Concurrency Control

金培权（jpq@ustc.edu.cn）

主要内容

- 并发操作与并发问题
- 并发事务调度与可串性 (Scheduling and Serializability)
- 锁与可串性实现 (Locks)
- 事务的隔离级别
- 死锁
- 乐观并发控制

Where are we ?

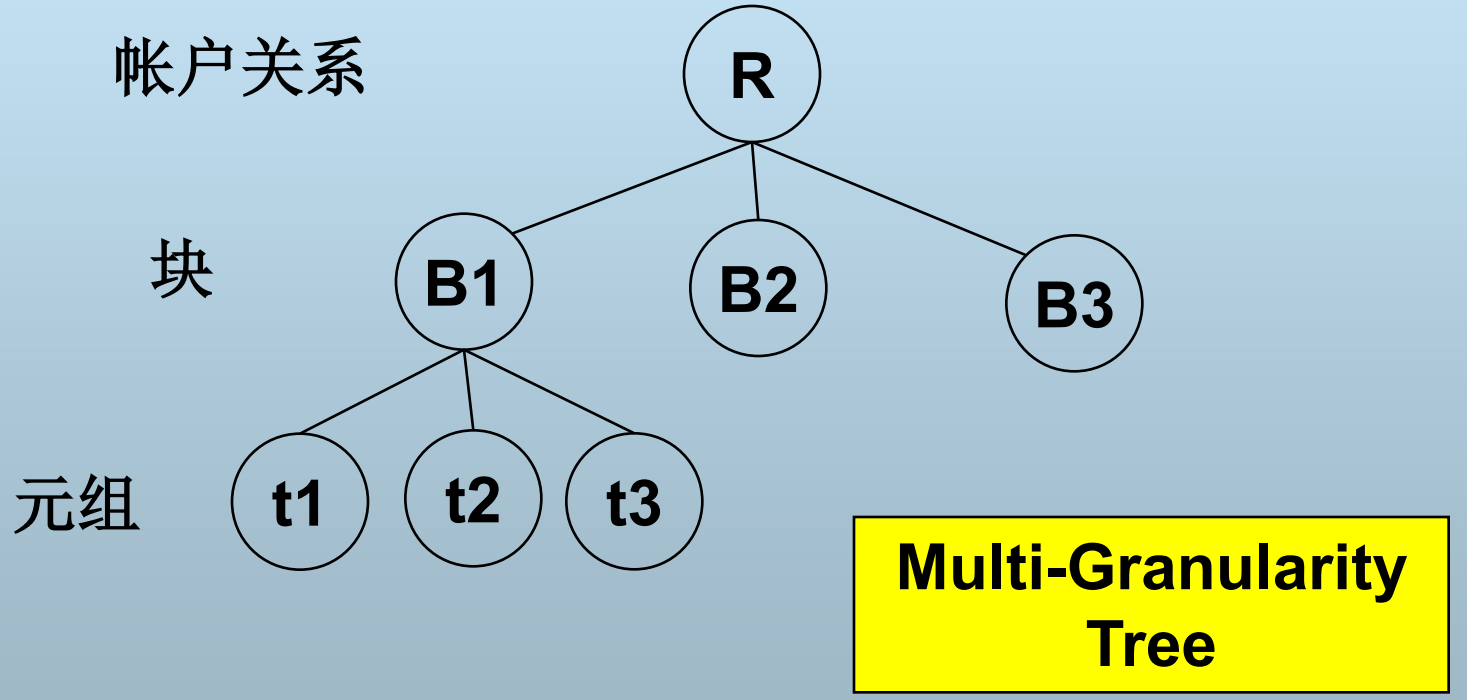
- 并发操作与并发问题
- 并发调度与可串性
- 锁与可串性实现
  - 2PL
    - S Lock
    - X Lock
    - U Lock
  - Multi-granularity Lock 多粒度锁
  - Intension Lock 意向锁

8、Multi-Granularity Lock

- **Lock Granularity**
  - 指加锁的数据对象的大小
    - 可以是整个关系、块、元组、整个索引、索引项
- **锁粒度越细，并发度越高；锁粒度越粗，并发度越低**

8、Multi-Granularity Lock

- 多粒度锁：同时支持多种不同的锁粒度



8、Multi-Granularity Lock

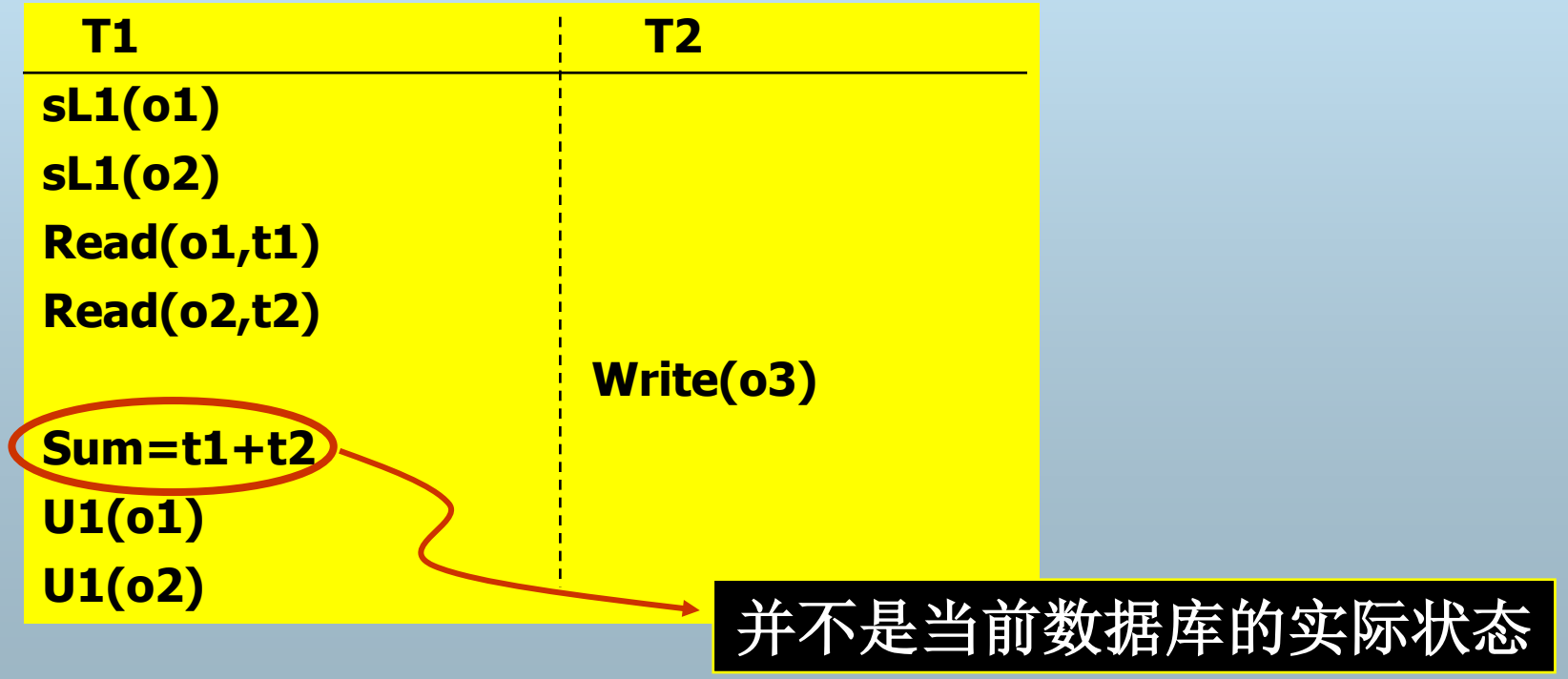
- 多粒度锁协议
  - 允许多粒度树中的每个结点被独立地加S锁或X锁，对某个结点加锁意味着其下层结点也被加了同类型的锁

8、Multi-Granularity Lock

- **Why we need MGL?**

8、Multi-Granularity Lock

T1: 求当前数据库中所有帐户的余额之和
T2: 增加一个新帐户(余额为1000)
Use tuple locks，suppose total two tuples in R



8、Multi-Granularity Lock

- 原因
  - Lock只能针对已存在的元组，对于开始时不存在后来被插入的元组无法Lock
  - o3：Phantom tuple 幻像元组
    - 存在，却看不到物理实体

Solution

- T2插入o3的操作看成是整个关系的写操作，对整个关系加锁
  - Need MGL！

8、Multi-Granularity Lock

Solution: Using MGL

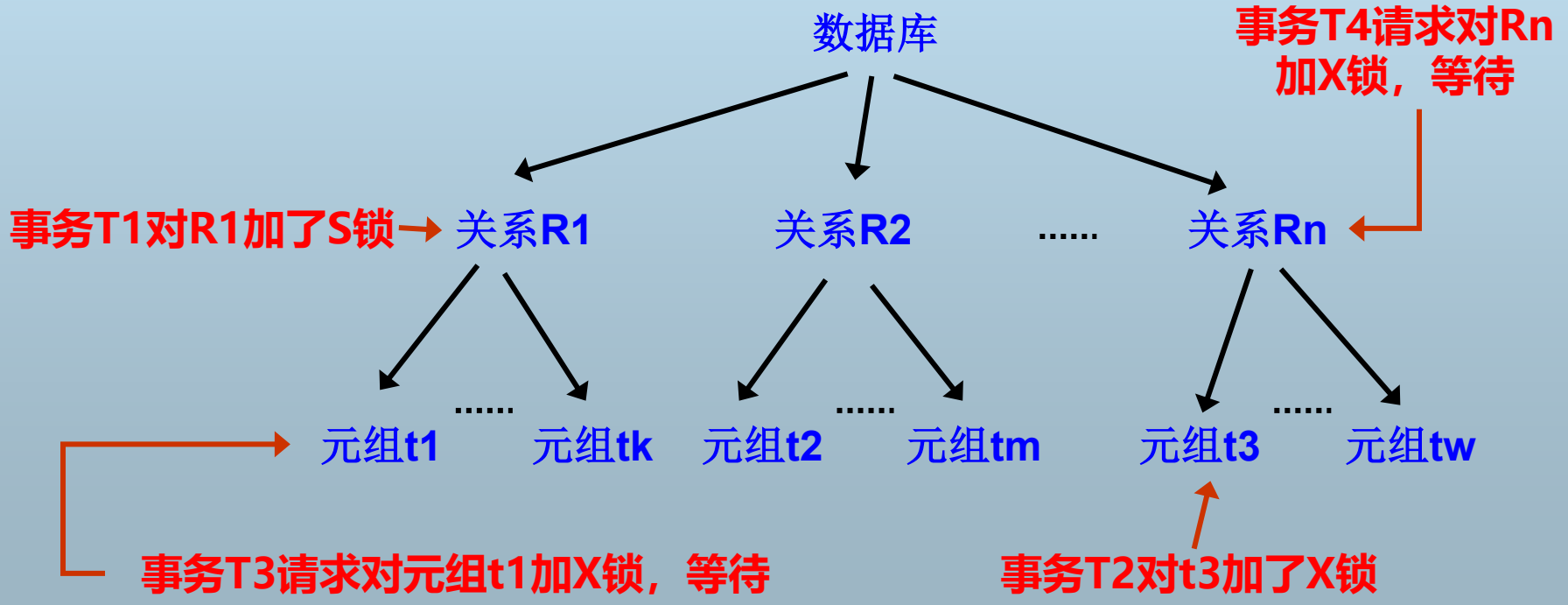
T1	T2
sL1(o1)	
sL1(o2)	
Read(o1,t1)	
Read(o2,t2)	
	xL2(R)
Sum=t1+t2	wait
U1(o1)	wait
U1(o2)	wait
	write(o3)

8、Multi-Granularity Lock

- 多粒度锁上的两种不同加锁方式
  - 显式加锁：应事务的请求直接加到数据对象上的锁
  - 隐式加锁：本身没有被显式加锁，但因为其上层结点加了锁而使数据对象被加锁
  - 给一个结点显式加锁时必须考虑
    - 该结点是否已有不相容锁存在
    - 上层结点是否已有不相容的的锁（上层结点导致的隐式锁冲突）
    - 所有下层结点中是否存在不相容的显式锁

8、Multi-Granularity Lock

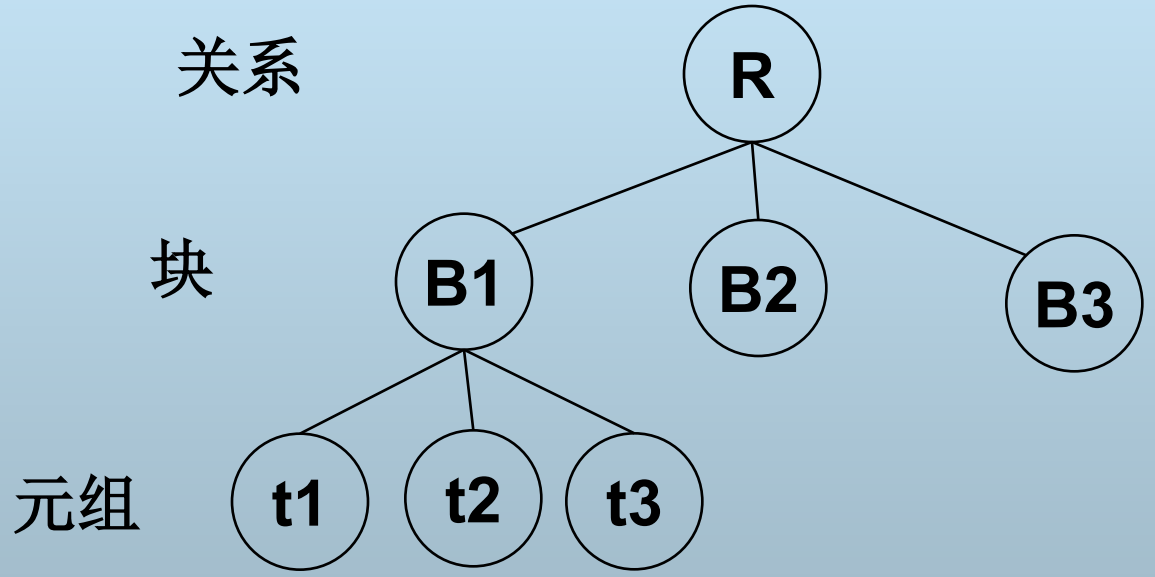
- 事务T1对关系R1显式加了S锁，意味着R1的所有元组也被隐式加了S锁。其它事务可以在R1的元组上加S锁，但不能加X锁
- 事务T2对元组t3加了X锁，其它事务不能请求对其上层结点Rn的S锁或X锁。



8、Multi-Granularity Lock

- 在对一个结点P请求锁时，必须判断该结点上是否存在不相容的锁
  - 有可能是P上的显式锁
  - 也有可能是P的上层结点导致的隐式锁
  - 还有可能是P的下层结点中已存在的某个显式锁
- 理论上要搜索上面全部的可能情况，才能确定P上的锁请求能否成功
  - 显然是低效的
  - 引入意向锁 (Intension Lock) 解决这一问题

9、Intension Lock



9、Intension Lock

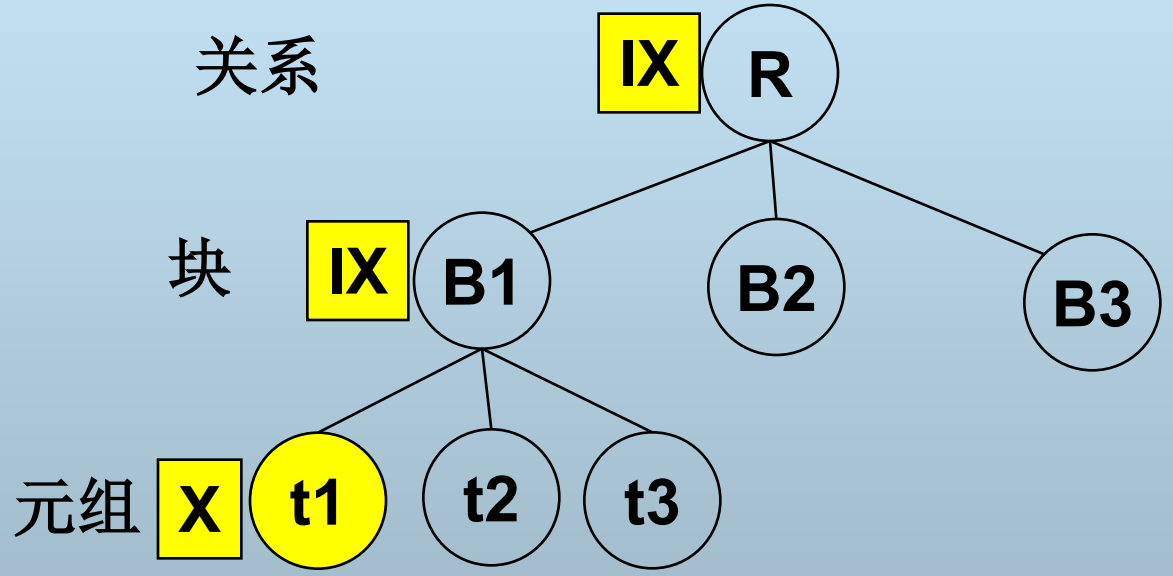
- IS锁（Intent Share Lock，意向共享锁，意向读锁）
- IX锁（Intent Exlusive Lock，意向排它锁，意向写锁）

9、Intension Lock

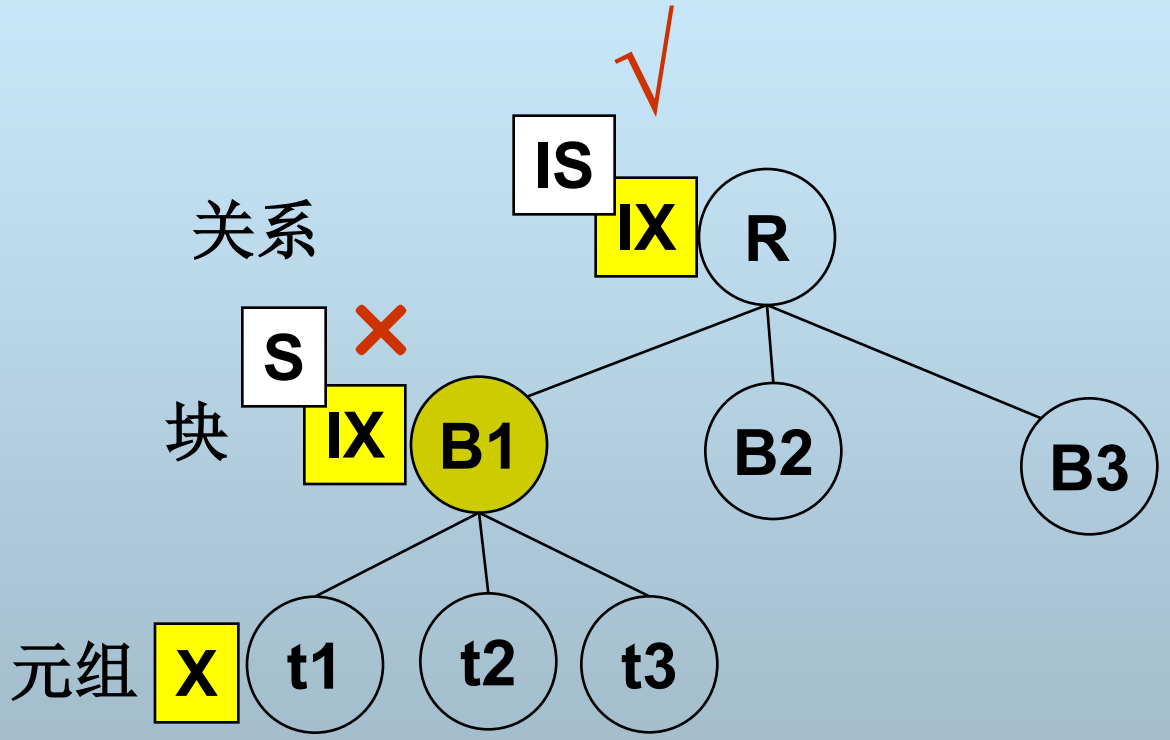
- 如果对某个结点加IS(IX)锁，则说明事务要对该结点的某个下层结点加S (X)锁；
- 对任一结点P加S(X)锁，必须先对从根结点到P的路径上的所有结点加IS(IX)锁

9、Intension Lock

Want to exclusively lock t1



9、Intension Lock



9、Intension Lock

- **Compatibility Matrix**

	IS	IX	S	X
IS	✓	✓	✓	×
IX	✓	✓	×	×
S	✓	×	✓	×
X	×	×	×	×

四、事务的隔离级别

- 并发控制机制可以解决并发问题。这使所有事务得以在彼此完全隔离的环境中运行
- 然而许多事务并不总是要求完全的隔离。如果允许降低隔离级别，则可以提高并发性

四、事务的隔离级别

- SQL92标准定义了四种事务隔离级别
  - Note 1：隔离级别是针对连接（会话）而设置的，不是针对一个事务
  - Note 2：不同隔离级别影响读操作。

Oracle
MS SQL Server
默认

MySQL默认

隔离级别 (Isolation Level)	脏读 (Dirty Read)	不可重复读 (Nonrepeatable Read)	幻读 (Phantom Read)
未提交读	可能	可能	可能
提交读	不可能	可能	可能
可重复读	不可能	不可能	可能
可串化	不可能	不可能	不可能

Oracle只支持提交读和可串行读，MySQL和MS SQL Server都支持四种隔离级别

四、事务的隔离级别

- 未提交读（脏读） Read Uncommitted
  - 允许读取当前数据页上的任何数据，不管数据是否已提交
  - 事务不必等待任何锁，也不对读取的数据加锁

隔离级别 (Isolation Level)	脏读 (Dirty Read)	不可重复读 (Nonrepeatable Read)	幻读 (Phantom Read)
未提交读	可能	可能	可能
提交读	不可能	可能	可能
可重复读	不可能	不可能	可能
可串化	不可能	不可能	不可能

四、事务的隔离级别

时间	连接1	连接2
1		Set transaction isolation level READ UNCOMMITTED
2		Begin tran
3	Begin tran	Select * from S Where SNAME='王红'
4	Update s set AGE=20 where SNAME='王红'	
5		Select * from S Where SNAME='王红'
6	Rollback tran	
7		Commit tran

脏读

四、事务的隔离级别

- 提交读 Read Committed
  - 保证事务不会读取到其他未提交事务所修改的数据（可防止脏读）
  - 事务必须在所访问数据上加S锁，数据一旦读出，就马上释放持有的S锁

隔离级别 (Isolation Level)	脏读 (Dirty Read)	不可重复读 (Nonrepeatable Read)	幻读 (Phantom Read)
未提交读	可能	可能	可能
提交读	不可能	可能	可能
可重复读	不可能	不可能	可能
可串化	不可能	不可能	不可能

四、事务的隔离级别

时间	连接1	连接2
1		Set transaction isolation level READ COMMITTED
2		Begin tran
3	Begin tran	Select * from S Where SNAME='王红'
4	Update s set AGE=20 where SNAME='王红'	
5		Select * from S Where SNAME='王红'
6	Commit tran	
7		Select * from S Where SNAME='王红'
8		Commit tran

四、事务的隔离级别

- 可重复读 **Repeatable Read**
  - 保证事务在事务内部如果重复访问同一数据（记录集），数据不会发生改变。即，事务在访问数据时，其他事务不能修改正在访问的那部分数据
  - 可重复读可以防止脏读和不可重复读取，但不能防止幻像
  - 事务必须在所访问数据上加S锁，防止其他事务修改数据，而且S锁必须保持到事务结束

隔离级别 (Isolation Level)	脏读 (Dirty Read)	不可重复读 (Nonrepeatable Read)	幻读 (Phantom Read)
未提交读	可能	可能	可能
提交读	不可能	可能	可能
可重复读	不可能	不可能	可能
可串化	不可能	不可能	不可能

四、事务的隔离级别

时间	连接1	连接2
1	此两步执行	Set transaction isolation level REPEATABLE READ
2		Begin tran
3	Begin tran	Select * from S Where SNAME='王红'
4	Insert into s values(s8,'王红',23)	
5	Update s set age=22 where sname='张三'	
6	Update s set age=22 where sname='王红'	
7	Commit tran	
8	此步须等待	Select * from s Where SNAME='王红' 出现幻象
9		Commit tran

四、事务的隔离级别

- 可串行读 Serializable
  - 保证事务调度是可串化的
  - 事务在访问数据时，其他事务不能修改数据，也不能插入新元组
  - 事务必须在所访问数据上加S锁，防止其他事务修改数据，而且S锁必须保持到事务结束
  - 事务还必须锁住访问的整个表
  - 不会出现丢失更新

隔离级别 (Isolation Level)	脏读 (Dirty Read)	不可重复读 (Nonrepeatable Read)	幻读 (Phantom Read)
未提交读	可能	可能	可能
提交读	不可能	可能	可能
可重复读	不可能	不可能	可能
可串化	不可能	不可能	不可能

四、事务的隔离级别

时间	连接1	连接2
1		Set transaction isolation level SERIALIZABLE
2		Begin tran
3	Begin tran	Select SNAME from S Where SNAME='王红'
4	Insert into s values(s08,'王红',23)	
5		Select SNAME from S Where SNAME='王红'
6	此步须等待	Commit tran

四、事务的隔离级别

- 不同隔离级别下DBMS加锁的动作有很大的差别

五、死锁(deadlock)

- Two transactions each acquire a lock on a DB element the other needs
- Two transactions try to upgrade locks on elements the other is reading

1、锁导致死锁

t	T1	T2
1	sL1(A)	
2		sL2(B)
3	Read(A)	Read(B)
4	xL1(B)	xL2(A)
5	Wait	Wait
6	……	……

1、锁导致死锁

t	T1	T2
1	sL1(A)	
2		sL2(A)
3	Read(A)	Read(A)
4	A=A+B	
5	Upgrade(A)	A=A+100
6	Wait	Upgrade(A)
7	Wait	Wait
8	Wait	Wait
9	Wait	Wait
10	……	……

使用Update Lock

2、死锁的两种处理策略

- 死锁检测 Deadlock Detecting
  - 检测到死锁，再解锁
- 死锁预防 Deadlock Prevention
  - 提前采取措施防止出现死锁

3、Deadlock Detecting

- **Timeout 超时**
  - **Simple idea:** If a transaction hasn't completed in x minutes, abort it
- **Waiting graph 等待图**

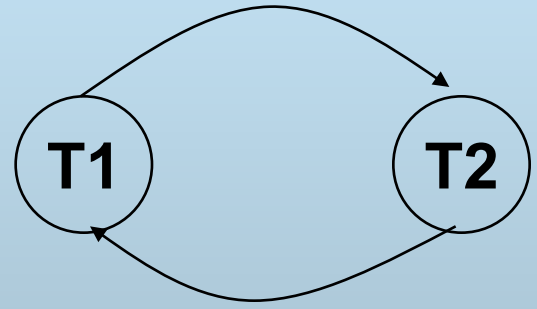
3、Deadlock Detecting

- **Waiting graph**
  - **Node：Transactions**
  - **Arcs：Ti → Tj, Ti必须等待Tj释放所持有的某个锁才能继续执行**

如果等待图中存在环路，
说明产生了死锁

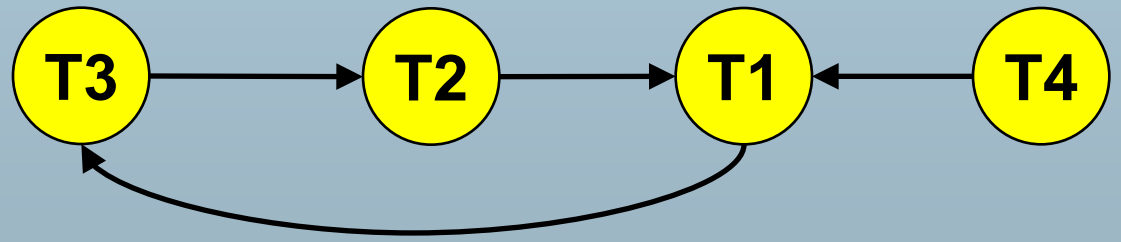
3、Deadlock Detecting

t	T1	T2
1	sL1(A)	
2		sL2(A)
3	Read(A)	Read(A)
4	sL1(B)	A=A+100
5	Read(B)	Upgrade(A)
6	A=A+B	Wait
7	Upgrade(A)	Wait
8	Wait	Wait
9	Wait	Wait
10	……	……



3、Deadlock Detecting

t	T1	T2	T3	T4
1	L1(A); r1(A)			
2		L2(C);r2(C)		
3			L3(B);r3(B)	
4				L4(D);r4(D)
5		L2(A); wait		
6			L3(C);wait	
7				L4(A);wait
8	L1(B);wait			



4、Deadlock Prevention

- 方法1： Priority Order
  - (按封锁对象的某种优先顺序加锁)
- 方法2：Timestamp
  - (使用时间戳)

4、Deadlock Prevention

- 方法1： Priority Order
  - 把要加锁的数据库元素按某种顺序排序
  - 事务只能按照元素顺序申请锁

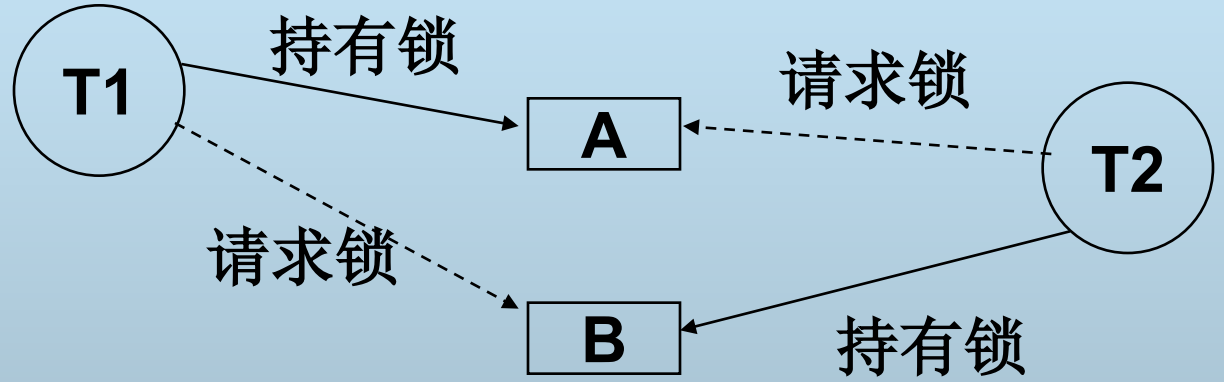
4、Deadlock Prevention

t	T1	T2	T3	T4
1	L1(A); r1(A)			
2		L2(A);wait		
3			L3(B);r3(B)	
4				L4(A);wait
5			L3(C);w3(C)	
6			U3(B);U3(C)	
7	L1(B);w1(B)			
8	U1(A);U1(B)			
9		L2(A);L2(C)		
10		r2(C);w2(A)		
11		u2(A);u2(C)		
12				L4(A);L4(D)
13				r4(D);w4(A)
14				U4(A);U4(D)

T1: A,B
T2: A,C
T3: B,C
T4: A,D

4、Deadlock Prevention

- 按序加锁可以预防死锁



Impossible!
T2获得B上的锁之前，必须先要获得A上的锁

4、Deadlock Prevention

- 方法2：Timestamp
  - 每个事务开始时赋予一个时间戳
  - 如果事务T被Rollback然后再Restart，T的时间戳不变
  - Ti请求被Tj持有的锁，根据Ti和Tj的timestamp决定锁的授予

4、Deadlock Prevention

- **Wait-Die Scheme 等待—死亡**
  - **T请求一个被U持有的锁**
    - If T is earlier than U then T WAITS for the lock
    - If T is later than U then T DIES 【*rollback*】
      - We later restart T with its original timestamp

Assumption：
timestamp(T) < timestamp(U) means T is earlier than U

4、Deadlock Prevention

t	T1	T2	T3	T4
1	L1(A); r1(A)			
2		L2(A);DIE		
3			L3(B);r3(B)	
4				L4(A);DIE
5			L3(C);w3(C)	
6			U3(B);U3(C)	
7	L1(B);w1(B)			
8	U1(A);U1(B)			
9				L4(A);L4(D)
10		L2(A);WAIT		
11				r4(D);w4(A)
12				U4(A);U4(D)
13		L2(A);L2(C)		
14		r2(C);w2(A)		
15		u2(A);u2(C)		

4、Deadlock Prevention

- **Wound-Wait Scheme 伤害－等待**
  - **T请求一个被U持有的锁**
    - If T is earlier than U then T WOUNDS U
      - U must release its locks then rollback and restart and the lock is given to T
    - If T is later than U then T WAITS for the lock

4、Deadlock Prevention

t	T1	T2	T3	T4
1	L1(A); r1(A)			
2		L2(A);WAIT		
3			L3(B);r3(B)	
4				L4(A);WAIT
5	L1(B);w1(B)		WOUNDED	
6	U1(A);U1(B)			
7		L2(A);L2(C)		
8		r2(C);w2(A)		
9		u2(A);u2(C)		
10				L4(A);L4(D)
11				r4(D);w4(A)
12				U4(A);U4(D)
13			L3(B);r3(B)	
14			L3(C);w3(C)	
15			U3(B);U3(C)	

4、Deadlock Prevention

- **Comparison**
  - **Wait-Die:**
    - Rollback总是发生在请求锁阶段，因此要Rollback的事务操作比较少，但Rollback的事务数会比较多
  - **Wound-Wait:**
    - 发生Rollback时，要Rollback的事务已经获得了锁，有可能已经执行了较长时间，因此Rollback的事务操作会较多，但Rollback的事务数预期较少，因为可以假设事务开始时总是先请求锁
    - 请求锁时WAIT要比WOUND要更普遍，因为一般情况下一个新事务要请求的锁总是被一个较早的事务所持有

4、Deadlock Prevention

- **Why wait-die and wound-wait work?**
  - 假设 T1→T2 →…… →Tk →T1 【*deadlock*】
    - 在wait-die scheme中，只有当Ti < Tj时才会有Ti→Tj，因此有
      - T1<T2<…<Tk<T1　　-- Impossible!
    - 在wound-wait scheme中，只有当 Ti > Tj时才会有Ti→Tj，因此有
      - T1>T2>…>Tk>T1　　-- Still impossible!

再论并发控制

- 哪些并发操作可能是冲突的？
  - 读-读 😀
  - 读-写 😢
  - 写-读 😢
  - 写-写 😢

  } 锁机制下都需要加锁，影响并发性能

- 代价
  - 加锁代价
  - 锁表存储代价
  - 死锁、活锁

  } 能否不用锁？

六、乐观并发控制

- 两种并发控制思路
  - 乐观并发控制 --- “乐观锁”
    - 乐观并发控制假定不太可能（但不是不可能）在多个用户间发生资源冲突，允许不锁定任何资源而执行事务。只有试图更改数据时才检查资源以确定是否发生冲突。如果发生冲突，应用程序必须读取数据并再次尝试进行更改。
  - 悲观并发控制 --- “悲观锁”
    - 立足于事先预防事务冲突
    - 采用锁机制实现，事务访问数据前都要申请锁

六、乐观并发控制

- 动机
  - 如果大部分事务都是只读事务，则并发冲突的概率比较低；即使不加锁，也不会破坏数据库的一致性；加锁反而会带来事务延迟
  - “读不加锁，写时协调”

六、乐观并发控制

- 在ADO程序中
  - Recordset打开时指定LockType
    - 0(adLockReadOnly)：recordset的记录为只读
    - 1(adLockPessimistic)：悲观并发控制，只要保持Recordset为打开,别人就无法编辑该记录集中的记录.
    - 2(adLockOptimistic)："乐观"并发控制，当update recordset中的记录时，将记录加锁
    - 3(adLockBatchOptimistic): 以批模式时更新记录时加锁

六、乐观并发控制

- 基于事后协调冲突的思想，用户访问数据时不加锁；如果发生冲突，则通过回滚某个冲突事务加以解决
- 由于读不需要加锁，因此开销较小，并发度高
- 但需要确定哪些事务发生了冲突
  - 使用“有效性确认(Validation)”

六、乐观并发控制

- 有效性确认协议
  - 每个更新事务Ti在其生命周期中按以下三个阶段顺序执行
    - 读阶段：数据被读入到事务Ti的局部变量中。此时所有write操作都针对局部变量，并不对数据库更新
    - 有效性确认阶段：Ti进行有效性检查，判定是否可以将write操作所更新的局部变量值写回数据库而不违反可串性
    - 写阶段：若Ti通过有效性检查，则进行实际的写数据库操作，否则回滚Ti

六、乐观并发控制

- 有效性检查方法（第二阶段）
  - 基于行版本的方式
    - Version --- MySQL
    - Timestamp --- MS SQL Server, Oracle
  - 基于值比较的方式
    - MS SQL Server

六、乐观并发控制

- 基于行版本的乐观并发控制 ——以MS SQL Server为例
  - MS SQL Server允许在游标中使用乐观并发控制

SQL Server支持的游标选项：

READ_ONLY
OPTIMISTIC WITH VALUES: 基于值比较的乐观并发控制
OPTIMISTIC WITH ROW VERSIONING: 基于时间戳的乐观并发控制
SCROLL_LOCKS: 悲观并发控制

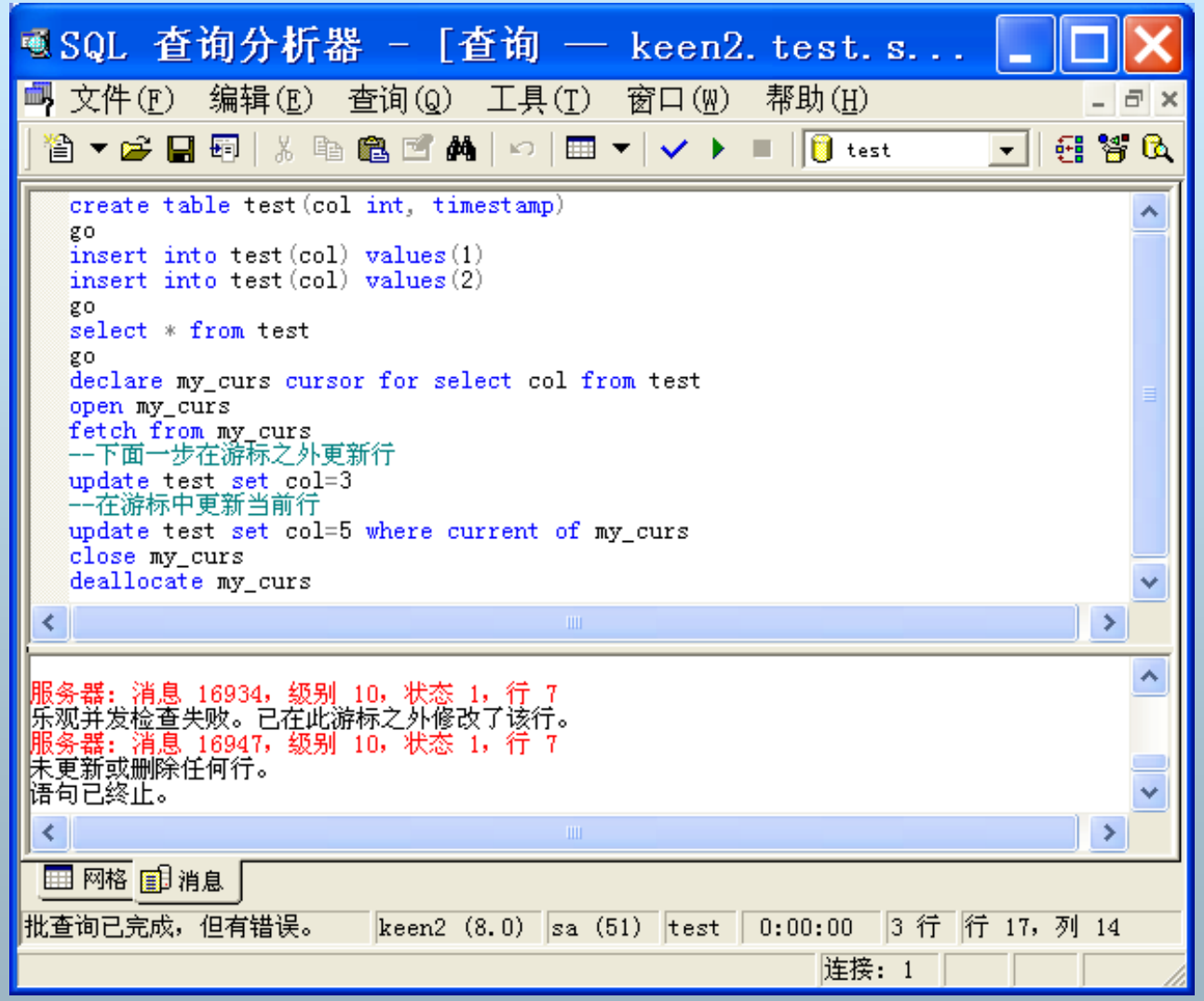
六、乐观并发控制

- 基于行版本的乐观并发控制 ——以MS SQL Server为例
  - MS SQL Server使用特殊数据类型timestamp（数据库范围内唯一的8字节二进制数）
  - 全局变量@@DBTS返回当前数据库最后所使用的时间戳值
  - 如果一个表包含 timestamp 列，则每次由 INSERT、UPDATE 或DELETE 语句修改一行时，此行的 timestamp 值就被置为当前的 @@DBTS 值，然后 @@DBTS 加1
  - 服务器可以比较某行的当前timestamp和游标提取时的timestamp值，确定是否更新

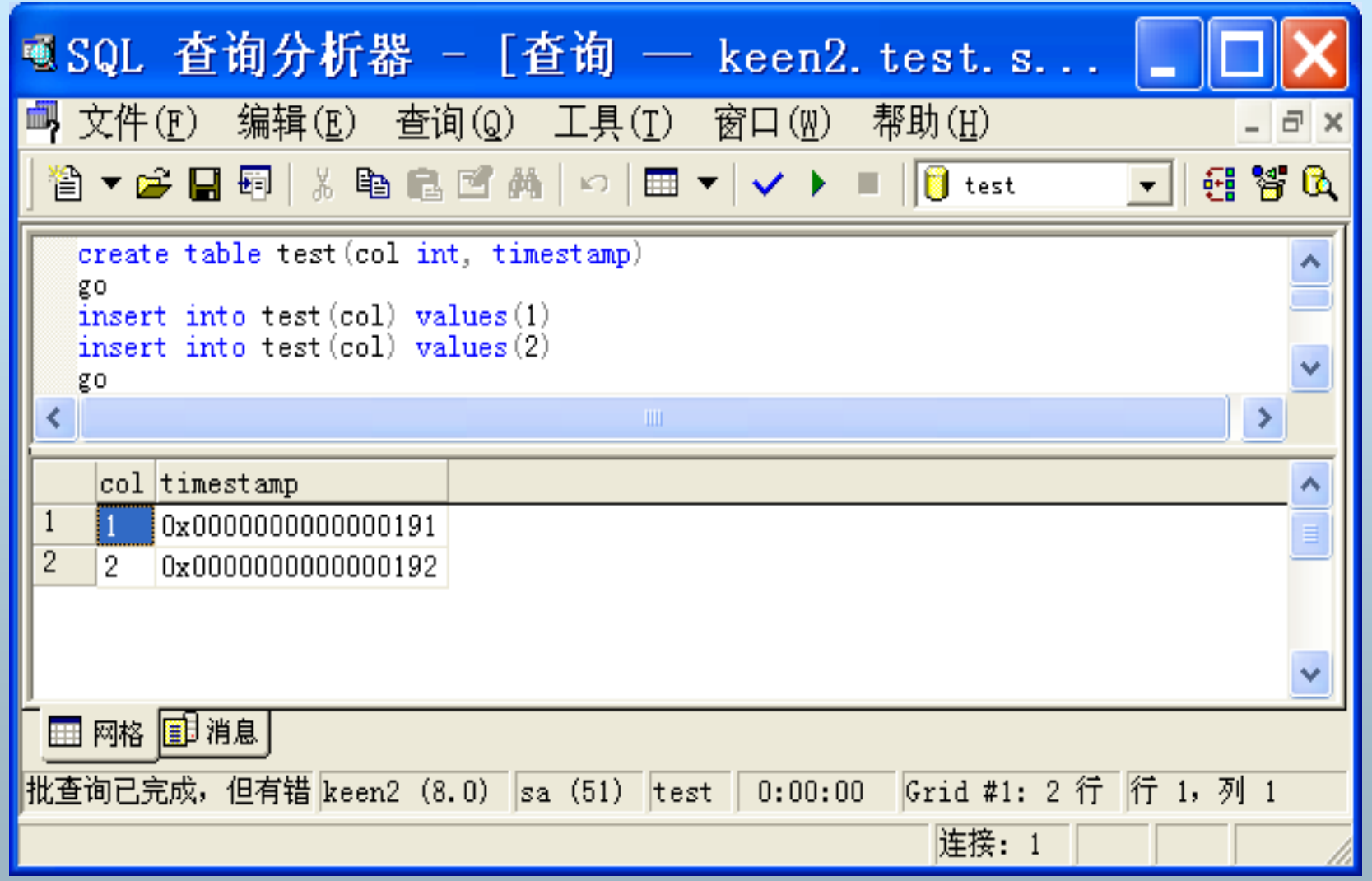
六、乐观并发控制

- 基于行版本的乐观并发控制 ——以MS SQL Server为例
  - 当用户打开游标时，SQL Server保存行的当前timestamp；当在游标中想更新一行时，SQL Server为更新数据自动添加一条Where子句
    - WHERE timestamp列 <= <old timestamp>
  - 如果不相等，则报错并回滚事务

六、乐观并发控制


SQL 查询分析器 - [查询 — keen2.test.s...
文件(F) 编辑(E) 查询(Q) 工具(T) 窗口(W) 帮助(H)
test
create table test(col int, timestamp)
go
insert into test(col) values(1)
insert into test(col) values(2)
go
select * from test
go
declare my_curs cursor for select col from test
open my_curs
fetch from my_curs
--下面一步在游标之外更新行
update test set col=3
--在游标中更新当前行
update test set col=5 where current of my_curs
close my_curs
deallocate my_curs
服务器: 消息 16934, 级别 10, 状态 1, 行 7
乐观并发检查失败。已在此游标之外修改了该行。
服务器: 消息 16947, 级别 10, 状态 1, 行 7
未更新或删除任何行。
语句已终止。
网格
消息
批查询已完成，但有错误。
keen2 (8.0)
sa (51)
test
0:00:00
3 行
行 17，列 14
连接: 1

六、乐观并发控制


SQL 查询分析器 - [查询 — keen2.test.s...
文件(F) 编辑(E) 查询(Q) 工具(T) 窗口(W) 帮助(H)
test
create table test(col int, timestamp)
go
insert into test(col) values(1)
insert into test(col) values(2)
go
col timestamp
1 1 0x0000000000000191
2 2 0x0000000000000192
网格 消息
批查询已完成，但有错 keen2 (8.0) sa (51) test 0:00:00 Grid #1: 2 行 行 1，列 1
连接: 1

六、乐观并发控制

- 基于值比较的乐观并发控制——MS SQL Server
  - 如果表中没有timestamp列，SQL Server在游标并发中将采用基于值比较的方式
  - 如果用户试图修改某一行，则此行的当前值会与最后一次提取此行时获取的值进行比较。如果任何值发生改变，则服务器就会知道其他人已更新了此行，并会返回一个错误。如果值是一样的，服务器就执行修改。

本章小结

- 并发操作问题
- 调度与可串性
  - 可串化调度
  - 冲突可串性及判断
- 锁与可串性实现
- 事务的隔离级别
- 死锁
- 乐观并发控制