



第18讲并发控制 (第18章)

关继宏教授

电子邮件: jhguan@tongji.edu.cn

计算机科学与技术系

同济大学

课程大纲

- **第0部分:概述**
 - Ch1:介绍
- **Part 1关系数据库**
 - Ch2:关系模型(数据模型, 关系代数)
 - Ch3&4: SQL(结构化查询语言)
 - Ch5:高级SQL
- **第二部分数据库设计**
 - Ch6:基于E-R模型的数据库设计
 - Ch7:关系型数据库设计
- **第三部分:应用程序设计与开发**
 - Ch8:复杂数据类型
 - Ch9:应用开发
- **Part 4大数据分析**
 - Ch10:大数据
 - Ch11:数据分析
- **第5部分:数据存储和索引**
 - Ch12:物理存储系统
 - Ch13:数据存储结构
 - Ch14:索引
- **第6部分:查询处理与优化**
 - Ch15:查询处理
 - Ch16:查询优化
- **第7部分:事务管理**
 - Ch17:交易
 - **Ch18:并发控制**
 - Ch19:恢复系统
- **第8部分:并行和分布式数据库**
 - Ch20:数据库系统架构
 - Ch21-23:并行和分布式存储, 查询处理和事务处理
- **第9部分**
 - **DB平台:OceanBase、MongoDB、Neo4J**

<s:1>并发控制

- 基于锁定的协议
- 基于协议
- 多粒度
- 死锁处理

并发控制问题

- 并发事务引起的问题


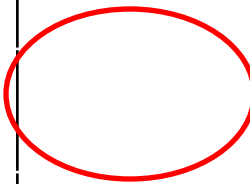
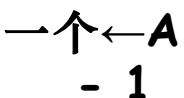
- Update丢失()
- 不可重复读取()
- 脏读()

- 符号

- $R(x)$: 读取 x
- $W(x)$: 写 x

失去了更新

- T1和t2读取相同的数据项并修改它
- t2的提交结果消除了t1的更新

T1	T2
① R (A) = 16	
②	R (A) = 16
③ 	
- 1	
W (A) = 15	
④	
	- 1

不可重复读

T1	T2
① R (A) = 50 R (B) = 100 金额 = 150	
②	R (B) = 100 ← B * 2 W (B) = 200
③ R (A) = 50 R (B) = 200 金额 = 250	

- T1读取B=100
- T2读取B, 然后更新B=200, 回写B
- T1再次读取B, B=200, 和第一次读取不一样
- 幻影现象()
 - 同一查询的记录消失或新记录出现

脏读

- T1将C修改为200, T2将C读为200
- T1由于某种原因回滚, 它的修改也回滚。然后C恢复到100
- T2将C读取为200, 与数据库不一致

T1	T2
① $R(C) = 100$ $C \leftarrow C * 2$ $W(C) = 200$	
②	$R(C) = 200$
③ 回滚 C 恢复到100	

大纲

- 并发控制

◁s:1▷基于锁的协议

- 基于协议
- 多粒度
- 死锁处理

基于锁定的协议

- 锁是一种控制对数据项并发访问的机制
- 数据项可以以两种模式锁定
 - **exclusive (X)模式()**。数据项可读可写。使用lock-X指令请求x锁
 - **共享(S)模式**。数据项只能读取。使用锁s指令请求s锁
- 锁请求是向并发控制管理器发出的。只有在请求被批准后，事务才能进行。

基于锁的协议(Cont.)

- 锁兼容矩阵()

	S	X
S	true	false
X	false	false

- 如果一个事务被请求的锁与其他事务在该数据项上已经持有的锁兼容，那么该事务可能会被授予该数据项上的锁。
- 如果不能授予锁，则请求事务等待，直到所有不兼容的锁都被释放。然后再授予锁。

无丢失更新

T1	T2
① Xlock A	
② $R(A) = 16$	
③ $A \leftarrow A - 1$	Xlock A
$W(A) = 15$	等待
提交	等待
解锁	等待
④	取Xlock A
	$R(A) = 15$
	$A \leftarrow A - 1$
⑤	$W(A) = 14$
	提交

可重复读取

T1	T2
①锁A Slock B R (A) = 50 R (B) = 100 金额 = 150	
②	Xlock B 等待 等待 等待 等待
③R (A) = 50 R (B) = 100 金额 = 150 提交 解锁 解锁B	等待 等待 等待 等待 等待 等待
④	得到XlockB R (B) = 100 ← B * 2
⑤	W (B) = 200 提交

无脏读

T1	T2
① Xlock C	
R (C) = 100	
$C \leftarrow C * 2$	
W (C) = 200	
②	Slock C
	等待
③ 回滚 (C rec. 100)	等待
打开C	等待
④	得到Slock C
	R (C) = 100
⑤	提交C
	打开C

基于锁定的协议

T:锁s (A):

 阅读(一);

 解锁(一个);

 锁s (B);

 阅读(B);

 解锁(B);

 显示器(A + B)

- 如上所述的锁定不足以保证可序列化性。如果在读取A和B之间对A和B进行更新，则显示的和将是错误的
- **锁定协议是一组规则**
 - 所有事务在请求和释放锁时都遵循
 - 锁定协议限制了可能的调度集

死锁()

- 考虑下面的部分时间表

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- 这种情况被称为死锁
 - 要处理死锁，必须回滚t3或t4并释放其锁
 - 死锁存在于大多数锁定协议中

饥饿()

- **饥饿**

- 例如，一个事务可能正在等待数据项上的x锁，而其他一系列事务请求并被授予相同数据项上的s锁
- **由于死锁，同一个事务被反复回滚**

- **并发控制管理器可以设计成防止饥饿**

两阶段锁定协议

- **2PL是一种确保冲突序列化调度的协议**
 - **阶段1:成长阶段**
 - 事务可以获得锁，但不能释放锁
 - **阶段2:收缩阶段()**
 - 事务可以释放锁，但不能获取锁
- **协议保证了可序列化性。可以证明事务可以按照其锁点的顺序序列化**
 - 锁点:

两阶段锁定协议

- 满足2PL

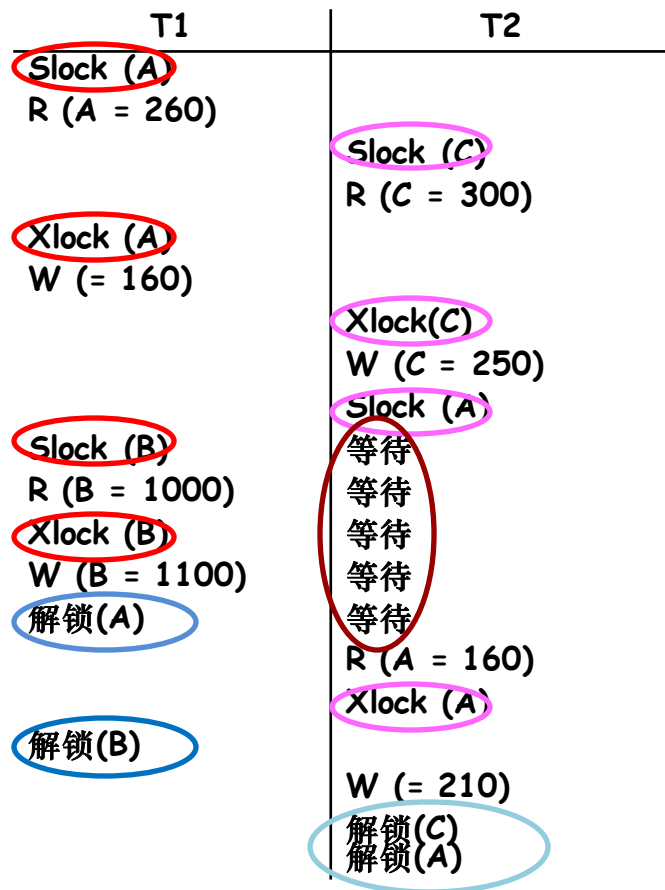
Slock A Slock B Xlock C 解锁B 解锁A 解锁C;

|←生长→| |←收缩→|

- 不满足2PL

Slock A 解锁A Slock B Xlock C 解锁C 解锁B;

两阶段锁定协议



2PL确保可序列化的时间表

两阶段锁定协议(Two-Phase Locking Protocol)

- **两阶段锁定无法避免死锁。**
- **在两阶段锁定下，级联回滚是可能的**
 - 为了避免这种情况,遵循修改后的协议称为严格两阶段锁(严格两阶段封锁)
 - 事务在提交之前必须持有所有的排他锁
- **严格的两阶段锁定()则更加严格**
 - **所有锁都被持有，直到提交/中止**
 - 事务可以按照提交的顺序序列化

锁转换()

- 带锁转换的两相锁定
 - 升级(Upgrade)
 - lock-S -> lock-X
 - 降级
 - lock-X -> lock-S
- 该协议保证了序列化性

T8: read(a 1)

读(a 2)

...

读(a n)

写(a 1)

T9: 读(a 1)

读(a 2)

显示(一个1 + 一个2)

自动获取锁

- A transaction T_i issues the standard **read/write** instruction, without explicit locking calls
- The operation **read(D)** is processed as:
 - if T_i has a **lock** on **D**
 - then
 - read(D)**
 - else
 - begin
 - if necessary **wait until no other** transactions have **a lock-X** on **D**
 - grant T_i a lock-S** on **D**;
 - read(D)**
 - end

自动获取锁(续)

- **write(D)** is processed as:
 - if T_i has a **lock-X** on **D**
 - then
 - write(D)**
 - else
 - begin
 - if necessary **wait** until **no other** transactions **have any lock** on **D**
 - if T_i has a **lock-S** on **D**
 - then
 - upgrade lock** on **D** to **lock-X**
 - else
 - grant T_i a lock-X** on **D**
 - write(D)**
 - end;
 - All locks are released after commit

锁的实现

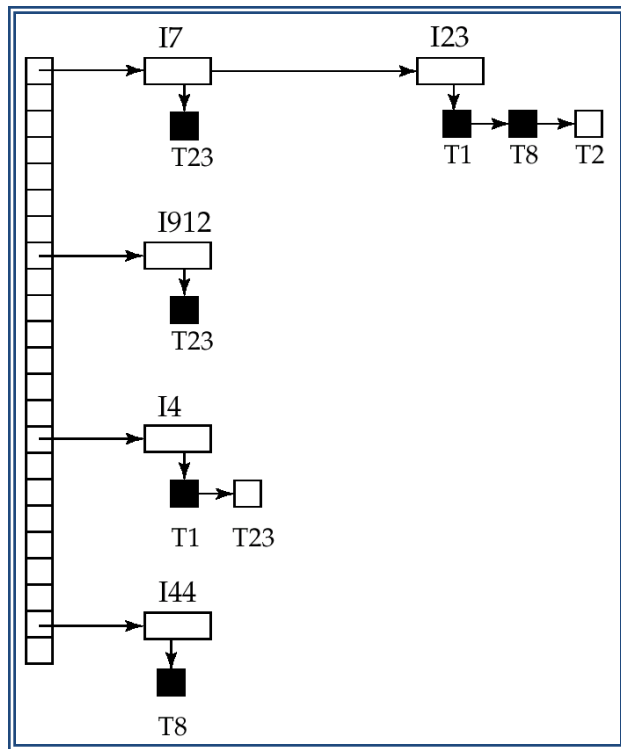
- **锁管理器()**

- 锁管理器可以作为一个单独的进程来实现，事务向它发送锁和解锁请求
- 锁管理器通过发送锁授予消息(或者发送请求事务回滚的消息，以防发生死锁)来应答锁请求
- 发出请求的事务等待，直到它的请求得到应答
- 锁管理器维护一个称为锁表(

- **锁表通常被实现为一个内存中的哈希表，以被锁数据项的名称为索引**

锁表

- **黑色矩形表示已授予的锁，白色矩形表示等待请求**
- 锁表还记录了授予或请求的锁的类型
- 新的请求被添加到数据项请求队列的末尾，如果它与所有早期的锁兼容，则授予
- **解锁请求会导致该请求被删除，之后的请求会被检查，看现在是否可以授予**
- 如果事务中止，则删除该事务的所有等待或授予的请求
 - **锁管理器可以保留每个事务持有的锁列表，以有效地实现这一点**



大纲

- 并发控制
- 基于锁定的协议

图文协议

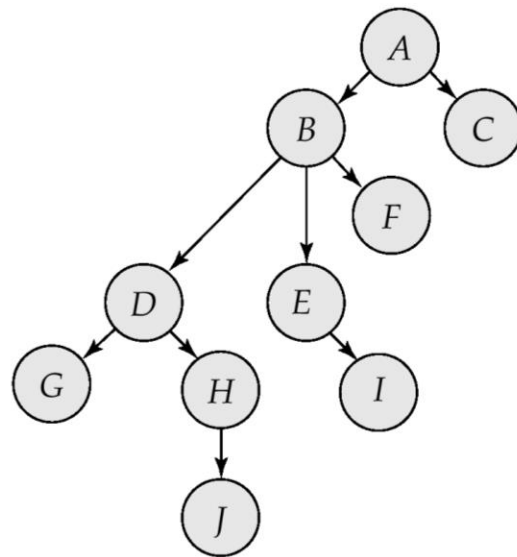
- 多粒度
- 死锁处理

基于协议

- **Graph-based protocols** are an alternative to two-phase locking
 - Impose a partial ordering \rightarrow (偏序) on the set $D = \{d_1, d_2, \dots, d_h\}$ of all data items
 - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j
 - Implies that the set D may now be viewed as a directed acyclic graph, called a database graph
- **The tree-protocol** is a simple kind of graph protocol.

树协议

- Only exclusive locks are allowed
 - The first lock by T_i may be on any data item
 - Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i
 - Data items may be unlocked at any time
 - A data item cannot be relocked by T_i



基于协议

- 树协议确保了冲突的可序列化性以及免于死锁
- 解锁可能比两阶段锁定协议- 2pl更早发生
 - 等待时间更短，并发性提高
 - 协议无死锁，不需要回滚
 - 事务的中止仍然会导致级联回滚
- 但是，可能必须锁定它不访问的数据项
 - 增加的锁定开销，以及额外的等待时间
 - 潜在的并发性降低

基于时间的协议

- Each transaction is issued a **timestamp** when it enters the system. If an **old transaction** T_i has timestamp $TS(T_i)$, a **new transaction** T_j is assigned timestamp $TS(T_j)$ such that **$TS(T_i) < TS(T_j)$** .
- The protocol manages concurrent execution such that the **timestamps determine the serializability order**
- In order to assure such behavior, the protocol maintains for each data **Q** two timestamp values:
 - **W-timestamp(Q)** is the **largest time-stamp** of any transaction that **executed write(Q) successfully**
 - **R-timestamp(Q)** is the **largest time-stamp** of any transaction that **executed read(Q) successfully**.

基于时间戳的协议(续)

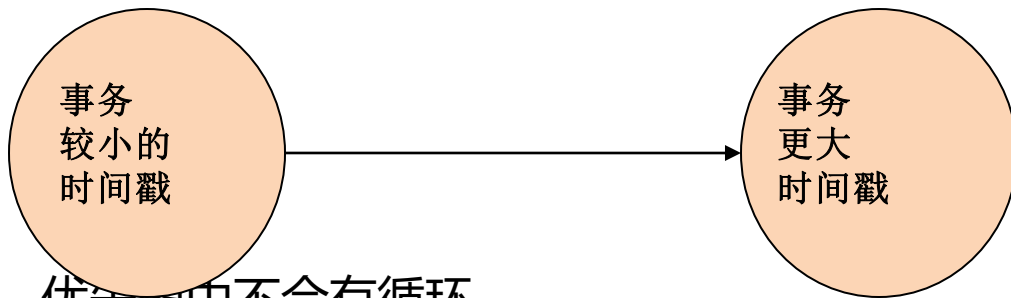
- The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order
- Suppose a transaction T_i issues a $\text{read}(Q)$
 - If $\text{TS}(T_i) \leq \text{W-timestamp}(Q)$, then T_i needs to read a value of Q that was already overwritten
 - the read operation is rejected, and T_i is rolled back
 - If $\text{TS}(T_i) \geq \text{W-timestamp}(Q)$, then the read operation is executed, and $\text{R-timestamp}(Q)$ is set to $\max(\text{R-timestamp}(Q), \text{TS}(T_i))$

基于时间戳的协议(Cont.)

- Suppose that transaction T_i issues $\text{write}(Q)$.
 - If $\text{TS}(T_i) < \text{R-timestamp}(Q)$, then the value of Q that T_i is producing was *needed previously*, and the system assumed that that value would never be produced.
 - Hence, the write operation is *rejected*, and T_i is *rolled back*.
 - If $\text{TS}(T_i) < \text{W-timestamp}(Q)$, then T_i is attempting to write an *obsolete* value of Q .
 - Hence, this write operation is *rejected*, and T_i is *rolled back*.
 - *Otherwise*, the write operation is executed, and $\text{W-timestamp}(Q)$ is set to $\text{TS}(T_i)$.

基于时间戳的协议(Cont.)

- 时间戳排序协议保证了序列化性，因为优先级图中的所有弧都是这样的形式：



- 因此，优先图中不会有循环
- 时间戳协议确保没有死锁，因为没有事务等待
- 但是调度可能不是没有级联的，甚至可能是不可恢复的

大纲

- 并发控制
- 基于锁定的协议
- 基于协议

◁s:1▷多粒度协议

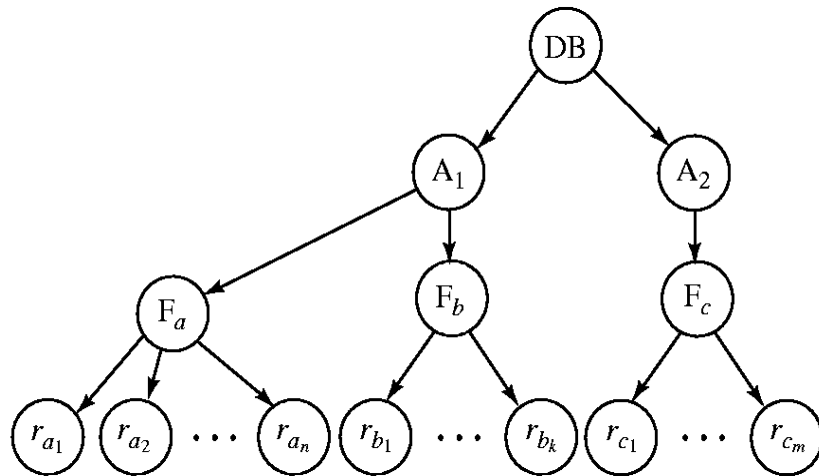
- 死锁处理

多粒度

- 允许数据项具有各种大小，并定义数据粒度的层次结构
- 可以图形化地表示为树吗
- 当一个事务显式地锁定树中的一个节点时，它以相同的模式隐式地锁定了该节点的所有后代
- **锁的粒度：**
 - **细粒度(树中较低):高并发性，高锁定开销**
 - **粗粒度(树中较高):锁开销低，并发性低**

粒度层次的例子

- 示例层次结构中的最高级别是整个数据库。
- 下面的级别依次是类型area, file和record。



意向锁()模式

- 多粒度附加三种锁模式:
 - **意向共享(IS)**
 - 指示在树的较低级别显式锁定，但仅使用共享锁
 - **intention-exclusive(第九)**
 - 指示在较低级别使用独占或共享锁进行显式锁定
 - **共享和意图排他(SIX)**
 - 由该节点根的子树在共享模式下显式锁定，而显式锁定是在较低级别上使用排他模式锁完成的
- 意图锁允许在不检查所有子节点的情况下，以S或X模式锁定更高级别的节点。

意向锁模式的兼容性矩阵

	IS	9。	年代	SIX	X
IS	✓	✓	✓	✓	×
9。	✓	✓	×	×	×
年代	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

多粒度锁定方案

- Transaction T_i can lock a node Q , using the following rules:
 - The lock compatibility matrix must be observed.
 - The root of the tree must be locked first, and may be locked in any mode.
 - A node Q can be locked by T_i in **S** or **IS** mode only if the parent of Q is currently locked by T_i in either **IX** or **IS** mode.
 - A node Q can be locked by T_i in **X**, **SIX**, or **IX** mode only if the parent of Q is currently locked by T_i in either **IX** or **SIX** mode.
 - T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
 - T_i can unlock a node Q only if none of the children of Q are currently locked by T_i .
- Locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.

大纲

- 并发控制
- 基于锁定的协议
- 基于协议
- 多粒度

◁s:1▷死锁处理

死锁处理

- 考虑以下两个事务:

t1:写(X) t2:写(Y)

写(Y)

写(X)

- 带死锁的调度

T1	T2
X上的锁X write (X)	Y上的锁X write (Y) 等待X上的锁X
等待Y上的锁X	

死锁处理

- 如果有一组事务，其中的每个事务都在等待另一个事务，系统就会死锁
- **死锁预防协议确保系统永远不会进入死锁状态。**
 - 要求每个事务在开始执行之前锁定它的所有数据项(预声明)。
 - 对所有数据项强制部分排序，并要求一个事务只能按照部分排序(基于图的协议)指定的顺序锁定数据项。

更多死锁预防策略

- 以下方案使用事务时间戳来防止死锁
 - **Wait-die方案——非抢占()**
 - 较老的事务等待较年轻的事务释放数据项，较年轻的事务从不等待较老的事务，而是回滚。
 - 在获取所需的数据项之前，一个事务可能会死亡几次
 - **Wound-wait方案- preemptive()**
 - 老的事务会强制回滚年轻的事务而不是等待它们，年轻的事务可能会等待老的事务。
 - 可能回滚比wait-die方案少

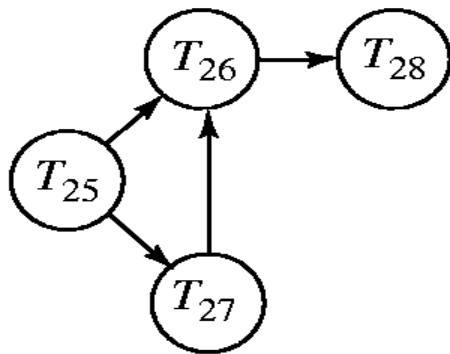
死锁预防(续)

- 在wait-die和wound-wait方案中都是如此
 - 回滚的事务使用其原始时间戳重新启动
 - 因此，较旧的事务优先于较新的事务，从而避免了饥饿
- **基于超时的方案**
 - 事务在指定的时间内等待锁。在此之后，事务被回滚
 - 因此，死锁是不可能的
 - 实现起来很简单，但是饿死是可能的。也很难确定超时间隔的好值。

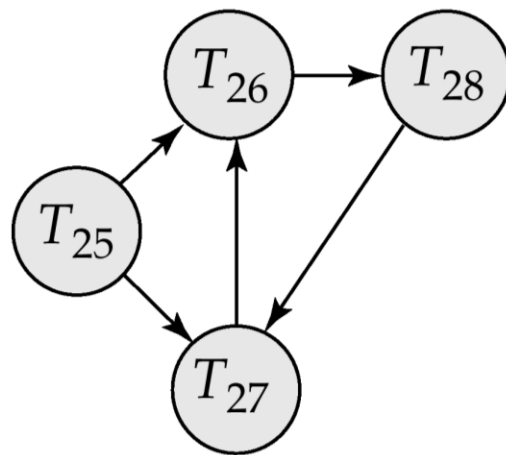
死锁检测

- Deadlocks can be described as **a wait-for graph(等待图)** $G = (V, E)$
 - **V** is a set of vertices (all the transactions in the system)
 - **E** is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$
 - If $T_i \rightarrow T_j$ is in **E**, then there is a **directed edge** from T_i to T_j , implying that T_i is waiting for T_j to release a data item
- The system is in a deadlock state **iff the wait-for graph has a cycle**.
Must invoke a deadlock-detection algorithm periodically to look for cycles.

死锁检测(续)



没有循环的Wait-for图



带循环的等待图

死锁的复苏

- **当检测到死锁时**
 - 某些事务需要回滚
 - **回滚——确定事务回滚多远**
 - **总回滚:中止事务，然后重新启动它**
 - **部分回滚:在需要打破死锁的情况下，更有效地回滚事务**
 - **如果总是选择相同的事务作为受害者，就会发生饥饿**
 - 在成本因子中包含回滚的次数以避免饿死

第18讲结束