DTCC

2017第八届中国数据库技术大会

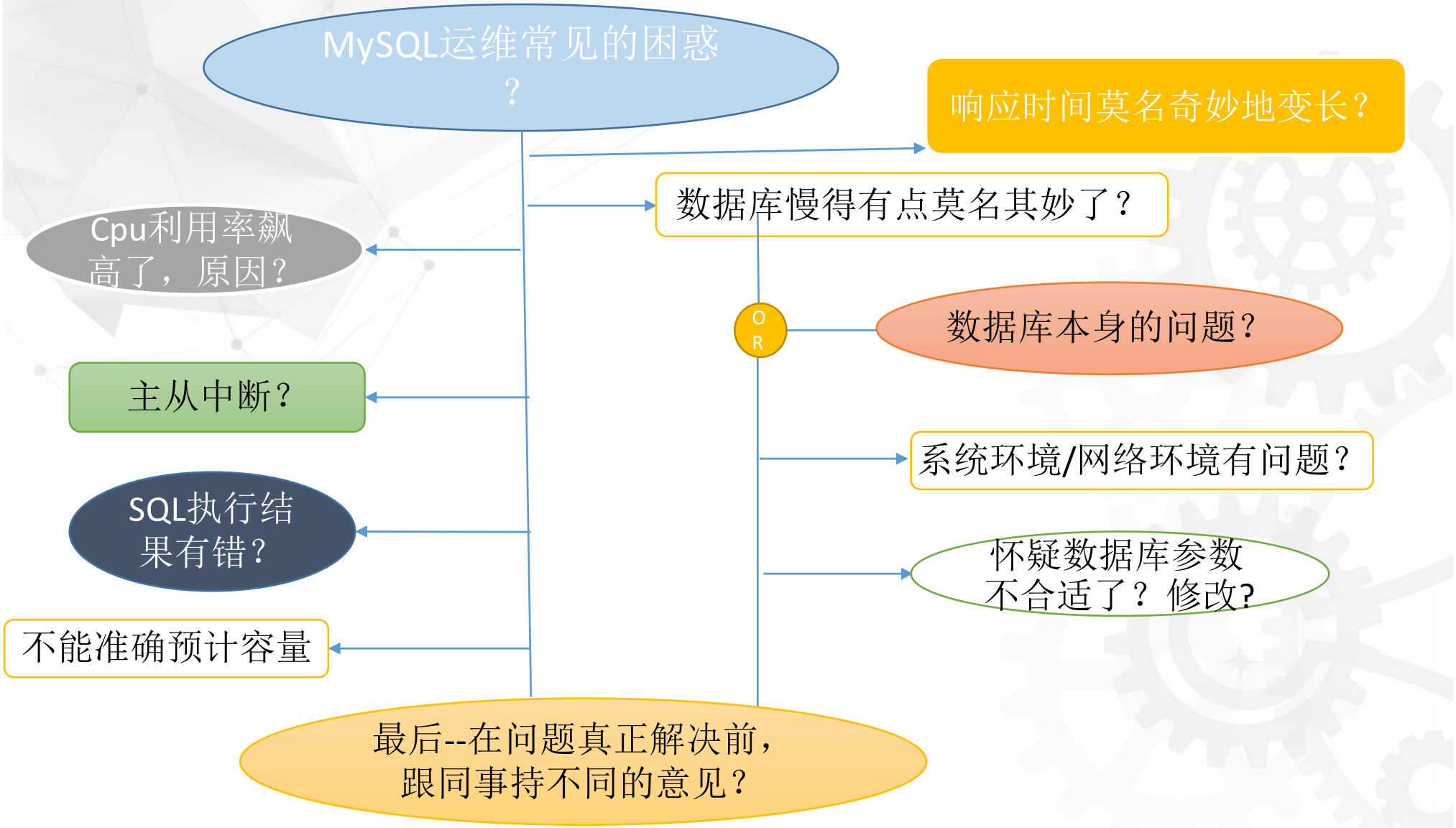
DATABASE TECHNOLOGY CONFERENCE CHINA 2017

MySQL核心参数含义源码解析

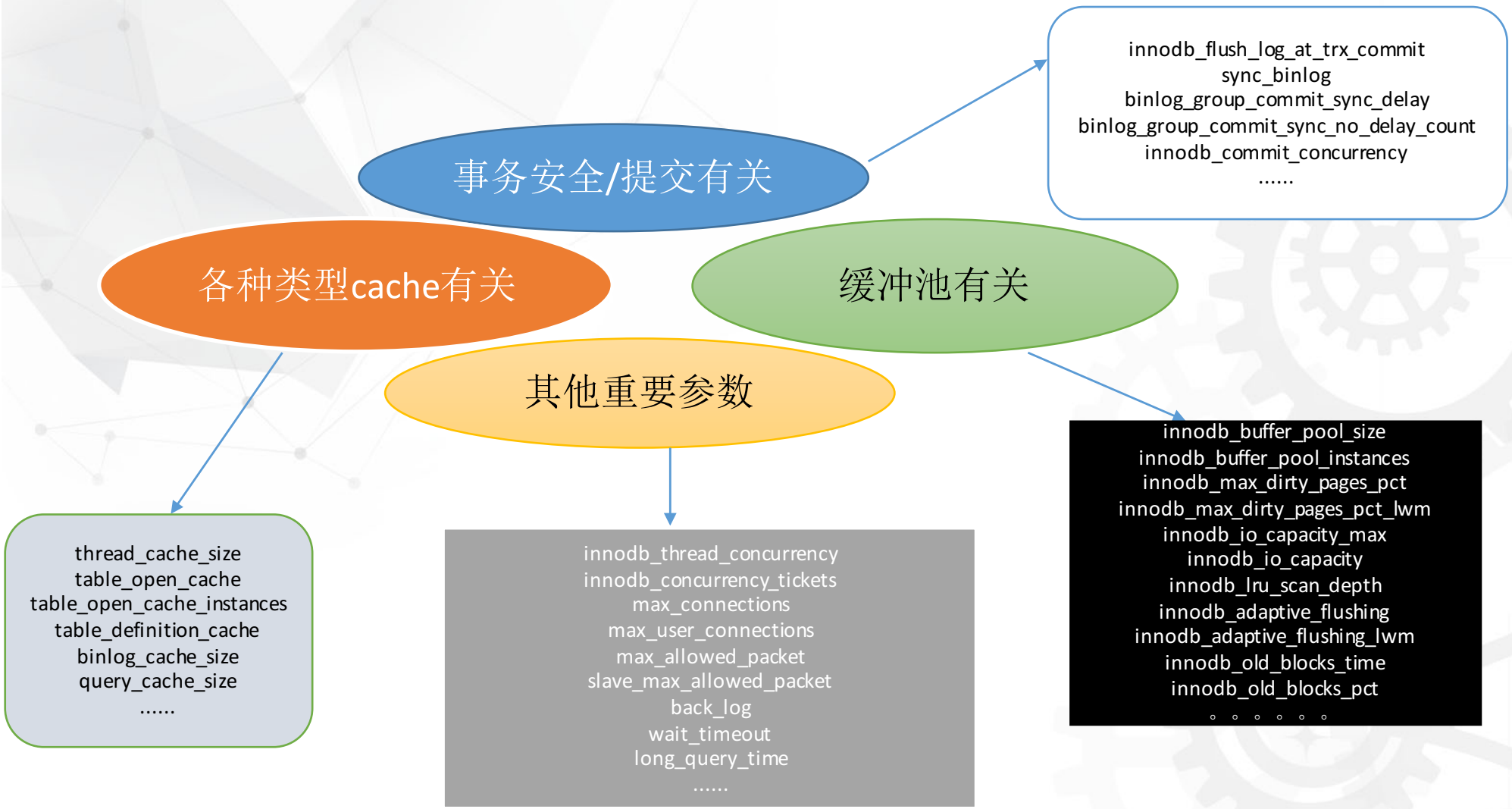
徐春阳

目录

1. 前言

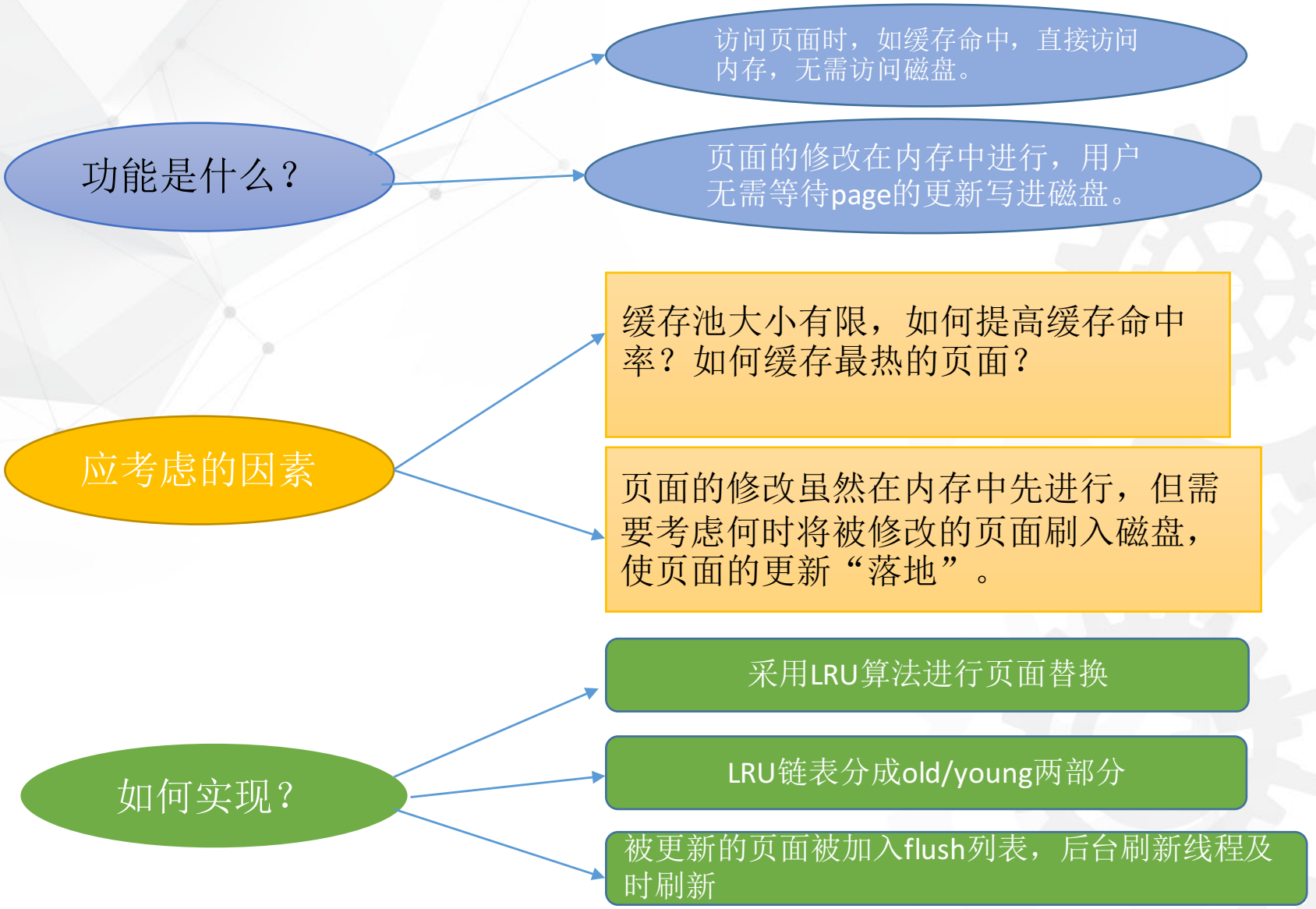

MySQL运维常见的困惑？
响应时间莫名奇妙地变长？
数据库慢得有点莫名其妙了？
Cpu利用率飙高了，原因？
O R
数据库本身的问题？
主从中断？
系统环境/网络环境有问题？
SQL执行结果有错？
怀疑数据库参数不合适了？修改?
不能准确预计容量
最后--在问题真正解决前，跟同事持不同的意见？

2. 数据库核心参数总览

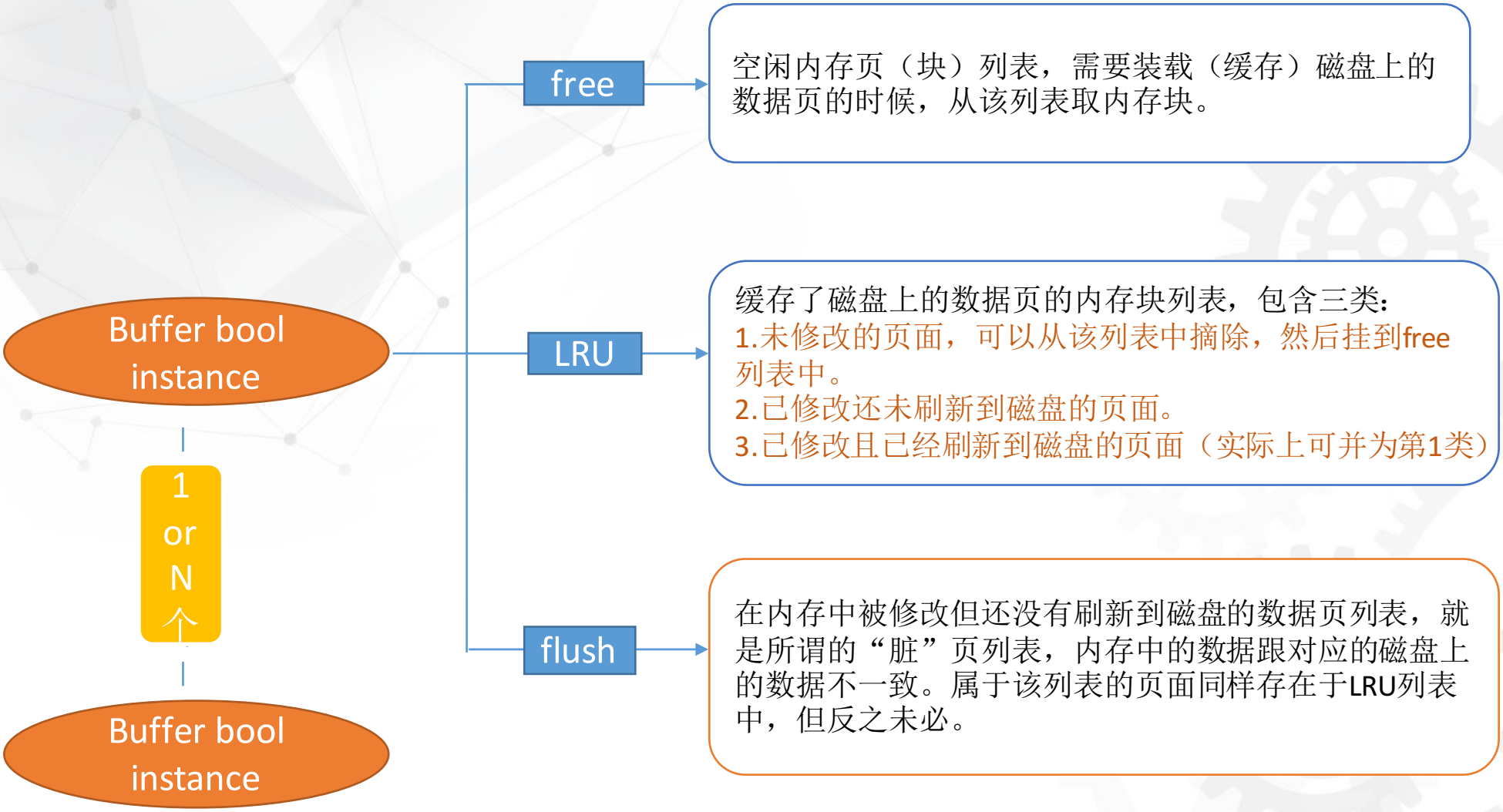


注：对于某些参数，如果视角不同，归类也可能不一样。限于时间，后面仅讲解缓存池的机制以及与其相关参数。

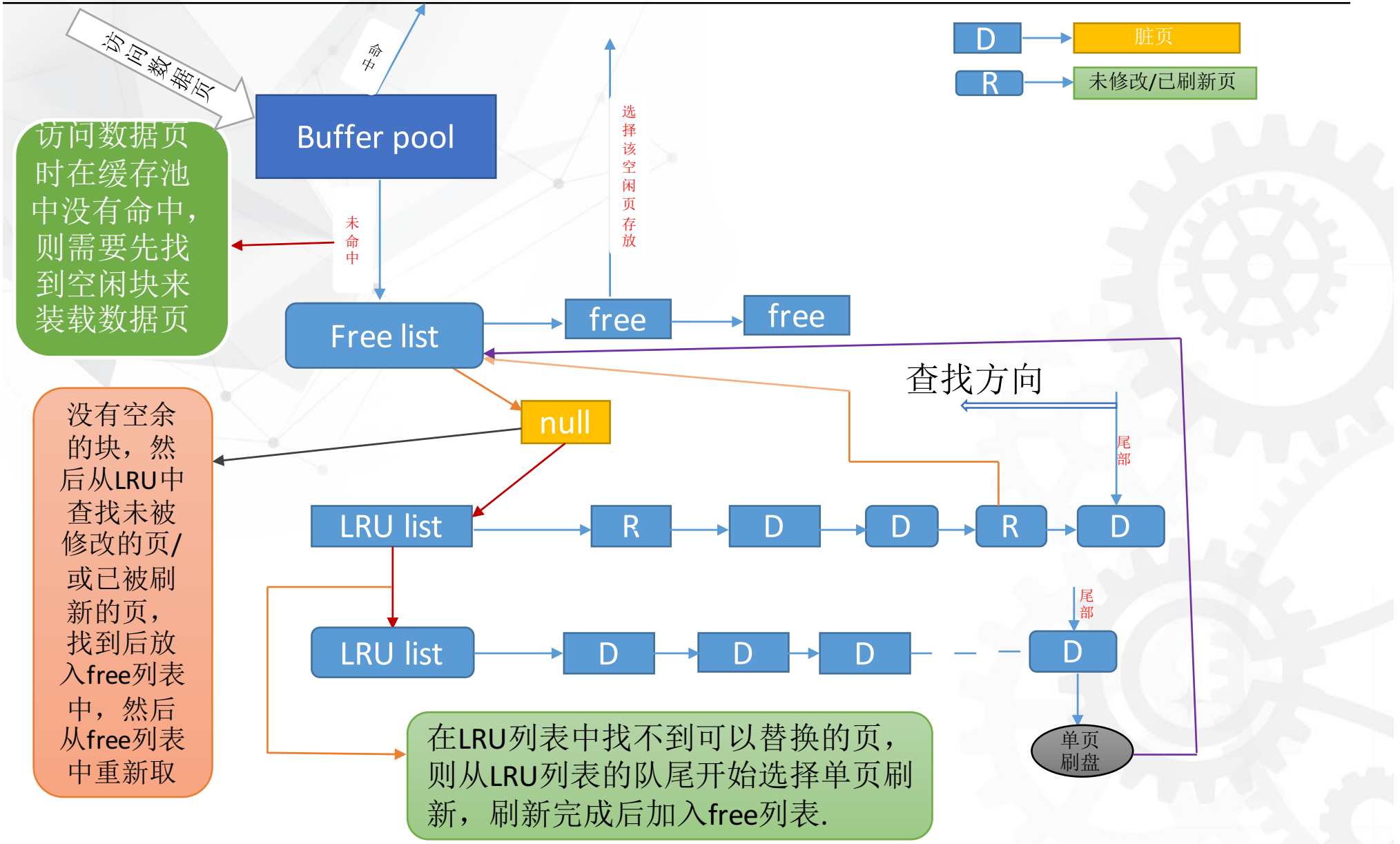
3. 缓存池工作机制概述


功能是什么？
访问页面时，如缓存命中，直接访问内存，无需访问磁盘。
页面的修改在内存中进行，用户无需等待page的更新写进磁盘。
应考虑的因素
缓存池大小有限，如何提高缓存命中率？如何缓存最热的页面？
页面的修改虽然在内存中先进行，但需要考虑何时将被修改的页面刷入磁盘，使页面的更新“落地”。
如何实现？
采用LRU算法进行页面替换
LRU链表分成old/young两部分
被更新的页面被加入flush列表，后台刷新线程及时刷新

3.1 缓存池中的列表


Buffer bool instance
1 or N 个
Buffer bool instance
free
空闲内存页（块）列表，需要装载（缓存）磁盘上的数据页的时候，从该列表取内存块。
LRU
缓存了磁盘上的数据页的内存块列表，包含三类：
1.未修改的页面，可以从该列表中摘除，然后挂到free列表中。
2.已修改还未刷新到磁盘的页面。
3.已修改且已经刷新到磁盘的页面（实际上可并为第1类）
flush
在内存中被修改但还没有刷新到磁盘的数据页列表，就是所谓的“脏”页列表，内存中的数据跟对应的磁盘上的数据不一致。属于该列表的页面同样存在于LRU列表中，但反之未必。

3.2 访问数据页的流程


访问数据页
命中
Buffer pool
选择该空闲页存放
D
脏页
R
未修改/已刷新页
访问数据页时在缓存池中没有命中，则需要先找到空闲块来装载数据页
未命中
Free list
free
free
查找方向
null
没有空余的块，然后从LRU中查找未被修改的页/或已被刷新的页，找到后放入free列表中，然后从free列表中重新取
尾部
LRU list
R
D
D
R
D
尾部
LRU list
D
D
D
D
单页刷盘
在LRU列表中找不到可以替换的页，则从LRU列表的队尾开始选择单页刷新，刷新完成后加入free列表.

3.3 访问数据页的源码之一寻找空闲的block函数

```
buf_block_t*
buf_LRU_get_free_block(
/*===================*/
        buf_pool_t*     buf_pool)       /*!< in/out: buffer pool instance */
{
loop:
        /* If there is a block in the free list, take it */
        block = buf_LRU_get_free_only(buf_pool);

        if (block != NULL) {

                return(block);
        }

        freed = false;
        if (buf_pool->try_LRU_scan || n_iterations > 0) {
                /* If no block was in the free list, search from the
                end of the LRU list and try to free a block there.
                If we are doing for the first time we'll scan only
                tail of the LRU list otherwise we scan the whole LRU
                list. */
                freed = buf_LRU_scan_and_free_block(
                        buf_pool, n_iterations > 0);

                if (!freed && n_iterations == 0) {
                        /* Tell other threads that there is no point
                        in scanning the LRU list. This flag is set to
                        TRUE again when we flush a batch from this
                        buffer pool. */
                        buf_pool->try_LRU_scan = FALSE;
                }
        }

        if (freed) {
                goto loop;
        }

        /* If we have scanned the whole LRU and still are unable to
        find a free block then we should sleep here to let the
        page_cleaner do an LRU batch for us. */

        if (!srv_read_only_mode) {
                os_event_set(buf_flush_event);
        }

        if (n_iterations > 1) {
                os_thread_sleep(10000);
        }

        /* No free block was found: try to flush the LRU list.
        This call will flush one page from the LRU and put it on the
        free list. That means that the free block is up for grabs for
        all user threads. */
        if (!buf_flush_single_page_from_LRU(buf_pool)) {
                ++flush_failures;
        }
        srv_stats.buf_pool_wait_free.add(n_iterations, 1);
        n_iterations++;
        goto loop;
}
```

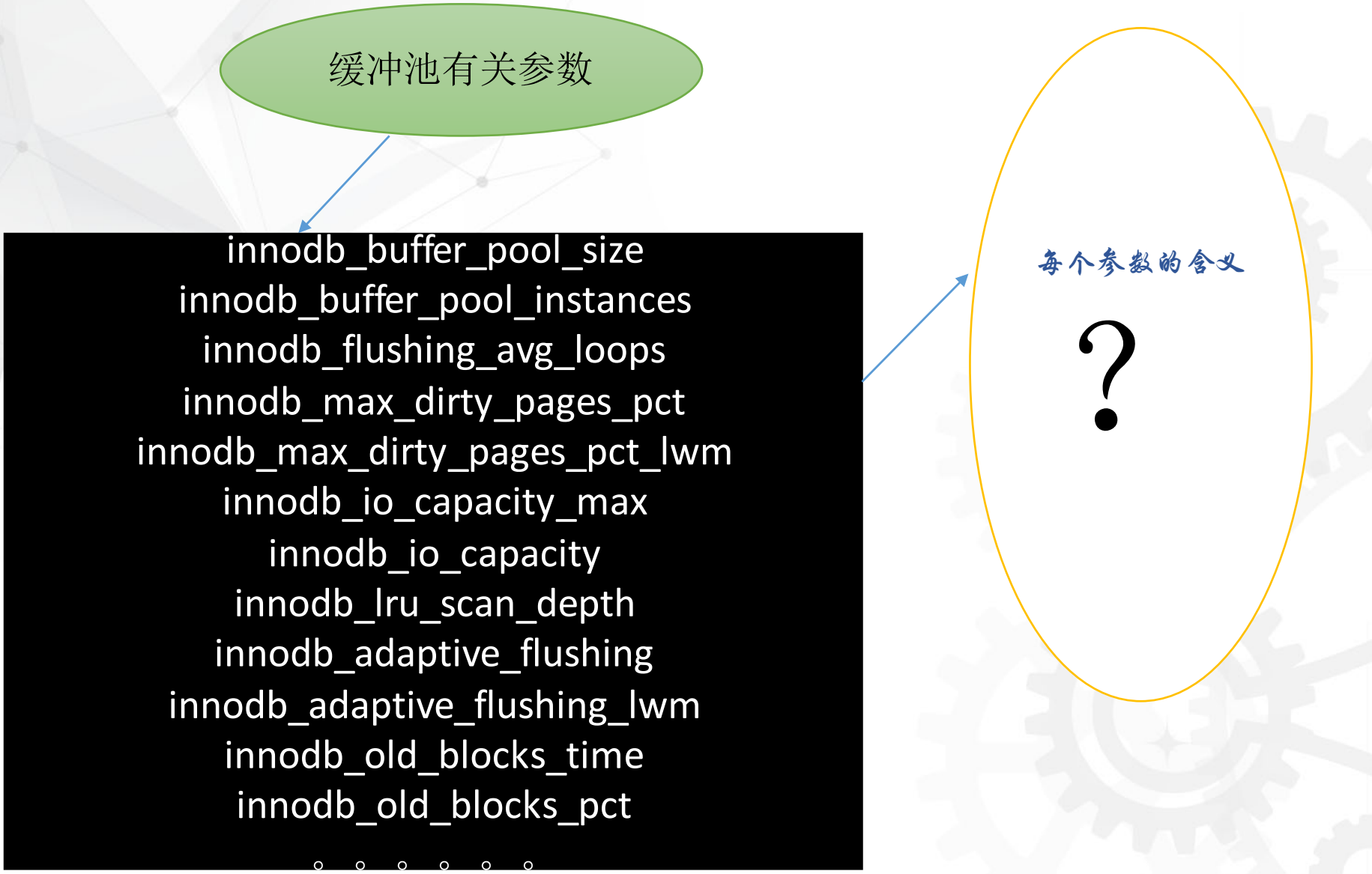
从free list 取block,找到就返回block

从LRU列表中查找可以替换的block然后放入free 列表，如果返回true,然后 跳转到函数loop位置，重新从free列表中找空闲的block。第一次扫描时仅scan 100个页(不考虑压缩页)，如果需要进行第二次扫描，则扫描整个lru列表。

从LRU列表中刷新一个页面(”脏”页)，然后加入free列表，跳转到loop位置，进行新的循环。

？？？如果需要用户线程来刷新“脏页”来产生空闲页，系统的性能将如何？Free or 可以替换的页大量存在太重要了。

3.4 缓存池的有关参数


缓冲池有关参数
innodb_buffer_pool_size
innodb_buffer_pool_instances
innodb_flushing_avg_loops
innodb_max_dirty_pages_pct
innodb_max_dirty_pages_pct_lwm
innodb_io_capacity_max
innodb_io_capacity
innodb_lru_scan_depth
innodb_adaptive_flushing
innodb_adaptive_flushing_lwm
innodb_old_blocks_time
innodb_old_blocks_pct
。。。。。。
每个参数的含义
？

4 缓存池源码解析

4.1 buf_flush_page_cleaner_coordinator 函数(1)—解析

简称协调函数，该函数为后台页面刷新协调线程的入口函数，该线程所有工作都是这个函数来完成的：它负责对innodb 缓存池刷新线程（即page cleaner线程）的调度，同时自己也会执行刷新函数。刷新协调线程期望每1秒钟进行一次的“脏页”刷新（如有必要）。同时，在调度刷新动作之前，会对每个buffer pool instance 生成需要刷新多少页的建议。 生成建议的函数为page_cleaner_flush_pages_recommendation。

如上所述，期望1秒钟进行一次刷新，如果两次刷新的间隔超过4000ms,error log 会出现相关信息.如下图:

```
2017-01-11T16:40:
2017-01-11T16:40:
2017-01-11T16:40:
2017-01-11T16:48:53.391483+08:00 0 [Note] InnoDB: page_cleaner: 1000ms intended loop took 6118ms. The settings might not be optimal. (flushed=2009
2017-01-11T16:50:09.289697+08:00 0 [Note] InnoDB: page_cleaner: 1000ms intended loop took 5452ms. The settings might not be optimal. (flushed=2007
2017-01-11T17:07:50.243999+08:00 0 [Note] InnoDB: page_cleaner: 1000ms intended loop took 4205ms. The settings might not be optimal. (flushed=1724
2017-01-11T17:10:19.425959+08:00 0 [Note] InnoDB: page_cleaner: 1000ms intended loop took 4055ms. The settings might not be optimal. (flushed=1770
2017-01-11T17:11:59.019423+08:00 0 [Note] InnoDB: page_cleaner: 1000ms intended loop took 4057ms. The settings might not be optimal. (flushed=1920
2017-01-11T17:12:04.605992+08:00 0 [Note] InnoDB: page_cleaner: 1000ms intended loop took 4586ms. The settings might not be optimal. (flushed=1958
2017-01-11T17:12:10.452216+08:00 0 [Note] InnoDB: page_cleaner: 1000ms intended loop took 4847ms. The settings might not be optimal. (flushed=1981
2017-01-11T17:14:16.802831+08:00 0 [Note] InnoDB: page_cleaner: 1000ms intended loop took 4420ms. The settings might not be optimal. (flushed=1879
2017-01-11T17:15:38.268664+08:00 0 [Note] InnoDB: page_cleaner: 1000ms intended loop took 4364ms. The settings might not be optimal. (flushed=2013
2017-01-11T17:15:44.872653+08:00 0 [Note] InnoDB: page_cleaner: 1000ms intended loop took 5604ms. The settings might not be optimal. (flushed=2012
2017-01-11T17:18:57.626811+08:00 0 [Note] InnoDB: page_cleaner: 1000ms intended loop took 4167ms. The settings might not be optimal. (flushed=2014
2017-01-11T17:28:04.963038+08:00 0 [Note] InnoDB: page_cleaner: 1000ms intended loop took 4349ms. The settings might not be optimal. (flushed=44706
2017-01-11T17:28:18.081262+08:00 0 [Note] InnoDB: page_cleaner: 1000ms intended loop took 4524ms. The settings might not be optimal. (flushed=44620
2017-01-11T17:34:37.724858+08:00 0 [Note] InnoDB: page_cleaner: 1000ms intended loop took 5416ms. The settings might not be optimal. (flushed=6736
2017-01-11T17:34:53.277845+08:00 0 [Note] InnoDB: page_cleaner: 1000ms intended loop took 4232ms. The settings might not be optimal. (flushed=5830
2017-01-11T17:34:58.337816+08:00 0 [Note] InnoDB: page_cleaner: 1000ms intended loop took 4060ms. The settings might not be optimal. (flushed=5796
```

4.1 buf_flush_page_cleaner_coordinator 函数(2)—相关代码

```
if (ret_sleep == OS_SYNC_TIME_EXCEEDED) {
    ulint   curr_time = ut_time_ms();

    if (curr_time > next_loop_time + 3000) {
        if (warn_count == 0) {
            ib::info() << "page_cleaner: 1000ms"
                " intended loop took "
                << 1000 + curr_time
                   - next_loop_time
                << "ms. The settings might not"
                " be optimal. (flushed="
                << n_flushed_last
                << " and evicted="
                << n_evicted
                << ", during the time.)";
            if (warn_interval > 300) {
                warn_interval = 600;
            } else {
                warn_interval *= 2;
            }

            warn_count = warn_interval;
        } else {
            --warn_count;
        }
    } else {
        /* reset counter */
        warn_interval = 1;
        warn_count = 0;
    }

    next_loop_time = curr_time + 1000;
    n_flushed_last = n_evicted = 0;
}
```

循环超时

调用刷新建议函数

```
} else if (srv_check_activity(last_activity)) {
        ulint   n_to_flush;
        lsn_t   lsn_limit = 0;

        /* Estimate pages from flush_list to be flushed */
        if (ret_sleep == OS_SYNC_TIME_EXCEEDED) {
                last_activity = srv_get_activity_count();
                n_to_flush =
                        page_cleaner_flush_pages_recommendation(
                                &lsn_limit, last_pages);
        } else {
                n_to_flush = 0;
        }

        /* Request flushing for threads */
        pc_request(n_to_flush, lsn_limit);

        ulint tm = ut_time_ms();

        /* Coordinator also treats requests */
        while (pc_flush_slot() > 0) {
                /* No op */
        }
```

4.1 buf_flush_page_cleaner_coordinator 函数(e)—主体流程

```
/* Estimate pages from flush_list to be flushed */
if (ret_sleep == OS_SYNC_TIME_EXCEEDED) {
    last_activity = srv_get_activity_count();
    n_to_flush =
        page_cleaner_flush_pages_recommendation(
            &lsn_limit, last_pages);
} else {
    n_to_flush = 0;
}
```

对每个缓冲池实例生成需要刷新多少脏页的建议

```
/* Request flushing for threads */
pc_request(n_to_flush, lsn_limit);
```

通过设置事件的方式，向刷新线程发出刷新请求

```
ulint tm = ut_time_ms();

/* Coordinator also treats requests */
while (pc_flush_slot() > 0) {
    /* No op */
}
```

协调线程作为刷新调度总负责人的角色，会保证每一个buffuer pool instance 都开始进行了刷新。如果某个buffer还没有开始刷新，则由协调函数自己进行刷新，直到所有的buffer pool instance都已开始/进行了刷新，才退出这个while循环。

```
/* only coordinator is using these counters,
so no need to protect by lock. */
page_cleaner->flush_time += ut_time_ms() - tm;
page_cleaner->flush_pass++ ;

/* Wait for all slots to be finished */
ulint   n_flushed_lru = 0;
ulint   n_flushed_list = 0;

pc_wait_finished(&n_flushed_lru, &n_flushed_list);
```

等待所有的buffer pool instance完成刷新。这也是相邻两次刷新的启动时间的间隔可能超过1秒，甚至几秒的原因，如间隔时间超过4秒，则错误日志有“page_cleaner:1000ms intended loop took 4055ms”类似记录。

4.2 page_cleaner_flush_pages_recommendation 函数概述(1)

计算脏页刷新与LSN增长的平均速度，但并非每次函数调用都计算，而是由参数innodb_flushing_avg_loops决定，默认为30，即函数被调用30次时或经过30秒之后再计算这两项的平均速度。分别用变量avg_page_rate与lsn_avg_rate保存。 计算规则为： 新的平均值=（原平均值+最近这段时间的平均速度）/2。

通过缓冲池中的脏页百分比计算innodb_io_capacity的百分比，由函数af_get_pct_for_dirty计算，结果用变量pct_for_dirty保存。

通过lsn的age来计算innodb_io_capacity百分比，由函数af_get_pct_for_lsn计算，结果用变量pct_for_lsn保存。

注:这两个函数后面将介绍

pct_total = ut_max(pct_for_dirty, pct_for_lsn);两值比较后取大的值赋给pct_total

根据lsn(重做日志）产生的平均速度即lsn_avg_rate， 以及脏页的oldest_modification（最老的日志号lsn）这两个因素来第一次计算每个buffer pool instance需要刷新多少页面，将结果存入page_cleaner->slots[i].n_pages_requested， 同时，将所有buffer pool instance需要刷新的总数计入变量sum_pages_for_lsn，将sum_pages_for_lsn 与srv_max_io_capacity * 2比较，取小值赋给变量pages_for_lsn。

n_pages = (PCT_IO(pct_total) + avg_page_rate + pages_for_lsn) / 3;

注： PCT_IO(pct_total)= pct_total * innodb_io_capacity/100

由此得出：Innodb_io_capacity参数的设置将直接影响对缓存池进行刷新的数量。

4.2 page_cleaner_flush_pages_recommendation 函数概述(e)

if (n_pages > srv_max_io_capacity) { n_pages = srv_max_io_capacity; }

再次为每个buffer pool instance计算最终建议的刷新页面的数量：代码如下

```
for (ulint i = 0; i < srv_buf_pool_instances; i++) {
    /* if REDO has enough of free space,
    don't care about age distribution of pages */
    page_cleaner->slots[i].n_pages_requested = pct_for_lsn > 30 ?
        page_cleaner->slots[i].n_pages_requested
        * n_pages / sum_pages_for_lsn + 1
        : n_pages / srv_buf_pool_instances;
}
```

解析：如果前面根据活跃重做日志量来计算得出的pct_for_lsn>30,则每个缓存池需要刷新的页面等于page_cleaner->slots[i].n_pages_requested * n_pages / sum_pages_for_lsn + 1；即根据前面介绍的根据redo的产生速度以及Page的age分布所计算出来的需要刷新页的数量乘以n_pages再除以sum_pages_for_lsn再加1. 这两个值的来源请查看上页ppt。

否则，则认为redo log file 还有足够的空间，不考虑脏页的age在缓存池中分布可能不均的情况，则建议每个缓存池实例的刷新页面的数量一致，即等于n_pages / srv_buf_pool_instances， 即总的页面数除以buffer pool实例的个数。

*lsn_limit = LSN_MAX; /*将函数入参指针所指向的内存地址赋值。*/
return(n_pages); /*返回建议刷新页面的总数*/

4.2.1 计算刷新页面的平均速度以及redo日志产生的平均速度

```
if (++n_iterations >= srv_flushing_avg_loops
    || time_elapsed >= srv_flushing_avg_loops) {

        if (time_elapsed < 1) {
                time_elapsed = 1;
        }

        avg_page_rate = static_cast<ulint>(
                ((static_cast<double>(sum_pages)
                  / time_elapsed)
                 + avg_page_rate) / 2);

        /* How much LSN we have generated since last call. */
        lsn_rate = static_cast<lsn_t>(
                static_cast<double>(cur_lsn - prev_lsn)
                / time_elapsed);

        lsn_avg_rate = (lsn_avg_rate + lsn_rate) / 2;


        /* aggregate stats of all slots */
        mutex_enter(&page_cleaner->mutex);

        ulint   flush_tm = page_cleaner->flush_time;
        ulint   flush_pass = page_cleaner->flush_pass;

        page_cleaner->flush_time = 0;
        page_cleaner->flush_pass = 0;
```

srv_flushing_avg_loops，即参数innodb_flushing_avg_loops，默认30，下面都以默认值做说明，即30次刷新循环之后做一次平均速度的统计与更新。

sum_pages为最近30次刷新循环所刷新的页面的总和。

prev_lsn为上一次统计平均值时的当时的lsn号。Cur_lsn减去prev_lsn表示的最近30次刷新循环期间的redo日志增长量。

根据这两个值再除以30次循环所历时时间等于这段时间的平均速度。

跟原来的值再平均，就是当前的最新的平均值。分别用avg_page_rate与lsn_avg_rate表示. 计算出来供后续使用。

4.2.2 根据脏页的百分比以及lsn的age来计算io_capacity的百分比

```
oldest_lsn = buf_pool_get_oldest_modification();

ut_ad(oldest_lsn <= log_get_lsn());

age = cur_lsn > oldest_lsn ? cur_lsn - oldest_lsn : 0;

pct_for_dirty = af_get_pct_for_dirty();
pct_for_lsn = af_get_pct_for_lsn(age);

pct_total = ut_max(pct_for_dirty, pct_for_lsn);
```

LSN为oldest_lsn之前的页面都已经刷新到磁盘。

lsn的age表示涉及到的脏页还没有刷入磁盘的重做日志的大小。

根据buffer pool中的脏页百分比来计算io_capacity的百分比。脏页的百分比越大，百分比越大。返回的值越大。

根据lsn的age，来计算io_capacity的百分比，age越大，返回值越大。结果pct_for_lsn后面会再次被利用。

取pct_for_dirty与pct_for_lsn中的大值最为pct_total

4.2.2.1 根据脏页百分比来计算io_capacity百分比—af_get_pct_for_dirty

```
/*********************************************************************//**
Calculates if flushing is required based on number of dirty pages in
the buffer pool.
@return percent of io_capacity to flush to manage dirty page ratio */
static
ulint
af_get_pct_for_dirty()
/*==================*/
{
        double  dirty_pct = buf_get_modified_ratio_pct();

        if (dirty_pct == 0.0) {
                /* No pages modified */
                return(0);
        }
        if (srv_max_dirty_pages_pct_lwm == 0) {
                /* The user has not set the option to preflush dirty
                pages as we approach the high water mark. */
                if (dirty_pct >= srv_max_buf_pool_modified_pct) {
                        /* We have crossed the high water mark of dirty
                        pages In this case we start flushing at 100% of
                        innodb_io_capacity. */
                        return(100);
                }
        } else if (dirty_pct >= srv_max_dirty_pages_pct_lwm) {
                /* We should start flushing pages gradually. */
                return(static_cast<ulint>((dirty_pct * 100)
                        / (srv_max_buf_pool_modified_pct + 1)));
        }

        return(0);
}
```

获取“脏”页百分比

没有“脏”页，直接返回0

如果参数innodb_max_dirty_pages_pct_lwm为0(默认值），且dirty_pct大于参数innodb_max_dirty_pages_pct，则返回100

否则，如果dirty_pct大于参数innodb_max_dirty_pages_pct_lwm，当dirty_pct越接近innodb_max_dirty_page_pct，返回值越接近100。

其他情况，返回0

4.2.2.2 根据lsn的age来计算io_capacity的百分比--af_get_pct_for_lsn

```
Calculates if flushing is required based on redo generation rate.
@return percent of io_capacity to flush to manage redo space */
static
ulint
af_get_pct_for_lsn(
        lsn_t   age)    /*!< in: current age of LSN. */
{
        lsn_t   max_async_age;
        lsn_t   lsn_age_factor;
        lsn_t   af_lwm = (srv_adaptive_flushing_lwm
                          * log_get_capacity()) / 100;
        if (age < af_lwm) {
                /* No adaptive flushing. */
                return(0);
        }
        max_async_age = log_get_max_modified_age_async();
        if (age < max_async_age && !srv_adaptive_flushing) {
                /* We have still not reached the max_async_po
                the user has disabled adaptive flushing. */
                return(0);
        }
        /* If we are here then we know that either:
        1) User has enabled adaptive flushing
        2) User may have disabled adaptive flushing but we ha
        max_async_age. */
        lsn_age_factor = (age * 100) / max_async_age;
        ut_ad(srv_max_io_capacity >= srv_io_capacity);
        return(static_cast<ulint>(
                ((srv_max_io_capacity / srv_io_capacity)
                * (lsn_age_factor * sqrt((double)lsn_age_factor)))
                / 7.5));
}
```

如果lsn的age大小还没有达到log_file的容量乘以innodb_adaptive_flushing_lwm参数（默认10）的百分比，则返回0，也意味着log file的可用空间足够，不用考虑该因素：

注：lsn的age代表已在内存中修改但还没有固化到磁盘的页面所涉及的日志量的多少。

如果没有开启innodb_adaptive_flushing 参数（默认on)，且lsn的age没有达到max_async_age,则返回0。max_async_age是根据log file大小计算出来的。也就是说，当age超过max_async_age，则必须考虑redo logfile的可用空间可能不足这个因素来做刷新建议。

剩下情况:根据lsn的age跟max_async_age的比值来计算io_capacity的百分比。返回值符给pct_for_lsn

4.2.3 试算每个buffer pool instance需要刷新的页面。

```
/* Estimate pages to be flushed for the lsn progress */
ulint   sum_pages_for_lsn = 0;
lsn_t   target_lsn = oldest_lsn
                   + lsn_avg_rate * buf_flush_lsn_scan_factor;

for (ulint i = 0; i < srv_buf_pool_instances; i++) {
        buf_pool_t*     buf_pool = buf_pool_from_array(i);
        ulint           pages_for_lsn = 0;

        buf_flush_list_mutex_enter(buf_pool);
        for (buf_page_t* b = UT_LIST_GET_LAST(buf_pool->flush_list);
             b != NULL;
             b = UT_LIST_GET_PREV(list, b)) {
                if (b->oldest_modification > target_lsn) {
                        break;
                }
                ++pages_for_lsn;
        }
        buf_flush_list_mutex_exit(buf_pool);

        sum_pages_for_lsn += pages_for_lsn;

        mutex_enter(&page_cleaner->mutex);
        ut_ad(page_cleaner->slots[i].state
              == PAGE_CLEANER_STATE_NONE);
        page_cleaner->slots[i].n_pages_requested
                = pages_for_lsn / buf_flush_lsn_scan_factor + 1;
        mutex_exit(&page_cleaner->mutex);
```

注1

注2

注3

注4

注5

注：下一页面将分别对注释1-5进行讲解

4.2.3.1 试算每个buffer pool instance需要刷新的页面—注释。

lsn_avg_rate表示redo log 增长的平均速度，该值在函数前面部分已计算得出。buf_flush_lsn_scan_factor是源码中的固定值，等于3。oldest_lsn的含义是old_modification小于该oldest_lsn的页面都已经刷新到磁盘。根据公式计算得出target_lsn.

注1

对buffer pool instances进行递归计算， instances数量由参数innodb_buffer_pool_instances 决定。

注2

计算每个buffer pool instance需要刷新多少页的临时，计算规则：从脏页列表的最后一个页开始查找，将所有oldest_modification小于target_lsn的页面进行计数，临时结果存入pages_for_lsn。

注3

将所有的buffer pool instance的pages_for_lsn的值相加后符给sum_pages_for_lsn，但该值在后续还需要除以buf_flush_lsn_scan_factor。因为target_lsn在前面被放大，因此所得的sum_page_for_lsn的值也被放大，在后续后缩小还原。

注4

得出当前的buffer pool instance 需要刷新的页面的数量。结果存入page_cleaner->slots[i].n_pages_requested。公式为pages_for_lsn / buf_flush_lsn_scan_factor + 1;，原因之前的target_lsn被放大，所以得出的临时结果pages_for_lsn也被放大buf_flush_lsn_scan_factor倍， 现在缩小还原。 这次计算得到的page_cleaner->slots[i].n_pages_requested值为第一次计算结果，这个结果后续还会被更新。

注5

4.2.4 每个buffer pool instance生成最终的刷新建议。

```
sum_pages_for_lsn /= buf_flush_lsn_scan_factor;
if(sum_pages_for_lsn < 1) {
        sum_pages_for_lsn = 1;
}

/* Cap the maximum IO capacity that we are going to use by
max_io_capacity. Limit the value to avoid too quick increase */
ulint   pages_for_lsn =
        std::min<ulint>(sum_pages_for_lsn, srv_max_io_capacity * 2);

n_pages = (PCT_IO(pct_total) + avg_page_rate + pages_for_lsn) / 3;

if (n_pages > srv_max_io_capacity) {
        n_pages = srv_max_io_capacity;
}

/* Normalize request for each instance */
mutex_enter(&page_cleaner->mutex);
ut_ad(page_cleaner->n_slots_requested == 0);
ut_ad(page_cleaner->n_slots_flushing == 0);
ut_ad(page_cleaner->n_slots_finished == 0);

for (ulint i = 0; i < srv_buf_pool_instances; i++) {
        /* if REDO has enough of free space,
        don't care about age distribution of pages */
        page_cleaner->slots[i].n_pages_requested = pct_for_lsn > 30 ?
                page_cleaner->slots[i].n_pages_requested
                * n_pages / sum_pages_for_lsn + 1
                : n_pages / srv_buf_pool_instances;
}
```

“脏”页的age,可以理解为脏页的第一次被修改时间。越早被修改,age越大。如果一个缓冲池中age大的脏页多，在考虑age的分布情况下，则被建议刷新的页面就多。

n_pages代表预计需要刷新页面的总和，取值由脏页的百分比，redo生成的平均速度，当前脏页刷新平均速度，当前lsn的age,以及脏页的age分布情况，以及参数io_capacity与max_innodb_capacity这些因素共同决定。

如果pct_for_lsn小于30，则认为redo有足够空间，而不需要考虑“脏”页的age分布情况，则建议每个instance刷新同样的数量。否则，将使用前面根据“脏页”的oldest_modification（也就是age的分布），计算得出每个instance的刷新数量，再综合其它因素，又进行一次计算。每个instance刷新的数量将不同。

4.3 请求刷新—pc_request函数

```
for (ulint i = 0; i < page_cleaner->n_slots;  i++) {
page_cleaner_slot_t*  slot = &page_cleaner->slots[i];
.....
.....
.....
/* slot->n_pages_requested  was already set by
page_cleaner_flush_pages_recommendation()  */

slot->state = PAGE_CLEANER_STATE_REQUESTED;
}
```

将所有buffer pool instances的刷新状态设置为PAGE_CLEANER_STATE_REQUESTED，即申请刷新。

os_event_set(page_cleaner->is_requested);

通过设置事件，唤醒/触发page cleaner 线程，然后调用pc_flush_slot函数来进行buffer pool的刷新。

4.4 执行刷新—pc_flush_slot函数

```
for (i = 0; i < page_cleaner->n_slots;  i++) {
   slot = &page_cleaner->slots[i];
   if (slot->state == PAGE_CLEANER_STATE_REQUESTED) {
      break;
   }
}
slot->state  = PAGE_CLEANER_STATE_FLUSHING;
```

寻找一个标志为申请刷新的缓存池实例，然后选为刷新对象，将状态修改为flushing.。然后执行后面的刷新。

```
/* Flush pages from end of LRU if required */
slot->n_flushed_lru  = buf_flush_LRU_list(buf_pool);
```

执行LRU列表的刷新。但除buffer_pool外没有其他参数，基于lru列表会刷新多少页？

```
slot->succeeded_list  = buf_flush_do_batch(
buf_pool, BUF_FLUSH_LIST,
slot->n_pages_requested,
page_cleaner->lsn_limit,
&slot->n_flushed_list);
```

执行“脏页”列表的批量刷新。入参slot->n_pages_requested为前面的刷新建议函数为该buffer pool instance生成的建议刷新的页面数。

4.4.1 buf_flush_LRU_list函数

```
Clears up tail of the LRU list of a given buffer pool instance:
* Put replaceable pages at the tail of LRU to the free list
* Flush dirty pages at the tail of LRU to the disk
The depth to which we scan each buffer pool is controlled by dynamic
config parameter innodb_LRU_scan_depth.
@param buf_pool buffer pool instance
@return total pages flushed */
static
ulint
buf_flush_LRU_list(
        buf_pool_t*     buf_pool)
{
        ulint   scan_depth, withdraw_depth;
        ulint   n_flushed = 0;

        ut_ad(buf_pool);

        /* srv_LRU_scan_depth can be arbitrarily large value.
        We cap it with current LRU size. */
        buf_pool_mutex_enter(buf_pool);
        scan_depth = UT_LIST_GET_LEN(buf_pool->LRU);
        if (buf_pool->curr_size < buf_pool->old_size
            && buf_pool->withdraw_target > 0) {
                withdraw_depth = buf_pool->withdraw_target
                                 - UT_LIST_GET_LEN(buf_pool->withdraw);
        } else {
                withdraw_depth = 0;
        }
        buf_pool_mutex_exit(buf_pool);

        if (withdraw_depth > srv_LRU_scan_depth) {
                scan_depth = ut_min(withdraw_depth, scan_depth);
        } else {
                scan_depth = ut_min(static_cast<ulint>(srv_LRU_scan_depth),
                                    scan_depth);
        }

        /* Currently one of page_cleaners is the only thread
        that can trigger an LRU flush at the same time.
        So, it is not possible that a batch triggered during
        last iteration is still running, */
        buf_flush_do_batch(buf_pool, BUF_FLUSH_LRU, scan_depth,
                           0, &n_flushed);

        return(n_flushed);
}
```

scan_depth第一次取值来源于buffer pool instance里的LRU列表的长度，也就列表中页面数量。

scan_depth第二次取值来自srv_LRU_scan_depth与LRU列表长度比较后的小值。也就是当LRU列表的长度大于参数innodb_lru_scan_depth(默认1024）时，等于innodb_lru_scan_depth，否则等于LRU长度。

scan_depth作为参数，传递给buf_flush_do_batch函数，再经过几次传递之后，最后传递给**buf_flush_LRU_list_batch 函数的max 参数变量**

4.4.1.1 buf_flush_LRU_list_batch函数

```
for (bpage = UT_LIST_GET_LAST(buf_pool->LRU);
     bpage != NULL && count + evict_count < max
     && free_len < srv_LRU_scan_depth + withdraw_depth
     && lru_len > BUF_LRU_MIN_LEN;
     ++scanned,
     bpage = buf_pool->lru_hp.get()) {

        buf_page_t* prev = UT_LIST_GET_PREV(LRU, bpage);
        buf_pool->lru_hp.set(prev);

        BPageMutex*     block_mutex = buf_page_get_mutex(bpage);

        mutex_enter(block_mutex);

        if (buf_flush_ready_for_replace(bpage)) {
                /* block is ready for eviction i.e., it is
                clean and is not IO-fixed or buffer fixed. */
                mutex_exit(block_mutex);
                if (buf_LRU_free_page(bpage, true)) {
                        ++evict_count;
                }
        } else if (buf_flush_ready_for_flush(bpage, BUF_FLUSH_LRU)) {
                /* Block is ready for flush. Dispatch an IO
                request. The IO helper thread will put it on
                free list in IO completion routine. */
                mutex_exit(block_mutex);
                buf_flush_page_and_try_neighbors(
                        bpage, BUF_FLUSH_LRU, max, &count);
        } else {
                /* Can't evict or dispatch this block. Go to
                previous. */
                ut_ad(buf_pool->lru_hp.is_hp(prev));
                mutex_exit(block_mutex);
        }

        ut_ad(!mutex_own(block_mutex));
        ut_ad(buf_pool_mutex_own(buf_pool));

        free_len = UT_LIST_GET_LEN(buf_pool->free);
        lru_len = UT_LIST_GET_LEN(buf_pool->LRU);
}
```

从buffer的LRU列表末尾取出block,满足下面所有条件，才进入循环体：

1.如果bpage存在。

2.且count+evict_count <max, (max为入参，由scan_depth传入）。

3.free列表的长度小于srv_LRU_scan_depth,即参数innodb_lru_scan_depth。withdraw_depth这个变量常规情况下都为0，忽略。

4.lru_len长度大于BUF_LRU_MIN_LEN（该值512）

反过来讲：

1.如果free列表的长度大于innodb_lru_scan_depth，则中止循环。

2.被替换（evict_count)+被刷新(count)的页面数最多为scan_depth.

所以，**innodb_lru_scan_depth 参数，在此起非常关键的作用，实际上也影响了buffer bool的free列表的长度。**

如果是一个可替换的页，则执行函数buf_LRU_free_page,将其加入free列表。

如果是脏页，则调用函数buf_flush_page_and_try_neighbors进行刷新。

4.4.2 buf_flush_do_batch函数

该函数在pc_flush_slot中的被调用语句：

```
slot->succeeded_list  = buf_flush_do_batch(
                buf_pool, BUF_FLUSH_LIST,
                slot->n_pages_requested,
                page_cleaner->lsn_limit,
                &slot->n_flushed_list);
```

参数解释：

buf_pool:需要进行刷新的缓存池实例指针。

slot->n_pages_requested： 为page_cleaner_flush_pages_recommendation 函数为该buffer_pool建议的需要被刷新的页面数量，实际刷新的页面并不一定等于该值。后面将详细介绍。

page_cleaner->lsn_limit： 也为recommendation函数最后部分的语句所赋值*lsn_limit = LSN_MAX。

BUF_FLUSH_LIST： 刷新类型，“脏”列表，即对“脏”页列表进行刷新。

&slot->n_flushed_list： 作为返回参数，被赋值该buffer bool instance实际的被刷新的页面数。

其最终调用buf_do_flush_list_batch(buf_pool, min_n, lsn_limit);来进行刷新

其中min_n来自**slot->n_pages_requested，**

lsn_limit来自page_cleaner->lsn_limit，即LSN_MAX

4.4.2.1 buf_do_flush_list_batch函数

```
ulint
buf_do_flush_list_batch(
        buf_pool_t*             buf_pool,
        ulint                   min_n,
        lsn_t                   lsn_limit)
{
        ulint           count = 0;
        ulint           scanned = 0;
        /* Start from the end of the list looking for a suitable
        block to be flushed. */
        buf_flush_list_mutex_enter(buf_pool);
        ulint len = UT_LIST_GET_LEN(buf_pool->flush_list);

        /* In order not to degenerate this scan to O(n*n) we attempt
        to preserve pointer of previous block in the flush list. To do
        so we declare it a hazard pointer. Any thread working on the
        flush list must check the hazard pointer and if it is removing
        the same block then it must reset it. */
        for (buf_page_t* bpage = UT_LIST_GET_LAST(buf_pool->flush_list);
             count < min_n && bpage != NULL && len > 0
             && bpage->oldest_modification < lsn_limit;
             bpage = buf_pool->flush_hp.get(),
             ++scanned) {

                buf_page_t*     prev;

                ut_a(bpage->oldest_modification > 0);
                ut_ad(bpage->in_flush_list);

                prev = UT_LIST_GET_PREV(list, bpage);
                buf_pool->flush_hp.set(prev);
                buf_flush_list_mutex_exit(buf_pool);

                buf_flush_page_and_try_neighbors(
                        bpage, BUF_FLUSH_LIST, min_n, &count);
                buf_flush_list_mutex_enter(buf_pool);

                --len;
        }

        buf_pool->flush_hp.set(NULL);
        buf_flush_list_mutex_exit(buf_pool);

        return(count);
}
```

循环中止条件：

直到 count >= min_n 或者脏页列表为空。即所刷新的page等于page_cleaner_flush_pages_recommendation函数所建议的刷新数量，或者“脏”页列表为空。因为page cleaner线程调用该函数做批量刷新的时候，lsn_limit 参数值为极大值，因此无需考虑page的oldest_modification。

4.5 等待所有缓冲池实例刷新完成—pc_wait_finished

在buf_flush_page_cleaner_coordinator函数的调用如下:

```
pc_wait_finished(&n_flushed_lru,  &n_flushed_list);
```

注释：

n_flushed_lru:用来接收从lru列表中刷新了多少页面。

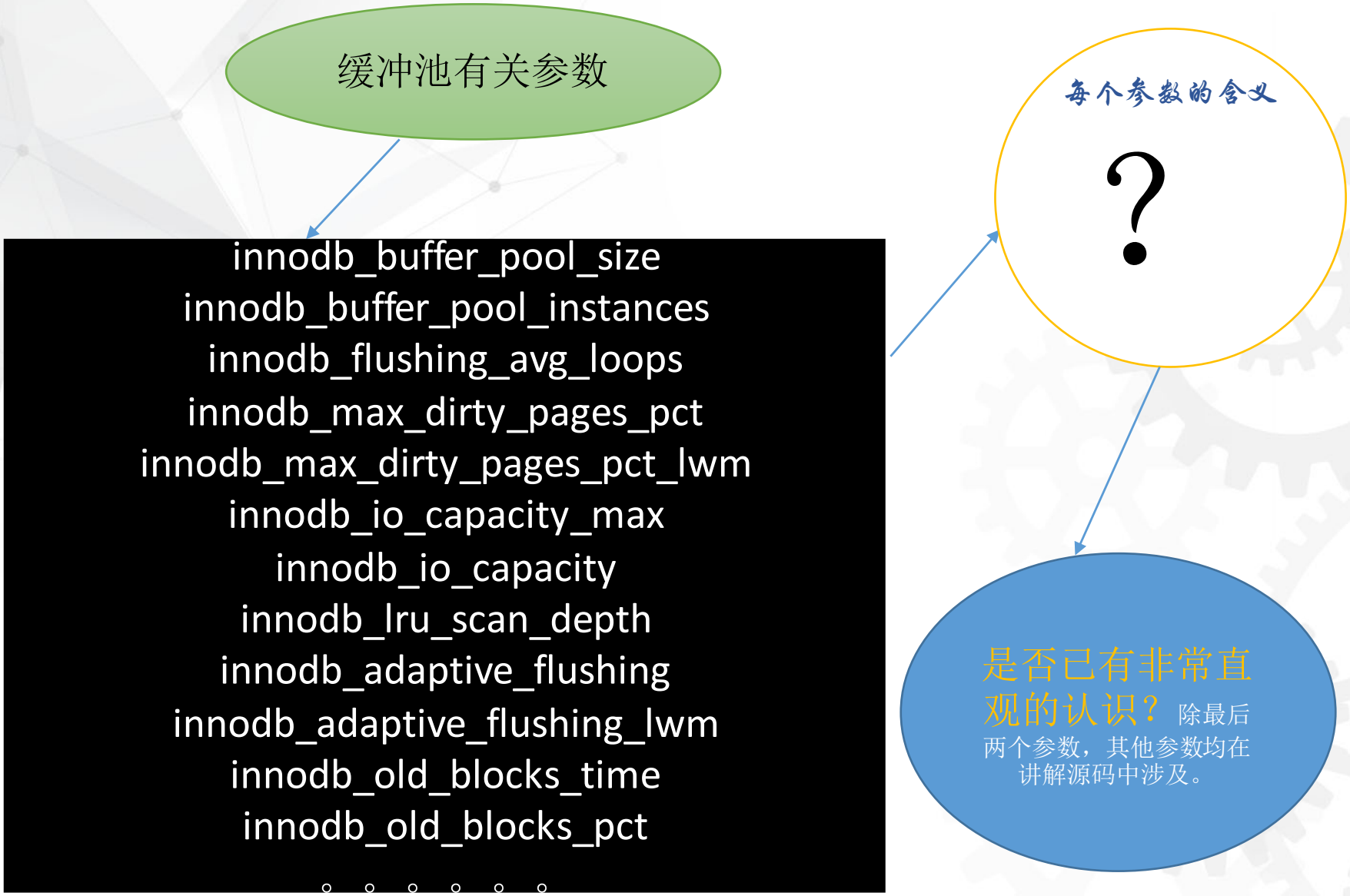
n_flushed_list:用于接收从”脏”页列表（flush列表）刷新了多少页面。

在pc_wait_finished函数体内

```
os_event_wait(page_cleaner->is_finished);
```

等待被刷新完成事件唤醒，如果需要刷新的数量比较多，而磁盘的io能力比较差等，将导致刷新不能及时完成。下一次刷新循环无法“准时”进行。如果相邻两次刷新的启动时间间隔4000ms,错误日志将有相关提示信息。

回顾“3.4 缓存池的有关参数”中提到的参数




THANKS
SequeMedia
盛拓传媒
ITPUB
ChinaUnix