8.1 OLTP与OLAP

两种应用模式刻画应用对数据库管理系统的需求

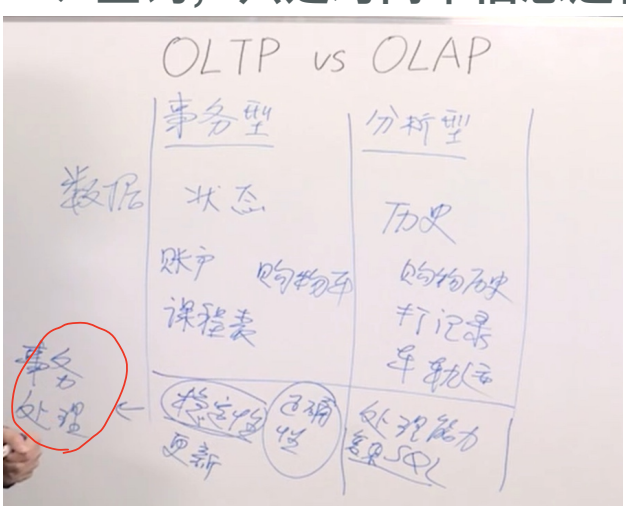
现实应用分成两大类 事务型应用和分析型应用

状态数据 保存现实世界的状态信息，比如银行账户，电商平台购物车 课程表 围绕状态数据构建的应用是事务型应用，帮助完成现实生活中的交易和管理，数据在跟踪现实世界的状态

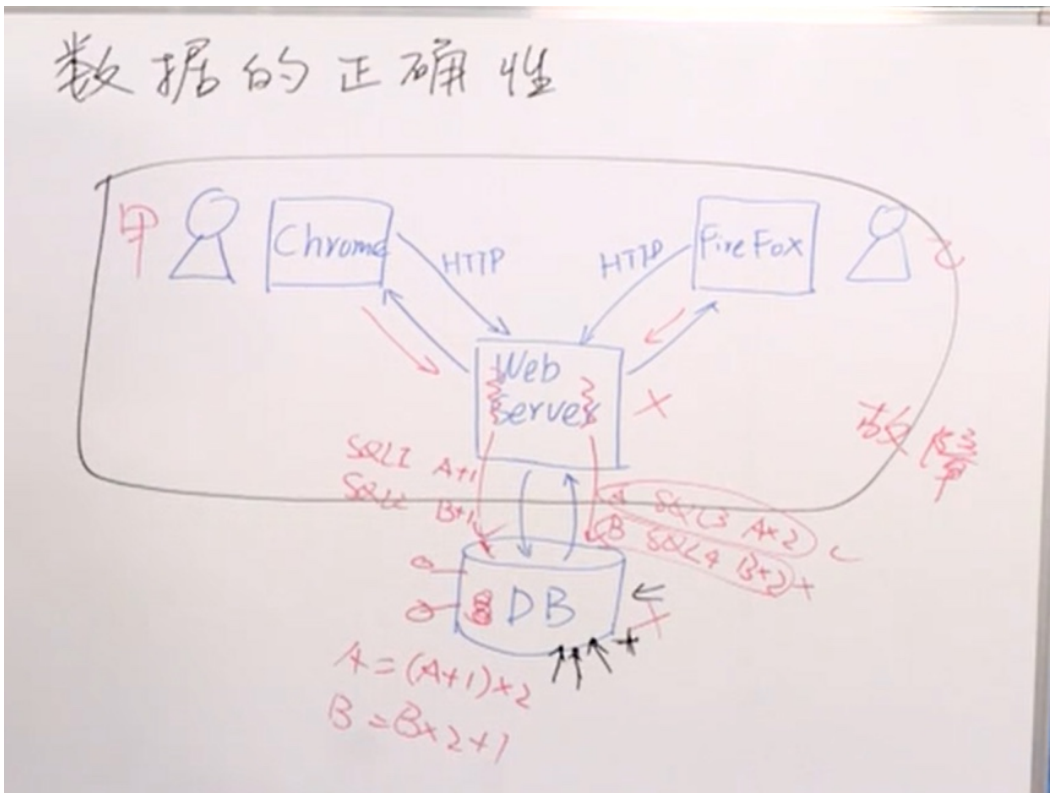
历史数据 刻画的是曾经发生过什么事的记录 比如购物历史 一年的消费清单 打车记录 车的轨迹 围绕历史数据构建的应用是分析型应用，从历史数据挖掘新的信息

分析型应用需要强大的数据处理能力，从大量数据获取规律和信息，复杂的Sql查询，对查询的应答速度要求高

事务型应用要求数据跟踪并且确保现实的状态是正确的，对数据库稳定性要求高，需要数据一直是正确的，对数据正确性要求高，数据库增删改查很多，要求快速进行，不会有复杂的SQL查询，只是对简单信息进行抽取



8.2 数据的正确性问题

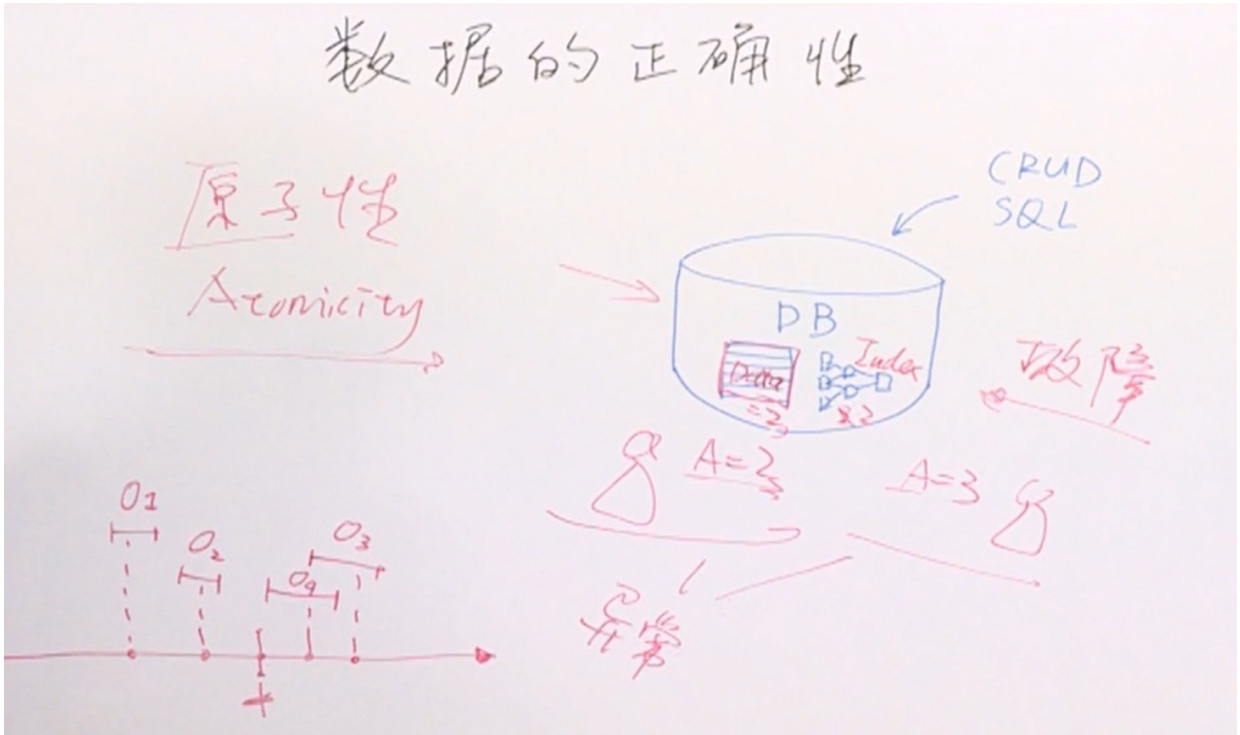


8.3 数据库操作的原子性

对数据库的访问操作需要满足原子性

原子性无论用户向数据库提交了什么操作，这个请求必须全部执行，要么当作没有发生过，一点都不执行。原子是不可分的一个颗粒，我们要求在数据库上执行的任何一个操作都是不可分的，要么完全执行要么不执行

如果操作在一个瞬间就能被完成，就不会担心故障了，并且两个用户提交的操作总是有一个先后次序，不是同时发生的，一个操作发生之后，另一个操作才开始，所以不会出现干扰



8.4 日志机制

如何实现数据操作的原子性 中断之后遗留的不正确的数据应该怎么解决

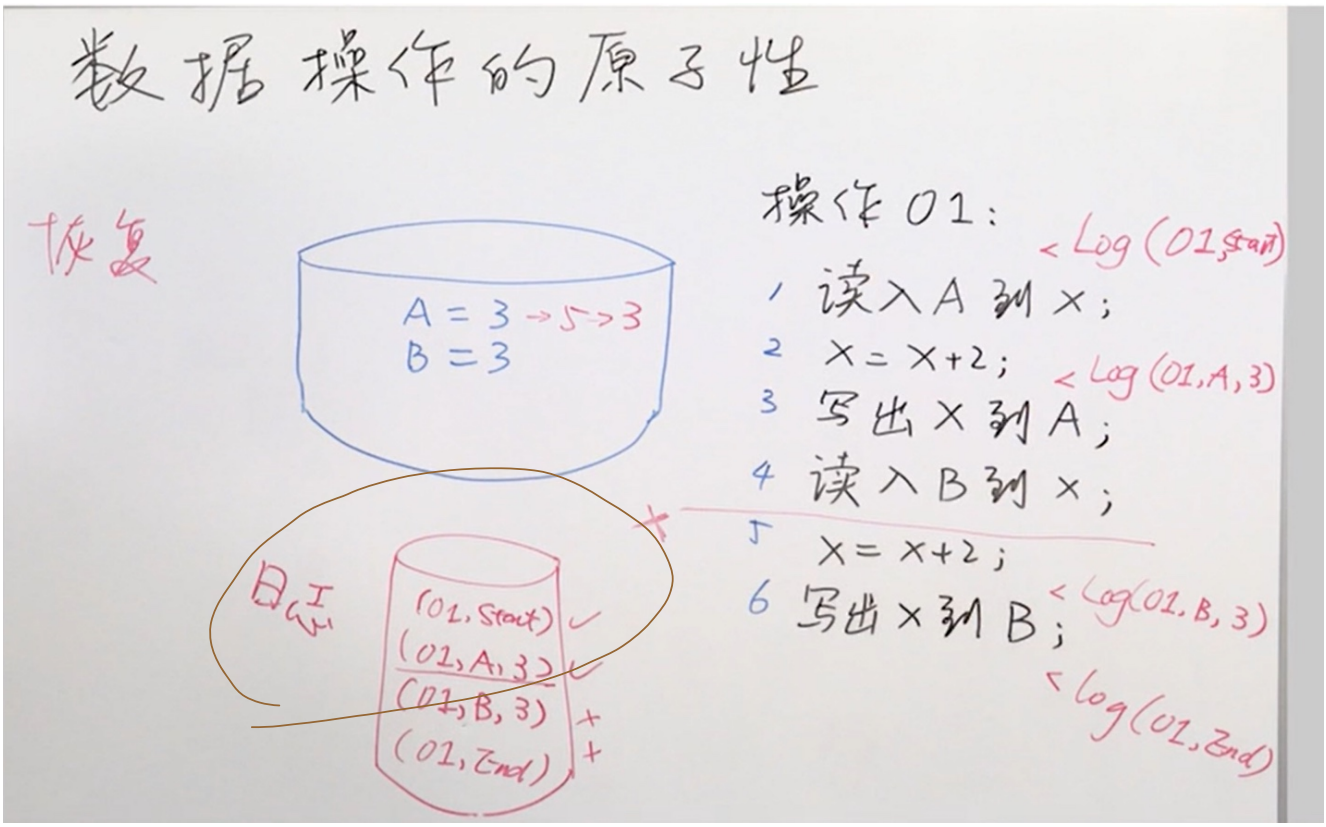
在硬盘上单独留出来一块记录日志，通过日志恢复数据库，故障发生的时候通过日志记录查看哪些操作实际上没完成，然后把这些操作回滚回去，把已经修改的操作恢复成原来的值

日志怎么记录呢?

当操作开始的时候，先不操作，先写一个日志，当我发现要修改硬盘了，我再写一个日志，记录 a 的原始值是3，然后再执行第三步。同样，要对b的值进行修改之前，在日志上记录b的原始值是3，再进行对b的修改操作。整个操作完成以后，再往日

志里添加记录代表01结束了。当把日志记录结束以后，操作才算完成。if第四第五步之间出现故障，日志里目前有前两条记录，这时重新启动系统，系统进入恢复的过程，会检查记录的日志，会发现01操作开始了，并且修改了 a，但是01操作并没有结束，所以要把系统恢复到01 开始之间的状态，重新把 a 的原始值3赋值回去，这时候你会发现 a 和b都是正确的状态了。

日志 帮助把某一个没有完成的操作恢复成操作开始之前的状态



8.5 undo 和 redo日志

上一节讲的是undo日志，主要功能是系统故障之后把执行一半的操作回滚回去，把系统状态恢复到操作发生之前

Redo日志的工作

操作开始之前，插入日志代表操作开始了。要修改之前记录日志，但不是记录原始值，记录修改之后的值，记录 a 现在被改成了5，然后不急着去修改 a 的值，把它推迟到操作最后（第七步），下一步修改b，同样记录b的日志，记录b被修改成5（修改后）。同样把这步推迟到最后（第八步）。所以我把对硬盘数据的修改都推迟到最后，延迟它的执行。在真正对硬盘上的数据修改之前，再写一条日志记录操作结束01end，记录完以后再把修改过的值写回硬盘上。

Redo日志怎么应对故障?

假设故障依旧发生在第四第五步之间，日志里第一第二条存在，第三第四条不存在，查阅日志发现01操作开始了但是没有结束，对于redo日志而言它什么都不需要做。因为它知道在end记录出现之前，数值都没有被写回硬盘上，硬盘里的数据还是原来的值，没有问题。

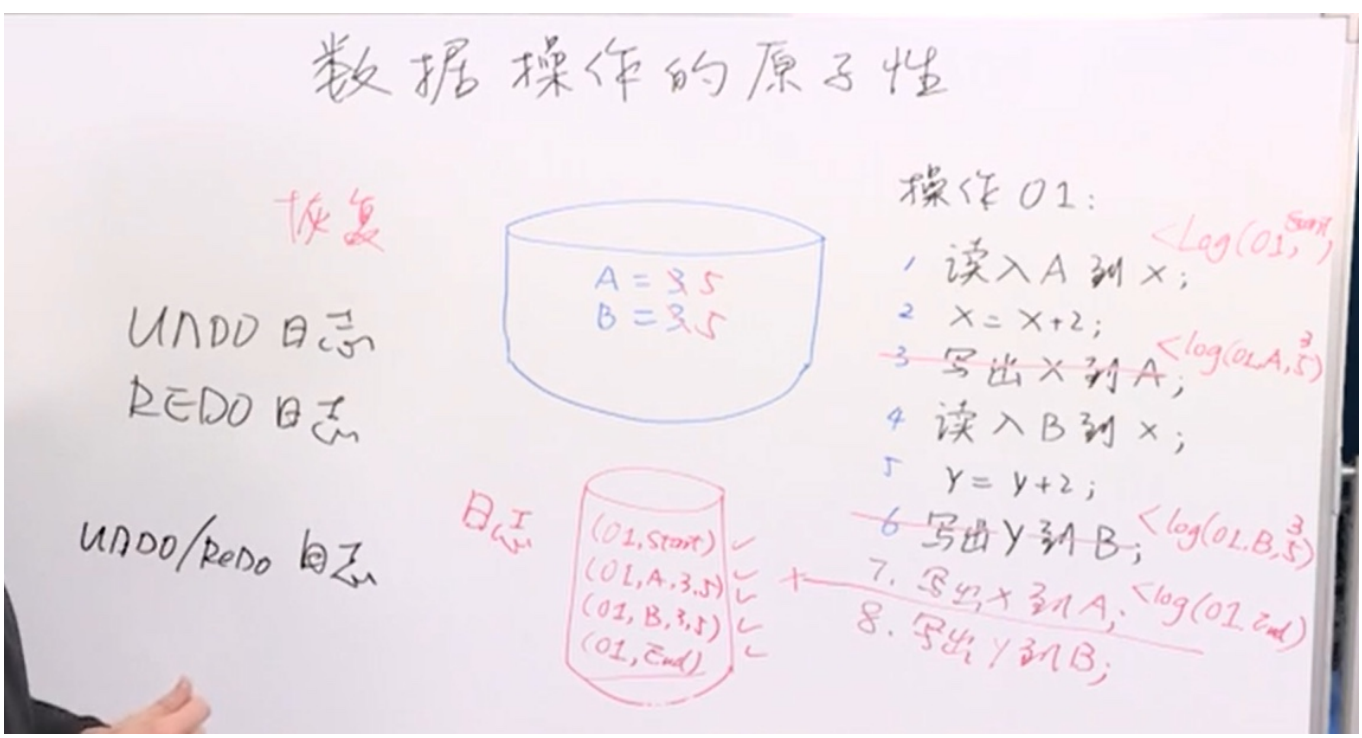
假设故障出现在第七第八步之间，此时 a 在硬盘上的值已经变成5，但是b还是3。此时系统发现四条日志记录全部存在，系

统会认为01操作已经结束，b的值应该是5但是硬盘上的值仍然是3，所以他就把硬盘上的b值改成5。对于故障，它会在日志里面查这个操作是不是结束了，如果操作已经结束了，则它所修改过的值已经记录在日志里米，它就能把硬盘上的值恢复成操作结束之后的状态。所以redo日志是把数据状态变成操作结束之后的状态

两种日志工作模式的优缺点
都可以保证操作的原子性
Redo日志
要求操作结束之前，所有的日志必须落到硬盘上。只有等所有的日志都落到硬盘上之后，才可以对数据进行修改。
有可能遇到的问题
对数据在内存里已经修改了，而且内存已经装不下了，这个时候操作还没结束，操作因为内存不够而进行不下去，新的修改加不进去了，所以对内存的消耗比较大。
优点是效率高，因为对数据的修改可以推迟到操作结束之后，不需要跟操作同步进行，只需要在适当的时候对日志进行记录就可以了（因为日志是往硬盘上一条一条顺序追加写入的，所以它的效率高速度快）
Undo日志相反，对内存消耗小，因为每修改一个数据，你就可以把它放硬盘上，内存里面就不需要再保存这个数据了。但是

它写日志的过程是和写数据的过程交错进行。你写一个日志然后马上写一个数据，然后再写一个数据再写一个日志再写一个数据，访问硬盘的过程不是顺序的是跳来跳去的，速度相对比较慢。

所以我们真正在数据库系统里使用的是undo/redo日志，它每一条记录不但包含数据修改之前的值，还包含数据修改之后的值，都进行记录。可以规避两个模式的问题，一旦发生故障，我们有两种选择。如果在日志里发现操作已经完整，我们就可以进行redo，把操作视为是完成的，把数据变成完成之后的状态。如果发现日志并不完整，可以做undo，把数据恢复成操作开始之前的状态。我有两种选择，什么时候修改数据什么时候记录日志，没有很严苛的限制



第 1 题(本题2分)：是系统使用了undo日志，在故障发生后，发现日志记录如下：<o1,start>, <o2,start>, <o2,A=5>, <o2,B=4>, <o2,end>, <o1,A=3>, <o1,C=3>。请问：系统恢复后A的取值是多少?

- A：5
- B：4
- C：3
- D：不知道

题干字体大小 小 中 大

第 2 题(本题2分)：我们用log表示将日志写到硬盘，用write表示把数据写到硬盘。如果系统使用undo日志，那么以下哪个操作执行序列是不能保证原子性的?

- A：log(o1,start), log(o1,A=5), write(A=6), log(o1,B=3), write(B=4), log(o1,end)
- B：log(o1,start), log(o1,A=5), write(A=6), write(B=4), log(o1,B=3), log(o1,end)
- C：log(o1,start), log(o1,A=5), log(o1,B=3), write(A=6), write(B=4), log(o1,end)
- D：log(o1,start), log(o1,A=5), log(o1,B=3), write(B=4), write(A=6), log(o1,end)

第 3 题(本题2分)：我们用log表示将日志写到硬盘，用write表示把数据写到硬盘。如果系统使用redo日志，那么以下哪个操作执行序列是不正确或不可能发生的?

- A：log(o1,start), log(o1,A=5), log(o1,end), write(A=5), log(o2,start), log(o2,A=6), log(o2,end), write(A=6)
- B：log(o1,start), log(o1,A=5), log(o1,end), log(o2,start), log(o2,A=6), write(A=5), log(o2,end), write(A=6)
- C：log(o1,start), log(o1,A=5), log(o1,end), log(o2,start), log(o2,A=6), log(o2,end), write(A=6)
- D：log(o1,start), log(o2,start), log(o2,A=6), log(o2,end), log(o1,A=5), log(o1,end), write(A=5), write(A=6)

第 4 题(本题2分)：以下哪种情况是数据库系统的操作原子性（atomicity）无法保证的?

- A：两个并发的CRUD操作从效果上一定一个在前一个在后。
- B：任意一个CRUD操作要么发生在故障之前，要么发生在故障之后。
- C：在故障发生前已经开始的CRUD操作一定会顺利完成。
- D：在故障发生前没有结束的CRUD操作一定会被撤销。

8.6 并发控制（上）

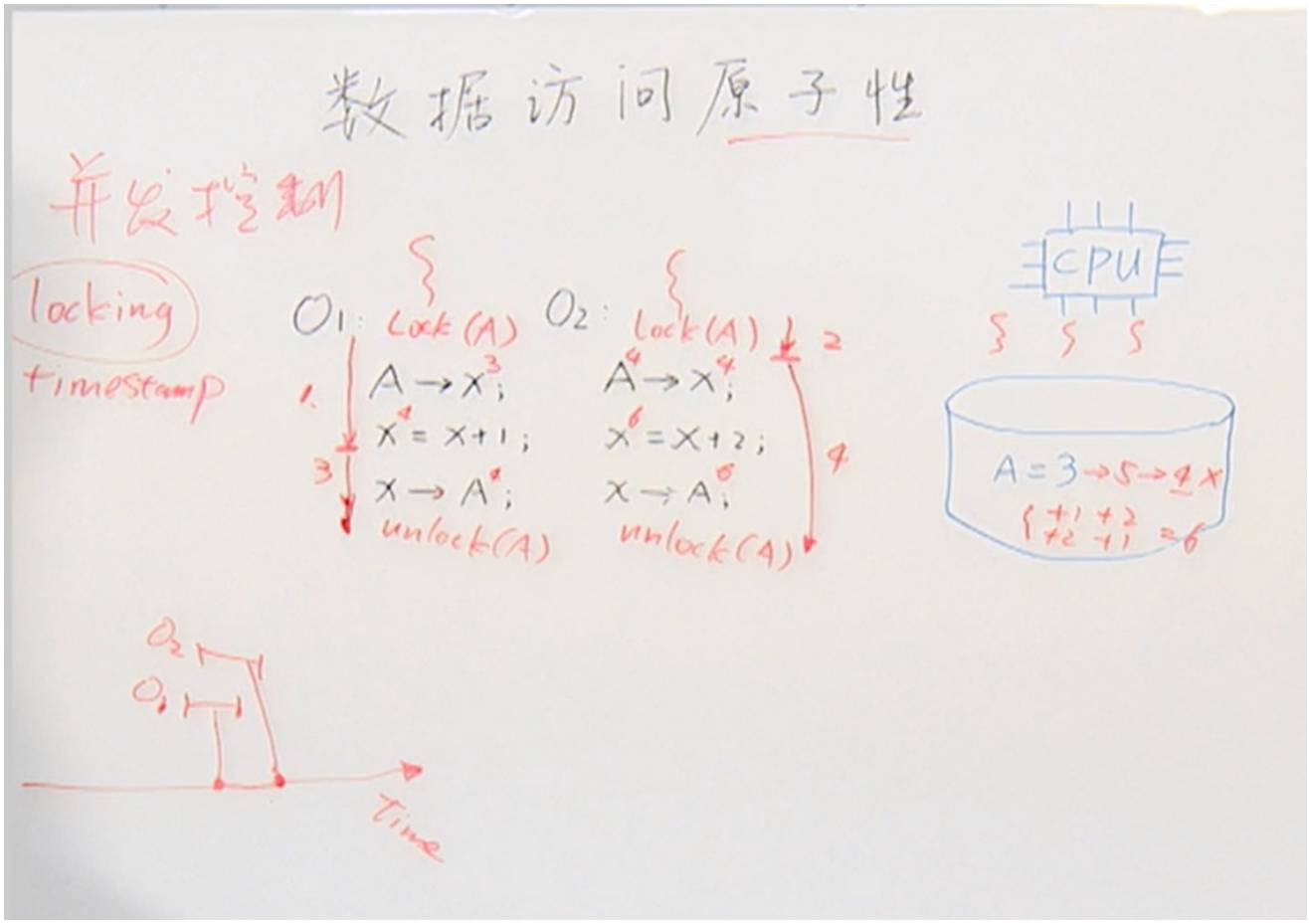
保证原子性 01 和02操作都需要执行完毕，且必须一个先一个后

需要一套机制保护操作之间不会相互干扰

实现方式有加锁 locking 时间戳 timestamp

对a进行访问之前，先把a锁住，完成对a的访问之后再把锁释放掉，这样01和02 之间就不会相互干扰了

还是那个例子 01执行一半后去02，02要加锁的时候发现a已经被锁住了，所以只能停在那里等待，因为它发现那把锁已经被 01 拿住了。锁是互斥的，只有一个线程可以拿到这把锁。cpu看到02等待了，又会回到01，01拿到锁继续执行，x变成4然后解锁，解锁之后01 结束。02 看到锁被解除掉以后才能拿到那把锁，做接下来的步骤。02 开始做的时候01已经结束了，所以x值是4，最后赋给a的值是6。有了锁我们可以保证不同操作对数据的访问是互斥的，他们必须排着顺序做



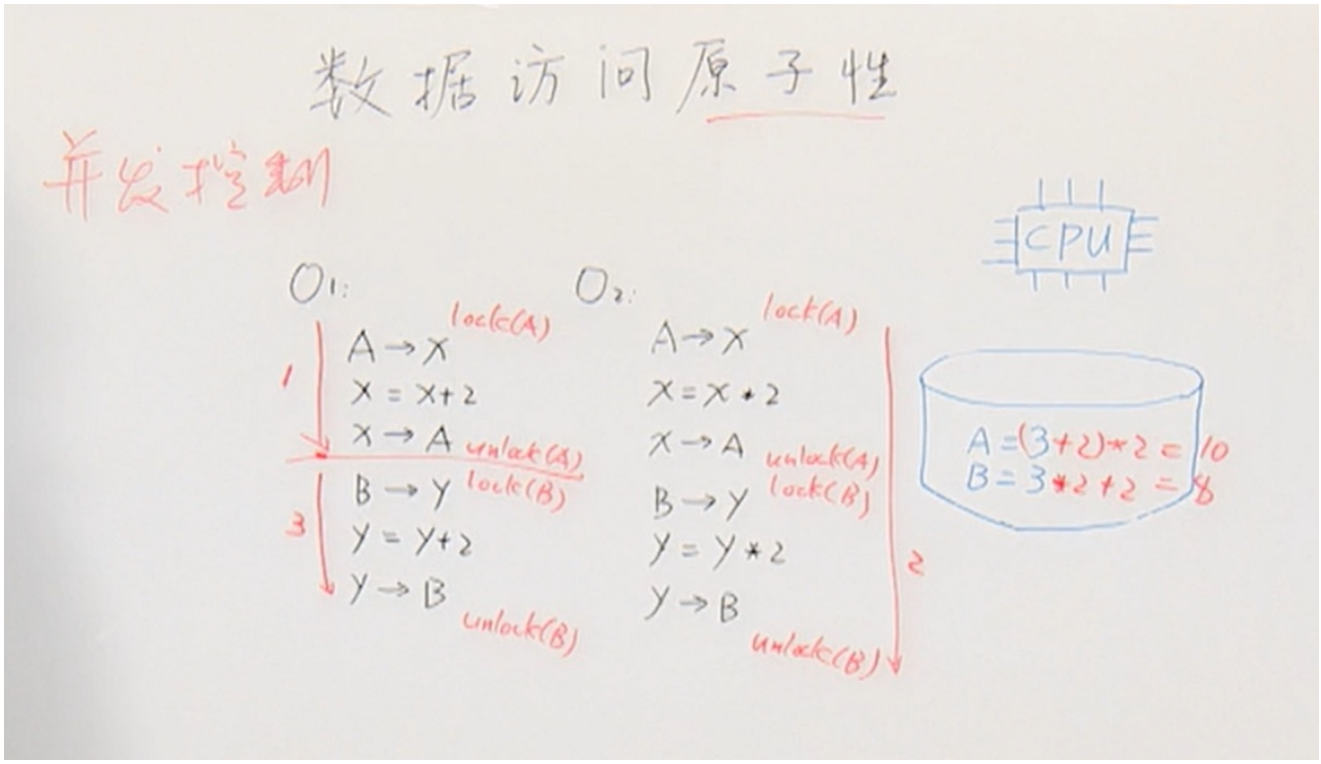
8.6 并发控制（下）

一种方法：访问a（b）之前上锁，访问结束之后解锁，但是有问题。比如01执行一半，cpu被调度去执行02并且把02执行完毕，之后再回去执行01。

对于a，先+2再x2，是10

对于b，先x2再+2，是8

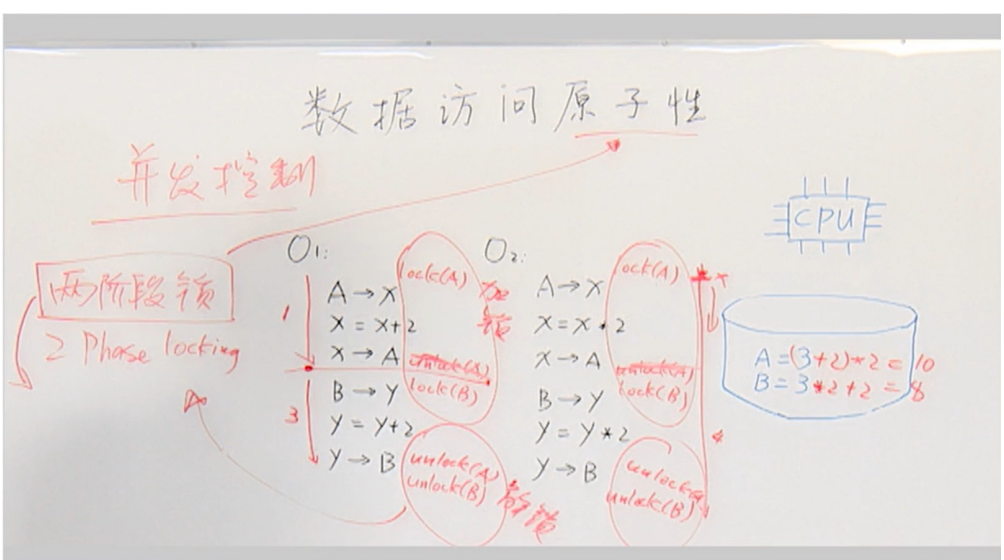
会相互干扰，最后 a 和b的取值不一样



所以上面那个加锁的次序不对。

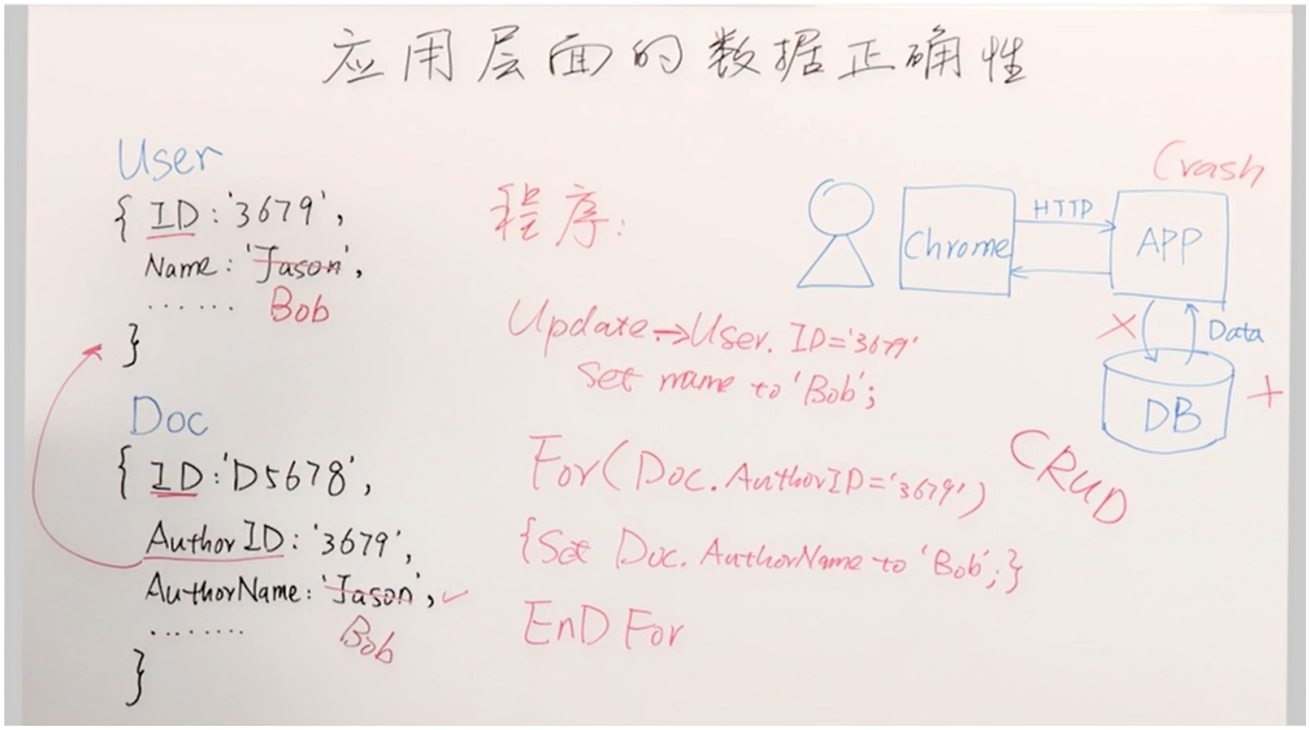
并发控制加锁的规则：两阶段锁，当你在加最后一把锁之前，你不能释放任何一把锁，01 加了a的锁也加了b的锁，把加锁和解锁看成是两个阶段，这两个阶段是完全的先后关系

延迟释放锁，保证01和02互不干扰



8.7 应用层面的数据正确性

更新用户名的一段程序



对应用而言，它要保证整个程序的运行是正确且完整的，需要一些机制去实现数据正确性的保护。

8.8 用标志位防止数据异常

一段程序运行的中途可能会被中断掉，因此出现数据不一致

方法一：程序中断以后记得自己在哪里中断了，重启的时候接着做没做完的

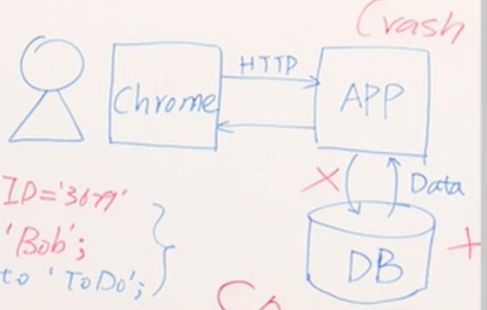
更新完名字还没来得及更新文章的作者名 - todo

全部更新完了 - done

应用层面的数据正确性

User
{ ID: '3679',
Name: 'Jashon', 'Bob'
Namesync: 'Done', ToDo Done
}

Doc
{ ID: 'D5678',
AuthorID: '3679',
AuthorName: 'Jashon', Bob
.....
}

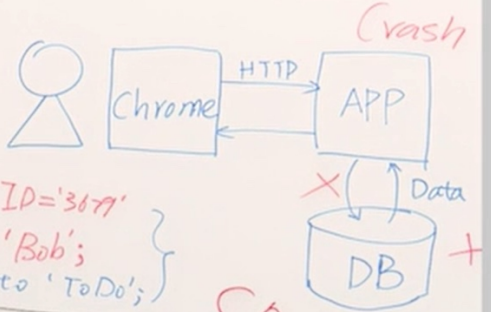


Update→User. ID='3679'
Set name to 'Bob';
set namesync to 'ToDo';
For(Doc.AuthorID='3679')
{Set Doc.AuthorName to 'Bob';}
EnD For
Update→User. ID='3679'
Set namesync to 'Done';

应用层面的数据正确性

User
{ ID: '3679',
Name: 'Jashon', 'Bob'
Namesync: 'Done', ToDo Done
}

Doc
{ ID: 'D5678',
AuthorID: '3679',
AuthorName: 'Jashon', Bob
.....
}



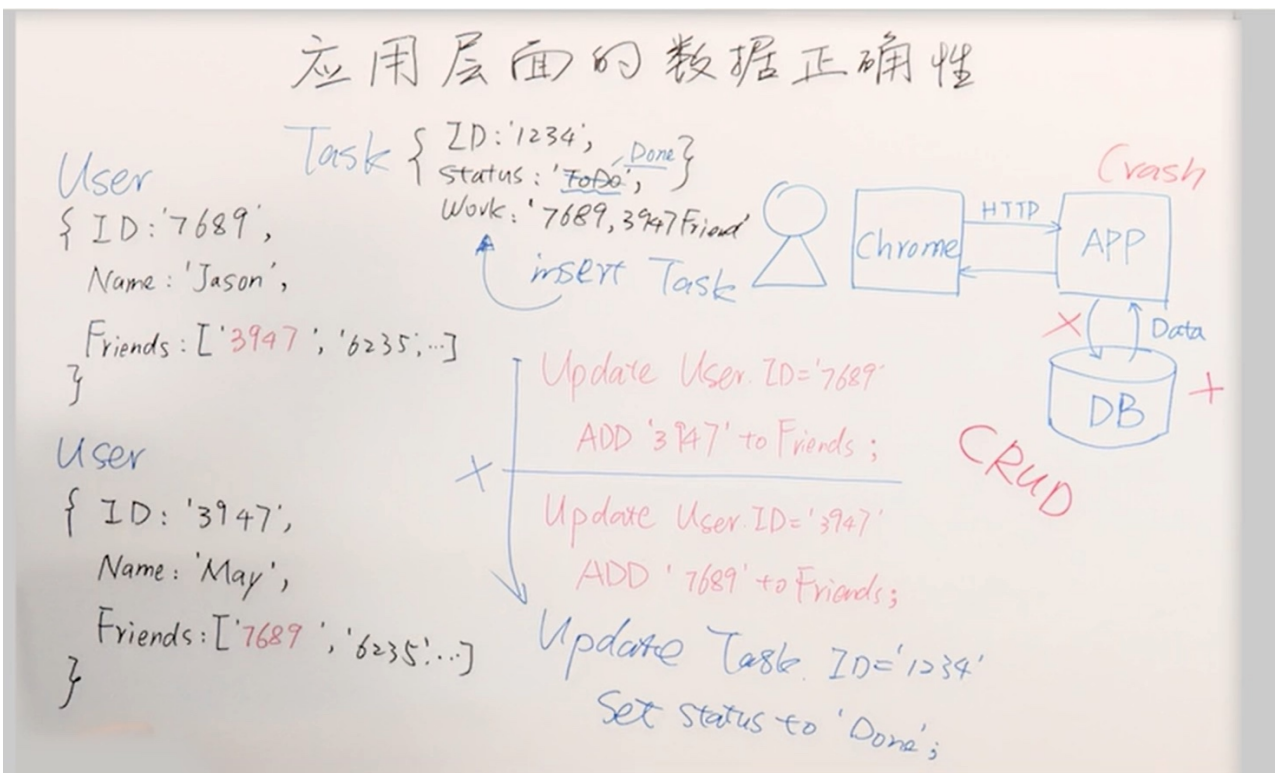
Update→User. ID='3679'
Set name to 'Bob';
set namesync to 'ToDo';
For(Doc.AuthorID='3679')
{Set Doc.AuthorName to 'Bob';}
EnD For
Update→User. ID='3679'
Set namesync to 'Done';

重做一遍

8.9 用消息队列防止数据异常

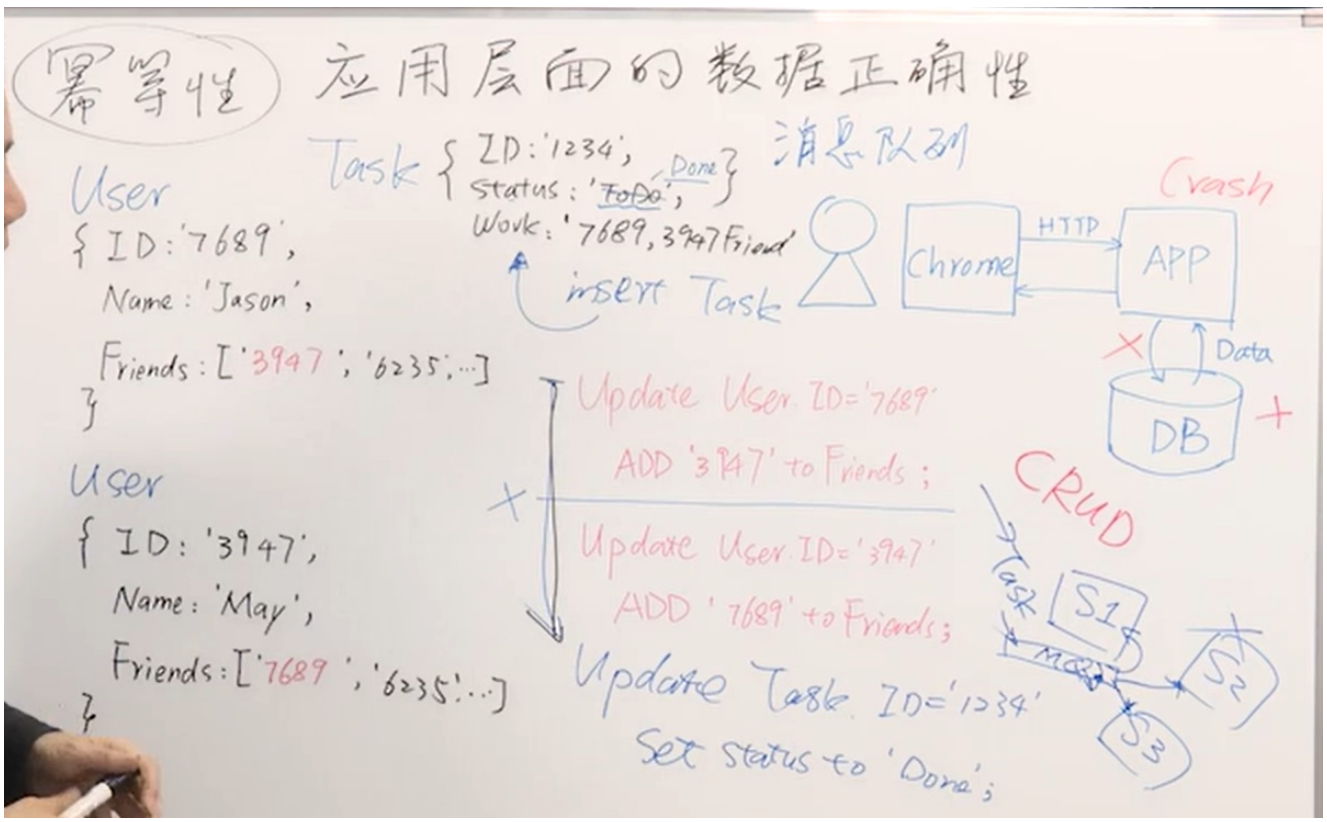
创建一个新的文档task

完成加好友之前先增加task文档，status是todo，完成之后



update task文档的状态到done，再宣告加好友的过程完全结束。

这个方法叫消息队列，要求幂等性，你要完成的程序必须满足幂等性，指这个程序完成很多遍或者做一部分的结果都是一样的。它保证你的程序即使执行了一半又重新开始执行，也不会出现意外的结果。你发现一个任务没做完，其实你不清楚哪一部分没做完，但你也可以把整个任务重新做一遍



第 1 题(本题2分)：以下哪种加锁方式可以保证操作的原子性?

- A：lock(A); update(A); lock(B); update(B); unlock(A); unlock(B);
- B：lock(A); lock(B); update(A); update(B); unlock(A); unlock(B);
- C：lock(A); update(A); lock(B); unlock(A); update(B); unlock(B);
- D：都可以

在 lock(B) 之前不 unlock(A) 即可

第 2 题(本题2分)：日志和锁需要配合起来使用才能完整确保操作的原子性。用log表示将日志写到硬盘，用write表示把数据写到硬盘，用lock和unlock表示加锁与解锁。如果系统使用undo日志，那么以下哪个执行序列是日志和锁的最合理搭配方式？

end之后再unlock

先start再lock

A：log(o1,start); log(o1,A=5); lock(A); write(A=6); unlock(A); log(o1,end);

B：log(o1,start); lock(A); log(o1,A=5); write(A=6); log(o1,end); unlock(A);

C：log(o1,start); log(o1,A=5); lock(A); write(A=6); log(o1,end); unlock(A);

D：lock(A); log(o1,start); log(o1,A=5); write(A=6); log(o1,end); unlock(A);

第 3 题(本题2分)：以下哪个操作不是幂等（idempotent）的?

A：向一个集合里插入一个元素

B：从一个集合里删除一个元素

C：对一个数据进行赋值：X=10

D：对一个数据进行自增：X=X+1

做很多次一次一直加。

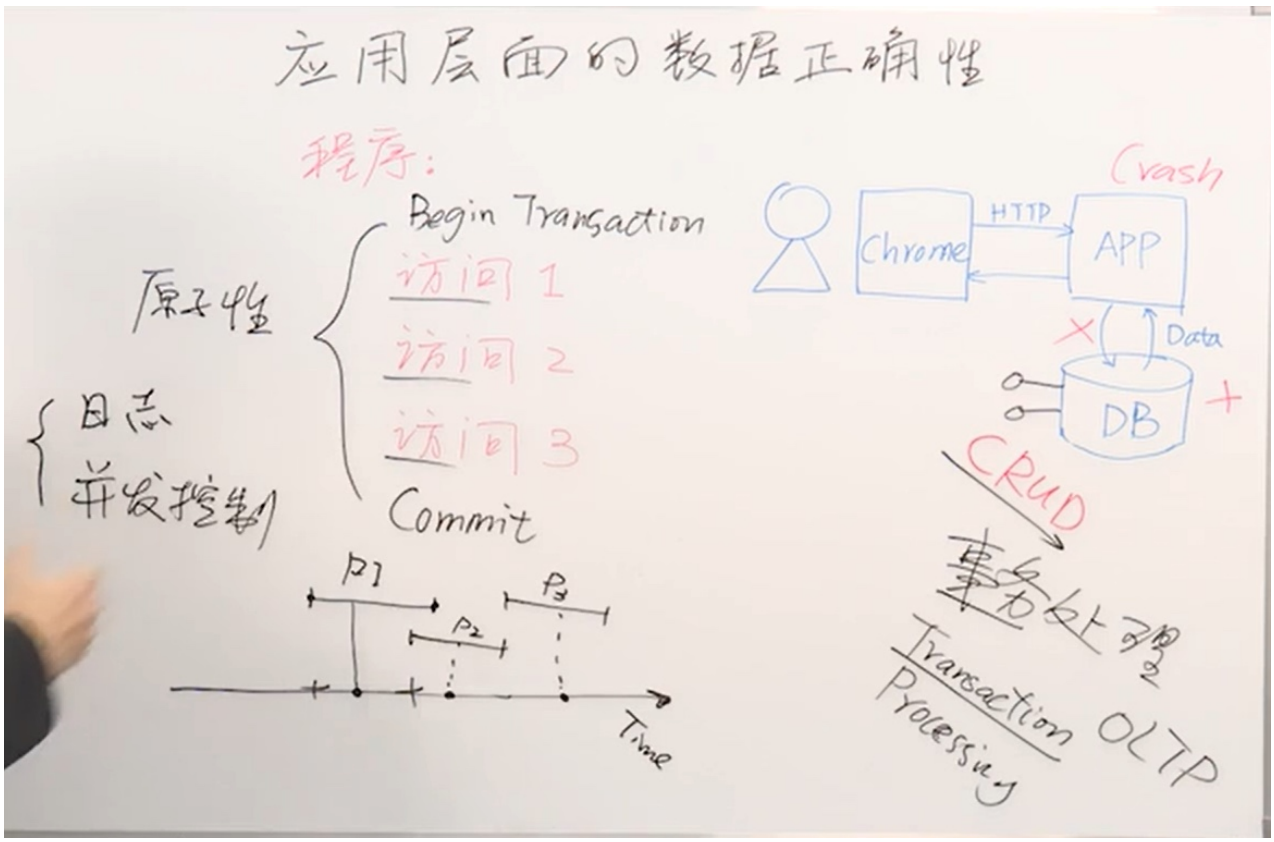
8.10 事务处理的概念

把保护数据一致性的任务交给数据库

事务处理功能 数据库上的额外的功能帮助程序帮助程序保护数据的一致性和正确性

对于数据库而言 一个事务就是一段程序，数据库保护一段程序的原子性。把所有数据访问的操作打包在一起看成是一个操作，用同样的机制，like日志 并发控制，保证整个事务的原子

性。一段程序执行的效果就像在一个瞬间被完成一样。告诉数据库，事务什么时候开始什么时候结束。把数据保护的责任交给数据库完成。



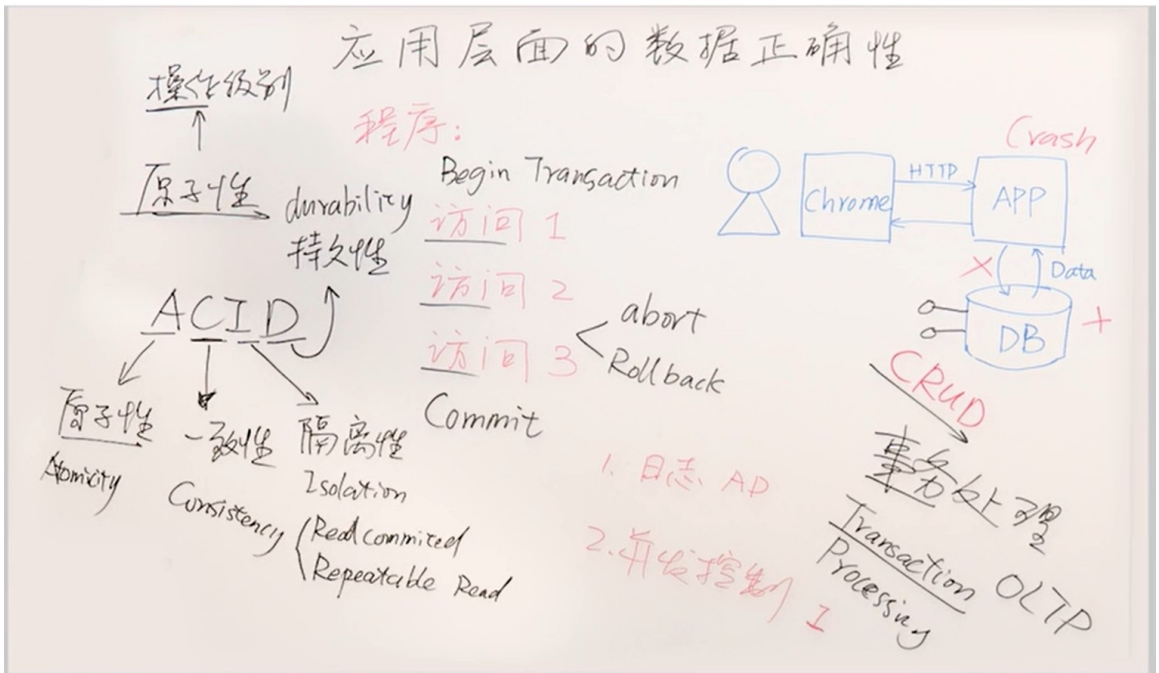
8.11 事务的功能与性质

调用abort或者roll back，事务也算结束了，但是它结束以后就

像什么都没有发生过一样，回到事务开始之前的状态。

对数据库而言，保证事务是正确的，达不到操作级别的原子性，我们把事务的属性掰开成ACID四种属性。
A 也叫原子性 但是事务级别的原子性和操作级别的原子性不是一个概念。单指事务是一个整体，要么做了要么没做过，不会有中间做了一半的状态。不是指事务发生是一个瞬间状态
C 一致性 事务处理的程序保证数据是正确的，由程序的逻辑保护，数据库一般不会特意保证一致性
I 隔离性 多个事务执行的时候不要相互干扰，一般会分成若干个层次。操作级别的原子性隔离是非常彻底的，完全是一个先一个后。但是数据库中事务的隔离有一些更模糊的层次
D 持久性 一旦事务提交以后就铁板钉钉了，不能反悔了，无法撤销了。

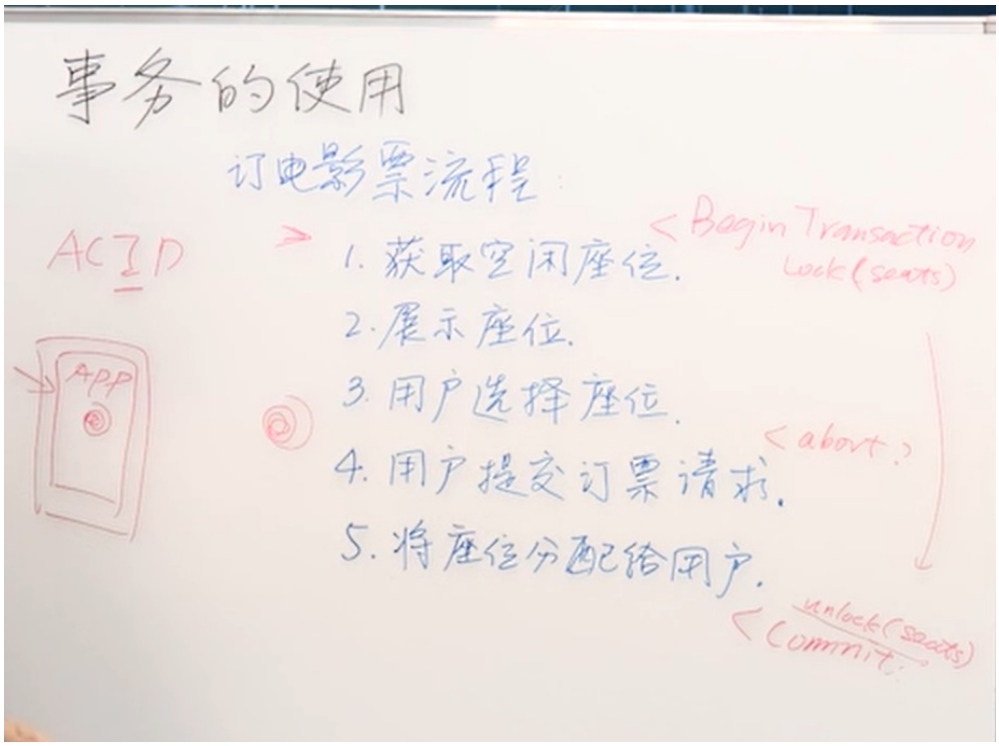
数据库提供事务处理的功能，提供接口，就应该保证ACID四种性质。C是数据库和程序配合保证的，AID是数据库一定要有能力自己保证
数据库使用日志保证事务的原子性和持久性，并发控制保证事务的隔离性。数据库提供事务处理功能是使用日志和并发控制机制



8.12 合理使用事务

第一个用户释放锁以后其他用户才能走下去。一个用户的迟疑影响所有用户订票的过程。

把整个过程包裹在一个事务里面：可以保证acid的性质，不会出现多个用户订到一个座位的情况。问题：为了保证事务的隔离性，多个事务之间不能相互干扰，就需要对数据进行加锁。也就是说事务一开始访问空闲座位的时候，会把座位锁上。等他提交事务，整个订票结束以后，才会解锁座位信息。一旦最快的用户开始事务，其他订票的用户会被卡住，获取空闲座位会出问题。如果一个用户选座位选了很久，其他用户都得等着。



另一种事务处理的方式（黑色）

在用户选择完座位以后再启动事务，判断座位是否还空闲，如果空闲，分配给用户，再结束事务。如果不空闲，则abort，返回订票错误。

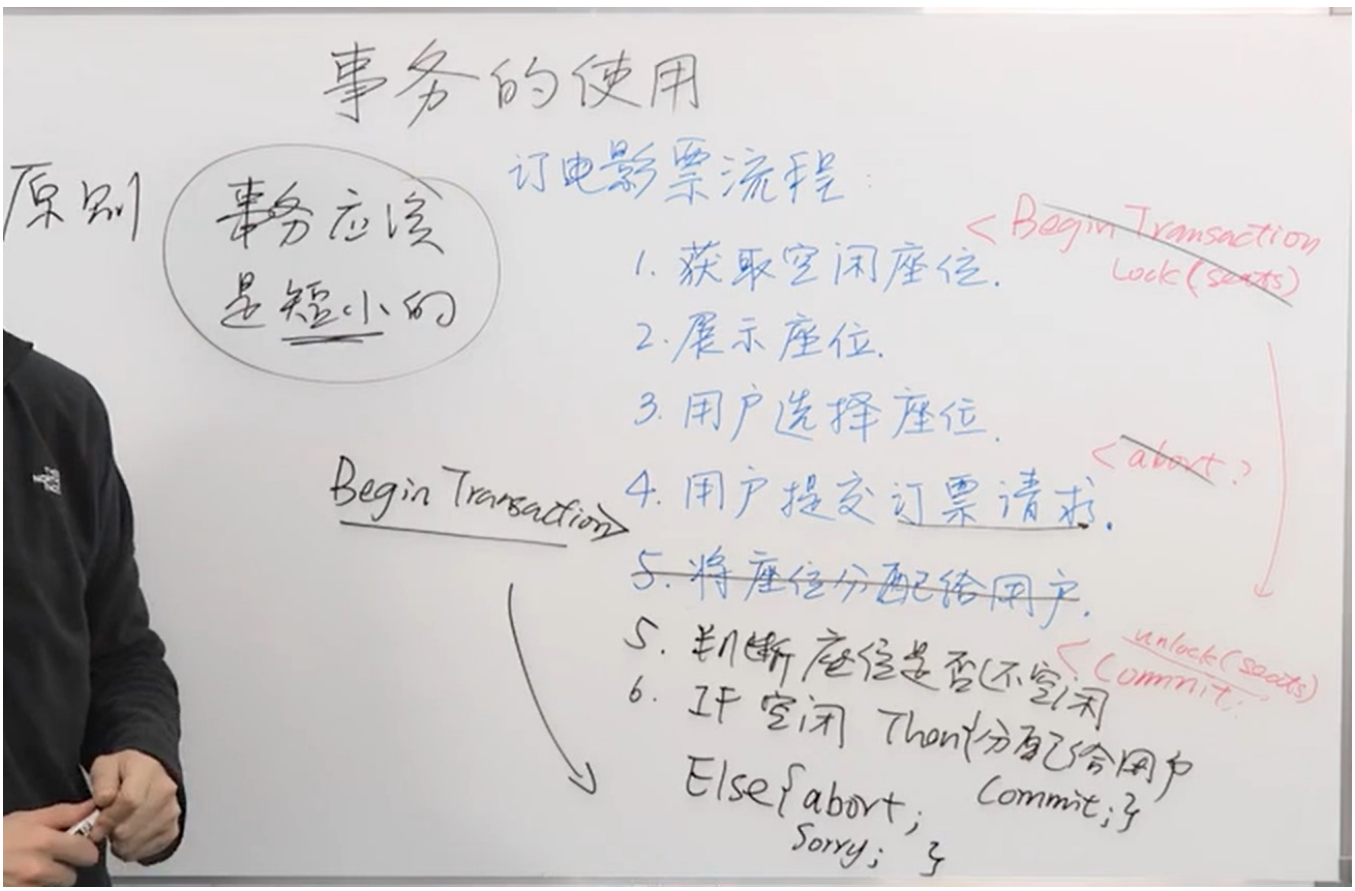
用户浏览座位，与应用进行交互的过程被排除在事务处理过程外面，不会出现由于一个用户的迟疑而出现阻塞的现象。这个方法从事务开始到结束都没有用户的交互，没有跟用户进行沟通，很快，效率很高，但是订票有更高概率不成功。

使用事务的原则

事务应该是短小的！！

不能把所有业务逻辑包裹在事务里面，事务一旦过长，尤其涉及到人机交互的时候（延迟不可控），会容易出现拥堵，因为

事务开始的时候，所有被访问的数据通常会被加锁。一旦数据被加锁，其他事务对数据的修改则无法进行下去。我们应尽量把人机交互或者应用间通信的过程排除在事务之外。



8.13 合理使用事务（续）

第一步和第三步密切相关，但是第二步跟付款没有直接关系，一旦把三步都放在事务里面，你不得不把第二步也放在事务里面，这样无辜增加了事务的长度。

所以，我们把第三步变成事务的第二步，把原本的第二步放在第二步之后执行（也就是事务完成之后再调用商品运输的服

务），事务由此变得更短小，效率更高。
事务的考试和结束需要能起到数据保护的作用，也不能过长。
也需要看看业务逻辑的顺序，必要的时候调换以缩短事务，使事务处理的执行效率更高