

# 第三十讲：InnoDB Buffer Pool 的设计与实现

知春路遇上八里桥

<2025-02-16 Sun>



1 BufferPool 引入

2 BufferPool 启动

3 BufferPool 中的页面读写

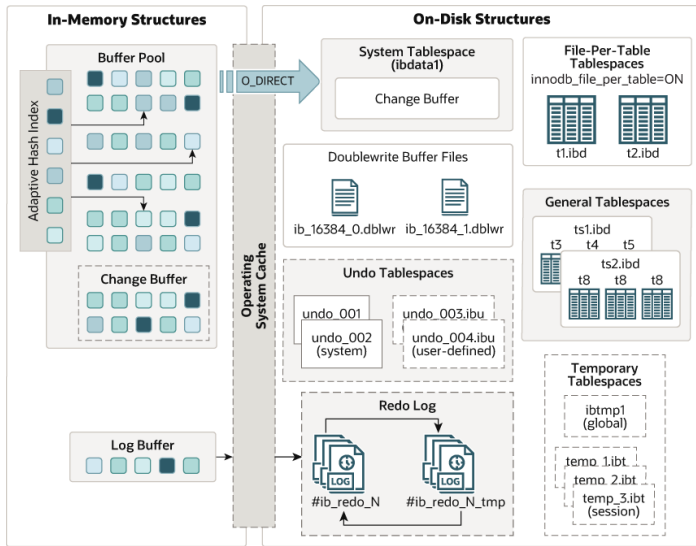


1

## BufferPool 引入

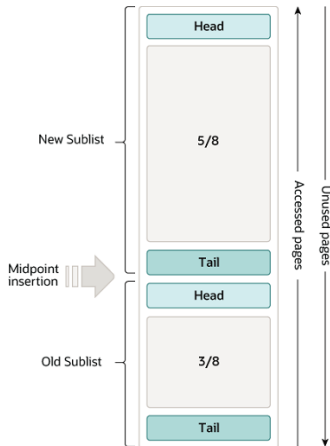


# InnoDB 架构



# BufferPool 设计

- BufferPool<sup>1</sup> 是缓存 InnoDB 表和索引数据的内存结构
  - ▶ InnoDB 没有使用操作系统的缓存, 自己实现了一套 BufferPool
  - ▶ 缓存数据到内存中可以提高数据库的性能
  - ▶ BufferPool 是 MySQL 中最占用内存的内存模块
  - ▶ 在专用数据库服务器中, 80% 的内存都被 BufferPool 使用
- BufferPool 相关的核心概念
  - ▶ Buffer Pool Instance, Buffer Pool 实例
  - ▶ Buffer Chunks, 包括两部分:
    - ① 数据页和数据页对应的控制体, 控制体中有指针指向数据页
    - ② Buffer Chunks 是操作系统最低层的物理块
    - ③ 在启动阶段从操作系统申请, 直到数据库关闭才释放
  - ▶ Page 数据页, 与磁盘结构中的 Page 一致
  - ▶ 常见链表: free list / LRU list / flush list 等



<sup>1</sup><https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html>



# 查看 BufferPool 状态

```
mysql> show engine innodb status\G
```

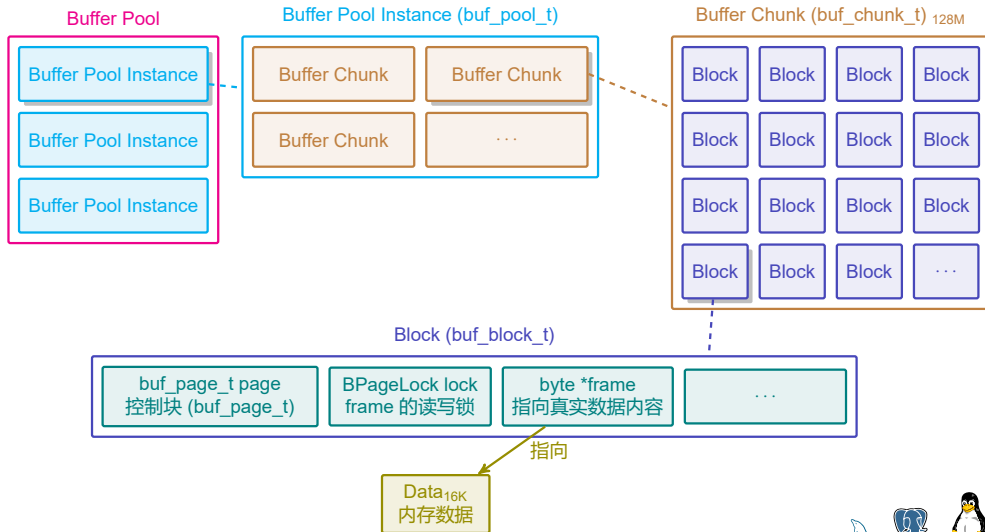
此处省略一些其他输出

## ----- BUFFER POOL AND MEMORY -----

```
Total large memory allocated 0  
Dictionary memory allocated 639306  
Buffer pool size      8192  
Free buffers          5687  
Database pages        2496  
Old database pages    941  
Modified db pages     0  
Pending reads         0  
Pending writes: LRU 0, flush list 0, single page 0  
Pages made young 0, not young 0  
0.00 youngs/s, 0.00 non-youngs/s  
Pages read 2354, created 142, written 201  
38.27 reads/s, 0.00 creates/s, 0.03 writes/s  
Buffer pool hit rate 570 / 1000, young-making rate 0 / 1000 not 0 / 1000  
Pages read ahead 31.96/s, evicted without access 0.00/s, Random read ahead 0.00/s  
LRU len: 2496, unzip_LRU len: 0  
I/O sum[0]:cur[0], unzip sum[0]:cur[0]
```



# BufferPool 中的逻辑结构



# 重要数据结构说明

- ① `buf_pool_t`
  - ▶ 存储 BufferPool 实例级别的控制信息，例如 `mutex`，`instance_no` 等
  - ▶ 还存储了各种逻辑链表的链表根节点，例如：Zip Free 这个二维数组也在其中
- ② `buf_chunk_t`
  - ▶ 是一块连续分配的内存
  - ▶ 由若干 `block` 组成，多个 `chunk` 构成了 `buf_pool_t` 实例
- ③ `buf_block_t`
  - ▶ 这个就是数据页的控制体，用来描述数据页部分的信息（大部分信息在 `buf_page_t` 中）
  - ▶ `buf_block_t` 中首个字段就是 `buf_page_t`，放在首位为了 `buf_block_t` 和 `buf_page_t` 相互转换
  - ▶ `buf_block_t` 中另一个重要字段是 `frame`，指向真正存数据的数据页
  - ▶ `buf_block_t` 还存储了 Unzip LRU List 链表的根节点
  - ▶ 另外一个比较重要的字段就是 `block` 级别的 `mutex`
- ④ `buf_page_t`
  - ▶ 这个可以理解为另外一个数据页的控制体，大部分的数据页信息存在其中
  - ▶ 压缩页的信息包括压缩页的大小，压缩页的数据指针
  - ▶ 如果某个压缩页被解压了，解压页的数据指针是存储在 `buf_block_t` 的 `frame` 字段里



# BufferPool 核心的数据结构

⑤ buf_block_t
General fields (通用字段)
<ul style="list-style-type: none"><li>buf_page_t page</li><li>BPageLock lock</li><li>byte *frame</li><li>UT_LIST_NODE_T(buf_block_t) unzip_LRU</li><li>bool in_unzip_LRU_list</li><li>bool in_withdraw_list</li></ul>
Optimistic search field
<ul style="list-style-type: none"><li>uint64_t modify_clock</li></ul>
其他
<ul style="list-style-type: none"><li>BPageMutex mutex</li><li>const page_id_t &amp;get_page_id()</li><li>page_no_t get_page_no()</li><li>page_no_t get_next_page_no()</li><li>page_no_t get_prev_page_no()</li><li>page_type_t get_page_type()</li><li>const char *get_page_type_str()</li><li>page_zip_des_t *get_page_zip()</li><li>page_zip_des_t const *get_page_zip()</li></ul>

③ buf_page_t
磁盘结构管理
<ul style="list-style-type: none"><li>uint16_t get_dblwr_batch_id()</li><li>space_id_t space()</li><li>page_no_t page_no()</li></ul>
General fields (通用字段)
<ul style="list-style-type: none"><li>lsn_t get_newest_lsn()</li><li>lsn_t get_oldest_lsn()</li><li>bool is_dirty()</li><li>void set_newest_lsn(lsn_t lsn)</li><li>void set_oldest_lsn(lsn_t lsn)</li><li>void set_clean()</li><li>page_id_t id</li><li>page_size_t size</li><li>buf_fix_count_atomic_t buf_fix_count</li><li>buf_page_state state</li><li>buf_flush_t flush_type</li><li>uint8_t buf_pool_index</li></ul>
Page flushing algorithm (刷脏页算法)
<ul style="list-style-type: none"><li>lsn_t newest_modification</li><li>lsn_t oldest_modification</li></ul>
LRU replacement algorithm (LRU替换算法)
<ul style="list-style-type: none"><li>UT_LIST_NODE_T(buf_page_t) LRU</li><li>page_zip_des_t zip</li><li>Flush_observer *m_flush_observer</li><li>fl_space_t *m_space</li><li>uint32_t freed_page_clock</li></ul>
其他
<ul style="list-style-type: none"><li>uint32_t m_version</li><li>std::chrono::steady_clock::time_point access_time</li><li>uint16_t m_dblwr_id</li><li>bool old</li><li>bool file_page_was_freed</li><li>bool in_flush_list</li><li>bool in_free_list</li><li>bool in_LRU_list</li><li>bool in_page_hash</li><li>bool in_zip_hash</li></ul>

⑤ buf_pool_t
互斥锁
<ul style="list-style-type: none"><li>BufListMutex chunks_mutex</li><li>BufListMutex LRU_list_mutex</li><li>BufListMutex free_list_mutex</li><li>BufListMutex zip_free_mutex</li><li>BufListMutex zip_hash_mutex</li></ul>
状态信息
<ul style="list-style-type: none"><li>uint instance_no</li><li>uint curr_pool_size</li><li>uint LRU_old_ratio</li><li>volatile uint n_chunks</li><li>volatile uint n_chunks_new</li><li>buf_chunk_t *chunks</li><li>buf_chunk_t *chunks_old</li><li>uint curr_size</li><li>uint old_size</li><li>page_no_t read_ahead_area</li><li>hash_table_t *page_hash</li><li>hash_table_t *zip_hash</li><li>std::atomic&lt;uint&gt; n_pending_reads</li><li>std::atomic&lt;uint&gt; n_pending_unzip</li><li>std::chrono::steady_clock::time_point last_printout_time</li><li>buf_buddy_stat_t buddy_stat[BUF_BUDDY_SIZES_MAX + 1]</li><li>buf_pool_stat_t stat</li><li>buf_pool_stat_t old_stat</li></ul>
Page flushing algorithm (刷脏页算法)
<ul style="list-style-type: none"><li>BufListMutex flush_list_mutex</li><li>FlushHp flush_hp</li><li>FlushHp oldest_hp</li><li>UT_LIST_BASE_NODE_T(buf_page_t, lsn) flush_list</li><li>bool init_flush(BUF_FLUSH_N_TYPES)</li><li>uint n_flush[BUF_FLUSH_N_TYPES]</li><li>os_event_t no_flush[BUF_FLUSH_N_TYPES]</li><li>lib_rbt_t *flush_rbt</li><li>uint freed_page_clock</li><li>bool try_LRU_scan</li><li>lsn_t track_page_lsn</li><li>lsn_t max_lsn_lo</li></ul>
LRU replacement algorithm (LRU替换算法)
<ul style="list-style-type: none"><li>UT_LIST_BASE_NODE_T(buf_page_t, lsn) free</li><li>UT_LIST_BASE_NODE_T(buf_page_t, lsn) withdraw</li><li>uint withdraw_target</li><li>LRUHp lru_hp</li><li>LRUltr lru_scan_itr</li><li>LRUltr single_scan_itr</li><li>UT_LIST_BASE_NODE_T(buf_page_t, LRU) LRU</li><li>buf_page_t *LRU_old</li><li>uint LRU_old_len</li><li>UT_LIST_BASE_NODE_T(buf_block_t, unzip_LRU) unzip_LRU</li></ul>
Buddy allocator (伙伴分配器)
<ul style="list-style-type: none"><li>UT_LIST_BASE_NODE_T(buf_buddy_free_t, lsn) zip_free[BUF_BUDDY_SIZES_MAX]</li><li>buf_page_t *watch</li><li>bool allocate_chunk(uint64_t mem_size, buf_chunk_t *chunk)</li><li>void deallocate_chunk(buf_chunk_t *chunk)</li><li>bool madvise_dump()</li><li>bool madvise_dont_dump()</li></ul>

⑤ buf_chunk_t
<ul style="list-style-type: none"><li>uint size</li><li>unsigned char *mem</li><li>buf_block_t *blocks</li></ul>
<ul style="list-style-type: none"><li>size_t mem_size()</li><li>bool contains(const buf_block_t *ptr)</li></ul>

# BufferPool 页面管理功能概述

- ❶ page hash (快速访问)
  - ▶ 所有的页都由一张哈希表组织, 叫 `page_hash`, 可快速的访问到一个页
  - ▶ `page_hash` 的 key 就是 `page_id`
- ❷ free list (页面申请)
  - ▶ 当页面加载到 Buffer Pool 中后, 需要为其分配一个 `buf_block_t` 结构作为描述符
  - ▶ free list 记录所有分配空闲的未被占用的 `buf_block_t` 描述符
- ❸ LRU list (页面淘汰)
  - ▶ Buffer Pool 的空间是有限的, 由系统变量 `innodb_buffer_pool_size`<sup>2</sup> 控制
  - ▶ 当没有空间来存放从磁盘载入的页时, 便需要把现有的一些页淘汰掉。使用的就是 LRU 算法
- ❹ flush list (脏页写回)
  - ▶ flush list 记录 Buffer Pool 中的所有脏页
  - ▶ 同步刷脏, 在执行 Query 是调用 `buf_flush_page()` 写盘
  - ▶ 异步刷脏, `page cleaner` 线程定期的将其写回到磁盘

<sup>2</sup><https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool-resize.html>



2

## BufferPool 启动



# 初始化 BufferPool

- mysql 启动时调用 buf\_pool\_init() 开启多线程创建 BufferPool 实例

```
1437  /** Creates the buffer pool.  
1438  @param[in]  total_size    Size of the total pool in bytes.  
1439  @param[in]  n_instances  Number of buffer pool instances to create.  
1440  @return DB_SUCCESS if success, DB_ERROR if not enough memory or error */  
1441  dberr_t buf_pool_init(ulint total_size, ulint n_instances) {
```

- buf\_pool\_init() 开启多线程快速初始化实例逻辑 (代码有删减)

```
    for (i = 0; i < n_instances; /* no op */) {  
        std::vector<std::thread> threads;  
        for (ulint id = i; id < n; ++id) {  
            threads.emplace_back(std::thread(buf_pool_create, &buf_pool_ptr[id], size,  
                                              id, &m, std::ref(errs[id])));  
        }  
  
        for (ulint id = i; id < n; ++id) {  
            threads[id - i].join();  
        }  
  
        /* Do the next block of instances */  
        i = n;  
    }
```



# 创建 BufferPool 实例

- 每个线程调用 `buf_pool_create()` 创建一个 BufferPool 实例, 见 [.../buf/buf0buf.cc](#)

```
1193  /** Initialize a buffer pool instance.  
1194  @param[in]      buf_pool      buffer pool instance  
1195  @param[in]      buf_pool_size size in bytes  
1196  @param[in]      instance_no   id of the instance  
1197  @param[in,out]  mutex        Mutex to protect common data structures  
1198  @param[out] err              DB_SUCCESS if all goes well */  
1199  static void buf_pool_create(buf_pool_t *buf_pool, ulint buf_pool_size,  
1200                             ulint instance_no, std::mutex *mutex,  
1201                             dberr_t &err) {
```

- 调用 `buf_chunk_init()` 为 BufferPool 初始化 Buffer Chunks
- 调用 `buf_block_init()` 初始化 Buffer Block 控制块
- 最终存储在全局变量中, 具体查看代码 [storage/innobase/include/buf0buf.h](#)

```
114  /** The buffer pools of the database */  
115  extern buf_pool_t *buf_pool_ptr;
```



# 3

## BufferPool 中的页面读写



# 读取 Page

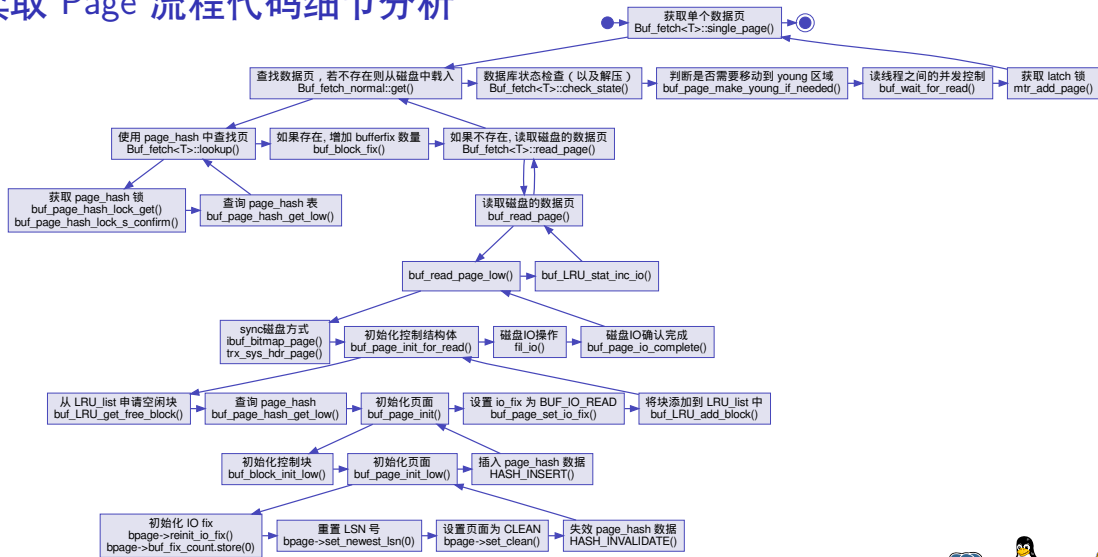
- `buf_page_get()` 是其他模块获取数据页的外部接口函数
  - ▶ `buf_page_get()` 是一个宏定义, 真正的函数为 `buf_page_get_gen`
- `buffer_page_get_gen()` 见代码 [storage/innobase/include/buf0buf.h](#)

```
417  buf_block_t *buf_page_get_gen(const page_id_t &page_id,  
418                               const page_size_t &page_size, ulint rw_latch,  
419                               buf_block_t *guess, Page_fetch mode,  
420                               ut::Location location, mtr_t *mtr,  
421                               bool dirty_with_no_latch = false);
```

- 其中 `page_id` 获取指定的页 / `page_size` 获取指定的页大小
- 其中 `mode` 表示读取模式, NORMAL, SCAN, IF\_IN\_POOL 等
  - ① NORMAL: 若不在 BufferPool 中则从磁盘文件载入到 BufferPool 中
  - ② SCAN: 同 NORMAL, 但是暗示这是一次大规模的扫描 (例如 Table Scan)
  - ③ IF\_IN\_POOL: 只在 BufferPool 中查找没, 如果不在则返回
  - ④ PEEK\_IF\_IN\_POOL: 同 IF\_IN\_POOL, 但不要将数据页放在 LRU list young 区域
  - ⑤ IF\_IN\_POOL\_OR\_WATCH: 同 IF\_IN\_POOL, 但如果不在则设置为 “watch”



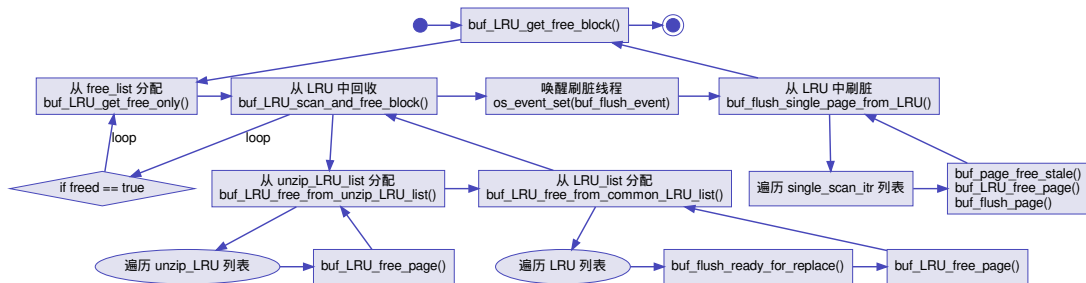
# 读取 Page 流程代码细节分析





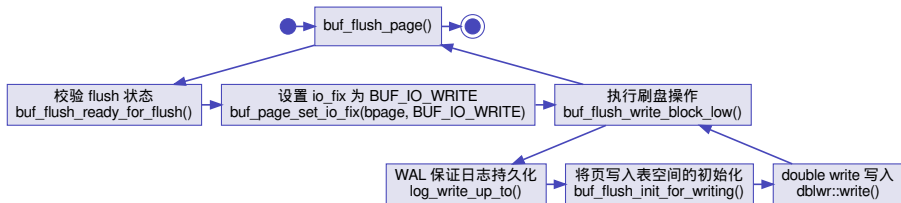
# 申请 Block

- 从 BufferPool 中申请 1 个 Block 供后续的 Page 读取使用
- 入口函数 buf\_LRU\_get\_free\_block() 流程如下



# 写入 Page

- 同步刷脏的入口函数为 `buf_flush_page()`，同步刷脏过程仅会刷 1 个 Page
- InnoDB 通过严格的 WAL 机制保证数据的一致性，刷脏过程同样如此
  - ▶ DoubleWrite 保证 16KB 数据的原子性，不会因为写了 8KB 导致无法恢复
  - ▶ 往磁盘上写 16KB 的数据页并不保证是原子的，一般先写 DoubleWrite 后写数据页
    - ① 使用同步 IO (`fil_io` 参数 `sync = true`) 将脏页写到系统表空间中的 DoubleWrite 区域
    - ② 使用异步 IO (`fil_io` 参数 `sync = false`) 将脏页写到用户表空间中
- 需要保证对应的 REDO 日志文件落盘，然后再写入数据页



# 结束

