MVCC与实例恢复

2018.01

唐成（网名osdba）

《PostgreSQL修炼之道：从小工到专家》的作者，PostgreSQL中国用户会常务委员，中国开源软件推进联盟PostgreSQL分会特聘专家。从业近20年，拥有十几年数据库、操作系统、存储领域的工作经验，历任过阿里巴巴高级数据库专家，从事过阿里巴巴Greenplum、PostgreSQL、 MySQL数据库的架构设计和运维。

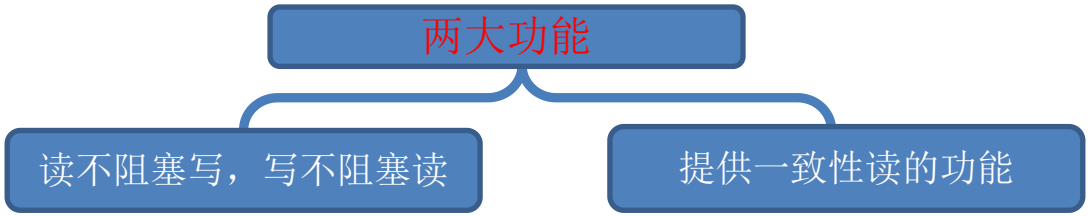
既熟悉数据库的运维工作，是最早的Oracle 9i的OCP，又懂开发，精通Java和python，擅长使用Java和python写数据迁移任务。

目录

什么是MVCC?

借助wiki上的解释：

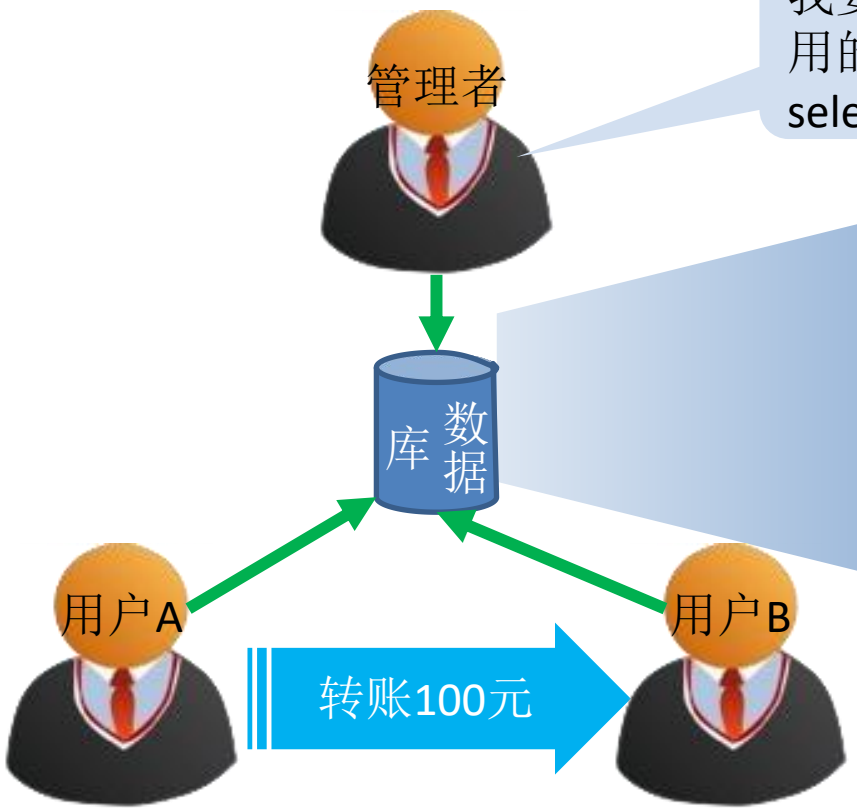
Multiversion concurrency control (MCC or MVCC), is a **concurrency control method commonly** used by database management systems to provide **concurrent access to the database** and in programming languages to implement transactional memory.



产生的原因

- 在并发操作中，当正在写时，如果有用户在读，这时写可能只写了一半，如一行的前半部分刚写入，后半部分还没有写入，这时读取到的数据行，可能是前半部分是新数据，后半部分是旧数据的不一致的行。解决这个问题的最简单的办法是使用读写锁，写的时候，不允许读，正在读的时候也不允许写，但这种方法导致读和写不能并发。于是，有人就想到了一种能够让读写并发的方法，这种方法就是MVCC。MVCC的方法是写数据时，旧的版本数据并不删除，并发的读还能读到旧的版本数据，这样就不会有问题了

为什么需要MVCC:考虑一个转账场景



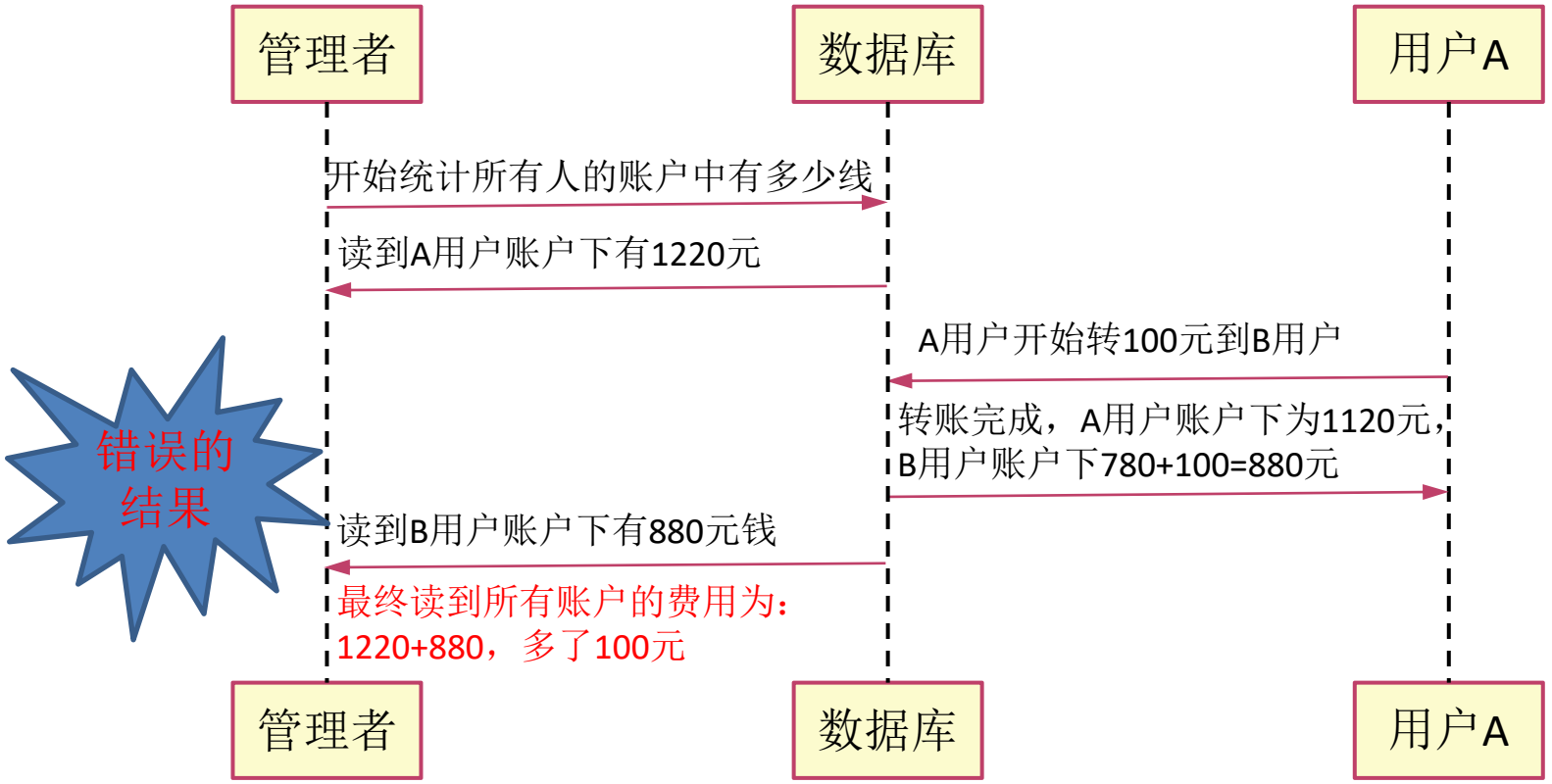
账户表T

户名	账户费用	行号
…	…	…
用户A	1220	2000
…	…	
用户B	780	6000
…	…	…

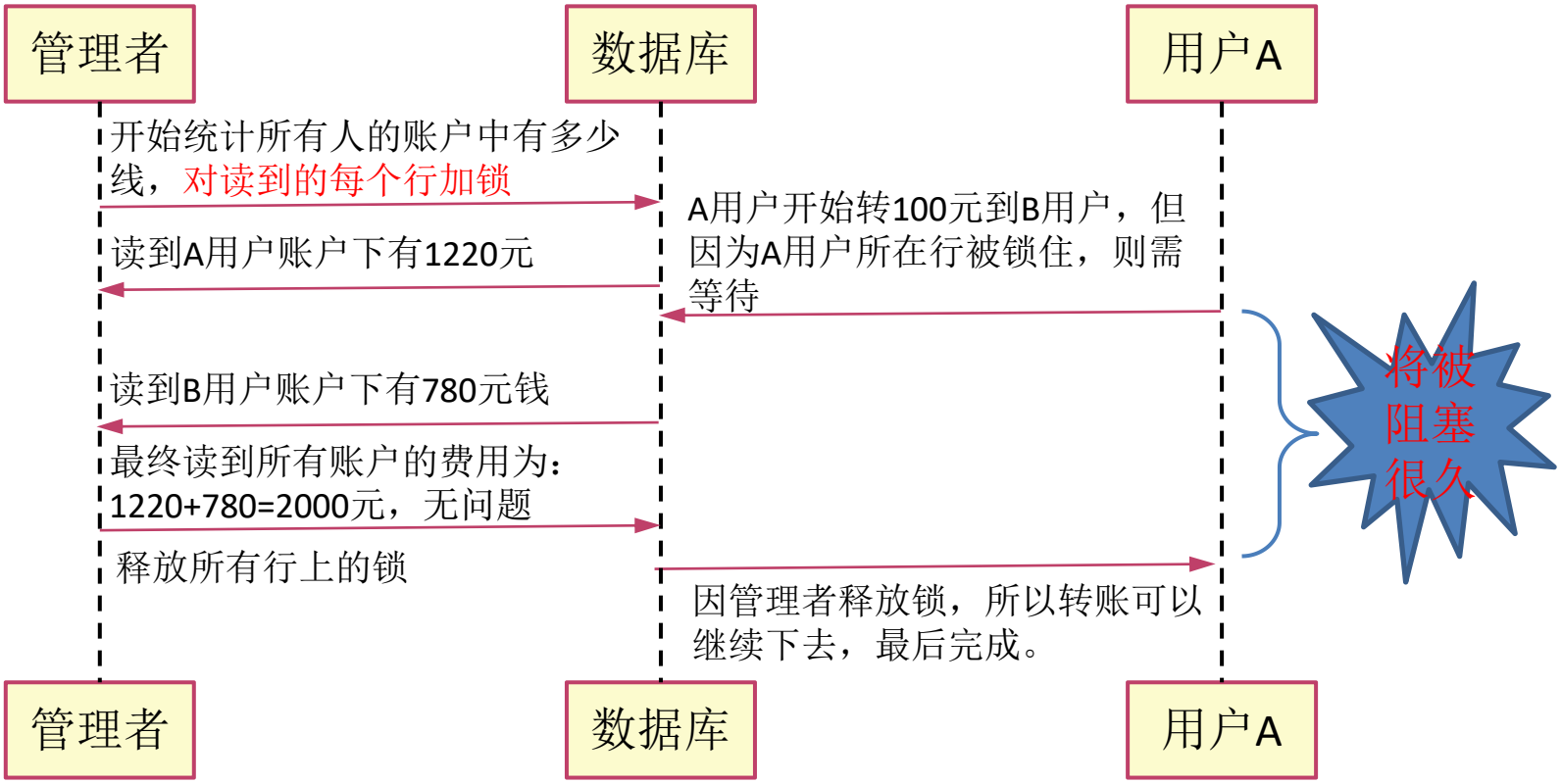
1万行

假设除A用户和B用户外，其它账户费用都为0，所以所有账户合计费用为1220+780=2000元

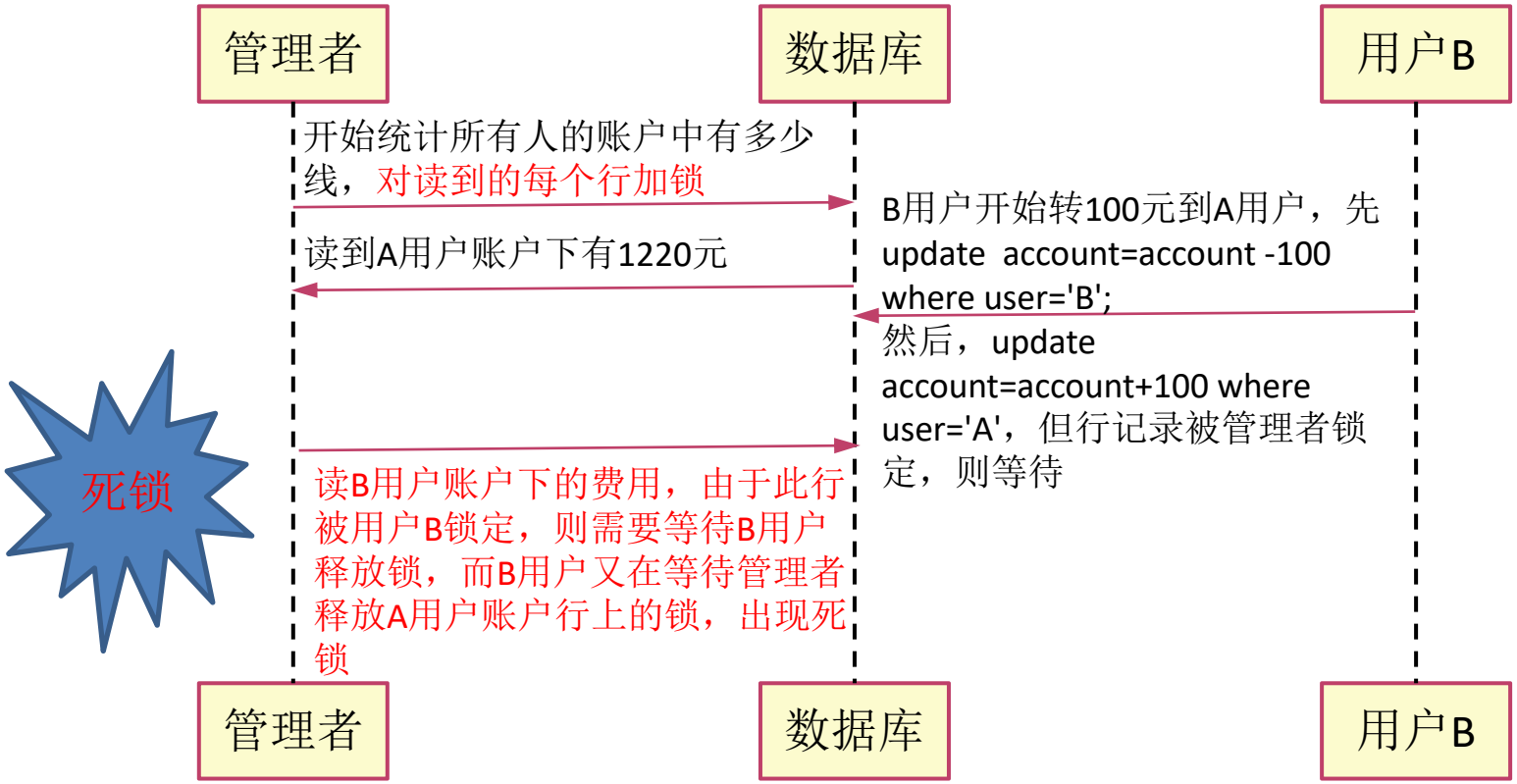
转账与统计总账的一个时序图



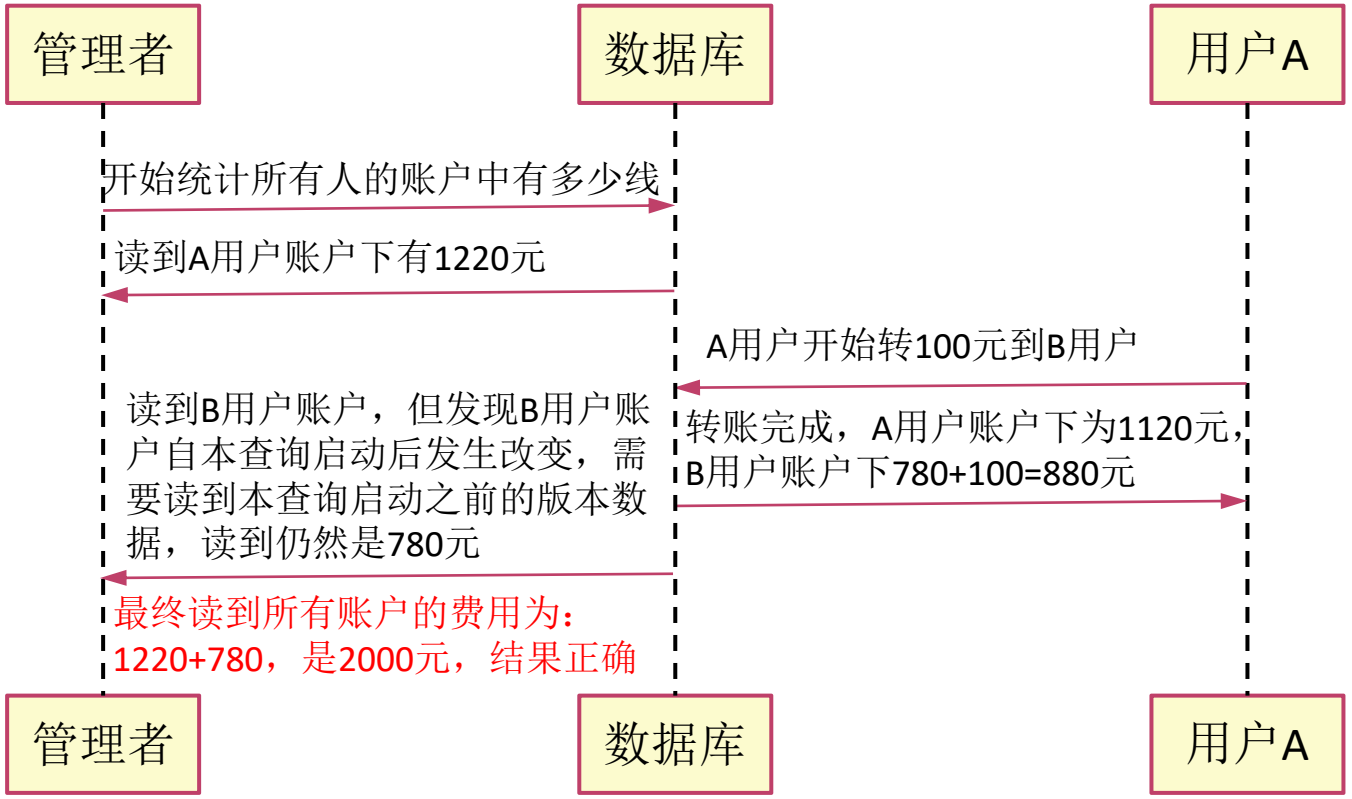
无MVCC数据库解决办法



无MVCC产生的新问题：死锁



MVCC下的转账与统计



MVCC的总结

- 查询和更新、删除、插入操作互相不阻塞
- 当开始一个查询后，读到的数据总是查询开始时那个时间点的快照
  - 在查询开始后，发生的变更（即使已提交），这次查询也是看不到的。
  - 一个事务无论运行多长时间，看到数据都是相同的
  - 不同开始时间的事务中相同的查询，返回的数据也可能不同

MVCC

MVCC实现方法

- 第一种：写新数据时，把旧数据移到一个单独的地方，如回滚段中，其他人读数据时，从回滚段中把旧的数据读出来。
- 第二种：写新数据时，旧数据不删除，而是把新数据插入。 PostgreSQL是使用的第二种方法，即旧的数据在数据块中不删除而Oracle数据库和MySQL中的innodb引擎使用的是第一种方法，即回滚段的方法

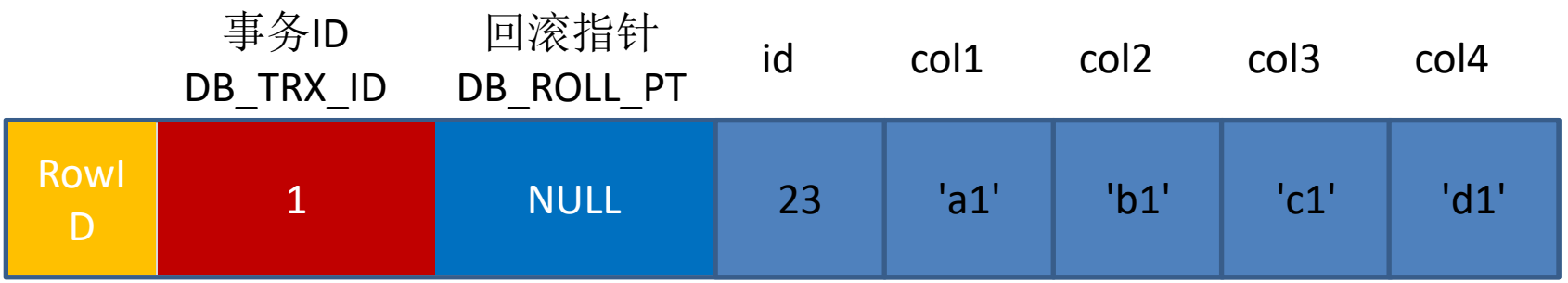
PostgreSQL是在数据库表增加了两个系统列：xmin和xmax，用于记录数据行的版本信息。xmin和xmax记录的是事务ID，需要我们先学习事务ID的相关知识后，再来了解多版本的具体实现。

Innodb中MVCC的实现

- redo log
  - 在变更数据之前，把变更先记录到一个文件中，称为redo log
- undo log
  - 与redo log相反，undo log是为回滚而用
- rollback segment
  - 在Innodb中，undo log被划分为多个段，具体某行的undo log就保存在某个段中，称为回滚段。

Innodb中MVCC的实现

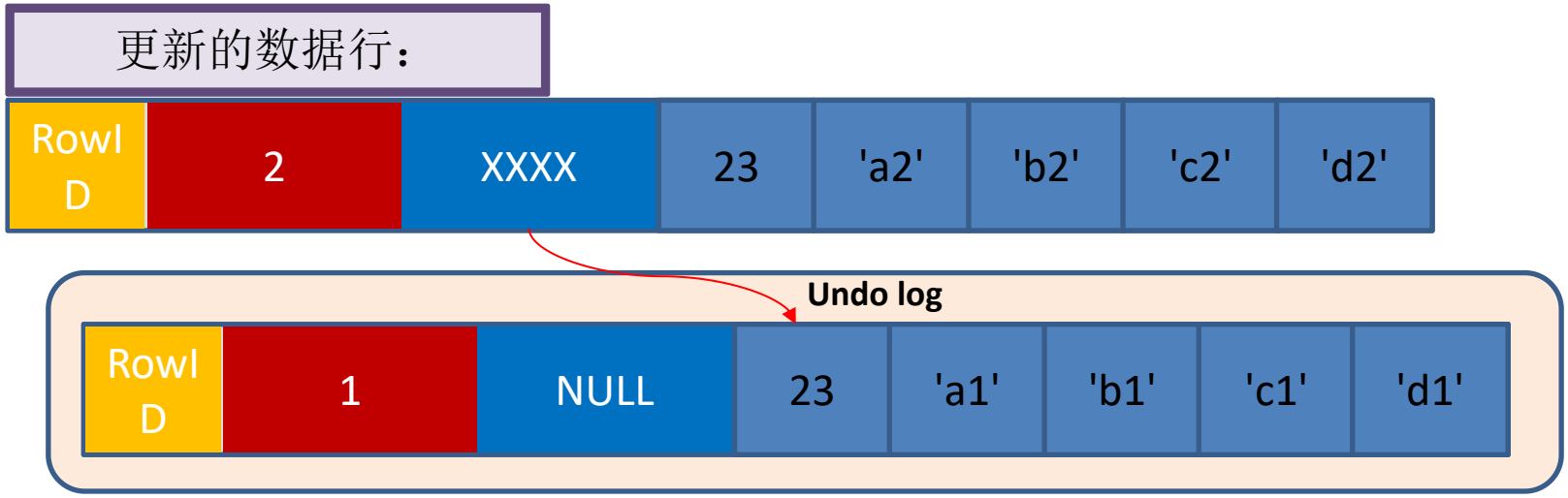
事务1：insert t(id, col1,col2,col3,col4) values(23,'a1','b1','c1','d1');



行上有三个隐含字段：分别对应该行的rowid、事务号和回滚指针，
id、Col1～Col4是表各列的名字，23、'a1'～'d1'是其对应的数据。

Innodb中MVCC的实现

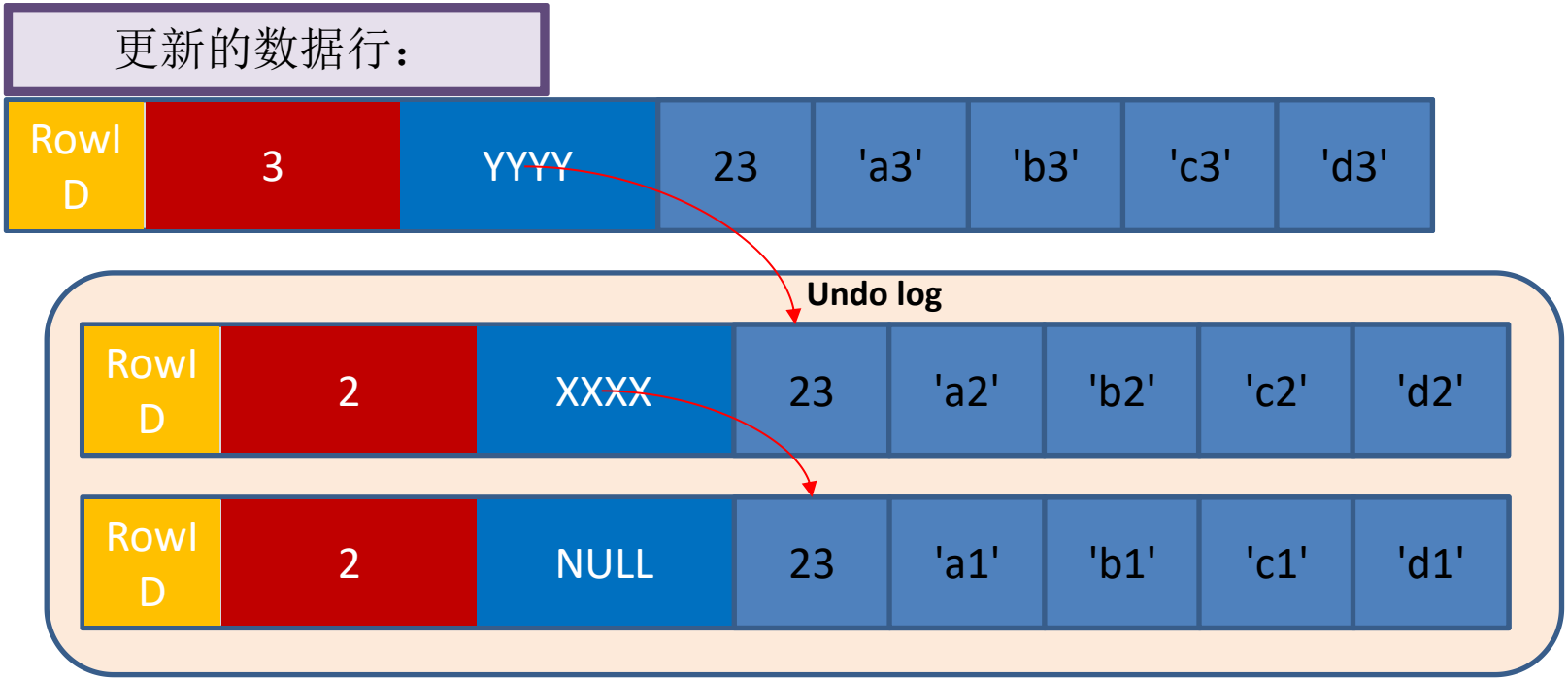
事务2: update table t set col1='a2', col2='b2', col3='c2', col4='d2' where id=23；



- 用排他锁锁定该行
- 把该行修改前的值Copy到undo log中。
- 修改当前行的值，填写事务编号，使回滚指针指向undo log中的修改前的行
- 记录redo log，包括undo log中的变化

Innodb中MVCC的实现

事务3: update table t set col1='a3', col2='b3', col3='c3', col4='d3' where id=23；



Innodb中MVCC的实现

- 多次更新后，回滚指针会把不同版本的记录串在一起。
- 在Innodb中存在purge线程，它会查询那些比现在最老的活动事务还早的undo log，并删除它们，从而保证undo log文件不至于无限增长。

Innodb的事务的提交与回滚

- 提交与回滚
  - 当事务正常提交时，Innodb只需要更改事务状态为COMMIT即可，不需做其他额外的工作
  - Rollback需要根据当前回滚指针从undo log中找出事务修改前的版本，并恢复。
    - 如果事务影响的行非常多，回滚则可能会很慢，根据经验值没提交的事务行数在1000～10000之间，Innodb效率还是非常高的。
    - 回滚时，也会产生redo日志
  - Innodb的COMMIT效率高，Rollback代价大

Innodb的可见性判断

- InnoDB表会有三个隐藏字段
  - 6字节的DB_ROW_ID
  - 6字节的DB_TX_ID
    - InnoDB内部维护了一个递增的tx id counter，其当前值可以通过show engine innodb status获得
  - 7字节的DB_ROLL_PTR(指向回滚段的地址)

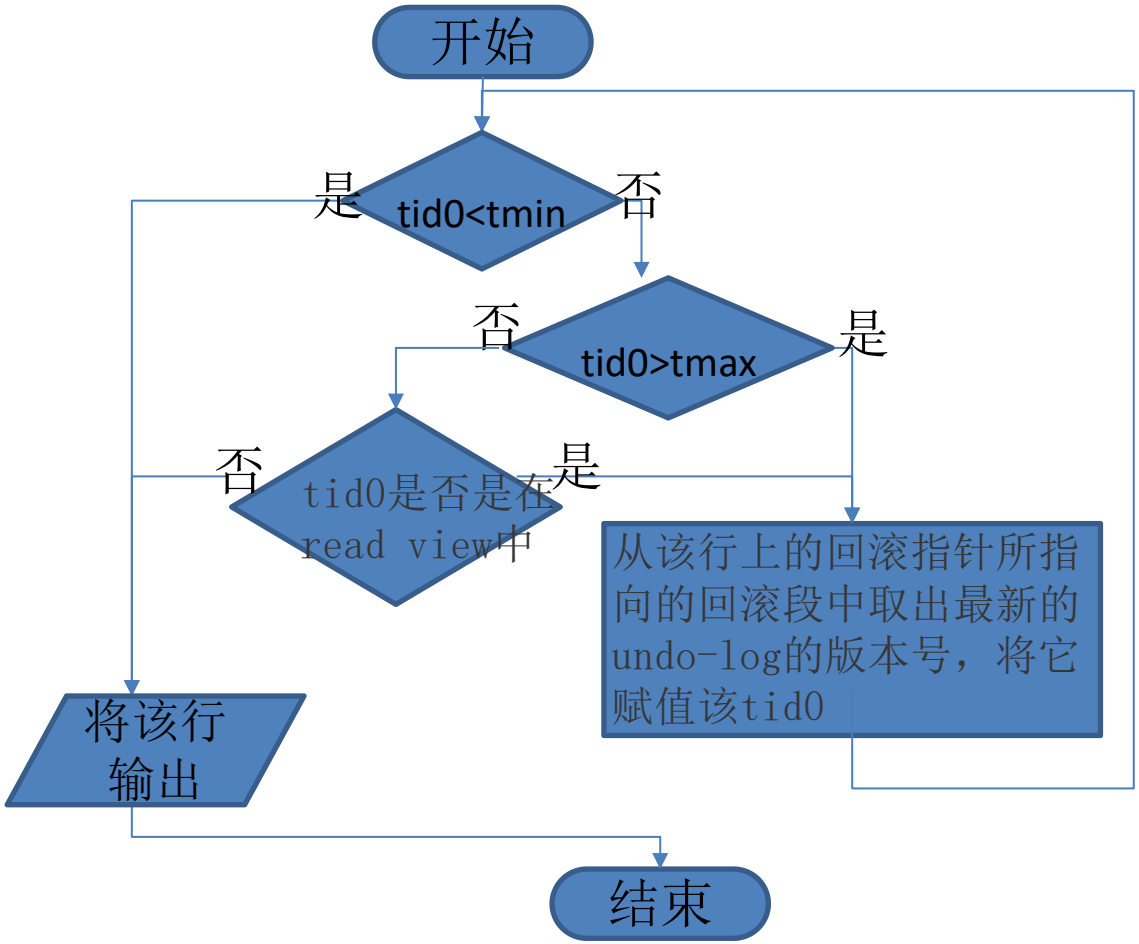
Innodb的可见性判断

- 可见性比较的方法：
  - 并不是用当前事务ID与表中各个数据行上的事务ID去比较的
  - 在每个事务开始的时候，会将当前系统中的所有的活跃事务拷贝到一个列表中 (read view)，，根据read view最早一个事务ID和最晚的一个事务ID来做比较的，这样就能确保在当前事务之前没有提交的所有事务的变更以及后续新启动的事务的变更，在当前事务中都是看不到的。
  - 当然，当前事务自身的变更还是需要看到的。

InnoDB可见性判断的流程

当开始一个事务时，把当前系统中活动的事务的ID都拷贝到一个列表（read view)中，这个列表中最早的事务ID为tmin，最晚的事务ID为tmax

当读到一行时，该行上当前事务id为tid0，当前行是否可见的判断逻辑见右图



Oracle的MVCC实现

- Oracle也是通过回滚段来实现多版本的
  - 但Oracle的实现更复杂，更精细一些。
- Oracle中也有事务ID，但不是递增的
  - Undo Segment Number + Transaction Table Slot Number + Wrap
- 与innodb不一样的地方：
  - 事务信息并不是记录在每个数据行上的，而是在块头中的ITL槽上，所以相对来说更省空间

Oracle的MVCC实现

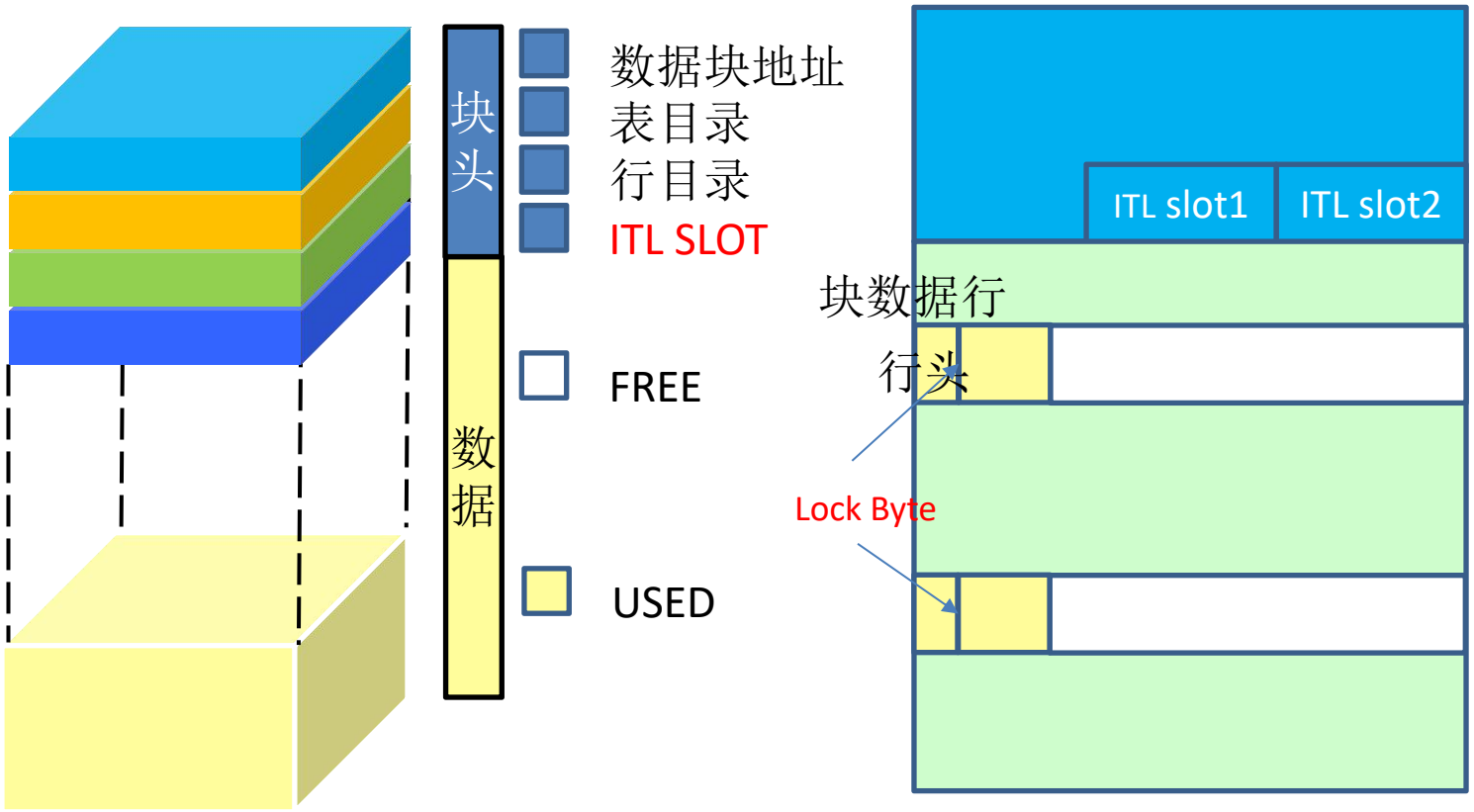
- ITL解释
  - ITL(Interested Transaction List)在Oracle数据块的头部
  - ITL记录在一个数据块中有多条，每一条itl可以看作是一个记录，每条记录常被称为槽位（itl slot)
  - 一个itl slot只可以记录一个事务的信息，如果这个事务已经提交或回滚了，那么这个itl的位置就可以被反复使用。

Oracle的MVCC实现

- ITL解释（续）
- 每个数据块上itl槽的多少可以动态创建，建表时可以指定：
  - initrans：每个数据块默认ITL槽数目，默认为2
    maxtrans：每个数据块最多的ITL槽数，最大255

Itl	Xid	Uba	Flag	Lck	Scn/Fsc
0x01	0x0006.002.0000158e	0x0080104d.00a1.6e	--U-	734	fsc 0x0000.6c9deff0
0x02	0x0000.000.00000000	0x00000000.0000.00	----	0	fsc 0x0000.00000000
…	…	…	…	…	…

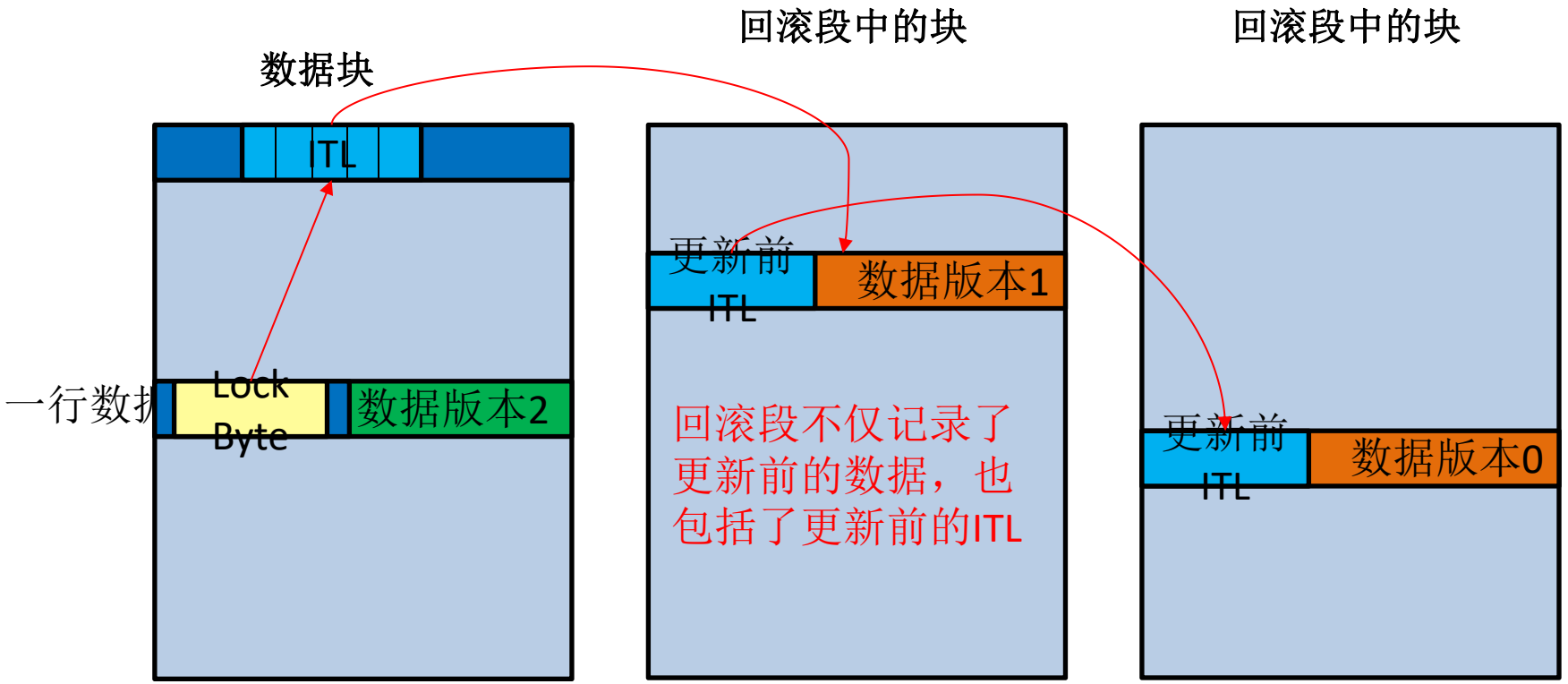
Oracle的MVCC实现：数据块的结构



Oracle的MVCC实现

- 由于Oracle中的事务ID不是递增的，为了判断事务之间的先后关系，需要一个递增序号，这个序号在Oracle中就叫SCN（当然SCN还有其它用处）
- SCN（System Change Number）
  - 是顺序递增的一个数字，在Oracle 中用来标识数据库的每一次改动，及其先后顺序。
  - SCN是由6字节组成，最大值是0xffff.ffffffff
  - 单节点的instance中，SCN值存在SGA区，由system commit number latch保护。任何进程要得到当前的SCN值，都要先得到这个latch。
  - RAC中是通过排队机制(Enqueue)实现SCN在各并行节点之间的顺序增长，这里不再赘述。

Oracle的MVCC实现：旧版本在回滚段中的结构



ITL中的Uba字段指向了回滚段中旧镜像的数据位置
ITL中的SCN用于比较版本的新旧

PostgreSQL MVCC实现

- 记住：PostgreSQL没有回滚段！！！
  - 旧数据是放在原有数据文件中的
  - 如果放在原有的数据文件中，旧数据越来越多怎么办？
    - 垃圾回收操作vacuum来做这个事。
    - 有自动垃圾回收autovacuum
  - 更新操作中新行的物理位置发生了变化，非更新列的索引是不是也要更新？
    - 通常不会，HOT技术。如果原有的数据块之间有空间，旧行与新行之间会建一个链接，索引上仍然指向旧的数据行。
  - 垃圾回收的代价会不会影响性能？
    - 有很多参数控制这个影响：vacuum_cost_delay，vacuum_cost_limit

PostgreSQL MVCC实现

- 实现方法
  - 每行上有xmin和xmax两个系统字段
  - 当插入一行数据时，将这行上的xmin设置为当前的事务id，而xmax设置为0
  - 当更新一行时，实际上是插入新行，把旧行上的xmax设置为当前事务id，新插入行的xmin设置为当前事务id，新行的xmax设置为0
  - 当删除一行时，把当前行的xmax设置为当前事务id
  - 当读到一行时，到commitlog中查询xmin和xmax对应的事务状态是否是已提交还是回滚了，就能判断出此行对当前行是否是可见。
  - autovacuum进程会把一些不要的旧行清理掉

PostgreSQL MVCC实现：事务ID

- 与innodb类似，是一个递增的数字，常常被称为xid
  - 但是一个无符号的32bit的数字表示
- 如何知道事务是提交了还是回滚了？
  - 事务的状态记录一个叫commitlog的位图文件中，即pg_clog目录下的文件中
  - 每个事务的状态用两个bit来表示：
    - #define TRANSACTION_STATUS_IN_PROGRESS 0x00
    - #define TRANSACTION_STATUS_COMMITTED 0x01
    - #define TRANSACTION_STATUS_ABORTED 0x02
    - #define TRANSACTION_STATUS_SUB_COMMITTED 0x03

PostgreSQL MVCC实现：事务ID

- 事务ID可以无限多个吗？

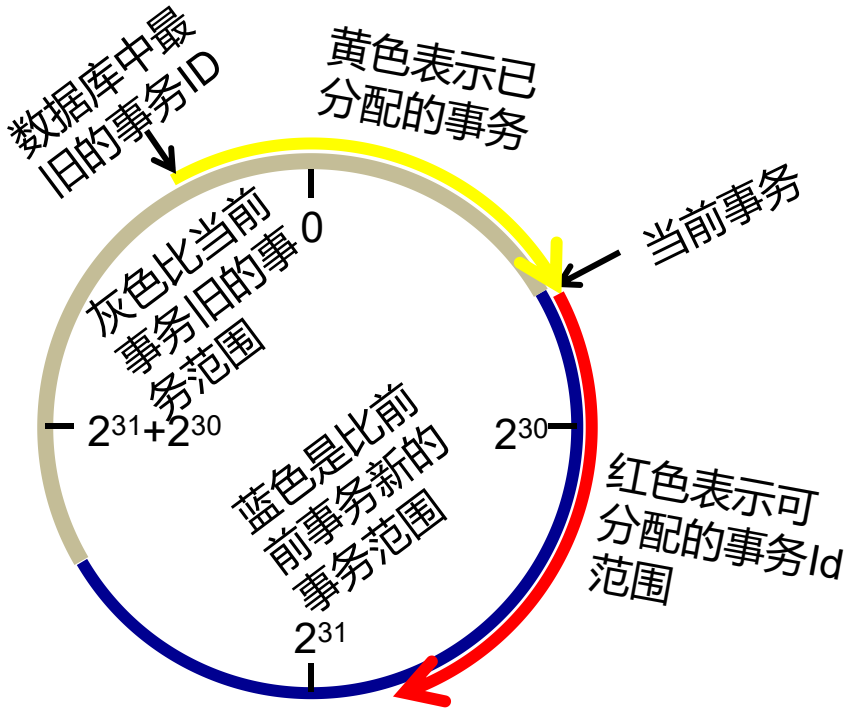
- 不能
  - 如果能的话，commit log就会无限制的增长下去。
  - 同时每次查询一个事务ID的状态时，也会变慢
  - 是一个32bit的数字
  - 当到达一个最大值时，又会从一个最小值开始
  - 可以想像是需要一种回收的机制来复用xid

PostgreSQL MVCC实现：事务ID

- 到大最大之后，如何又重新开始？
  - 当旧数据中的xid用一个特殊的值FrozenXID，即2来替换，原xid就能被再次重用。
  - 此问题被称之为事务ID回卷问题(Transaction ID Wraparound)

PostgreSQL MVCC实现：事务id回卷问题的解决

- 事务id的范围可以认为组成了一个圆
- 事务id从3开始，到达最大之后又变回3（0,1,2被占用了，后面再详解）
- 由vacuum把一些已提交的事务ID换成FrozeXID，保证当前最小的事务ID到当前事务ID的范围小于2^{31}，这样就能有新的事务ID可分配。
- 把FrozeXID认为是已提交的事务
- vacuum把已回滚的行直接回收掉，把行上xmin为已提交事务的ID换成FrozeXID。



PostgreSQL MVCC实现：事务ID

- 有三个特殊值的事务ID
  - 0: InvalidXID，无效事务ID
  - 1: BootstrapXID，表示系统表初使化时的事务ID，比任务普通的事务ID都旧。
  - 2:FrozenXID，冻结的事务ID，比任务普通的事务ID都旧。
  - 大于2的事务ID都是普通的事务ID
- commitlog的大小
  - 理论上，数据库中事务ID最多2^{31}个，每个事务占用2bit，所以commitlog最大512M字节
  - 实际上参数autovacuum_freeze_max_age为2亿，表示最旧事务ID到当前事务ID为2亿，所以commitlog通常最大大小为50M左右。

PostgreSQL MVCC实现： vacuum的注意事项

- 如果没有及时把旧的事务ID换成FrozeXID，会发生什么？
  - 不能分配新的事务ID，数据库会宕机！
  - 通常不会发生这个问题，当快用完时，会强制启动vacuum对事务ID进行回收
- autovacuum_freeze_max_age设置为2亿
  - 如果最旧的事务ID到当前要分配的事务ID接近2亿时，会强制启动vacuum。
  - 所以要经常运行vacuum，保证数据库中太旧的事务ID的年龄不要接近2亿。

PostgreSQL MVCC实现： 可见性判断

- 与innodb一样，事务开始时，会把当前活跃的事务ID记录到一个列表中，这称之为快照。
- 系统先通过判断t_xmin是否在全局活跃事务列表中、是否在事务快照活跃事务列表中、根据事务提交日志判断事务是提交还是回滚了等来判断t_xmin事务是否在事务开始时已经提交

PostgreSQL MVCC实现： 可见性判断

- 新旧比较方法
  - 普通事务的比较方法：(int32) (id1 - id2) < 0
  - 表达式算出来值为真，则id1比id2更旧一些，为假则id1比id2新
    - 例子：id1=4294967290，id2=5，id2是当事务回卷后的值，id1-id2=4294967285，而4294967285因大于2^31，转成int32后会变成一个负数，表过式为真，所以id1比id2更旧
  - BootstrapXID比所有其它事务都旧，包括FrozenXID
  - FrozenXID比普通事务旧

PostgreSQL MVCC实现： 可见性判断

- 当每执行一个SQL前，会生成一个SnapshotData的数据结构，同时把当前数据库中活动的事务ID，记录到此数据结构中，即SnapshotData.xip中，同时还需要把明显是已完成的的最大事务ID记录到SnapshotData.xmin中，未开始的事务ID记录到SnapshotData.xmax中
- 当读到一行时，读取这一行上的xmin和xmax，我们称为tuple.xmin和tuple.max
- 然后用tuple.xmin及tuple.max与在SnapshotData中的xip、xmax、xmin进行比较
- 比较的情况比较多，见后续

PostgreSQL MVCC实现： 可见性判断

- 首先是tuple.xmin与SnapshotData.xip、xmax、xmin的比较
- 如果tuple.xmin是在SnapshotData.xip中，说明此行是当前活动事务插入的行，可以忽略
- 如果tuple.xmin >SnapshotData.max中，说明此行是后来插入的行，也可以忽略
- 如果tuple.xmin <= SnapshotData.xmin，说明是已结束的事务，然后到commit log中查询事务状态，如果事务状态是已回滚，则也忽略，如果事务状态为已提交，则看tuple.max的值，具体请看下一页

MVCC

- 多版本实现中，首先要解决的就是旧数据的空间释放问题。PostgreSQL通过运行vacuum进程来回收之前的存储空间，autovacuum默认是打开的，也可以关闭，但vacuum就需要手工来执行了。

- 数据是否有效是用过记录事务的状态来实现的，因为数据行上记录了tmin,tmax，所以了解了tmin和tmax对应的事务是成功提交了还是回滚了，就可以指导这些数据行是有效。事务的状态记录在commit log中，简称clog，

  - [postgres@postgres data]$ ls pg_clog/
  - 0363 0375 0387 0399 03AB 03BD 03CF 03E1 03F3 0405 0417

PostgreSQL MVCC实现： 可见性判断

- 如果tuple.xmin <= SnapshotData.xmin，且tuple.min已提交，判断tuple.max的值
  - 如果tuple.max为零，返回当前行
  - tuple.max in SnapshotData.xip，返回当前行
  - tuple.max<SnapshotData.xmin，则需要查询tuple.max在commit log中的状态，如果tuple.max为已回滚，则返回当前行，如果为已提交，则忽略此行

PostgreSQL MVCC实现： xmin、xmax演示

- xmin 在创建（insert）记录（tuple）时，记录此值为插入tuple的事务ID
- xmax 默认值为0.在删除tuple时，记录此值
- cmin和cmax 标识在同一个事务中多个语句命令的序列值，从0开始，用于同一个事务中实现版本可见性判断

PostgreSQL MVCC实现： xmin、xmax演示

```
create table test(id int,name text);
postgres@postgres :begin;
BEGIN
postgres@postgres :select txid_current();
 txid_current
--------------
       114773


postgres@postgres :insert into test values(1,'a');
INSERT 0 1
postgres@postgres :select *,xmin,xmax,cmin,cmax from test;
 id | name |  xmin  | xmax | cmin | cmax
----+------+--------+------+------+------
  1 | a    | 114773 |    0 |    0 |    0
```

```
postgres@postgres :insert into test values(2,'b');
INSERT 0 1
postgres@postgres :insert into test values(3,'c');
INSERT 0 1
postgres@postgres :select *,xmin,xmax,cmin,cmax from test;
 id | name |  xmin  | xmax | cmin | cmax
----+------+--------+------+------+------
  1 | a    | 114773 |    0 |    0 |    0
  2 | b    | 114773 |    0 |    1 |    1
  3 | c    | 114773 |    0 |    2 |    2
(3 rows)
postgres@postgres :commit;
COMMIT
```

session1	session2
postgres@postgres :begin; postgres@postgres :update test set name='d' where id=1;	
	id=1的xmax将被置为114774,xmax被设置为删除tuple的事务的ID postgres@postgres :select *,xmin,xmax,cmin,cmax from test; id | name | xmin | xmax | cmin | cmax ----+------+--------+--------+------+------ 1 | a | 114773 | 114774 | 0 | 0 2 | b | 114773 | 0 | 1 | 1 3 | c | 114773 | 0 | 2 | 2
postgres@postgres :commit;	
id=1 xmin为11473的不见了，新插入了一行xid为11474的行 postgres@postgres :select *,xmin,xmax,cmin,cmax from test; id | name | xmin | xmax | cmin | cmax ----+------+--------+------+------+------ 2 | b | 114773 | 0 | 1 | 1 3 | c | 114773 | 0 | 2 | 2 1 | d | 114774 | 0 | 0 | 0	

PostgreSQL MVCC实现：表龄问题

- 根据 PostgreSQL 的 MVCC 机制，数据被插入时 PostgreSQL 会分配给每行 tuples 一个 事务ID，即表上的隐含字段 xmin， 而 PostgreSQL 的事务号由 32 bit 位 (40 亿) 组成，事务号分配完了后会循环，这样会造成过去的记录不可见的情况，为了解决这个问题，理论上在 20 亿事务之内需要 vacuum 每一个数据库的每一个表， 而vacuum 操作会替换某些老记录的xid 成 FrozenXID，这样即使事务号循环，这些被替换成 FrozenXID 的记录依然可见。

PostgreSQL MVCC实现： 表龄演示

- postgres@postgres :create table test1(id integer,name text);
- postgres@postgres :insert into test1 values (1,'a');
- postgres@postgres :insert into test1 values (2,'b');
- postgres@postgres :insert into test1 values (3,'c');
  - postgres@postgres :select relname,relfrozenxid,age(relfrozenxid) from pg_class where relname='test1';

```
 relname | relfrozenxid | age
---------+--------------+-----
test1   |       114775 |   4
```

PostgreSQL MVCC实现： 表龄演示

- postgres@postgres :select xmin,age(xmin),* from test1;

```
 xmin  | age | id | name
--------+-----+----+------
 114776 |   3 |  1 | a
 114777 |   2 |  2 | b
 114778 |   1 |  3 | c
```

- 执行vacuum freeze
  - postgres@postgres : vacuum freeze test1;
  - postgres@postgres :select relname,relfrozenxid,age(relfrozenxid) from pg_class where relname='test1';

```
 relname | relfrozenxid | age
---------+--------------+-----
 test1   |       114779 |   0
```

PostgreSQL MVCC实现： vacuum

- autovacuum 是 postgresql 里非常重要的一个服务端进程。它能够自动地执行，在一定条件下自动地对 dead tuples 进行清理并对表进行分析
  - autovacuum (boolean)
    - autovacuum参数控制 autovacuum 进程是否打开,默认为 on
  - log_autovacuum_min_duration(integer)
    - 这个参数用来记录 autovacuum 的执行时间，当 autovaccum 的执行时间超过多少 -1表示不记录
  - autovacuum_max_workers (integer)
    - 指定同时运行的 最大的 autovacuum 进程，默认为3个
  - autovacuum_naptime (integer)
    - 指定 autovacuum 进程运行的最小间隔，默认为 1 min。也就是说当前一个 autovacuum 进程运行完成后，第二个 autovacuum 进程至少在一分钟后才会运行。

PostgreSQL MVCC实现： vacuum

- autovacuum_vacuum_threshold (integer)
  - autovacuum 进程进行vacuum 操作的阀值条件一，(指修改，删除的记录数。)
- autovacuum_analyze_threshold (integer)
  - autovacuum 进程进行 analyze 操作的阀值条件一，(指插入，修改，删除的记录数。)
- autovacuum_vacuum_scale_factor (floating point)
  - autovacuum因子, autovacuum 进程进行 vacuum 操作的阀值条件二,，默认为 0.2
  - autovacuum进程进行 vacuum 触发条件
  - 表上(update,delte 记录) >= autovacuum_vacuum_scale_factor* reltuples(表上记录数) + autovacuum_vacuum_threshold

PostgreSQL MVCC实现： vacuum

- autovacuum_analyze_scale_factor (floating point)
  - autoanalyze 因子,autovacuum 进程进行 analyze 操作的阀值条件二,，默认为 0.1
  - autovacuum进程进行 analyze 触发条件
    - 表上(insert,update,delte 记录) >= autovacuum_analyze_scale_factor* reltuples(表上记录数) + autovacuum_analyze_threshold
- autovacuum_freeze_max_age (integer)
  - 指定表上事务的最大年龄，默认为2亿，达到这个阀值将触发 autovacuum进程，从而避免 wraparound。
  - 表上的事务年龄可以通过 pg_class.relfrozenxid 查询

PostgreSQL MVCC实现： vacuum

- autovacuum_vacuum_cost_delay (integer)
  - 当autovacuum进程即将执行时，对 vacuum 执行 cost 进行评估，如果超过 autovacuum_vacuum_cost_limit设置值时，则延迟，这个延迟的时间即为 autovacuum_vacuum_cost_delay。如果值为 -1, 表示使用 vacuum_cost_delay 值，默认值为 20 ms
- autovacuum_vacuum_cost_limit (integer)
  - 这个值为 autovacuum 进程的评估阀值, 默认为 -1, 表示使用 "vacuum_cost_limit " 值，如果在执行 autovacuum 进程期间评估的 cost 超autovacuum_vacuum_cost_limit, 则 autovacuum 进程则会休眠

PostgreSQL MVCC实现：优缺点

- PostgreSQL的多版本优势
  - 事务回滚可以立即完成，无论事务进行了多少操作
  - 数据可以进行很多更新
  - 旧版本数据需要清理，有autovacuum进程和vacuum命令处理
  - 旧版本的数据存在与数据文件中，需要及时清理，否则可能会导致查询慢
- PostgreSQL中的MVCC缺点
  - 事务ID个数有限制，事务ID由32位数保存，而事务ID递增，当事务ID用完时，会出现wraparound问题。

PostgreSQL MVCC实现：思考

- 删除数据之后能否恢复?
  - delete from testtab01；删除一张表的内容之后，能否恢复？

- 如果开启了autovacuum
  - 数据还不有被autovacuum回收掉
  - autovacuum的运行周期通常在10秒到数分钟之间，所以如果要想恢复数据，立即把postgresql的进程kill掉，或直接关电源，这才可能。
  - 还有一种可能，就是数据库中有一个在删除操作之前就开始的长事务在运行，还没有结束，这时，这部分旧数据不会被回收。
- 如果没有开启autovacuum
  - 这时不要做vacuum，数据都还是可能恢复的。

PostgreSQL MVCC实现：vacuum经验

- 对于频繁更新的表
  - 应该更频繁的执行普通vacuum操作以防止表膨胀
- 不要有长时间idle事务
  - 这会导致这个事务之后产生的垃圾数据都不能被回收，因为vacuum认认这部分数据有可能会被你这个事务访问。
  - lock_timeout、statement_timeout

事务隔离级别

SQL标准定义了四个级别的事务隔离。最严格的是串行化，它是通过标准来定义的，也就是说， 保证一组可序列化事务的并发执行以产生同样顺序依次运行它们的同一效果

其他三个层次是通过现象术语被定义，导致并发事务之间的相互作用，这不应该发生在每个级别中

脏读：一个事务读取了另一个并行的未提交事务写入的数据。(包括insert，update，delete)

不可重复读：一个事务A重新读取前面读取过的数据，发现该数据已经被另一个已提交事务B修改（这个事务B的提交是在事务A第一次读之后发生的）。(同一行，update)

幻读：一个事务重新执行一个查询，返回一套符合查询条件的行，发现这些行因为其它最近提交的事务而发生了改变。(结果集，insert、delete)

事务隔离级别

1,读未提交(READ UNCOMMITTED)
2,读已提交(READ COMMITTED)
3,可重复读(REPEATABLE READS)
4,可序列化(SERIALIZABLE)

隔离级别	脏读	不可重复读	幻读
读未提交	可能	可能	可能
读已提交	不可能	可能	可能
可重复读	不可能	不可能	可能
可串行化	不可能	不可能	不可能

事务隔离级别

在PostgreSQL里，你可以请求四种可能的事务隔离级别中的任意一种。但是在内部，实际上只有三种独立的隔离级别，分别对应读已提交，可重复读和可串行化。如果你选择了读未提交的级别， 实际上你用的是读已提交，在重复读的PostgreSQL执行时，幻读是不可能的， 所以实际的隔离级别可能比你选择的更严格。这是 SQL 标准允许的：四种隔离级别只定义了哪种现像不能发生， 但是没有定义那种现像一定发生。PostgreSQL只提供三种隔离级别的原因是：这是把标准的隔离级别与多版本并发控制架构映射相关的唯一合理方法

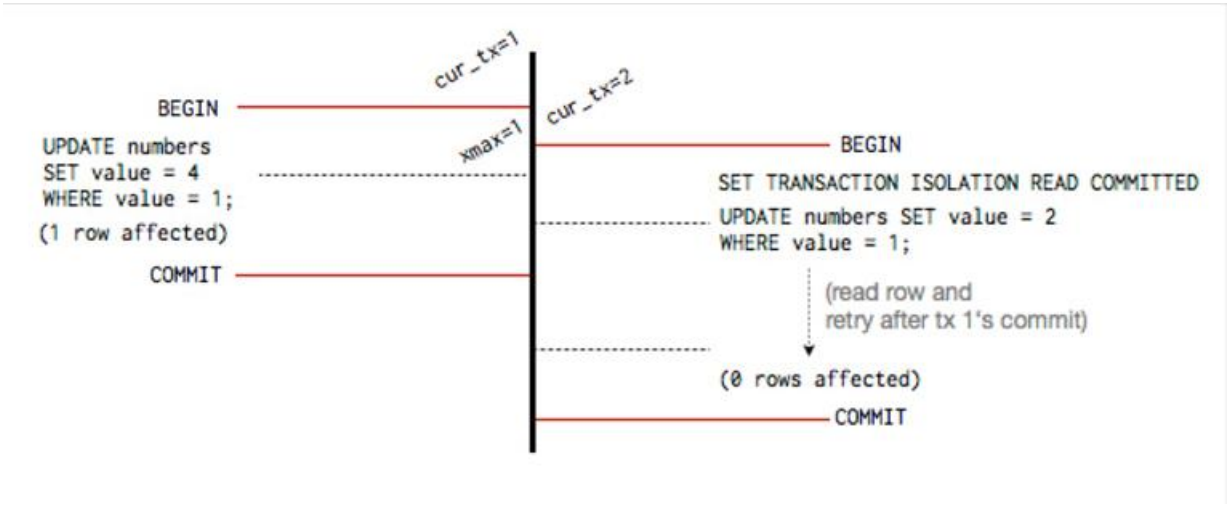
事务隔离级别

读已提交隔离级别(BEGIN TRANSACTION ISOLATION LEVEl READ COMMITTED;)

是PostgreSQL里的缺省隔离级别。当一个事务运行在这个隔离级别时: SELECT查询(没有FOR UPDATE/SHARE子句)只能看到查询开始之前已提交的数据而无法看到未提交的数据或者在查询执行期间其它事务已提交的数据 (仅读当时数据库快照)。不过，SELECT看得见其自身所在事务中前面更新执行结果，即使它们尚未提交。（注意：在同一个事务里两个相邻的SELECT命令可能看到不同的快照，因为其它事务会在第一个SELECT执行期间提交。）

事务隔离级别

读已提交隔离级别(BEGIN TRANSACTION ISOLATION LEVEl READ COMMITTED;)
要是同时有两个事务修改同一行数据会怎么样？这就是事务隔离级别（transaction isolation levels）登场的时候了。Postgres支持两个基本的模型来让你控制应该怎么处理这样的情况。默认情况下使用读已提交（READ COMMITTED），等待初始的事务完成后再读取行记录然后执行语句。如果在等待的过程中记录被修改了，它就从头再来一遍。举一个例子，当你执行一条带有WHERE子句的UPDATE时，WHERE子句会在最初的事务被提交后返回命中的记录结果，如果这时WHERE子句的条件仍然能得到满足的话，UPDATE才会被执行。在下面这个例子中，两个事务同时修改同一行记录，最初的UPDATE 语句导致第二个事务的WHERE不会返回任何记录，因此第二个事务根本没有修改到任何记录：

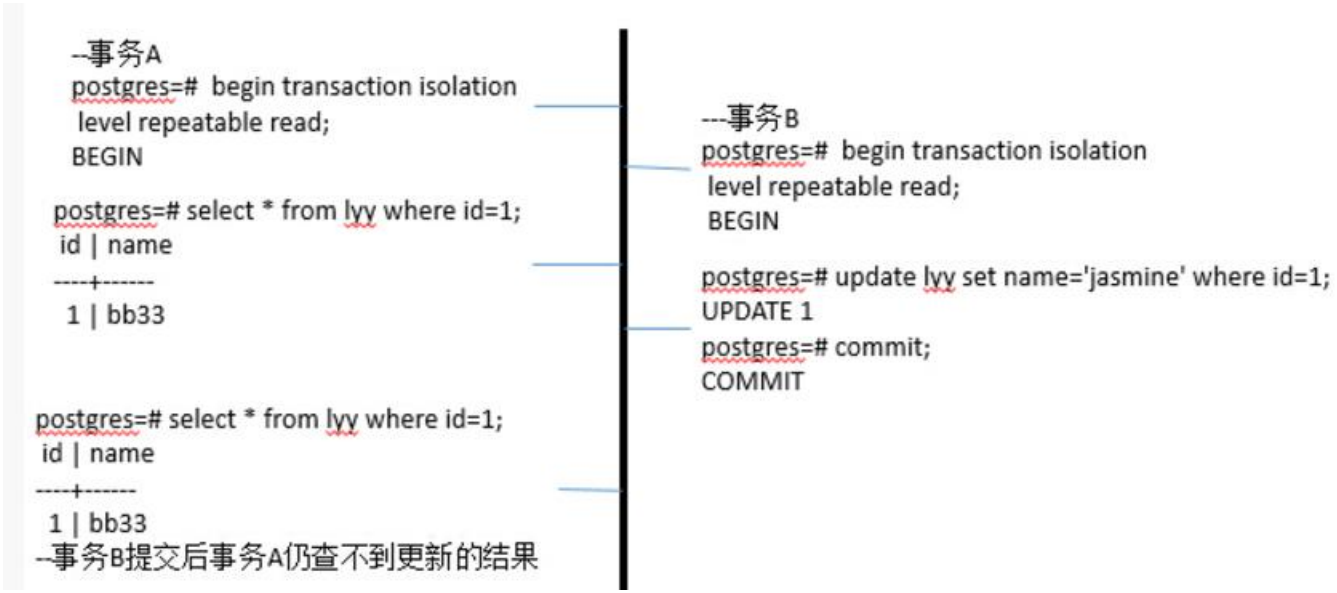


事务隔离级别

可重复读隔离级别(BEGIN TRANSACTION ISOLATION LEVEl REPEATABLE READ;)

这个级别和读已提交级别是不一样的。重复读事务中的查询看到的只是事务开始时的快照， 而不是该事务内部当前查询开始时的快照，这样，同一个事务内部后面的SELECT命令总是看到同样的数据等，它们没有看到通过 自身事务开始之后提及的其他事务做出的改变。

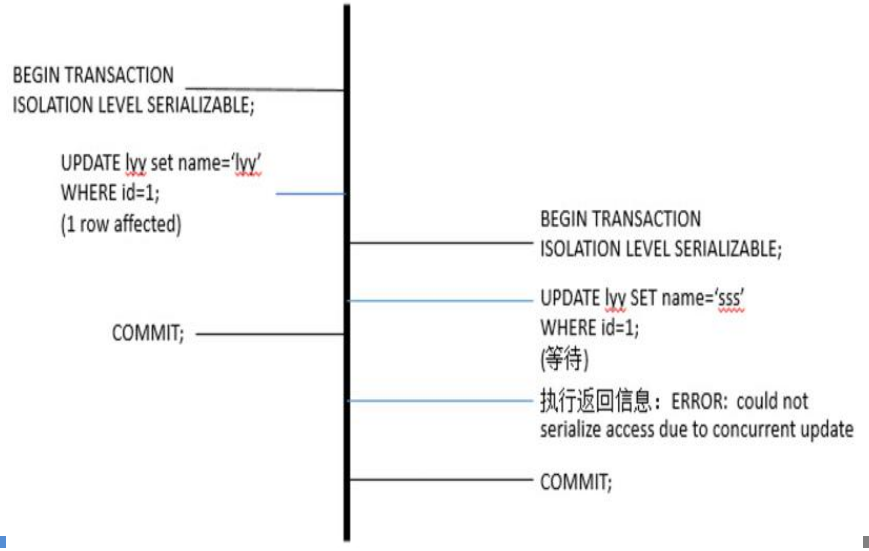
使用这个级别的应用必须准备好重试事务，因为串行化失败



事务隔离级别

可串行化隔离级别：(BEGIN TRANSACTION ISOLATION LEVEl SERIALIZABLE;)

可串行化级别提供最严格的事务隔离。这个级别为所有已提交事务模拟串行的事务执行， 就好像事务将被一个接着一个那样串行(而不是并行)的执行。不过， 正如可重复读隔离级别一样， 使用这个级别的应用必须准备在串行化失败的时候重新启动事务。 事实上，该隔离级别和可重复读希望的完全一样，它只是监视这些条件，以所有事务的可能的序列不一致的（一次一个）的方式执行并行的可序列化事务执行的行为。 这种监测不引入任何阻止可重复读出现的行为，但有一些开销的监测， 检测条件这可能会导致序列化异常 将触发序列化失败



目录

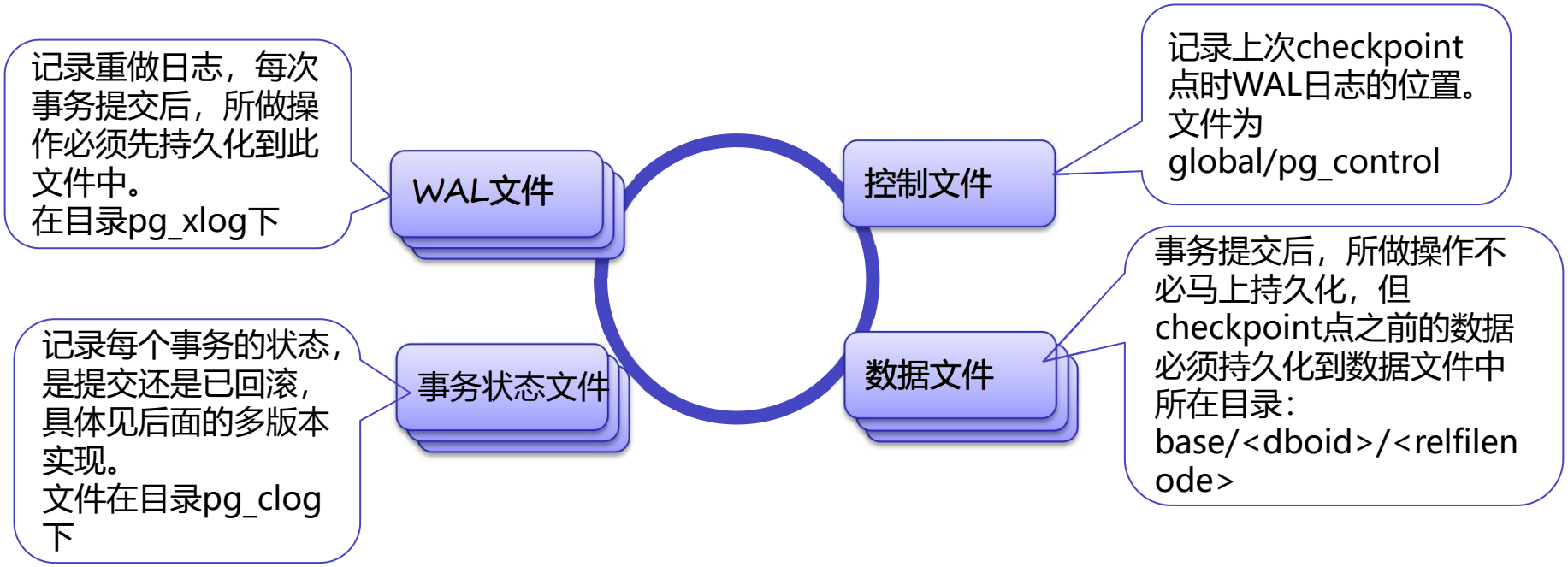
实例恢复原理：如何做到不丢数据

- 导致数据库实例异常终止的原因
  - 被kill掉，如内存不足时被OOM Killer给kill掉了
  - 操作系统崩溃
  - 硬件故障导致机器停机或重启
- 只要磁盘上的数据没有丢失，PostgreSQL就不会丢失数据
- 不丢数据的定义
  - 数据库实例还能再次启动
  - 已提交的数据，数据库重启后还在
  - 不会出现数据错乱的情况

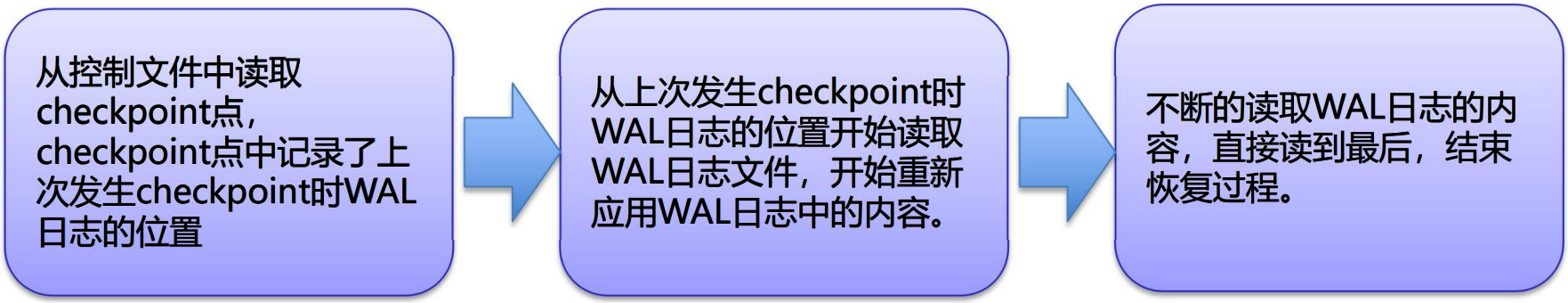
实例恢复原理：如何做到不丢数据

- 每项操作记录到重做日志中，实例重新启动后，重演（replay）日志，这个动作称为“前滚”
  - “前滚”完成后，多数数据库还会把未完成的事务取消掉，就象这些事务从来没有执行过一样。这个动作称之为“回滚”
  - 在“前滚”过程中，数据库是不能被用户访问的。
  - 每次“前滚”时，从哪个点开始？
    - Checkpoint点的概念登场了：保证Checkpoint点之前的数据已持久化到硬盘中了。
    - 发生Checkpoint点的周期通常在几分钟

实例恢复：总体设计


记录重做日志，每次事务提交后，所做操作必须先持久化到此文件中。
在目录pg_xlog下
WAL文件
记录上次checkpoint点时WAL日志的位置。
文件为global/pg_control
控制文件
事务提交后，所做操作不必马上持久化，但checkpoint点之前的数据必须持久化到数据文件中
所在目录：base/<dboid>/<relfilenode>
数据文件
记录每个事务的状态，是提交还是已回滚，具体见后面的多版本实现。
文件在目录pg_clog下
事务状态文件

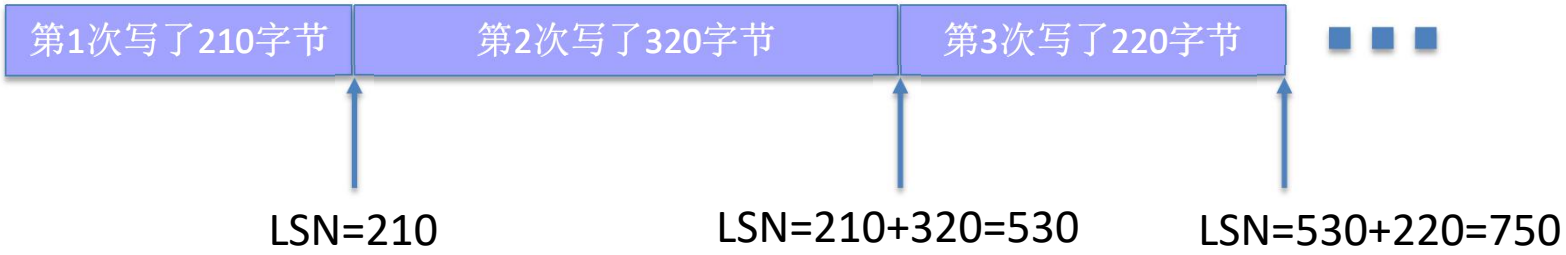
实例恢复：执行过程


从控制文件中读取checkpoint点，checkpoint点中记录了上次发生checkpoint时WAL日志的位置
从上次发生checkpoint时WAL日志的位置开始读取WAL日志文件，开始重新应用WAL日志中的内容。
不断的读取WAL日志的内容，直接读到最后，结束恢复过程。

实例恢复：如何定位WAL日志文件

实例恢复：WAL日志位置的表示

- 日志位置用LSN来表示：Log Sequence Number，是WAL日志的绝对位置



LSN实际上是用64bit的一个数字来表示

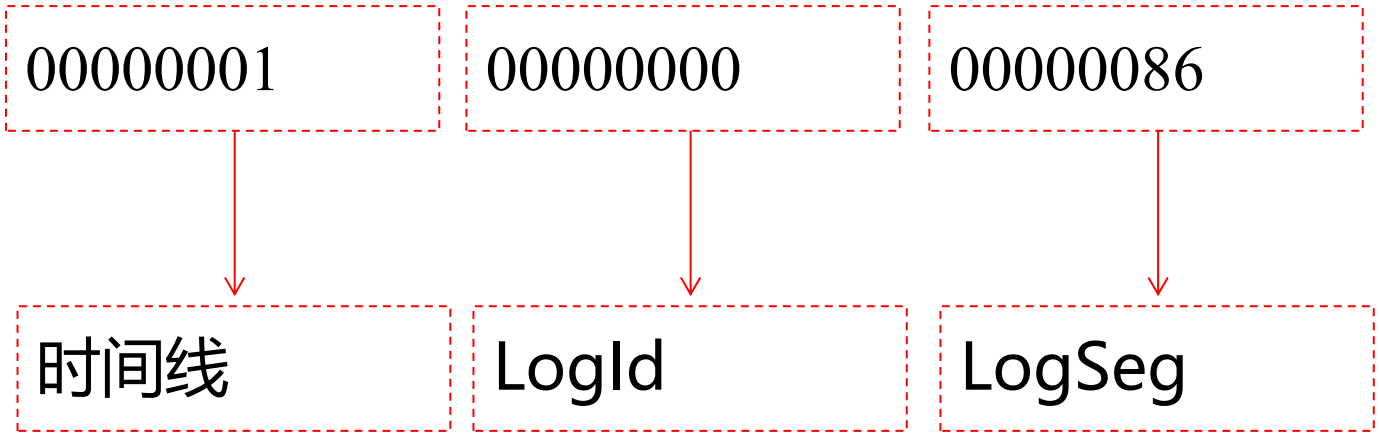
实例恢复：WAL日志文件名的秘密

- WAL日志文件名由24个字母组成，文件名中就有起始的WAL日志的位置

```
osdba-mac:~ osdba$ ls -l  $PGDATA/pg_xlog
total 262144
-rw-------  1 osdba  osdba  16777216 Oct  8 10:57 0000000100000000000000B6
-rw-------  1 osdba  osdba  16777216 Jun 17 22:12 0000000100000000000000B7
-rw-------  1 osdba  osdba  16777216 Jun 17 22:12 0000000100000000000000B8
-rw-------  1 osdba  osdba  16777216 Jun 17 22:12 0000000100000000000000B9
-rw-------  1 osdba  osdba  16777216 Jun 17 22:12 0000000100000000000000BA
-rw-------  1 osdba  osdba  16777216 Jun 17 22:12 0000000100000000000000BB
-rw-------  1 osdba  osdba  16777216 Jun 17 22:13 0000000100000000000000BC
-rw-------  1 osdba  osdba  16777216 Jul 23 19:01 0000000100000000000000BD
drwx------  2 osdba  osdba        68 Mar  9  2015 archive_status
```

实例恢复：WAL日志文件名的秘密

- 文件名的组成：



实例恢复： WAL文件名的组成

- 文件名由24字符组成
  - 时间线：英文为timeline，是以1开始的递增数字，如1,2,3...
  - LogId：32bit长的一个数字，是以0开始递增的数字，如0,1,2,3...。实际为LSN的高32bit
  - LogSeg：32bit长的一个数字，是以0开始递增的数字，如0,1,2,3...。
  - LogSeg是LogSeg是LSN的低32bit的字节的值再除以WAL文件大小（16M）的结果。注意：当LogId为0时，LogSeg是从1开始的。

实例恢复： WAL文件名的组成

- WAL日志文件默认大小为16M
  - 如果想改变大小，需要重新编译程序。
- 所以LogSeg最大为FF，即从000000~0000FF，即在文件名中，最后8字节中前6字节总是0。
- 2**32/(16M) = 256，即FF。

实例恢复： 读取控制文件中的起始WAL文件

```
[postgres@pg01 pgdata]$ pg_controldata
pg_control version number:            960
Catalog version number:               201608131
Database system identifier:           64461573383395
Database cluster state:               in production
pg_control last modified:             Fri 06 Oct 201
Latest checkpoint location:           F/5C2279A0
Prior checkpoint location:            F/5C2278C0
Latest checkpoint's REDO location:    F/5C227968
Latest checkpoint's REDO WAL file:    0000000100000000F0000005C
Latest checkpoint's TimeLineID:       1
Latest checkpoint's PrevTimeLineID:   1
Latest checkpoint's full_page_writes: on
Latest checkpoint's NextXID:          0:2770284
```

恢复时，会从此WAL文件开始
0000000100000000F0000005C

实例恢复： 恢复到哪个WAL文件结束？

```
[postgres@pg01 pg_xlog]$ ls
000000010000000F0000000C  000000010000000F00000024  000000010000000F0000003C  000000010000000F00000054  000000010000000F0000006C  000000010000000F00000084
000000010000000F0000000D  000000010000000F00000025  000000010000000F0000003D  000000010000000F00000055  000000010000000F0000006D  000000010000000F00000085
000000010000000F0000000E  000000010000000F00000026  000000010000000F0000003E  000000010000000F00000056  000000010000000F0000006E  000000010000000F00000086
000000010000000F0000000F  000000010000000F00000027  000000010000000F0000003F  000                                           0000000F00000087
000000010000000F00000010  000000010000000F00000028  000000010000000F00000040  000                                           0000000F00000088
000000010000000F00000011  000000010000000F00000029  000000010000000F00000041  000                                           0000000F00000089
000000010000000F00000012  000000010000000F0000002A  000000010000000F00000042  000                                           0000000F0000008A
000000010000000F00000013  000000010000000F0000002B  000000010000000F00000043  000                                           0000000F0000008B
000000010000000F00000014  000000010000000F0000002C  000000010000000F00000044  000                                           10000000F0000008C
000000010000000F00000015  000000010000000F0000002D  000000010000000F00000045  000000010000000F0000005D  000000010000                0000000F0000008D
000000010000000F00000016  000000010000000F0000002E  000000010000000F00000046  000000010000000F0000005E  000000010000000                0000000F0000008E
000000010000000F00000017  000000010000000F0000002F  000000010000000F00000047  000000010000000F0000005F  000000010000000F00                 000010000000F0000008F
000000010000000F00000018  000000010000000F00000030  000000010000000F00000048  000000010000000F00000060  000000010000000F0000                00010000000F00000090
000000010000000F00000019  000000010000000F00000031  000000010000000F00000049  000000010000000F00000061  000000010000000F000000              010000000F00000091
000000010000000F0000001A  000000010000000F00000032  000000010000000F0000004A  000000010000000F00000062  000000010000000F0000007A            10000000F00000092
000000010000000F0000001B  000000010000000F00000033  000000010000000F0000004B  000000010000000F00000063  000000010000000F0000007B  000               0000000F00000093
000000010000000F0000001C  000000010000000F00000034  000000010000000F0000004C  000000010000000F00000064  000000010000000F0000007C  000000              00000F00000094
000000010000000F0000001D  000000010000000F00000035  000000010000000F0000004D  000000010000000F00000065  000000010000000F0000007D  00000001            0000F00000095
000000010000000F0000001E  000000010000000F00000036  000000010000000F0000004E  000000010000000F00000066  000000010000000F0000007E  00000001000        00F00000096
000000010000000F0000001F  000000010000000F00000037  000000010000000F0000004F  000000010000000F00000067  000000010000000F0000007F  000000010000000F00000097
000000010000000F00000020  000000010000000F00000038  000000010000000F00000050  000000010000000F00000068  000000010000000F00000080  000000010000000F00000098
000000010000000F00000021  000000010000000F00000039  000000010000000F00000051  000000010000000F00000069  000000010000000F00000081  archive_status
000000010000000F00000022  000000010000000F0000003A  000000010000000F00000052  000000010000000F0000006A  000000010000000F00000082
000000010000000F00000023  000000010000000F0000003B  000000010000000F00000053  000000010000000F0000006B  000000010000000F00000083
```

最后一个WAL文件
000000010000000F00000098

实例恢复： 恢复到哪个WAL文件结束？

- 恢复时，是否是一直重放日志到WAL日志目录下的最后一个WAL文件？如000000010000000F00000098？

实例恢复： 恢复到哪个WAL文件结束？

- 答案：不是
- 因为这个文件通常并不是最后一个WAL日志文件
- 为什么？

实例恢复： WAL文件的复用机制

- 为了解释上一个疑问，我们需要说明WAL日志的循环复用机制。
- PostgreSQL数据库并不象Oracle数据库那个有固定多个Redo log文件。
- Oralce有固定多个Redo log文件，然后进行循环写。
- 循环写是为了提升性能
  - 如果每次生成新文件，新文件是否要初使化成16M？如果要初使化成16M，会产生额外的IO。
  - 如果不初使化，使用append的方式，文件的长度每次增加，这时需要更新文件的元数据，也会产生额外的开销。

实例恢复： WAL文件的复用机制

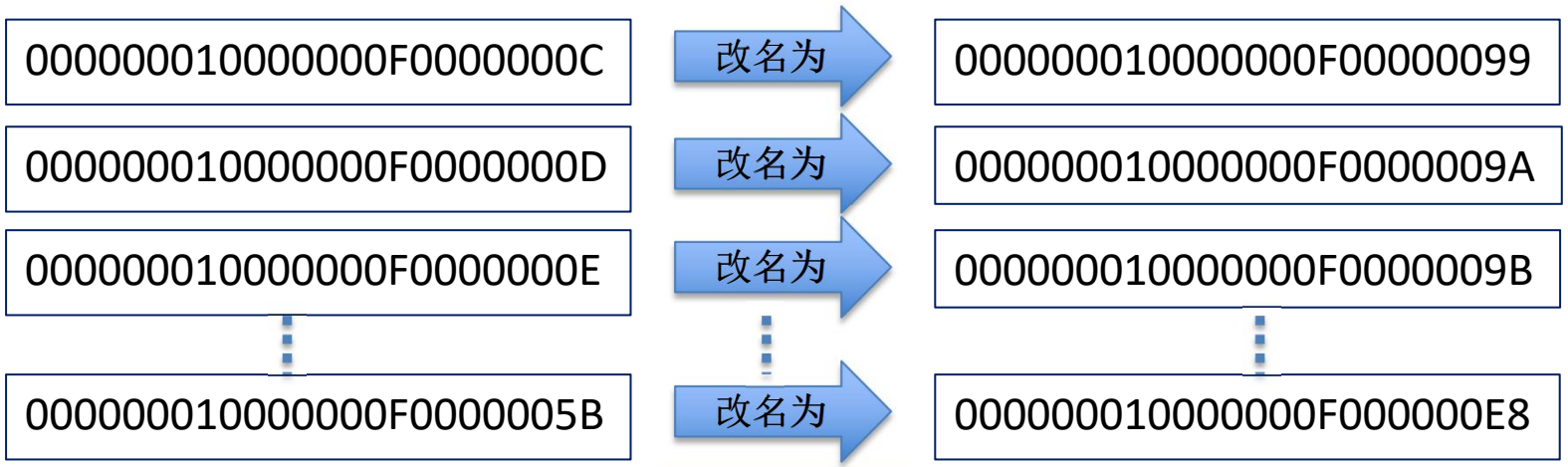
- Oralce循环写Redo log时，当覆盖一个redo log文件时，需要保证这个redo log所代表的脏数据已被checkpoint刷新到数据文件中了。
- PostgreSQL也是通过循环写的方式，覆盖一个WAL日志文件时，也需要把相应的脏数据刷到数据文件中。
- PostgreSQL的WAL日志文件名，看起来是一直增长的，不象是循环写啊？
- 看起来象是新建了一个文件，然后把旧文件删除掉了。

实例恢复： WAL文件的复用机制

- WAL日志复用是通过“重命名”来实现的。
- 当发生一次checkpoint之后，这个checkpoint点之前的WAL日志文件都可以删除掉了，而PostgreSQL并不是删除掉，而是“重命名”这个旧的WAL文件。
- 假设当前目录下的文件：
  000000010000000F0000000C~
  000000010000000F00000098
- 这时发生了一个checkpoint点，checkpiont点所在的文件000000010000000F0000005C

实例恢复： WAL文件的复用机制

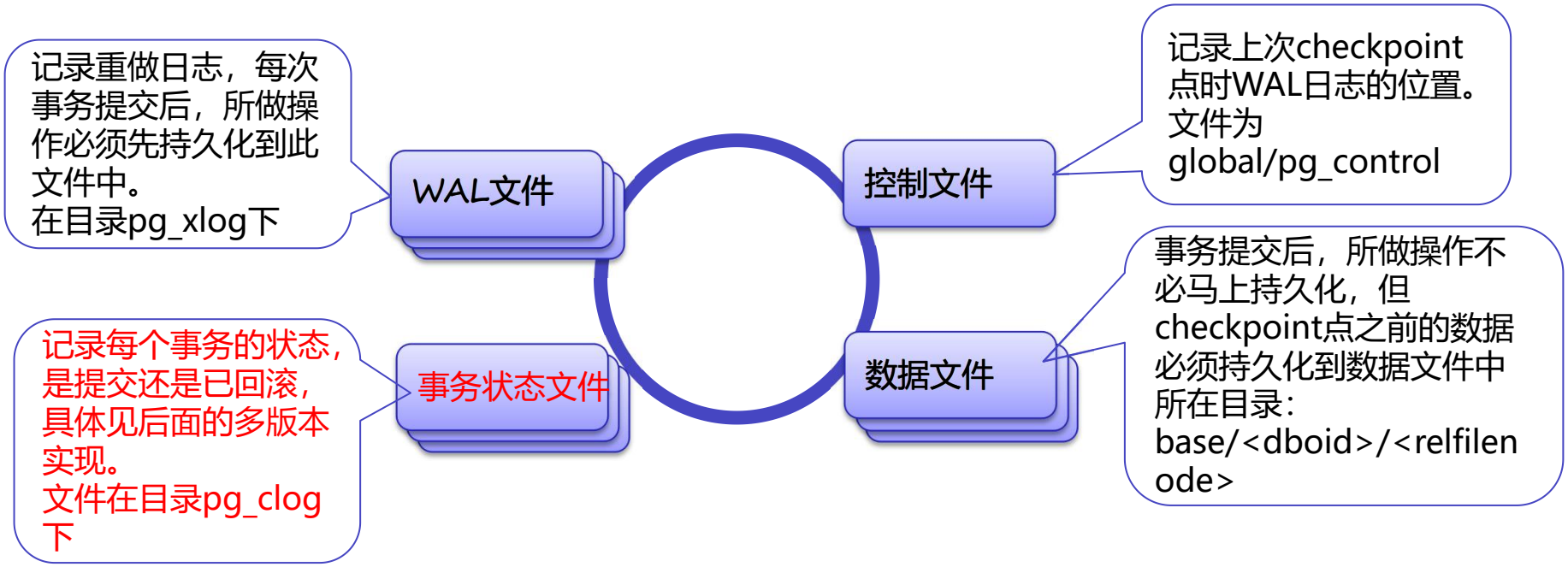
- checkpoint点之前的WAL文件都可以被改名掉，即 00000001000000F0000000C ~00000001000000F0000005B
- 当前最后一个WAL日志文件名为 00000001000000F00000098，所以生成的第一个文件名为： 00000001000000F00000099



实例恢复： WAL文件的复用机制

- 改名用的技巧
  - 并不是用rename，而是使用建新的硬链接，删除旧文件的方法
  - 这种方法更通用，同时也一样可靠
- 重命名WAL，有时也不一定会把checkpoint点之前的WAL文件都重删除掉，还受一些参数的影响：
  - wal_keep_segments
  - hot_standby_feedback
  - replication slots
  - min_wal_size、max_wal_size

实例恢复：回顾


记录重做日志，每次事务提交后，所做操作必须先持久化到此文件中。
在目录pg_xlog下
WAL文件
记录上次checkpoint点时WAL日志的位置。
文件为global/pg_control
控制文件
事务提交后，所做操作不必马上持久化，但checkpoint点之前的数据必须持久化到数据文件中
所在目录：
base/<dboid>/<relfilenode>
数据文件
记录每个事务的状态，是提交还是已回滚，具体见后面的多版本实现。
文件在目录pg_clog下
事务状态文件

实例恢复：Commit Log

- 简称clog
  - 在数据目录的pg_clog，pg_log目录下的文件可以删除，这个目录下的文件千万不能删除
  - 每个事务的状态用两个bit来表示：
    - #define TRANSACTION_STATUS_IN_PROGRESS 0x00
    - #define TRANSACTION_STATUS_COMMITTED 0x01
    - #define TRANSACTION_STATUS_ABORTED 0x02
    - #define TRANSACTION_STATUS_SUB_COMMITTED 0x03

实例恢复：控制文件

```
osdba-mac:~ osdba$ pg_controldata
pg_control version number:            942
Catalog version number:               201409291
Database system identifier:           6211732180664338143
Database cluster state:               in production
pg_control last modified:             Thu Nov  5 10:05:09 2015
Latest checkpoint location:           0/172A568
Prior checkpoint location:            0/172A490
Latest checkpoint's REDO location:    0/172A530
Latest checkpoint's REDO WAL file:    000000010000000000000001
Latest checkpoint's TimeLineID:       1
Latest checkpoint's PrevTimeLineID:   1
Latest checkpoint's full_page_writes: on
Latest checkpoint's NextXID:          0/1013
Latest checkpoint's NextOID:          24590
Latest checkpoint's NextMultiXactId:  1
Latest checkpoint's NextMultiOffset:  0
Latest checkpoint's oldestXID:        990
Latest checkpoint's oldestXID's DB:   1
Latest checkpoint's oldestActiveXID:  1013
Latest checkpoint's oldestMultiXid:   1
Latest checkpoint's oldestMulti's DB: 1
Time of latest checkpoint:            Thu Nov  5 10:05:09 2015
```

实例恢复：控制文件

- 数据库的唯一标识串的秘密
  - Database system identifier: 6197591927813975882
  - initdb时生成的一个64bit的整数，生成过程为：
    - gettimeofday(&tv, NULL);
    - sysidentifier = ((uint64) tv.tv_sec) << 32;
    - sysidentifier |= ((uint64) tv.tv_usec) << 12;
    - sysidentifier |= getpid() & 0xFFF;

实例恢复：控制文件

- 通过下面这条SQL就知道此数据库是什么时候创建的：
  - SELECT to_timestamp(((6197591927813975882>>32) & (2^32 -1)::bigint));

```
postgres=# SELECT to_timestamp(((6197591927813975882>>32) & (2^32 -1)::bigint));
      to_timestamp
------------------------
 2015-09-23 14:21:57+08
(1 row)
```

实例恢复：控制文件

```
Latest checkpoint location:           0/177B780
Prior checkpoint location:            0/177B6A8
Latest checkpoint's REDO location:    0/177B748
Latest checkpoint's REDO WAL file:    000000010000000000000001
Latest checkpoint's TimeLineID:       1
Latest checkpoint's PrevTimeLineID:   1
Latest checkpoint's full_page_writes: on
Latest checkpoint's NextXID:          0/1021
Latest checkpoint's NextOID:          24590
```

为什么有两个last checkpoint?

实例恢复：控制文件

- 实例恢复时，会从上面的哪个checkpoint点开始apply redo日志？
  - checkpoint本身也会在WAL中写一个日志，这条日志的位置为就是“Lastest checkpoint location”
  - 当发生checkpoint点时WAL日志的当前位置为“Latest checkpoint’s REDO location”

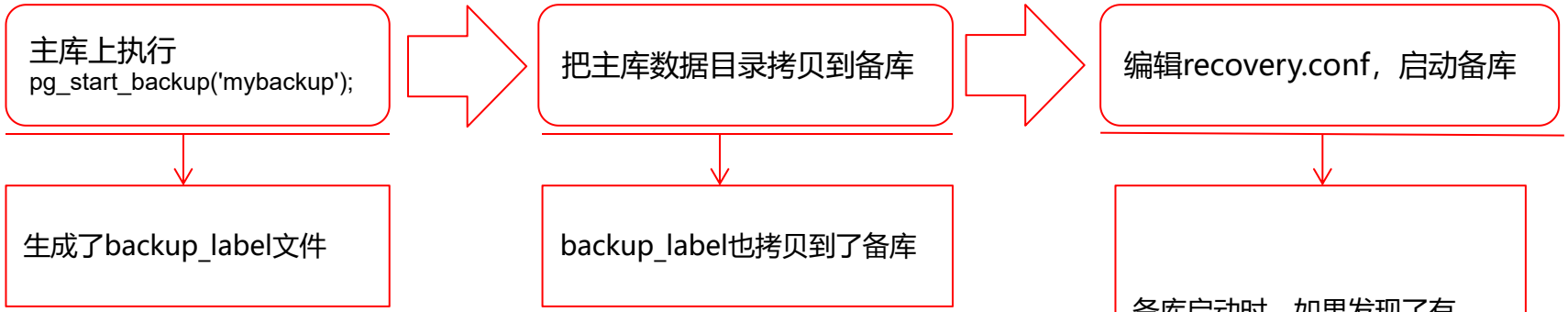
实例恢复：控制文件

- Standby库的控制文件的秘密
  - Standby库的控制文件与主库是否相同？
  - 主库控制文件中的checkpoint信息是否会复制到Standby库？
  - “Minimum recovery ending location”?
- 备库中每replay一些WAL日志后，就会做一次checkpoint点，然后把这个checkpoint点的信息记录到控制文件中。
- 备库replay一些日志后，会把产生脏数据的最新日志的位置记录到“Minimum recovery ending location”。
- 为什么要记录呢？为了Standby库只读的功能。
- 备库异常停机后再启动，需要replay日志到超过“Minimum recovery ending location”位置后，才能对外提供只读服务，或才能激活成主库。

实例恢复：控制文件

- 以下几个参数的意义是什么？
  - Backup start location
  - Backup end location
  - End-of-backup record required

实例恢复：控制文件




```
START WAL LOCATION: 0/4000028 (file 000000010000000000000004)
CHECKPOINT LOCATION: 0/4000060
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2015-09-22 19:44:23 CST
LABEL: osdba201509230923
```

Q&A