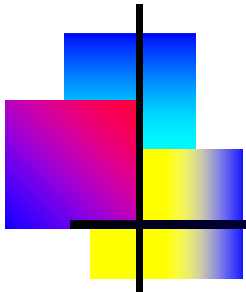




教材：数据库实用教程（第四版）

《数据库原理》课程

清华大学出版社
2024年12月30日

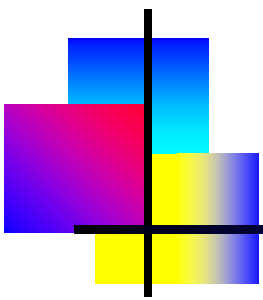


《数据库原理》

第八章 数据库管理

清华大学出版社

2024年12月30日



第八章 数据库管理

教学内容:

- 事务的定义，事务的ACID性质，事务的状态变迁图。
- 恢复的定义、基本原则和实现方法, 故障的类型, 检查点技术。
- 并发操作带来的三个问题，X锁、PX协议...,活锁、死锁，
并发调度、串行调度、并发调度的可串行化，两段封锁法。
- 完整性的定义，完整性子系统的功能，完整性规则的组成；
SQL中的三大类完整性约束，SQL3中的触发器技术。
- 安全性的定义、级别，权限，SQL中的安全性机制。



教学重点：

- 检查点技术
- 并发操作、封锁带来的若干问题，并发调度的可串行化。
- SQL中完整性约束的实现：断言、触发器技术。
- 安全性中的授权语句。



§ 1 事务的概念

一、事务的定义

形成一个逻辑工作单元的数据库操作的汇集,
称为事务(transaction)。

例：在关系数据库中，一个事务可以是一条SQL语句、
一组SQL语句
或整个程序。

事务和程序是两个概念。一般地说：一个程序中包含多个事务。
事务的开始和结束可以由用户显式控制。

如果用户没有显式地定义事务,则由DBMS按照缺省自动划分事务。

在SQL语言中，定义事务的语句由三条：

事务开始： **BEGIN TRANSACTION**

事务提交： **COMMIT**

事务回滚： **ROLLBACK**

二、事务的ACID性质

为了保证数据完整性（数据是正确的），要求事务

具有下列四个性质：

原子性 (Atomicity)

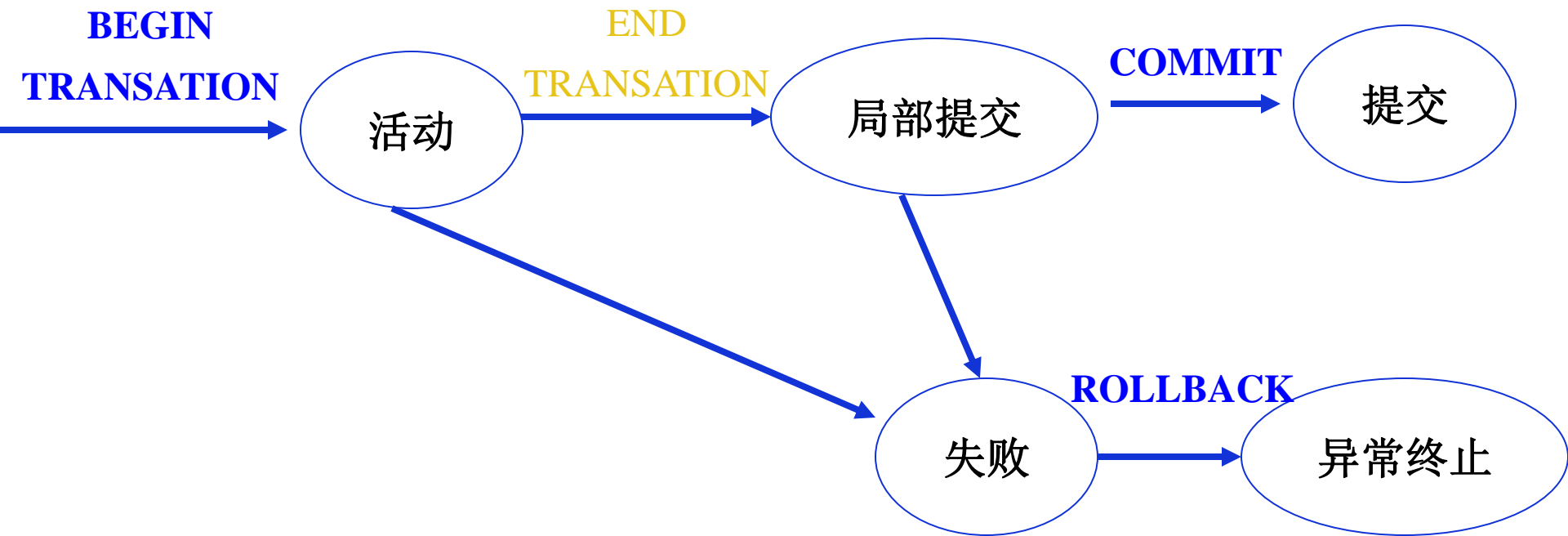
一致性 (Consistency)

隔离性 (Isolation)

持久性 (Durability) 。

上述四个性质称为事务的ACID性质。

三、事务的状态变迁





§ 2 数据库的恢复

DBMS的恢复管理子系统：

采取一系列措施保证在任何情况下保持事务的原子性和持久性，确保数据不丢失、不破坏；当发生系统故障时，数据库可恢复到正确状态。

一、故障分类

事务故障

系统故障

介质故障

二、数据库恢复技术

恢复机制涉及的两个关键问题：

如何建立冗余数据；如何利用这些冗余数据实施数据库的恢复。

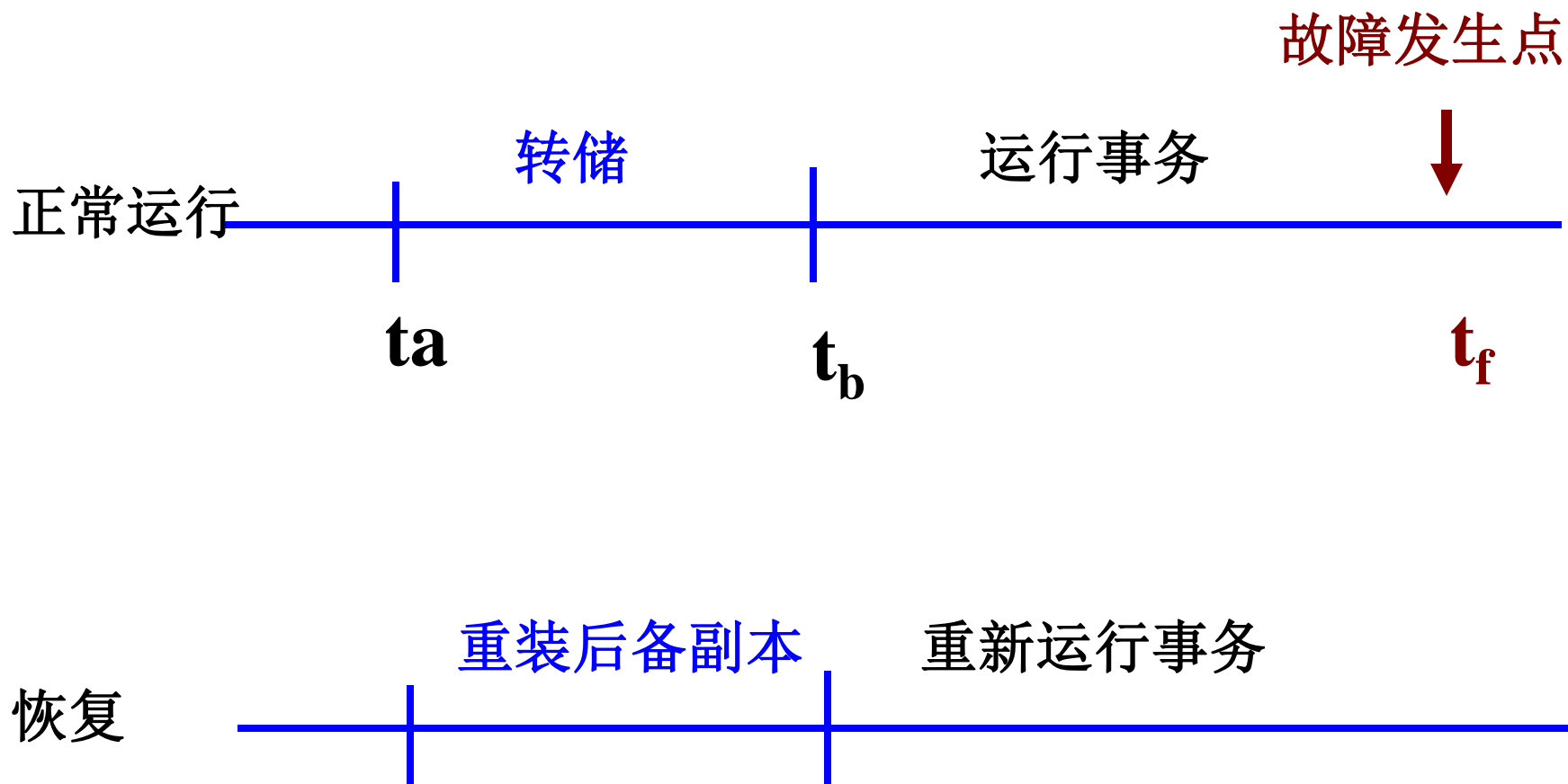
建立冗余数据最常用的技术是数据库转储和登录日志文件。

1. 数据转储

转储是指DBA将整个数据库复制到永久存储器的过程。

这些备用的数据文本称为后备副本或后援副本。

一旦系统发生介质故障，数据库遭到破坏，可以将副本重新装入，把数据库恢复起来。



2. 登记日志文件 (Logging)

日志文件是记录事务对数据库的更新操作的文件。

日志文件的存储结构——日志记录的表示：

- ① 事务开始记录： $\langle T_i, \text{START} \rangle$
- ② 更新数据记录： $\langle T_i, X, A, V_1, V_2 \rangle$
- ③ 事务终止记录： $\langle T_i, \text{COMMIT} \rangle$

更新数据日志记录 $\langle T_i, X, A, V_1, V_2 \rangle$ 与每一个数据库写操作
WRITE (Q) 相对应其中：

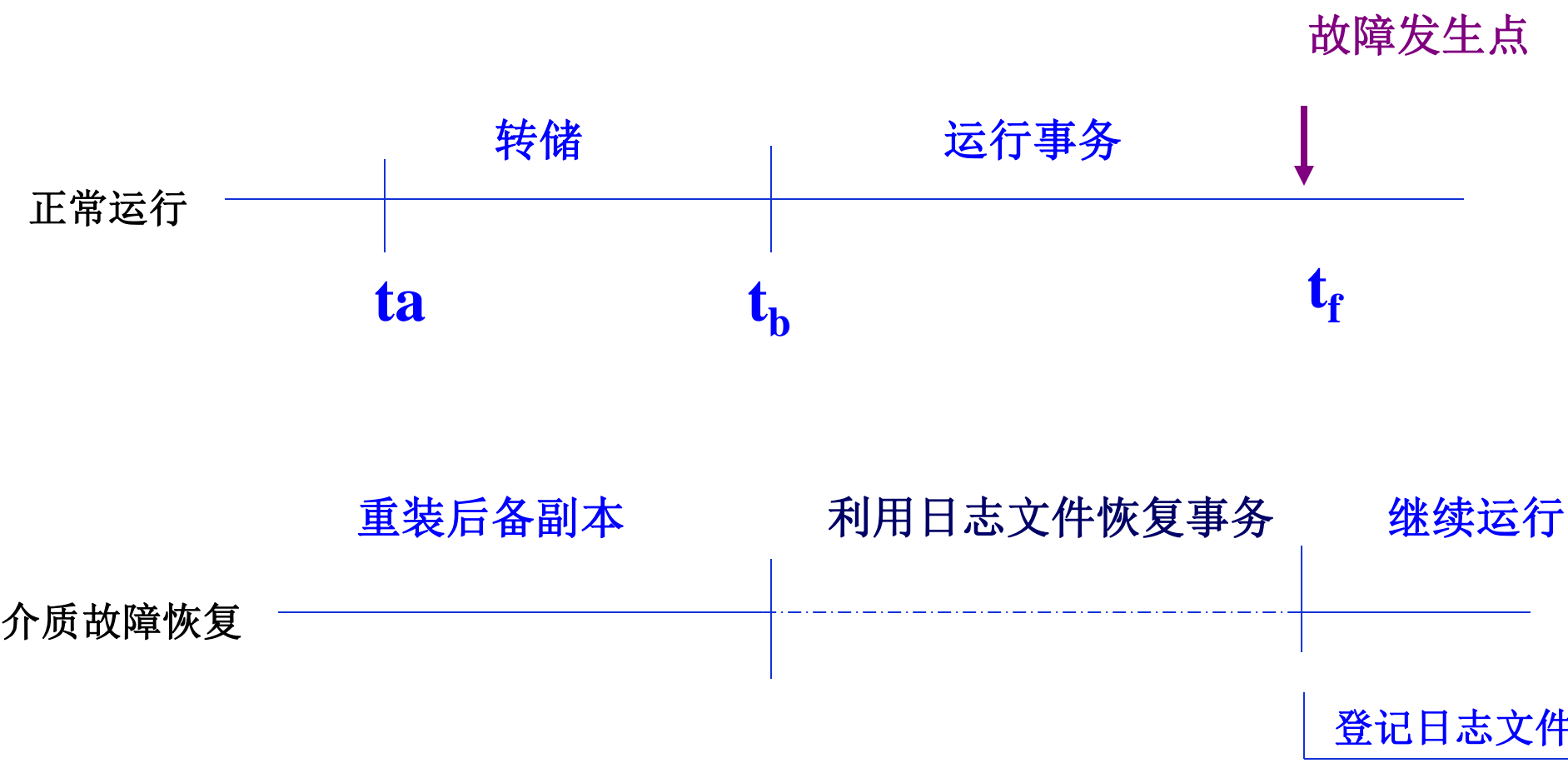
T_i :事务名； X:操作类型； A:数据项； V_1 :原始值； V_2 :新值

日志文件在数据库恢复过程中起着重要的作用。

为保证数据库的可恢复性，

登记日志文件必须遵循两条原则：

- ① 登记的次序必须严格按并行事务执行的时间次序。
- ② 必须先写日志文件，再写数据库。



三、恢复策略

当系统运行过程中发生故障，利用数据库后备副本和日志文件将数据库恢复到故障前的某个一致性状态。不同故障其恢复技术不一样：

1. 事务故障的恢复

事务故障是指事务在运行至正常终止点前被中止，此时恢复子系统应撤销（UNDO）此事务已对数据库进行的修改。

事务故障恢复的具体做法如下：

① 反向扫描日志文件（即从最后向前扫描日志文件），

查找该事务的更新操作。

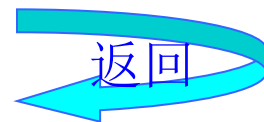
② 对该事务的更新操作执行逆操作。

即将日志记录中“更新前的值”写入数据库：

若记录中是插入操作，则相当于做删除操作；

若记录中是删除操作，则做插入操作；

若是修改操作，则用修改前值代替修改后值。



③ 继续反向扫描日志文件，查找该事务的其他更新操作，并做同样处理。

④ 如此处理下去，直至读到此事务的开始标记，事务故障恢复就完成了。

事务故障的恢复是由系统自动完成的, 不需要用户干预。

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有两个：

未完成事务对数据库的更新已写数据库；

已提交事务对数据库的更新还留在缓冲区没来得及真正
写入数据库。

恢复操作：撤销故障发生时未完成的事务，
重做已完成的事务。

具体做法如下:

① 正向扫描日志文件(即从头开始扫描日志文件):

找出在故障发生前:

已提交事务(既有 $\langle Ti, START \rangle$ 记录, 也有 $\langle Ti, COMMIT \rangle$ 记录), 将其事务标识**记入重做队列**。

尚未完成的事务(有 $\langle Ti, START \rangle$ 记录, 无 $\langle Ti, COMMIT \rangle$ 记录), 将其事务标识**记入撤销队列**。

② 对撤销队列中的各个事务进行撤销（UNDO）处理

进行撤销（UNDO）处理的方法是：

反向扫描日志文件，

对每个UNDO事务的更新操作执行逆操作。

即将日志记录中“更新前的值”写入数据库。

③ 对重做队列中的各个事务进行重做（REDO）处理

进行重做REDO处理的方法是：

正向扫描日志文件，

对每个REDO事务重新执行登记操作。

即将日志记录中“更新后的值”写入数据库。

系统故障的恢复也由系统自动完成的,不需要用户干预。

3. 介质故障的恢复

在发生介质故障和遭受病毒破坏时，磁盘上的物理数据库遭到毁灭性破坏。此时恢复的过程如下：

① 装入最新的后备副本到新的磁盘，使数据库恢复到最近一次转

储时的一致状态。

② 装入有关的日志文件副本，重做已提交的所有事务。

这样就可以将数据库恢复到故障前某一时刻的一致状态。

四、检测点机制

为提高系统效率，DBMS定时设置检查点。

在检查点时刻才真正做到把对DB的修改写到磁盘，并在日志文件写入一条检查点记录（以便恢复时使用）。

1. 检查点方法

DBMS定时设置检查点，在检查点时，做下列事情：

第一步： 将日志缓冲区中的日志记录写入磁盘。

第二步： 将数据库缓冲区中修改过的缓冲块内容写入磁盘。

第三步： 写一个检查点记录到磁盘，内容包括：

① 检查点时刻，所有活动事务；

② 每个事务最近日志记录地址。

第四步：把磁盘中日志检测点记录的地址写入“重新启动文件中”。

2. 检查点恢复步骤

① 正向扫描日志文件，建立事务重做队列和事务撤销队列。

重做队列：将已完成的事务加入重做队列；

撤销队列：未完成的事务加入撤销队列。

② 对撤销队列做UNDO处理的方法是：

反向扫描日志文件，根据撤销队列的记录对每一个撤销

事务的更新操作执行逆操作，使其恢复到原状态。

③ 对重做队列做REDO处理的方法是：

正向扫描日志文件，根据重做队列的记录对每一个重做事
务实施对数据库的更新操作。

板书举例：

五、运行记录（日志记录）优先原则

为了安全，定义“运行记录优先原则”包含以下两点：

- ① 至少要等相应运行记录（日志记录）已经写入运行日志文件后，才能允许事务往数据库中写记录；
- ② 直至事务的所有运行记录（日志记录）都已经写入到运行日志文件后，才能允许事务完成COMMIT处理。

这样，如果出现故障，则可能在运行日志中而不是在数据库中记录了一个修改。在重新启动时，就有可能请求UNDO / REDO处理原先根本没有对数据库做过的修改。

五、运行记录（日志记录）优先原则

为了安全，定义“运行记录优先原则”包含以下两点：

- ① 至少要等相应运行记录（日志记录）已经写入运行日志文件后，才能允许事务往数据库中写记录；
- ② 直至事务的所有运行记录（日志记录）都已经写入到运行日志文件后，才能允许事务完成COMMIT处理。

这样，如果出现故障，则可能在运行日志中而不是在数据库中记录了一个修改。在重新启动时，就有可能请求UNDO / REDO处理原先根本没有对数据库做过的修改。



§ 3 数据库的并发控制

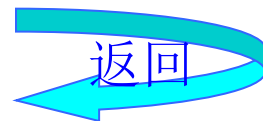


原子性 (Atomicity)

一个事务对数据库的所有操作，是一个不可分割的工作单元。这些操作要么全部执行，要么什么也不做（就效果而言）。

保证原子性是数据库系统本身的职责，

由DBMS的事务管理子系统来实现。



一致性 (Consistency)

一个事务独立执行的结果，应保持数据库的一致性，

即数据不会应事务的执行而遭受破坏。

编写事务的应用程序员的职责：确保单个事务的一致性。

在系统运行时，由DBMS的完整性子系统执行测试任务。



返回

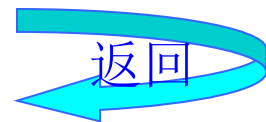
隔离性 (Isolation)

在多个事务并发执行时, 系统应保证与这些事务先后单独执行时的结果一样, 此时称事务达到了隔离性的要求。

即：多个事务并发执行时，保证执行结果是正确的，

如同单用户环境一样。

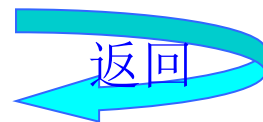
隔离性是由DBMS的并发控制子系统实现的。



持久性 (Durability)

一个事务一旦完成全部操作后，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，也应保留这个事务执行的痕迹。

持久性由DBMS的恢复管理子系统实现的。





§ 3 数据库的并发控制

教学内容:

并发操作带来的三个问题，X锁、PX协议...,活锁、死锁，并发调度、串行调度、并发调度的可串行化，两段封锁法。

教学重点: 并发操作、封锁带来的若干问题，

并发调度的可串行化。

为充分利用数据库资源，发挥数据库共享资源的特点,应该允许多个用户并行地存取数据库。

并发控制机制的好坏是衡量一个数据库管理系统性能的重要标志之一。

一、并发操作带来的三个问题

对并发操作如果不进行合适的控制，可能会导致数据库中数据的不一致性。

典型的并发操作的例子：火车订票系统中的订票操作。

在该系统中的一个活动序列：

- ①甲售票员读出某列车的车票余数为A，设： $A=18$ ；
- ②乙售票员读出同一列车车票余数为A，也是： $A=18$ ；
- ③甲售票点卖出一张车票，修改车票余数 $A=A-1$ ，所以 $A=17$ ，
把A写回数据库；
- ④乙售票点卖出二张车票，修改车票余数 $A=A-2$ ，所以 $A=16$ ，
把A写回数据库。

事实上是卖出三张车票，而数据库中车票余额只减少2。

这种情况称为数据库的不一致性。这种不一致性是由甲乙两个售票员并发操作引起的。（在一个CPU上，利用分时方法实行多个事务同时执行）。

在并发操作情况下，对甲、乙两个事务的操作序列的调度是随机的。若按上面的调度序列执行，甲事务的修改就被丢失。这是由于第4步中乙事务修改A并写回后覆盖了甲事务的修改。

并发操作带来的数据不一致性包括三类：

丢失修改；

不一致分析（不可重复读）；

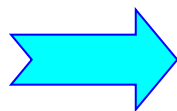
读“脏”数据。

并发操作带来的数据不一致性包括三类：

丢失修改；

不一致分析（不可重复读）；

读“脏”数据。



1. 丢失更新 (Lost update)

指事务Ti与事务Tj从数据库中读入同一数据并修改,事务2的提交结果破坏了事务1提交的结果,导致事务1的修改被丢失。

时间	事务Ti	数据库中A的值	事务Tj
t_0		18	
t_1	检索A: $A=18$		
t_2			检索A: $A=18$
t_3	修改A: $A \leftarrow A-1$		
t_4	写回A: $A=17$		
t_5		17	
t_6			修改A: $A \leftarrow A-2$
t_7			写回A: $A=16$
t_8		16	

2.不一致分析（不可重复读nonrepeatable read）

指事务Ti读取数据后，事务Tj执行更新操作，使事务Ti无法再读取前一次结果。

时间	事务Ti	数据库中A、B的值	事务Tj
t ₀		50、100	
t ₁	检索A、B：A=50，B=100		
t ₂	求和：A+B=150		
t ₃			检索B：B=100
t ₄			修改B：B←B*2
t ₅			写回B：B=200
t ₆		50、200	
t ₇	检索(验算)：A=50，B=200		
t ₈	求和：A+B=250		

具体地讲，不一致分析（不可重复读）包括三种情况：

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

② 事务 T_i 按一定条件从数据库中读取某些数据记录后，事务 T_j 删除了其中部分记录，当事务 T_i 再次按相同条件读取数据时：
发现某些记录消失了。

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

② ③两种不可重复读有时也称为幻行（phantom row）现象。

3.读“脏”数据（dirty read）

指:事务Ti修改某一数据，并将其写回磁盘，事务Tj读取同一数据后，事务Ti由于某种原因被撤销，这时事务Ti已修改过的数据恢复原值，事务Tj读到的数据就与数据库中的数据不一致，是不正确的数据，称为“脏”数据。

时间	事务Ti	数据库中C的值	事务Tj
t ₀		100	
t ₁	检索C： C=100		
t ₂	修改C： C←C*2		
t ₃	写回C:: C=200		
T ₄		200	
t ₅			检索C： C=200
t ₆	回滚： ROLLBACK		
t ₇	C恢复为： 100		
t ₈		100	

产生上述三类数据不一致性的主要原因是：

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。

DBMS的并发控制子系统的职责：

负责协调并发事务的执行，

保证数据库的完整性，

同时避免用户得到不正确的数据。

计算机系统对并事务中并行操作的调度是随机的，而不同的调度可能会产生不同的结果，那么哪个结果是正确的，哪个是不正确的呢？

二、并发调度的可串行化

1. 概念

事务的调度：事务的执行次序称为“调度”。

串行调度：如果多个事务依次执行，则称为事务的串行调度（Serial Schedule）。

并发调度：如果利用分时的方法，同时处理多个事务，则称为事务的并发调度（Concurrent Schedule）。

在事务并发执行时, 有可能破坏数据库的一致性, 或用户读了脏数据。

如果有 n 个事务串行调度, 可有 $n!$ 种不同的有效调度。

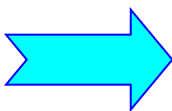
如果有 n 个事务并发调度, 可能的并发调度数目远远大于 $n!$ 。

DBMS的并发控制子系统实现:

如何产生正确的并发调度。

如何判断一个并发调度是正确的,

用并发调度的可串行化概念解决。



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

假设A的初值为10，B的初值为2。

下图给出了对这两个事务的三种不同的调度策略。

(a) 和 (b) 为两种不同的串行调度策略，虽然执行结果不同，但它们都是正确的调度。

(c) 中两个事务是交错执行的，由于其执行结果与 (a)、(b) 的结果都不同，所以是错误的调度。

(d) 中两个事务也是交错执行的，由于其执行结果与串行调度1（图 (a)）的执行结果相同，所以是正确的调度。

(a) 串行调度1(先T1后T2)

时间	事务T1	数据库中A、B的值	事务T2
t_0		10、2	
t_1	检索B: $B=2$		
t_2	修改: $A \leftarrow B+1$		
t_3	写回A: $A=3$	3、2	
t_5			检索A: $A=3$
t_6			修改: $A \leftarrow A+1$
t_7			写回B: $B=4$
t_8		3、4	

(b) 串行调度2 (先T2后T1)

时间	事务T1	数据库中A、B的值	事务T2
t_0		10、2	
t_1			检索A: $A=10$
t_2			修改B: $B \leftarrow A+1$
t_3			写回B: $B=11$
t_5		10、11	
t_6	检索B: $B=11$		
t_7	修改A: $A \leftarrow B+1$		
t_8	写回A: $A=12$	12、11	

(c) 不可串行化调度(交错执行)

时间	事务T1	数据库中A、B的值	事务T2
t_0		10、2	
t_1	检索B: $B=2$		
t_2			检索A: $A=10$
t_3	修改A: $A \leftarrow B+1$		
t_5	写回A: $A=3$		
t_6		3、2	修改B: $B \leftarrow A+1$
t_7			写回B: $B=11$
t_8		3、11	

(d) 可串行化调度 (结果同串行调度1)

时间	事务T1	数据库中A、B的值	事务T2
t_0		10、2	
t_1	检索B: $B=2$		
t_2			等待
t_3	修改A: $A \leftarrow B+1$		等待
t_5	写回A: $A=3$		等待
t_6		3、2	检索A: $A=3$
t_7			修改B: $B \leftarrow A+1$
t_8			写回B: $B=4$
t_9		3、4	

为了保证并行操作的正确性：

DBMS的并行控制机制必须提供一定的手段来
保证调度是可串行化的。

从理论上讲，在某一事务执行时禁止其他事务执行的调度策略一定是可串行化的调度，这也是最简单的调度策略，但这种方法实际上是不可行的，因为它使用户不能充分共享数据库资源。

2. 可串行化调度定义：

每个事务中，语句的先后顺序在各种调度中始终保持一致。

在这个前提下：如果一个并发调度的执行结果与某一串行调度的执行结果等价，那么这个并发调度称为“可串行化的调度”，否则是不可串行化的调度。

可串行性（serializable）是并发事务正确性的唯一准则。

为保证并行操作调度的可串行性

目前DBMS普遍采用封锁方法来保证调度的正确性，

另外还有其他一些方法：时标方法、乐观方法等。

三、封锁

封锁是实现并发控制的一个非常重要的技术。

封锁：

事务T在对某个数据对象（表或记录等）操作之前，

先向系统发出请求，对其加锁。

加锁后事务T对该数据对象有了一定的控制，

在事务T释放它的锁之前，其他的事务不能更新此数据对象。

1. 封锁类型

基本的封锁类型有两种：

排它锁（exclusive lock, 简记为X锁）；

共享锁（share lock, 简记为S锁）。

(1) 排它锁(X锁):

如果事务T对某个数据实现X锁，那么其他事务T' 要等T解除X锁以后，才能对这个数据进行封锁。也就是不允许其他事务T' 再对该数据加任何类型的锁。

采用X锁的并发控制并发度低，只允许一个事务独占数据。而其他申请封锁的事务只能排队去等。

为此，降低要求，允许并发的读，就引入了共享型封锁(Shared Lock)，这种锁简称为S锁，又称为读锁。

(2) 共享锁(S锁):

如果事务T对某数据加上S锁后，仍允许其他事务再对该数据加S锁，但在对该数据的所有S锁都解除之前决不允许任何事务对该数据加X锁。

相容矩阵:

T1 \ T2			
	X	S	—
X	N	N	Y
S	N	Y	Y
—	Y	Y	Y

2. 封锁粒度

X锁和S锁都是加在某一个数据对象上的。

封锁的对象可以是逻辑单元，也可以是物理单元。

在关系数据库中，封锁对象可以是：

属性值、属性值集合、元组、关系、索引项、整个索引、整个数据库等逻辑单元；也可以是页（数据页或索引页）、块等物理单元。

封锁对象可以很大，比如对整个数据库加锁，也可以很小，比如只对某个属性值加锁。

封锁对象的大小称为封锁的粒度（granularity）。

封锁粒度与系统的并发度和并发控制的开销密切相关。

封锁的粒度越大，系统中能够被封锁的对象就越少，

并发度也就越小，但同时系统开销也越小；

相反，封锁的粒度越小，并发度越高，但系统开销也就越大。

因此，如果在一个系统中同时存在不同大小的封锁单元供不同的事务选择使用是比较、理想的。

选择封锁粒度时必须同时考虑封锁机构和并发度两个因素，对系统开销与并发度进行权衡，以求得最优的效果。

一般说来，需要处理大量元组的用户事务可以以关系为封锁单元；需要处理多个关系的大量元组的用户事务可以以数据库为封锁单位；而对于一个处理少量元组的用户事务，可以以元组为封锁单位以提高并发度。

3. 封锁协议

封锁的目的是为了保证能够正确地调度并发操作。

在运用X锁和S锁这两种基本封锁，对一定粒度的数据对象加锁时，还需要约定一些规则，例如，应何时申请X锁或S锁、持锁时间、何时释放等。称这些规则为封锁协议（**locking protocol**）。

对封锁方式规定不同的规则，就形成了各种不同的封锁协议，它们分别在不同的程度上为并发操作的正确调度提供一定的保证。

(1) 保证数据一致性的封锁协议

① PX协议和PXC协议——使用X锁的规则。

PX协议：任何事务T在更新记录R之前必须先执行“XFIND R”操作，以获得对R的X锁，才能读或写记录R；如果未获准X锁，那么这个事务进入等待状态。一直到获准X锁，事务才能继续做下去。（如果过早地解锁，有可能使其他事务读了未提交数据（且随后被回退），引起丢失其他事务的更新。

PXC协议： PX协议加上X锁的解除操作应该合并到事务的结束（COMMIT或ROLLBACK）操作中。（可解决丢失更新）

等事务T1更新完成后再执行事务T2：（可解决丢失更新）

时间	事务T1	数据库中A的值	事务T2
t ₀		18	
t ₁	加锁：XFIND A	T1 ♀ 加锁	
t ₂			XFIND A（失败）
t ₃	更新：A:=A-1		等待
T ₄	写回A=17：UPD A		等待
t ₅		17	等待
t ₆	COMMIT(包括解锁)	T1 ♂ 解锁	等待
t ₇		T2 ♀ 加锁	XFIND A（重做）
t ₈			更新：A:=A-2
t ₉			写回A=15：UPD A
T ₁		15	COMMIT(包括解锁)

② PS协议和PSC协议——使用S锁的规则

PS协议： 任何要更新记录R的事务必须先执行“SFIND R”操作，以获得对R的S锁。当事务获准对R的S锁后，若要更新记录R必须用“UPDX R”操作，这个操作首先把S锁升级为X锁，若成功则更新

记录，否则这个事务进入等待队列。

注意： 获准S锁的锁事务只能读数据，不能更新数据，若要更新，则先要把S锁升级为X锁。

PSC协议： PS协议加上S锁的解除操作应该合并到事务的结束（COMMIT或ROLLBACK）操作中。

（PSC协议解决不一致分析和读“脏”数据问题）

解决不一致分析和读“脏”数据问题：

时间	事务T1	数据库中A、B的值	事务T2
t ₀		50、100	
t ₁	加锁 SFIND A, B	T1: ♀A、B加S锁	
t ₂	求和: A+B=150		
t ₃		T2: ♀A、B加S锁	加锁: SFIND B
T ₄			更新: B:=B*2
t ₅			写回B=200 UPDX B 失败
t ₆			等待
t ₇	检索(验算)A=0, B=100		等待
t ₈	求和: A+B=150		等待
t ₉	COMMIT(包括解锁A, B)	T1: ♂ 解锁A, B	等待
t ₁₀		T2: ♀B加X锁	写回B=200:UPDX B重做
t ₁₁		50, 200	COMMIT(包括解锁B)

封锁技术可以有效地解决并行操作的一致性问题；

但也带来新的问题，即活锁和死锁的问题。

“活锁”：系统可能使某个事务永远处于等待状态，得不到封锁

的

机会。

“死锁”：系统中有两个或两个以上的事务都处于等待状态，并

且每个事务都在等待其中另一个事务解除封锁，它才能

继续执行下去，结果造成任何一个事务都无法继续执行。

活 锁:

事务T1	数据A	事务T2	事务T3	事务T4
XFIND A	T1 ♀ 加X锁		。	
。。		XFIND A (失败)		
。		等待	XFIND A (失败)	
解锁 A	T1 ♂ 解锁	等待	等待	XFIND A (失败)
。	T3 ♀ 加X锁	等待	XFIND A (重做)	等待
		等待	。	。
		等待	。	。
		等待	。	。
	T3 ♂ 解锁	等待	解锁 A	等待
		等待	。	等待
	T4 ♀ 加 X 锁	等待	。	XFIND A (重做)
		等待	。	。

避免活锁的 简单方法是：采用先来先服务的策略。

“死 锁”：系统中有两个或两个以上的事务都处于等待状态，并且每个事务都在等待其中另一个事务解除封锁，它才能继续执行下去，结果造成任何一个事务都无法继续执行。

事务T1	数据A、B	事务T2
加锁:XFIND A	T1 ♀ A 加 X锁	
。。	T2 ♀ B 加 X锁	加锁:XFIND B
。		
加锁:XFIND B（失败）		
等待		加锁:XFIND A（失败）
等待		等待
等待		等待
。		。
。		。
。		。

用事务依赖图的形式测试系统中是否存在死锁。

事务依赖图中：每一个结点表示“事务”；

箭头表示事务间的依赖关系。

例:并发执行中两个事务的依赖关系如下图所示:

① 事务T1需要数据B, 但B已被事务T2封锁,

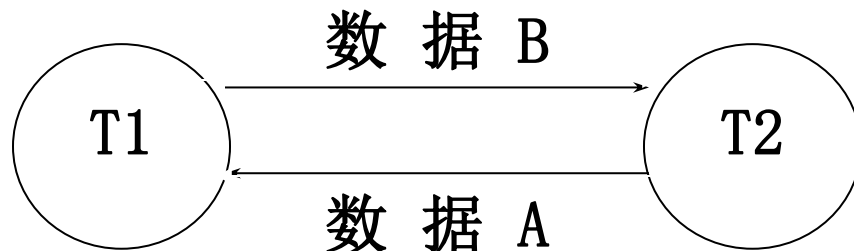
那么从T1到T2画一个箭头;

② 事务T2需要数据A, 但A已被事务T1封锁,

那么从T2到T1也应画一个箭头。

如果在事务依赖图中沿着箭头方向存在一个循环, 那么死锁的条件就形成了, 系统进入死锁状态。

事务依赖图:



DBMS中有一个死锁测试程序：

每隔一段时间检查并发的事务之间是否发生死锁。

如果发生死锁：只能抽取某个事务作为牺牲品，把它撤消，

做回退操作，解除它的所有封锁，恢复到该事务的初始

状态。释放出来的资源就可以分配给其他事务，使其他

事务有可能继续运行下去，就有可能消除死锁现象。

理论上讲，

系统进入死锁状态时可能会有许多事务在相互等待，但是System R的实验表明，实际上绝大部分的死锁只涉及到两个事务，也就是事务依赖图中的循环里只有两个事务。

死锁也被形象地称作“死死拥抱”。

(2) 保证并行调度可串行性的封锁协议——两段锁协议

可串行性是并行调度正确性的唯一准则,两段锁 (two-phase locking) 协议是为了保证并行调度可串行性提供的封锁协议。

两段锁协议规定:

- ①在对任何数据进行读写操作之前,事务必须获得对该数据的封锁
- ② 在释放一个封锁之后, 事务不再获得任何其他封锁。

两段锁的实际含义是事务分为两个阶段:

第一阶段是获得封锁, 也称为扩展阶段;

第二阶段是释放封锁, 也称为收缩阶段。

例: 事务T1的封锁序列是: (遵守两段锁协议)

SFIND A...SFIND B...XFIND C 解锁B... 解锁A... 解锁C;

事务T2的封锁序列是: (不遵守两段锁协议)

SFINDA 解锁A...SFIND B XFIND C... 解锁C... 解锁B;

可以证明，若并行执行的所有事务均遵守两段锁协议，则对这些事务的所有并行调度策略都是可串行化的。

结论： 所有遵守两段锁协议的事务，并行执行的结果
一定是正确的。

注意：事务遵守两段锁协议是可串行化调度的充分条件，而不是必要条件。即可串行化的调度中，不一定所有事务都必须符合两段锁协议。

两段锁协议仍有可能导致死锁，而且会增多，因为每个事务都不能及时解除被它封锁的数据

§ 4 数据库的完整性

教学内容:

- 完整性的定义;
- 完整性子系统的功能;
- 完整性规则的组成;
- SQL中的三大类完整性约束;
- SQL3中的触发器技术。

重点: SQL中的三大类完整性约束;
SQL3中的触发器技术。

数据库的完整性是指数据的正确性和相容性。

数据库的完整性机制：

检查数据库中数据是否满足规定的条件(完整性检查)。

完整性约束条件：数据库中数据应该满足的条件。

也称为完整性规则。

DBMS中执行完整性检查的子系统称为“完整性子系统”。

一、完整性子系统的主要功能：

①监督事务的执行，并测试是否违反完整性规则。

②如有违反现象，则采取恰当的操作。

如拒绝，报告违反情况，改正错误等方法来处理。

二、完整性规则的组成与分类

1. 每个规则由三部分组成：

- ①什么时候使用规则进行检查（规则的“触发条件”）；
- ②要检查什么样的错误（“约束条件”或“谓词”）；
- ③若检查出错误，该怎样处理（“ELSE子句”，即违反时
要做的动作）。

2. 在关系数据库中，完整性规则可分为三类：

- ①域完整性规则：定义属性的取值范围；
- ②基本表约束；
- ③断言。

三、SQL中的完整性约束

1. 域完整性规则：定义属性的取值范围——属性值约束。

包括：域约束子句、非空值约束、基于属性的检查子句。

①用“CREATE DOMAIN”语句定义新的域,并可出现CHECK子句。

例：定义一个新的域DEPT, 可用下列语句实现：

```
CREATE DOMAIN DEPT CHAR (20) DEFAULT '计算机软件'
```

```
CONSTRAINT VALID_ DEPT /*域约束名字*/
```

```
CHECK (VALUE IN ('计算机科学与技术', '计算机软件'));
```

允许域约束上的CHECK子句中可以有任意复杂的条件表达式。

②非空值约束 (NOT NULL)

例： SNO char(4) NOT NULL

③基于属性的检查子句(CHECK)：

例： CHECK (GRADE IS NULL) OR

(GRADE BETWEEN 0 AND 100)

2. 基本表约束:

①主键约束: 主键可用主键子句或主键短语定义;

②外键约束: 用外键子句定义外键:

FOREIGN KEY (〈列名序列1〉) .

REFERENCES 〈参照表〉 [(〈列名序列2〉)]

[ON DELETE 〈参照动作〉]

[ON UPDATE 〈参照动作〉]

其中: 列名序列1是外键;

列名序列2是参照表中的主键或候选键。

参照动作可以有五种方式：

NO ACTION（无影响）、

CASCADE（级联方式）、

RESTRICT（受限方式）、

SET NULL（置空值）

SET DEFAULT（置缺省值）。

③基于元组的检查子句——全局约束 CHECK（条件表达式）

3. 断言：

如果完整性约束与多个关系有关，或者与聚合操作有关，SQL提供“断言”（Assertions）机制让用户书写完整性约束。

断言可以像关系一样，用CREATE语句定义。

定义断言：

CREATE ASSERTION <断言名> CHECK (<条件>)

<条件>与SELECT语句中WHERE子句中的条件表达式一样。

撤消断言： DROP ASSERTION <断言名>

例：设有三个关系模式：

EMP (ENO, ENAME, AGE, SEX, ECITY)

COMP (CNO, CNAME, CITY)

WORKS (ENO, CNO, SALARY)

试用SQL的断言机制定义下列完整性约束：

①每个职工至多可在3个公司兼职工作：

```
CREATE ASSERTION ASSE1 CHECK
```

```
( 3 >= ALL (SELECT COUNT(CNO)
```

```
FROM WORK
```

```
GROUP BY ENO ) ) ;
```

② 每门公司男职工的平均年龄不超过40岁：

```
CREATE ASSERTION ASSE2 CHECK
```

```
(40 >= ALL (SELECT AVG (EMP. AGE)
```

```
FROM EMP, WORK
```

```
WHERE EMP. ENO=WORK. ENO
```

```
AND SEX='男'
```

```
GROUP BY CNO) ) ;
```

③ 不允许女职工在建筑公司工作:

```
CREATE ASSERTION ASSE3 CHECK  
( NOT EXISTS (SELECT *  
                FROM WORK  
                WHERE CNO IN (SELECT CNO  
                               FROM COMP  
                               WHERE CNAME =‘建筑公司’ )  
                AND ENO IN (SELECT ENO  
                              FROM EMP  
                              WHERE SEX=‘女’ )));
```

断言也可以在关系定义中用检查子句形式定义，但是检查子句不一定能保证完整性约束彻底实现，而断言能保证不出差错。

四、SQL3的触发器———主动规则

系统能自动根据条件转去执行各种操作，甚至执行与原操作无关的一些操作。

1. 触发器结构

定义： 触发器（Trigger）是一个能由系统自动执行对数据库修改的语句。触发器有时也称为主动规则（Active Rule）或事件—条件—动作规则（Event—Condition—Action Rule, ECA规则）。

一个触发器由三部分组成：

- ① **事件**：指对数据库的插入、删除、修改等操作。
触发器在这些事件发生时，将开始工作。
- ② **条件**：触发器将测试条件是否成立。如果条件成立，就执行相应的动作，否则什么也不做。
- ③ **动作**：如果触发器测试满足预定的条件，那么就由DBMS执行相应的动作（即对数据库的操作）。

触发器结构的组成：

- ◆ 触发器的命名
- ◆ 触发时间 触发事件 目标表名
- ◆ 旧值和新值的别名表(设置变量)
- ◆ 触发动作间隔时间
- ◆ 动作条件
- ◆ 动作体。

触发器结构的组成：

- (1) 触发时间：BEFORE、 AFTER、 INSTEAD OF ；
- (2) 触发事件：有三类—— UPDATE、 DELETE、 INSERT
- (3) 目标表名：当目标表的数据被更新（插入、删除、修改）时，将激活触发器。
- (4) 旧值和新值的别名表：REFERENCES子句
- (5) 触发动作：

触发事件中的时间关键字有三种：

① **BEFORE**：在触发事件进行前，测试WHEN条件是否满足。若满足则先执行动作部分的操作，然后再执行触发事件的操作（此时可不管WHEN条件是否满足）。

② **AFTER**：在触发事件完成以后，测试WHEN条件是否满足，若满足则执行动作部分的操作。

③ **INSTEAD OF**：在触发事件发生时。只要满足WHEN条件，就执行动作部分的操作，而触发事件的操作不再执行。

触发动作：

①动作时间间隔：FOR EACH ROW 或 FOR EACH STATEMENT。

②动作条件：动作条件用WHEN子句定义

③动作体：当触发器被激活时DBMS要执行的SQL语句。

动作体若是一个SQL语句，直接写上即可；

若是一系列的SQL语句，则用分号定界，再使用BEGIN

ATOMIC.....END限定。

如果触发事件是UPDATE:

应该用 “OLD AS”和 “NEW AS”子句

定义修改前后的元组变量;

如果是DELETE: 只要用 “OLD AS”子句定义元组变量;

如果是INSERT: 只要用 “NEW AS”子句定义元组变量。

触发器有两类：元组级触发器和语句级触发器。

元组级触发器 带 “FOR EACH ROW” 子句，

而语句级触发器没有；

元组级触发器对每一个修改的元组都要检查一次，

而语句级触发器对SQL语句的执行结果去检查。

语句级触发器，不能直接引用修改前后的元组，但可以引用修改前后的元组集。旧的元组集由被删除的元组或被修改元组的旧值组成，而新的元组集由插入的元组或被修改元组的新值组成。

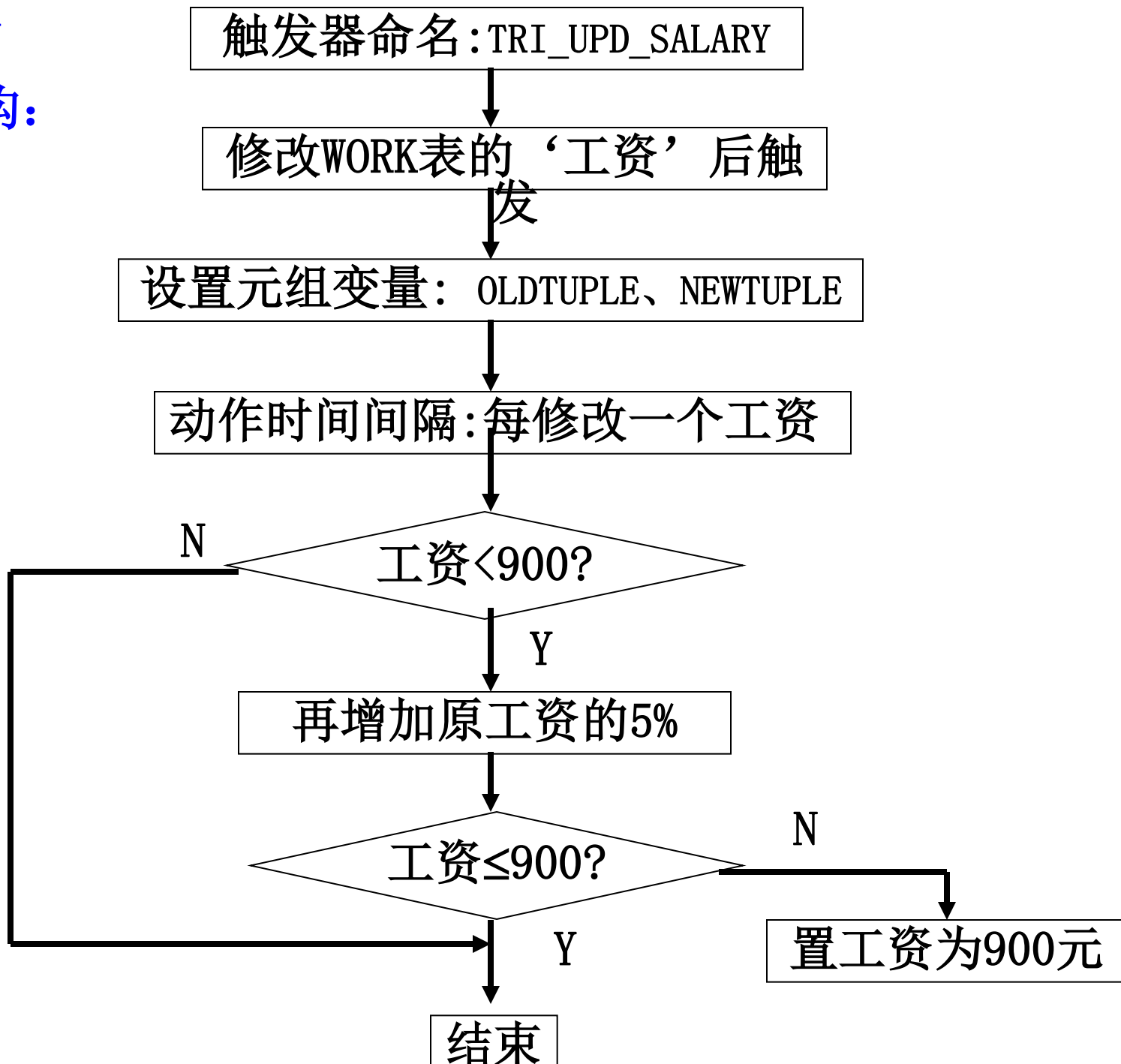
2. SQL3的触发器实例

【实例1】 某单位修改工资原则：如果职工工资修改后仍低于900元，那么，在修改后的工资基础上再增加原工资的5%，但不得超过900元（元组级触发器）。

【实例2】 在学习关系SC表中修改课程号CN0, 即学生的选课登记需作变化。在关系SC中的约束：要求保持每门课程选修人数不超过50。如果更改课程号后，违反这个约束，那么这个更改应该不做。（语句级触发器）

修改工资

触发器结构:



CREAT TRIGGER TRI_UPD_SALARY

/*触发器命名*/

AFTER UPDATE OF SALARY ON WORK

/*触发时间、触发事件、目标表*/

REFERENCING

/*设置必要的变量*/

OLD AS OLDTUPLE

/*为元组级触发器设置变量*/

NEW AS NEWTUPLE

FOR EACH ROW

/*触发器的动作时间间隔*/

WHEN (900 > NEWTUPLE. SALARY)

/*触发器的动作条件部分*/

BEGIN ATOMIC

UPDATE WORK

/*触发器的动作部分1*/

SET SALARY=NEWTUPLE. SALARY+OLDTUPLE. SALARY*0. 5

WHERE ENO=NEWTUPLE. ENO

AND(900 > NEWTUPLE. SALARY + OLDTUPLE. SALARY*0. 5) ;

UPDATE WORK

/*触发器的动作部分2*/

SET SALARY=900

WHERE ENO=NEWTUPLE. ENO

AND(900 ≤ NEWTUPLE. SALARY+OLDTUPLE. SALARY*0. 5) ;

END;

CREAT TRIGGER TRI_UPD_SALARY /*触发器命名*/

AFTER UPDATE OF SALARY ON WORK /*触发时间, 触发事件, 目标表
*/

REFERENCING /*设置必要的变量*/

OLD AS OLDTUPLE /*为元组级触发器设置变量*/

NEW AS NEWTUPLE

FOR EACH ROW /*触发器的动作时间间隔*/

WHEN (900 > NEWTUPLE.SALARY) /*触发器的动作条件部分*/

BEGIN ATOMIC

UPDATE WORK

SET SALARY=NEWTUPLE. SALARY+OLDTUPLE. SALARY*0. 5

WHERE ENO=NEWTUPLE. ENO

AND (900>NEWTUPLE. SALARY+OLDTUPLE. SALARY*0. 5) ;

UPDATE WORK

SET SALARY=900

WHERE ENO=NEWTUPLE. ENO

AND (900≤NEWTUPLE. SALARY+OLDTUPLE. SALARY*0. 5) ;

END ;

【实例2】 在学习关系SC表中修改课程号CNO,即学生的选课登记需作变化。在关系SC中的约束：要求保持每门课程选修人数不超过50。如果更改课程号后，违反这个约束，那么这个更改应该不做。（语句级触发器）

```
CREATE TRIGGER TRI_UPD_SC                                /*触发器的命名*/
  INSTEAD OF UPDATE OF CNO ON SC                         /*时间、事件、目标*/
  REFERENCING                                           /*设置变量*/
    OLD_TABLE AS OLDSTUFF                               /*为语句级触发器设置变量*/
    NEW_TABLE AS NEWSTUFF
  WHEN (50 >= ALL (SELECT COUNT(SNO)                   /*动作时间条件*/
    FROM ((SC EXCEPT OLDSTUFF) UNION NEWSTUFF)
    GROUP BY CNO) ) )
  BEGIN ATOMIC                                           /*动作体*/
    DELETE FROM SC                                       /*触发动作1*/
    WHERE (SNO, CNO, GRADE) IN OLDSTUFF;
    INSERT INTO SC                                       /*触发动作2*/
    SELECT * FROM NEWSTUFF
  END;
```

```
CREATE TRIGGER TRI_UPD_SC /*触发器的命名*/  
  
INSTEAD OF UPDATE OF CNO ON SC /*时间, 事件, 目标*/  
  
REFERENCING /*设置变量*/  
  
    OLD_TABLE AS OLDSTUFF /*为语句级触发器设置变量*/  
  
    NEW_TABLE AS NEWSTUFF  
  
WHEN (50 >= ALL (SELECT COUNT(SNO) /*动作时间条件*/  
  
    FROM ((SC EXCEPT OLDSTUFF) UNION NEWSTUFF)  
  
    GROUP BY CNO) ) )
```

BEGIN ATOMIC

/*动作体*/

DELETE FROM SC

/*触发动作1*/

WHERE (SNO, CNO, GRADE) IN OLDSTUFF;

INSERT INTO SC

/*触发动作2*/

SELECT * FROM NEWSTUFF

END;

触发器的动作时间:INSTEAD OF（第2行），任何企图修改关系SC中CNO值都被这个触发器截获，并且触发事件的操作（即修改CNO）不再进行，由触发器的条件真假值来判断是否执行动作部分的操作。

INSTEAD OF 表示：在触发事件发生时,只要满足WHEN条件，就执行动作部分的操作，而触发事件的操作不再执行。

动作部分的操作由两个SQL语句组成，前一个语句是从关系SC中删除修改前的元组，后一个语句是在关系SC中插入修改后的元组。用这样的方式完成触发事件的操作。

因为是语句级触发器，所以没有FOR EACH ROW，在这里FOR EACH STATEMENT也省略了。

五、SQL Server的数据库完整性及实现方法

SQL Server具有较健全的数据库完整性控制机制。

SQL Server使用约束、缺省，规则和触发器4种方法

定义和实施数据库完整性功能。

1. SQL Server的数据完整性的种类

SQL Server中的数据完整性包括 域完整性、实体完整性和参照完整性3种。

- (1) 域完整性为列级和元组级完整性----为列或列组指定一个有效的数据集，并确定该列是否允许为空。
- (2) 实体完整性为表级完整性---要求表中所有的元组都应该有一个唯一的标识符(主码)。
- (3) 参照完整性是表级完整性---维护参照表中的外码与被参照表中主码的相容关系。

2. SQL Server数据完整性的两种方式

SQL Server 使用声明数据完整性和过程数据完整性两种方式实现 数据完整性控制。

(1) 声明数据完整性：通过在对象定义中定义、系统本身自动

强制来实现。声明数据完整性包括各种约束、缺省和规则。

(2) 过程数据完整性：通过使用脚本语言（主语言或

TransactSQL）定义，系统在执行这些语言时强制完整性实现。

过程数据完整性包括触发器和存储过程等。

3. SQL Server 实现数据完整性的具体方法有4种：

约束、缺省、规则和触发器。

(1) SQL约束类型————效率高

可在定义、修改表语句中定义。

约束是通过限制列中的数据、行中的数据和表之间数据

来保证数据完整性的方法：

(2) 缺省和规则—————功能较低开支大

缺省（**DEFAULT**）和规则（**RULE**）都是数据库对象。当它们被创建后，可以绑定到一列或几列上，并可以反复使用。

(3) 触发器—————高功能高开支的数据完整性方法

① **Inserted**和**deleted** 表

当触发器被执行时，SQL Server 创建一个或两个临时表（**Inserted**或者**deleted** 表）。当一个记录插入到表中时，相应的插入触发器创建一个**inserted**表, 该表镜像该触发器相连接的表的列结构。

② **Update ()** 函数

Update () 函数只在插入和更新触发器中可用，它确定用户传递给它的列是否已经被引起触发器激活的**insert**或**update**语句所作用。

(3) 触发器——高功能高开支的数据完整性方法

① Inserted和deleted 表

当触发器被执行时，SQL Server 创建一个或两个临时表（Inserted或者deleted 表）。当一个记录插入到表中时，相应的插入触发器创建一个inserted表, 该表镜像该触发器相连接的表的列结构。

② Update () 函数

Update () 函数只在插入和更新触发器中可用，它确定用户传递给它的列是否已经被引起触发器激活的insert或update语句所作用。

实例3:限制修改考试成绩(GGRADE),不能低于原考试成绩。

```
CREATE TRIGGER SC_UPDATA_GGRADE ON [dbo].[SC]
FOR UPDATE                                /*事件*/
AS
DECLARE @old_ggrade real,                /*定义变量*/
        @new_ggrade real
BEGIN                                    /*动作体*/
    SELECT  @old_ggrade =ggrade
    FROM    deleted
    SELECT  @new_ggrade = ggrade
    FROM    inserted
    IF update(ggrade)                    /*条件*/
        IF @old_ggrade > @new_ggrade
            ROLLBACK TRANSACTION        /*动作*/
END
```

过程数据完整性： 通过使用脚本语言（主语言或 TransactSQL）定义，系统在执行这些语言时强制完整性实现。

过程数据完整性包括触发器和存储过程等。

◆ 存储过程的使用

(上机验证 P.290存储过程举例)



§ 5 数据库的安全性

教学内容:

- 安全性的定义、级别，权限；
- SQL中的安全性机制。

数据库的安全性（Security）：

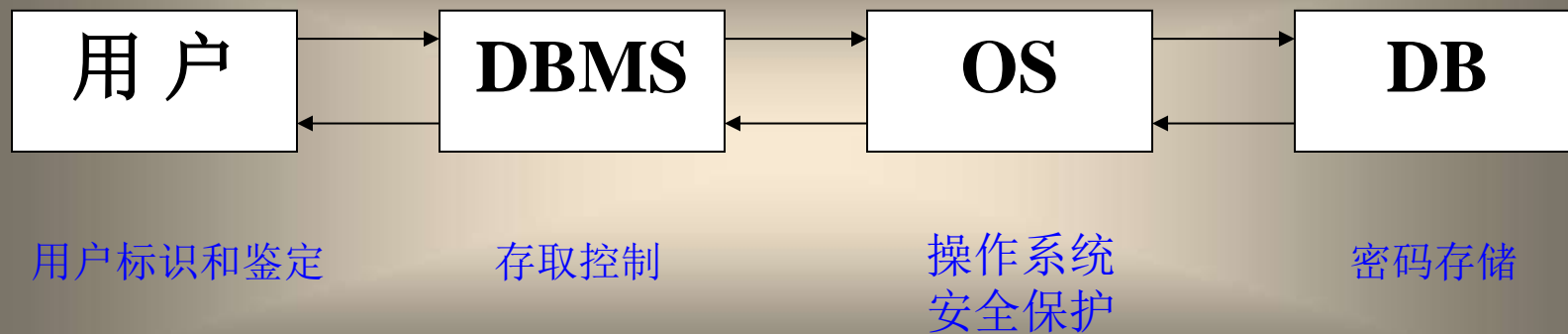
是指保护数据库，防止不合法的使用，
以免数据的泄密、更改或破坏。

对数据库不合法的使用，称为数据库的滥用。

为了保护数据库，防止恶意的滥用，可以在从低到高的五个级别上设置各种安全措施：

- (1) **环境级：** 计算机系统的机房和设备应加以保护，防止有人进行物理破坏；
- (2) **职员级：** 工作人员应清正廉洁, 正确授予用户访问数据库的权限。
- (3) **OS级：** 应防止未经授权的用户从OS处着手访问数据库；
- (4) **网络级：** 由于大多数DBS都允许用户通过网络进行远程访问，因此网络软件内部的安全性是很重要的；
- (5) **DBS级：** DBS的职责是检查用户的身份是否合法及使用数据库的权限是否正确。

一、计算机系统安全模型：



1. 用户标识和鉴定

用户标识和鉴定是系统提供的最外层安全保护措施。其方法是由系统提供一定的方式让用户标识自己的名字或身份。

- (1) 用输入用户名（用户标识号）来标明用户身份；
- (2) 通过回答口令（Password）标识用户身份；
- (3) 通过回答对随机数的运算结果表明用户身份。

2. 存取控制

DBMS的存取控制机制是数据库安全的一个重要保证，它确保具有数据库使用权的用户访问数据库，同时令未被授权的人员无法接近数据。

(1)存取机制的构成 存取控制机制主要包括两部分：

- ①定义用户权限，并将用户权限登记到数据字典中。
- ②当用户提出操作请求时，系统进行权限检查，拒绝用户的非法操作

(2)存取机制的类别

- ①自主存取控制（**DAC**）；
- ②强制存取机制（**MAC**）。

用户（或应用程序）使用数据库（DB）的方式称为“权限”。

用户访问DB有四种权限：

- ♥读(Read)权限：允许用户读数据，但不能修改数据。
- ♥插入(Insert)权限：允许用户插入新的数据，但不能修改数据。
- ♥修改(Update)权限：允许用户修改数据，但不能删除数据。
- ♥删除(Delete)权限：允许用户删除数据。

用户修改DB模式的权限：

- ♥索引(Index)权限：允许用户创建和删除索引。
- ♥资源(Resource)权限：允许用户创建新的关系。
- ♥修改(Alteration)权限：允许用户在关系结构中加入或删除属性。
- ♥撤消(Drop)权限：允许用户撤消关系。

二、SQL中的安全性机制

SQL中有两个机制提供了安全性：

视图机制： 用来对无权用户屏蔽数据；

授权子系统： 允许有特定存取权的用户有选择地和动态地把这些权限授予其他用户。

1. 视图

视图（View）是从一个或多个基本表导出的表。但视图仅是一个定义，视图本身没有数据，不占磁盘空间。视图一经定义就可以和基本表一样被查询，也可以用来定义新的视图，但更新（插、删、改）操作将有一定限制。

SQL的视图机制使系统具有三个优点：

数据安全性，

逻辑独立性，

操作简便性。

创建视图句法： CREATE VIEW 视图名（列名表）

AS SELECT 查询语句；

例：对工程项目零件供应数据库，用户经常要用到有关项目使用零件情况信息：工程号、工程项目名称、供应商号、供应商名、零件号、零件名、供应数量等列的数据。可用下列语句建立视图：

```
CREAT VIEW JSP_NAME (JNO, JNAME, SNO, SNAME, PNO, PNAME, QTY)
  AS SELECTS (J. JNO, JNAME, S. SNO, SNAME, P. PNO, PNAME, QTY)
  FROM S, P, J
WHERE  S. SNO=SPJ. SNO
AND  P. PNO=SPJ. PNO
AND  J. JNO=SPJ. JNO
AND  J. JNAME=:USER ;
```

视图的撤消句法如下：

```
DROP VIEW 视图名；
```

例：撤消JSP_NAME视图：

```
DROP VIEW JSP_NAME；
```

2. SQL中的用户权限及其操作

(1) 用户权限

SQL定义了六类权限供用户选择使用：

SELECT INSERT DELETE UPDATE REFERENCES USAGE

(2) 授权语句

授予其他用户使用关系和视图的权限的语句：

GRANT <权限表> ON <数据库元素>

TO <用户名表> [WITH GRANT OPTION]

(3) 回收语句

从其他用户回收权限：

REVOKE <权限表> ON <数据库元素> FROM <用户名表>

[RESTRICT | CASCADE]

例1: 把对关系S的查询、修改权限授给用户WANG, 并且WANG还可以把这些权限转授给其他用户:

```
GRANT  SELECT, UPDATE  ON  S  TO  WANG  
WITH  GRANT  OPTION
```

例2: 允许用户BAO建立新关系, 并可以引用关系C的主键CNO作为新关系的外键, 并有转让权限。

```
GRANT  REFERENCES  (CNO)  ON  C  TO  BAO  
WITH  GRANT  OPTION
```

例3: 从用户WANG连锁回收对关系S的查询、修改权限。

```
REVOKE  SELECT, UPDATE  ON  S  FROM  WANG  CASCADE
```

三、数据加密法

为了更好地保证数据库的安全性，可用密码存储口令和数据，数据传输采用密码传输防止中途非法截获等方法。我们把原始数据称为源文，用加密算法对源文件进行加密。加密算法有两种：普通加密法和明键加密法。

1、普通加密法

加密算法的输入是源文和加密键，输出是密码文。加密算法可以公开，但加密键是一定要保密的。密码文对于不知道加密键的人来说，是不容易解密的。

例： 设源文是PHYSICIST，加密键是LIGHT。具体的加密算法操作步骤为：

① 把源文分成等长的块，每块的长度和加密键的长度一样。空格用符号b表示（为简化操作，这里只处理大写英文字母和空格）：

PHYSI CIST+

② 对源文的每个字符用0-26中一个整数替换， += 00， A = 01， ...
， Z = 26： 1608251909 0309192000

③ 对加密键LIGHT也做同样的替换，替换为： 1209070820。

④ 对每块源文的每个字符的整数码和加密键相应字符的整数码以27为模相加：

	1608251909	0309192000
+)	1209070820	1209070820
	0117050002	1518260120

⑤ 用相应字符代替整数码，得到密码文：
AQEbB ORZAT

(2) 明键加密法:

公开加密算法和加密键，但相应的解密密键是保密的。因此明键法有两个键，一个用于加密，一个用于解密。

明键加密法具体步骤如下:

① 任意选择两个100位左右的质数 p 和 q ，计算 $r=p*q$ 。

② 任意选择一个整数 e ，而 e 与 $(p-1) * (q-1)$ 是互质的，
把 e 作为加密键（一般，比 q 和 p 大的质数就可选作为 e ）。

③ 求解密键 d ，使得 $(d*e) \bmod ((p-1)*(q-1))=1$ 。

④ r 和 e 可以公开，但 d 是保密的。

⑤ 对源文 p 进行加密，得到密码文 c ，计算公式是 $c=p^e \bmod r$

⑥ 对密码文 c 进行解密，得到源文 p ，计算公式是 $p=C^d \bmod r$

由于只公开 r 和 e ，而求 r 的质因子几乎是不可能的，因为从 r 、 e 求 d 也几乎不可能

这个方法的依据是：

① 已经存在一快速算法, 能测试一个大数是不是质数;

②还不存在一个快速算法, 去求一个大数的质因子。曾有人计算过。测试一个130位整数是否是质数, 计算机约需7分钟时间, 但在同样机器上, 求两个63位质数的乘积的质因子约要花 4×10^{16} 年时间。

举例：设 $p=3$ ， $q=5$ ； $r=p*q=15$ ， $(p-1)*(q-1)=8$ 。

设 $e=11$ (比 q 和 p 大的质数)，从 $(d*11) \bmod 8 = 1$

求得： $d=3$ 。 则加密键： $e=11$ ，解密键： $d=3$ 。

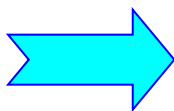
如果源文： $p=13$ ，那么密码文 c 可从下式获得：

$$c = P^e \bmod r = 13^{11} \bmod 15 = 1792160394037 \bmod 15 = 7$$

从密码文： 7 可用下式求得源文 p ：

$$p = C^d \bmod r = 7^3 \bmod 15 = 343 \bmod 15 = 13$$

明键加密法已广泛应用于DBS中，还被用于“ ” 数字签名 “以识别用户的真伪。





精读和习题要求

精 读： 教材P. 173 ~194

习 题8： P. 195 3、 5 ~ 9

15 ~ 18

19 ~20