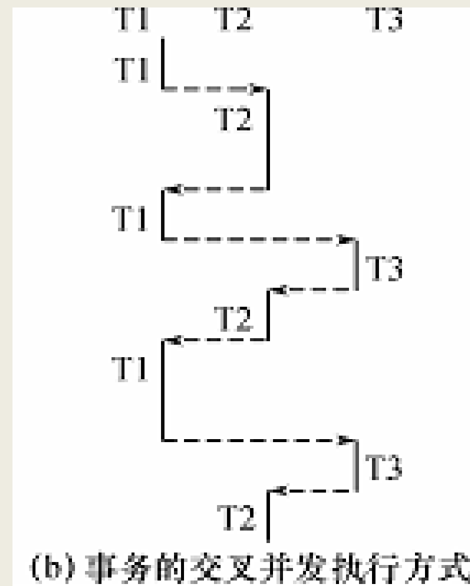
第十一章 并发控制

本章内容

- 并发控制概述
- 封锁及封锁协议
- 活锁和死锁
- 并发调度的可串行性
- 两段锁协议
- 封锁的粒度

并发控制概述

- 串行事务
  - 一个事务结束，另一个事务才开始
- 并行事务
  - 多个事务同时执行
  - **单处理器**：分时并发（交叉并发）
  - 多处理器：每个处理器处理一个事务，同时并发



(b) 事务的交叉并发执行方式

并发控制的目标

* **事务**是并发控制的基本单位
* 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

并发控制要解决的问题

- 问题
  - 多个事务同时存取同一数据，导致数据一致性被破坏。
  - **丢失修改**
  - **不可重复读**
  - **读脏数据**

事务并发——典型的不一致类型

- 丢失修改（Lost Update）
  - 事务A读取数据C
  - 并发事务B（随后）读取C
  - 事务A修改数据C
  - 事务B修改数据C
  - 事务A提交
  - 事务B提交
  - 事务B的修改（提交）使得事务A的修改被丢失

事务并发——典型的不一致类型

- 读‘脏’数据（Dirty read）
  - 事务A读取数据C并修改
  - 并发事务B（随后）读取C，并作处理
  - 事务A发现错误，回滚
  - 并发事务B并不知晓获得的数据是脏数据

事务并发——典型的不一致类型

- 不可重复读（Non-repeatable read）
  - 事务A读取数据C
  - 并发事务B读取数据C，并作修改，然后Commit
  - 事务A继续（再次）读取数据C，发现两次读取的数据C，是不一致的

事务并发——典型的不一致类型

- 幻影数据行（Phantom row）
  - 事务A读取数据C（满足某条件的数据）
  - 并发事务B插入数据（满足该条件的数据），然后Commit
  - 事务A继续（再次）读取数据C（满足该条件的数据），发现多出一些数据（而其他数据未被修改），就象是幻影一样

并发控制概述——典型的不一致类型

- 数据不一致性：由于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
- 对数据库的应用有时允许某些不一致性，例如有些统计工作涉及数据量很大，读到一些“脏”数据对统计精度没什么影响，可以降低对一致性的要求以减少系统开销

并发控制概述

- 事务管理器
  - 恢复控制部件、并发控制部件
- 并发控制部件
  - 对并发事务进行调度
  - 目标：
    - 事务的并发、穿插的结果与事务串行运行的结果一致（可串行性）
    - 保证事务的ACID

并发控制采用的技术

- 并发控制的主要技术
  - 封锁(Locking)
  - 时间戳(Timestamp)
  - 乐观控制法
  - 多版本并发控制(MVCC)

封锁（Locking）

- 并发调度的主要技术——锁
- 事务对需操作的数据进行加锁
  - 实现数据的互斥访问
  - 当一个事务访问某个数据项时，其他事务都不能修改该数据项
- 对数据的访问均必须加锁

封锁（Locking）——类型

- 排它锁（Exclusive Locks，X锁）
  - 写锁
  - 一旦事务T对数据A加上了X锁，只允许T对其进行读写
  - 其他事务需等T释放X锁之后，才能获锁
- 共享锁（Share Locks）
  - 读锁
  - 多个事务可对同一数据重复申请加读锁

封锁（Locking）——类型

- 读锁S与写锁X是不相容的锁

T2 / T1	X?	S?	-
X	N	N	Y
S	N	Y	Y
-	Y	Y	Y

封锁（Locking）——类型

- 锁的提升
  - 当事务T独占数据A的读锁时，可根据需要将其提升为写锁

封锁协议

封锁协议是：

- 运用锁的规则
- **何时加锁、持锁时间、何时释放**

- 不同的协议，达到不同的并发效果
- 不同的协议，在不同的程度上为并发操作的正确调度提供一定的保证。

封锁协议——一级封锁协议

- 修改数据必须加写锁，直至事务结束（Commit/Rollback）才释放锁
  - 读取数据不必加锁
  - 可以防止丢失修改
  - 仍存在读脏数据的问题（因为读操作未申请读锁）
  - 这是DB 的缺省状态

使用封锁机制解决丢失修改问题

例：

	T_1	T_2
①	Xlock A	
②	R(A)=16	
		Xlock A
③	A←A-1	等待
	W(A)=15	等待
	Commit	等待
	Unlock A	等待
④		获得Xlock A
		R(A)=15
		A←A-1
⑤		W(A)=14
		Commit
		Unlock A

没有丢失修改

- 事务T_1在读A进行修改之前先对A加X锁
- 当T_2再请求对A加X锁时被拒绝
- T_2只能等待T_1释放A上的锁后获得对A的X锁
- 这时T_2读到的A已经是T_1更新过的值15
- T_2按此新的A值进行运算，并将结果值A=14写回到磁盘。避免了丢失T_1的更新。

封锁协议——二级封锁协议

- 修改数据必须加写锁，直到事务结束
- 读数据也必须加读锁，读完即可释放
  - 可以解决读脏数据的问题（在前一事务未提交修改之前，读锁申请不成功）
  - 仍存在不可重复读的问题

使用封锁机制解决读“脏”数据问题

例 T_1	T_2
① Xlock C	
R(C)=100	
C←C*2	
W(C)=200	
②	Slock C
	等待
③ROLLBACK	等待
(C恢复为100)	等待
Unlock C	等待
④	获得Slock C
	R(C)=100
⑤	Commit C
	Unlock C

不读“脏”数据

- 事务T_1在对C进行修改之前，先对C加X锁，修改其值后写回磁盘
- T_2请求在C上加S锁，因T_1已在C上加了X锁，T_2只能等待
- T_1因某种原因被撤销，C恢复为原值100
- T_1释放C上的X锁后T_2获得C上的S锁，读C=100。避免了T_2读“脏”数据

封锁协议——三级封锁协议

- 修改数据必须加写锁，直到事务结束
- 读数据也必须加读锁，直到事务结束
  - 可以解决不可重复读的问题
  - 仍存在幻影数据行的问题
  - 可增加幻影锁

使用封锁机制解决不可重复读问题

T_1	T_2
① Slock A	
Slock B	
R(A)=50	
R(B)=100	
求和=150	
②	Xlock B
	等待
③ R(A)=50	等待
R(B)=100	等待
求和=150	等待
Commit	等待
Unlock A	等待
Unlock B	等待
④	获得XlockB
	R(B)=100
	B←B*2
⑤	W(B)=200
	Commit
	Unlock B

可重复读

- 事务T_1在读A，B之前，先对A，B加S锁
- 其他事务只能再对A，B加S锁，而不能加X锁，即其他事务只能读A，B，而不能修改
- 当T_2为修改B而申请对B的X锁时被拒绝只能等待T_1释放B上的锁
- T_1为验算再读A，B，这时读出的B仍是100，求和结果仍为150，即可重复读
- T_1结束才释放A，B上的S锁。T_2才获得对B的X锁

封锁协议

* 不同的需求，使用不同的锁协议
  * 并发程度
  * 数据要求
* 不同协议的效果

封锁协议级别越高，一致性程度越高

	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
一级封锁协议		√			√		
二级封锁协议		√	√		√	√	
三级封锁协议		√		√	√	√	√

封锁带来的问题

- 活锁
- 死锁

锁的异常——活锁

- 当多个事务请求封锁同一数据对象时，有可能出现锁的异常
- 活锁
  - 多个事务申请对数据R加锁，而系统随机地加锁，导致某些事务长等
- 活锁策略
  - 设定系统按事务申请锁的时间戳顺序进行排队

活锁（示例）

T_1	T_2	T_3	T_4
Lock R	• • •	• • •	• • •
•	Lock R		
•	等待	Lock R	
•	等待	•	Lock R
Unlock R	等待	•	等待
	等待	Lock R	等待
•	等待	•	等待
•	等待	Unlock	等待
•	等待	•	Lock R
	等待	•	• •

锁的异常——死锁

- 死锁
  - 两个（或多个）事务互相申请对方加锁对象的排它锁，造成循环等待。

死锁（示例）

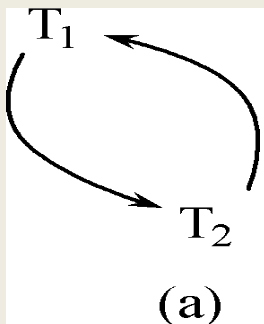
T_1	T_2
Lock R_1	• • •
•	Lock R_2
• •	• •
Lock R_2	•
等待	
等待	
等待	Lock R_1
等待	等待
等待	等待
	• • •

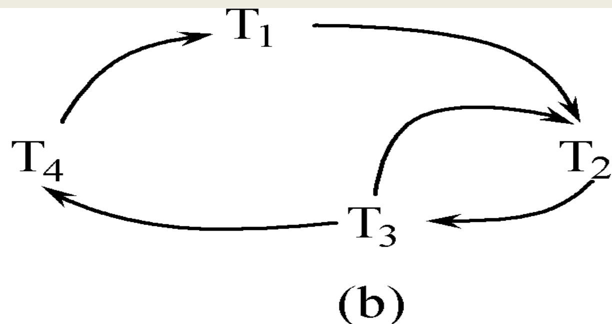
锁的异常——死锁的策略

* 预防
  * 一次加锁法
    * 事务一次性对需要的数据进行加锁
    ? 降低了系统的并发性，数据的不可预见性
  * 顺序加锁法
    * 估算需要的锁，对系统中的锁设定一个加锁顺序，所有事务均按照该顺序进行加锁
    ? 很难估算
  * 死锁几乎无法避免，允许死锁的发生，对其进行诊断和解除

锁的异常——死锁的策略

- 诊断
  - 超时法——设定时限，超过即为死锁
  - 等待图法——回路即为死锁





- 解除
  - 杀死
    - 时间戳
    - 代价最小

并发调度的可串行性

- 事务相互隔离的理想状态：串行运行
- 并发事务的调度目标
  - 等同于串行运行的结果
- 可串行化
  - 多个事务的并发执行是正确的，当且仅当其结果与按某一顺序串行地执行的结果相同

串行调度: 正确的调度

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

串行调度(a)

- 假设A、B的初值均为2。
- 按$T_1 \rightarrow T_2$次序执行结果为

  A=3，B=4
- 串行调度策略,正确的调度

串行调度: 正确的调度

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

串行调度(b)

- 假设A、B的初值均为2。
- $T_2 \rightarrow T_1$次序执行结果为B=3，A=4
- 串行调度策略,正确的调度

不可串行化调度：错误的调度

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

不可串行化的调度

- 执行结果与(a)、(b)的结果都不同
- 是错误的调度

可串行化调度：正确的调度

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

可串行化的调度

- 执行结果与串行调度(a)的执行结果相同
- 是正确的调度

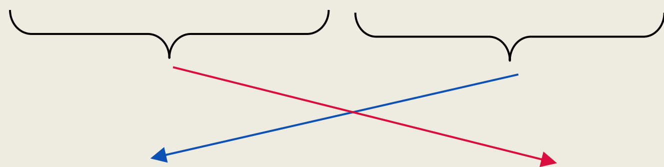
冲突可串行化调度

- 冲突操作：
  事务1读x,事务2写x
  事务1写x,事务2写x
- 定义：如果一个调度在保证冲突操作的次序不变的情况下，通过交换两个事务不冲突操作的次序得到另一个调度，而该调度是串行的，则称为**冲突可串行化的调度**。

冲突可串行化调度（举例）

- 若一个调度是冲突可串行化，则一定是可串行化的调度

$Sc_1=r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$



$Sc_2=r_1(A)w_1(A)r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B)$

Sc_2等价于一个串行调度T_1，T_2。所以Sc_1冲突可串行化的调度

冲突可串行化调度

- 冲突可串行化调度是可串行化调度的充分条件，不是必要条件。还有不满足冲突可串行化条件的可串行化调度。

[例]有3个事务

$T_1=W_1(Y)W_1(X)$，$T_2=W_2(Y)W_2(X)$，$T_3=W_3(X)$

- 调度$L_1=W_1(Y)\mathbf{W_1(X)}W_2(Y)W_2(X)\ W_3(X)$是一个串行调度。
- 调度$L_2=W_1(Y)W_2(Y)W_2(X)\mathbf{W_1(X)}W_3(X)$ 不满足冲突可串行化。但是调度L_2是可串行化的，因为L_2执行的结果与调度L_1相同，Y的值都等于T_2的值，X的值都等于T_3的值

实现并发调度的可串行性

- 两段锁协议
  - 所有事务必须分两阶段对数据项加锁和解锁
  - 扩展阶段
    - 事务可对需要的数据项进行加锁或提升锁
  - 收缩阶段
    - 一旦任一事务释放了一个锁，则进入收缩阶段
    - 事务只能释放锁，而不能申请获得任何锁
- 两阶段锁协议的并发调度是可串行的。
- 两阶段锁协议，仍有死锁的可能。

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
    - 事务可以申请获得任何数据项上的任何类型的锁，但是不能释放任何锁
  - 第二阶段是释放封锁，也称为收缩阶段
    - 事务可以释放任何数据项上的任何类型的锁，但是不能再申请任何锁

两段锁协议（续）

例： 事务T_i遵守两段锁协议，其封锁序列是：

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

|← 扩展阶段 →||← 收缩阶段 →|

事务T_j不遵守两段锁协议，其封锁序列是：

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

两段锁协议（续）：**遵守两段锁协议的可串行化调度**

事务T_1	事务T_2
Slock A	
R(A)=260	
	Slock C
	R(C)=300
Xlock A	
W(A)=160	
	Xlock C
	W(C)=250
	Slock A
Slock B	等待
R(B)=1000	等待
Xlock B	等待
W(B)=1100	等待
Unlock A	等待
	R(A)=160
	Xlock A
Unlock B	
	W(A)=210
	Unlock C

- 左图的调度是遵守两段锁协议的，因此一定是一个可串行化调度。

两段锁协议（续）

- 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**。
- 若并发事务都遵守两段锁协议，则对这些事务的任何并发调度策略都是可串行化的。
- 若并发事务的一个调度是可串行化的，不一定所有事务都符合两段锁协议。

两段锁协议（续）

[例]　遵守两段锁协议的事务发生死锁

事务T_1	事务T_2
Slock B	
R(B)=2	
	Slock A
	R(A)=2
Xlock A	
等待	Xlock A
等待	等待

遵守两段锁协议的事务可能发生死锁

锁的粒度

- 封锁对象的大小，即为锁的**粒度**
- 库级、表级、页面、行级、属性级
- 锁粒度与并发度
  - 锁粒度越大（可封锁的数据越大），并发度越小
  - 锁粒度越小，并发度越大
- 多粒度封锁
  - 在一个系统中，同时支持多种粒度，供事务选择

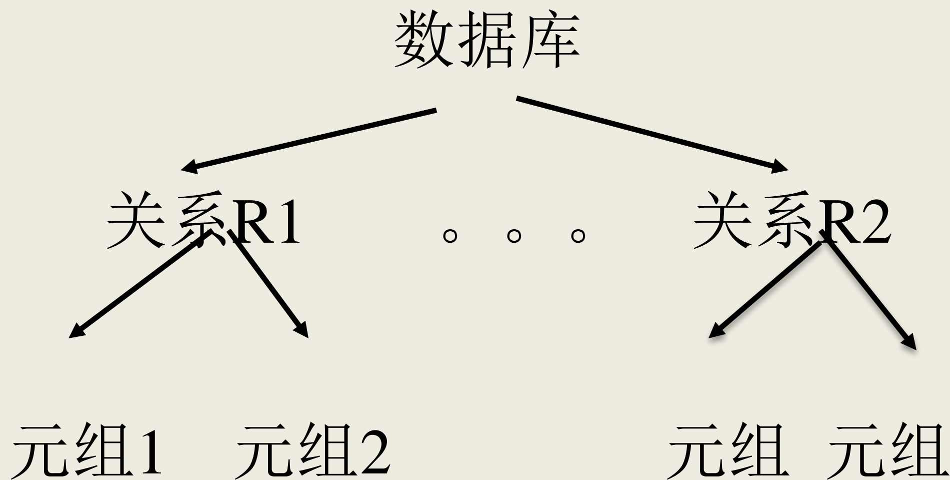
选择封锁粒度的原则（续）

- 若封锁粒度是数据页，事务T_1需要修改元组L_1，则T_1对包含L_1的整个数据页A加锁。如果T_1对A加锁后事务T_2要修改A中元组L_2，则T_2被迫等待，直到T_1释放A。
- 如果封锁粒度是元组，则T_1和T_2可以同时对L_1和L_2加锁，不需要互相等待，提高了系统的并行度。
- 如事务T需要读取整个表，若封锁粒度是元组，T必须对表中的每一个元组加锁，开销极大

选择封锁粒度

- 同时考虑封锁开销和并发度两个因素，适当选择封锁粒度
  - 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



- 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**:是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）

- 系统检查封锁冲突时
  - 要检查显式封锁
  - 还要检查隐式封锁
- 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

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

意向锁

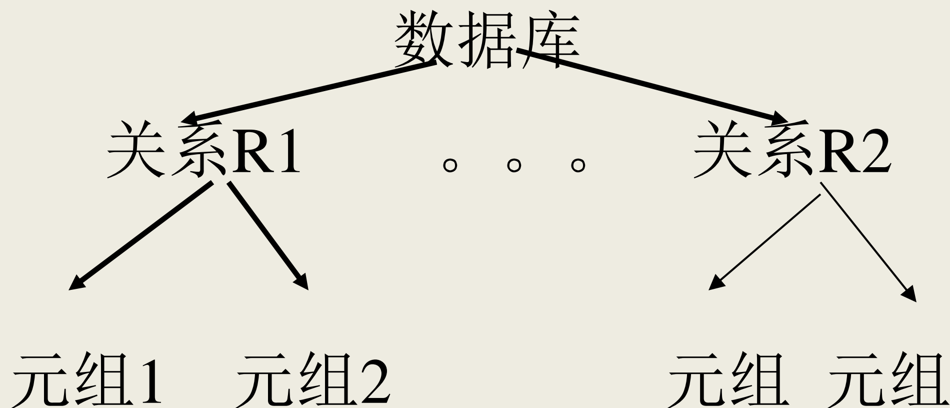
- **目的**：**提高对某个数据对象加锁时系统的检查效率**
- 含义：对某个结点加意向锁，则说明该结点的下层结点正在被加锁；对某一结点加锁，必须对它的上层加意向锁。
- 类型

IS锁：子结点加S锁

IX锁：子结点加X锁

SIX锁= S锁（结点）+ IX锁（子结点）

意向锁的使用



- 如果T1事务想修改元组1，则对数据库和关系R1加IX锁，元组1加X锁。
- 如果T2想读元组2，则对数据库和关系R1加IS锁，元组2加S锁。

问题：T1和T2可以同时运行吗？

意向锁（续）

意向锁的相容矩阵

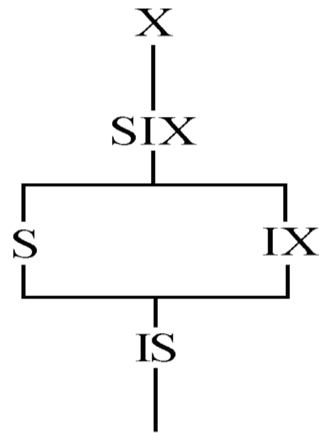
T_1 \ T_2	S	X	IS	IX	SIX	–
S	Y	N	Y	N	N	Y
X	N	N	N	N	N	Y
IS	Y	N	Y	Y	Y	Y
IX	N	N	Y	Y	N	Y
SIX	N	N	Y	N	N	Y
–	Y	Y	Y	Y	Y	Y

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T_1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

总结

- 并发控制采用的目的和技术
- 封锁协议
- 活锁和死锁的解决方法
- 调度的原则：可串行性
- 两段锁协议
- 多粒度封锁与意向锁