
华中科技大学计算机科学与技术学院
SCHOOL OF COMPUTER SCIENCE & TECHNOLOGY, HUST

第十一章 并发控制

第十一章 并发控制

11.1 并发控制概述

多事务执行方式

(1)事务串行执行

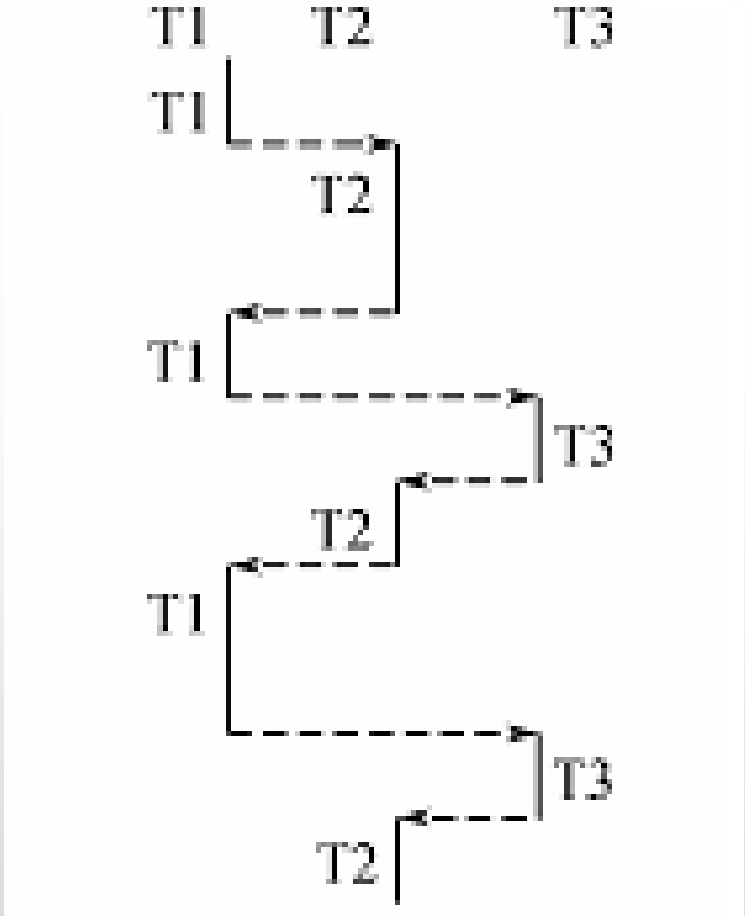
- 每个时刻只有一个事务运行，其他事务必须等到这个事务结束以后方能运行
- 不能充分利用系统资源，发挥数据库共享资源的特点

事务的串行执行方式

并发控制（续）

（2）交叉并发方式（interleaved concurrency）

- 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
- 是单处理机系统中的并发方式，能够减少处理机的空闲时间，提高系统的效率



并发控制（续）

(3)同时并发方式（simultaneous concurrency）

- 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
- 最理想的并发方式，但受制于硬件环境
- 更复杂的并发方式机制

事务并发执行带来的问题

- 可能会存取和存储不正确的数据，破坏事务的隔离性和数据库的一致性

并发操作带来的数据不一致性

- 丢失修改（lost update）
- 不可重复读（non-repeatable read）
- 读“脏”数据（dirty read）

1. 丢失修改

丢失修改是指事务1与事务2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。

三种数据不一致性-丢失更新

T_1	T_2
① 读A=16	
②	读A=16
③ A←A-1 写回 A=15	
④	A←A-1 写回A=15

(a) 丢失修改

2. 不可重复读

不可重复读是指事务1读取数据后，事务2执行更新操作，使事务1无法再现前一次读取结果。

三种数据不一致性(续) –不可重复读

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

(b) 不可重复读

不可重复读-幻象

- 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
- 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
- 这两种不可重复读有时也称为“幻影/幻象” 现象（Phantom Row）

3. 读“脏”数据

事务1修改某一数据，并将其写回磁盘

事务2读取同一数据后

事务1由于某种原因被撤消，这时事务1已修改过

的数据恢复原值

事务2读到的数据就与数据库中的数据不一致，

是不正确的数据，又称为“脏”数据。

三种数据不一致性(续)-读“脏”数据

T_1	T_2
① 读C=100 C←C*2 写回C	
②	读C=200
③ ROLLBACK C恢复为100	

(c) 读“脏”数据

三种数据不一致性(续)

- 关于丢失更新
  - 第一类丢失更新（回滚丢失）
  - 第二类丢失更新（覆盖丢失）
- 关于不可重复读
  - 幻想读的问题

三种数据不一致性(续)

- DBMS必须提供并发控制机制
- 并发控制机制是衡量一个DBMS性能的重要标志之一
- 并发控制机制的任务
  - 对并发操作进行正确调度
  - 保证事务的隔离性
  - 保证数据库的一致性
- 并发控制的基本方法是采用封锁

第十一章 并发控制

11.2 封锁

一、什么是封锁

二、基本封锁类型

三、基本锁的相容矩阵

一、什么是封锁

- 封锁就是事务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释放它的锁之前，其它的事务不能更新此数据对象。
- 封锁是实现并发控制的一个非常重要的技术

一、什么是封锁

1、是并发控制的一种技术。主要并发控制方法:

① 锁（Locking）——商用主要方法

② 乐观（Optimistic）

③ 时标（timestamping）

2、封锁规则

① 将要存取的数据须先申请加锁;

② 已被加锁的数据不能再加不相容锁;

③ 一旦退出使用应立即释放锁;

④ 未被加锁的数据不可对之解锁。

一、什么是封锁

3.申请时机

事务

- 无死锁;
- 锁开销少;
- 并发性低。

一个SQL语句

- 并发性高;
- 锁开销大;
- 死锁;
- 申请频繁。

一、什么是封锁

4. 申请方式

显式

应事务的要求直接加到数据对象上

隐式

该数据对象没有独立加锁，由于数据对象的多粒度层次结构中的上级结点加了锁，使该数据对象隐含的加了相同类型的锁。

LOCKS(锁)，LATCHES(闩锁)比较

	Locks	*Latches*
分离…	用户事务	线程
保护…	数据库内容	内存数据结构
范围…	整个事务	临界区
模式…	共享、互斥、更新、意向锁	读、写
死锁…	检测和解决	避免
方法…	等待、超时、终止	编码规则
保存…	锁管理器	受保护的数据结构

二、基本封锁类型

- DBMS通常提供了多种类型的封锁。一个事务对某个数据对象加锁后究竟拥有什么样的控制是由封锁的类型决定的。
- 封锁类似于操作系统中的锁，然而又有所不同
- 基本封锁类型
  - 排它锁（eXclusive lock，简记为X锁）
  - 共享锁（Share lock，简记为S锁）

排它锁

- 排它锁又称为写锁
- 若事务T对数据对象A加上X锁，则只允许T读取和修改A，其它任何事务都不能再对A加任何类型的锁，直到T释放A上的锁

共享锁

- 共享锁又称为读锁
- 若事务T对数据对象A加上S锁，则其它事务只能再对A加S锁，而不能加X锁，直到T释放A上的S锁

三、锁的相容矩阵

T_2 \ T_1	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

对应问题1：加锁解决丢失更新

时间	T_A	X 值	T_B	说明
t1	X Lock X R(X)=100	100		T_A 对 X 加锁成功后读 X
t2			X Lock X 等待	T_B 对 X 加 X 锁未成功则等待
t3	W(X)=X－1 COMMIT UnLock X	99	等待	T_A 修改，X 结果写回 DB 释放 X 锁
t4			X Lock X R(X)=99 W(X)=X－1 COMMIT UnLock X	T_B 获得 X 的 X 锁读 X 得 99（已更新后结果）将修改后 X（98）写回 DB

对应问题2：加锁解决不可重复读

时间	T_A	DB 中 A、B 值	T_B	说明
t1	S Lock A R(A)=50 S Lock B R(B)=100 C = A+B	A：50 B：100 A：50		T_A 对 A、B 加 S 锁 T_B 不能再对之加 X 锁
t2			X Lock B	T_B 对 B 加锁不成功 T_B 等待
t3	R(A)=50 R(B)=100 COMMIT Unlock A Unlock B	A：50 B：100		T_B 等待 T_B 重读 A、B 计算、结果相同
t4		A：50 B：200	X Lock B R(B)=100 W(B)：=B*2 写回 B=200	T_B 获得 B 的 X 锁

对应问题3：加锁解决读脏

时间	T_A	X 值	T_B	说明
t1	X Lock X R(X)=100 W(X)=X*2	100 200		T_A 先获得 X 锁
t2			S Lock X 等待	T_B 申请 S 锁 T_A 未释放，T_B 等待
t3	Rollback (W(X)=100) UnLock X	100		T_A 因故撤消 X 值恢复为 100
t4			S Lock X R(X)=100	T_B 获得 S 锁 T_B 读到值与 DB 中值一致 防止了读脏

关于隔离级别

- Read uncommitted（未提交读）
- Read committed（提交读）
- Repeatable read（可重复读）
- Serializable（可串行读）

隔离级别	脏读	不可重复读取	幻像
未提交读	是	是	是
提交读	否	是	是
可重复读	否	否	是
可串行读	否	否	否

第十一章 并发控制

11.3 活锁和死锁

- 封锁技术可以有效地解决并行操作的一致性问题，但也带来一些新的问题
  - 死锁
  - 活锁

11.3.1 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	.	.

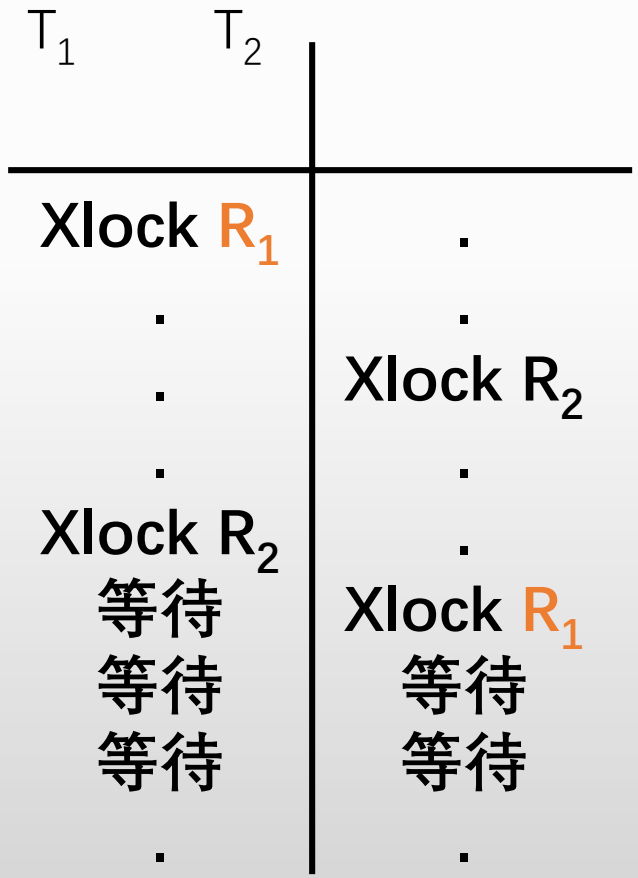
如何避免活锁

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。

11.3.2 死锁



解决死锁的方法

两类方法

1. 预防死锁
2. 死锁的诊断与解除

1. 死锁的预防

- 产生死锁的原因是两个或多个事务都已封锁了一些数据对象，然后又都请求对已为其他事务封锁的数据对象加锁，从而出现死等待（循环等待）。

- 预防死锁的发生就是要破坏产生死锁的条件

死锁的预防（续）

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
- 一次封锁法存在的问题：降低并发度
  - 扩大封锁范围
  - 将以后要用到的全部数据加锁，势必扩大了封锁的范围，从而降低了系统的并发度

一次封锁法（续）

- 难以事先精确确定封锁对象
  - 数据库中数据是不断变化的，原来不要求封锁的数据，在执行过程中可能会变成封锁对象，所以很难事先精确地确定每个事务所要封锁的数据对象
  - 解决方法：将事务在执行过程中可能要封锁的数据对象全部加锁，这就进一步降低了并发度。

（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。
- 顺序封锁法存在的问题
  - 维护成本高
  - 数据库系统中可封锁的数据对象极其众多，并且随数据的插入、删除等操作而不断地变化，要维护这样极多而且变化的资源的封锁顺序非常困难，成本很高

顺序封锁法（续）

- 难于实现
- 事务的封锁请求可以随着事务的执行而动态地决定，很难事先确定每一个事务要封锁哪些对象，因此也就很难按规定的顺序去施加封锁。

例：规定数据对象的封锁顺序为A,B,C,D,E。事务T3起初要求封锁数据对象B,C,E，但当它封锁了B,C后，才发现还需要封锁A，这样就破坏了封锁顺序。

2. 死锁的诊断与解除

- 允许死锁发生
- 解除死锁
  - 由DBMS的并发控制子系统定期检测系统中是否存在死锁
  - 一旦检测到死锁，就要设法解除

检测死锁：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
  - 时限若设置得太短，有可能误判死锁
  - 时限若设置得太长，死锁发生后不能及时发现

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*G*=(*T*，*U*)
  - *T*为结点的集合，每个结点表示正运行的事务
  - *U*为边的集合，每条边表示事务等待的情况
  - 若T_1等待T_2，则T_1，T_2之间划一条有向边，从T_1指向T_2

- 并发控制子系统周期性地（比如每隔1 min）检测事务等待图，如果发现图中存在回路，则表示系统中出现了死锁。

死锁的诊断与解除（续）

- 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的锁，使其它事务能继续运行下去。
  - 选择合适的事务撤销取决于多种因素….
    - 事务年龄 (最小的时间戳)
    - 事务进度 (执行最少/最多查询)
    - 所锁定的数据库对象数量
    - 需回滚事务的数量

11.3.2 封锁协议

- 不同的封锁协议，在不同的程度上为并发操作的提供了一定的数据一致性保证
- 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
- 常用的封锁协议：三级封锁协议

1级封锁协议

- 事务T在修改数据R之前必须先对其加X锁，直到事务结束才释放
  - 正常结束（COMMIT）
  - 非正常结束（ROLLBACK）
- 1级封锁协议可防止丢失修改
- 在1级封锁协议中，如果是读数据，不需要加锁的，所以它不能保证可重复读和不读“脏”数据。

1级封锁协议

T_1	T_2
① Xlock A 获得	
② 读A=16	
	Xlock A
③A←A-1	等待
写回A=15	等待
Commit	等待
Unlock A	等待
④	获得Xlock A
	读A=15
	A←A-1
⑤	写回A=14
	Commit
	Unlock A

没有丢失修改

1级封锁协议

T_1	T_2
①读A=50 读B=100 求和=150	
②	Xlock B 获得 读B=100 B←B*2 写回B=200 Commit Unlock B
③读A=50 读B=200 求和=250 (验算不对)	

不可重复读

1级封锁协议

T_1	T_2
① Xlock A 获得	
② 读A=16 A←A-1 写回A=15	
③	读A=15
④ Rollback Unlock A	

读“脏”数据

2级封锁协议

- 1级封锁协议+事务T在读取数据R前必须先加S锁，读完后即可释放S锁
- 2级封锁协议可以防止丢失修改和读“脏”数据。
- 在2级封锁协议中，由于读完数据后即可释放S锁，所以它不能保证可重复读。

三种数据不一致性(续)

T_1	T_2
Xlock C 读C=100 C←C*2 写回C ROLLBACK Unlock C	 Slock C 等待 等待 等待 获得Slock 读C=200 Unlock C commit

不读“脏”数据

2级封锁协议

T_1	T_2
① Slock A 获得 读A=50 Unlock A ② Slock B 获得 读B=100 Unlock B ③ 求和=150	 Xlock B 等待 等待 获得Xlock B 读B=100 B←B*2 写回B=200 Commit Unlock B

T_1 (续)	T_2
④Slock A 获得 读A=50 Unlock A Slock B 获得 读B=200 Unlock B 求和=250 (验算不对)	

不可重复读

3级封锁协议

- 1级封锁协议 + 事务T在读取数据R之前必须先对其加S锁，直到事务结束才释放

- 3级封锁协议可防止丢失修改、读脏数据和不可重复读。

3级封锁协议

T_1	T_2
① Slock A	
读A=50	
Slock B	
读B=100	
求和=150	
②	
	Xlock B
③ 读A=50	等待
读B=100	等待
求和=150	等待
Commit	等待
Unlock A	等待
Unlock B	等待
④	获得Xlock B
	读B=100
	B←B*2
⑤	写回B=200
	Commit
	Unlock B

可重复读

3级封锁协议

T_1	T_2
① Xlock C	
读C= 100	
C←C*2	
写回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
	读C=100
⑤	Commit C
	Unlock C

不读“脏”数据

封锁协议小结

- 三级协议的主要区别
  - 什么操作需要申请封锁
  - 何时释放锁（即持锁时间）

封锁协议小结(续)

	X锁		S锁		一致性保证		
	操作结束释放	事务结束释放	操作结束释放	事务结束释放	不丢失修改	不读"脏"数据	可重复读
1级封锁协议		√			√		
2级封锁协议		√	√		√	√	
3级封锁协议		√		√	√	√	√

第十一章 并发控制

11.4 并发调度的可串行性

一、什么是调度

二、什么样的并发操作调度是正确的

三、如何保证并发操作的调度是正确的

四、可恢复性的问题

一、什么是调度

调度是一个或多个事务的操作（sql语句）在系统中按时间排序的一个序列。一组事务的一个调度必须包含这一组事务的全部语句，并且必须保持语句在各个事务中出现的顺序。

举例：

T1(Begin)- $R_{T1}(x)$- T2(Begin)- $R_{T2}(y)$- $W_{T2}(y)$- T2(Commit)- T1(Commit)…

二、什么样的并发操作调度是正确的

- 计算机系统对并行事务中并行操作的调度是的随机的，而不同的调度可能会产生不同的结果。
- 由于“一致性”，单个事务的执行将使数据库状态从一个一致性状态转变到另一个一致性状态。
- 现实中一个事务通常与其他事务一起并发执行，将所有事务串行起来的调度策略一定是正确的调度策略。
  - 如果一个事务运行过程中没有其他事务在同时运行，也就是说它没有受到其他事务的干扰（隔离性未被打破），那么就可以认为该事务的运行结果是正常的或者预想的。

什么样的并发操作调度是正确的

- 单个事务
  - 每个事务都能保证DB的正确性
- 多个事务
  - 多个事务以任意串行方式执行都能保证DB的正确性

什么样的并发操作调度是正确的（续）

- 以不同的顺序串行执行事务也有可能会产生不同的结果，但由于不会将数据库置于不一致状态，所以都可以认为是正确的。

- 几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它们时的结果相同。这种并行调度策略称为可串行化（Serializable）的调度。

什么样的并发操作调度是正确的（续）

- 可串行性是并发事务正确性的唯一准则

例：现在有两个事务，分别包含下列操作：

事务1：读B；A=B+1；写回A；

事务2：读A；B=A+1；写回B；

假设A的初值为2，B的初值为2。

什么样的并发操作调度是正确的（续）

- 对这两个事务的不同调度策略
  - 串行执行
    - 串行调度策略1：T1T2
    - 串行调度策略2：T2T1
  - 交错执行
    - 不可串行化的调度
    - 可串行化的调度

(a) 串行调度策略，正确的调度

T_1	T_2
Slock B	
Y=B=2	
Unlock B	
Xlock A	
A=Y+1	
写回A(=3)	
Unlock A	
	SlockA
	X=A=3
	Unlock A
	Xlock B
	B=X+1
	写回B(=4)
	Unlock B

(b) 串行调度策略，正确的调度

T_1	T_2
	SlockA
	X=A=2
	Unlock A
	Xlock B
	B=X+1
	写回B(=3)
	Unlock B
Slock B	
Y=B=3	
Unlock B	
Xlock A	
A=Y+1	
写回A(=4)	
Unlock A	

(c) 不可串行化的调度

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

- 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) 可串行化的调度

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

- 由于其执行结果与串行调度（a）的执行结果相同，所以是正确的调度。

三、如何保证并发操作的调度是正确的

- 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。
- 关于可串行化的等价
  - 冲突可串行化
  - 视图可串行化

冲突可串行化

冲突:

同一事务的两个动作;

不同事务涉及同一数据库元素，且有一个是写操作。

不是冲突的情况（可交换顺序）

$(R_i(x), R_j(y))$； $(R_i(x), W_j(y))$； $(W_i(x), R_j(y))$； $(W_i(x), W_j(y))$

冲突可串行化

- 冲突可串行化判定
  - 优先图(precedence graph)

    一个调度S的优先图是这样构造的：它是一个有向图G =（V，E），V是顶点集，E是边集。顶点集由所有参与调度的事务组成，边集由满足下述条件之一的边$T_i \rightarrow T_j$组成：

    ①在T_j执行read(Q)之前，T_i执行write(Q)

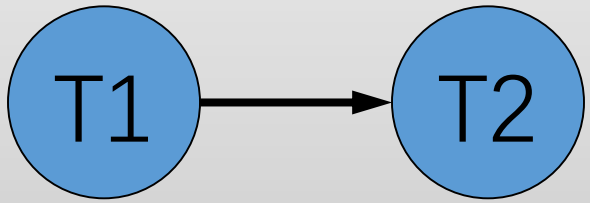
    ②在T_j执行write(Q)之前，T_i执行read(Q)

    ③在T_j执行write(Q)之前，T_i执行write(Q)

冲突可串行化（举例）

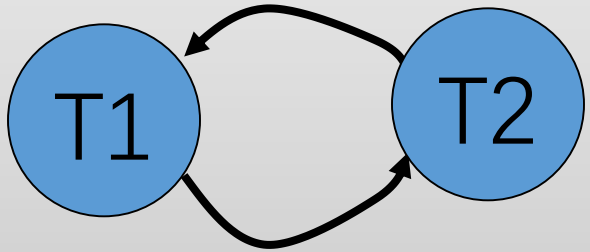
并行调度3

T1	T2
read(A);	
write(A);	
	read(A);
	write(A);
read(B);	
write(B);	
	read(B);
	write(B);



并行调度4

T1	T2
read(A);	
	read(A);
	write(A);
	read(B);
write(A);	
read(B);	
write(B);	
	write(B);

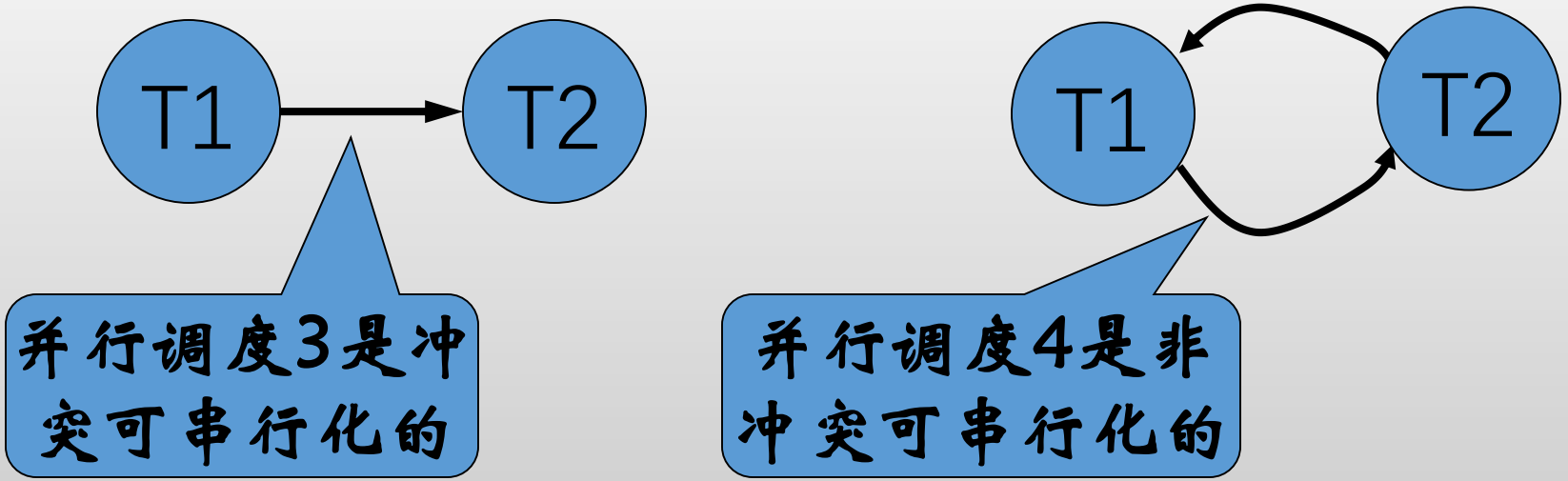


冲突可串行化

如果优先图中存在边$T_i \rightarrow T_j$，则在任何等价于S的串行调度S'中，T_i都必须出现在T_j之前

➢冲突可串行化判定准则

如果调度S的优先图中有环，则调度S是非冲突可串行化的。如果图中无环，则调度S是冲突可串行化的



冲突可串行化

冲突可串行性对可串行化来说不是必要的，而只是充分条件

考虑：

S1:$w_1(Y)$; $w_1(X)$; $w_2(Y)$; $w_2(X)$; $w_3(X)$;

S2:$w_1(Y)$; $w_2(Y)$; $w_2(X)$; $w_1(X)$; $w_3(X)$;

优先图测试发挥作用的原因

不存在环路，则该调度是冲突可串行化的（归纳法）

基础：如果n=1，即调度中只有一个事务，则该调度已经是串行的

归纳：设调度S由n个事务T_1，T_2，…，T_n组成

假设S无环，则至少有一个节点没有到达该节点的弧，设T_i是这样的一个节点，则S中不可能有位于T_i之前的某事务T_j的某冲突动作，因此我们可以交换T_i的所有动作，保持其顺序，将这些动作移动到S的前部，该调度具有如下形式：

（T_i的动作）（其他n-1个事务的动作）

考虑到后半部分依然无环，归纳假设同样适用。。。

四、可恢复性的问题

- 事务故障不可避免，若T_i失败，那么依赖T_i的事务T_j呢？（T_j读取了T_i写的数据）
- 可恢复调度应满足：对于每一对事务T_i和T_j，如果T_j读取了由T_i所写的数据项，则T_i先于T_j提交。

一个不可恢复的调度

T1	T2
read(A)	
write(A)	
	read(A)
	commit
read(B)	

可恢复性的问题

- 级联回滚：因一个事务故障导致一系列事务回滚的现象

T1	T2	T3
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
Abort T1		

可恢复性的问题

- 无级联调度应满足
  - 对于每对事务 T_i和 T_j，如果 T_j读取了由 T_i所写的数据项，则 T_i必须在 T_j这一读取前提交
- 若某调度是无级联调度，则该调度一定是可恢复调度。

如何保证并发操作的调度是正确的（续）

- 保证并发操作调度正确性的方法
  - 封锁方法：两段锁协议（Two-Phase Locking，简称2PL）
  - 时标方法
  - 乐观方法

第十一章 并发控制

11.5 两段锁协议

- 两段锁协议的内容

  1. 在对任何数据进行读、写操作之前，事务首先要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。

两段锁协议（续）

- “两段”锁的含义
  - 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。

两段锁协议（续）

例:

事务1的封锁序列:

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

事务2的封锁序列:

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

事务1遵守两段锁协议，而事务2不遵守两段协议。

两段锁协议（续）

- 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。所有遵守两段锁协议的事务，其并行执行的结果一定是正确的
- 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。
- 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（续）

遵守两阶段锁协议则调度可串行化：

与2PL事务的调度S冲突等价的串行调度是事务顺序与其第一个解锁顺序相同的串行调度。

证明：（归纳法）

基础：如果n = 1，则S已经是一个串行调度；

归纳：假设S：T1,T2,…,Tn，并设Ti是在S中第一个有解锁动作（如Ui(x)），则可以断言：将Ti的所有动作不经过任何冲突动作而向前移动到调度的开始是可能的。

1）考虑Ti的某个动作如Wi(y)，假定此动作前有一个冲突动作： Wj(y)，则S中，必出现如下的动作序列：

…;Wj(y);…;Uj(y);…; Li(y);…;Wi(y);…

2）既然Ti是第一个解锁的，则Ui(y)必在Uj(y)前，即会出现：

…;Wj(y);…; Ui(x);…; Uj(y);…; Li(y);…;Wi(y);…

两段锁协议（续）

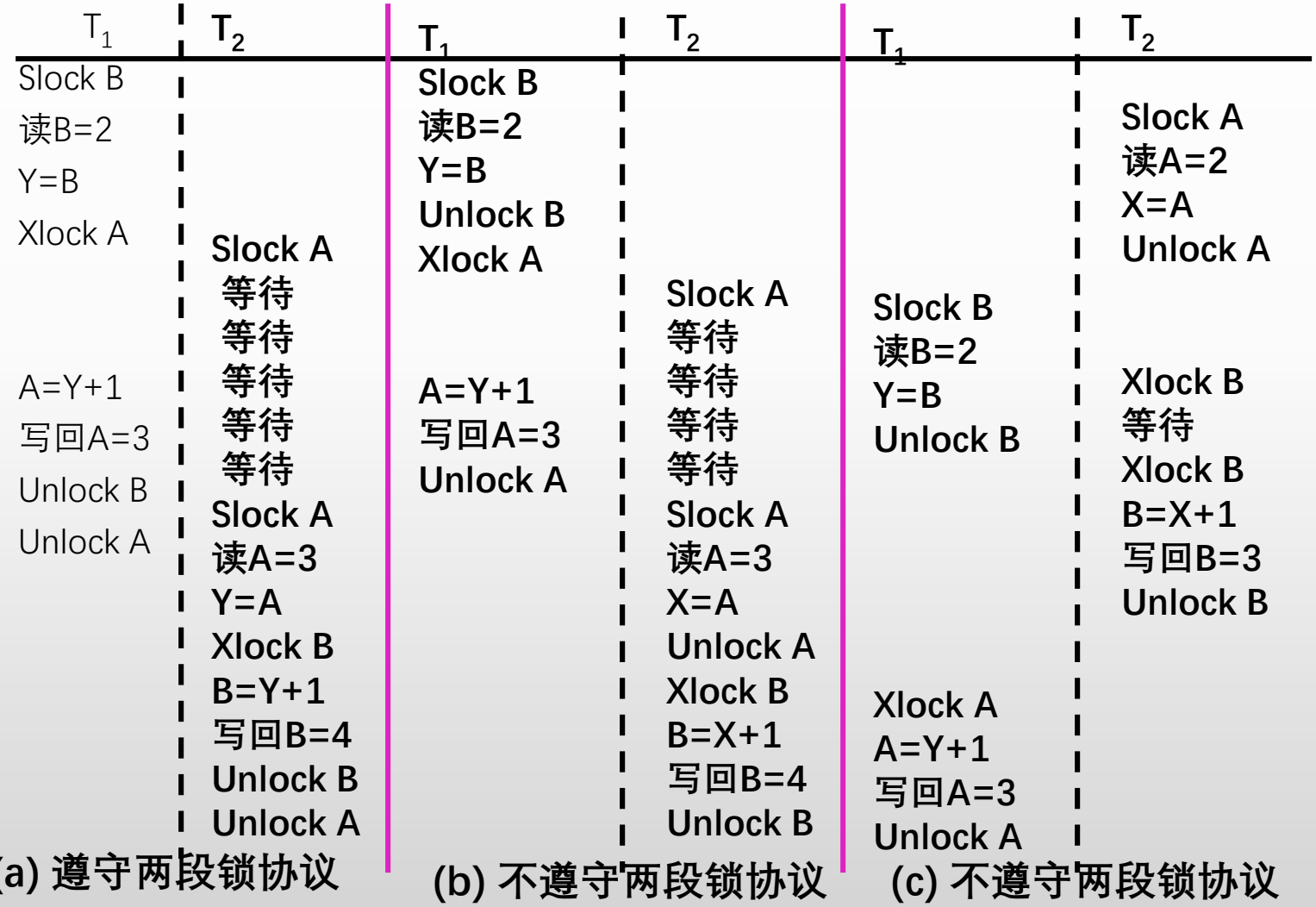
3）显然对于Ti这样的动作序列违反了两阶段锁协议，所以这样的动作序列不存在，也就是说Wi(y)前不存在来自其他事务的冲突动作，同样的证明也适用于任意一对由来自Ti的一个动作和来自Tj的一个动作构成的可能的冲突。

结论：由于Ti所有动作之前不可能存在冲突动作，则Ti的所有动作可以非冲突的进行交换，从而将其全部移动到S的开始，于是S可记为：

（Ti的动作）（其他n－1个事务的动作）

由n－1个事务构成的后半部分仍然是一致的2PL事务的一个合法调度，因此归纳假设在其上也适用，亦可转换为冲突等价串行调度。

两段锁协议（续）



(a) 遵守两段锁协议　(b) 不遵守两段锁协议　(c) 不遵守两段锁协议

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，否则就不能继续执行，因此一次封锁法遵守两段锁协议
  - 但是两段锁协议并不要求事务必须一次将所有要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）

T_1	T_2
Slock B	
读B=2	
	Slock A
	读A=2
Xlock A	
等待	Xlock A
等待	等待

遵守两段锁协议的事务发生死锁

两段锁协议（续）

- 两段锁协议与三级封锁协议
  - 两类不同目的的协议
    - 两段锁协议
      - 保证并发调度的正确性
    - 三级封锁协议
      - 在不同程度上保证数据一致性

第十一章 并发控制

11.6.1 封锁粒度

一、什么是封锁粒度

二、选择封锁粒度的原则

一、什么是封锁粒度

- X锁和S锁都是加在某一个数据对象上的
- 封锁的对象:逻辑单元，物理单元
- 例：在关系数据库中，封锁对象：
  - 逻辑单元: 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
  - 物理单元：页（数据页或索引页）、物理记录等

什么是封锁粒度（续）

- 封锁对象可以很大也可以很小

  例：　对整个数据库加锁

  　　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，举例：

封锁粒度：数据页

事务T1需要修改元组L1，则T1要对包含L1的整个数据页A加锁

事务T2需要修改数据页A中元组L2，必须等待，直到T1释放A

系统的并发度不高

封锁粒度：元组

事务T1需要修改元组L1，则T1要对元组L1加锁

事务T2需要修改元组L2，则T2要对元组L2加锁

互不影响，提高了系统的并发度

缺点：若事务T3要读取整个表， T3必须对该表中每个元组加锁，增加了系统开销

什么是封锁粒度（续）

多粒度封锁(multiple granularity locking)

- 在一个系统中同时支持多种封锁粒度供不同的事务选择

二、选择封锁粒度的原则

- 封锁的粒度越　　大，小，
- 系统被封锁的对象　　少，多，
- 并发度　　小，高，
- 系统开销　　小，大，
- 选择封锁粒度：

  对系统开销与并发度进行权衡

选择封锁粒度的原则（续）

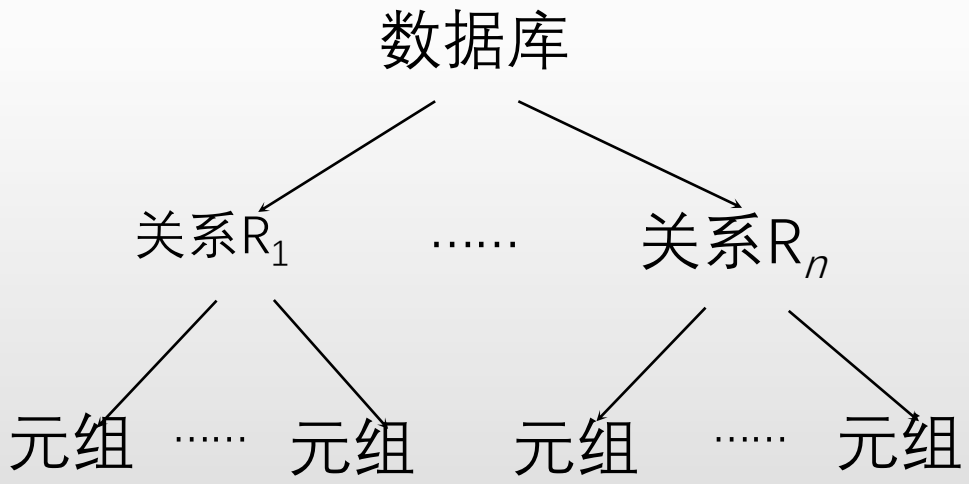
- 需要处理多个关系的大量元组的用户事务:
  - 以数据库为封锁单位;
- 需要处理大量元组的用户事务:
  - 以关系为封锁单元;
- 只处理少量元组的用户事务:
  - 以元组为封锁单位

多粒度封锁

- 多粒度树
  - 以树形结构来表示多级封锁粒度
  - 根结点是整个数据库，表示最大的数据粒度
  - 叶结点表示最小的数据粒度

多粒度封锁（续）

例：三级粒度树。根结点为数据库，数据库的子结点为关系，关系的子结点为元组。



三级粒度树

多粒度封锁协议

- 允许多粒度树中的每个结点被独立地加锁
- 对一个结点加锁意味着这个结点的所有后裔结点也被加以同样类型的锁
- 在多粒度封锁中一个数据对象可能以两种方式封锁：
  - 显式封锁和隐式封锁

显式封锁和隐式封锁

- 显式封锁: 直接加到数据对象上的封锁
- 隐式封锁: 该数据对象没有独立加锁，是由于其上级结点加锁而使该数据对象加上了锁
- 显式封锁和隐式封锁的效果是一样的

显式封锁和隐式封锁（续）

- 对某个数据对象加锁，系统要检查
  - 该数据对象
    - 有无显式封锁与之冲突
  - 所有上级结点
    - 检查本事务的显式封锁是否与该数据对象上的隐式封锁冲突：（由上级结点已加的封锁造成的）
  - 所有下级结点
    - 看上面的显式封锁是否与本事务的隐式封锁（将加到下级结点的封锁）冲突

显式封锁和隐式封锁（续）

- 例如事务T要对关系*R*1加X锁
  - 系统必须搜索其上级结点数据库、关系*R*1
  - 还要搜索*R*1的下级结点，即*R*1中的每一个元组
  - 如果其中某一个数据对象已经加了不相容锁，则T必须等待

11.6.2 意向锁

- 引进意向锁（intention lock）目的
  - 提高对某个数据对象加锁时系统的检查效率

意向锁（举例）

- 假设有一个张表Student，有100万条数据，其中有三行：
- Name Age

  张三 -11

  李四 -13

  王五 14
- 程序员A发现数据有错误，有一些学生的年龄有负数，他连接数据库，打算把所有的负数变成正数，并且希望在修改的时候，别人不可以读取数据。此时他对两行数据加了X排他锁。
- 程序员B发现有几个的姓名写错了，这时他打算给整张表加上S锁，也就是，别人可以读取数据，但是不能修改。这时数据库需要判断，这张表是否可以加S锁？如何判断呢：要看这张表中的100万行数据中有没有X锁，如果被加锁的这两行数据刚好在最后，那么要判断100万次才能得出结论：有一行加了X锁，该表不能加S锁，请等待该锁释放！显然这速度慢得多。假如有了意向锁，在A连接的时候，给行加上X锁的时候，对该表加上IX锁。B连接申请表的S锁之前，先看到该表有IX锁，就马上知道需要等待，而不需要去判断每一行的锁了。

意向锁(续)

- 如果对一个结点加意向锁，则说明该结点的下层结点正在被加锁
- 对任一结点加基本锁，必须先对它的上层结点加意向锁
- 例如，对任一元组加锁时，必须先对它所在的数据库和关系加意向锁

意向锁的类型

- 由两种基本的锁类型（S锁、X 锁），可以自然地派生出两种意向锁:
- 意向共享锁（Intent Share Lock，简称 IS 锁）：如果要对一个数据库对象加S锁，首先 要对其上级结点加IS 锁，表示它的后裔结点拟（意向）加 S锁;
- 意向排它锁（Intent Exclusive Lock，简称 IX 锁）：如果要对一个数据库对象加 X 锁，首先要对其上级结点加 IX锁，表示它的后裔结点拟（意向）加X 锁。
- 另外，基本的锁类型（S、X）与意向锁类型（IS、IX）之间还可以组合出新的锁类型，理论上可以组合出4种，即：S+IS，S+IX，X+IS，X+IX，但稍加分析不难看出，实际上只有 S+IX 有新的意义，其它三种组合都没有使锁的强度得到提高（即：S+IS=S，X+IS=X，X+IX=X，这里的“=”指锁的强度相同）。所谓锁的强度是指对其它锁的排斥程度

意向锁的类型

这样我们又可以引入一种新的锁的类型：

- 共享意向排它锁（Shared Intent Exclusive Lock,简称 SIX 锁）：如果对一个数据库对象加 SIX 锁，表示对它加 S 锁，再加 IX 锁，即 SIX=S+IX。例如：事务对某个表加 SIX 锁，则表示该事务要读整个表（所以要对该表加S 锁），同时会更新个别行（所以要对该表加 IX锁）。
- 这样数据库对象上所加的锁类型就可能有5 种：即S、X、IS、IX、SIX。

意向锁的相容矩阵

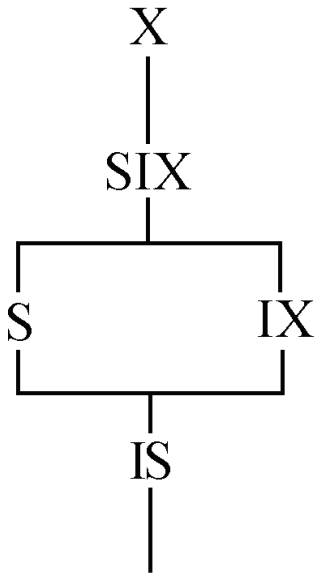
T_1 \ T_2	S	X	IS	IX	SIX	–
S	Y	N	Y	N	N	Y
X	N	N	N	N	N	Y
IS	Y	N	Y	Y	Y	Y
IX	N	N	Y	Y	N	Y
SIX	N	N	Y	N	N	Y
–	Y	Y	Y	Y	Y	Y

Y=Yes，表示相容的请求　　N=No，表示不相容的请求

(a) 数据锁的相容矩阵

意向锁（续）

- 锁的强度
  - 锁的强度是指它对其他锁的排斥程度
  - 一个事务在申请封锁时以强锁代替弱锁是安全的，反之则不然



(b) 锁的强度的偏序关系

意向锁（续）

- 具有意向锁的多粒度封锁方法
  - 申请封锁时应该按自上而下的次序进行
  - 释放封锁时则应该按自下而上的次序进行
- 例如：事务T1要对关系*R*1加S锁
  - 要首先对数据库加IS锁
  - 检查数据库和*R*1是否已加了不相容的锁(X或IX)
  - 不再需要搜索和检查*R*1中的元组是否加了不相容的锁(X锁)

意向锁（续）

- 具有意向锁的多粒度封锁方法
  - 提高了系统的并发度
  - 减少了加锁和解锁的开销
  - 在实际的数据库管理系统产品中得到广泛应用

第十一章 并发控制

10.7 小结

- 数据共享与数据一致性是一对矛盾
- 数据库的价值在很大程度上取决于它所能提供的数据共享度
- 数据共享在很大程度上取决于系统允许对数据并发操作的程度
- 系统并发程度又取决于数据库中的并发控制机制
- 另一方面，数据的一致性也取决于并发控制的程度。施加的并发控制愈多，数据的一致性往往愈好

小结（续）

- 数据库的并发控制以事务为单位
- 数据库的并发控制通常使用封锁机制
  - 两类最常用的封锁
- 不同级别的封锁协议提供不同的数据一致性保证，提供不同的数据共享度。

小结（续）

- 并发控制机制调度并发事务操作是否正确的唯一判别准则是

  可串行化准则

  - 并发操作的正确性则通常由两段锁协议来保证
  - 两段锁协议是可串行化调度的充分条件，但不是必要条件

小结（续）

- 对数据对象施加封锁，带来问题
- 活锁：先来先服务
- 死锁：
  - 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

小结（续）

- 不同的数据库管理系统提供的封锁类型、封锁协议、达到的系统一致性级别不尽相同。但是其依据的基本原理和技术是相同的。