事务与并发控制


PART 01
事务的概念及特性


PART 02
事务并发执行的问题


PART 03
可串行化调度


PART 04
冲突/视图可串行化


PART 05
可串行性判定


PART 06
封锁


PART 07
两段锁协议


PART 08
活锁与死锁

产生不一致的结果

Read（A）

A=A*1.1

Write(A)

并发执行的错误

T1	T2
READ（A）	
	READ（A）
A：=A*0.1 WRITE（A）	
	A：=A+10 WRITE（A） COMMIT
COMMIT	

何时更新数据库

内存与磁盘之间的I/O操作由操作系统完成，程序无从知晓内存中数据何时写入磁盘。

- 允许多个用户同时使用的数据库系统
  - 飞机定票数据库系统
  - 银行数据库系统
  - 特点：在同一时刻并发运行的事务数可达数百上千个

- 事务

事务是用户自定义的一个数据库操作序列。这些操作要么全做要么全不做，是一个不可分割的工作单位。

- 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
```

```
Begin Tran
read A
A=A-x
If A<0 then
begin
        display “A余额不足”
         rollback
 end
else
begin
       write(A)
        read B
         B=B+x
         write(B)
         display  “缴费成功”
        commit
end
End  Tran
```

● 原子性（Atomicity）

事务的所有操作在数据库中要么全部正确反映，要么全部不反映。

● 隔离性（Isolation）

当多个事务并发执行时，一个事务的执行不能被其他事务干扰。

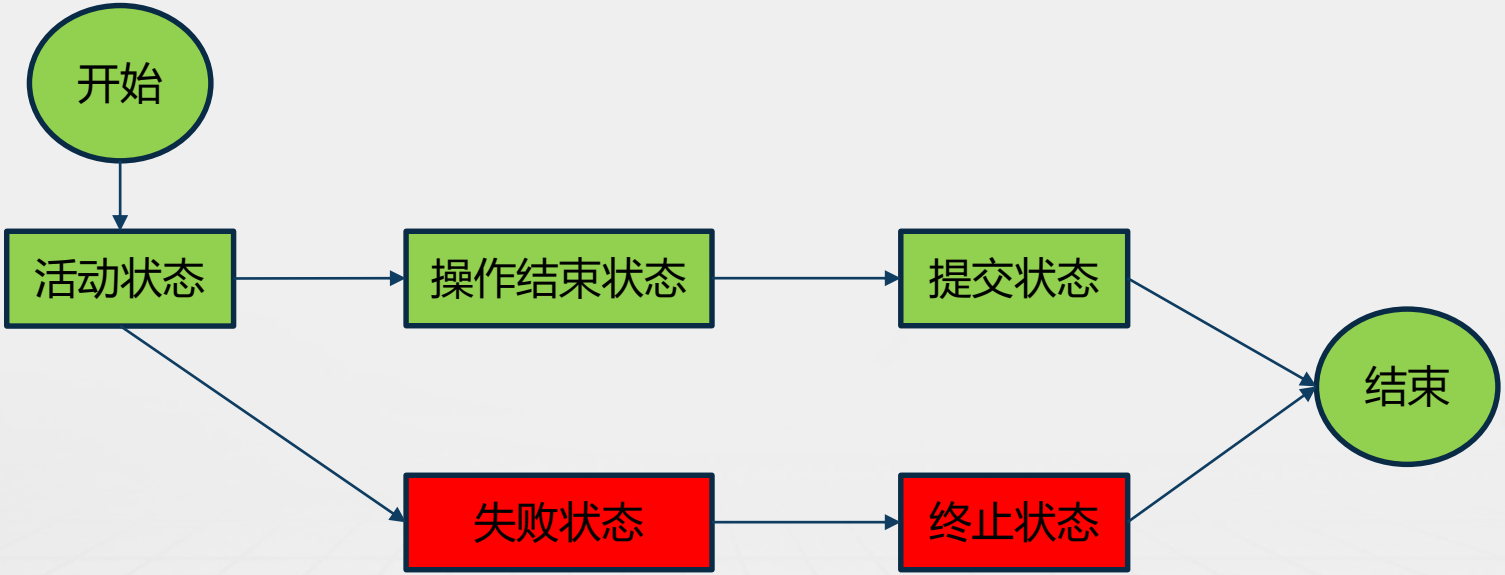
● 一致性（Consistency）

当事务完成时，必须使所有数据都具有一致的状态。

● 持续性（Durability）

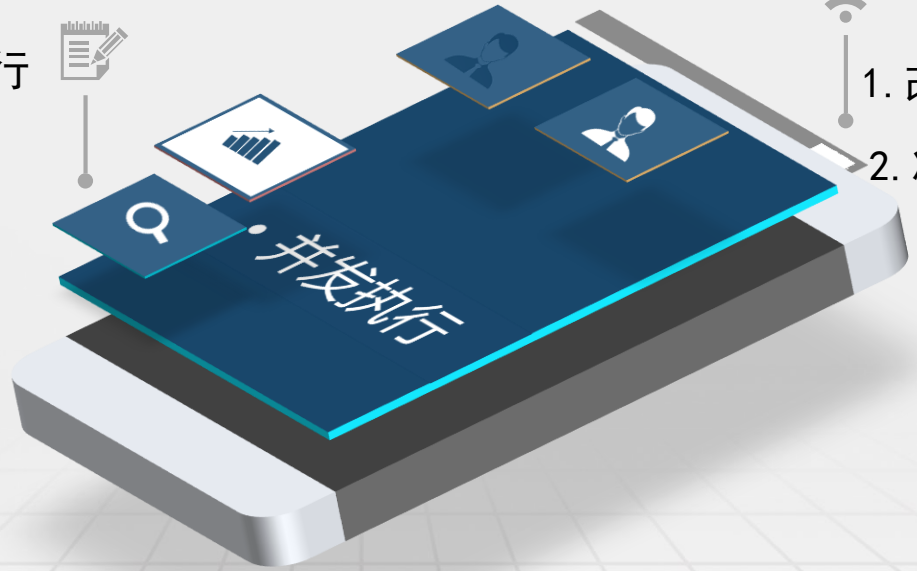
一个事务一旦提交，它对数据库中数据的改变应该是永久性的，即使系统可能出现故障。

- 事务的状态变迁



2. 事务并发执行的问题

I/O与CPU等可以并行交叉运行

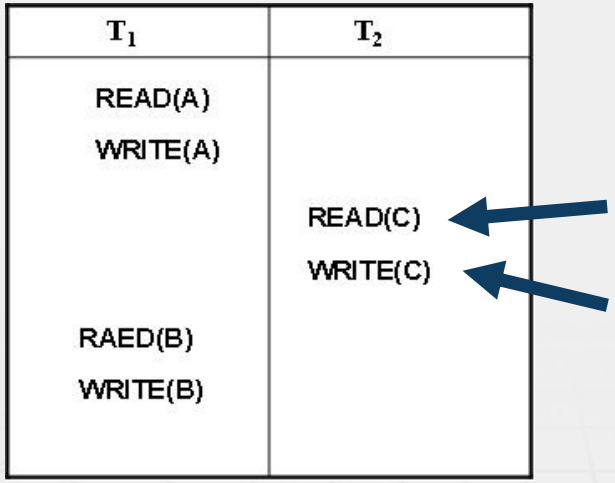


并发执行的优点

1. 改善系统的资源利用率
2. 减少短事务的等待时间

- 调度

一个或多个事务的操作按时间排序的一个序列



一个事务的两个操作在调度中出现的顺序必须与其在事务内定义的先后顺序一致。

读脏数据（dirty read）

脏数据(dirty data)是对未提交事务所写数据的统称。

T1	T2
READ(A)	
A:=A*0.1	
WRITE(A)	
	READ(A)
	COMMIT
ROLLBACK	

若脏读带来的影响足够小，偶尔可读一次脏数据。它可以提高并发性，减少事务的等待时间

若脏读就造成了数据库的不一致状态，应严格禁止。

不可重复读（ unrepeatable read）

在同一事务内部前后多次读同一数据得到不同的结果。

T1	T2
READ(A)	
READ(B)	
READ(C)	
READ(D)	READ(A)
READ(E)	A:=A*0.1
…..	WRITE(A)
…….	COMMIT
READ(A)	
…….	
…..	

三类不可重复读

事务1读取某一数据后：
1. 事务2对其做了修改，当事务1再次读该数据时，得到与前一次不同的值。

2. 事务2删除了其中部分记录，当事务1再次读取数据时，发现某些记录神密地消失了。
3. 事务2插入了一些记录，当事务1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幻影现象（Phantom Row）

丢失更新（ lost update ）

一个事务对数据的更新结果被后提交事务的结果覆盖，没有在数据库得到体现

T1	T2
READ（A）	
	READ（A）
A：=A*0.1 WRITE（A）	
	A：=A+10 WRITE（A） COMMIT
COMMIT	

3. 可串行化调度

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)

最终写入数据库的药品A、B的价格为43元和16元

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

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

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)

最终写入数据库的药品A、B的价格为44元和12元

●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

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)

并行调度与串行调度的结果相同，交叉调度是可串行的调度

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

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

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)

与两个串行调度的结果都不一致，交叉调度是一个不可串行化的调度。

丢失更新!

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)

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。

- 简记符号

WRITE简写为W,
READ简写为R,
WT(X)：事务T写数据库元素X,
RT(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)

- 读写不同数据：不冲突

调度中两个事务发生冲突，必须：
- **对同一数据对象进行操作**
- **两个操作指令中有一个是写操作W**

- 读相同数据：不冲突

若事务Ti 和Tj都是读取数据A，则Ri(A)，Rj(A)指令不发生冲突。

- 读写相同数据：冲突

若事务Ti 和Tj一个是读数据，一个是写数据，则事务的执行顺序是重要的。
Ri(A)和Wj(A)指令是冲突的。

- 写相同数据：冲突

若事务Ti 和Tj都是写数据A，则Wi（A）和Wj（A）指令也是冲突的。

示例

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

- 冲突指令

  R2(A) W1(A)

  W1(A) W2(A)

  R1(B) W2(B)

  W2(B) W1(B)

- 非冲突指令

  R1(A) R2(A)

  R2(B) R1(B)

若调度S中属于不同事务的两条操作指令是不冲突的，则可以交换两条指令的执行顺序，得到一个新的调度S′。称调度S与调度S′冲突等价的（conflict equivalent）。

S= R_1（A）W_1（A）R_2（A）W_2（A）R_1（B）W_1（B）R_2（B）W_2（B）

S1= R_1（A）W_1（A） R_1（B） R_2（A）W_2（A）W_1（B）R_2（B）W_2（B）

S2= R_1（A）W_1（A） R_1（B） R_2（A）W_2（A）W_1（B）R_2（B）W_2（B）

S3= R_1（A）W_1（A） R_1（B） W_1（B） R_2（A）W_2（A）R_2（B）W_2（B）

若一个调度冲突等价于一个串行调度，则该调度是冲突可串行化的。

S= R_1（A）W_1（A）R_2（A）W_2（A）R_1（B）W_1（B）R_2（B）W_2（B）

S1= R_1（A）W_1（A） R_1（B） R_2（A）W_2（A）W_1（B）R_2（B）W_2（B）

S2= R_1（A）W_1（A） R_1（B） R_2（A）W_2（A）W_1（B）R_2（B）W_2（B）

S3= R_1（A）W_1（A） R_1（B） W_1（B） R_2（A）W_2（A）R_2（B）W_2（B）

调度S等价于串行调度S3，是冲突可串行化的。

对同一事务集，如果两个调度S1和S2在任何时候都保证每个事务读取相同的值，写入数据库的最终状态也是一样的，则称调度S1和S2视图等价。

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)

左右两个调度不是视图等价的

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

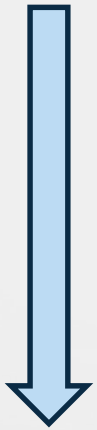
左右两个调度是视图等价的

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

如果某个调度视图等价于一个串行调度，则称这个调度是视图可串行化的

如果调度是冲突可串行化的，则该调度一定是视图可串行化的。但反过来未必成立。

S1= R1(A) W3(A) R2(B) W1(B)



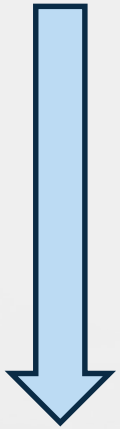
冲突等价

冲突可串行化的

视图可串行化

S2 = R2(B) R1(A) W1(B) W3(A)

$S=R_1(A)\ W_2(A)\ W_1(A)\ W_3(A)$



视图等价

不是冲突可串行化的

串行调度T1->T2->T3

前驱图是一个有向图G=（V，E）：

顶点代表调度S 中的事务；由Ti→Tj 的边表示在调度S中Ti 和Tj之间存在一对冲突指令，并且Ti中的指令先于Tj 中的指令执行

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

若前驱图中存在环，则表示调度S是不可串行化的；否则，表示调度S是冲突可串行化的，可用拓扑排序得到调度S 的一个等价的串行调度

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

数据库系统要求所有的调度都是可恢复的

T1	T2
READ(A) A:=A*0.1 WRITE(A) ROLLBACK	 READ(A) COMMIT

- 可恢复条件

调度S中，事务Ti如果读取了事务Tj修改过的数据，则事务Ti必须等事务Tj提交后才能提交

S1=R1（A）W1（A）R2（A）W1（B）W2（B）C1 C2

调度S1是可恢复的，可串行的

S2=R1（A）W1（A）R2（A）W2（B）W1（B）C1 C2

调度S2是可恢复的，不可串行的

S3=R1（A）W1（A）R2（A）W1（B）W2（B）C2 C1

调度S3是不可恢复的，可串行的

假定T2事务读取A 的值并修改；
还有T3事务读取T2修改后的值，并做了修改；依次类推...
若事务T1发生故障时，后续的事务T2、T3、T4...都已提交，则事务T1的回滚导致级联回滚，产生大量的撤销工作。

●无级联回滚条件

调度S中的每对事务Ti和Tj，事务Ti如果读取了事务Tj修改过的数据，则事务Tj必须在Ti读取前提交

●事务隔离性级别

- 可串行化（Serializable）：保证可串行化调度。
- 可重复读（Repeatable Read）：只允许读取已提交数据，且在一个事务两次读取一个数据项期间，其他事务不得更新数据。
- 已提交读（Read Commited）：只允许读取已提交数据，但不要求可重复读。
- 未提交读（Read Uncommited）：允许读取未提交数据，这是SQL允许的最低一致性级别。
- 以上所有隔离性级别都不允许“脏写”。

多个事务并发执行时，为保持事务的隔离性，DBMS必须对并发事务之间的相互影响加以控制。这种控制是通过一种叫并发控制的机制来实现的。
确保事务隔离性的方法之一是要求对数据的访问以互斥的方式进行。即，当一个事务访问某一数据时，其它事务不能对该数据进行修改。实现该需求最常用的方法就是在访问数据前先持有该数据上的锁。

- 锁管理器

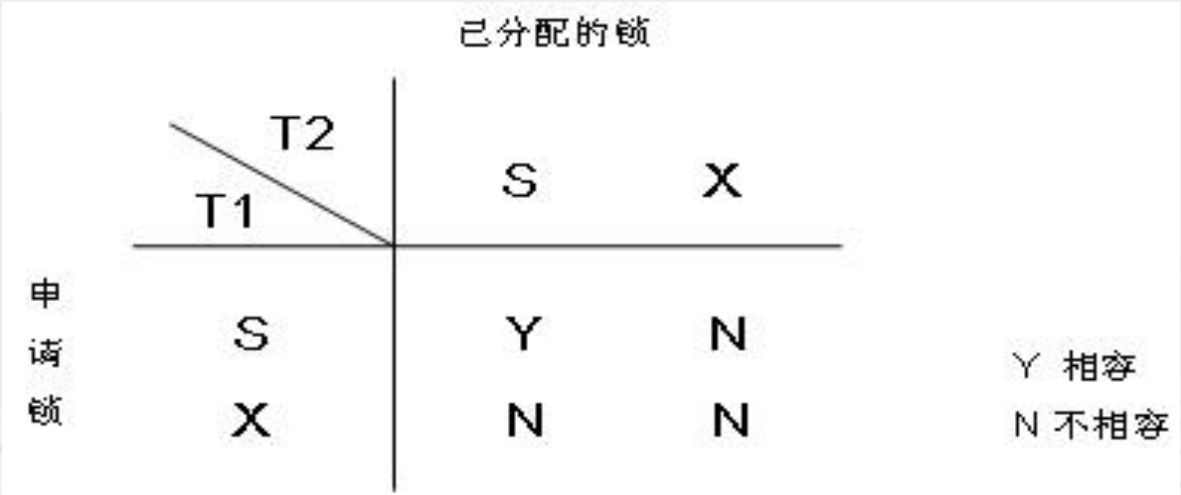
事务执行过程中锁的申请和释放由DBMS中的锁管理器负责。

- 锁表

锁管理器维护一张哈希表，表内信息包括：每个数据库对象上已有的锁的个数、锁的类型以及一个指向申请锁队列的指针。

共享锁（S锁）：如果事务Ti申请到数据项Q的共享锁，则Ti可以读数据项Q，但不能写Q。

排它锁（X锁）：如果事务Ti申请到数据项Q的排它锁，则Ti可以读数据项Q，也可以写Q。



共享锁（S锁、读锁）：若事务T对数据对象Q加上S锁，事务T可读但不能写Q，其它事务只能再对Q加S锁，而不能加X锁，直到T释放Q上的S锁。

排它锁（X锁、写锁）：若事务T对数据对象Q加上X锁，则事务T既可以读又可以写Q，其它任何事务都不能再对Q加任何类型的锁，直到T释放Q上的锁。

T_1 \ T_2	X	S	-
X	N	N	Y
S	N	Y	Y
-	Y	Y	Y

Y=Yes，相容的请求
N=No，不相容的请求

- 在运用X锁和S锁对数据对象加锁时，需要约定一些规则：封锁协议（Locking Protocol）
  - 何时申请X锁或S锁
  - 持锁时间、何时释放
- 不同的封锁协议，在不同的程度上为并发操作的正确调度提供一定的保证。

加入封锁的调度

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

(a)

T1	T2
XLOCK(A) READ(A) A=A+A*0.1 WRITE(A) XLOCK(B) READ(B) B=B+B*0.2 WRITE(B) UNLOCK(A) UNLOCK(B)	XLOCK(A) 等待 … … READ(A) A=A+10 WRITE(A) XLOCK(B) READ(B) B=B-20 WRITE(B) UNLOCK(A) UNLOCK(B)

(b)

T1	T2
XLOCK(A) READ(A) A=A+A*0.1 WRITE(A) XLOCK(B) 等待 ……	XLOCK(B) READ(B) B=B-20 WRITE(B) XLOCK(A) 等待 ……

(c)

更新后立即释放锁，可能脏读

事务的最后释放锁，避免脏读和确保可串行性。但降低并发度

相互等待出现死锁

- 两段锁协议（two-phase locking protocol，2PL）

指所有事务分两个阶段提出加锁和解锁申请:

增长阶段（growing phase）：在对任何数据进行读、写操作之前，首先申请并获得该数据的封锁；

收缩阶段（shrinking phase）：在释放一个封锁后，事务不再申请和获得其它的任何封锁。

- 两段锁协议是保证冲突可串行化的充分条件，但该协议不保证不发生死锁。

8. 两段锁协议

示例

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

(a)

非两段锁，可串行化

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

(b)

两段锁协议

T1	T2
XLOCK(A) READ(A) A:=A+A*0.1 WRITE(A) XLOCK(B) 等待 ..... XLOCK(B) READ(B) B:=B-20 WRITE(B) XLOCK(A) 等待 ....

(c)

两段锁协议

T1	T2	T3
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) UNLOCK(A)	SLOCK(A) READ(A)

级联回滚

严格两段锁协议

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
- 避免了脏读和丢失修改的问题

强两段锁协议

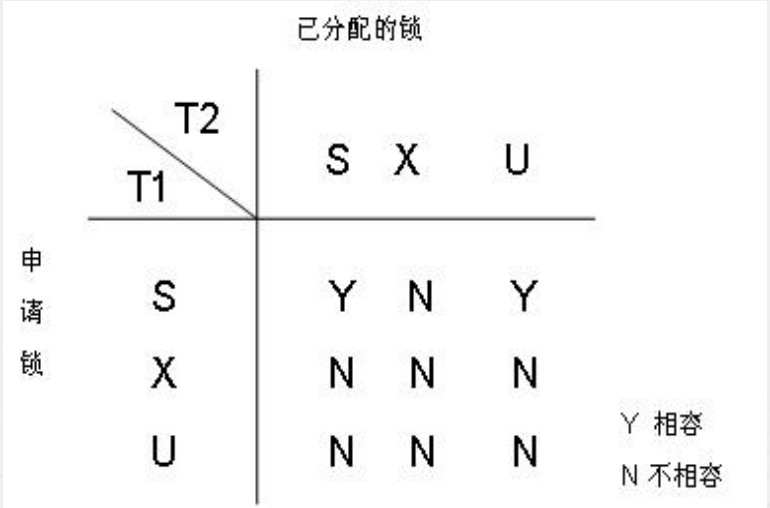
除要求满足两段锁协议规定外，还要求事务的所有锁都必须在事务提交之后释放。

- 进一步解决数据项不能重复读的问题

- 锁的升级及更新锁

U锁：只允许事务读取数据项而不能修改

T_1	T_2
	SLOCK(A)
	READ(A)
SLOCK(A)	
READ(A)	
SLOCK(B)	
A:=A+A*0.1	
B:B+B*0.2	
XLOCK(B)	
WRITE(B)	
XLOCK(A)	
等待	
…	
	UNLOCK(A)
WRITE(A)	
UNLOCK(B)	
UNLOCK(A)	



活锁（饿死）

- 事务T_1封锁了数据R
- 事务T_2又请求封锁R，于是T_2等待。
- T_3也请求封锁R，当T_1释放了R上的封锁之后系统首先批准了T_3的请求，T_2仍然等待。
- T_4又请求封锁R，当T_3释放了R上的封锁之后系统又批准了T_4的请求……
- T_2有可能永远等待，这就是**活锁**的情形

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	.	.

T1	T2	T3	T4	T5
SLOCK(A) READ(A)				
	XLOCK(A) 等待			
		SLOCK(A) READ(A)		
UNLOCK(A)				
	等待		SLOCK(A) READ(A)	
	等待		UNLOCK(A)	SLOCK(A) READ(A)
		UNLOCK(A)		
	等待			

- 解决活锁方法

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

- 事务T_1封锁了数据R_1
- T_2封锁了数据R_2
- T_1又请求封锁R_2，因T_2已封锁了R_2，于是T_1等待T_2释放R_2上的锁
- 接着T_2又申请封锁R_1，因T_1已封锁了R_1，T_2也只能等待T_1释放R_1上的锁
- 这样T_1在等待T_2，而T_2又在等待T_1，T_1和T_2两个事务永远不能结束，形成**死锁**

T_1	T_2
Lock R_1	• • •
•	Lock R_2
• •	• •
Lock R_2	•
等待	
等待	
等待	Lock R_1
等待	等待
等待	等待
	• • •

T1	T2
SLOCK(A)	
READ(A)	XLOCK(B)
	READ(B)
XLOCK(B)	
等待	
	XLOCK(A)
	等待

死锁的两种处理方式：

一种是进行死锁的预防，不让并发执行的事务出现死锁的状况；

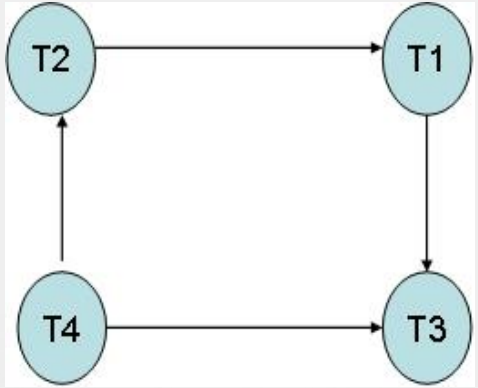
一种是允许死锁的发生，在死锁出现后采取措施解决，为此系统中需增加死锁的检测及死锁的解除算法。

等待图法

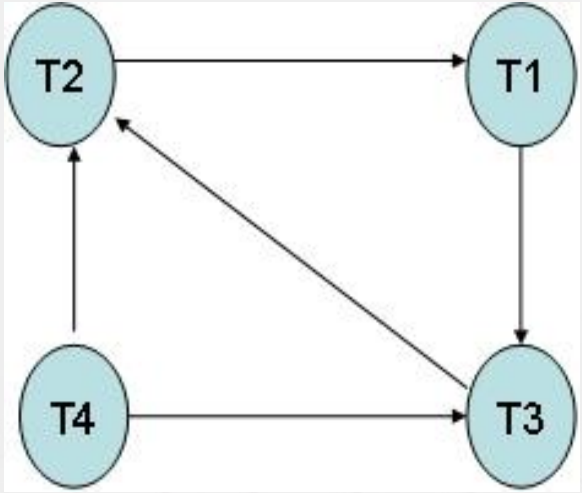
G=（U，V）是一个有向图。顶点U为当前系统中运行事务T的集合，V是边的集合，表示事务的等待情况。

当且仅当等待图中出现环路时，表示系统中存在死锁。环路中的每个事务都处于死锁状态。

T1	T2	T3	T4
SLOCK(A)	SLOCK(D)		
	SLOCK(B)	SLOCK(D)	
	XLOCK(A)	XLOCK(C)	
	等待		
XLOCK(C)			
等待			XLOCK(D)
			等待



T1	T2	T3	T4
SLOCK(A)	SLOCK(D)		
	SLOCK(B)	SLOCK(D)	
	XLOCK(A)	XLOCK(C)	
	等待		
XLOCK(C)			
等待			XLOCK(D)
			等待
		XLOCK(B)	



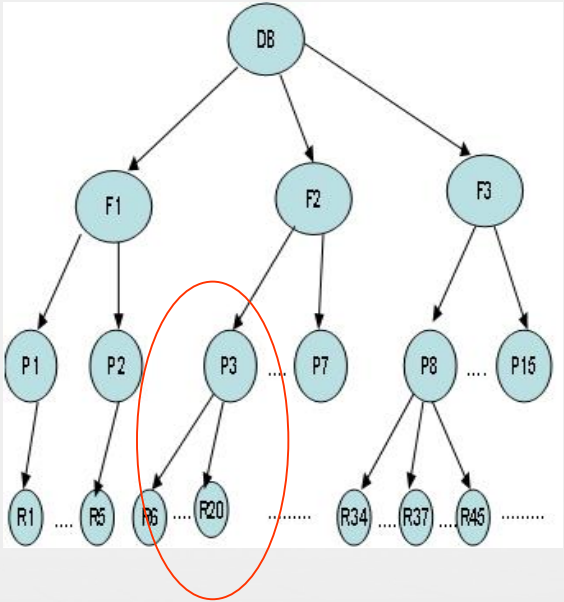
- 选择一个或多个事务执行撤销操作

释放事务拥有的封锁资源。

- 撤销事务的选择原则

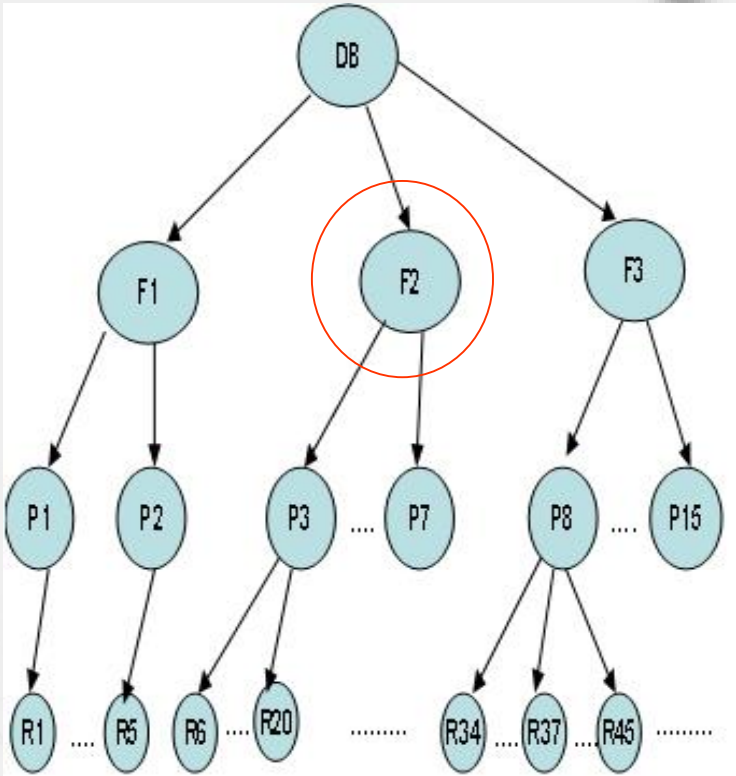
事务撤销所需付出的系统代价最小。

- 封锁对象的大小称为封锁粒度（granularity）



- 封锁对象：
  - 数据库的逻辑单位，如属性、元组、关系、索引、数据库等；
  - 数据库的物理单位，如页、块等；
- 封锁粒度对并发度和资源消耗的影响：
  - 若封锁粒度小，则系统并发度高，资源消耗多；
  - 若封锁粒度大，则系统并发度低，资源消耗小；
- 不同事务可能需要不同的封锁粒度，系统允许同时为不同的事务提供不同的封锁粒度选择。

- 意向锁（intention lock）
  - 如果对一个结点加意向锁，则意味着要对该结点的所有子孙结点显式加锁；
  - 在一个结点显式加锁前，该结点的所有祖先结点都应加上意向锁。



- 意向锁类型
  - 意向共享锁（IS锁）：如果一个结点加IS锁，那么将在该结点的子孙结点进行显式封锁，加S锁
  - 意向排它锁（IX锁）：如果一个结点加IX锁，那么表示将在该结点的子孙结点进行显式封锁，可以加排它锁和共享锁
  - 共享意向排它锁（SIX锁）：若一个结点加SIX锁，那么将对该结点的子结点显式地加共享锁，对更低层的结点显式地加排它锁

已分配的锁

申请锁 T1 \ T2	S	X	IS	IX	SIX
S	Y	N	Y	N	N
X	N	N	N	N	N
IS	Y	N	Y	Y	Y
IX	N	N	Y	Y	N
SIX	N	N	Y	N	N

Y 相容
N 不相容