PostgreSQL 深入理解缓冲区

作者：李海翔

博客：http://blog.163.com/li_hx/

邮箱：lhx3000@163.com

1 什么是缓冲区

百度上讲：http://baike.baidu.com/view/266782.htm[①]

缓冲器为暂时置放输出或输入资料的内存。

缓冲器内资料自储存设备（如硬盘）来，放置在缓冲器中，须待机送至 CPU 或其他运算设备。

缓冲区(buffer)这个中文译意源自当计算机的高速部件与低速部件通讯时,必须将高速部件的输出暂存到某处,以保证高速部件与低速部件相吻合. 后来这个意思被扩展了,成为"临时存贮区"的意思。

确如上述，缓冲区，在计算机中，就是一个高速部件和低速部件的一个中介，如内存，是 CPU 和外存设备（硬盘）等中转站。如果数据被预先存入内容，CPU 读取到的数据的速度就会快许多。再如 CPU 上的高速 cache，异曲同工。

2 常见的缓冲区

2.1 常见缓冲区类型

只要存在衔接，在高低层次间有交互，中间地带必然存在，如同边界线上的军事缓冲区一样，缓冲区的存在，尤其实质意义。计算机上的缓存，可以有效提高数据的吞吐效率。

硬件级：主板的缓存、CPU 中的缓存

操作系统层：操作系统的缓冲区、网络协议层的缓冲区

应用程序层：应用程序的缓冲区、数据库系统的缓冲区（大型应用程序，通常都会自己管理内存，自己提供缓冲区管理的方式）

2.2 缓冲区的衍生---缓存组件

一些网站，为了提高访问速度，常使用一些缓存组件如 memcached 和 ehcache。

常见的还有 SysCache，MemCache，Prevalence 等等。

Memcached，一个高性能的分布式的内存缓存系统，是一个开源组件。

这些组件的研究，我们不深究，列在此，是扩展对于缓冲区的认识。

[①] http://www.hudong.com/wiki/%E7%BC%93%E5%86%B2%E5%8C%BA，也是一样的文字

2.3 缓存与池的关系

池，停水曰池。——《广韵》。

所以，池，能蓄积物。引申后，池能蓄积线程，则称之为线程池；池能蓄积进程则称之为进程池。

另外常见的还有数据库连接池、IIS6.0 中的地址池、内存池[①]等等。

其中，内存池的概念，类似缓冲区。

我这里讨论的，着重于 PostgreSQL 数据库系统的实现实例。

2.4 缓存区与 cache 辨析

Cache，偏于硬件系统，称为高速缓冲存储器。一个高速且有着较小容量的、可以用高速的静态存储器芯片实现的存储器。主要是集成到 CPU 芯片内部，存储 CPU 最经常访问的指令或者操作数据。

所以，本文所讲的缓冲区，有别于 cache。

3 缓存的作用和常见场景

3.1 缓存的作用

缓存，可以把常用的一些“对象”（暂且称为对象）蓄积起来，供使用，这样，将减少被缓存对象的生成、销毁等时间，如果对象被反复使用，且其生成、销毁需要花费时间，则缓存这样对象对于时间的节省可能很有效。

缓存组件提供 retrieve、 add、remove 等功能。过期和自我清除策略也是给缓存管理的一部分。

缓存带来的明显好处是：

提高性能。采用一种更为密切的方式，为数据的消费者存储相关的数据；以避免重复性的数据创建、处理和传输。使用缓存，可确保一些应用能够在网络的的延时、WEB 服务和硬件的问题中节约大量的资源（时间等）。

[①] http://blogold.chinaunix.net/u3/112227/showart_2262661.html

3.2 缓存的常用场景

缓存适用于以下情况：

必须重复的访问静态的或是很少变动的数据

数据访问在创建、访问和传输、销毁上花费很大

4 数据库中的缓冲区

在一个 DBMS 系统中，缓冲区经常被使用，用以提高系统的性能。

如，PostgreSQL 的“src\backend\utils\cache”下，就有如下文件：

文件名称	功能
attoptcache.c	列信息的缓存（表对象的列），数据源自 pg_attribute 系统表
catcache.c	每个系统表抽象为一个 catalog，对于每个系统表都进行缓存，被 syscache.c 中的函数调用，以缓存所有的系统表
inval.c	非某个具体的缓存。是缓存失效调度代码
lsyscache.c	非某个具体的缓存。是从系统缓存（syscache.c）获取信息的函数集合，如取得函数名称，如取得一个表的列数
plancache.c	查询计划缓存，对于查询计划，如果能复用，可以节约查询计划的生成时间。PG 提供了查询计划缓存机制，很多数据库还有“结果集缓存”，目的是为了减少数据库的逻辑操作
relcache.c	系统文件缓存，如全局的“global”数据库的信息
relmapper.c	系统文件缓存的辅助文件，做一些文件的加载、关闭系统时的写出等操作
spccache.c	表空间缓存，对于表空间信息缓存（tablespace）进行的管理
syscache.c	缓存所有的系统表（可关注 InitCatalogCache 函数、cacheinfo 结构。所有需要被缓存的系统表，都在 cacheinfo 结构中注册）。系统表使用的接口都在本文件中定义
ts_cache.c	全文检索缓存，对于全文检索缓存管理的支持
typcache.c	数据类型缓存，数据来源于 pg_type 系统表（记录系统提供的数据类型和用户自定义的类型）

4.1 PostgreSQL 数据缓冲区管理

4.1.1 文件介绍

位置：src/backend/storage/buffer/

文件名称	功能
buf_init.c	Buf 初始化功能，初始化出三处内存空间，分别被 BufferDescriptors、BufferBlocks、SharedBufHash、StrategyControl 描述，注意这四个变量，他们表述了数据缓冲区的基本结构
buf_table.c	Buf 管理的辅助文件，完成对 SharedBufHash 变量（此变量便于对缓冲区中的缓存块进行查找使用）表示的内存的管理操作（如初始化、查找、插入、删除等），主要有如下函数调用： SharedBufHash InitBufTable BufTableHashCode BufTableLookup BufTableInsert BufTableDelete
bufmgr.c	Buf 的管理文件，完成对 buf 的管理操作，如 buf 的分配、回收等。主要的一些函数如 ReadBuffer、ReadBufferExtended 、ReleaseBuffer、MarkBufferDirty 等
freelist.c	Buf 替换策略相关代码，完成对缓冲区替换策略的管理，主要有函数 AddBufferToRing、FreeAccessStrategy、GetAccessStrategy、GetBufferFromRing、StrategyFreeBuffer、StrategyGetBuffer、StrategyInitialize、StrategyRejectBuffer、StrategyShmemSize、StrategySyncStart
localbuf.c	本地缓存管理。本地缓存，指的是对临时表的管理（PG 有临时表的概念，create temp table，这些临时表被创建即进入内存，实则进入缓存）

4.1.2 相关代码

1 **数据访问层**，位于 buffer 上层的代码：

位于 buffer 上层的代码，是**数据访问层**，主要位于：src/backend/access/；这一层，主要是数据库引擎中对于数据的逻辑使用的层次，如用户查询数据，数据通过索引被访问到，索引不是直接从存储介质上请求 IO，而是直接从 buffer 层中的 buf 中读取，如果 buf 中没有，则由 buf 层负责向底层（**数据存储层**）要求相应数据。

查询 ReadBuffer 函数的调用关系可以更好理解。

buff-access.Gif

另外，还可以参阅 ReadBufferExtended 函数的调用管理，理解 buf 作为衔**接数据访问层**和

数据存储层的作用。

2 **数据存储层**，位于 buffer 下层的代码：

位于 buffer 下层的代码，是数据存储层，主要位于 src/backend/storage/smgr；这一层，主要是数据库数据的存储层，直接和存储介质打交道（实则是和 OS 的 IO 调度交互）。

注意，在 ReadBufferExtended 函数极其调用的 ReadBuffer_common 子函数中，可能都涉及类似 smgrXXX 函数的调用，如 smgrread，这是 buf 层的函数发现 buf 中没有相应的数据可向**数据访问层**提供，则 buf 管理器直接向**数据库存储层**要求 IO，使得被要求的数据能够进入 buf。

关键代码如下（src/backend/storage/smgr/smgr.c）：

```
typedef struct f_smgr
{
    void        (*smgr_init) (void);/* may be NULL */
    void        (*smgr_shutdown) (void);            /* may be NULL */
    void        (*smgr_close) (SMgrRelation reln, ForkNumber forknum);
    void        (*smgr_create) (SMgrRelation reln, ForkNumber forknum,
                                                    bool isRedo);
    bool        (*smgr_exists) (SMgrRelation reln, ForkNumber forknum);
    void        (*smgr_unlink) (RelFileNodeBackend rnode, ForkNumber forknum,
                                                    bool isRedo);
    void        (*smgr_extend) (SMgrRelation reln, ForkNumber forknum,
                                    BlockNumber blocknum, char *buffer, bool skipFsync);
    void        (*smgr_prefetch) (SMgrRelation reln, ForkNumber forknum,
                                                        BlockNumber blocknum);
    void        (*smgr_read) (SMgrRelation reln, ForkNumber forknum,
                                                  BlockNumber blocknum, char *buffer);
    void        (*smgr_write) (SMgrRelation reln, ForkNumber forknum,
                                    BlockNumber blocknum, char *buffer, bool skipFsync);
    BlockNumber (*smgr_nblocks) (SMgrRelation reln, ForkNumber forknum);
    void        (*smgr_truncate) (SMgrRelation reln, ForkNumber forknum,
                                                    BlockNumber nblocks);
    void        (*smgr_immedsync) (SMgrRelation reln, ForkNumber forknum);
    void        (*smgr_pre_ckpt) (void);            /* may be NULL */
    void        (*smgr_sync) (void);    /* may be NULL */
    void        (*smgr_post_ckpt) (void);           /* may be NULL */
} f_smgr;
```

存储层的再底层，具体的存储方式，PostgreSQL 是以文件方式存储数据的（注意有“表空间”的概念，他文另述），这些文件的读写管理，可以参照带有“md”起头的各种函数，如“mdinit”。这里，smgr 的管理（存储层的管理），使用了函数指针，巧妙实现了上下层之间的

解耦，使得存储层有效作为一个接口存在，使得真正的存储层，有条件以“插件”思维的方式，换做各种存储引擎。

```
static const f_smgr smgrsw[] = {
    /* magnetic disk */
    {mdinit, NULL, mdclose, mdcreate, mdexists, mdunlink, mdextend,
        mdprefetch, mdread, mdwrite, mdnblocks, mdtruncate, mdimmedsync,
        mdpreckpt, mdsync, mdpostckpt
    }
};
```

4.1.3 Buf 作用与位置

在存储层和数据引擎之间，是**数据缓冲区**，起着联系内外存储介质的作用。把数据从外存调入缓冲区，供数据库引擎读写。写数据，实则是改写缓冲区，然后把缓冲区标识为“脏”，在必要的时候（如缓冲区满需要新缓冲区、系统做 checkpoint 时被要求刷出内存的脏数据等），被刷出缓存（写被修改的数据到外存文件）。

数据访问层次图

4.1.3.1ReadBuffer 表明的层次关系

如下表，表述了 buf 被使用的关系，

ReadBufferExtended 被调用的情况	ReadBuffer 被调用的情况
ReadBufferExtended acquire_sample_rows ReadBuffer fsm_readbuf	ReadBuffer ReleaseAndReadBuffer ginPrepareFindLeafPage ginFindLeafPage

ginVacuumPostingTreeLeaves	ginFindParents
ginDeletePage	ginHeapTupleFastInsert
ginDeletePage	shiftList
ginDeletePage	ginInsertCleanup
ginScanToDelete	scanGetCandidate
ginbulkdelete	scanPendingInsert
ginbulkdelete	GinNewBuffer
ginbulkdelete	ginGetStats
ginvacuumcleanup	ginUpdateStats
gistvacuumcleanup	gistfindleaf
gistbulkdelete	gistFindPath
_hash_getinitbuf	gistFindCorrectParent
_hash_getnewbuf	gistnext
_hash_getbuf_with_strategy	gistNewBuffer
heapgetpage	gistContinueInsert
ReadBufferBI	_hash_getbuf
btvacuumscan	_hash_getnewbuf
btvacuumpage	heap_fetch
lazy_scan_hea	heap_hot_search
lazy_vacuum_heap	heap_get_latest_tid
count_nondeletable_pages	heap_delete
vm_readbuf	heap_update
	heap_lock_tuple
	heap_inplace_update
	ReadBufferBI
	RelationGetBufferForTuple
	_bt_getbuf
	DefineSequence
	read_info
	GetTupleForTrigger

从表中可以看出：

1. ReadBufferExtended 被 ReadBuffer 调用，是 ReadBuffer 的子函数，实则是 ReadBufferExtended 函数实现读缓存的功能
2. 无论是 ReadBufferExtended 还是 ReadBuffer，基本上都被 heapXXX、ginXXX 等类似函数调用，这些函数，都隶属于**数据访问层**，所以 buf 为数据访问层提供服务
3. 例如，heap_update 函数:
   a) heap_update 函数调用了 ReadBuffer 函数，从 heap_update 函数中可以看出其他函数调用 ReadBuffer 函数的方式、使用目的都是一致的。heap_update 函数通过 ReadBuffer 函数先取到合适的数据信息（page），然后修改 buf，使之页的标识变为“dirty”，即脏页，这样就有机会被缓存调度置换（写出）出内存，修改的信息保存在外存。

b) 在 heap_update 函数中注意“RelationPutHeapTuple(relation, newbuf, heaptup); /* insert new tuple */”语句中三个参数的使用方式，逐一查看，可以了解 buf 的使用状况

c) “buffer = ReadBuffer(relation, ItemPointerGetBlockNumber(otid));”语句，跟踪 buffer 变量的用法和另外一个变量 newbuf 的用法，可以学习到缓存的使用、生成等原因和方式。

4.1.3.2从 buf 分配看 buf 的使用

BufferAlloc 函数，当有 buf 可以被分配时，从缓冲区拿出“空闲”（有个 freelist 结构）的缓存块进行分配，如果缓冲区已经满，则根据缓存淘汰算法，对缓存中的被选中的脏页进行刷出，然后分配给提出需求者。

BufferAlloc 被调用关系：

BufferAlloc

ReadBuffer_common

ReadBufferExtended

ReadBufferWithoutRelcache

BufferAlloc 代码分析：

1. 变量定义区，查看“volatile BufferDesc *buf;”的使用之处，可以更好把握 buf 的分配情况
2. “buf_id = BufTableLookup(&newTag, newHash);”，调用 BufTableLookup 函数确定要求的块是否在缓冲区中？
3. 接第二步，如果找到这样的缓存块，则作为 BufferAlloc 函数的返回值返回找到的缓存块
4. 接第二步，如果没有找到，反复循环，淘汰一些缓冲块出缓冲区（通过对 StrategyGetBuffer 函数的调用、对得到的 buf 判断是否是 dirty，如果是，则淘汰。注意 StrategyGetBuffer 函数中，对于缓冲区淘汰策略的使用方式，如何淘汰，取决于淘汰算法------仔细看 StrategyGetBuffer 函数的实现），直到找到一个可以分配的缓存块
5. 接第四步，为找到的缓存块，赋予恰当的值

4.1.3.3其他函数表明的 buf 作用

DropDatabaseBuffers

FlushDatabaseBuffers

通过阅读 BufferAlloc 函数，我们了解 buf 的分配。

```
 * BufferAlloc -- subroutine for ReadBuffer.  Handles lookup of a shared
 *      buffer.  If no buffer exists already, selects a replacement
 *      victim and evicts the old page, but does NOT read in new page.
```

4.1.4 Buf 结构

缓冲区：一块地址连续的空间，分为几种类型的缓冲区（元信息块、数据缓冲区块、其他信息块）
数据缓冲区块：地址连续的、多个缓存块的组合。内存中真正的数据存放区。缓冲区的组成之一。
缓存块：最小的缓存数据块的缓存单位。被数据缓存块包含。

4.1.4.1Buf 的整体结构

在 src/backend/storage/buffer/buf_init.c 中有 InitBufferPool 函数，描述了基本的 buf 结构：

```
void
InitBufferPool(void)
{
    bool        foundBufs,
                foundDescs;

    BufferDescriptors = (BufferDesc *)
        ShmemInitStruct("Buffer Descriptors",
                        NBuffers * sizeof(BufferDesc), &foundDescs);

    BufferBlocks = (char *)
        ShmemInitStruct("Buffer Blocks",
                        NBuffers * (Size) BLCKSZ, &foundBufs);
……// 略去其他代码
    /* Init other shared buffer-management stuff */
    StrategyInitialize(!foundDescs);
}
```

变量名称	空间大小	功能
BufferDescriptors	NBuffers * sizeof(BufferDesc)	缓冲区的元信息，表述数据库缓冲区分

		配出去的缓存块等的使用情况
BufferBlocks	NBuffers * (Size) BLCKSZ	形式上是“char *”，本质上是内存开辟出的一块地址连续的区域
SharedBufHash	NBuffers + NUM_BUFFER_PARTITIONS	为便于搜索 buf 的一个 hash 结构
StrategyControl	sizeof(BufferStrategyControl)	置换策略指示器。注意不要被变量名蒙蔽，和“缓存淘汰策略”并没有直接关系

4.1.4.2Buf 的元信息结构

Buf 的元信息数据块的结构信息如下，描述了每一个缓存块的使用情况：

```
typedef struct sbufdesc
{
    BufferTag    tag;            /* ID of page contained in buffer */ // 起着连接内存和外存地址映射的作用，通过这个，可以决定缓存块的脏信息写到外存的什么位置；如果是新读入的块，则根据新读入的块的地址给本变量赋值
    BufFlags flags;              /* see bit definitions above */
    uint16       usage_count;   /* usage counter for clock sweep code */
    unsigned refcount;       /* # of backends holding pins on buffer */
    int          wait_backend_pid;       /* backend PID of pin-count waiter */

    slock_t      buf_hdr_lock; /* protects the above fields */

    int          buf_id;             /* buffer's index number (from 0) */
    int          freeNext;       /* link in freelist chain */

    LWLockId     io_in_progress_lock;     /* to wait for I/O to complete */
    LWLockId     content_lock;  /* to lock access to buffer contents */
} BufferDesc;
```

结构中的其他成员，也很重要，如果想掌握全部用法，请核对代码，一一细看其调用方式。

第二个成员，标识了 buf 的状态，如下：

```
#define BM_DIRTY                (1 << 0)        /* data needs writing */
#define BM_VALID                (1 << 1)        /* data is valid */
#define BM_TAG_VALID            (1 << 2)        /* tag is assigned */
#define BM_IO_IN_PROGRESS       (1 << 3)        /* read or write in progress */
#define BM_IO_ERROR             (1 << 4)        /* previous I/O failed */
#define BM_JUST_DIRTIED         (1 << 5)        /* dirtied since write started */
```

```
#define BM_PIN_COUNT_WAITER     (1 << 6)    /* have waiter for sole pin */
#define BM_CHECKPOINT_NEEDED    (1 << 7)    /* must write for checkpoint */
```

不同状态，buf 有需要不同的处理。

需要注意的是，buf 层起着连接物理 IO 和上层数据访问的桥梁作用，所以，对上对下的重要信息，都在这个结构中。

如成员“io_in_progress_lock”，与物理 IO 相关，被调用关系如下：

sbufdesc.**io_in_progress_lock**

UnpinBuffer

WaitIO

InvalidateBuffer

StartBufferIO

WaitIO

StartBufferIO // 与缓存相关的一个重要函数，从物理存储往缓存块中加载数据

TerminateBufferIO

AbortBufferIO

InitBufferPool

sbufdesc

再有，与锁（content_lock、buf_hdr_lock 等）相关的，和并发有关，需要仔细查看，本文讲解缓冲区，不再多述并发相关内容。

4.1.4.3 内外存地址是如何映射的

1 缓存对应的内外存的映射关系：

```
typedef struct buftag
{
    RelFileNode rnode;          /* physical relation identifier */ // 数据库对象的位置标识，具体说明见下
    ForkNumber  forkNum;   // 可以查看“relpathbackend”函数，理解其所起的地址映射作用
    BlockNumber blockNum;       /* blknum relative to begin of reln */
} BufferTag;
```

2 数据库、表空间、和对象（主要是表对象）的对应关系，构成了内存和数据库对象的映射。

```
typedef struct RelFileNode
{
    Oid         spcNode;    /* tablespace */ //表空间，本身是一个物理存储的概念，每个表空间的 id 与其具体的数据文件对应，这样使得缓存块可以和数据文件对应
    Oid         dbNode;         /* database */
    Oid         relNode;    /* relation */
```

} RelFileNode;
在这个结构中，注意，成员 relNode，很重要，是数据和数据库对象映射的重要链条。

GetNewRelFileNode 函数的如下代码：

```
/* Generate the OID */
if (pg_class)
    rnode.node.relNode = GetNewOid(pg_class);
else
    rnode.node.relNode = GetNewObjectId();
```

3 数据对象和数据的映射关系：pg_class.h

在 pg_class.h 文件中，有一列，信息如下：

```
int4        relpages;       /* # of blocks (not always up-to-date) */
```

标识了本行所表示的对象的具体位置（在数据文件中的具体位置）。

而 pg_class 系统表，存放了数据库对象（表、视图、序列等）的元信息，本表隶属于具体的数据库（即系统表是局部的，每个数据库都有一份；有的系统表是全局的，全系统只有一份如 pg_database）。

有了如上三种信息，即可以串联起数据库对象、内存（缓存）、外存间的映射关系，即可知晓内外存地址是如何映射的了。

从代码的角度，可以看到：RelationOpenSmgr->smgropen

4.1.4.4Buf 的主体结构

形式上是“char *”，本质上是内存开辟出的一块地址连续的区域。

关键问题是：数据库如何使用这块区域（缓冲区调度算法）。

4.1.4.5Buf 的置换调度策略管理相关数据结构

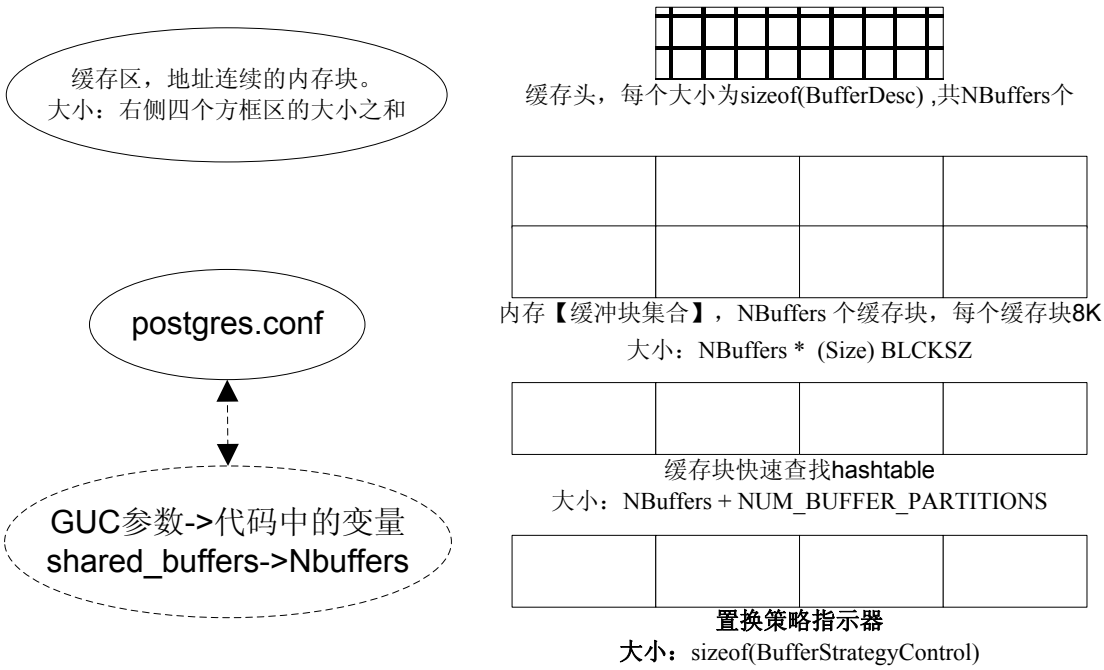
第一个相关结构：SharedBufHash

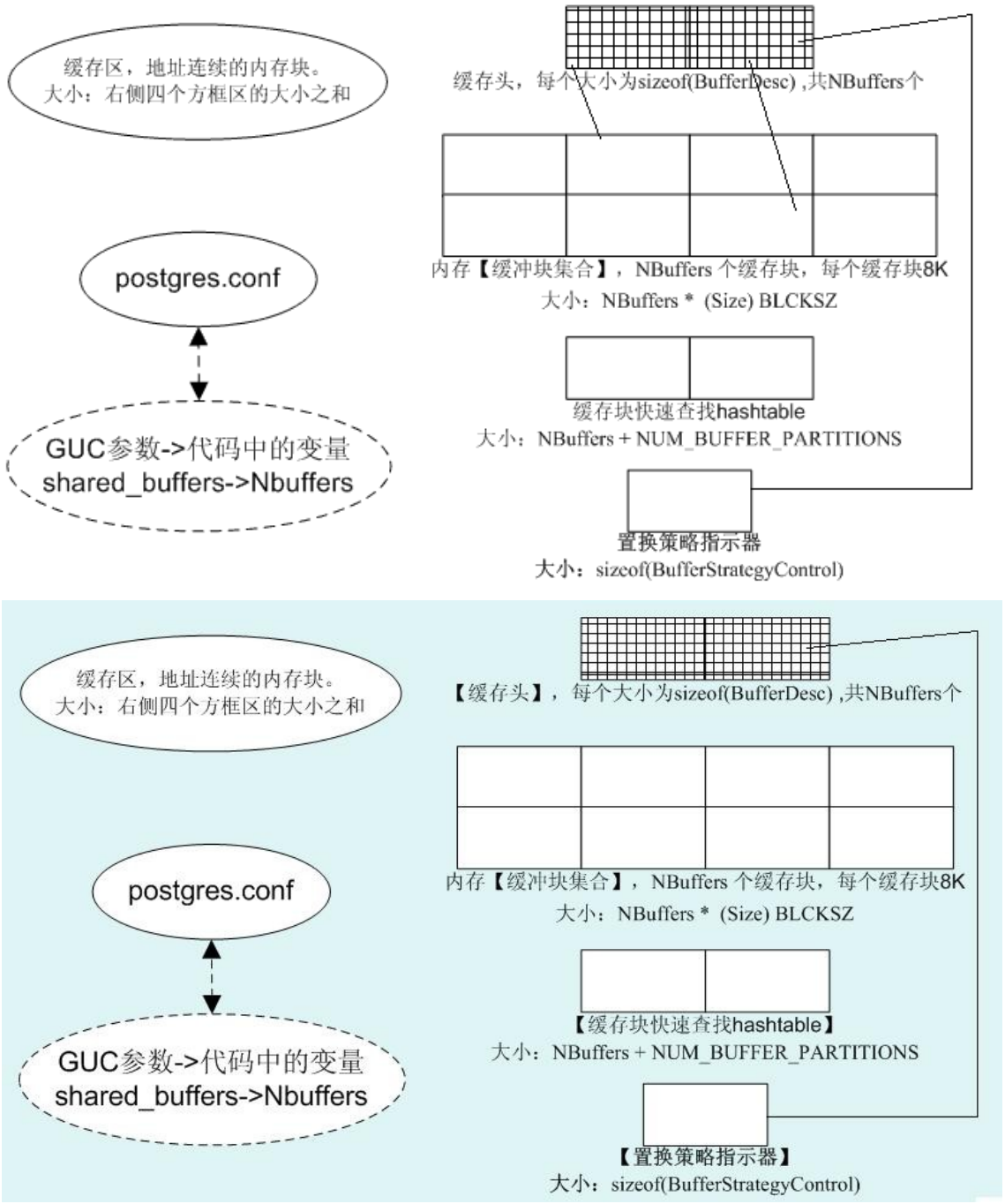
作用：哈希表，用于缓存块的快速查找。

第二个相关结构：BufferStrategyControl

```
/*
 * The shared freelist control information.
 */
typedef struct
{
```

```
    /* Clock sweep hand: index of next buffer to consider grabbing */
    int             nextVictimBuffer; // 指向缓存块（缓存块是地址连续的，所以把缓存块当作数组，用 int 型下标即可表示下一个要被置换的缓存块的位置）

    int             firstFreeBuffer;    /* Head of list of unused buffers */
    int             lastFreeBuffer; /* Tail of list of unused buffers */

    /*
     * NOTE: lastFreeBuffer is undefined when firstFreeBuffer is -1 (that is,
     * when the list is empty)
     */

    /*
     * Statistics.  These counters should be wide enough that they can't
     * overflow during a single bgwriter cycle.
     */
    uint32          completePasses; /* Complete cycles of the clock sweep */
    uint32          numBufferAllocs;    /* Buffers allocated since last reset */
} BufferStrategyControl;
```





说明：

1. 上图，左侧，是解释区，第一椭圆说明了整体缓存的情况；第二个椭圆表示 PostgreSQL 数据库初始化后，在 data（数据目录）中初始化出的配置文件，此文件存放数据启动、运行时需要的各种参数，其中，有关缓存的参数是“shared_buffers”，表示缓冲区大小，实际大小的计算方式，参见上图；“NBuffers”是代码中和缓冲区相关的变量，其值源于 GUC 参数“shared_buffers”。
2. 上图，右侧，是缓冲区。由四块地址连续的区域组成。分别是：缓存头、缓存块集合、缓存块快速查找 hashtable、置换策略指示器。
3. 缓冲区的第一个部分：【缓存头】，与【缓存块集合】一一对应，每一个缓存块，都有自己的一份使用情况信息，保存在【缓存头中】。

4. 缓冲区的第二个部分：【缓存块集合】，与外存的数据块对应，是 n:m 的关系。当缓存块还有空闲的时候，则从空闲链表中取缓存块共上层（数据访问层）使用，否则，使用“clock”算法，淘汰一个页面（页面大小对应一个块，8k），然后给这个页面置相应值（如可能读入新数据到缓存块）并他的元信息（缓存头中对应的使用情况信息，如是否加锁等），最后返回这个块
5. 缓冲区的第三个部分：【缓存块快速查找 hashtable】，把所有缓存块注册到一个 hash 表中，便于快速查找缓存块
6. 缓冲区的第四个部分：【置换策略指示器】，与缓存块的页面淘汰算法直接相关，指示了哪个缓存块可以被淘汰（注意图中从【置换策略指示器】发出的线的指向，对应的变量是 nextVictimBuffer，其用法“nextVictimBuffer++”,表明缓存块是从前到后逐个被淘汰的，这就是简单的“clock”算法，但并非说：缓存块是依次被淘汰，而是从前到后，找到没有被使用的、可被淘汰的---如脏页、引用计数为零的---进行淘汰，当 nextVictimBuffer 达到最大值，则变为零，从前面再重新开始淘汰）。

4.1.5 Buf 置换管理算法

Clock sweep 算法总结如下（代码参加 StrategyGetBuffer 函数）：

1. PG 维护一个“freelist”，将空闲的块，置于其上，当有空闲块时，直接从空闲块中取；当有块不再使用，可以归入到空闲块中。
2. 当空闲块已经用完，则执行淘汰策略，如下：
3. 缓冲块使用频率较高，不将其替换，只是将它的使用计数减一
4. 如果缓冲块的引用计数是 0，使用计数也是 0，就是说当前使用这个缓冲块，而且这个缓冲块最近无人使用，那么就可以重新使用了，这时，淘汰这个块上写的信息（脏页，刷出信息到外存），然后供重新使用；如果应用计数为零但使用计数不为零，则使用技术减一。
5. 如果已经遍历了所有的缓冲块，发现所有缓冲块都被别人占用，则报错，因为无法找到可用的缓冲区块了

PostgreSQL 对于缓冲区管理算法有如下描述（src/backend/storage/buffer/README）：

```
Normal Buffer Replacement Strategy
----------------------------------
There is a "free list" of buffers that are prime candidates for replacement.
In particular, buffers that are completely free (contain no valid page) are
always in this list.    We could also throw buffers into this list if we
consider their pages unlikely to be needed soon; however, the current
algorithm never does that.    The list is singly-linked using fields in the
buffer headers; we maintain head and tail pointers in global variables.
(Note: although the list links are in the buffer headers, they are
```

considered to be protected by the BufFreelistLock, not the buffer-header spinlocks.) **To choose a victim buffer to recycle when there are no free buffers available, we use a simple clock-sweep algorithm, which avoids the need to take system-wide locks during common operations.**

这段话表明，缓冲区置换调度策略，首先是 buf 维护一个空闲链表，如果空闲列表有空闲块，则优先使用空闲块；否则，则从缓冲区的数据库缓存块区根据置换策略淘汰缓存块，刷出脏数据，然后把这个以刷出数据的缓存块分配出去。

而置换，则是简单的（蓝色粗体字）“clock-sweep”（时钟置换）算法。

另外，除了“clock-sweep”算法，PostgreSQL 还在其他场景下使用了其他算法，如下（非本文主要讨论之处，不在详述）：

Buffer Ring Replacement Strategy

When running a query that needs to access a large number of pages just once, such as VACUUM or a large sequential scan, a different strategy is used.

4.1.6 并行计算对 buf 置换的影响

并行计算对 buf 置换的影响，是指在多用户状态下，对 buf 中的缓存块同时读写时缓冲区的调度、处理、需要考虑的问题等等。

假设，A、B 是三名仓库管理员，各管着 100 个仓库，每个仓库装入、取出哪个货主的货物，由管理员负责管理分配

A 管理员，当有货主想取货物时，A 都带着货主去每个仓库辨认一下货物是否是货主的，如是，则给货主支取；当有货主想存货物时，A 都去每个仓库看一下，找到一个空的仓库，把货物存放进去；当货主想存货物而没有空仓库时，上级曾经规定，把最长不来取货的仓库腾空给新货主使用，所以，A 管理员有个清理仓库的职责，但如果货物有价值，则先保存（到另外地方）货物使得仓库可被再用。

B 管理员，是个聪明的人，他不想在有货主提取或存入货物时老跑每个仓库查看具体情况，而是把每个仓库、货主的情况记载下来（元信息），每次货主存取货物，都先查记载，这样便于管理。

C 管理员，管理着高级仓库，每个仓库存储的货物，可以为多个货主提供存取服务（即每个仓库中的货物，可以被一个货主独占，也可以被多个或者共同查看—读、独自取—写）。

这样的假设，表明了对于仓库管理的需求和实际管理方式， B、C 结合的管理，是最好的选择。

从代码的角度，在“4.1.4.2 Buf 的元信息结构”节中，给出的 BufferDesc 结构的内容：

BufferDesc 结构的	加减锁	锁模式	说明

成员	操作函数/宏		
buf_hdr_lock	LockBufHdr UnlockBufHdr	PG 的自旋锁（SpinLock）	1. 对自己（BufferDesc）进行保护 2. 当有 refcount、usage_count 等需要被改写时，加 buf_hdr_lock 锁
io_in_progress_lock	LWLockAcquire LWLockRelease	LW_SHARED LW_EXCLUSIVE	1. 为**本层（数据缓冲区）**与**下层（数据存储层）**交换数据提供防止并发错误的机制
content_lock	LockBuffer UnlockBuffers	BUFFER_LOCK_SHARE BUFFER_LOCK_EXCLUSIVE	1. 对缓冲区进行保护 2. 当**上层（数据访问层）**读写缓冲区可能发生冲突时，可以使得读读并发，读写并发时先读不影响写、写不影响读 3. 为上层访问缓冲区提供防止并发错误的机制

BufferDesc 结构的其他成员	数据类型	作用	说明
tag	BufferTag	外存的物理块的地址，是联系外存（物理地址）和内容（buf）的关键部分	
flags	BufFlags	描述 buf 处于的状态 Buf 从空闲到被使用，期间，可以被修改(脏页,BM_DIRTY)，脏页需要被刷出才可重用； Buf 也可以出于正在被读入或刷出状态，这时，会特别标识为 BM_IO_IN_PROGRESS； 当 buf 出现错误时，会报告错误。	可以有下述状态： BM_DIRTY BM_VALID BM_TAG_VALID BM_IO_IN_PROGRESS BM_IO_ERROR BM_JUST_DIRTIED BM_PIN_COUNT_WAITER BM_CHECKPOINT_NEEDED
usage_count	uint16	使用计数器。被使用多少次，都自增 1，直到涨到最大（用 BM_MAX_USAGE_COUNT 表示最大的限制数）	
refcount	unsigned	Pin 计数器（引用计数器）。被 backend（带有用户连接的进	

		程）pin 住多少次。	
buf_id	int	指向 buf 缓冲区的“指针”（buf 缓冲区使用数组，所以用 int 做下标暂称之为指针）	
freeNext	int	执行空闲缓冲区的“指针”（buf 缓冲区使用数组，所以用 int 做下标暂称之为指针）	

特别注意：

Buf，是内存操作，是内外存交流的汇集点，起着承上启下的作用，但其本身位于内存中，故很多操作，和内存紧密相关，需要特别的注意。

很多细节，需要阅读代码+仔细揣摩、才能明白体会深刻。很多与其他模块相关联的点，没有提及，但可能也影响着 buf 的管理。

4.2 PostgreSQL 日志缓冲区管理

前述介绍的 buf，是一种双向的缓存，即“向外存发出读请求→读入数据到缓冲区”和“缓冲区中的数据被更新→写出到外存”。

在 PostgreSQL 系统中，还存在一种单向缓存，是 REDO 日志的缓存，此种缓存，是系统启动时读日志文件建立的，在运行过程中，缓存数据库引擎操作产生的 RODO 日志数据，然后按照一种规则，不断从内存刷出数据到外存，方向是从内到外，所以是单向缓存。

4.2.1 日志缓存相关代码

日志缓存的初始化

调用 XLOGShmemInit 函数完成日志缓存区的初始化工作（主要是**在共享内存中分配一块区域-- XLogCtl，以当作日志缓存使用**）。

主要代码

主要函数	作用	调用关系
XLogInsert	把内存中生成的日志信息插入到日志缓存中	XLogInsert[①] WriteZeroPageXlogRec WriteTruncateXlogRec

[①] 通过调用关系，可以看到，数据库中有什么样的操作需要写日志。几乎所有操作，只要是需要被恢复的（和 ACID 特性紧密相关），都需要构造好要恢复的数据，然后调用 XLogInsert 把信息保存在日志中

		createdb movedb movedb remove_dbtablespaces ginInsertValue writeListPage ginHeapTupleFastInsert shiftList createPostingTree ginbuild ginUpdateStats xlogVacuumPage ginDeletePage gistbuild gistplacetopage gistplacetopage gistnewroot gistbulkdelete gistContinueInsert gistxlogInsertCompletion heap_insert heap_delete heap_lock_tuple heap_inplace_update log_heap_cleanup_info log_heap_clean log_heap_freeze log_heap_update log_newpage CreateMultiXactId WriteMZeroPageXlogRec _bt_insertonpg _bt_split _bt_newroot _bt_getroot _bt_log_reuse_page _bt_delitems_vacuum _bt_delitems_delete _bt_pagedel write_relmap_file

		DefineSequence AlterSequenceInternal nextval_internal do_setval LogCurrentRunningXacts LogAccessExclusiveLocks RelationCreateStorage RelationTruncate CreateTableSpace DropTableSpace EndPrepare RecordTransactionCommitPrepared RecordTransactionAbortPrepared AssignTransactionId RecordTransactionCommit RecordTransactionAbort CreateCheckPoint XLogPutNextOid RequestXLogSwitch XLogReportParameters pg_stop_backup
XLogWrite	调用 write 函数（）把日志缓存的内容刷出到外存。调用 XLogFileClose 和 XLogFileOpen 进行文件切换（日志文件写满了，关掉当前日志文件，打开下一个日志文件写入日志信息）	XLogWrite XLogInsert AdvanceXLInsertBuffer XLogFlush XLogBackgroundFlush
XLogFlush	把日志缓存的内容刷出到外存（调用 XLogWrite）	XLogFlush FlushBuffer WriteTruncateXlogRec write_relmap_file SlruPhysicalWritePage RelationTruncate EndPrepare RecordTransactionCommitPrepared RecordTransactionAbortPrepared RecordTransactionCommit xact_redo_commit

		CreateCheckPoint
AdvanceXLInsertBuffer	写如日志时，空余空间不够，则预先分配一块 buf（方式是：先调用 XLogWrite 刷出一部分日志，然后把日志信息放入日志缓存）	AdvanceXLInsertBuffer XLogInsert（3 次调用） CreateCheckPoint

4.2.2 日志缓存的管理方式

日志缓存的管理，主要依赖于如下结构：

```
typedef struct XLogCtlData
{
    /* Protected by WALInsertLock: */
    XLogCtlInsert Insert;       //标识插入位置

    /* Protected by info_lck: */
    XLogwrtRqst LogwrtRqst;     //标识刷出请求的位置，请求未必已经被执行
    XLogwrtResult LogwrtResult;     //标识刷出结果的位置，已经刷出日志的位置
    uint32          ckptXidEpoch;     /* nextXID & epoch of latest checkpoint */
    TransactionId ckptXid;
    XLogRecPtr    asyncXactLSN; /* LSN of newest async commit/abort */
    uint32          lastRemovedLog; /* latest removed/recycled XLOG segment */
    uint32          lastRemovedSeg;

    /* Protected by WALWriteLock: */
    XLogCtlWrite Write;

    /*
     * These values do not change after startup, although the pointed-to pages
     * and xlblocks values certainly do.    Permission to read/write the pages
     * and xlblocks values depends on WALInsertLock and WALWriteLock.
     */
    char      *pages;                 /* buffers for unwritten XLOG pages */      //要写出的日志信息
```

//注意，日志信息，大小不同，指针指向不同的块，实际上，放的主要是数据缓存块的信息，这样，恢复时，不是以记录为单位恢复，而是以数据库运行态中的实际的“数据块”（数据缓存中的以块为单位的信息）做恢复。不好之处是，可能使得日志文件中的信息量很大。基于逻辑信息（SQL 语句）和基于物理信息（数据块）做恢复，是个争论点。教科书上，通常讲到 REDO 日志的时候，只会使用“前映像”和“后映像”指代恢复的单位，不明确说出

恢复的单位，这是讲原理和工程实现之间的差别所在。

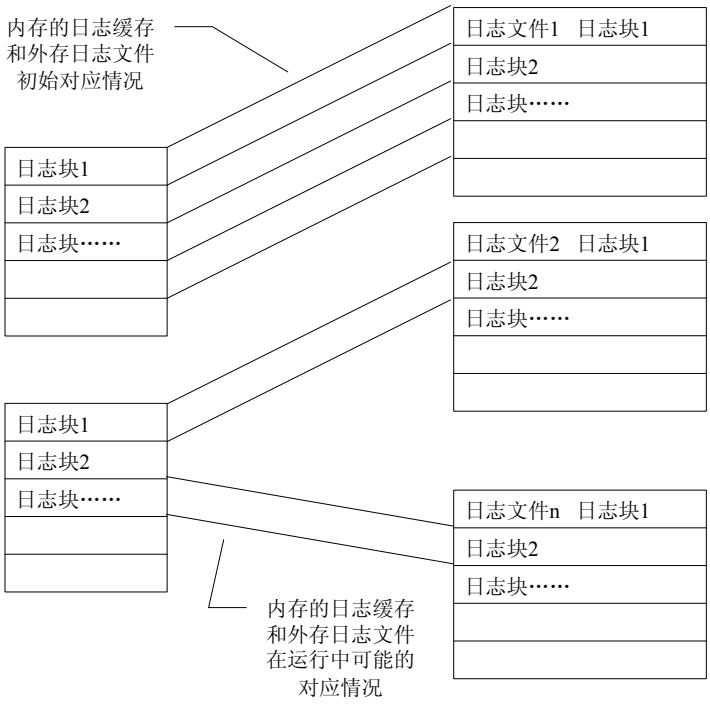
```
XLogRecPtr *xlblocks;        /* 1st byte ptr-s + XLOG_BLCKSZ */
int             XLogCacheBlck;    /* highest allocated xlog buffer index */
TimeLineID    ThisTimeLineID;
TimeLineID    RecoveryTargetTLI;

/*
 * archiveCleanupCommand is read from recovery.conf but needs to be in
 * shared memory so that the bgwriter process can access it.
 */
char        archiveCleanupCommand[MAXPGPATH];

/*
 * SharedRecoveryInProgress indicates if we're still in crash or archive
 * recovery.    Protected by info_lck.
 */
bool        SharedRecoveryInProgress;

/*
 * recoveryWakeupLatch is used to wake up the startup process to
 * continue WAL replay, if it is waiting for WAL to arrive or failover
 * trigger file to appear.
 */
Latch           recoveryWakeupLatch;

/*
 * During recovery, we keep a copy of the latest checkpoint record here.
 * Used by the background writer when it wants to create a restartpoint.
 *
 * Protected by info_lck.
 */
XLogRecPtr    lastCheckPointRecPtr;
CheckPoint     lastCheckPoint;

/* end+1 of the last record replayed (or being replayed) */
XLogRecPtr    replayEndRecPtr;
/* end+1 of the last record replayed */
XLogRecPtr    recoveryLastRecPtr;
/* timestamp of last COMMIT/ABORT record replayed (or being replayed) */
TimestampTz recoveryLastXTime;
```

```
    slock_t     info_lck;    /* locks shared variables shown above */
} XLogCtlData;
```

日志缓存管理的方式，主要是如下算法（方法）构成：

日志缓存，由 8k 大小的共享缓存构成（8k 不准确，可以查看 XLOGShmemSize 理解更为精确）。日志缓存，有个“元信息”块，就像数据缓冲区的“BufferDesc”一样，有个“XLogCtlData”描述了日志缓冲区的使用情况。

日志缓存，是个固定大小的区域（如下图左侧，2 个方块，代表的是同一个日志缓冲区），而日志文件，是个线性连续不断的文件（如下图右侧，使用了 3 个方块表示日志文件，但日志文件根据日志号连续标识，可以看作是一个线性增长的文件序列），如何以固定区域的一块内存对应外存的线性文件，是我们需要掌握的问题。



对于日志缓存而言（如上图左侧），其大小是固定的，内存总是要外外存小，这样，以有限的内存映射无限的外存文件，则必须循环使用内存。其用法，有点像我们学习过的“循环数组”。

1. 内存日志缓存，其大小固定
2. 内存日志缓存中存放的日至块，大小不固定。（图示的左侧的日志缓存中的小块，看起来均匀，但实际上，其中存放的日志信息的大小是不一样的）
3. 当内存日志缓存中有足够的空间（每次存放日志信息的时候，主要要比较空余空间大小），则可以把信息放到日志缓存中
4. 当日志缓存没有足够的空间，则需要写出（调用 XLogWrite）一些信息到日志文件，以便

重复利用日志（循环使用）

5. 当日志缓存写到尾部且空间不足时，除了要刷出一部分数据外，还要注意，从日子缓存头部刷出信息后要被重复利用
6. 日志缓存的信息是线性刷出的，这样日志文件中的信息必然是连续的（连续通过“LSN”这样的东西表示，可以通过 PageGetLSN 函数入手追踪 LSN，实则是日志的 ID 和文件中的偏移的组合标识，日志 ID 对应一个日志文件，偏移则标识了文件内的位置，这样就唯一标识了一个物理位置）
7. 日志缓存到日志文件，是单项的，只写不读，对 IO 的影响不大
8. 特别需要注意阅读代码的地方是：跨页写（调用 XLByteInPrevSeg 函数之处）、日志文件的切换（写满后怎么再写到下一个文件，XLogWrite 函数中如何调用 XLogFileClose 和 XLogFileInit 和 XLogFileOpen 函数）

4.2.3 双向缓存和单向缓存的比较

REDO 缓存和 REDO 日志文件，前者是固定大小的区域，后者是一个线性的文件序列，如何使得固定区域和线性文件对应，是 REDO 缓存要解决的问题。这点，笔者在上一节进行了描述。

对于数据库的数据缓存，其缓存和外存的数据文件的对应关系，是少对多、一个对一个的关系；而日志缓存和日志文件的对应关系，是少对多、单个对线性序列的关系。要仔细体会这 2 者的不同，还需要很好的学习源码。

4.3 PostgreSQL 数据缓存区改进

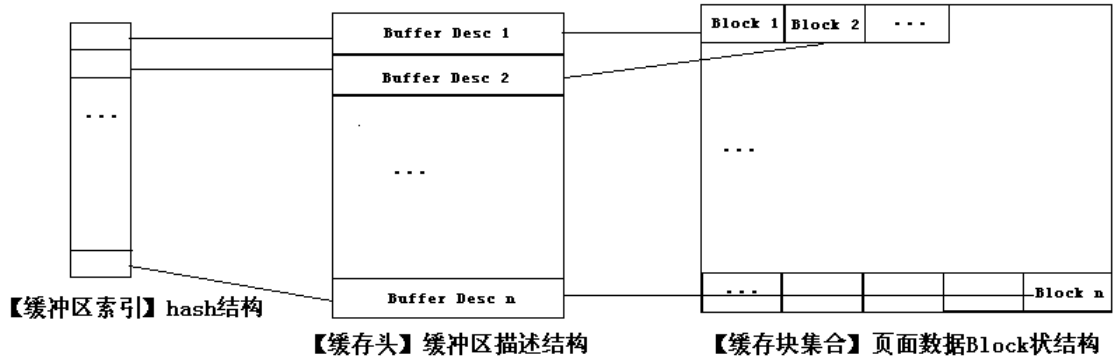
对于数据库的数据缓存区，PG 采取的是在一块连续的数据块对应的内存中，进行局部缓存块的淘汰。当这样的缓冲区被分配好后，则不可再发生动态调整。

PG 的分配内存的整体方式，是把所有需要内存的都统一求出大小，在系统初始启动过程中，一次从内存中申请出硕大的一块区域，然后在这个区域上，划分出不同的小区域，为不同的内存需求者使用。这样，严重制约了内存动态使用的需求。

有的系统对于数据缓冲区的管理，是采取动态的方式，在运行中，可以自动根据实际情况，自己进行调整，这对应的，应该是另一套内存的管理方式了（如使用链表、hash 等，都可以动态管理内存）。

4.3.1 从 buf 结构看改进---动态调整 buf 缓存大小

在“4.1.4 Buf 结构”节描述了 buf 的内部结构，下图，描述了 buf 的整体结构。这个是从 buf 管理的代码的整体角度描述的。

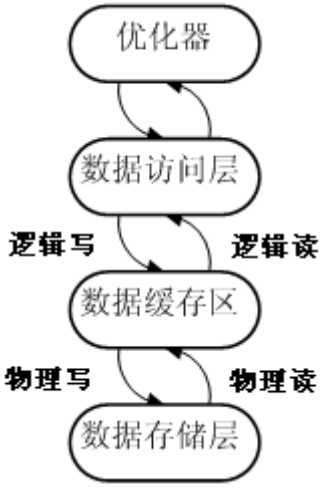


上图是 PG 的缓存结构图。

缓存块集合，是一个固定大小的内存区域，通过**缓存头（数组）的下标**，标识出数据页（Block）的位置。PG 通过这种方式，实现的是静态缓冲区。

如果改造缓存头，通过指针指向缓存块，则可以实现动态缓冲区。这样，数据库的缓冲区就可以根据系统的实际情况动态扩展了。

4.3.2 从 buf 与 IO 看改进---异步 IO



与 IO 相关的操作，如上图所示。

其中，分为两种方式的读写：逻辑 IO 和物理 IO。

逻辑 IO，是数据据访问层向缓存区读写数据的操作，请求数据，称为逻辑读；写入数据，称为逻辑写。

物理 IO，是数据缓冲区向外存读写数据的操作，请求数据，称为物理读；写入数据，称为物理写。

物理层的读写，是通过 smgr 接口进行的。调用序列如下：

ReadBuffer→ReadBuffer_common→smgrread→mdread;

FlushBuffer→smgrwrite→mdwrite

而阅读 md.c 中的代码，可以发现，物理读写操作，都是通过调用 C 的库函数 read 和 write 等实现的(注意 win32.h 中，read 和 write 实际是_read 和_write)。

而 read 和 write 都是同步的方式进行读写的，所以，物理 IO 有可能有改进的方式，改法是，在 smgr 层，实现**异步 IO**[①]操作。然后在数据缓冲区层改造 buf 的管理方式，使之能适应异步 IO（AIO）。比如，加入预读方式等。

Linux 的 AIO 在 2.5 版本的内核中首次出现，现在已经是 2.6 版本的产品内核的一个标准特性了。

如果数据存储层的物理 IO 改写为异步 IO，还需要封装一套屏蔽多操作系统的接口，供上层的数据缓冲区层调用，这里不再展开。

4.3.3 从 buf 淘汰方式看改进---freelist

在“4.1.4.5 Buf 置换管理算法”中，我们描述了 buf 调度算法。PG 首先是从 freelist 中取出空余的优先供缓存请求使用。如果 freelist 空，才去真正淘汰缓存块。

所以，一个思路是：尽可能增加 freelist 中的成员，使得淘汰减少，是否物理 IO 也可以减少呢？

先看 PG 对于 freelist 的操作方式：

1. 从结构上看，首先定义出一个指针，用来指向空余的 buf，见如下蓝色字体

```
typedef struct BufferAccessStrategyData
{
    BufferAccessStrategyType btype;
    int             ring_size;
    int             current;
    bool        current_was_in_ring;
    /*
     * Array of buffer numbers.  InvalidBuffer (that is, zero) indicates we
     * have not yet selected a buffer for this ring slot.  For allocation
     * simplicity this is palloc'd together with the fixed fields of the
     * struct.
     */
    Buffer          buffers[1];         /* VARIABLE SIZE ARRAY */
} BufferAccessStrategyData;
```

2. 从这个指针的调用关系看，共被：增加空余 buf 到链表操作（AddBufferToRing）和从链表取出空余 buf 操作构成（GetBufferFromRing）

```
BufferAccessStrategyData.buffers
    BufferAccessStrategyData（就是如上的数据结构）
    GetBufferFromRing
    AddBufferToRing
    StrategyRejectBuffer
```

① http://www.ibm.com/developerworks/cn/linux/l-async/ linux 下异步 IO
http://blog.csdn.net/bokee/article/details/5268894 windows 下异步 IO

3. 从函数的调用关系看，GetBufferFromRing 和 AddBufferToRing 函数，都是被 StrategyGetBuffer 函数调用。而当有缓存块被申请使用时，才会调用到 StrategyGetBuffer 函数（从 StrategyGetBuffer 函数的代码中可以看出，只有使用指定“淘汰策略”，才会从一个“Ring”上拿出空余的 buf 供上层使用）。这样，freelist 上的空余 buf 增加的机会并不多。这样，可能会影响着 buf 的有效利用。


```
/*
 * If given a strategy object, see whether it can select a buffer. We
 * assume strategy objects don't need the BufFreelistLock.
 */
if (strategy != NULL)
{
    buf = GetBufferFromRing(strategy);
    if (buf != NULL)
    {
        *lock_held = false;
        return buf;
    }
}
```

所以，我们可以知道，此处也有改进余地。

改法：增加一个线程，在适当的时机，遍历缓冲区，当发现空余的缓冲区存在时，即补充到 freelist 上，这样当需要申请缓冲区时，freelist 尽可能地多存在空闲缓存块。

什么时候让这个线程启动去干活呢？上一节，增加了异步 IO 操作（提出了预取功能）。当发出异步 IO 读写请求时，数据还没有到位，也许是启动这样线程的最佳时机。