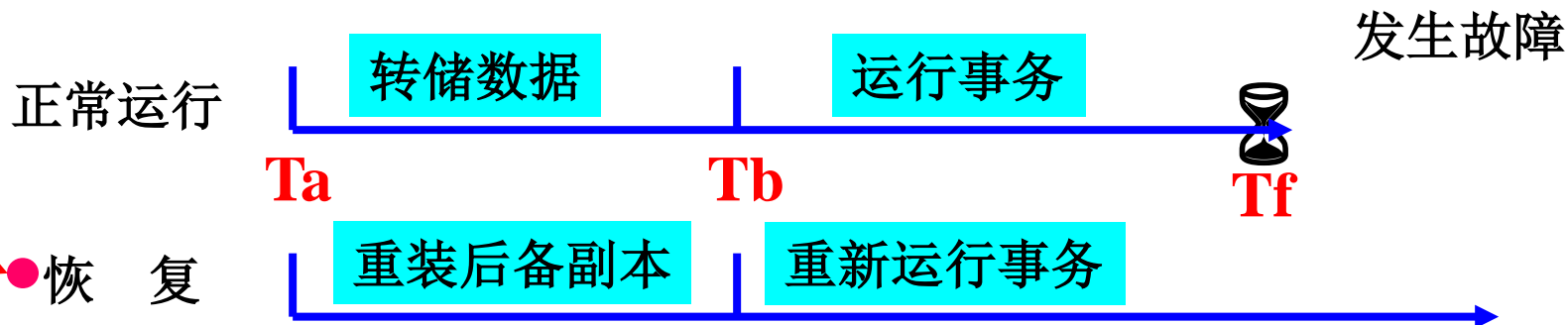
1、数据转储与恢复

转储：DBA定期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。

（这些备用的数据称为后备副本或后援副本）

● **恢复：**当数据库被破坏后可将后备副本重新装入，并重新运行转储以后的所有更新事务。

例：**Ta**时刻系统停止运行事务开始转储，**Tb**时刻转储完毕重新开始运行事务，**Tf**时刻发生故障。



§ 3 恢复的实现技术

转储的状态

静态转储：转储期间 不允许对数据库进行操作

特点：静态转储得到的一定是一个数据一致性的副本。因为转储必须等用户事务全部结束才能进行，而且新的事务必须等待转储完毕才能开始执行。但数据库的可用性被降低。

动态转储：转储期间 允许对数据库进行操作

特点：转储和用户事务可并发执行，即不必等待正在运行的事务结束，也不影响新事务的运行。但转储的数据可能已过时。

为此，必须**建立日志文件**，记录转储期间对数据库的更新活动。这样，后援副本加日志文件就能把数据库恢复到某个时刻的一致性状态。

§ 3 恢复的实现技术

转储方式

海量转储：每次转储全部数据库（一般每周一次）

增量转储：只转储上次转储后更新过的数据

（一般每天一次）

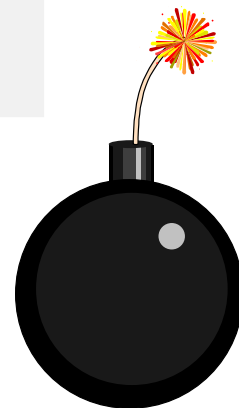
转储方法 { 动态海量 动态增量
 静态海量 静态增量

转储的缺点

费时

在转储后和故障点之间的数据更新不能恢复
动态转储时转储的数据可能已过时


注意：对大中型数据库系统来说，
转储是非常重要的！



§ 3 恢复的实现技术

2、日志文件和恢复

日志 (log) : 用来记录对数据库的更新操作的文件。

 动态转储方式必须建立日志文件
静态转储方式最好建立日志文件

日志文件的格式和内容

* **以记录为单位的日志文件**

系统把  事务开始 (**BEGIN TRANSACTION**)
事务提交 (**COMMIT**) 或
事务撤消 (**ROLLBACK**)
对数据库的插入、删除、修改等

每一个操作作为一条记录存放到日志文件中

§ 3 恢复的实现技术

每条日志记录的主要内容

事务标识（哪个事务）

操作类型（插删改）

操作对象（哪条记录）

更新前数据的旧值,称为前像AI

← 对插入此项为空

更新后数据的新值,称为后像BI

← 对删除此项为空

* 以数据块为单位的日志文件

将事务标识及更新前后的数据块均放在日志文件中。

日志文件的作用

静态转储：数据库毁坏后，重装后援副本，根据日志文件，重做已完成的事务，并撤消未完成的事务。

动态转储：用后援副本和日志文件综合起来恢复数据库

为保证数据库可恢复应遵守的规则

1.提交规则(Commit Rule)

该规则要求：**后像**应在提交前写入磁盘**DB**或日志。

① **ACID**持久性要求。

② 提交规则不要求后像一定在提交前写入**DB**，**写入日志中亦可**，即：如后像已写入日志，即使未写入**DB**或未完全写入**DB**，事务仍可提交，待事务提交后再继续写入**DB**。如此期间出现故障，可用日志的后像重做；其它事务可从缓冲区访问更新后的内容。

2.先记后写规则(Log Ahead Rule)—即先写日志后写数据库

该规则要求：如果**后像**在事务**提交前写入DB**，需先将**前像写入日志**，以便事务失败后，撤销事务所做的更新。

登记日志文件

原则：严格按并发事务执行的时间次序登记；

先写前像AI到日志文件，后写后像BI到数据库。

日志提前写规则，即：先记后写规则(Log Ahead Rule)

写数据库和写日志文件是两个不同的操作，在这两个操作之间有可能发生故障，若先写数据库数据，再写日志的话，万一在写日志前发生故障，则这次的数据库修改未登记，从而不能恢复。若写日志后发生故障而未修改数据库，则事务一定未完成，在恢复时会执行撤消处理。

如：欲将数据库中某记录字段的值由5改为8，登记日志文件后发生故障，则字段值仍为5，日志中不会登记该事务的COMMIT或ROLLBACK记录，事务未完成，恢复时对该操作做撤消处理，将字段值改为该修改操作的旧值5，数据库内容不变。

§ 4 恢复策略

发生故障时，利用数据库后援副本和日志文件可以将数据库恢复到某个一致性状态，但不同故障的恢复策略和方法是不一样的。

利用日志文件进行恢复

基本策略：

对于尚未提交的事务，执行撤消处理（**UNDO**）

对于已经提交的事务，执行重做处理（**REDO**）

基本方法：

扫描日志文件，确定所有已开始但尚未提交的事务（对它们需**UNDO**），再确定所有已提交的事务（对它们需**REDO**）

UNDO处理：若事务提交前出现异常，则对已执行的操作进行撤消处理，使数据库恢复到该事务开始前的状态。具体做法是：**反向扫描日志文件**，对每个需UNDO的事务的更新操作执行反操作。即对已插入的记录执行删除，对已删除的记录重新插入，对已修改的记录用旧值代替新值。

UNDO处理是维护事务的原子性所必须的

REDO处理：重做已提交事务的操作。

具体做法是：**正向扫描日志文件**重新执行登记的操作

有些事务虽已发出**COMMIT**操作，但更新的结果可能只是写到缓冲区而未能写入磁盘，或磁盘上数据库被破坏，因此需要**REDO**处理。

例如：事务T1在学生表S上执行下面三个操作：

```
INSERT INTO S VALUES ( 'S4', 'D', 'CS', 19 ) ;
```

```
DELETE FROM S WHERE S# = 'S1';
```

```
UPDATE S SET SD='CS' WHERE S# = 'S2';
```

执行事务并记录日志文件

写日志

S1	A	CS	20
S2	B	CI	21
S3	C	MA	19

事务T1执行前的S

事务T1开始

S1	A	CS	20
S2	B	CI	21
S3	C	MA	19
S4	D	CS	19

```
INSERT INTO S  
VALUES ( 'S4', 'D',  
        'CS', 19 );
```

T1
在S中插入键
为S4的记录

S4	D	CS	19
----	---	----	----

S2	B	CI	21
S3	C	MA	19
S4	D	CS	19

```
DELETE FROM S  
WHERE S#='S1';
```

T1
在S中删除键
为S1的记录

S1	A	CS	20
----	---	----	----

S2	B	CS	21
S3	C	MA	19
S4	D	CS	19

```
UPDATE S  
SET SD='CS'  
WHERE S#='S2';
```

T1
在S中修改键
为S2的记录

S2	B	CI	21
----	---	----	----

S2	B	CS	21
----	---	----	----

如果事务已提交执行REDO的处理过程

日志

若数据库
中的状态是:

S1	A	CS	20
S2	B	CI	21
S3	C	MA	19
S4	D	CS	19

S2	B	CI	21
S3	C	MA	19
S4	D	CS	19

S2	B	CS	21
S3	C	MA	19
S4	D	CS	19

REDO处理

正向
扫描

此时无数据修改

事务T1开始

T1

在S中插入键
为S4的记录

S4	D	CS	19
----	---	----	----

T1

在S中删除键
为S1的记录

S1	A	CS	20
----	---	----	----

T1

在S中修改键
为S2的记录

S2	B	CI	21
S2	B	CS	21

如果事务未提交执行UNDO的处理过程

日志

S1	A	CS	20
S2	B	CI	21
S3	C	MA	19

S1	A	CS	20
S2	B	CI	21
S3	C	MA	19
S4	D	CS	19

S2	B	CI	21
S3	C	MA	19
S4	D	CS	19

S2	B	CS	21
S3	C	MA	19
S4	D	CS	19

UNDO处理

事务T1开始

T1

在S中插入键
为S4的记录

S4 D CS 19

T1

在S中删除键
为S1的记录

S1 A CS 20

T1

在S中修改键
为S2的记录

S2 B CI 21

S2 B CS 21

反向
扫描

故障恢复过程的处理

1、事务故障的恢复

事务故障是指事务被非正常终止，应根据日志文件对未完成事务做**UNDO**处理，步骤如下：

- (1) 反向扫描日志文件，查找未完成事务的更新操作；
- (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，对遇到的更新操作做同样 处理；
- (4) 当遇到某事务的开始标记时，停止对该事务的处理。
- (5) 重复上述过程，直到所有未完成事务全部**UNDO**完毕。

2、系统故障的恢复

系统故障造成数据库不一致的原因，一是未完成事务对数据库的更新已写入数据库，二是已提交事务的结果在故障发生前留在缓冲区没来得及写入数据库。恢复操作是撤消未完成事务，重做已完成事务。步骤如下：

- (1) 正向扫描日志文件，找出在故障发生前已提交的事务，将它们记入重做（**REDO**）队列，同时找出故障发生前尚未完成的事务，将它们记入撤消（**UNDO**）队列。
- (2) 反向扫描日志文件，对**UNDO**队列的每个事务执行逆操作，即做撤消处理。
- (3) 正向扫描日志文件，对**REDO**队列中的每个事务重新执行日志文件登记的操作。

3、介质故障的恢复

介质故障发生后，磁盘上的数据文件和日志文件均被破坏，恢复的方法是重装数据库和日志文件，然后重做自转储以来已完成的事务。步骤如下：

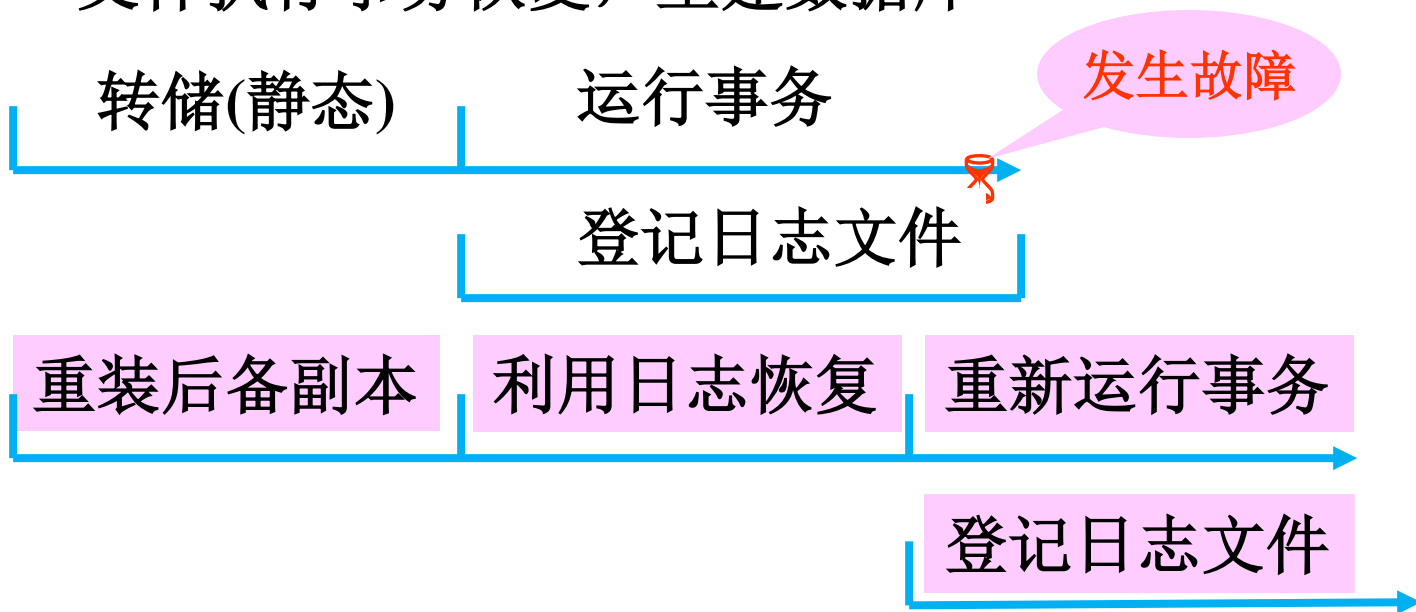
(1) 装入最近转储的数据库后援副本，若是动态转储，则还应装入转储期间的日志文件，将数据库恢复到一致性状态。

(2) 装入转储结束后的日志副本，重做已完成的事务。

系统故障与事务故障的恢复由系统自动完成，对用户透明，介质故障的恢复，需要DBA重装数据库和日志文件副本，然后执行相应的恢复命令。不论那种恢复，一般都要扫描整个日志文件。

恢复策略总结:

- ☆ 当数据库被破坏时, 要重装后备副本, 然后利用日志文件执行事务恢复, 重建数据库



- ✧ 数据库本身未被破坏, 但有些内容可能不正确, 则可只利用日志文件恢复, 使数据库回到某一正确状态

§ 5 具有检查点的恢复技术

利用日志文件恢复数据库，一般要扫描整个日志文件，日志是个流水帐，往往很长，这样做具有两个问题：

- * 搜索整个日志文件将**耗费大量的时间**；
- * 许多已提交事务的**更新结果实际上已写入数据库中**，重新做这些事务只会浪费大量的时间。

因此，确定哪些事务需REDO，哪些不需REDO，就很有意义。解决的方法是：**在日志文件中设置检查点记录**

DBMS周期性地在日志中记录一个**检查点**：将当前正在执行（尚未提交）的所有事务记录于一个记录中——检查点记录。具体工作为：

检查点恢复技术的数据库和日志写入磁盘的过程

- ① 将内存中所有日志记录写入磁盘；
- ② 在磁盘日志文件中写入一个检查点记录；
- ③ 将内存中所有数据库记录写入磁盘数据库中；
- ④ 把检查点记录在日志文件中的地址写入一个重新开始文件中。

检查点记录的内容包括：

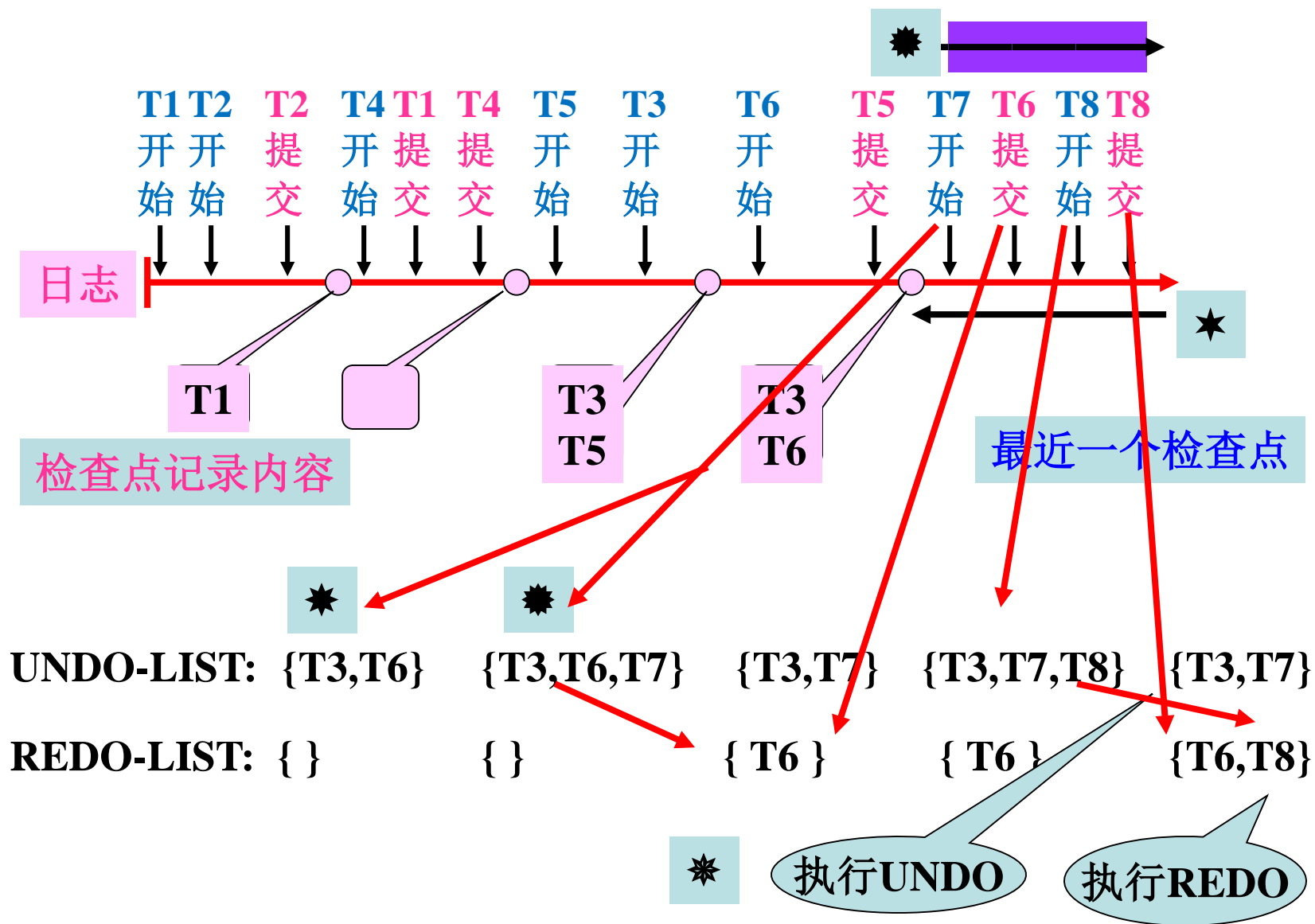
- ① 建立检查点时所有正在执行的事务清单；
- ② 这些事务中最近的一个日志记录地址。

在检查点之前已提交的事务对数据库的修改在检查点之前或检查点建立时已记入磁盘，只要数据库未被破坏，**不需要对这些事务执行重做（REDO）**。

具有检查点的恢复算法

- ① 根据重新开始文件中最后一个检查点记录的地址，在日志文件中找到最近的一个检查点记录；
- ② 设置两个队列，将检查点中的所有事务放入**UNDO-LIST**，并令**REDO-LIST**暂为空集；
UNDO-LIST：需要UNDO操作的事务集合；
REDO-LIST：需要REDO操作的事务集合；
- ③ 从该检查点开始扫描日志文件到文件结束为止：
凡遇有begin_transaction的事务放入**UNDO-LIST**；
凡遇有commit的事务，将它从**UNDO-LIST**移入**REDO-LIST**；
- ④ 对**UNDO-LIST**中的事务执行UNDO操作
- ⑤ 对**REDO-LIST**中的事务执行REDO操作

§ 5 具有检查点的恢复技术



§ 6 数据库镜像

前面已介绍，当数据库系统发生故障时，可利用日志文件进行数据库恢复，但前提是日志文件必须完好。然而当发生介质故障时，往往不仅数据库被摧毁，日志文件也难逃恶运，此时恢复操作就无法实施。这在银行数据库等系统中是绝对不允许的。

解决办法：

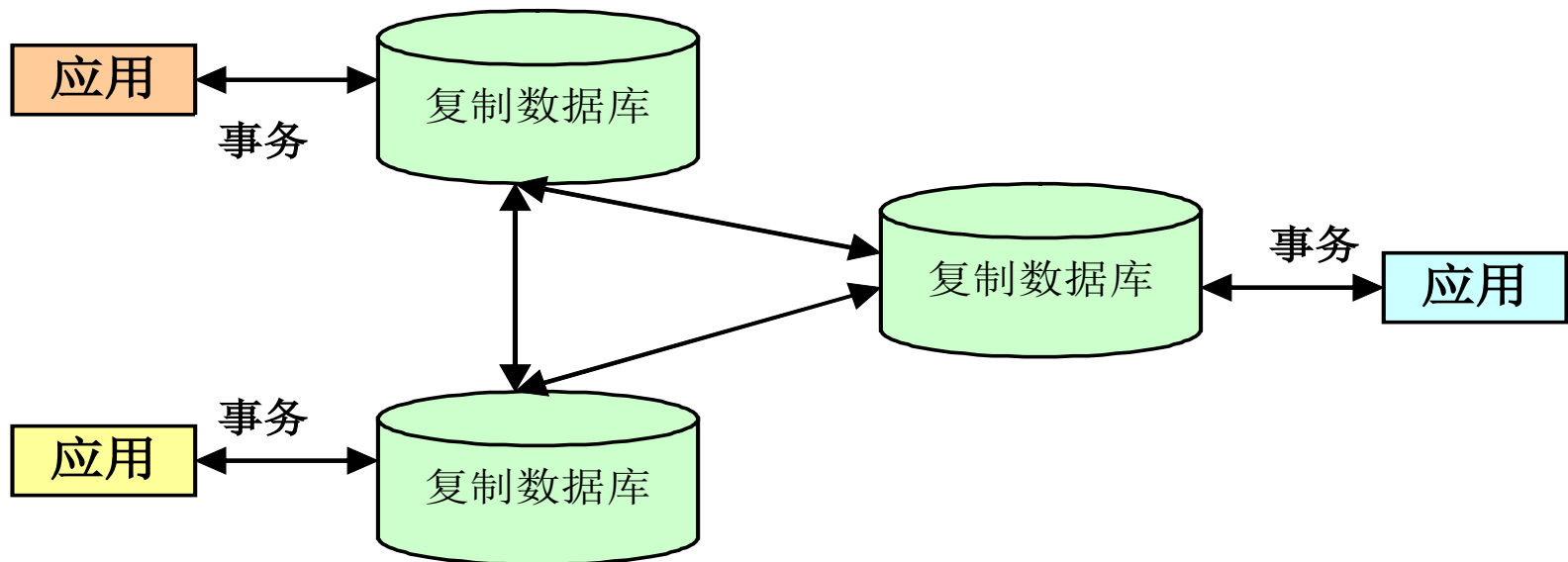
1、数据库镜像：将整个数据库或其中的关键数据同时存放在两个分离的物理磁盘上。每当主数据库更新时，DBMS自动把更新后的数据复制到另一个磁盘上，从而自动保证主数据库与镜像数据库的一致性。

但镜像的内容可选，如只是事务日志，或服务器上所有内容，等等。

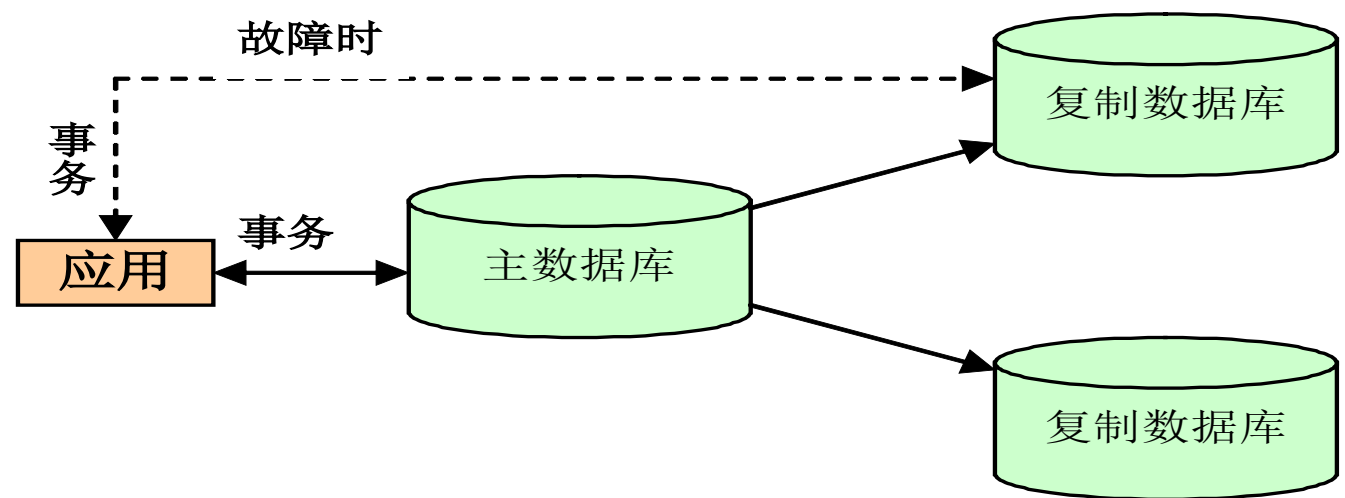
DBMS级的镜像技术

数据库复制的三种方式：对等(peer-to-peer)复制、主从(master/slave)复制和级联(cascade)复制。

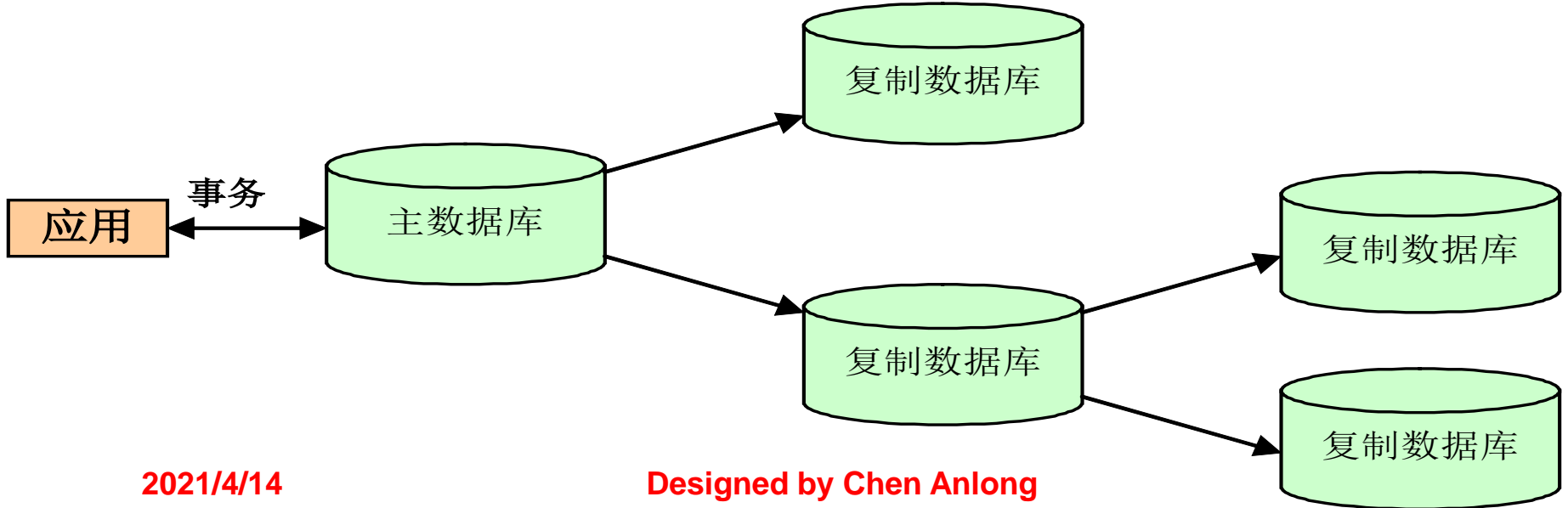
对等复制：是理想的复制方式。各场地DB地位平等，可互相复制数据。用户可在任一场地读取/更新公共数据，DBMS应将更新数据复制到所有副本。



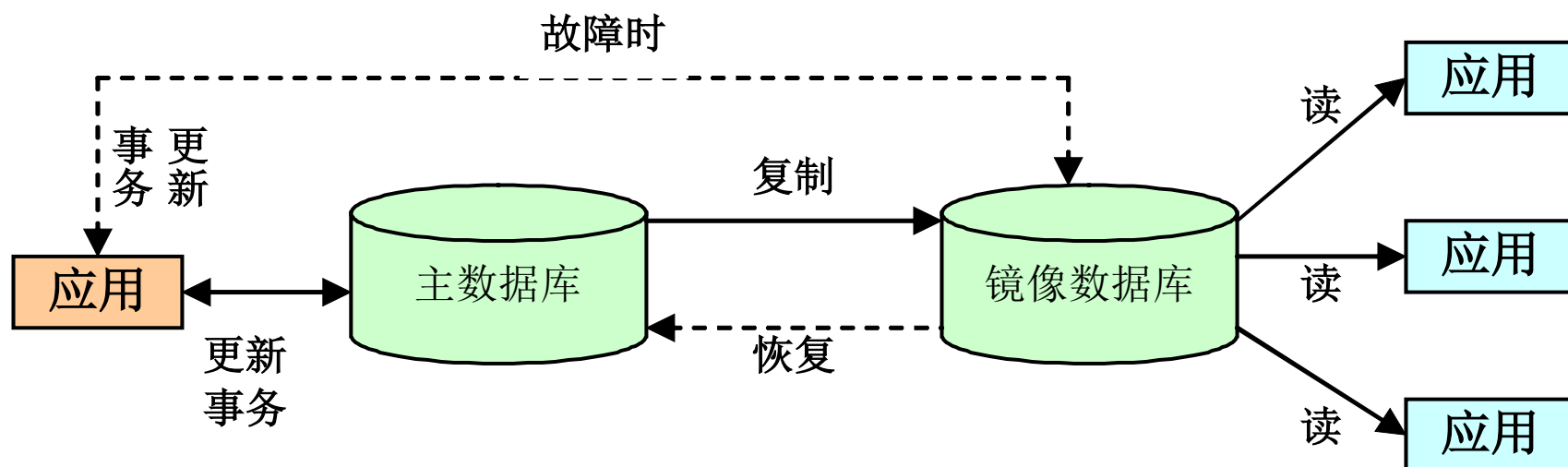
主从复制：数据从主DB复制到从DB。只能主DB更新数据，从DB供用户读数据。当主场地出现故障时，应用可转到某从BD。



级联复制：从主DB复制来的数据又从该DB复制到其他DB。级联复制可平衡当前各种数据需求对网络流量的压力。



双数据库镜像: DBMS为避免磁盘介质故障，提供了数据库镜像，由DBMS根据DBA要求，自动将整个DB或其中关键数据复制到另一个磁盘上。当主DB更新时，DBMS自动将复制更新后的数据。出现介质故障时，可由镜像磁盘提供应用服务，并进行数据库恢复。



设备级的数据库恢复技术

方法：利用多副本互为备份、恢复。

NOS/OS级的可靠性技术：镜像磁盘（Mirroring Disk）、双工磁盘（Duplex Disk）、镜像服务器（Mirroring Server）、群集（Cluster）系统。

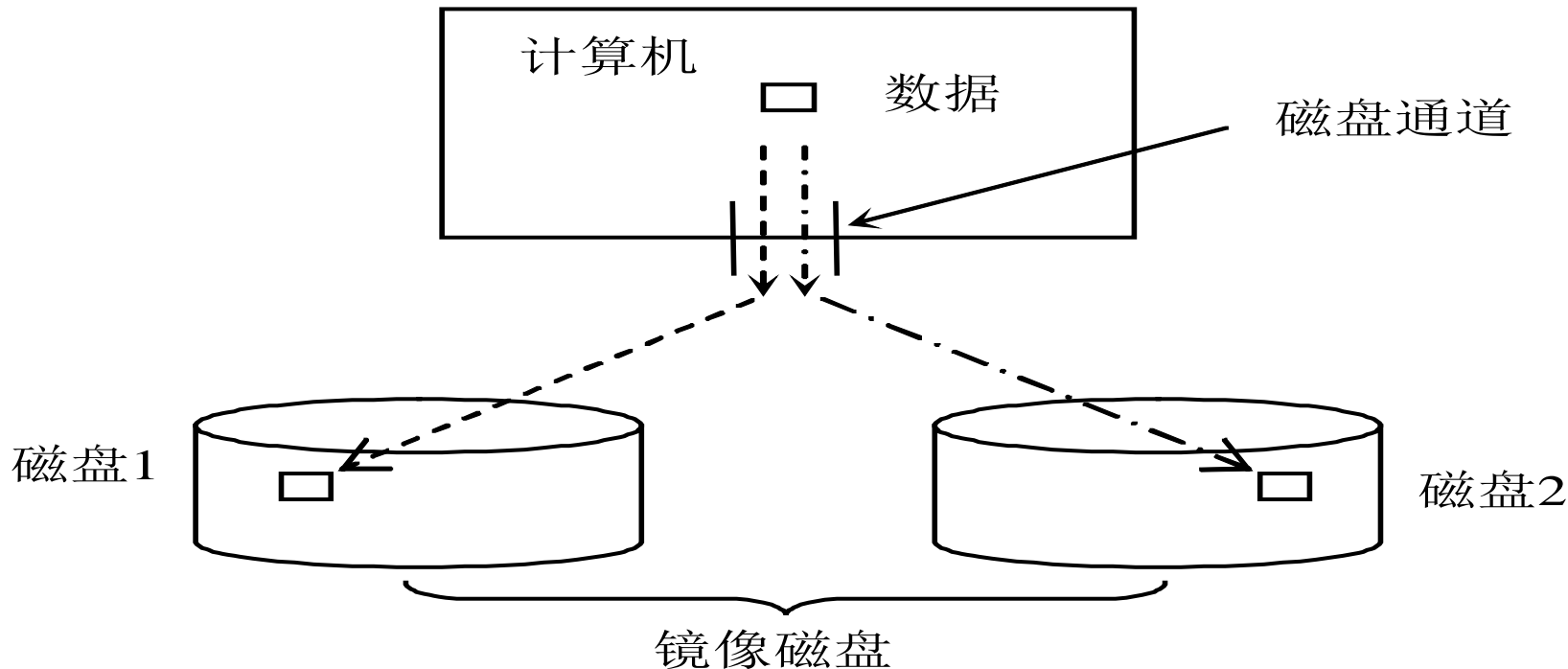


图 磁盘镜像

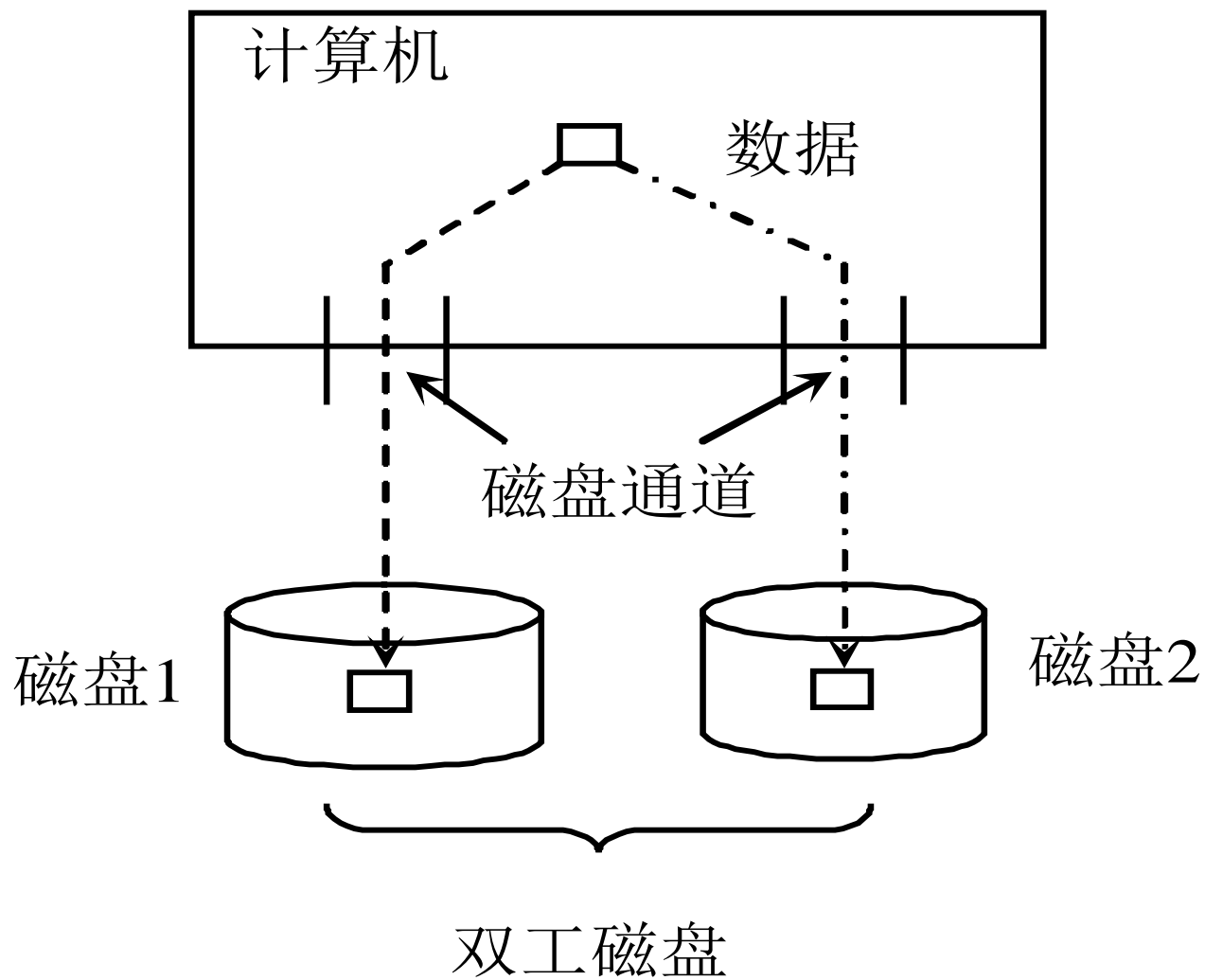


图 磁盘双工

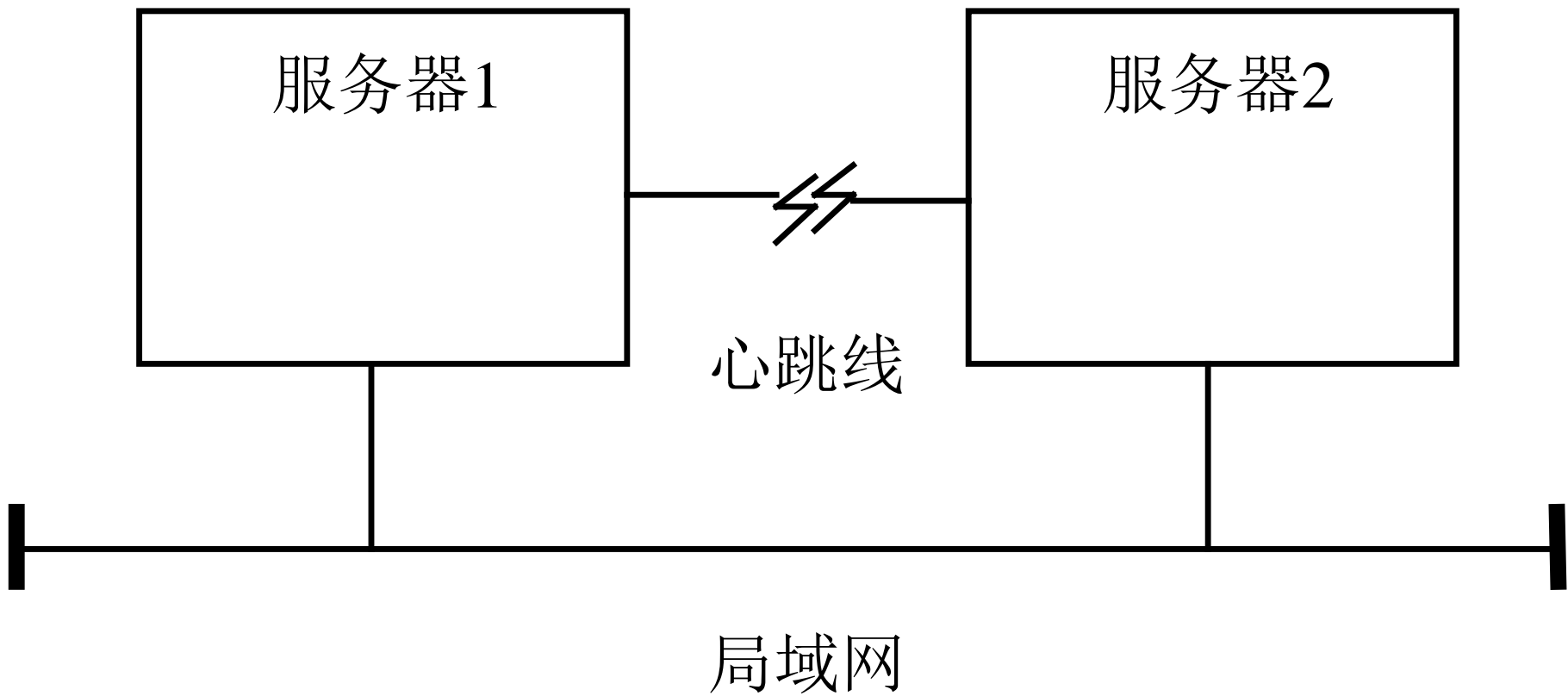


图 服务器镜像

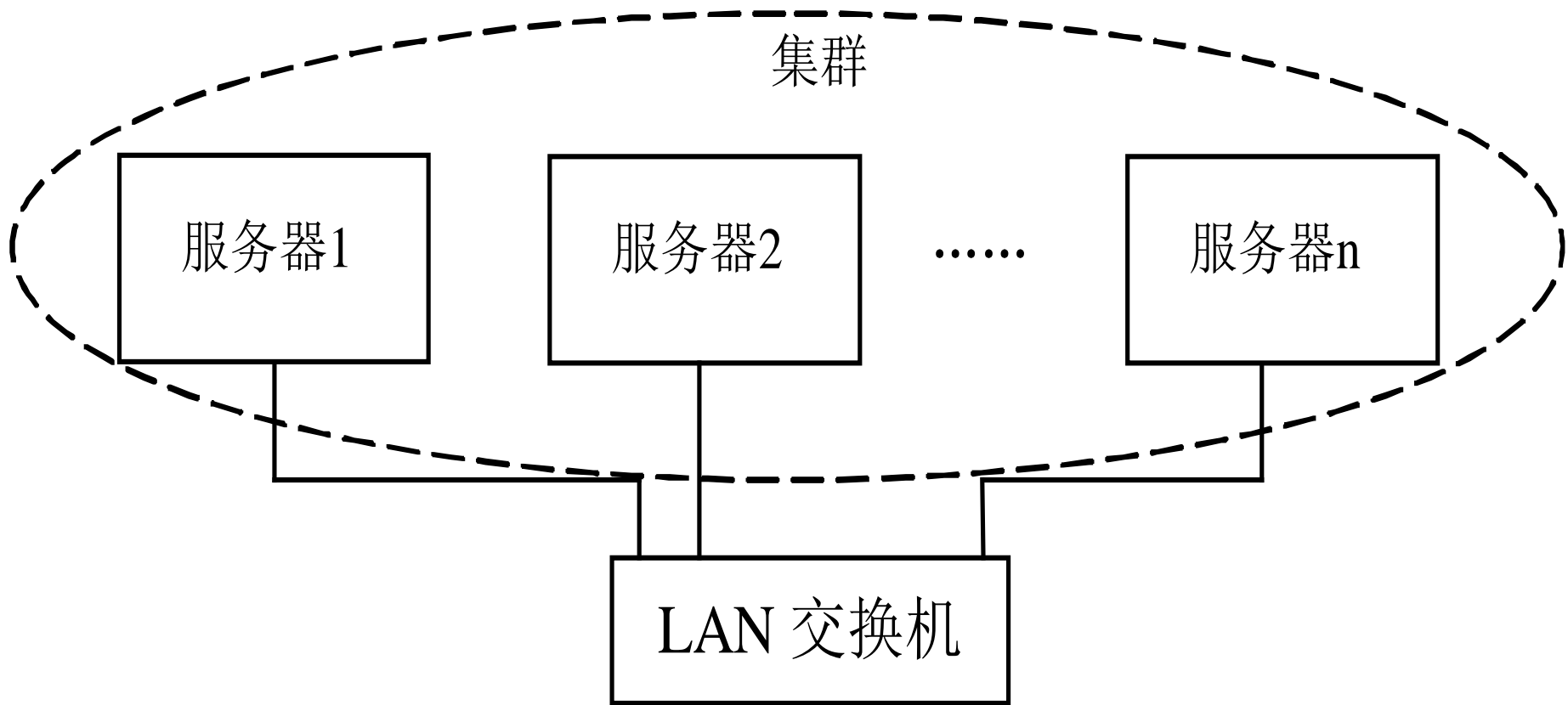


图 集群服务器

数据库镜像的优缺点

优点：可提高数据库的可用性。

- ① 在介质故障时，不需关闭系统和重装后援副本，保证“不间断”地恢复；
- ② 便于并发操作，当主数据库的某个对象被加排它锁时，其它应用可以读镜像数据库。

缺点：

- ① 由于频繁地复制数据，会降低系统的运行效率；
- ② 使用更多的磁盘设备。

PostgreSQL备份与恢复数据库--SQL转储

(1) pg_dump备份数据库

pg_dump是 PostgreSQL客户端应用程序，须具有对要备份的**所有表的读访问权限**；pg_dump只备份某个数据库的数据，它**不会导出角色和表空间相关的信息**，因为这些信息是整个数据库集群共用的，不属于某个单独的数据库。

① 首先使用下列命令进入postgresql的bin目录

```
cd \program files\PostgreSQL\12\bin
```

② 执行下列命令完成数据库备份

```
pg_dump -h localhost -U postgres -p 5432 -d coursedb -c -C  
-f f:\databackup\coursedb.backup
```

```
C:\Program Files\PostgreSQL\12\bin>pg_dump -h localhost -U postgres -p 5432 -d coursedb -c -C -f f:\databackup\coursedb.backup
```

□令：

pg_dump命令参数

- h: 数据库服务器地址;
- p: 数据库端口号;
- U: U 大写,表示用户名;
- d: 数据库名称;
- f: 备份文件路径以及备份文件名称;
- c: 在重新创建之前,先删除数据库对象;
- C: 在转储中包括命令,以便创建数据库;
- t: 只转储指定名称的表;
- w: 永远不提示输入口令
- W: 强制口令提示(自动)

(2) 使用psql命令恢复数据库

psql是postgresql数据库提供的连接数据库shell命令：

psql -h IP地址 -p 端口 -U 数据库名 -f 需执行的文件

例如： **psql -h 127.0.0.1 -U postgres -p 5432 -f f:\databackup\coursedb.backup**

例如: `psql -h 127.0.0.1 -U postgres -p 5432 -f f:\databackup\coursedb.backup`

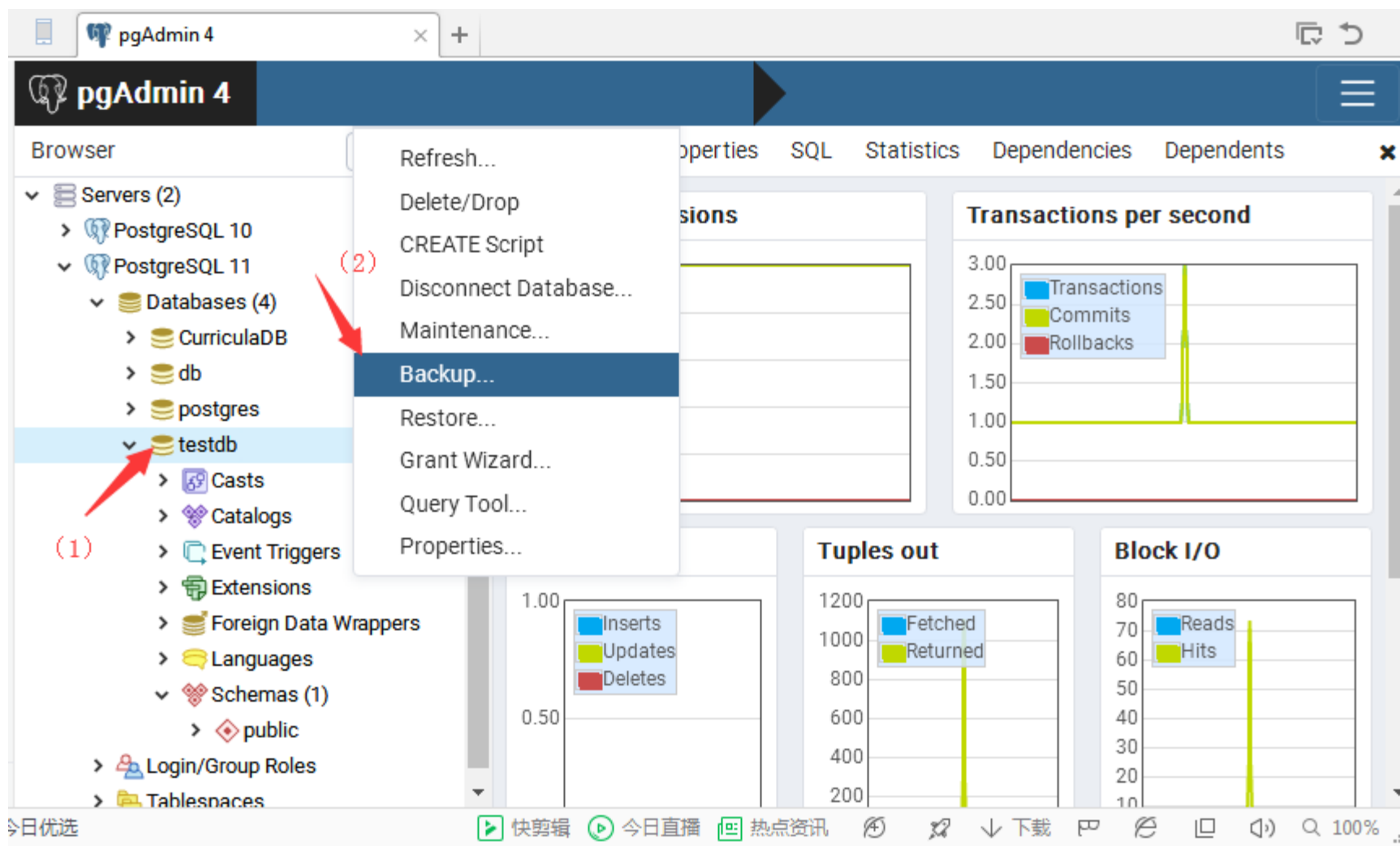
```
C:\Program Files\PostgreSQL\12\bin>psql -h 127.0.0.1 -U postgres -p 5432 -f f:\databackup\coursedb.backup
用户 postgres 的口令:
SET
SET
SET
SET
SET
SET
set_config
-----
(1 行记录)

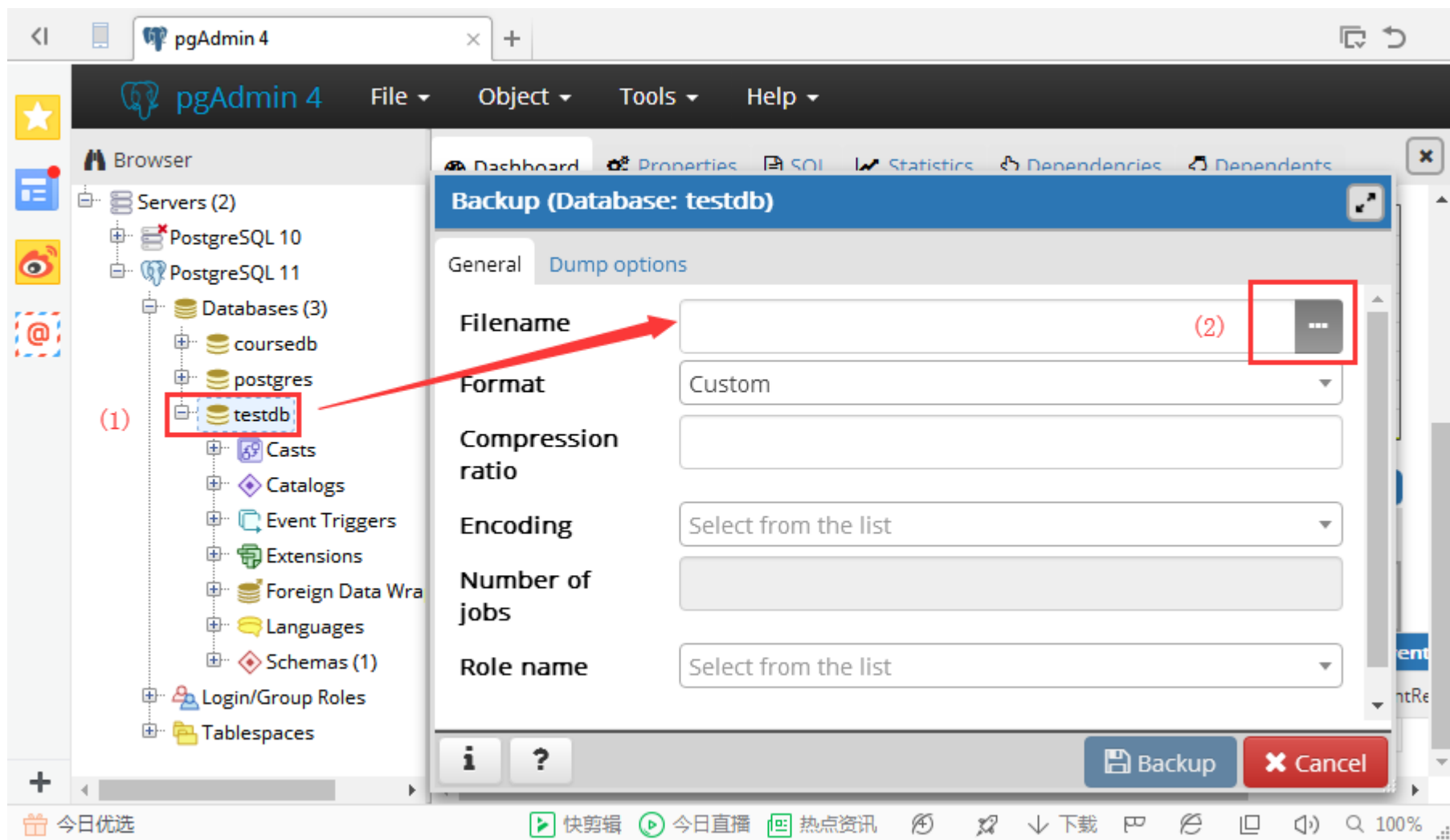
SET
SET
SET
SET
DROP DATABASE
CREATE DATABASE
ALTER DATABASE
您现在已经连接到数据库 "coursedb", 用户 "postgres".
SET
SET
SET
SET
SET
set_config
-----
(1 行记录)

SET
SET
SET
SET
CREATE FUNCTION
ALTER FUNCTION
SET
SET
CREATE TABLE
ALTER TABLE
COPY 0
CREATE TRIGGER

C:\Program Files\PostgreSQL\12\bin>
```


(4) 使用pgAdmin4实现postgresql数据库备份







Servers (2)

> PostgreSQL 10

> PostgreSQL 11

Databases (4)

> CurriculaDB

> db

> postgres

testdb

> Casts

> Catalogs

> Event Triggers

> Extensions

> Foreign Data W

> Languages

Schemas (1)

> public

> Login/Group Roles

> Tablespaces

Select file



E:\dbbak\testdb.backup



Name

Size

Modified

coursedb.backup

0.0 B

Mon Apr 15 12:29:42 2019

CurriculaDB.backup

3.7 KB

Mon Apr 15 13:15:57 2019

db.backup

868.0 B

Mon Apr 15 13:12:33 2019

testdb.backup

10.8 KB

Mon Apr 15 13:45:39 2019

Show hidden files and folders? ☐

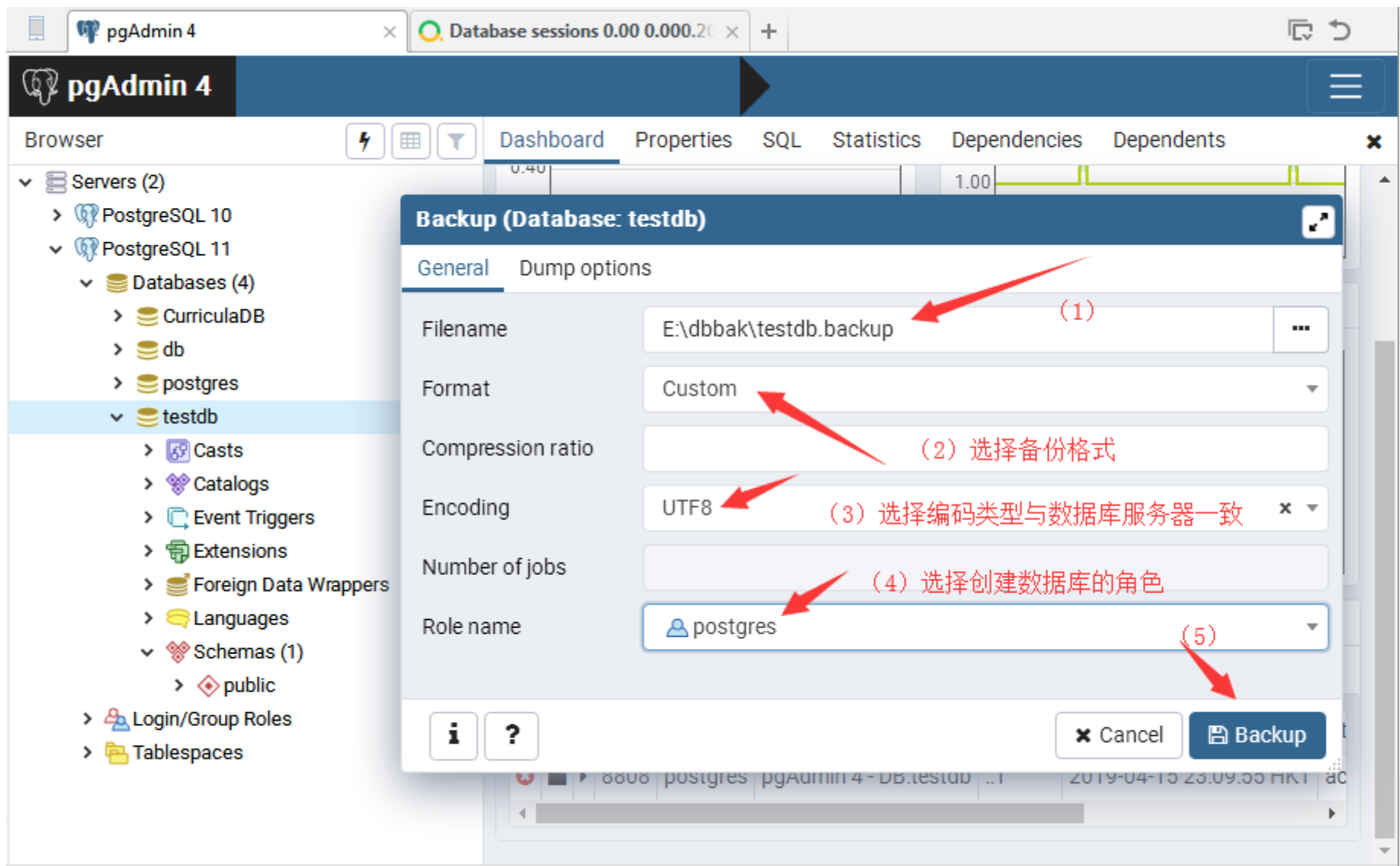
Format

backup

Cancel

Create





Browser



Dashboard

Properties

SQL

Statistics

Dependencies

Dependents



Servers (2)

> PostgreSQL 10

> PostgreSQL 11

Databases (4)

> CurriculaDB

> db

> postgres

> testdb

> Casts

> Catalogs

> Event Triggers

> Extensions

> Foreign Data Wrappers

> Languages

> Schemas (1)

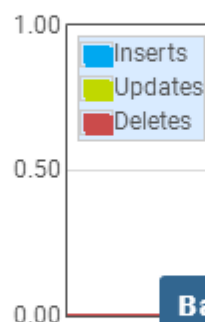
> public

> Login/Group Roles

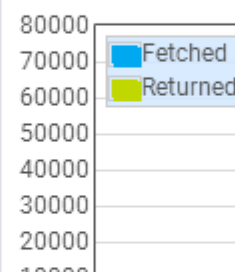
> Tablespaces



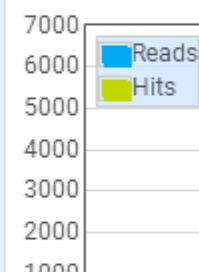
Tuples in



Tuples out

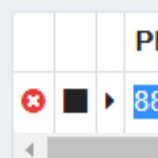


Block I/O



Server activity

Sessions



Backing up an object on the server



Backing up an object on the server 'PostgreSQL 11 (localhost:5433)' from database 'testdb'

Mon Apr 15 2019 23:31:45 GMT+0800 (China Standard Time)

⌚ 1.35 seconds

OK

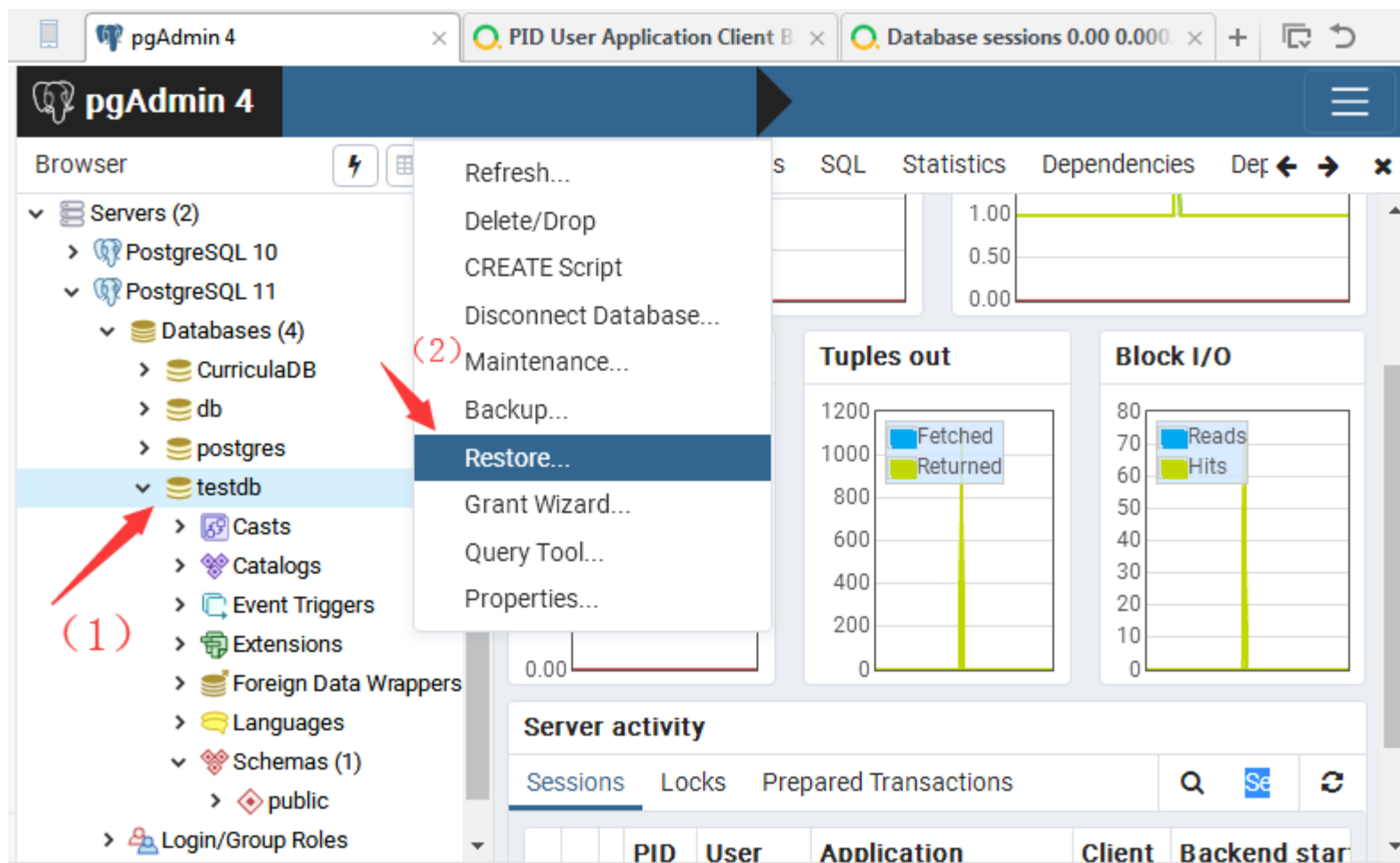
ⓘ More details...

⌛ Stop Process



Successfully completed.

使用pgAdmin4实现postgresql数据库恢复



请务必注意：需要恢复的数据库一定是新建的空数据库，否则有很大可能会报错！



Servers (2)

> PostgreSQL 10

> PostgreSQL 11

Databases (4)

> CurriculaDB

> db

> postgres

> testdb

> Casts

> Catalogs

> Event Tri

> Extensio

> Foreign D

> Languag

> Schemas (1)

> public

> Login/Group Roles

Restore (Database: testdb)

General

Restore options

Format

Custom or tar

Filename

...

Number of jobs

Role name

Select from the list



X Cancel

Restore

Sessions

Locks

Prepared Transactions



Se



PID

User

Application

Client

Backend star



Restore (Database: testdb)

General

Restore options

Format

Custom or tar

Filename

E:\dbbak\testdb.backup

Number of jobs

Role name

postgres



X Cancel

Restore

Browser



Dashboard

Properties

SQL

Statistics

Dependencies

Depend



Servers (2)

> PostgreSQL 10

> PostgreSQL 11

Databases (4)

> CurriculaDB

> db

> postgres

> testdb

> Casts

> Catalogs

> Event Triggers

> Extensions

> Foreign Data Wrappers

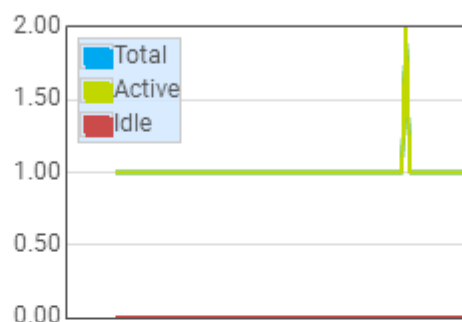
> Languages

> Schemas

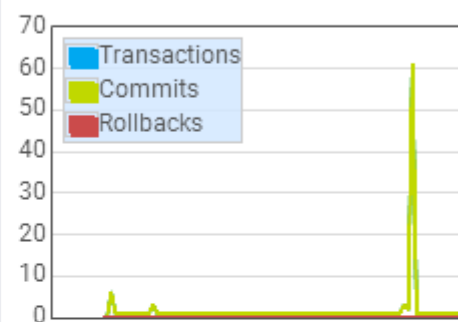
> Login/Group Roles

> Tablespaces

Database sessions



Transactions per second



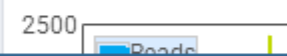
Tuples in



Tuples out



Block I/O



Restoring backup on the server

Restoring backup on the server 'PostgreSQL 11 (localhost:5433)'

Mon Apr 15 2019 23:52:46 GMT+0800 (China Standard Time)

🕒 0.68 seconds

OK

🔍 More details...

⏹ Stop Process



Successfully completed.

事务的并发执行

在多用户数据库系统中，当多个用户并发存取数据库时就会产生多个事务同时存取同一数据的情形。若不加控制，可能会存取和存储不正确的数据，造成数据库的不一致性。

在并发操作情况下，对事务的操作序列的调度是随机的，考虑飞机订票系统，若按下面的序列调度：

考虑飞机订票系统中的一个活动序列：

- 甲售票点读出某航班的机票余额A，设 $A=16$ ，
- ✕ 乙售票点读出同一航班的机票余额A，也为16，
- ✕ 甲售票点卖出一张机票，修改余额 $A \leftarrow A-1$ ，
A变为15，把A写回数据库
- ✕ 乙售票点也卖出一张机票，修改余额 $A \leftarrow A-1$ ，
A也为15，把A写回数据库。

卖出两张机票，而余额只减少1。错误！**X**

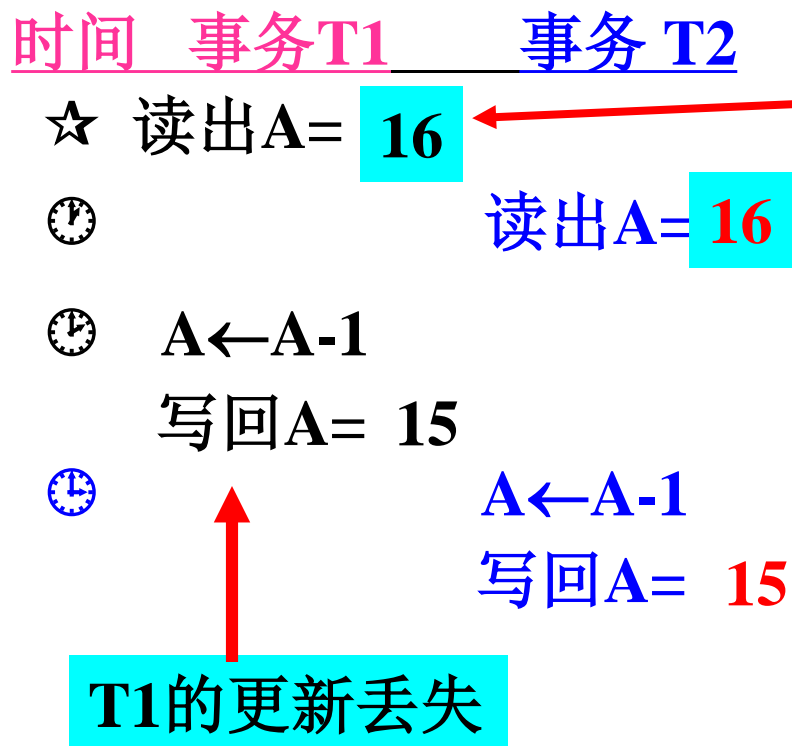
这种情况就造成数据库的不一致性，这种不一致性是由并发操作引起的。

并发操作带来的数据不一致性包括三类：

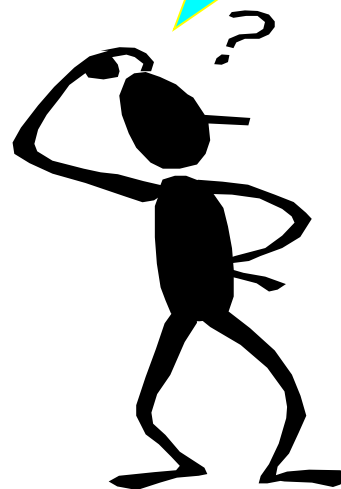
一、并发操作可能造成的不一致性

1、丢失更新：

两事务读出同一数据并修改，先写回的数据更新丢失



我的数据呢？



2、不可重复读

事务T1读取某一数据，事务T2读取并修改了同一数据；
事务T1为了对读取值进行校对再读此数据，得到了不同的结果。

<u>时间</u>	<u>事务T1</u>	<u>事务 T2</u>
☆	读出A=50, B=100 求和 =150	
🕒		读出B=100 计算 $B \leftarrow B \times 2$, 写回B
🕒	读出A=50, B= 200 求和 = 250	

T1读出B的值与原来的不符，验算结果不对

有三种情况可造成不可重复读

(1) 事务T1读取某一数据后，事务T2对其做了修改，事务T1再次读取该数据时，发现与前次不同；

(2) 事务T1按一定条件读取了某些数据记录后，事务T2删除了其中的部分记录，事务T1再次按相同条件读取记录时，发现有些记录不存在；

(3) 事务T1按一定条件读取了某些数据记录后，事务T2插入了一些记录，事务T1再次按相同条件读取记录时，发现多了一些记录。

3、读出“脏”数据

事务T1修改某一数据，事务T2读取同一数据；

事务T1由于某种原因被撤消，则T2读到的就是“脏”数据。



产生上述三类不一致性的主要原因就是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，使某个事务的执行不受其它事务的干扰。

必须对并发操作进行控制

并发控制的技术是封锁，即事务在修改某个对象前，先锁住该对象，不允许其它事务读取或修改该对象，修改完毕或事务完成后再将锁打开。

一.数据库系统中的并发

事务串行(顺序)执行：一个事务完全结束后才开始另一事务。

并发访问：多个事务同时执行，即各事务中的操作可交错执行。

操作的交错执行：需要调度（Schedule）。

二.调度概念

调度：安排多个事务操作的执行序列。

事务调度的要求：事务内部操作的调度顺序应该与它们在事务中的顺序一致。

事务结束：COMMIT和ROLLBACK

调度的表示：

$$S = R(A)W(A)R(C)W(C)R(B)W(B)$$

T1	T2
Read(A)	
Write(A)	
	Read(C)
	Write(C)
	Commit
Read(B)	
Write(B)	

涉及两个事务的并发调度

调度序列的符号表示

■ 简记符号

- ✓ **WRITE**事务写操作，简记为**W**，
- ✓ **READ**事务读操作，简记为**R**，
- ✓ $W_T(X)$ ：事务**T**写数据库元素**X**，
- ✓ $R_T(X)$ ：事务**T**读数据库元素**X**，
- ✓ **S**表示一个调度。

■ 调度(事务序列)表示：

$S = R1(A) R2(A) W1(A) W2(A) R2(B) R1(B) W2(B) W1(B)$

三.并发的目的（与串行执行比较）

- (1) 改善系统的资源利用率;
- (2) 改善短事务的响应时间。



串行调度

- 串行调度：不同事务的活动在调度中是一个接一个执行的，没有交叉的运行。

T1	T2
READ(A) A:=A+A*0.1 WRITE(A) READ(B) B:=B+B*0.2 WRITE(B)	 READ(A) A:=A+10 WRITE(A) READ(B) B:=B-20 WRITE(B)

T1	T2
 READ(A) A:=A+A*0.1 WRITE(A) READ(B) B:=B+B*0.2 WRITE(B)	READ(A) A:=A+10 WRITE(A) READ(B) B:=B-20 WRITE(B)

- 两个串行调度的结果不同。但只要保持了数据库的一致性，最终的结果并不重要

可串行化调度

由于并发执行可能产生上述问题，什么样的调度序列的执行结果是正确的呢？

概念：对一事务集，如一个并发调度与一个串行调度等价，则称此并发调度是可串行化的(**Serializable**)。

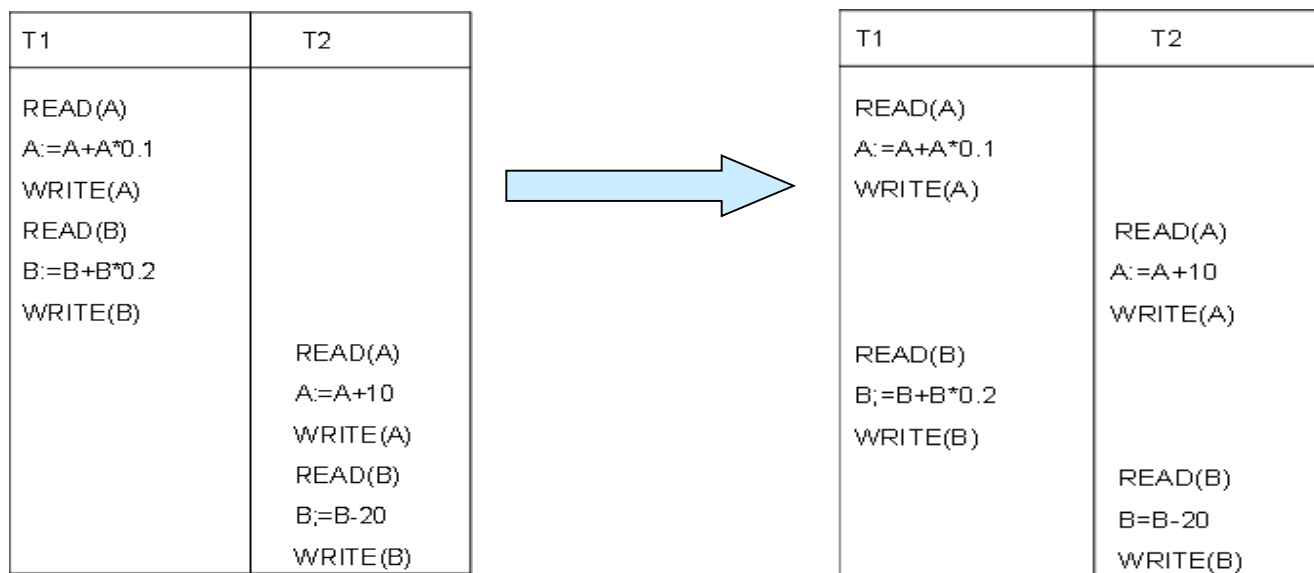
调度等价：如有两个调度S1和S2，在DB的任意相同初始状态下，所有读出的数据都是一样的，留给DB的最终状态也是一样的，则称S1和S2是等价 (**Equivalence**) 的。

注意：① 对n个事务有n! 种串行调度。每个串行调度执行结果可能不一样，可串行化只要求调度和其中某一个串行调度等价。

② 不同的可串行化调度是不一定等价的。

可串行化调度

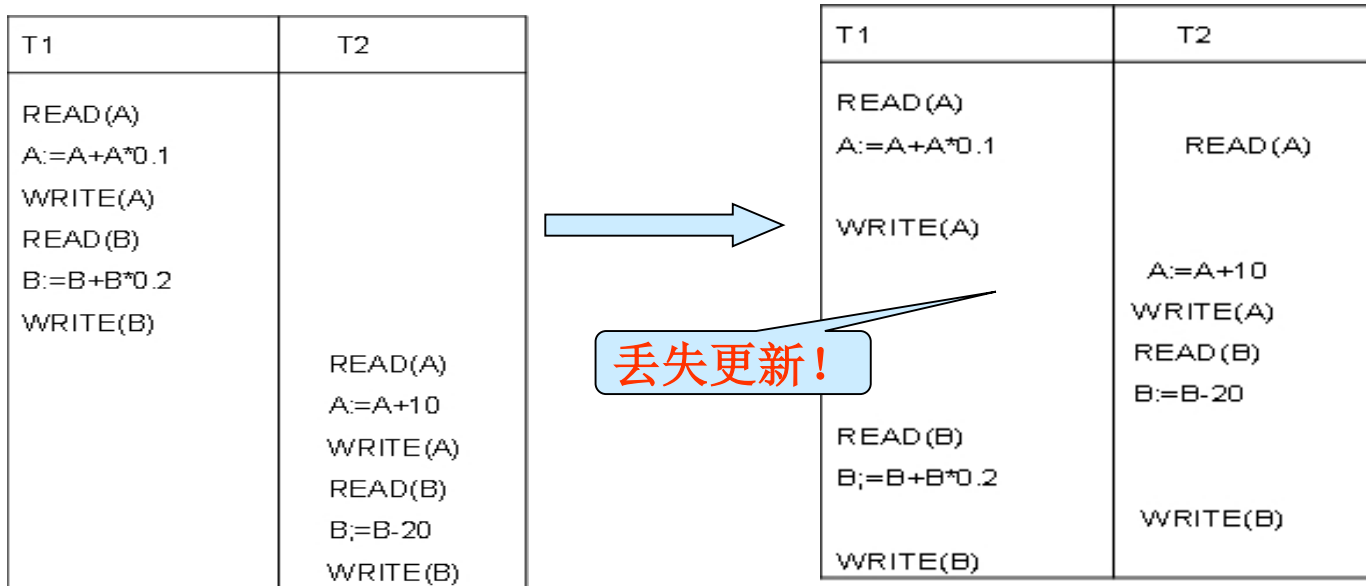
- 调度是可串行化的：多个事务交叉调度的结果与某一个串行调度的结果相同
- **DBMS**认为事务串行调度的结果保持了数据库的一致性，都是正确的
- 一个调度如果是可串行化的，系统认为其调度是一个正确的调度，保持了数据库的一致性



➤ 并行调度与串行调度的结果相同，因此该调度是可串行的调度

不可串行化调度

对于由多个事务组成的调度序列，如果这些事务的**任何串行调度的结果**都不与该调度序列的结果相等，则该调度是不可串行化的调度。



DBMS需要事务调度管理

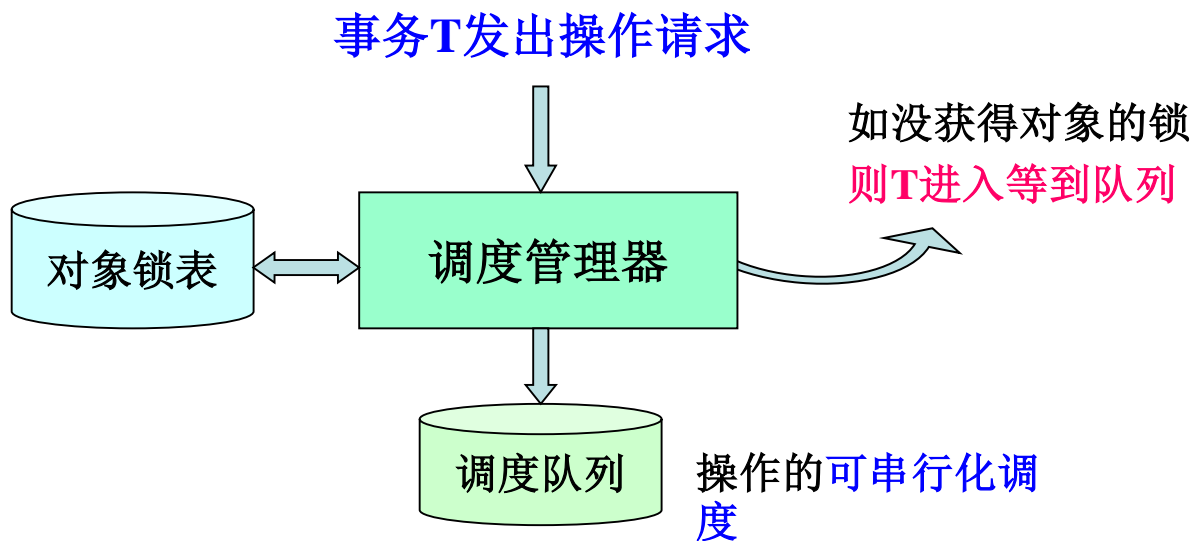
- 如果将事务的并发执行完全交给操作系统，则任何一种调度方式都有可能出现。
- 有的调度能保持数据库的一致，有的调度却会产生错误的结果。
- **DBMS**必须对事务的运行加以控制，确保交叉调度完毕后的结果与某一串行调度的结果相同，数据库不会出现不一致的状态。

基于锁的并发控制技术

为了实现可串行化调度，需要制定某种规则（协议），只要事务按照该协议调度执行，就可以保证调度可串行化。

现有数据库主要采用加锁方式来保证调度执行的可串行化

即对敏感数据的读写进行某种限制，如同加了一把锁。



锁的基本类型

- 排它锁（eXclusive，也称 X锁，用于写操作）

某数据对象在没有加任何锁的情况下，一个事务可以对其加X锁，而其他事务就不能对其再加任何锁。

- 共享锁（Share，也称 S锁，用于读操作）

一个事务对某数据对象加了S锁后，其他事务就不能对其加X锁，但可以加S锁。

加锁的相容表

T2申请锁	T1 \ T2	X锁	S锁	无
	X锁	否	否	是
	S锁	否	是	是

是：可以加锁

否：不能加锁

(1) 一级封锁协议

任一事务在写某数据前，必须对其加上X锁，该事务结束后才释放。
不采用S锁，读数据不用加锁。

S1		S2		S3	
T1	T2	T1	T2	T1	T2
Xlock(A)		Xlock(A)		Read(A)	
Write(A)		Write(A)			Xlock(A)
	Xlock(A)被拒绝		Read(A)		Read(A)
Commit			Xlock(B)		A:=A*2
Ulock(A)			B:=A+10		Write(A)
	Xlock(A)		Write(B)	Read(A)	
	Write(A)		Commit		Commit
	Commit		Ulock(B)		Ulock(A)
	Ulock(A)	Rollback		

消除了丢失更新

产生读脏数据

产生不可重复读

(2) 二级封锁协议

满足一级封锁协议，且任一事务在读取某数据前，必须对其加上S锁，读完后就释放。

S2	
T1	T2
Xlock(A)	
Write(A)	
	Slock(A)被拒绝
Commit	
Unlock(A)	
	Slock(A)
	Read(A)
	B:=A+10
	Unlock(A)
	Xlock(B)
	Write(B)
	Commit
	Unlock(B)

避免了读脏数据

S3	
T1	T2
Slock(A)	
Read(A)	
Unlock(A)	
	Xlock(A)
	Read(A)
	A:=A*2
	Write(A)
	Commit
	Ulock(A)
Slock(A)	
Read(A)	
Unlock(A)	

原因：过早释放S锁

产生不可重复读

(3) 三级封锁协议

满足一级封锁协议，且任一事务在读取某数据前，必须对其加上S锁，事务结束后就释放。

S3	
T1	T2
Slock(A)	
Read(A)	Xlock(A)被拒绝
Read(A)	
Commit	
Unlock(A)	
	Xlock(A)
	Read(A)
	A:=A*2
	Write(A)
	Commit
	Ulock(A)

该协议可以解决丢失更新、读脏数据以及不可重读的问题。

不同级别锁协议比较

加锁协议级别	排它锁	共享锁	不丢失更新	不脏读	可重复读
一级	全程加锁	不加	是	否	否
二级	全程加锁	开始时加锁，读完数据释放锁定	是	是	否
三级	全程加锁	全程加锁	是	是	是

(4) 两阶段加锁协议

两阶段加锁事务：在一个事务中，如果**加锁**都在**所有释放锁**之前，则称这样的事务加锁规则称为两阶段加锁协议(**2PL**协议)。

(1) 两阶段事务中，**加锁阶段**为其拥有的锁逐步**增长**的阶段，而**解锁阶段**为其拥有的锁逐步**缩减**的阶段，故称两阶段事务。

(2) 两阶段事务中，如**开始解锁**，就**不能再**对任何对象**加锁**。

(3) 如事务遵循**两阶段加锁**协议，同时遵循**先加锁后操作**原则，则该调度是**冲突可串行化**的。

T1	T2
XLock (A) Read (A) $A := A + A * 0.1$ Write (A) XLock (B) UNLock (A) Read (B) $B := B + B * 0.2$ Write (B) UNLock (B)	 XLock (A) Read (A) $A := A + 10$ Write (A) XLock (B) UNLock (A) Read (B) $B := B - 20$ Write (B) UNLock (B)

■ 两段锁协议的级联回滚现象

T1	T2	T3
XLock (A) Read (A) $A := A + A * 0.1$ Write (A) XLock (B) UNLock (A) Read (B)	 XLock (A) Read (A) $A := A + 10$ Write (A) UNLock (A)	 SLock (A) Read (A)
$B := B + B * 0.2$ Write (B) UNLock (B)		

- ① 每个事务都遵从两段锁协议;
- ② 若T1事务在WRITE(B)时刻发生故障, 将导致事务T2、T3级联回滚。
- ③ 主要原因是锁的过早释放

◆ **严格两阶段锁：**除要求满足两段锁协议规定外，还要求事务的**排它锁**必须在事务**提交之后**释放。

解决级联回滚问题；避免了脏读和丢失更新的问题。

思考：可能出现不可重复读吗？ Why?

◆ **强两阶段锁：**除要求满足两段锁协议规定外，还要求事务的**所有锁**都必须在事务**提交之后**释放。

进一步解决了不能重复读的问题

(5) (S,U,X) 协议

(S, U, X) 协议：除S, X两种锁外，又增加了U锁，即更新锁。

增设U锁原因：事务在做更新时，分两步：先读后写。即先读老内容，在内存中修改后；然后再写入修改后的内容。除最后的写入阶段外，为保证与其他事务共享数据，而新增更新锁。

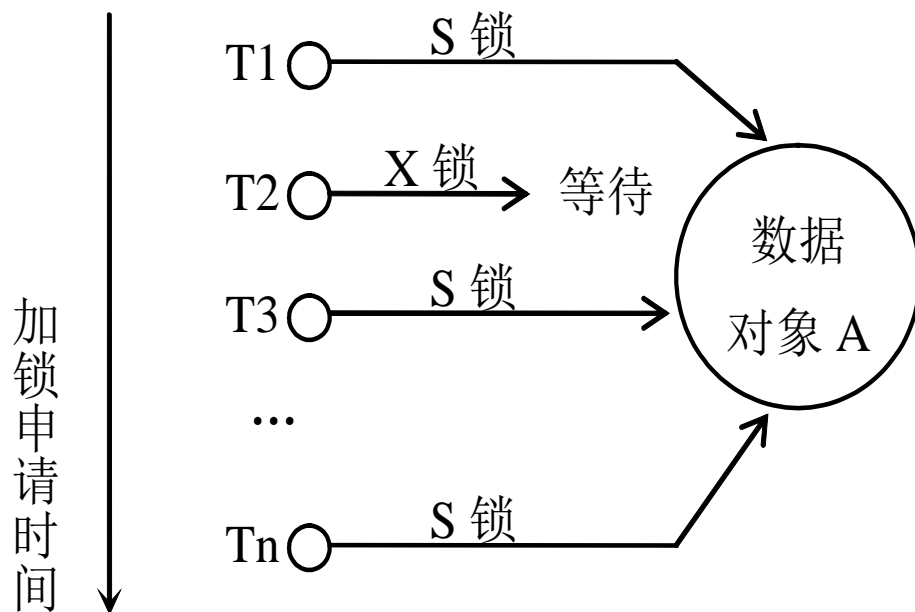
U锁的使用及作用：事务要更新数据对象时，先申请该对象的U锁。对象加了U锁，允许其他事务对它加S锁。在最后写入时，再申请将U锁升级为X锁。不必在全过程中加X锁，可提高并发度。

T2申请锁	T2 \ T1	S锁	X锁	U锁	无
	S锁	是	否	是	是
	X锁	否	否	否	是
	U锁	是	否	否	是

是：可以加锁 否：不能加锁

活锁与死锁

- **活锁**：当多个事务同时申请对数据的封锁时，由于选择策略的问题导致**长时间甚至永远的等待**，称为**活锁**。



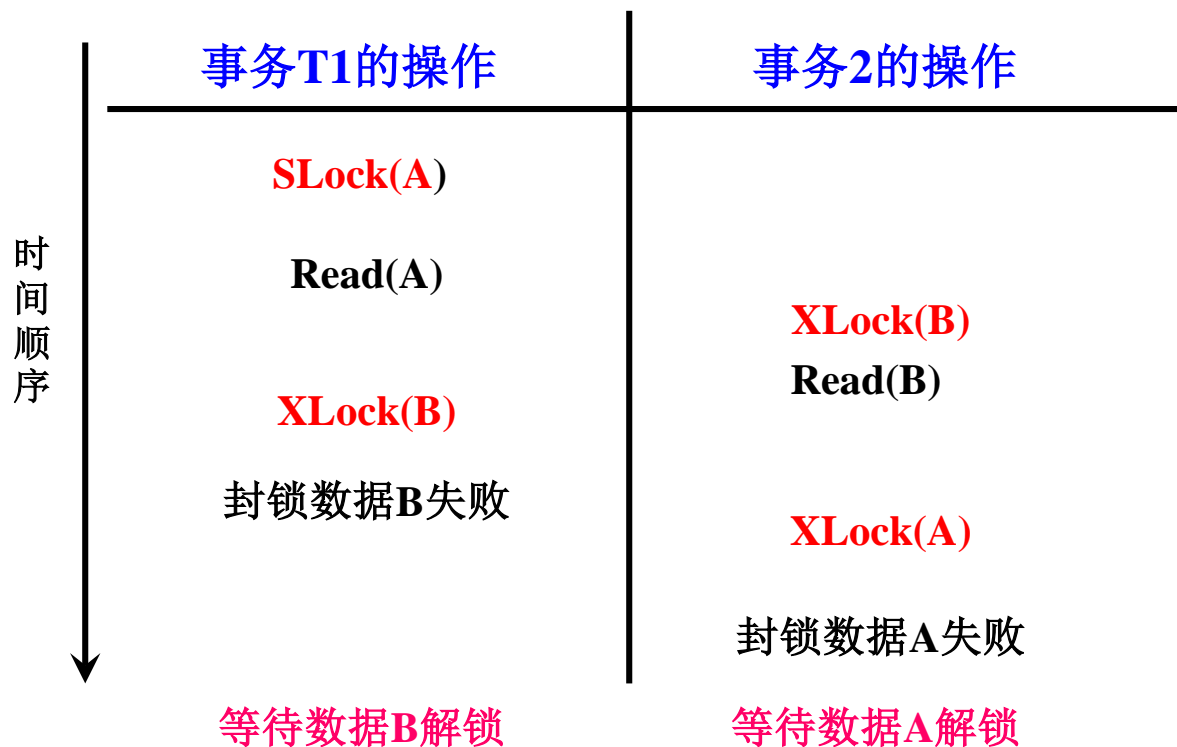
注： T_i ($i=1,2,\dots,n$) 表示第 i 个事务

- 活锁的解决方法

采用**先来先服务**（即队列方式）并结合优先级的选择策略。

- **死锁**：在多个事务竞争共享资源的情况下，出现的相互永远等待的封锁请求，若无外力作用，这些事务永远不能向前推进，称为**死锁**。

例如：事务T1 和事务T2 并发执行，且都需要数据A 和数据B，假设数据A 和数据B 初始时为无锁状态。



最后结果：事务T1 和 T2 相互僵持，永远等待。

- ① 预防死锁
- ② 检测死锁
- ③ 消除死锁

(6) 多粒度加锁协议

问题提出：在数据库中，一个数据对象的大小可以大到整个数据库、一页、一个表，也可以小到一行。对加锁来说，数据对象的大小即是加锁的粒度。

粒度、控制及并发度间关系：加锁单位愈大，加锁起来愈简单，但同时会降低并发度。反之，加锁单位越小，往往需要加很多的锁，所需的控制越复杂，不过并发度可提高。

实际需要：实际应用中，有时要访问大片数据，而有时只访问个别数据，为此，可提供多级加锁单位，根据应用需要加以选用，此即：多粒度加锁(Multiple Granularity Locking)，它使得对包含有其它对象的对象加锁更加有效。

实际RDBMS做法：现代大型DBMS一般均支持多粒度加锁，而在微机DBMS中，并发度要求不高，一般以表作为加锁单位，此即：单粒度加锁(Single Granularity Locking)。

RDBMS提供的多粒度锁（从小到大排列）：

- ① 行级锁，或称行锁；
- ② 页级锁，或称页锁；
- ③ 表级锁，或称表锁。

2.多粒度加锁的实现

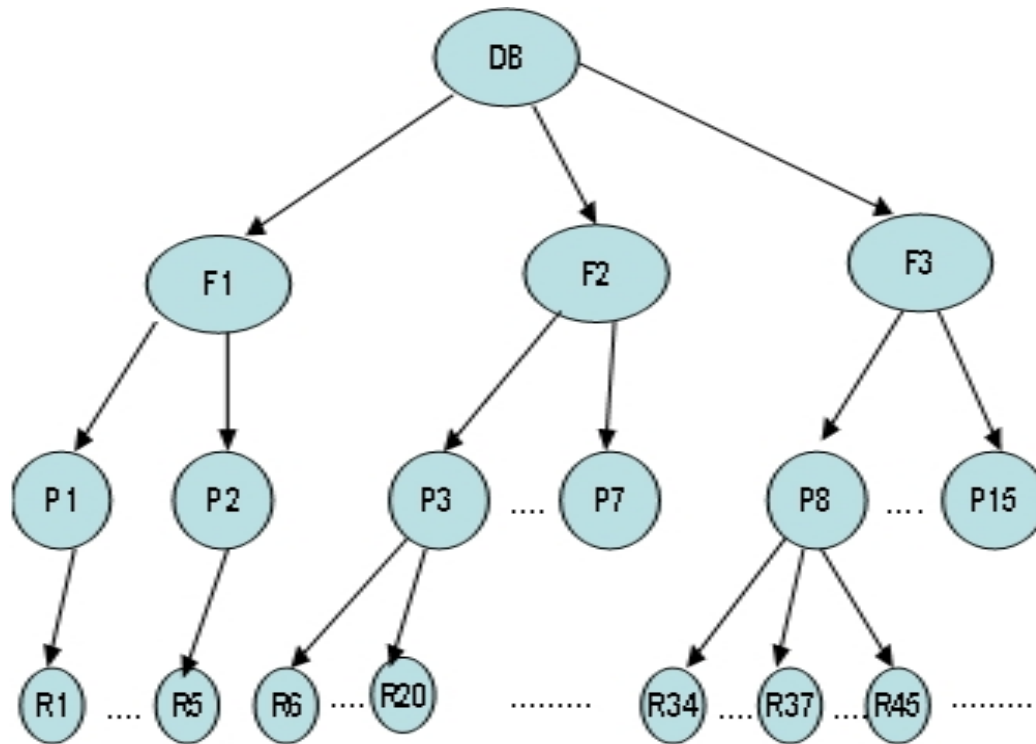
1)锁冲突检测问题

数据对象的祖先与子孙关系：按数据对象大小（或上下级）关系，区分为祖先（Ancestor）与子孙（Offspring）。

多粒度加锁下，一个数据对象可能的两种加锁方式：

- ① **显式加锁(Explicit Locking)：**应事务的要求，直接对该数据对象的加锁。
- ② **隐式加锁(Implicit Locking)：**该数据对象本身并未被显式加锁，但由于其上级被加锁，故这个数据对象被隐含加锁了。例如，一个表被加锁，则其所有元组和列均隐式被加锁。

例: 若事务T1 显示地给图中的P3结点加排它锁，则事务T1也隐式地给所有属于该页的元组加了排它锁



说明：显式加锁和隐式加锁的效果是相同的。

多粒度加锁下的锁冲突检测问题：如只有显式锁，锁的冲突较易发现；若有隐式锁，则检查冲突较复杂。

多粒度加锁下，要对某数据对象加锁时的锁冲突检测：

- ① 检查其本身，有无显式锁与本事务显式锁冲突；
- ② 检查其所有祖先，以防本事务的显式锁与其他事务的隐式锁冲突；
- ③ 检查其所有子孙，以防本事务的隐式锁与其它事务的显式锁相冲突。

2) 锁冲突检测问题的解决

意向锁的提出：为简化锁冲突的检测，多粒度加锁协议中，除S锁和X锁外，另外引入了三种意向锁(**Intent Locks**)。该方法在System R首创，并得到广泛应用。

① 意向共享IS (Intention Share)锁

加IS锁：一个事务要给一个数据对象加S锁，必须首先将其祖先加IS锁。也就是说，如果一个数据对象被加了IS锁，表示其某些子孙加了或准备加S锁。

锁冲突关系：IS锁与X锁冲突。

锁冲突检测：

① 如果事务对所操作对象的祖先的IS锁申请成功，就可保证再给该对象加S锁时，不会造成S锁与该对象的隐式锁发生冲突。因为该对象的祖先的锁是本事务加的，当然就不再有这种冲突存在；

② 如事务给数据对象加了S锁，那么其它事务就不能给其任何祖先加X锁，因为其祖先已被该事务加了IS锁。

② 意向排他IX (Intention eXclusive)锁

加IX锁：一个事务要给一个数据对象加X锁，必须首先将其祖先加IX锁。也就是说，如果一个数据对象被加了IX锁，表示其某些子孙加了或准备加X锁。

锁冲突关系：IX锁与S、X以及后面的SIX锁冲突。

锁冲突检测：

① 如果事务对所操作对象的祖先申请IX锁成功，就可保证再给该对象加X锁时，不会造成X锁与该对象的隐式锁发生冲突，因为该对象的祖先的锁是本事务加的，当然就不再有这种冲突存在；

② 如事务给数据对象加了X锁，那么其它事务就不能给其任何祖先加S或X锁，因为其祖先已被该事务加了IX锁，而IX锁与S锁和X锁是冲突的。

③共享意向排它SIX (Share Intention eXclusive)锁

SIX锁的引入或SIX的应用情形：实际应用中，事务需要读整张表，并修改其中个别记录。即该事务需要整张表的**S锁**和**IX锁**，这样，该事务才能接着以**X锁**锁住其中要修改的行。由于**S锁与IX锁是不相容的**，不可能同时给整张表既加S锁又加IX锁。为解决此问题，定义了**SIX锁**。

SIX锁的等价关系：**SIX锁**在逻辑上等价于同时拥有**S锁和IX锁**。

锁冲突关系：**SIX锁**与那些同S锁或IX锁不相容的锁冲突，即**SIX与IX、S、X及SIX冲突**。

多粒度加锁协议的相容矩阵

加锁申请	数据对象的锁状态					
	NL	IS	IX	S	SIX	X
IS	Y	Y	Y	Y	Y	N
IX	Y	Y	Y	N	N	N
S	Y	Y	N	Y	N	N
SIX	Y	Y	N	N	N	N
X	Y	N	N	N	N	N

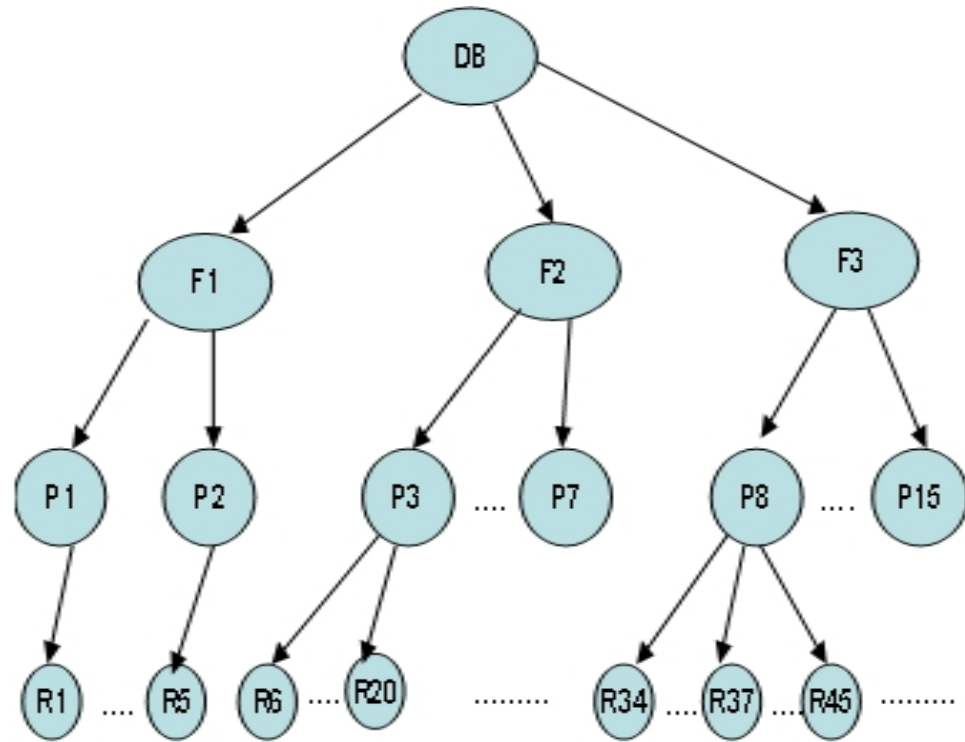
多粒度加锁协议相容矩阵

多粒度加锁/解锁顺序

- ① 要对数据对象加锁，必须对这个数据对象的**所有祖先**加相应的意向锁，即：申请锁时，应按**自上而下**（从根到叶）的次序申请，以便及时发现冲突；
- ② 解锁时，应按**自下而上**（从叶到根）的次序进行，以免出现锁冲突。

多粒度封锁的示例

- 例: (1) 若事务T3读取P1页的元组R1, 则事务T3 需对数据库DB、文件F1、页P1 加IS锁, 最后对R1加共享锁S。
- (2) 若事务T4修改P1页的元组R2, 则事务T4 需对数据库DB、文件F1、页P1 加IX锁, 最后对R2加排它锁X。
- (3) 若事务T5要读取P1页下的所有元组, 那么T5需对数据库DB、文件F1 加IS锁, 最后对P1加共享锁S。
- (4) 若事务T6读取整个数据库, 那么给数据库加S锁就可以了。
- (5) 事务T3、T4可以并发执行, T3、T5、T6 也可以并发执行, 但T4与T5、T6不能并发执行。



事务隔离级别

第1级别：Read Uncommitted(读取未提交内容)

- (1)所有事务都可以看到其他未提交事务的执行结果
- (2)本隔离级别很少用于实际应用，因为性能不比其它级别好多少
- (3)该级别引发的问题是：**脏读(Dirty Read)**取到了未提交的数据

第2级别：Read Committed(读取提交内容)

- (1) PostgreSQL、ORACLE、SQL Server和DB2 的默认隔离级别
- (2) 一个事务只能看见已经提交事务所做的改变
- (3)这种隔离级别出现的问题是——不可重复读。在同一个事务中执行完全相同的select语句时可能看到不一样的结果。导致这种情况的原因可能有：

- 1)有一个交叉的事务有新的commit，导致了数据的改变；
- 2)一个数据库被多个实例操作时,同一事务的其他实例在该实例处理其间可能会有新的commit

事务隔离级别

第3级别: Repeatable Read(可重读)

- (1) MySQL数据库的默认事务隔离级别
- (2) 同一事务的多个实例在并发读取数据时, 会看到同样的数据行
- (3) 此级别可能出现的问题——**幻读(Phantom Read)**: 当用户读取某一范围的数据行时, 另一个事务又在该范围内插入了新行, 当用户再读取该范围的数据行时, 会发现有了新的“幻影”行
- (4) 有些数据库存储引擎通过多版本并发控制机制解决了该问题

第4级别: Serializable(可串行化)

- (1) 这是最高的隔离级别
- (2) 它通过强制事务排序, 使之不可能相互冲突, 从而解决幻读问题。简言之,它是在每个读的数据行上加上共享锁。
- (3) 在这个级别, 可能导致大量的超时现象和锁竞争

事务隔离级别

隔离级别	脏读	不可重复读	丢失更新	幻读
读未提交	可能	可能	可能	可能
读已提交	不可能	可能	可能	可能
可重复读	不可能	不可能	可能	可能
可串行化	不可能	不可能	不可能	不可能

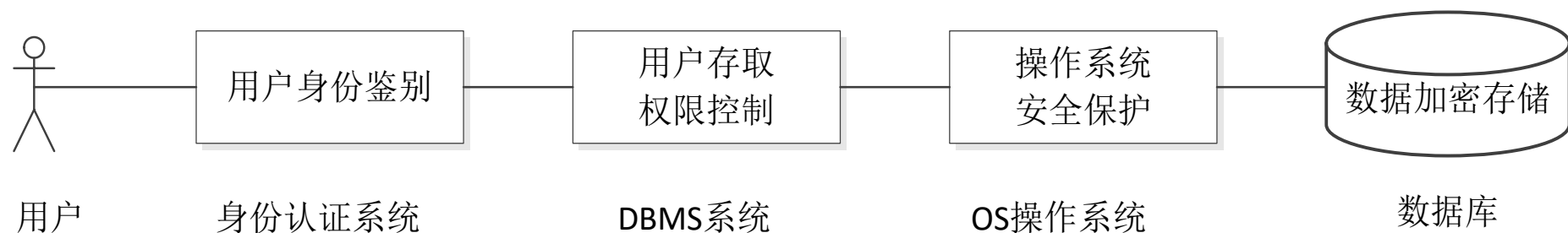
说明：每个事务都有一个所谓的隔离级。事务隔离级别设置越高，出现数据不一致的可能性越小，但系统吞吐量也越小。

数据库安全

一、数据库安全问题

- ① 黑客利用系统漏洞，攻击系统运行、窃取和篡改系统数据。
- ② 内部人员非法地泄露、篡改、删除系统的用户数据。
- ③ 系统运维人员操作失误导致数据被删除或数据库服务器系统宕机。
- ④ 系统故障导致数据库的数据损坏、数据丢失、数据库实例无法启动。
- ⑤ 意外灾害事件（火灾、水灾、地震等自然灾害）导致系统被破坏。

二、数据库系统安全模型



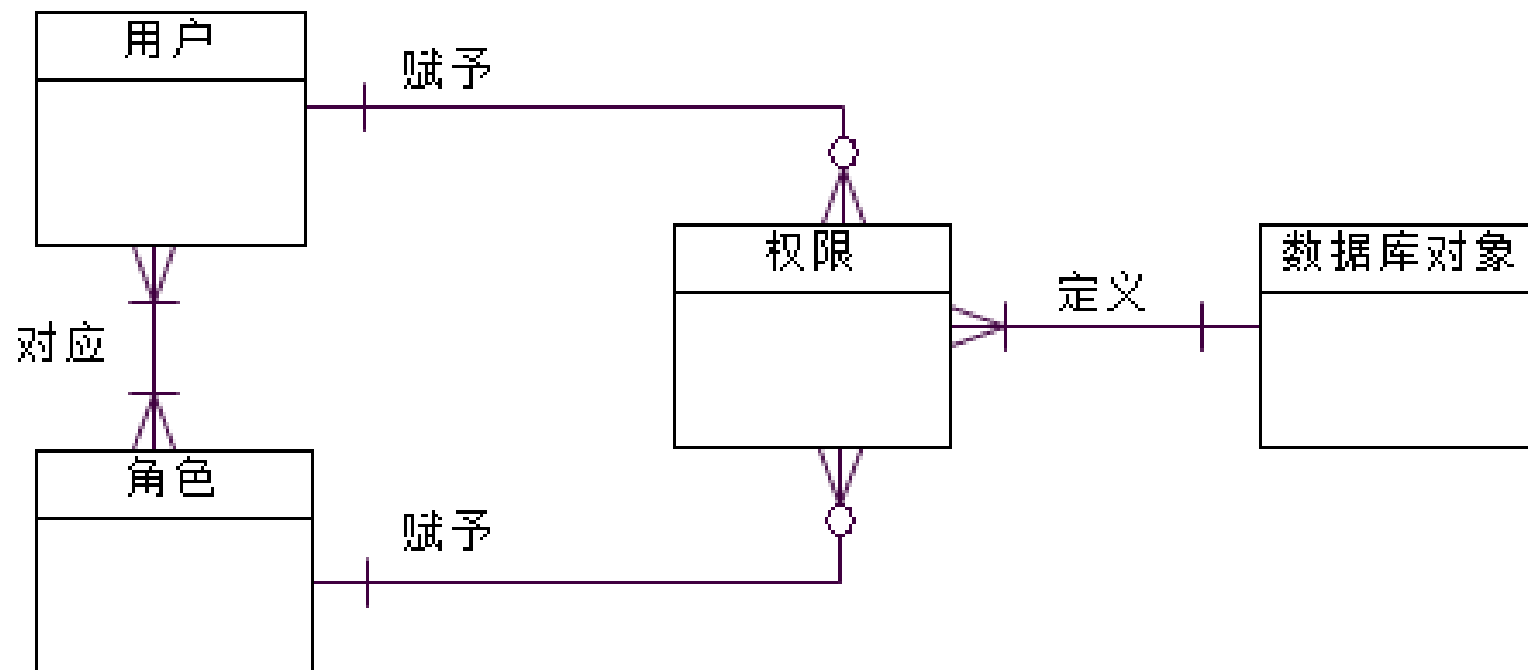
身份验证：用来确认登录用户是否是合法使用者

权限控制：通过权限机制控制用户对数据的访问

系统防护：OS系统安全机制防范非法系统访问

数据加密：通过加密算法对数据库中数据进行加密存储

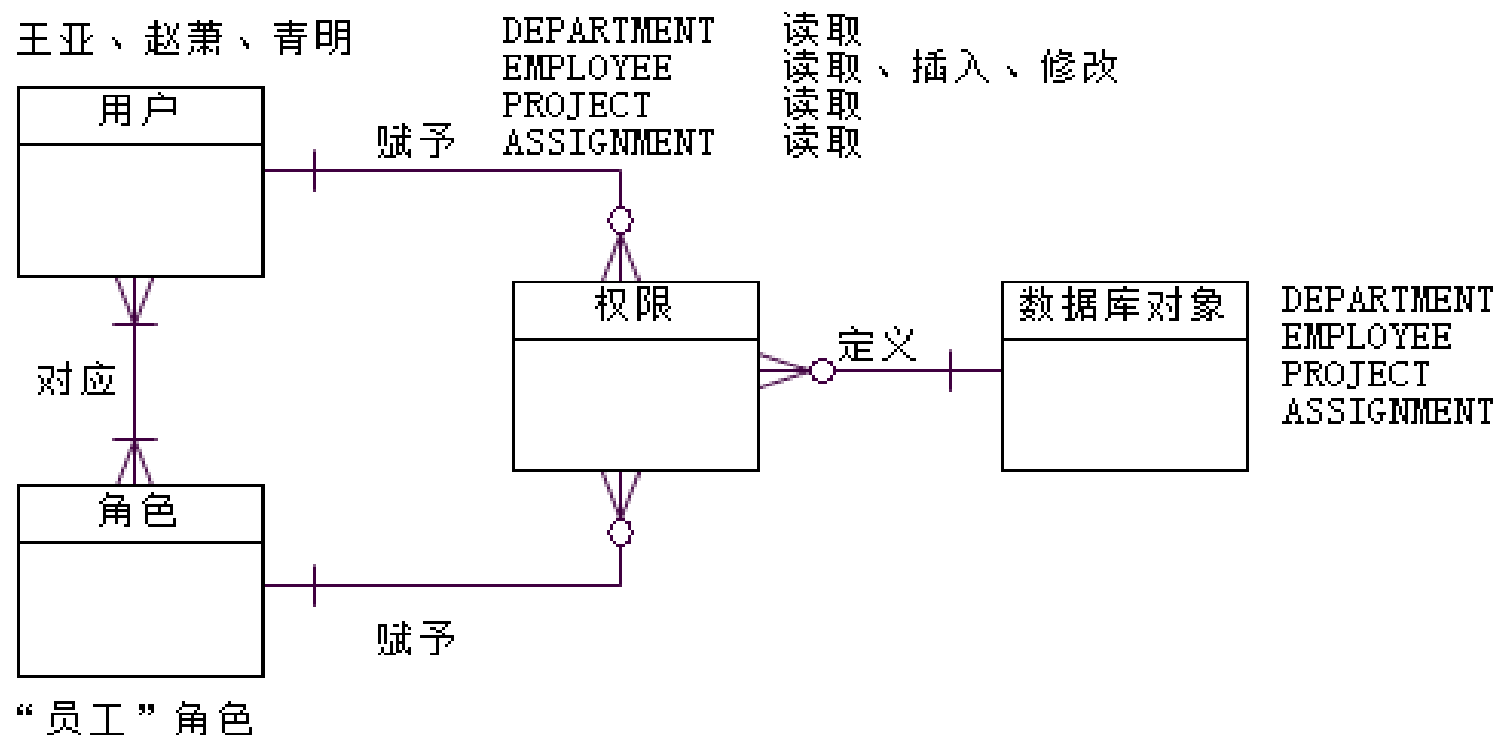
三、数据库存取权限控制安全模型



例 在3. 7. 1节的工程项目管理系统中，假定系统用户有三类角色：员工、经理和系统管理员。它们对数据库各个表对象的拥有权限见下表所示。

表	员工	经理	系统管理员
DEPARTMENT	读取	读取、插入、修改、删除	赋予权限、修改结构
EMPLOYEE	读取、插入、修改	读取、插入、修改、删除	赋予权限、修改结构
PROJECT	读取	读取、插入、修改、删除	赋予权限、修改结构
ASSIGNMENT	读取	读取、插入、修改、删除	赋予权限、修改结构

工程项目管理系统数据库存取权限控制安全模型设计



“员工”角色的用户存取访问权限

练习：在选课管理系统中，有学生、教师和教务管理员角色。如何设计各角色的数据库表对象的访问操作权限？

数据库表	学生 (StudentRole)	教师 (TeacherRole)	教务管理员 (AcademicRole)
College			
Course			
Teacher			
Student			
Plan			
Register			

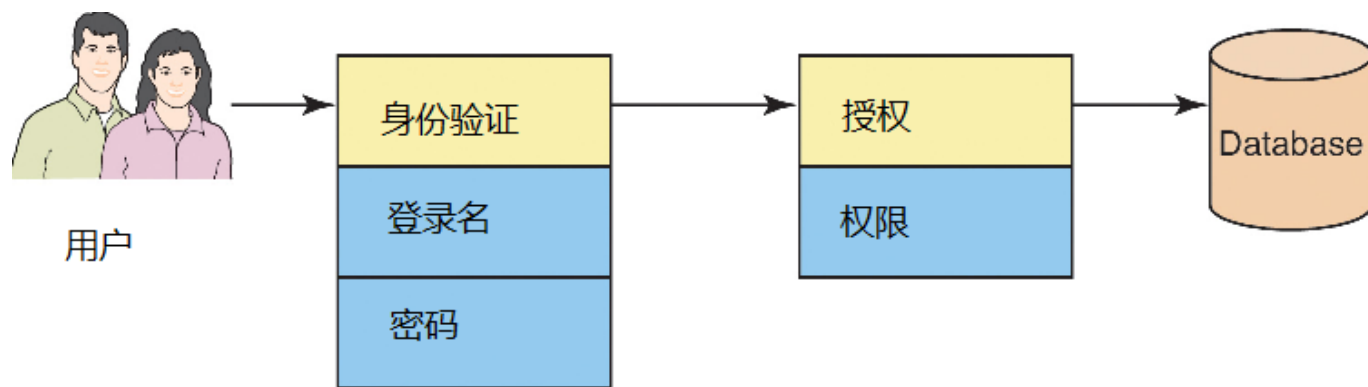
选课管理系统中，学生、教师和教务管理员角色的数据库表对象访问权限设计

数据库表	学生 (StudentRole)	教师 (TeacherRole)	教务管理员 (AcademicRole)
College	查询	查询	查询、插入、修改、删除
Course	查询	查询	查询、插入、修改、删除
Teacher	查询	查询、修改	查询、插入、修改、删除
Student	查询、修改	查询	查询、插入、修改、删除
Plan	查询	查询	查询、插入、修改、删除
Register	查询	查询	查询、插入、修改、删除

安全管理——用户、角色、权限管理

一、用户管理

用户要访问数据库，必须先**在DBMS中创建其账号**，并成为数据库的用户。此后，用户每次访问数据库，都需要在**DBMS进行身份验证**，只有合法用户才能进入系统，访问操作数据库对象。



用户管理——在数据库安全管理中，DBMS需要对每个用户进行管理，如用户创建、用户修改、用户删除管理等。

实现用户管理方式：

- ① 数据库服务器执行SQL语句管理用户
- ② 通过管理工具GUI操作管理用户

1. 用户创建SQL语句

CREATE USER <用户账号名> [[WITH] option [...]];

例 创建一个新用户，其账号名字为“userA”，密码为“123456”。该用户具有登录权限（Login）和角色继承权限（Inherit）系统权限，但它不是超级用户（SuperUser），不具有创建数据库权限（CreateDB）、创建角色权限（CreateRole）、数据库复制权限（Replication），此外数据库连接数（Connection Limit）不受限。

```
CREATE USER "userA" WITH  
LOGIN  
NOSUPERUSER  
NOCREATEDB  
NOCREATEROLE  
INHERIT  
NOREPLICATION  
CONNECTION LIMIT -1  
PASSWORD '123456';
```

用户创建SQL语句执行

运行按钮

The screenshot shows the pgAdmin 4 application window. On the left is a tree view of the database structure. The main pane on the right is divided into two sections. The top section contains a SQL editor with a query to create a user. The bottom section shows the execution results.

pgAdmin 4 文件 ▾ 对象 ▾ 工具 ▾ 帮助 ▾

浏览器

- 数据库 (2)
 - CurriculaDB
 - 事件触发器
 - 外部数据包装
 - 强制转换
 - 扩展
 - 模式 (1)
 - 目录 (2)
 - 语言 (1)
 - plpgsql
 - postgres
- 登录/组角色 (3)
 - pg_signal_backend
 - postgres
 - userA

仪表板 属性 SQL 统计信息 依赖关系 依赖组件 Query

No limit

```
1 CREATE USER "userA" WITH
2     LOGIN
3     NOSUPERUSER
4     NOCREATEDB
5     NOCREATEROLE
6     INHERIT
7     NOREPLICATION
8     CONNECTION LIMIT -1
9     PASSWORD '123456';
```

SQL 语句

数据输出 解释 消息 历史

CREATE ROLE

耗时167 msec 成功返回查询

结果消息

2. 用户修改SQL语句

① 修改用户的属性

ALTER USER <用户名> [[**WITH**] option [...]];

② 修改用户的名称

ALTER USER <用户名> **RENAME TO** <新用户名>;

③ 修改用户的参数值

ALTER USER <用户名> **SET** <参数项> { **TO** | **=** } { value | **DEFAULT** };

④ 重置用户参数值

ALTER USER <用户名> **RESET** <参数项>;

例：修改用户“userA”的账号密码为“gres123”。同时也限制该用户的数据库连接数为10。

```
ALTER USER 'userA'  
CONNECTION LIMIT 10  
PASSWORD 'gres123';
```

用户修改SQL语句执行

运行按钮

The screenshot shows a database management tool interface. At the top, there is a toolbar with icons for dashboard, properties, SQL, statistics, dependencies, and components. The main window displays the title 'Query - CurriculaDB on postgres@PostgreSQL 10 *'. Below the toolbar, there is a blue header bar with the text 'CurriculaDB on postgres@PostgreSQL 10'. The main area contains a SQL query editor with the following text:

```
1 ALTER USER "userA"  
2 CONNECTION LIMIT 10  
3 PASSWORD 'gres123';  
4  
5
```

Below the query editor, there is a tabbed interface with '数据输出' (Data Output), '解释' (Explain), '消息' (Messages), and 'Query History'. The '消息' tab is selected, showing the following output:

```
ALTER ROLE  
  
耗时59 msec 成功返回查询
```

Annotations in the image point to the '运行按钮' (Run button) in the toolbar, the SQL query text, and the '结果消息' (Result message) output.

SQL语句

结果消息

3. 用户删除SQL语句

DROP USER <用户名>;

例：在数据库中，删除用户“userA”。可以通过执行如下用户删除SQL语句实现用户删除。

DROP USER userA;

用户删除SQL语句执行

The screenshot shows a database management tool interface. At the top, there is a toolbar with icons for dashboard, properties, SQL, statistics, dependencies, and components. The main window displays the title "CurriculaDB on postgres@PostgreSQL 10". Below the title, there is a text area containing the SQL statement: `DROP USER "userA";`. To the right of the text area, there is a yellow callout box labeled "SQL语句" (SQL Statement) with a line pointing to the text. Above the text area, there is a yellow callout box labeled "运行按钮" (Run Button) with a line pointing to a lightning bolt icon in the toolbar. At the bottom of the interface, there is a tabbed area with "数据输出" (Data Output), "解释" (Explain), "消息" (Messages), and "Query History". The "消息" tab is selected, showing the output: `DROP ROLE` and `耗时119 msec 成功返回查询` (Consumed 119 msec, successfully returned query). To the right of the output, there is a yellow callout box labeled "结果消息" (Result Message) with a line pointing to the output text.

运行按钮

SQL语句

结果消息

二、权限管理

数据库权限管理是指DBA管理员或数据库对象拥有者对其所拥有对象进行权限控制设置。

权限管理基本操作：

- 授予权限
- 收回权限
- 拒绝权限

权限类别：

- 数据库系统权限
- 数据库对象访问操作权限
- 数据库对象定义操作权限

1. 权限管理SQL语句

GRANT <权限名> **ON** <对象名> **TO** {数据库用户名|用户角色名};

REVOKE <权限名> **ON** <对象名> **FROM** {数据库用户名|用户角色名};

DENY <权限名> **ON** <对象名> **TO** {数据库用户名|用户角色名};

2. 权限管理实例

例 在3.7.1节的工程项目管理系统中，DBA管理员赋予员工用户（userA）对部门表（Department）、员工表（Employee）、项目表（Project）和任务表（Assignment）的读取数据权限。

对用户 “userA” 实现授权SQL程序如下

```
GRANT SELECT ON Department TO userA;
```

```
GRANT SELECT ON Employee TO userA;
```

```
GRANT SELECT ON Project TO userA;
```

```
GRANT SELECT ON Assignment TO userA;
```

用户授权SQL语句执行

运行按钮

The screenshot shows a SQL execution tool interface. At the top is a toolbar with icons for file operations, search, and execution. The main area contains a list of SQL statements. Below this is a tabbed interface with '数据输出' (Data Output), '解释' (Explain), '消息' (Messages), and '历史' (History). The '消息' tab is active, showing the execution result of the first statement.

```
1 GRANT SELECT ON Department TO "userA";
2 GRANT SELECT ON Employee TO "userA";
3 GRANT SELECT ON Project TO "userA";
4 GRANT SELECT ON Assignment TO "userA";
5
```

SQL语句

数据输出 解释 消息 历史

GRANT

耗时98 msec 成功返回查询

结果消息

三、角色管理

在DBMS中，为了方便对众多用户及其权限进行管理，通常将一组具有相同权限的用户定义为**角色(Role)**。

角色管理内容：

- 创建角色
- 修改角色
- 删除角色

角色管理实现方式：

- 执行SQL语句管理角色
- 通过GUI操作管理角色

1. 角色管理SQL语句

① 创建角色的属性

ALTER ROLE <角色名> [[**WITH**] option [...]];

② 修改角色的属性

ALTER ROLE <角色名> [[**WITH**] option [...]];

③ 修改角色的名称

ALTER ROLE <角色名> **RENAME TO** <新角色名>;

④ 修改角色的参数值

ALTER ROLE <角色名> **SET** <参数项> { **TO** | **=** } { value | **DEFAULT** };

④ 删除指定角色

DROP ROLE <角色名>;

2. 角色管理实例

例：在工程项目管理系统中，假定需要在ProjectDB数据库内创建经理角色Role_Manager。该角色具有登录权限（Login）和角色继承权限（Inherit）系统权限，但它不是超级用户（SuperUser），不具有创建数据库权限（CreateDB）、创建角色权限（CreateRole）、数据库复制权限（Replication），此外数据库连接数（Connection Limit）不受限。

```
CREATE ROLE "Role_Manager" WITH  
LOGIN  
NOSUPERUSER  
NOCREATEDB  
NOCREATEROLE  
INHERIT  
NOREPLICATION  
CONNECTION LIMIT -1;
```

角色创建SQL语句执行

运行按钮

The screenshot displays the pgAdmin 4 application window. On the left, the '浏览器' (Browser) pane shows a tree structure of database objects, with '登录/组角色 (4)' (Logins/Groups/Roles (4)) selected. The main pane shows the 'SQL' tab with the following SQL statement:

```
1 CREATE ROLE "Role_Manager" WITH
2     LOGIN
3     NOSUPERUSER
4     NOCREATEDB
5     NOCREATEROLE
6     INHERIT
7     NOREPLICATION
8     CONNECTION LIMIT -1;
```

Below the SQL editor, the '消息' (Messages) tab is active, showing the execution result:

```
CREATE ROLE
耗时100 msec 成功返回查询
```

Annotations with arrows point to the '运行按钮' (Run button) in the SQL editor toolbar, the SQL statement itself, and the execution result message.

SQL语句

结果消息

3. 角色权限授予

例：在创建好经理角色Role_Manager后，还需要赋予该角色对数据库表Department、Employee、Project、Assignment的插入、修改、删除、查询权限。

```
GRANT SELECT,INSERT,UPDATE,DELETE  
ON Department TO "Role_Manager";
```

```
GRANT SELECT,INSERT,UPDATE,DELETE  
ON Employee TO "Role_Manager";
```

```
GRANT SELECT,INSERT,UPDATE,DELETE  
ON Project TO "Role_Manager";
```

```
GRANT SELECT,INSERT,UPDATE,DELETE  
ON Assignment TO "Role_Manager";
```

角色授权SQL语句执行

The screenshot displays the pgAdmin 4 application window. On the left, the '浏览器' (Browser) pane shows a tree structure of database objects, with 'Role_Manager' under '登录/组角色 (4)' (Logins/Groups/Roles (4)) selected. The main pane shows the 'Query - ProjectDB on postgres@PostgreSQL 9.6' window. The SQL editor contains four GRANT statements:

```
1 GRANT SELECT, INSERT, UPDATE, DELETE ON Department TO "Role_Manager";
2 GRANT SELECT, INSERT, UPDATE, DELETE ON Employee TO "Role_Manager";
3 GRANT SELECT, INSERT, UPDATE, DELETE ON Project TO "Role_Manager";
4 GRANT SELECT, INSERT, UPDATE, DELETE ON Assignment TO "Role_Manager";
```

Below the SQL editor, the '消息' (Messages) tab is active, showing the execution result: 'GRANT' and '耗时117 msec 成功返回查询' (Took 117 msec, successfully returned query). Three callout boxes with arrows point to specific elements: '运行按钮' (Run button) points to the lightning bolt icon in the toolbar; 'SQL语句' (SQL statement) points to the text in the SQL editor; '结果消息' (Result message) points to the output in the Messages tab.

pgAdmin 4

文件 ▾ 对象 ▾ 工具 ▾ 帮助 ▾

浏览器

- 外部数据包装
- 强制转换
- 扩展
- 模式 (1)
- 目录
- 语言
- postgres
- 登录/组角色 (4)
 - Role_Manager
 - pg_signal_backend
 - postgres
 - userA
- 表空间

仪表板 属性 SQL 统计信息 依赖关系 依赖组件 Query - ProjectDB on postgres@PostgreSQL 9.6

ProjectDB on postgres@PostgreSQL 9.6

```
1 GRANT SELECT, INSERT, UPDATE, DELETE ON Department TO "Role_Manager";
2 GRANT SELECT, INSERT, UPDATE, DELETE ON Employee TO "Role_Manager";
3 GRANT SELECT, INSERT, UPDATE, DELETE ON Project TO "Role_Manager";
4 GRANT SELECT, INSERT, UPDATE, DELETE ON Assignment TO "Role_Manager";
```

数据输出 解释 消息 历史

GRANT

耗时117 msec 成功返回查询

运行按钮

SQL语句

结果消息

本章结束，谢谢大家！