事务管理，并发控制和恢复

事物的基本概念

事物定义

事务是用户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。例如,在关系数据库中,一个事务可以是一条 SQL 语句、一组SQL 语句或整个程序

> A transaction is a unit of program execution that accesses and possibly updates various data items. A transaction is delimited by statements (or functions calls) of the form begin transaction and end transaction. The transaction consists of all operations executed between the begin transaction and end transaction. This collection of steps must appear to the user as a single, indivisible unit.

事物的ACID特性

事务具有4个特性:**原子性(Atomicity)**、**一致性(Consistency)**、**隔离性(Isolation)**和**持续性(Durability)**。这4个特性简称为ACID特性(ACID properties)。

- 原子性：事务是数据库的逻辑工作单元，事务中包括的诸操作要么都做，要么都不做

  > Since a transaction is indivisible, it either executes in its entirety or not at all. Thus, if a transaction begins to execute but fails for whatever reason, any changes to the database that the transaction may have made must be undone. This requirement holds regardless of whether the transaction itself failed, the operating system crashed, or the computer itself stopped operating. This “all-or-none” property is referred to as atomicity(原子性).

- 一致性：要保证事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态

  > Because of the above three properties, transactions are an ideal way of structuring interaction with a database. This leads us to impose a requirement on transactions themselves. A transaction must preserve database consistency-if a transaction is run atomically in isolation starting from a consistent database, the database must again be consistent at the end of the transaction. This consistency requirement goes beyond the data-integrity constraints (教材第2-5章，数据完整性约束). Rather, transactions are expected to go beyond that to ensure preservation of those application-dependent consistency constraints that are too complex to state using the SQL constructs for data integrity. How this is done is the responsibility of the programmer who codes a transaction. This property is referred to as consistency(一致性).

- 隔离性：一个事务的执行不能被其他事务干扰，一个事务的内部操作及使用的数据对其他并发事务是隔离的，并发执行的各个事务之间不能相互干扰

  > Transaction-processing systems usually allow multiple transactions to run concurrently. The database system must control the interaction among the concurrent transactions to prevent them from destroying the consistency of the

> database. It does so through a variety of mechanisms called concurrency-control schemes(并发控制策略，第11章内容).
> Since a transaction is a single unit, its actions cannot appear to be separated by other database operations not part of the transaction. Therefore, the database system must take special actions to ensure that transactions operate properly without interference from concurrently executing database statements. This property is referred to as isolation.

- 持续性：一个事务一旦提交，它对数据库中数据的改变就应该是永久性的，接下来的其他操作或故障不应该对其执行结果有任何影响

> Even if the system ensures correct execution of a transaction, this serves little purpose if the system subsequently crashes and, as a result, the system "forgets" about the transaction. Thus, a transaction's actions must persist across crashes. This property is referred to durability(持续性).

故障的种类

- Transaction failure——事务故障（包括：logical error，逻辑错误；system error，系统错误）
- System crash——系统故障（断电，CPU故障，影响正在运行的事物但是不破坏数据库）（软故障）
- Disk failure——介质故障（磁盘损坏，强磁场干扰，破坏数据库）（硬故障）

登记日志文件

The most widely used structure for recording database modifications is the log (日志). The log is a sequence of - log records, recording all the update activities in the database.

- 日志文件是用来记录事务对数据库的更新操作的文件

并发控制

The transaction is the unit of the concurrency control in database system.

并发控制可能带来的数据不一致性

Concurrency operating can cause the following data inconsistency(数据不一致性)：

丢失修改 (Lost Update)

这种问题发生在两个或多个事务同时读取相同数据并尝试更新它时。如果一个事务的更新在另一个事务的更新之后被提交，那么先前的修改就会丢失。

假设有两个事务T1和T2同时工作在同一个账户余额记录上。

- T1读取账户余额为100，并基于这个值计算新的余额为150。
- 同时，T2也读取了同一个账户余额100，并计算新的余额为120。
- 如果T2先提交了它的更新，将余额改为$120。
- T1之后提交其更改，余额变为$150。 结果，T2的更新就被“丢失”了。

不可重复读 (Non-repeatable Read)

不可重复读发生在一个事务试图两次读取同一数据行时，却发现中间被另一个事务更新，导致两次读取的结果不一致。

- T1读取某个客户的账户余额，此时是$200。
- T2接着更新该客户的账户余额为$300，并提交了事务。
- T1再次读取同一账户余额时，发现余额已经变为300，与之前的200不一致。

读取“脏”数据 (Dirty Read)

读取“脏”数据是指一个事务读取了另一个未提交事务修改的数据，如果那个修改最终回滚了，那么读取的数据就是临时且无效的。

- T1修改一条记录，将销售记录的数量从10增加到15，但是还没有提交事务。
- T2读取了这条销售记录，看到数量为15。
- T1由于某种原因回滚了事务，数量重置为10。
- T2此时具有了一条无效的销售记录信息，因为它读取了“脏”数据。

封锁

封锁技术

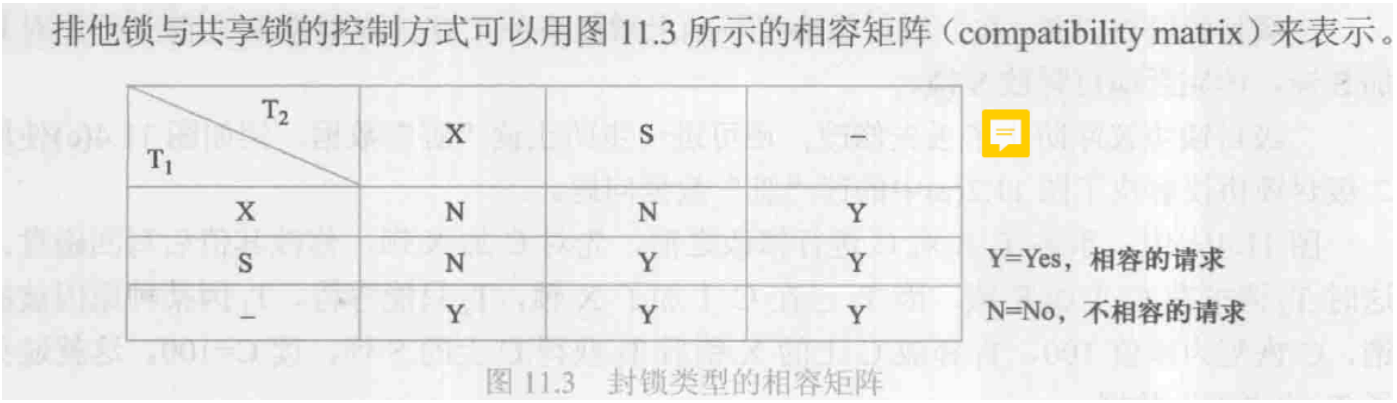
基本的封锁类型有两种：排他锁(exclusive locks,简称X锁)和共享锁(share locks,简称S锁)。

1.排他锁（写锁）

排他锁，或称为写锁，是一种保护数据对象不被其他事务修改或读取的锁。当一个事务对数据对象加上排他锁之后，该事务可以自由地读取和修改这个数据对象，但是在这个锁被释放之前，其他任何事务都不能对这个数据对象加任何类型的锁，无论是排他锁还是共享锁。这样做的目的是为了防止数据在一个事务处理期间被其他事务更改，从而保证事务的隔离性和数据的一致性。

2. 共享锁（读锁）

共享锁，或称为读锁，允许事务对一个数据对象进行读取而不进行修改。如果一个事务对数据对象加上共享锁，那么这个事务可以读取数据对象但不能修改它。其他事务只能再这个数据对象加上共享锁（不能加x锁）来读取数据，但它们不能加排他锁来修改数据。这样做能确保多个事务可以同时读取数据，但在任何事务尝试修改数据之前，需要等待所有共享锁被释放。


排他锁与共享锁的控制方式可以用图 11.3 所示的相容矩阵（compatibility matrix）来表示。

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

图 11.3　封锁类型的相容矩阵

- 读读不互斥，读写，写写互斥

封锁协议

在运用X锁和S锁这两种基本封锁对数据对象加锁时,还需要约定一些规则。例如, 何时申请X锁或S锁、持锁时间、何时释放等。这些规则称为封锁协议(locking protocol)。

数据库三级封锁协议 | Hong's Blog (hongscar.github.io)

1.一级锁协议：事务T修改数据R之前，先加X锁，直到事务结束时释放（无论是正常结束commit还是非正常结束rollback）

作用：可防止 更新丢失问题。

不能防止： 脏读，不可重复读，幻读等。

PS：任何数据库都至少满足一级锁的协议，因为更新丢失是不能接受的错误，所以更新丢失一般只存在于理论讨论中，实际应用中基本不会出现这个问题。

2.二级锁协议：在一级锁的条件下，T在读取R之前先加S锁，读完后释放。

作用：可防止更新丢失，脏读。

不能防止： 不可重复读，幻读。

3.三级锁协议：在一级锁的条件下，T在读取R之前先加S锁，事务结束后释放。

作用：可防止 更新丢失，脏读，不可重复读，幻读。

活锁和死锁

活锁

如果事务T1封锁了数据R,事务T2又请求封锁R,于是T2等待;T3也请求封锁R, 当T1释放了R上的封锁之后系统首先批准了T3的请求,T2仍然等待;然后T4又请求封锁 R,当T3释放了R上的封锁之后系统又批准了T4的请求 …… T2有可能永远等待,这就是 活锁的情形

避免活锁的简单方法是**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,封锁子系统按请求封锁的先后次序对事务排队,数据对象上的锁一旦释放就批准申请队列中第一个事务获得锁。

```
It is possible that there is a sequence of transactions that each requests a
shared-mode lock in the data item, and each transaction releases the lock a short
while after it is granted, but another transaction never gets the exclusive-mode
lock on the data item. The transaction may never make progress, and is said to be
starved, also called live lock.
A simple way to avoid live locks is to adopt the first come first serve. (先来先服务
策略.)
```

死锁

死锁是指两个或多个事务在执行过程中，由于竞争资源而形成的一种相互等待的现象，若无外力作用，他们都将无法继续执行。这一集合中的每一个独立线程都在等待另一单元释放锁，诸事务相互之间等待，形成闭环，谁也不愿意放弃自己已获得的资源。

死锁产生的条件和处理死锁的方法-CSDN博客

> There are two approaches to deadlock prevention. One approach ensures that no cyclic waits can occur by ordering the requests for locks, or requiring all locks to be acquired together. The other approach is closer to deadlock recovery.

并发控制机制

> Concurrency-control schemes —并发控制机制
> The database system must control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database. It does so through a variety of mechanisms called concurrency-control schemes(并发控制机制).

并发调度的可串行性

> Transactions run serially : one at a time, each starting only after the previous one has completed. However, there are two good reasons for allowing concurrency:
> - Improved throughput and resource utilization. (提高吞吐量和资源利用率)
> - Reduced waiting time. (减少等待时间)
>
> When several transactions run concurrently, the isolation property may be violated, resulting in database consistency destroyed despite the correctness of each individual transaction.

定义 多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这些事务时的结果相同，称这种调度策略为**可串行化**（serializable）**调度**。

可串行性（serializability）是并发事务正确调度的准则。按这个准则规定，一个给定的并发调度，当且仅当它是可串行化的，才认为是**正确调度**。

冲突的可串行化

数据库事务中调度串行化、冲突可串行化、前趋图(优先图)_可串行化调度-CSDN博客

Conflict serializability —冲突可串行化
The execution sequences are called schedules (调度). They represent the chronological order in which instructions are executed in the system. Clearly, a schedule for a set of transactions must consist of all instructions of those transactions and they must preserve the order in which the instructions appear in each individual transaction.
When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial. It is the job of the database system to ensure that any schedule that is executed will leave the database in a consistent stake. The concurrency-control component of the database system carries out this task.

We can ensure consistency of the database under concurrent execution by making sure that any schedule that is executed has the same effect as a schedule that could have occurred without any concurrent execution. That is, the schedule should, in some sense, be equivalent to a serial schedule. Such schedules are called serializable (可串行化) schedules. We focus on a particular form called conflict serializability (冲突可串行化).

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

(a) 串行调度

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	

(b) 串行调度

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)=2
	Unlock A
	Xlock B
	B=X+1=3
	W(B)
	Unlock B

(c) 不可串行化的调度

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

(d) 可串行化的调度

图 11.7 并发事务的不同调度

冲突操作是指不同的事务对同一个数据的读写操作和写写操作：

$R_i(x)$与 $W_j(x)$ /* 事务 T_i 读 x，T_j 写 x，其中 $i \neq j$*/

$W_i(x)$与 $W_j(x)$ /* 事务 T_i 写 x，T_j 写 x，其中 $i \neq j$*/

其他操作是不冲突操作。

不同事务的冲突操作和同一事务的两个操作是不能交换（swap）的。对于 $R_i(x)$与 $W_j(x)$，若改变二者的次序，则事务 T_i 看到的数据库状态就发生了改变，自然会影响到事务 T_i 后面的行为。对于 $W_i(x)$与 $W_j(x)$，改变二者的次序也会影响数据库的状态，x 的值由等于 T_j 的结果变成了等于 T_i 的结果。

一个调度 Sc 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 Sc'，如果 Sc'是串行的，称调度 Sc 为**冲突可串行化**的调度。若一个调度是冲突可串行化，则一定是可串行化的调度。因此可以用这种方法来判断一个调度是否是冲突可串行化的。

［例 11.3］ 今有调度 $Sc_1=r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)\ w_1(B)r_2(B)w_2(B)$

可以把 $w_2(A)$与 $r_1(B)w_1(B)$交换，得到

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

再把 $r_2(A)$与 $r_1(B)w_1(B)$交换

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

Sc_2 等价于一个串行调度 T_1、T_2。所以 Sc_1 为冲突可串行化的调度。

应该指出的是，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

- **冲突可串行化的调度（Conflict Serializable Schedule）**：是指一个并发调度，通过重新排列其中不冲突的操作的顺序（不改变冲突操作的顺序），可以转换成一个串行调度。串行调度是指事务一个接一个地执行，不会出现交错执行的情况。

换句话说，如果一个并发调度可以通过重新排列操作顺序（而不违反事务内部的操作顺序和冲突操作的顺序），变得和某个串行调度等价，那么这个并发调度就是冲突可串行化的。冲突可串行化是判断一个并发调度是否能保证数据库一致性的一个重要标准。

举个例子：

假设有两个事务 T1 和 T2：

```
T1: R(A) W(A)      // T1 读取数据项 A，然后写入数据项 A
T2:       R(B) W(B) // T2 读取数据项 B，然后写入数据项 B
```

一个可能的并发调度 S 可能是这样的：

```
S: R(A) R(B) W(A) W(B)
```

在调度 S 中，T1 和 T2 之间的操作是不冲突的，因为它们操作的是不同的数据项。我们可以交换 T1 和 T2 的操作，得到一个新的调度 S'：

```
S': R(A) W(A) R(B) W(B)
```

调度 S' 是串行的，因为它等同于先执行 T1，再执行 T2。因此，我们可以说调度 S 是冲突可串行化的。这意味着即使 T1 和 T2 并发执行，我们也可以通过调整它们的操作顺序，得到一个等价于串行执行的结果，从而保证数据库的一致性。

两段锁协议

为了保证并发调度的正确性，数据库管理系统的并发控制机制必须提供一定的手段来保证调度是可串行化的。目前数据库管理系统普遍采用两段锁（TwoPhase Locking，简称2PL）协议的方法实现并发调度的可串行性，从而保证调度的正确性。

所谓两段锁协议是指所有事务必须分两个阶段对数据项加锁和解锁。

- 在对任何数据进行读、写操作之前，首先要申请并获得对该数据的封锁；
- 在释放一个封锁之后，事务不再申请和获得任何其他封锁。

所谓“两段”锁的含义是，事务分为两个阶段，第一阶段是获得封锁，也称为扩展阶段，在这个阶段，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，在这个阶段，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。

例如，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　→|

又如，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；

可以证明，若并发执行的所有事务均遵守两段锁协议，则对这些事务的任何并发调度策略都是可串行化的。

例如，图 11.8 所示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。可以验证如下：忽略图中的加锁操

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

图 11.8　遵守两段锁协议的可串行化调度