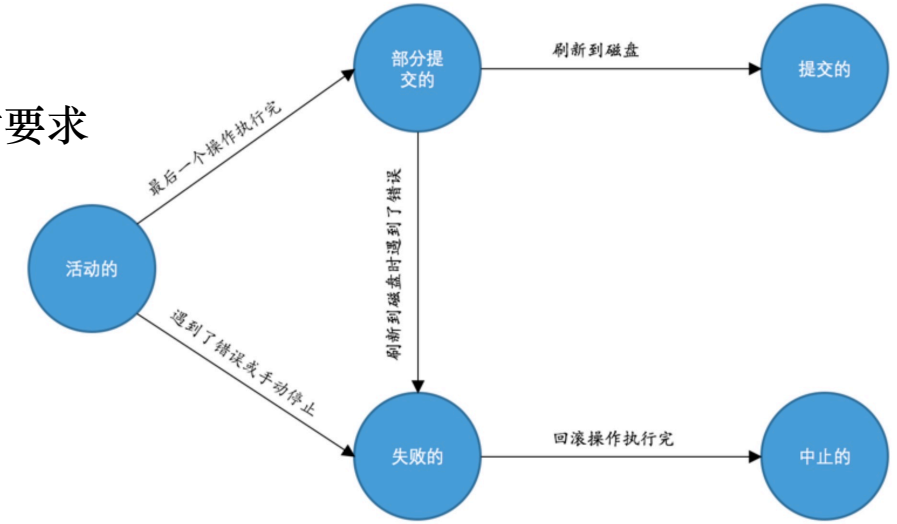
05 事务处理与恢复

Transaction Processing and Recovery

南京大学软件学院

事务处理（复习）

- ACID「HAERDER83」——事务处理必须遵循的原则
- 事务的本质是多个操作一个步骤（操作包括读取和写入数据记录）
  - 原子性（Atomicity）：事务的本质要求
  - 一致性（Consistency）：数据完整的本质要求
    - 定义最弱的属性，也是唯一一个可以由开发者控制而不是仅凭数据库自身保证的属性
  - 隔离性（Isolation）：并发的本质要求
  - 持久性（Durability）：数据库系统的本质要求



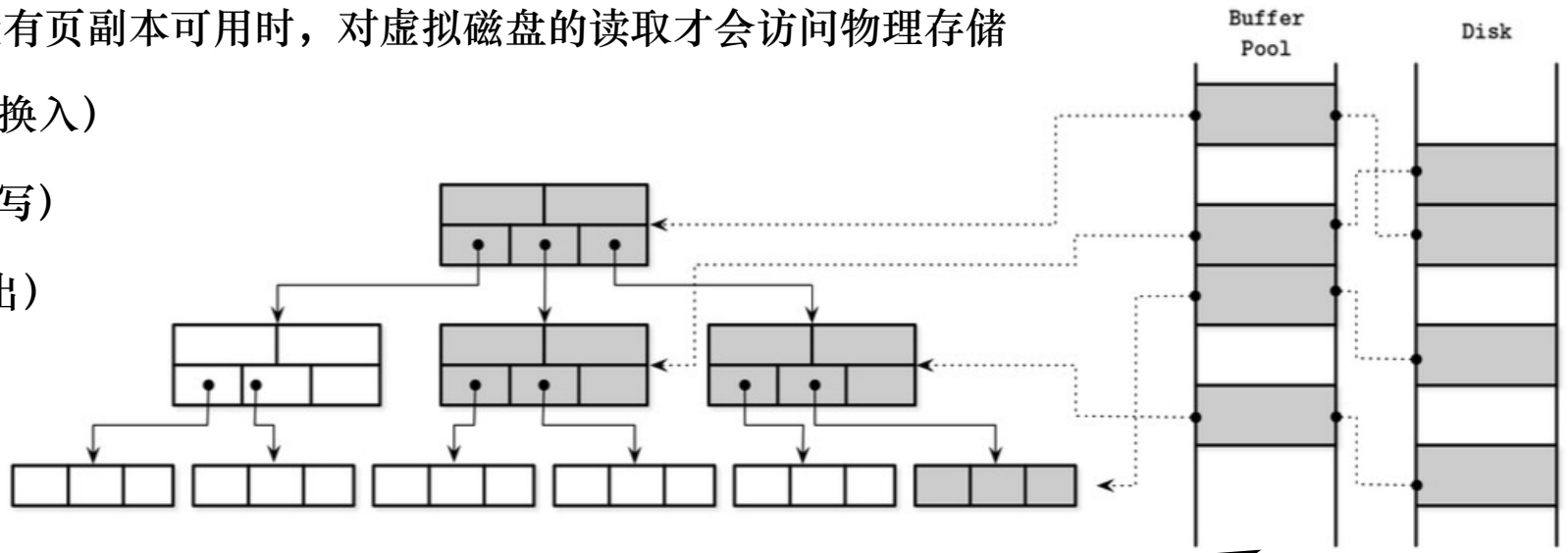
事务处理的要求下，需要哪些组件协同呢？（复习）

- 事务管理器（Transaction manager）
  - 协调、调度和跟踪事务的各个步骤
- 锁管理器（Lock manager）
  - 保护对资源的访问，防止能可能破坏数据完整性的并发访问
- 页缓存（Page cache）
  - 充当持久性存储（磁盘）和存储引擎其余部分之间的中介
- 日志管理器（Log manager）
  - 记录已应用在缓存页上的操作（日志条目），以便撤销已中止的事务所作出的更改
- 分布式事务协调（Distributed/multipatition）

Part 1: 缓冲区管理（Buffer Management）

- 双层存储体系是大多数数据库的基础
- 页缓存（page cache/虚拟磁盘（virtual disk））
  - 当内存中没有页副本可用时，对虚拟磁盘的读取才会访问物理存储
  - Page in （换入）
  - Flush（刷写）
  - Evict（换出）

我个人更喜欢page cache这个词，它更好的反映了这个结构在数据库中真正的目的。
缓冲区是一个实现逻辑，或者说一个有用的功能，而非应用目的。



小问题：这个Buffer在数据库中的结构大概应该是什么样的？（OS复习）

缓冲区管理（Buffer Management）需要完成的功能

- 在内存中保留被缓存的页的内容
- 把对磁盘页的修改缓冲起来，并且修改的是缓存版本
- 当被请求的页不在内存中且可用空间足够，页缓存换入并返回缓存的版本
- 如果请求的页在缓存中，则直接返回缓存的版本
- 如果可用空间不足以放下新页，则换出其他页，被换出的页的内容会刷写回磁盘

未必缓存指操作系统级的内存，很多数据库使用O_DIRECT标志打开文件，允许I/O系统调用绕过内核的页缓存直接访问磁盘，并使用数据库专用的缓冲区管理。

缓存的语义（复习）

- 缓存的修改同步是单向的
  - 对磁盘访问的抽象，并将逻辑写操作与物理写操作分离
  - 页缓存使我们可以将比如树部分保留在内存中，无需修改算法本身或是在内存中物化对象。我们要做的仅仅是把磁盘访问替换成对页缓存的调用
- 请求页的基本步骤
  - 检查该页是否已被缓存，如果该页在缓存中，直接返回缓存的页；
  - 如果没有缓存，则页缓存会将其逻辑地址或页id转化为物理地址，加载到内存，并返回；
  - 一旦返回，这个存有缓存页内容的缓冲区就被称为被引用的（referenced）
  - 用完之后将其归还给页缓存或解除引用
  - 若想让页缓存不要换出某些页，则可以将其固定（pin）
  - 如果某些页被修改，标记为脏页（dirty page），脏页表示内容与磁盘不同步，换出时必须将其刷写到磁盘

缓存的回收（复习）

- 保持缓存填满的状态是不是好事?
- 换出的设计逻辑
  - 检查该页是否与磁盘同步（已经刷写过，或者从未修改）；
  - 并且检查该页是否被固定或引用，直接立即可以将其换出；
  - 脏页需要刷写后被换出；
  - 如果被其他线程使用，则无法换出；

目标的权衡

推迟刷写以减少磁盘访问次数；（尽可能少的中断率）

提早刷写以让页能被快速换出；

选择要换出的页，并以最优的顺序刷写；

将缓存大小保持在内存范围内；

避免因数据没有被持久化而丢失。

每次换出页都刷写磁盘，性能可能会更差，解决方案是？——独立后台进程循环刷写（PostgreSQL的后台刷写器）

持久性：如果数据库崩溃，所有未刷写的数据均会丢失，怎么办?

检查点进程，预写日志（WAL）和页缓存系统工作，当刷写后，丢弃WAL

技术1:在缓存中锁定页

- 根据不同逻辑结构和应用方式，选择特定技术处理
- 堆文件的随机性和B+树结构在业务调用逻辑上存在差异
  - B+树越靠近顶部越窄，所以层次较高的节点在大多数读取中会被命中
  - 同时分裂与合并操作最终也会传播到较高层次的节点
  - 频繁的子树结构变化，可以一起处理
    - 比如：多次删除导致节点合并，随后写入导致分裂
    - 比如：不同子树的合并和分裂操作，共同传播到上层结构

“锁定”大概率会被用到的页，pin。
减少磁盘访问次数并提高性能。

仅通过内存中的应用更改，将操作缓冲到一起，只做一次磁盘写入即可，无须多次写入，减少磁盘写入的次数
But，这就用到了页置换的策略选择。

技术2:页置换策略选择

- 页置换策略选择：本质是估计后续页被访问的可能性
- 页置换算法选择的理想状态：
  - 预测未来的页访问次序，仅换出最长时间内不会用到的页
  - 现实是：请求不一定遵循任何特定的模式或分布，很难精准
  - 但，正确的页置换策略能帮助减少换出次数
  - 等等，如果我就是有钱，使用更大的缓存能否减少换出次数?
    - Bedaly 异常[BEDALY69]
    - 它表明 ，如果使用的页置换算法不是最优的，则增加页数可能导致换出的次数增加

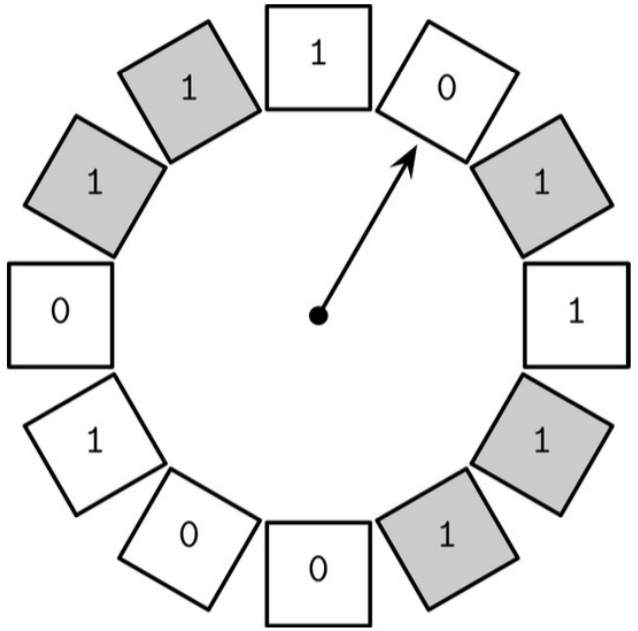
技术2:页置换策略选择（复习）

- FIFO & LRU
  - FIFO的问题是什么？（比如针对B+树）
  - LRU（最长时间未使用策略）——FIFO基础上，重复访问时，放回队列尾部，问题是什么？
  - 2Q的LRU（双队列LRU，k=2）——后续访问移入第二个热队列，从而区分最近访问和经常访问
  - LRU-K，跟踪最近K次访问识别频繁用到的页，并使用此信息估计访问时间（Leetcode146）

大量一次性操作会冲刷缓存，被冲刷的page可能是未来经常访问的page，出现缓存污染问题。

技术2:页置换策略选择（复习）

- CLOCK（Linux）
  - 效率可能比精度更重要
  - CLOCK-sweep算法将页的引用和与之关联的访问位保存在环形缓冲中
    - 访问页面，设置为1（有些变体用的是计数器，不是比特位，来描述频率）
    - 如果访问位为1且该页未被引用，则修改为0，并检查下一页
    - 如果访问页已经是0了，则作为换出候选，并安排在后续换出
    - 如果正在引用，则访问位不变
  - 好处是啥?
    - 算法简单（CAS比较-置换）不需额外加锁
    - 容易理解且易于实现



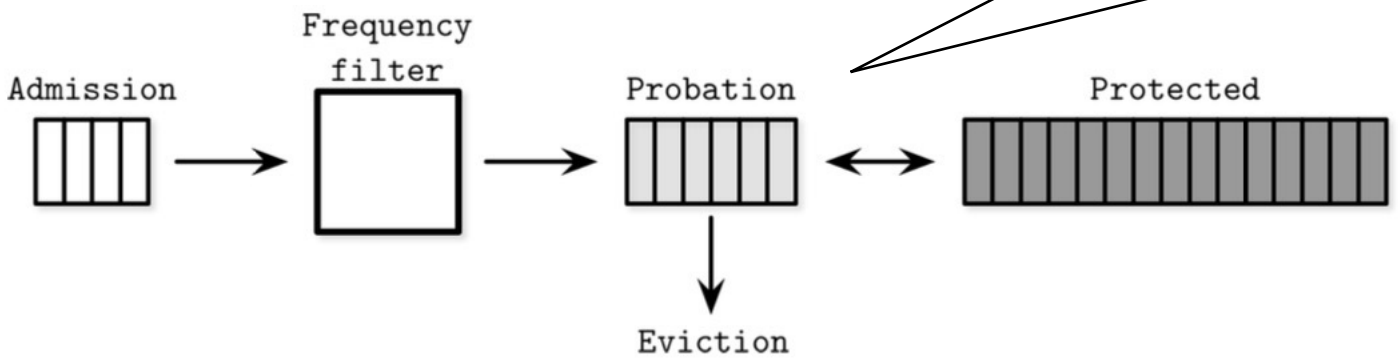
但CLOCK算法的本质仍然是倾向于最近使用时间，而非使用频率作为预测因子。同时对于负载很重的数据库系统来说，最近使用时间不具有参考性。

技术2:页置换策略选择（复习）

好，那么好，复习完了，
你觉得数据库该选哪一种缓存交换策略?

- 目标，以频率替代最近，以追踪页引用事件代替页换入事件
- 代表的算法是最小使用频率（LFU），常用的是TinyLFU策略
  - 不根据页换入的时间来换出页，而用频率排序（java的Caffeine库）
  - 频率直方图维护一个紧凑的缓存访问历史记录（用简化版）
  - 入场（admission）队列：维护新加入元素，使用LRU
  - 考察（probation）队列：最有可能被换出缓存的队列
  - 保护（protected）队列：其中的元素将在队列中保留更长的时间

本质上，TinyLFU策略不是选择要换出哪些元素，而是选择要保留的元素，为的是如何将需要保存的页放入保护队列。



数据库应该采用什么策略?

- “应该”这个词就是错的……可能没有什么是“应该”的，所有的选择都是权衡
- 选择的目标是权衡——尽可能少的“中断率”和足够简单
  - 减少额外访问的开销，以及同时能减少复杂计算的成本
- LRU/LRU-K
  - InnoDB（新页面在队列中间，前5/8是频繁使用的，后3/8是相对最近访问较少的
  - 同时，提供一些配置选项调整算法设计的参数，缓存池大小，随机/线性预读

[1]:《The LRU-K page replacement algorithm for database disk buffering》《Making B+- trees cache conscious in main memory》、《B-tree indexes and CPU caches》
[2]:https://dev.mysql.com/doc/refman/8.3/en/myisam-key-cache.html、https://dev.mysql.com/doc/refman/8.3/en/innodb-buffer-pool.html

Part 2: 恢复（Recovery）

- Buffer Pool 和 Disk 双存储机制必须协同，对于一个已经提交的事务，在事务提交后即便数据库发生崩溃，事务的更改也不应该丢失（咋办？）
- 最简单的方法是——在事务提交完成之前，把该事务所有的页面都刷新到磁盘（问题是？）

1）刷新一个完整的数据页太浪费了，如果只修改了一个字节，却要刷一个完整的 16k 的页面

2）随机I/O 的刷新效率不高

记录日志——“把 #1 表空间的#293 页面的偏移量 23495 的值更新为 2”（Redo 日志）

Redo log（重做日志）

1）占用空间很小

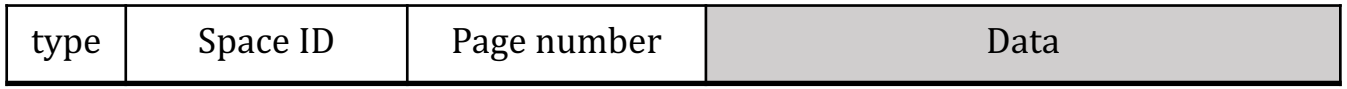
2）顺序写入磁盘（顺序 I/O）

redo日志的设计

- 静态结构
  - redo记录的结构
  - redo日志页需不需要设置大小？要不要比正常的Page大或者小？
  - redo日志是顺序组织还是随机组织？
- 动态结构
  - 事务的原子性如何保证？冲突如何控制？
  - redo日志也是双存储结构的，应该如何刷写？
- 维护实现
  - 如何循环使用redo？

Redo 日志格式

- 本质是记录事务对数据库物理上的修改，有没有通用格式?



Type：日志类型（MySQL 共 53 种）

Space ID：表空间 ID

Page number：页 ID

Data：日志的就内容

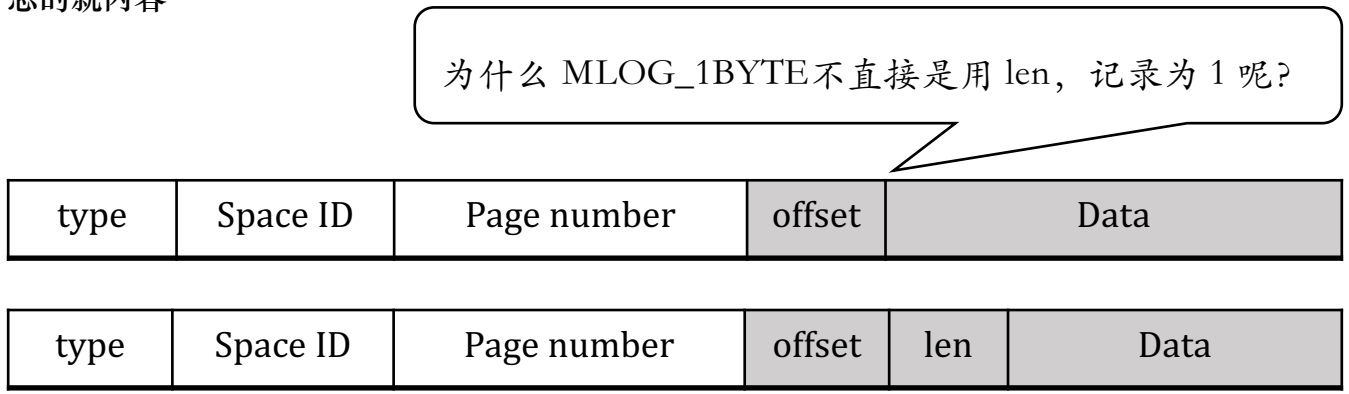
MLOG_1BYTE：页面某个偏移量处写入 1 个字节的 redo 类型

MLOG_2BYTE：页面某个偏移量处写入 2 个字节的 redo 类型

MLOG_4BYTE：页面某个偏移量处写入 4 个字节的 redo 类型

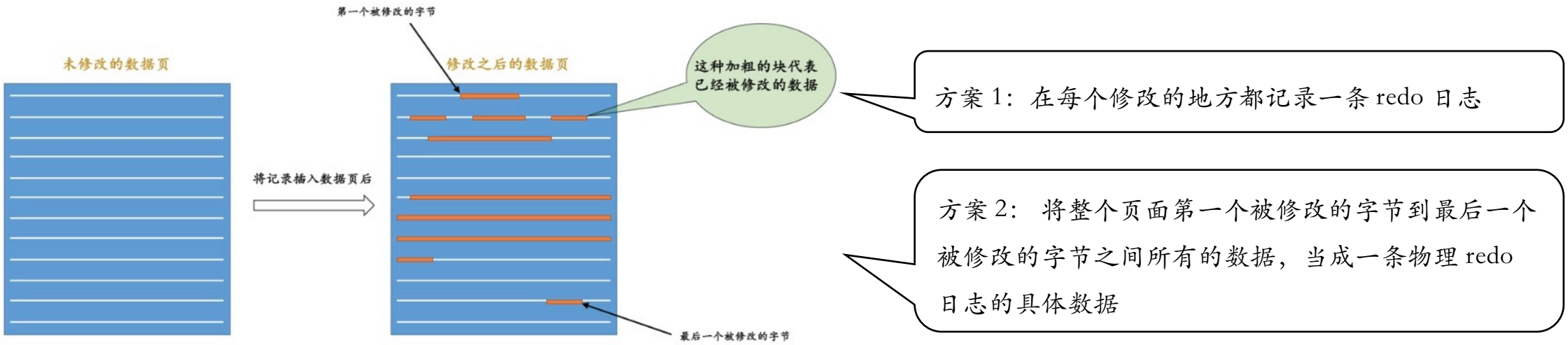
MLOG_8BYTE：页面某个偏移量处写入 8 个字节的 redo 类型

MLOG_WRITE_STRING：页面某个偏移量处写入字节序列的 redo 类型



Redo 日志复杂一些的格式

- 一条SQL 语句会修改很多地方（数据页面、聚簇索引页面、二级索引等等）
  - 表中有多少索引，一条 INSERT语句可能更新多少个 B+树
  - 对某一棵 B+树来说，可能又更新页节点页面，又分裂合并更新到内部节点或新页节点
  - 还有 File Header 、 Page Header 、 Page Directory也要修改……



Redo 日志复杂一些的格式

- 有没有另一种记录日志的可能性?
  - 只记录操作，不记录物理变化
  - 物理层面的记录：指明那个表空间，哪个页被修改了
  - 逻辑层面的记录：记录操作，系统崩溃后，重新执行这个操作

MLOG_REC_INSERT： 表示插入一条使用非紧凑行格式的记录时的 redo 日志类型

MLOG_COMP_REC_INSERT： 表示插入一条使用紧凑行格式的记录时的 redo 日志类型

MLOG_COMP_PAGE_CREATE： 表示创建一个存储紧凑行格式记录的页 面的 redo 日志类型

MLOG_COMP_REC_DELETE：表示删除一条使用紧凑行格式记录的 redo 日志类型

MLOG_COMP_LIST_START_DELETE： 表示从某条给定记录开始删除页面中的一系列使用紧凑行格式记录的 redo 日志类型。

MLOG_COMP_LIST_END_DELETE： 与 MLOG_COMP_LIST_START_DELETE 类型的 redo 日志呼应，表示删除一系列记录直到 MLOG_COMP_LIST_END_DELETE 类型的 redo 日志对应的记 录为止

MLOG_ZIP_PAGE_COMPRESS （ type 字段对应的十进制数字为 51 ）： 表示压缩一个数据页的 redo 日志类 型。

数据页中的记录是按照索引列大小的顺序组成单向 链表的。有时候我们会有删除索引列的值在某个区间范围内的所有记录的需求，这时候如果我们每删除 一条记录就写一条redo日志的话，效率可能有点低，所以提出MLOG_COMP_LIST_START_DELETE和MLOG_COMP_LIST_END_DELETE类型的redo日志，可以很大程度上减少redo日志的条数。

Redo 日志复杂一些的格式

- 物理层面和逻辑层面的融合日志格式
  - 记录本页面插入一条记录的必备物理要素记录下来
  - 但不去管 Page 这一层面的系统值
  - 日志本质上是调用恢复函数的参数组

Redo 日志格式不同数据库有不同的定义，但整体的分类就这三种：

1）记录具体位置的物理修改

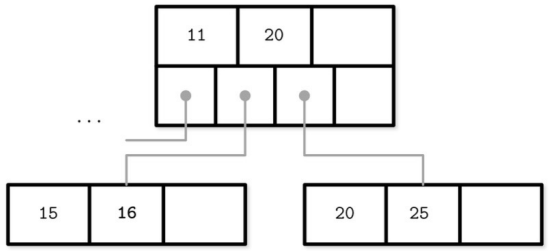
2）记录一个 page 的全部修改

3）记录操作（执行恢复的参数）

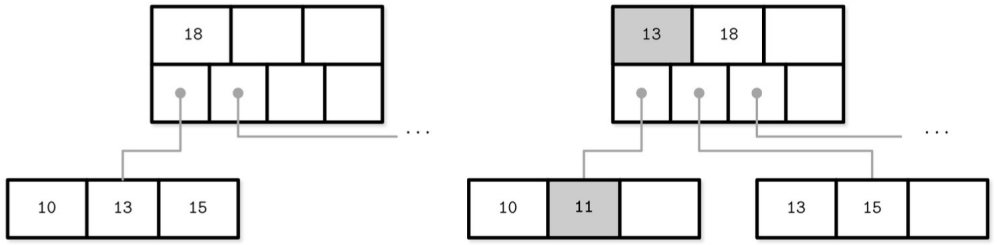
一些数据库也会做一些压缩操作，比如 space id，page number 等

Mini-Transaction

- 以组的形式写入 redo 日志
  - 一组操作，一组日志的不可分割性
  - 索引、基本表、聚簇、二级索引、目录等多个页的操作
  - 插入一条记录的操作本质是原子的，那么问题是——怎么保证原子性?



乐观插入



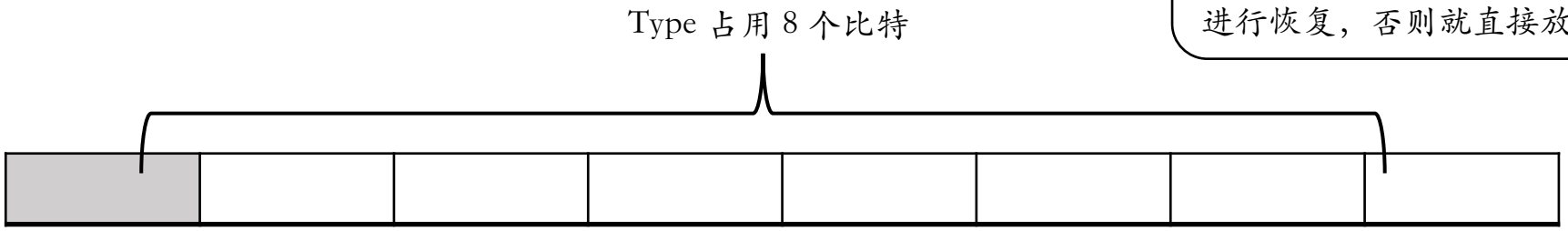
悲观插入

Mini-Transaction

- 增加一个特殊类型的 redo 日志（MySQL）
  - MLOG_MULTI_REC_END，结构简单，只有一个 type 字段（对应十进制 31）

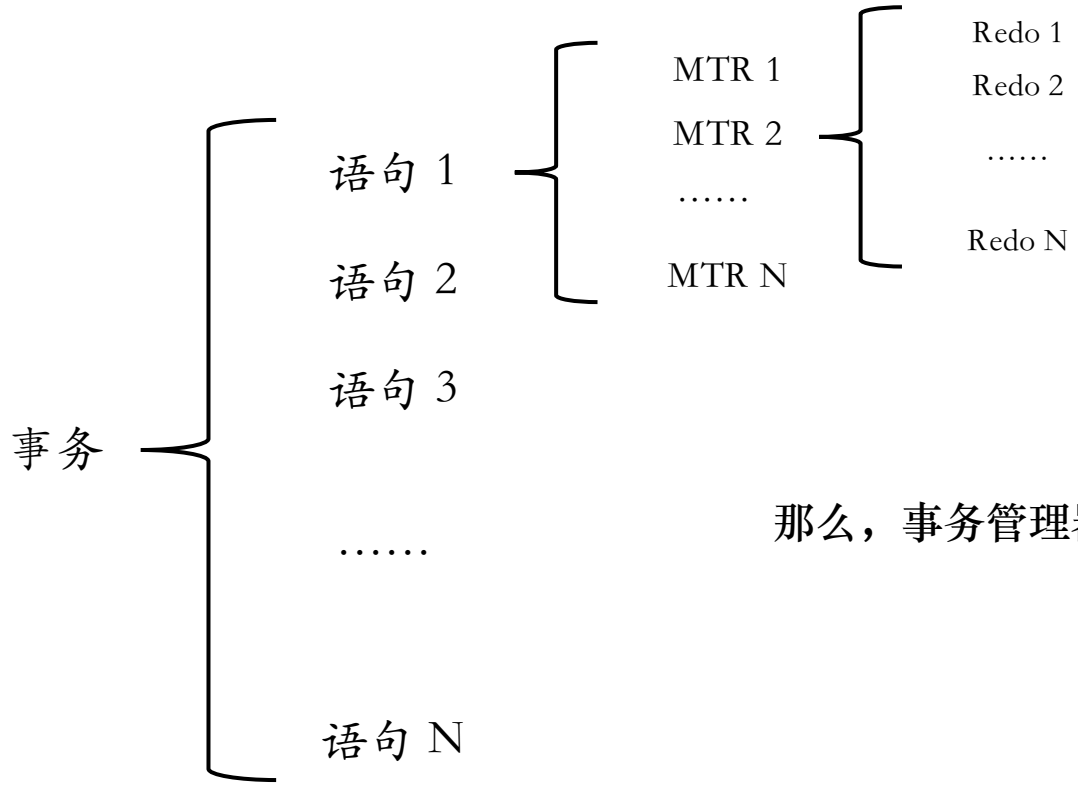
系统恢复重启的时候，只有解析到这个类型的 rodo 日志时，才认为解析到了一组完整的 redo 日志才会进行恢复，否则就直接放弃前面解析到的 redo 日志

  - 如果只有一条日志的怎么办？（比如更新 Max Row ID 的操作）



第一位表示是否为单一日志，后7 位代表 redo 日志类型（MySQL）

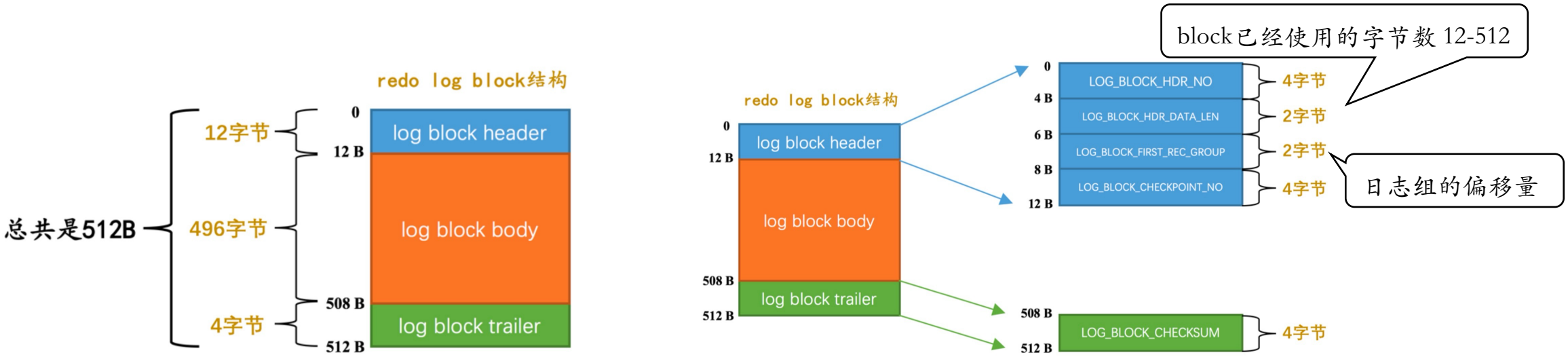
Mini-Transaction



那么，事务管理器该怎么做?

redo log block

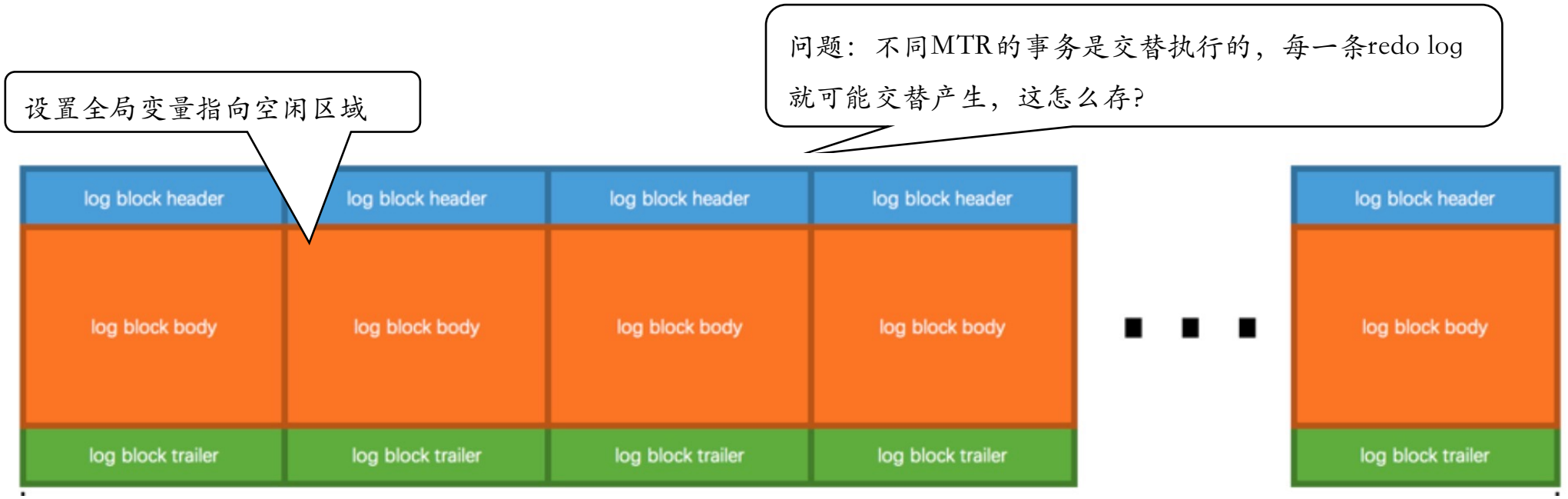
- redo日志的page大小往往比正常page大（block，MySQL 512K）
- 结构和page几乎是一样的



MySQL的redo log block

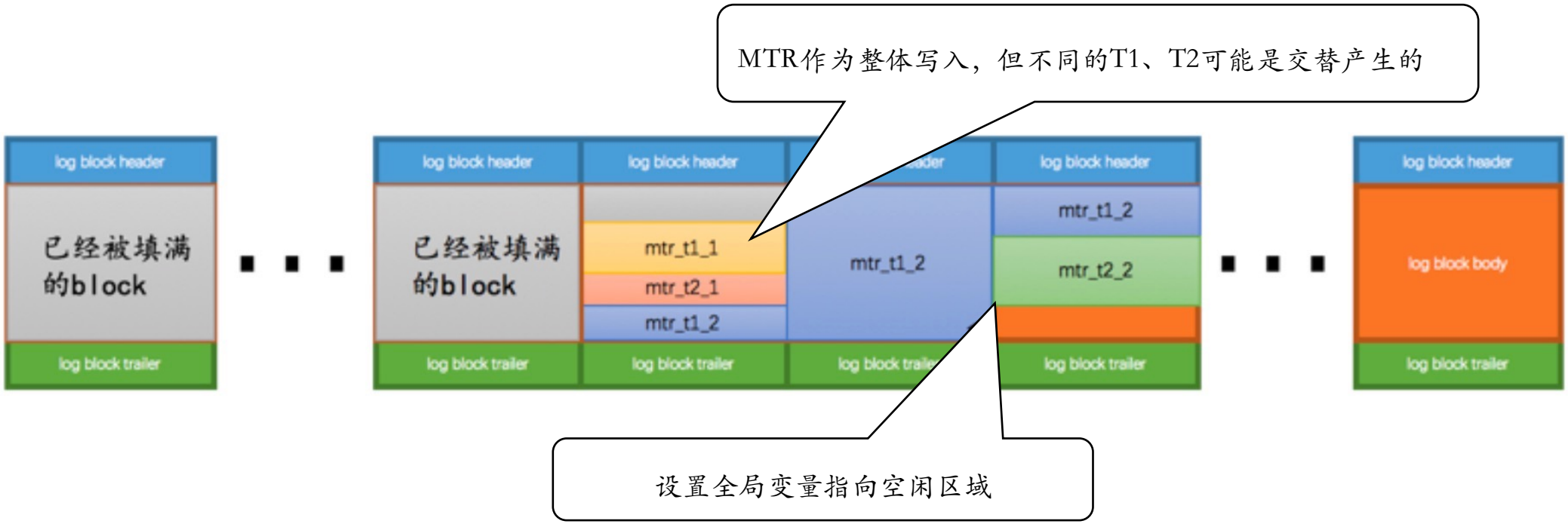
redo日志缓冲（redo log Buffer Pool）

- redo日志依旧是双存储结构，有独立的Buffer，由若干连续redo log block组成
- 顺序写入，速度是最快的



MySQL的redo log Buffer Pool

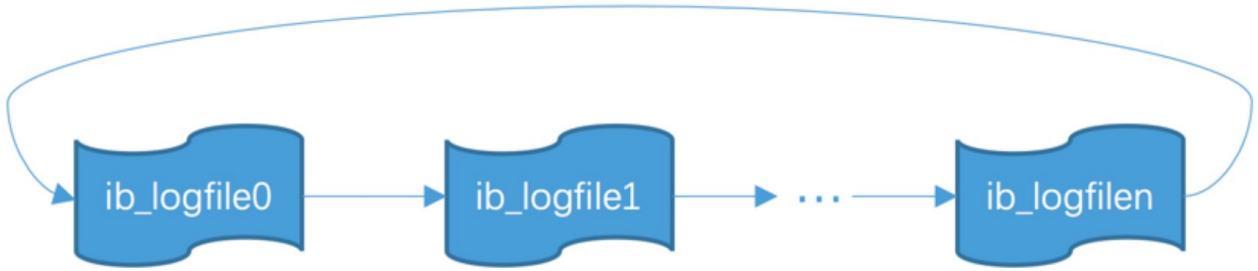
redo日志缓冲（redo log Buffer Pool）



MySQL的redo log Buffer Pool

redo 日志文件（redo log file）

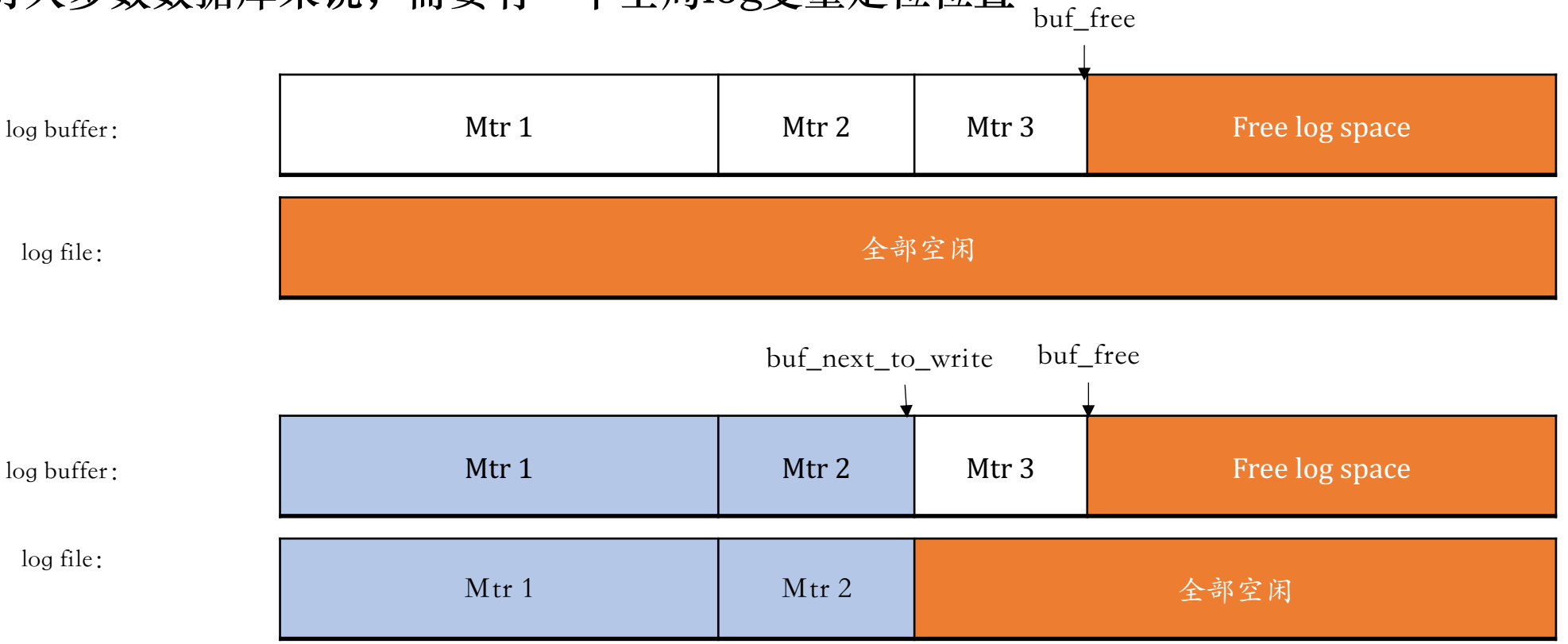
- redo log 的刷盘时机
  - log buffer空间不足（50%的阈值）
  - 事务提交时
  - 脏页刷新
  - 定时进程，固定频率刷新（1s，log buffer中的redo log刷新到硬盘）
  - 正常关闭服务器
- 硬盘中日志文件
  - 数量和大小（2-100，48M）
  - 循环写入



lsn值（log sequence number）

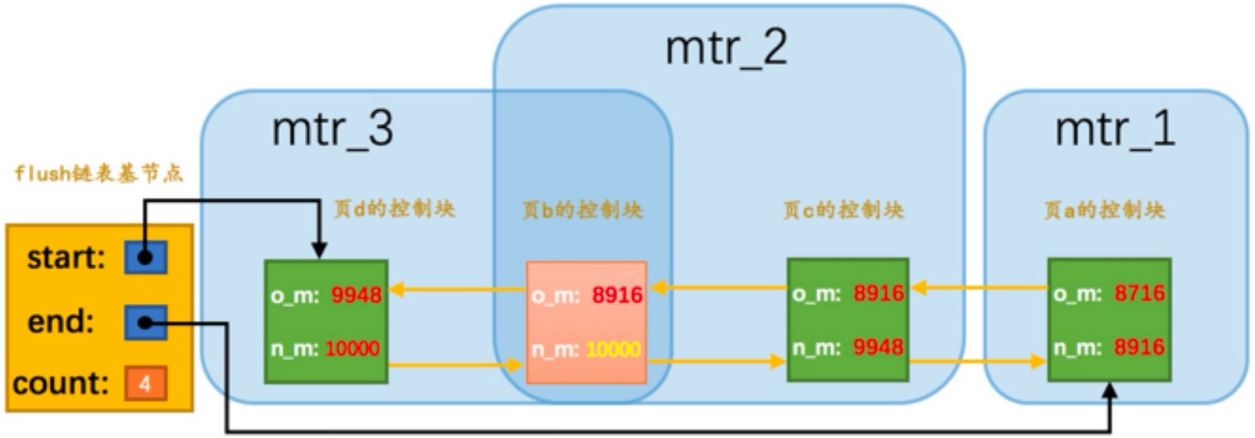
- 总是要记录那些log已经刷写了，哪些log尚未刷写
- 对大多数数据库来说，需要有一个全局log变量定位位置

记录block头尾和log的大小，包括跨页，这样可以完整计算所有的偏移量，而不需要记录block的id+偏移量，因为顺序插入



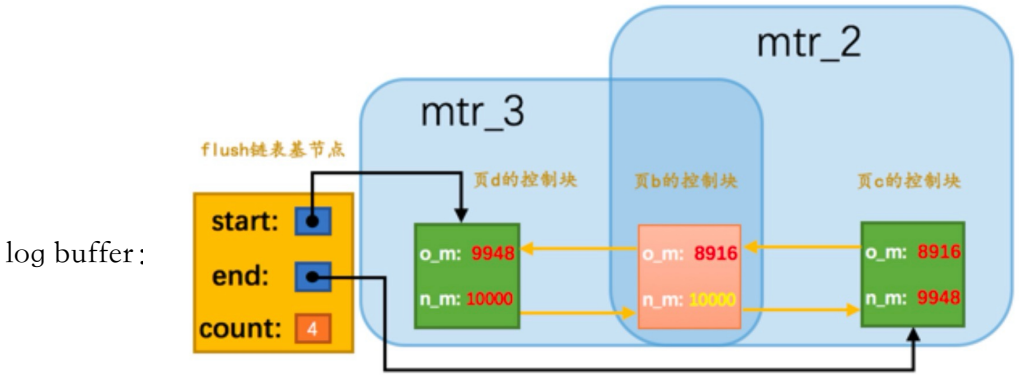
check point

- redo日志组容量是有限的，不得不循环使用redo日志文件组的文件
- 但，问题是——我怎么知道两件事
  - buffer中，哪个日志组已经刷写到硬盘了?
  - 日志文件中，哪个日志组所涉及的操作已经刷写到数据文件中了?

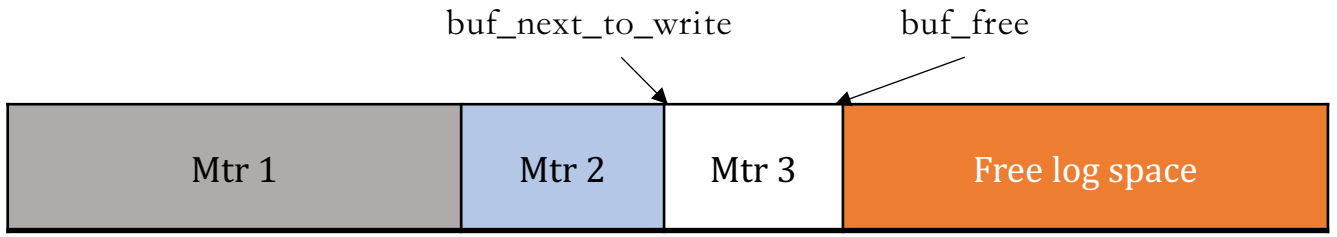


check point的步骤

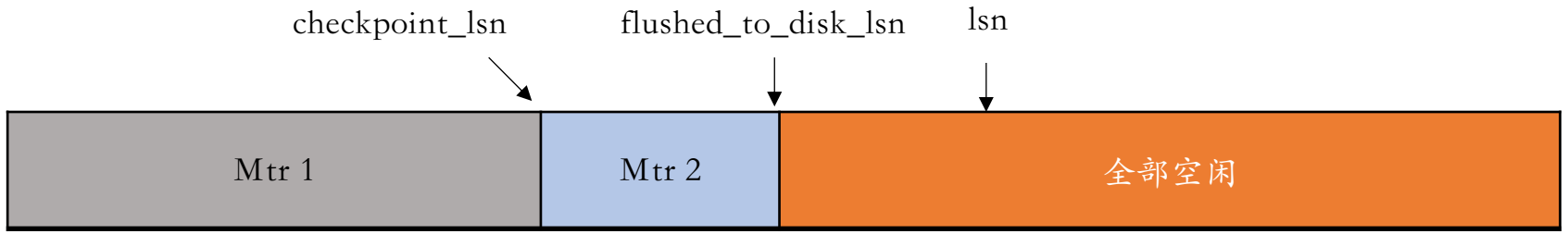
- 计算当前系统可以被覆盖的redo日志对应的lsn值最大是多少
- 将信息写入日志文件的管理信息中，记录check point的操作



log buffer：



log file：

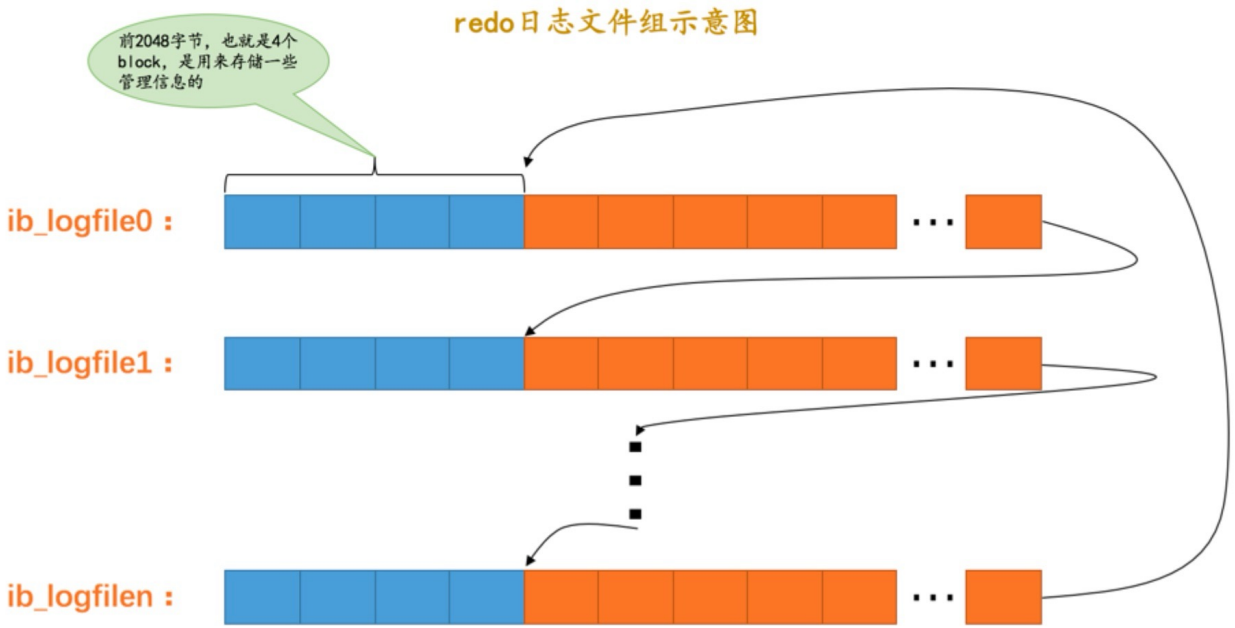


check point

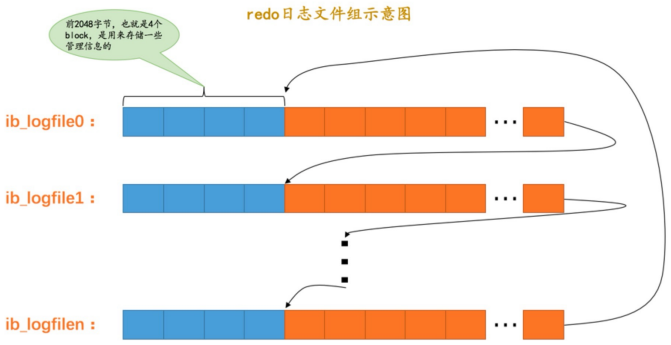
- 理想状况，check point记录一次就行，但如果更新过快，可能需要多次记录
- check point每执行一次，均要修改redo日志文件的管理信息
- 后台刷脏操作和check point操作是两个并行的操作
  - 修改页面非常频繁，导致lsn快速增长，无法及时做checkpoint，则线程做刷脏操作
  - 定时完成check point操作

redo日志文件格式

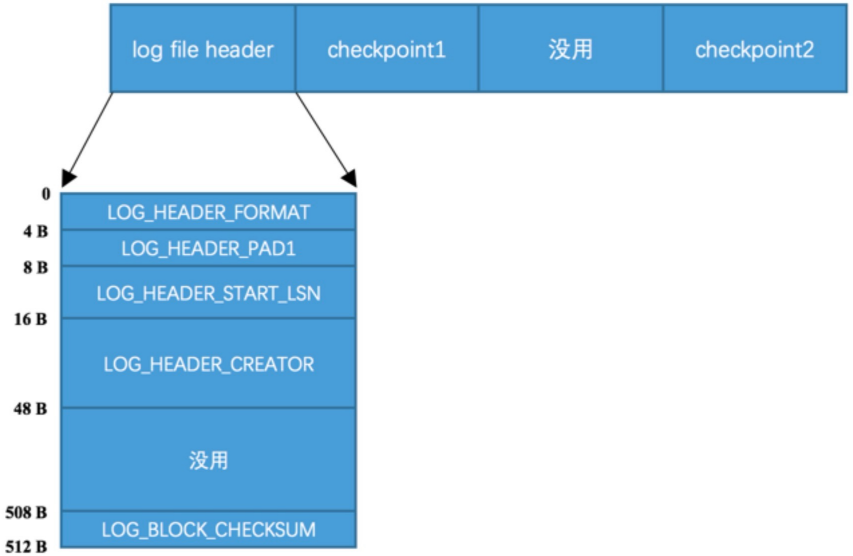
- log buffer本质上是一片连续的内存空间，被划分为若干个512k大小的block
- redo日志刷新到磁盘是把block的镜像写入日志文件，文件也是若干个512k的block

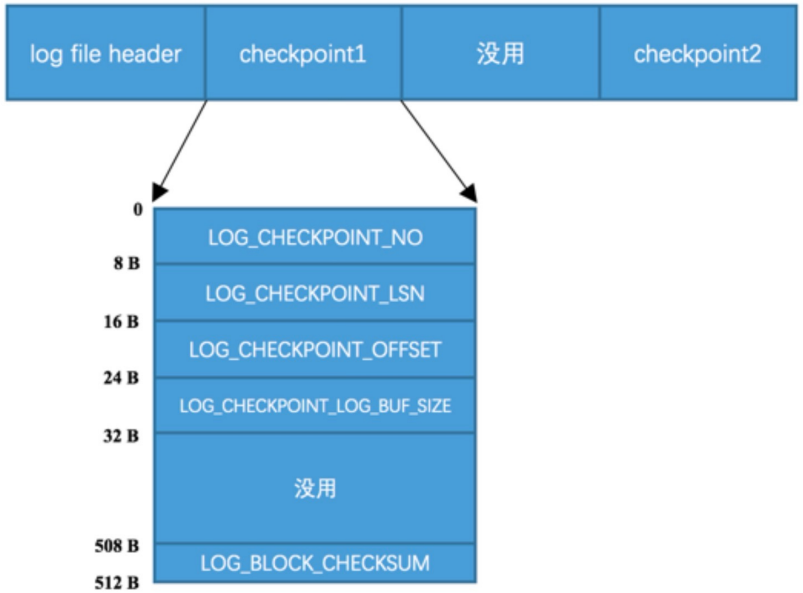


redo日志文件格式


redo日志文件组示意图
前2048字节，也就是4个block，是用来存储一些管理信息的
ib_logfile0：
ib_logfile1：
ib_logfilen：


log file header
checkpoint1
没用
checkpoint2


log file header
checkpoint1
没用
checkpoint2
0
4 B
8 B
16 B
48 B
508 B
512 B
LOG_HEADER_FORMAT
LOG_HEADER_PAD1
LOG_HEADER_START_LSN
LOG_HEADER_CREATOR
没用
LOG_BLOCK_CHECKSUM

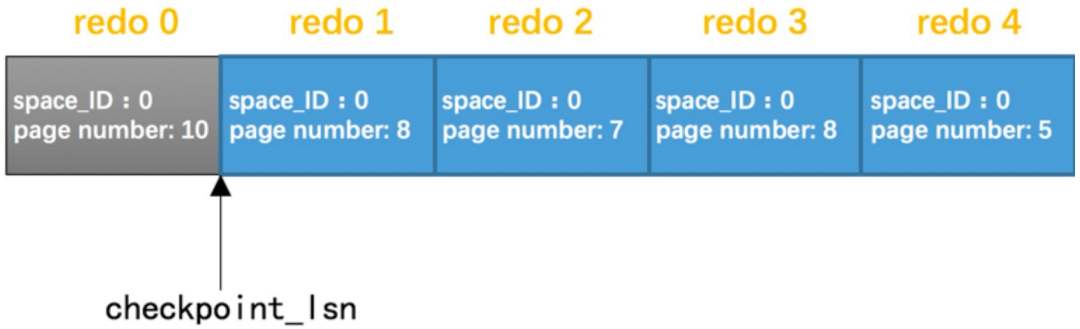

log file header
checkpoint1
没用
checkpoint2
0
8 B
16 B
24 B
32 B
508 B
512 B
LOG_CHECKPOINT_NO
LOG_CHECKPOINT_LSN
LOG_CHECKPOINT_OFFSET
LOG_CHECKPOINT_LOG_BUF_SIZE
没用
LOG_BLOCK_CHECKSUM

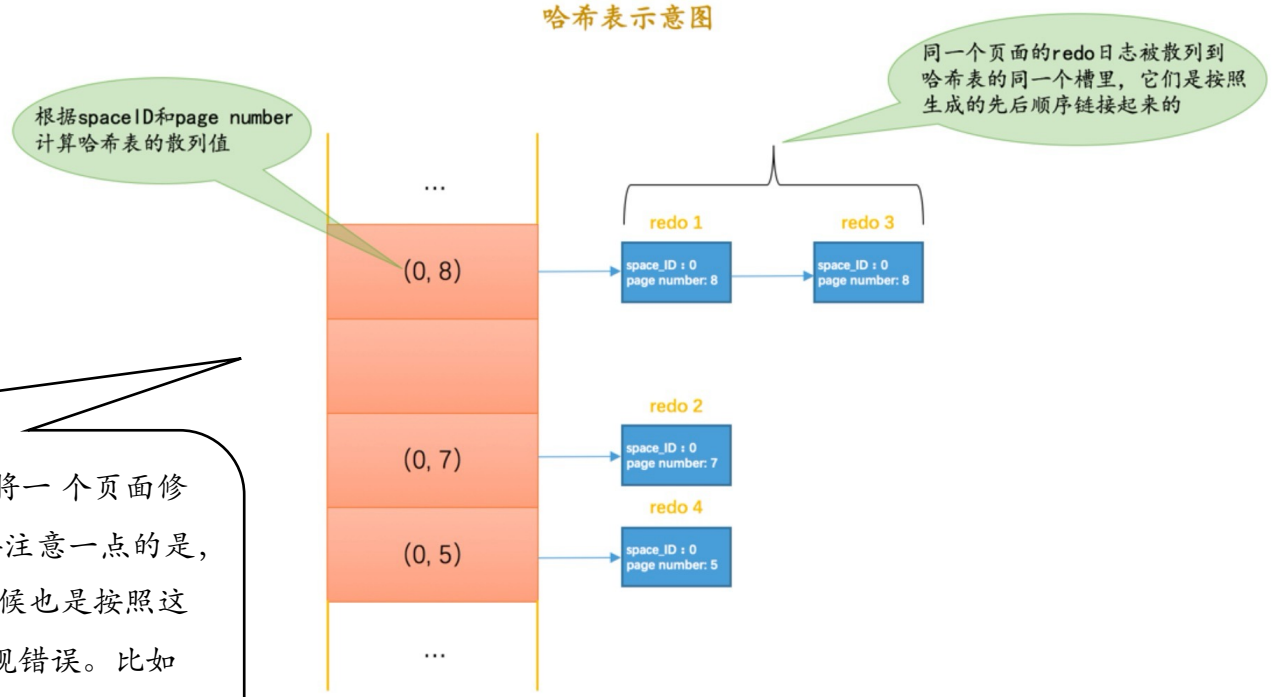
恢复（Recovery）

- 确定恢复的起点（checkpoint_lsn）
  - 选取最近发生的那次checkpoint的信息，checkpoint_no比较一下大小
  - 然后找checkpoint_lsn以及checkpoint_offset
- 确定恢复的终点（log中记录了每个block的字节空间，找没满的那个）

怎么恢复（Recovery）

- 按照日志的内容依次扫描，checkpoint_lsn之后的redo日志，进行页面恢复

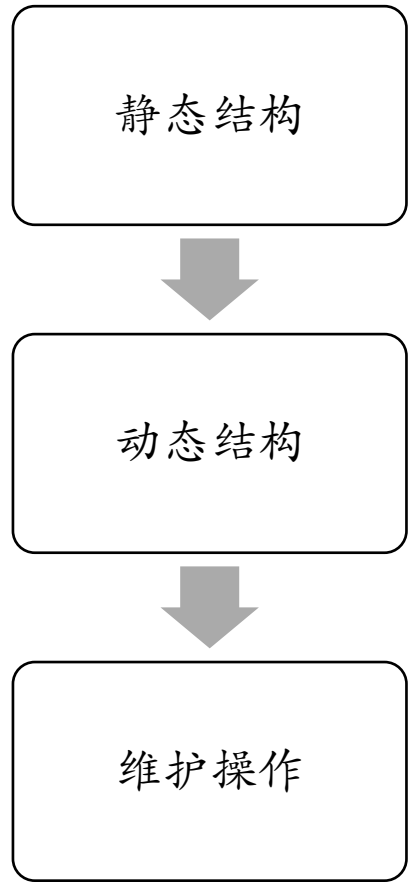




对同一个页面进行修改的 redo 日志都放在了一个槽里，所以可以一次性将一 个页面修复好(避免了很多读取页面的随机IO)，这样可以加快恢复速度。另外需要注意一点的是，同一个 页面的 redo 日志是按照生成时间顺序进行排序的，所以恢复的时候也是按照这个顺序进行恢复，如果不按 照生成时间顺序进行排序的话，那么可能出现错误。比如原先的修改操作是先插入一条记录，再删除该条记 录，如果恢复时不按照这个顺序来，就可能变成先删除一条记录，再插入一条记录，这显然是错误的。

Undo日志（Undo log）

- 事务保证原子性靠的就是日志
  - 错误：服务器、操作系统、断电
  - 手动或自动rollback
- 对每一条记录进行改动的时候，需要留一手
  - INSERT，记录主键，rollback就删除主键
  - DELECT，记录内容，rollback恢复记录
  - UPDATE，记录内容，rollback恢复记录



各位同学有意识地可以学习手边具体数据库的undo log的结构和实现

更一般的恢复（Recovery）小结

- 预写日志（Write-Ahead Log，WAL，也叫提交日志 commit log）
  - 在允许页缓存将页的修改缓存的同时，保证数据库系统的持久性语义
  - 受操作影响的缓存页同步磁盘之前，将操作持久化，先写日志，再修改页的内容
  - 当发生崩溃时，系统可以从操作日志中重建内存中丢失的更改
- 这也是事务处理的基本保障，重放日志恢复为提交数据

每次换出页都刷写磁盘，性能可能会更差，解决方案是?

——独立后台进程循环刷写（PostgreSQL的后台刷写器）以及定期checkpoint刷写

日志语义

- 预写日志是原地更新还是仅追加?
  - WAL是仅追加的，已写入内容是不可变的，顺序加入
  - 安全访问写入边界之前的内容，并在日志尾部增加新增日志数据
- 日志的语义要求
  - 日志序列号（LSN），唯一，单调递增，内部计数器或者是时间戳
  - 强制刷盘操作、事务管理器、页缓存触发
  - WAL会保存事务完成的记录，只有事务提交记录完成刷盘之后，才视为“已提交”

事务完成的标志，不同数据库，同一个数据库的不同设置，均会不同。

日志语义

我们记录的是操作，redo和undo，涉及到大量数据问题，性能如何提升呢?

- 事务回滚或恢复期间会不会发生崩溃?
  - 某些数据库会在撤销操作时，同时开始记录补偿日志记录（CLR）并存储在日志中
- 存在内存中的日志什么时候持久化且不再需要?
  - 检查点的修剪接口（trim）
  - 强制将脏页刷写到磁盘上的过程叫同步检查点（sync checkpoint）——完全同步了主存储结构
- 本质上要确定一旦出现故障的恢复点——模糊检查点（fuzzy checkpoint）
  - 把全部内容刷写到磁盘上，需要暂停所有正在运行的操作，直到检查点完成
  - 日志头部：last_checkpoint记录最后一次成功的检查点信息；
  - 模糊检查点以特殊的（begin_checkpoint, end_checkpoint）包含脏页信息和事务表内容；

操作日志&数据日志

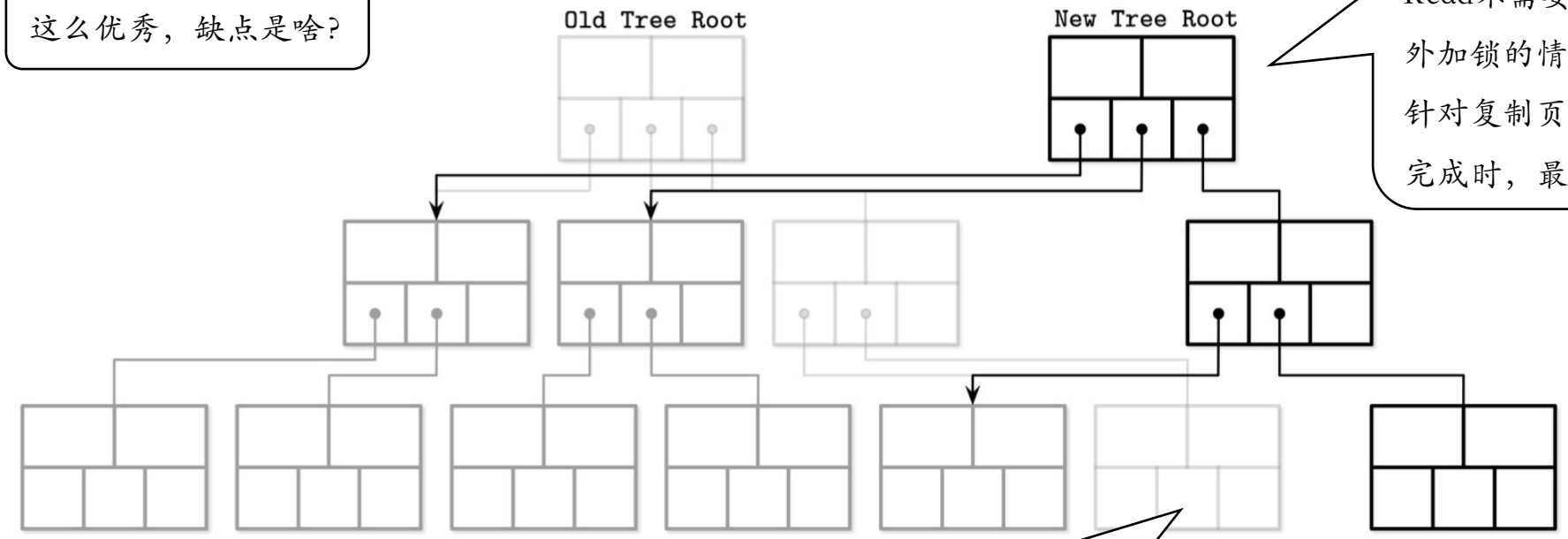
- 影子页（shadow paging）——写时复制（copy-on-write）
  - 新更改的内容被存放在一个新的、未发布的影子页
  - 并通过指针翻转使其可见，从旧页切换到包含更新内容的新页
- 数据日志（保存对完整页状态或字节级的更改，物理日志）
- 操作日志（保存在当前状态上执行的操作）

空间换时间，是数据库永恒的主题

好，那么这两种日志技术怎么应用在redo和undo日志中？

写时复制（copy-on-write）

这么优秀，缺点是啥？



Read不需要同步，可以在不需要额外加锁的情况下被访问。Write 是针对复制页，只有当所有页的修改完成时，最顶端的指针才会切换

旧版本会保留很短的时间（并发事务操作完成后，页就会被收回，）因为 B 树非常浅，所以有一定的优势

Redo log & Undo log

- 前像（before-image）和后像（after-image）的相互转换
- Undo：一个事务在执行过程中，还未提交，发生崩溃或者需要回滚
  - Undo log 撤销回滚的日志，记录更新前的数据到undo日志文件中
  - Undo日志记录的是**操作记录**，插入记录主键、删除记录内容、更新记录旧值
  - Undo日志只在乎“操作之前”（roll_pointer指针串成链表/**版本表**，trx_id事务id）
- Redo：掉电，磁盘I/O崩了，之前提交的记录如何保存（crash-safe 奔溃恢复）
  - 事务提交时，未必检查点同步，事务提交成功的标记是——redo日志持久化了
  - Redo日志记录的是**物理修改**，（xxx数据页yyy的偏移量做了zzz的修改/影子页）
  - 循环写，不用于备份恢复、主从复制，用于掉电等故障恢复——binlog用于全局备份
- **被修改的Undo log本身，也会记录Redo log**

执行一条sql语句是否自动提交，看autocommit参数

保障原子性、实现MVCC（多版本并发控制）

保障持久性

steal和force策略

- steal/no-steal；force/no-force（策略用于页缓存，但对恢复有显著影响）
- steal 策略：事务提交之前允许刷写事务修改过的页
  - no-steal：不允许将未提交的事务内容刷写到磁盘
  - steal——将脏页偷窃出来，写到磁盘上
- force策略：事务提交前将事务修改的所有页刷写到磁盘上
  - no-force：即使事务修改的某些页尚未刷写到磁盘上，该事务也可以提交
  - force——强制脏页在事务提交前必须刷写到磁盘上
- 更一般地说，直到事务提交前，我们都需要保存有足够的信息来撤销它的结果
  - 如果事务涉及的页已经刷写到磁盘，我们需要在log中保留撤销信息（undo log），确保提交前可以被回滚
  - 否则，我们必须保留redo日志（只有undo和redo都写入日志文件，事务才能被提交）

no-steal策略，只需undo log就可以实现恢复，但并发性不够高

奔溃恢复无须任何工作即可重建已提交事务的结果，
缺点，事务提交必须I/O操作，提交时间更长

no-force策略，推迟对页的若干更新来缓冲它们，
需要更大的页缓冲

steal和force策略

	force	No-force
steal	Undo/No-Redo	Undo + Redo (performance：fastest Recover：slowest)
No-steal	No-Undo/no-Redo (performance：slowest Recovery：fastest)	Redo/No-Undo

通常数据库采用的方式

面向磁盘的数据库来说，不可用

PostgreSQL有redo log，却没有undo log，事务回滚不需要undo，而直接使用类似shadow paging方式（tuple）。

MySQL和Oracle使用的是Delta Storage，把差异变化记录在Delta Storage，形成一个链，称之为回滚段（rollback segment）

ARIES（steal/no-force的恢复算法）

- 物理redo日志（提高恢复性能），逻辑undo日志（提高并发性能）
  - 使用WAL来实现恢复时重访历史，完整重建数据库状态
  - 未提交事务的修改已被撤销，撤销期间构建补偿日志记录
- 当数据库崩溃后重启，恢复过程分为三个阶段
  - 分析阶段：识别页缓存中的脏页，以及崩溃时正在进行的事务，脏页信息用于标识重做阶段的起点。进行事务的清单用于在撤销阶段中回滚未完成的事务；
  - 重做阶段：重放历史记录直到崩溃点，并将数据库恢复到先前的状态，此阶段会处理未完成的事务，以及哪些已经提交但尚未将修改刷写到持久化存储的事务
  - 撤销阶段：回滚所有未完成的事务，并将数据库还原到最后的一致状态。所有操作均按反向时间顺序回滚，为了防止数据库在恢复过程中再次崩溃，撤销事务所做的操作也会被记录到日志中

Mohan, C., Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging." *Transactions on Database Sys - tems* 17, no. 1 (March): 94-162. *https://doi.org/10.1145/128765.128770*.

Part 3 并发控制

- 事务管理器和锁管理器协同工作以处理并发控制（复习）
- 乐观并发控制（OCC）
  - 允许多个事务执行并发的读取和写入，最后确定其执行结果能否被串行化，事务不会被阻塞，而是保留历史操作，并在提交前检查这些历史操作是否存在冲突的可能。

为什么叫“乐观”

- 多版本并发控制（MVCC）
  - 允许一条记录同时存在多个时间戳的版本，通过这种方式保证事务独到的是数据库过去某个时刻的一致视图。 MVCC 可以是用验证技术来实现（只允许多个更新或事务提交中的某一个获胜，也可以采用无锁机制或者基于二段锁机制。
- 悲观并发控制（PCC）
  - 严格管理和授权对共享资源的访问。基于锁的实现要求事务维护数据库记录上的锁，以防止其他事务修改被加锁的记录或访问当前事务正在修改的记录，直到锁被释放位置，或者不加锁，通过事务调度，维护读取与写入的操作列表以限制事务的执行。悲观的调度可能导致死锁。

数据库的事务隔离级别

隔离级别	脏读（Dirty Read）	不可重复读 (NonRepeatable Read)	幻读（Phantom Read）
未提交读（Read Uncommitted）	可能	可能	可能
已提交读（Read committed）	不可能	可能	可能
可重复读（Repeatable Read）	不可能	不可能	可能
可串行化（Serializable）	不可能	不可能	不可能

MySQL中可重复读（Repeatable read）能防住幻读吗?

Sample

id	name	dept_id
1	Jia	1
2	Zhenyu	1

脏读（Dirty Read）

- 当一个事务读取到另外一个事务修改但未提交的数据时，就可能发生脏读。

事务1	事务2
SELECT dept_id FROM users WHERE id=1	
/* 结果是 1 */	
	UPDATE users SET dept_id=2 WHERE id=1
	/* No commit here */
SELECT dept_id FROM users WHERE id=1	
/* 结果是 2 */	
	ROLLBACK;/* lock-based DIRTY READ */

不可重复读

- “不可重复读”：在当执行SELECT 操作时没有获得读锁或SELECT操作执行完后马上释放了读锁； 另外一个事务对数据进行了更新,读到了不同的结果.

事务1	事务2
SELECT dept_id FROM users WHERE id=1	
/* 结果是 1 */	
	UPDATE users SET dept_id=2 WHERE id=1
	COMMIT;
SELECT dept_id FROM users WHERE id=1	
COMMIT;	
/* 结果是 2 */	

幻读

- “幻读”又叫"幻象读“
- 是"不可重复读"的一种特殊场景
  - 当事务1两次执行"SELECT … WHERE"检索一定范围内数据的操作中间
  - 事务2在这个表中创建了(如[[INSERT]])了一行新数据，
  - 这条新数据正好满足事务1的“WHERE”子句。

幻读

事务1			事务2
SELECT * FROM users WHERE id=1			
id	name	dept_id	
1	Jia	1	
2	Zhenyu	1	
			INSERT INTO users VALUES(3,'Dong',1)
			COMMIT;
SELECT * FROM users WHERE id=1			
id	name	dept_id	
1	Jia	1	
2	Zhenyu	1	
3	Dong	1	

区别（请背诵，谨防变态老师直接考概念）

- 脏读：指读到了其他事务未提交的数据.
- 不可重复读：读到了其他事务已提交的数据(update).

- 不可重复读与幻读都是读到其他事务已提交的数据，但是它们针对点不同.
- 不可重复读：update.
- 幻读：delete，insert.

MySQL中的四种事务隔离级别

- 未提交读（READ UNCOMMITTED）
- 已提交读（READ COMMITTED）
- 可重复读（REPEATABLE READS）
- 可串行化（Serializable ）

未提交读（READ UNCOMMITTED）

- 未提交读（READ UNCOMMITTED）是最低的隔离级别，在这种隔离级别下，如果一个事务已经开始写数据，则另外一个事务则**不允许同时进行写操作，但允许其他事务读此行数据**。

事务1	事务2
SELECT dept_id FROM users WHERE id=1	
/* 结果是 1 */	
	UPDATE users SET dept_id=2 WHERE id=1
	/* No commit here */
SELECT dept_id FROM users WHERE id=1	
/* 结果是 2 */	

已提交读（READ COMMITTED）

- **读取数据的事务允许其他事务继续访问该行数据，但是未提交的写事务将会禁止其他事务访问该行，会对该写锁一直保持直到到事务提交.**

事务1	事务2
SELECT dept_id FROM users WHERE id=1	
/* 结果是 1 */	
	UPDATE users SET dept_id=2 WHERE id=1
	/* No commit here ， But Locked*/
SELECT dept_id FROM users WHERE id=1	
/* Waiting Waiting */	
	ROLLBACK;
SELECT dept_id FROM users WHERE id=1	
/* 结果是 1 */	

已提交读（READ COMMITTED）

- 能解决脏读的问题，却防不了不可重复读的问题。

事务1	事务2
SELECT dept_id FROM users WHERE id=1	
/* 结果是 1 */	
	UPDATE users SET dept_id=2 WHERE id=1
	COMMIT;
SELECT dept_id FROM users WHERE id=1	
COMMIT;	
/* 结果是 2 */	

可重复读（REPEATABLE READS）

- 可重复读（REPEATABLE READS）是介于已提交读和可串行化之间的一种隔离级别
  - 它是InnoDb的默认隔离级别。

- 可串行化（Serializable ）是高的隔离级别，它求在选定对象上的读锁和写锁保持直到事务结束后才能释放，所以能防住上诉所有问题，但因为是串行化的，所以效率较低。

好，我们开始进入今天的主题

可重复读（Reapeatable Read）能防止幻读嘛?

可重复读的实验

事务1			事务2	事务3
SELECT * FROM users WHERE id=1				
id	name	dept_id		
1	Jia	1		
2	Zhenyu	1		
			UPDATE users SET dept_id=2 WHERE id=1	
			COMMIT;	
				INSERT INTO users VALUES(3,'Dong',1)
				COMMIT;
SELECT * FROM users WHERE id=1				
id	name	dept_id		
1	Jia	1		
2	Zhenyu	1		

没有读到事务2的修改，也没有读到事务3新添加的数据，哎，没把DONG SHAO老师读进来，怎么能犯这么大的错呢?

它是怎么做到这么“弱鸡”的呢?

- **MVCC(多版本并发控制)**
- 在InnoDB中，会在每行数据后添加两个额外的隐藏的值来实现MVCC，这两个值一个记录这行数据何时被创建，另外一个记录这行数据何时过期（或者被删除）。 在实际操作中，存储的并不是时间，而是事务的版本号，每开启一个新事务，事务的版本号就会递增。
  - 在可重读Repeatable reads事务隔离级别下:
    - SELECT时，读取创建版本号<=当前事务版本号，删除版本号为空或>当前事务版本号。
    - INSERT时，保存当前事务版本号为行的创建版本号
    - DELETE时，保存当前事务版本号为行的删除版本号
    - UPDATE时，插入一条新纪录，保存当前事务版本号为行创建版本号，同时保存当前事务版本号到原来删除的行

到这里，我们已经证明了RR的隔离级别在InnoDB下可以有效防止脏读、幻读

哎，我们还是太年轻，为了讲这个已经讲烂的内容，还需要加一个ppt嘛？

我们再深入一点

Read的差异

- **快照读**
  - 我们平时只用使用select就是快照读，这样可以减少加锁所带来的开销。

```
select * from table ....
```

当前读

- 对于会对数据修改的操作(update、insert、delete)都是采用当前读的模式。在执行这几个操作时会读取最新的记录，即使是别的事务提交的数据也可以查询到。假设要update一条记录，但是在另一个事务中已经delete掉这条数据并且commit了，如果update就会产生冲突，所以在update的时候需要知道最新的数据。读取的是最新的数据，需要加锁。

```
select * from table where ? lock in share mode;
select * from table where ? for update;
insert;
update;
delete;
```

第一个语句需要加共享锁，其它都需要加排它锁。

来，我们比较一下两个隔离级别的差异

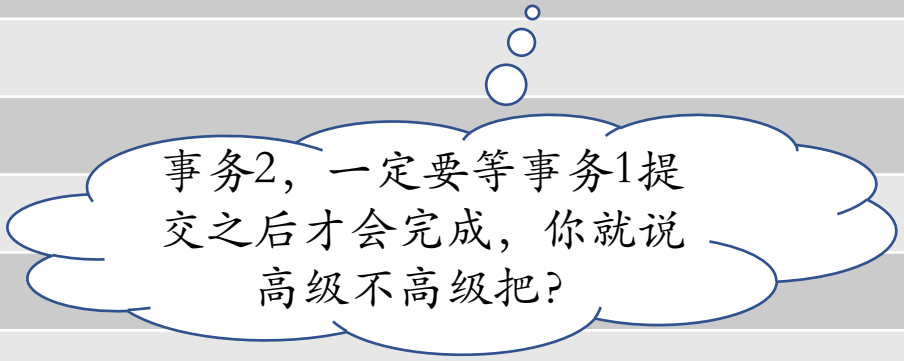
已提交读（Read committed）的隔离级别

事务1			事务2
SELECT * FROM users WHERE id=1			
id	name	dept_id	
1	Jia	1	
2	Zhenyu	1	
UPDATE users SET name='Qin' WHERE dept_id=1			
			INSERT INTO users VALUES(3,'Dong',1)
			COMMIT;
SELECT * FROM users WHERE id=1			
id	name	dept_id	
1	Qin	1	
2	Qin	1	
3	Dong	1	

这就是幻读，想想，钦老师，刚刚把所有老师都改名成自己，突入自己的事务中出现了栋老师的名字，是多么惊恐和幻灭的事情？

可重复读RR的隔离级别

事务1			事务2
SELECT * FROM users WHERE id=1			
id	name	dept_id	
1	Jia	1	
2	Zhenyu	1	
UPDATE users SET name='Qin' WHERE dept_id=1			
			INSERT INTO users VALUES(3,'Dong',1)
			Waiting;
SELECT * FROM users WHERE id=1			
id	name	dept_id	
1	Qin	1	
2	Qin	1	
	COMMIT;		



怎么实现的呢?

- 在可重复度的隔离级别下
  - 事务1在update后，对该数据加锁，事务2无法插入新的数据
  - 这样事务A在update前后数据保持一致，避免了幻读
- 怎么做到的呢?
  - 可以明确的是，update锁住的肯定不只是已查询到的几条数据，因为这样无法阻止insert
  - 那不就是表锁嘛?
- Young! Young! Young!
  - MySQL怎么会如此低效的使用表锁，它使用的是**NEXT-key**锁。

Next-Key锁

- 很简单，还记得我们说过的外键加锁是一个普遍的状况嘛?
- Users表中，dept_id如果有非聚簇索引，MySQL维护一个非聚簇索引与主键的关系，我们通过dept_id找到这个索引所对应所有的节点，这些节点存储着对应数据的主键信息，我们再通过主键id找到我们需要的数据，这个过程叫回表。

- B树索引所有的数据都在叶节点上，只要在外键索引上构建一个区间锁，就可以完成dept_id=1就插入不进来，但是可以任意处理dept_id的其它值的相关处理
  - 这种区间锁叫GAP锁

那…书上说的这张隔离表就是错的嘛?

隔离级别	脏读（Dirty Read）	不可重复读 (NonRepeatable Read)	幻读（Phantom Read）
未提交读（Read Uncommitted）	可能	可能	可能
已提交读（Read committed）	不可能	可能	可能
可重复读（Repeatable Read）	不可能	不可能	**可能**
可串行化（Serializable）	不可能	不可能	不可能

答案是：并没有……

6+，你是玩我是吗？一会儿说RR能避免幻读，一会儿又说并没有。

事务1			事务2
SELECT * FROM users WHERE id=1			
id	name	dept_id	
1	Jia	1	
2	Zhenyu	1	
			INSERT INTO users VALUES(3,'Dong',1)
			COMMIT;
SELECT * FROM users WHERE id=1			
id	name	dept_id	
1	Jia	1	
2	Zhenyu	1	
UPDATE users SET name='Qin'			
SELECT * FROM users WHERE id=1			
id	name	dept_id	
1	Qin	1	
2	Qin	1	
3	Qin	1	

事务2 insert加了一个GAP锁，提交后释放，事务1可以任意操作了

事务1，因为MVCC下，第二次select还是和第一次select显示的结果一样一样的

而如果不加条件的UPDATE，这个UPDATE就会作用于所有行，包括事务2新加的那一行，这时候再select就看出那个新加的行也被update了。钦哥终于得逞了。

结论

- Mysql官方给出的幻读解释是：只要在一个事务中，第二次select多出了row就算幻读，所以在刚刚钦哥的那个这个场景下，是算出现幻读的。

- 所以，**理论上，可重复读（Repeatable Read）是无法避免幻读的**。
- 实际上，MySQL 中可重复读（Repeatable Read）几乎都不会产生幻读
  - 你只要记住钦哥的那个行为能奏效的场景，在也许能用到的场景下注意就好了，只有这一种特例。