第 8 章

缓冲区管理器

缓冲区管理器管理着共享内存和持久存储之间的数据传输，对于数据库管理系统的性能有着重要的影响。PostgreSQL 的缓冲区管理器十分高效。

本章介绍了 PostgreSQL 的缓冲区管理器。图 8.1 展示了缓冲区管理器存储和后端进程之间的关系，后续的章节分别介绍以下内容：

- 缓冲区管理器的结构
- 缓冲区管理器的锁
- 缓冲区管理器的工作原理
- 环形缓冲区
- 脏页刷盘

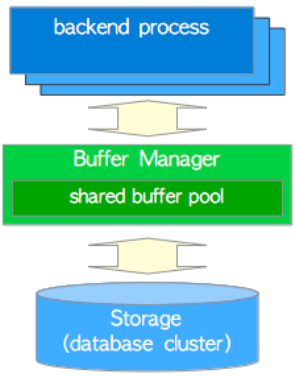

图 8.1　缓冲区管理器存储和后端进程之间的关系

8.1　概览

接下来介绍一些关键概念，以助于读者理解后续章节。

1．缓冲区管理器的结构

PostgreSQL 缓冲区管理器由缓冲表、缓冲区描述符和缓冲池组成，这几个组件将在接下来的小节中介绍。缓冲池层存储着数据文件页面，诸如表页与索引页，以及其相应的自由空间映射和可见性映射的页面。缓冲池是一个数组，数据的每个槽中存储数据文件的一页。缓冲池数组的序号索引称为 buffer_id。第 8.2、8.3 节描述了缓冲区管理器的内部细节。

2．缓冲区标签

PostgreSQL 中的每个数据文件页面都可以分配到唯一的标签，即缓冲区标签。当缓冲区管理器收到请求时，PostgreSQL 会用到目标页面的缓冲区标签。

其中，关系文件节点用于定位页面所属的关系，关系分支编号用于定位关系文件的具体分支文件，页面块号则在具体分支文件中指明相应页面的偏移量。一个关系可能有三种分支，分别是关系主体（main 分支，编号为 0）、空闲空间映射（fsm 分支，编号为 1）及可见性映射（vm 分支，编号为 2）。

缓冲区标签由三个值组成，分别是关系文件节点、关系分支编号和页面块号。第一个值分别代表了表空间、数据库和表的 oid；第二个值代表关系表的分支号；最后一个值代表页面号。那么该标签表示，在某个表空间（oid=16821）中，某个数据库（oid=16384)的某张表（oid=37721）的 0 号分支（0 代表关系表本体）的第 7 号页面。再比如，缓冲区标签{(16821, 16384, 37721), 1, 3}表示该表空闲空间映射文件的三号页面。关系本体 main 分支编号为 0，空闲空间映射 fsm 分支编号为 1。

```
/*
 * Buffer tag 标识了缓冲区中包含着哪一个磁盘块
 * 注意，BufferTag 中的数据必须足以在不参考 pg_class 或 pg_tablespace 中的数据项
 * 的前提下，能够直接确定该块需要写入的位置。不过可能出现这种情况，即刷写缓冲区的
 * 后端进程甚至都不认为自己能在那个时刻看见相应的关系（譬如，后段进程对应的事务
 * 开始时间早于创建该关系的事务）。无论如何，存储管理器都必须能应对这种情况
 *
 * 注意，如果结构中存在任何填充字节，INIT_BUFFERTAG 需要将所有字段抹为零，因为整个
 * 结构体被当成一个散列键来用
 */
typedef struct buftag
```

```
{
     RelFileNode rnode;          /* 关系的物理标识符 */
     ForkNumber    forkNum;        /* 关系的分支编号    */
     BlockNumber blockNum;       /* 相对于关系开始位置的块号 */
} BufferTag;

typedef struct RelFileNode
{
    Oid         spcNode;        /* 表空间 */
    Oid         dbNode;         /* 数据库 */
    Oid         relNode;        /* 关系 */
} RelFileNode;
```

3．后端进程如何读取数据页

本节描述了后端进程如何从缓冲区管理器中读取数据页，如图 8.2 所示。

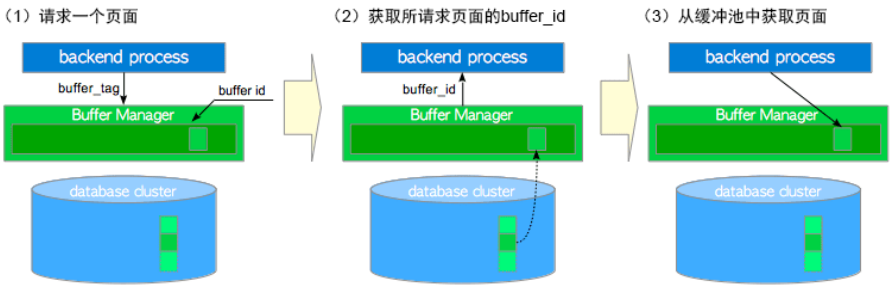

图 8.2　后端进程如何读取数据页

（1）当读取表或索引页时，后端进程向缓冲区管理器发送请求，请求中带有目标页面的 buffer_tag。

（2）缓冲区管理器会根据 buffer_tag 返回一个 buffer_id，即目标页面存储在数组中的槽位的序号。如果请求的页面没有存储在缓冲池中，那么缓冲区管理器会将页面从持久存储位置加载到其中一个缓冲池槽位中，然后再返回该槽位的 buffer_id。

（3）后端进程访问 buffer_id 对应的槽位（以读取所需的页面）。

当后端进程修改缓冲池中的页面时（例如向页面插入元组），这种尚未刷新到持久存储，但已被修改的页面被称为脏页。

第 8.4 节将描述缓冲区管理器的工作原理。

4．页面置换算法

当所有缓冲池槽位都被占用，且其中未包含所请求的页面时，缓冲区管理器必须在缓冲池中选择一个页面逐出，用于放置被请求的页面。在计算机科学领域中，选择页面的算法通常被称为页面置换算法，而所选择的页面被称为受害者页面。

自从计算机科学出现以来，针对页面置换算法的研究就一直在进行，先前已经提出很多置换算法。从 8.1 版本开始，PostgreSQL 使用时钟扫描算法，比起以前版本中使用的 LRU 算法，更为简单高效。

第 8.4.4 节将描述时钟扫描的细节。

5．脏页刷盘

脏页最终应该被刷入存储，但缓冲区管理器执行这个任务需要额外帮助。在 PostgreSQL 中，检查点进程和后台写入器这两个后台进程负责此任务。

第 8.6 节将描述检查点进程和后台写入器。

> **直接 I/O**
>
> PostgreSQL 并不支持直接 I/O，但有时会讨论它。如果你想了解更多详细信息，可以参考 https://lwn.net/Articles/580542/这篇文章，以及 pgsql-ML 中的讨论，详见 https://www.postgresql.org/message-id/529E267F.4050700@agliodbs.com。

8.2　缓冲区管理器的结构

PostgreSQL 缓冲区管理器由三层组成，即缓冲表层、缓冲区描述符层和缓冲池层，如图 8.3 所示。

- 缓冲表层是一个散列表，它存储着页面的 `buffer_tag` 与描述符的 `buffer_id` 之间的映射关系。
- 缓冲区描述符层是一个由缓冲区描述符组成的数组。每个描述符与缓冲池槽一一对应，并保存着相应槽的元数据。请注意，术语“缓冲区描述符层”只是在本章中为方便起见而使用的术语。
- 缓冲池层是一个数组。每个槽都存储一个数据文件页，数组槽的索引称为 `buffer_id`。

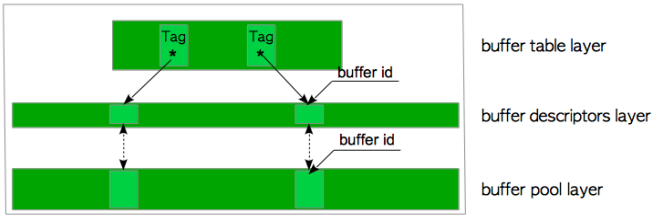

图 8.3　缓冲区管理器的三层结构

这些层将在以下内容中详细描述。

8.2.1　缓冲表

缓冲表可以在逻辑上分为三个部分，分别是散列函数、散列桶槽及数据项，如图 8.4 所示。

内置散列函数将 buffer_tag 映射到哈希桶槽。即使散列桶槽的数量比缓冲池槽的数量多，冲突也可能会发生。因此缓冲表采用了使用链表的分离链接方法来解决冲突。当数据项被映射至同一个桶槽时，该方法会将这些数据项保存在一个链表中，如图 8.4 所示。

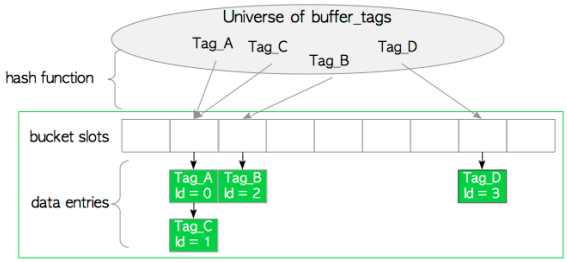

图 8.4　缓冲表

数据项包括两个值，即页面的 buffer_tag 和包含页面元数据的描述符的 buffer_id。例如，数据项 Tag_A,id=1 表示，在 buffer_id=1 对应的缓冲区描述符中，存储着页面 Tag_A 的元数据。

> **散列函数**
>
> **此处使用的散列函数由 calc_bucket()与 hash()组合而成。下面是用伪函数表示的形式。**

```
uint32 bucket_slot =
    calc_bucket(unsigned hash(BufferTag buffer_tag), uint32 bucket_size)
```

暂时还没有对诸如查找、插入、删除数据项的基本操作进行解释。这些常见的操作将在后续小节详细描述。

8.2.2 缓冲区描述符

本节将描述缓冲区描述符的结构，第 8.2.3 节将描述缓冲区描述符层。

缓冲区描述符保存着页面的元数据，这些与缓冲区描述符相对应的页面保存在缓冲池槽中。缓冲区描述符的结构由 BufferDesc 结构定义。这个结构有很多字段，主要字段如下所示：

```
/* src/include/storage/buf_internals.h  (before 9.6) */

/* 缓冲区描述符的标记位定义(since 9.6)
 * 注意，TAG_VALID 实际上意味着缓冲区散列表中有一条与本 tag 关联的项目
 */
#define BM_DIRTY                (1 << 0)    /* 数据需要写入 */
#define BM_VALID                (1 << 1)    /* 数据有效 */
#define BM_TAG_VALID            (1 << 2)    /* 已经分配标签 */
#define BM_IO_IN_PROGRESS       (1 << 3)    /* 读写进行中 */
#define BM_IO_ERROR             (1 << 4)    /* 先前的 I/O 失败 */
#define BM_JUST_DIRTIED         (1 << 5)    /* 写之前已经脏了 */
#define BM_PIN_COUNT_WAITER     (1 << 6)    /* 有人等着钉页面 */
#define BM_CHECKPOINT_NEEDED    (1 << 7)    /* 必须在检查点时写入 */
#define BM_PERMANENT            (1 << 8)    /* 永久缓冲(不是 unlogged) */

/* BufferDesc -- 单个共享缓冲区的共享描述符/共享状态
 *
 * 注意，读写 tag、 flags、 usage_count、refcount、wait_backend_pid 等字段时必须持有
 * buf_hdr_lock 锁。buf_id 字段在初始化之后再也不会改变，所以不需要锁。freeNext 是通过
 * buffer_strategy_lock 来保护的，而不是 buf_hdr_lock。LWLocks 字段可以自己管好自己
 * 注意，buf_hdr_lock *不是* 用来控制对缓冲区内数据的访问的
 *
 * 有一个例外是，如果我们钉住了缓冲区，它的标签除了我们自己之外不会被偷偷修改
 * 所以我们无须锁定自旋锁就可以检视该标签。此外，一次性的标记读取也无须锁定自旋锁，
 * 当我们期待测试标记位不会改变时，这种做法很常见
 *
 * 如果另一个后端钉住了该缓冲区，我们就无法从磁盘页面上物理移除项目。因此后端需要等待
 * 所有其他的钉被移除。移除时它会得到通知，这是通过将它的 PID 存到 wait_backend_pid，
 * 并设置 BM_PIN_COUNT_WAITER 标记位而实现的。就目前而言，每个缓冲区只能有一个等待者
 *
 * 对于本地缓冲区，我们也使用同样的首部，不过锁字段没用，一些标记位也没用
```

```
 */
typedef struct sbufdesc
{
   BufferTag    tag;                    /* 存储在缓冲区中页面的标识 */
   BufFlags     flags;                  /* 标记位 */
   uint16       usage_count;            /* 时钟扫描要用到的引用计数 */
   unsigned     refcount;               /* 在本缓冲区上持有 PIN 的后端进程数 */
   int          wait_backend_pid;       /* 等着 PIN 本缓冲区的后端进程 PID */
   slock_t      buf_hdr_lock;           /* 用于保护上述字段的锁 */
   int          buf_id;                 /* 缓冲的索引编号 (从 0 开始) */
   int          freeNext;               /* 空闲链表中的链接 */

   LWLockId     io_in_progress_lock; /* 等待 I/O 完成的锁 */
   LWLockId     content_lock;           /* 访问缓冲区内容的锁 */
} BufferDesc;
```

- tag 保存着目标页面的 buffer_tag，该页面存储在相应的缓冲池槽中，缓冲区标签的定义将在第 8.1 节给出。
- buffer_id 标识了缓冲区描述符，亦相当于对应缓冲池槽的 buffer_id。
- refcount 保存当前访问相应页面的 PostgreSQL 进程数，也被称为钉数。当 PostgreSQL 进程访问相应页面时，其引用计数必须自增 1（refcount ++）。访问结束后其引用计数必须减 1（refcount--）。当 refcount 为零，即页面当前并未被访问时，页面将取钉，否则它会被钉住。
- usage_count 保存着相应页面加载至相应缓冲池槽后的访问次数。usage_count 会在页面置换算法中被用到，见第 8.4.4 节。
- context_lock 和 io_in_progress_lock 是轻量级锁，用于控制对相关页面的访问。第 8.3.2 节将介绍这些字段。
- flags 用于保存相应页面的状态，主要状态如下：

  1. 脏位指明相应页面是否为脏页。
  2. 有效位指明相应页面是否可以被读写（有效）。例如，如果该位被设置为"valid"，那就意味着对应的缓冲池槽中存储着一个页面，而该描述符中保存着该页面的元数据，因而可以对该页面进行读写。反之如果有效位被设置为"invalid"，那就意味着该描述符中并没有保存任何元数据，即对应的页面无法读写，缓冲区管理器可

能正在将该页面换出。

3. IO 进行标记位指明缓冲区管理器是否正在从存储中读/写相应页面。换句话说，该位指示是否有一个进程正持有此描述符上的 io_in_pregr ess_lock。

- freeNext 是一个指针，指向下一个描述符，并以此构成一个空闲列表（freelist），具体细节将在第 8.2.3 节中介绍。

结构 BufferDesc 定义于 src/include/storage/buf_internals.h 中。

为了简化后续章节的描述，这里定义三种描述符状态。

- 空：当相应的缓冲池槽不存储页面时，即 refcount 与 usage_count 都是 0，该描述符的状态为空。
- 钉住：当相应缓冲池槽中存储着页面，且有 PostgreSQL 进程正在访问的相应页面时，即 refcount 和 usage_count 都大于等于 1，该缓冲区描述符的状态为钉住。
- 未钉住：当相应的缓冲池槽存储页面，但没有 PostgreSQL 进程正在访问相应页面时，即 usage_count 大于或等于 1，但 refcount 为 0，该缓冲区描述符的状态为未钉住。

每个描述符都处于上述状态之一。描述符的状态会根据特定条件而改变，这将在第 8.2.3 节中介绍。

缓冲区描述符的状态用方框表示。

- □空
- ■钉住
- ■未钉住

此外，脏页面会带有“X”的标记。例如一个未钉住的脏描述符用☒表示。

8.2.3 缓冲区描述符层

缓冲区描述符的集合构成了一个数组，本书称该数组为缓冲区描述符层。

当 PostgreSQL 服务器启动时，所有缓冲区描述符的状态都为空。在 PostgreSQL 中，这些描述符构成了一个名为 freelist 的链表，缓冲区管理器初始状态如图 8.5 所示。

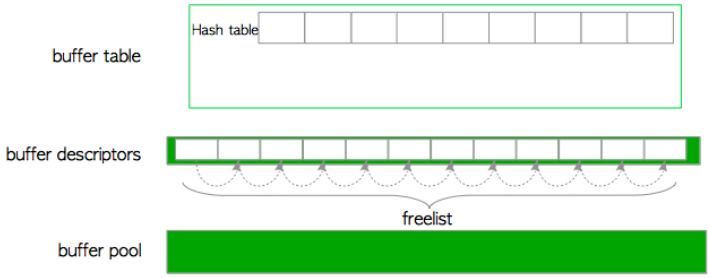

图 8.5　缓冲区管理器初始状态

> **注意**，PostgreSQL 中的 freelist 完全不同于 Oracle 中 freelist 的概念。PostgreSQL 的 freelist 只是空缓冲区描述符的链表。PostgreSQL 中与 Oracle 中的 freelist 相对应的对象是空闲空间映射（FSM），详见第 5.3.4 节。

图 8.6 展示了第一个页面是如何加载的。

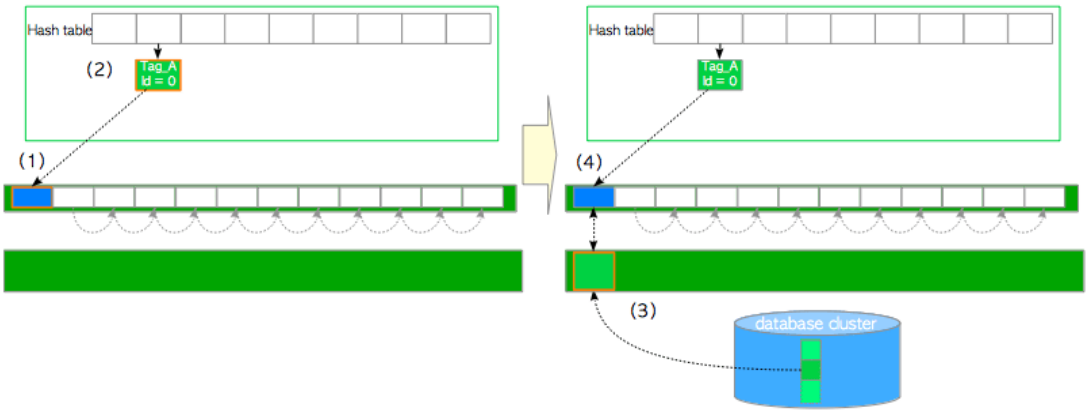

图 8.6　加载第一页

（1）从 `freelist` 的头部取一个空描述符，并将其钉住，即将 `refcount` 和 `usage_count` 增加 1。

（2）在缓冲表中插入新项，该缓冲表项保存了页面 `buffer_tag` 与所获描述符 `buffer_id` 之间的关系。

（3）将新页面从存储器加载至相应的缓冲池槽中。

（4）将新页面的元数据保存至所获取的描述符中。

第二页及后续页面都以类似方式加载，其他细节将在第 8.4.2 节中介绍。

从 freelist 中摘出的描述符始终保存着页面的元数据。换言之，仍然在使用的非空描述符不会返还到 freelist 中。但当下列任一情况出现时，描述符状态将变为“空”，并被重新插入至 freelist 中。

1．相关表或索引已被删除。

2．相关数据库已被删除。

3．相关表或索引已经被 VACUUM FULL 命令清理。

> **为什么使用 freelist 来维护空描述符**
>
> 保留 freelist 的原因是为了能立即获取一个描述符。这是内存动态分配的常规做法，详情参阅 https://en.wikipedia.org/wiki/Free_list 中的说明。

缓冲区描述符层包含着一个 32 位无符号整型变量 nextVictimBuffer。此变量将用于第 8.4.4 节中的页面置换算法。

8.2.4　缓冲池

缓冲池只是一个用于存储关系数据文件（例如表或索引）页面的简单数组。缓冲池数组的序号索引也就是 buffer_id。

缓冲池槽的大小为 8KB，等于页面大小，因而每个槽都能存储整个页面。

8.3　缓冲区管理器锁

缓冲区管理器会出于不同的目的使用各式各样的锁，本节将介绍理解后续部分所必备的一些锁。

> 注意，本节描述的锁，指的是缓冲区管理器同步机制的一部分。它们与 SQL 语句和 SQL 操作中的锁没有任何关系。

8.3.1　缓冲表锁

BufMappingLock 保护整个缓冲表的数据完整性。它是一种轻量级的锁，有共享模式与独占模式。在缓冲表中查询条目时，后端进程会持有共享的 BufMappingLock。插入或删除条目时，后端进程会持有独占的 BufMappingLock。

BufMappingLock 会被分为多个分区，以减少缓冲表中的争用（默认为 128 个分区）。每个 BufMappingLock 分区都保护着一部分相应的散列桶槽。

图 8.7 给出了一个 BufMappingLock 分区的典型示例。两个后端进程可以同时持有各自分区的 BufMappingLock 独占锁，以插入新的数据项。如果 BufMappingLock 是系统级的锁，那么其中一个进程就需要等待另一个进程完成处理。

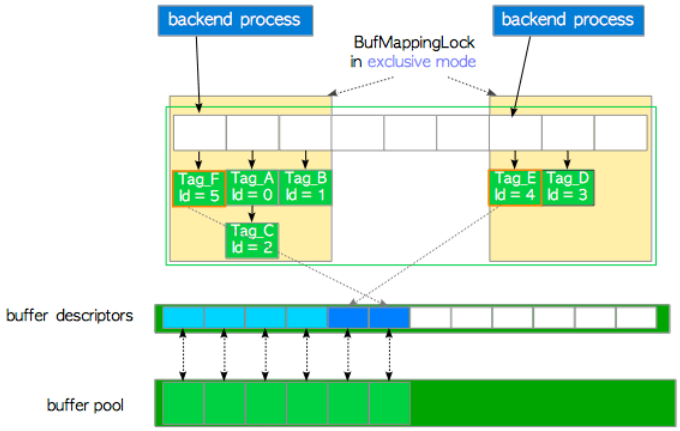

图 8.7　两个进程同时获取相应分区的 BufMappingLock 独占锁，以插入新数据项

缓冲表也需要许多其他锁。例如，在缓冲表内部会使用自旋锁（spin lock）来删除数据项。不过本章不需要这些锁的其他相关知识，因此这里省略了对其他锁的介绍。

> **在 9.4 版本之前，BufMappingLock 在默认情况下被分为 16 个独立的锁。**

8.3.2　缓冲区描述符相关的锁

每个缓冲区描述符都会用到内容锁（content_lock）与 IO 进行锁（io_in_progress_lock）这两个轻量级锁，以控制对相应缓冲池槽页面的访问。当检查或更改描述符本身字段的值时，就会用到自旋锁。

8.3.2.1　内容锁

内容锁（content_lock）是一个典型的强制限制访问的锁，它有共享与独占两种模式。

当读取页面时，后端进程以共享模式获取页面相应缓冲区描述符中的 content_lock。

执行下列操作之一时，则会获取独占模式的 content_lock。

- 将行（即元组）插入页面，或更改页面中元组的 t_xmin/t_xmax 字段时（t_xmin 和 t_xmax 在第 5.2 节中有过介绍，简单地说，这些字段会在相关元组被删除或更新行时发生更改）。
- 物理移除元组，或压紧页面上的空闲空间（由清理过程和 HOT 执行，分别在第 6 章和第 7 章中有过介绍）。
- 冻结页面中的元组（冻结过程在第 5.10 节与第 6.3 节中有过介绍）。

官方 README 文件包含更多的细节。

8.3.2.2　IO 进行锁

IO 进行锁（io_in_progress_lock）用于等待缓冲区上的 I/O 完成。当 PostgreSQL 进程加载/写入页面数据时，该进程在访问页面期间，持有对应描述符上独占的 io_in_progres_lock。

8.3.2.3　自旋锁

当检查或更改标记字段与其他字段时，例如 refcount 和 usage_count，会用到自旋锁。下面是两个使用自旋锁的具体例子。

1．钉住缓冲区描述符。

（1）获取缓冲区描述符上的自旋锁。

（2）将其 refcount 和 usage_count 的值增加 1。

（3）释放自旋锁。

```
LockBufHdr(bufferdesc);     /* 获取自旋锁 */
bufferdesc->refcont++;
bufferdesc->usage_count++;
UnlockBufHdr(bufferdesc);  /* 释放自旋锁 */
```

2．将脏位设置为"1"。

（1）获取缓冲区描述符上的自旋锁。

（2）使用位操作将脏位置位为"1"。

（3）释放自旋锁。

```
#define BM_DIRTY              (1 << 0)    /* 数据需要回写*/
#define BM_VALID              (1 << 1)    /* 数据有效 */
#define BM_TAG_VALID          (1 << 2)    /* 已经分配了 TAG */
#define BM_IO_IN_PROGRESS     (1 << 3)    /* 正在进行读写 */
#define BM_JUST_DIRTIED       (1 << 5)    /* 开始写之后数据刚被修改 */

LockBufHdr(bufferdesc);
bufferdesc->flags |= BM_DIRTY;
UnlockBufHdr(bufferdesc);
```

其他标记位也是通过同样的方式来设置的。

用原子操作替换缓冲区管理器的自旋锁

在 9.6 版本中，缓冲区管理器的自旋锁被替换为原子操作，可以参考 https://commitfest.postgresql.org/ 9/408/中提交日志的内容。如果想进一步了解详情，可以参考 http://www.postgresql.org/message-id/flat/2400449.GjM57CE0Yg@dinodell#2400449.GjM57CE0Yg@dinodell 中的讨论。

附 9.6 版本中缓冲区描述符的数据结构定义。

```
/* src/include/storage/buf_internals.h  (9.6 版本之后, 移除了一些字段) */

/* 缓冲区描述符的标记位定义(9.6 版本之后)
 * 注意，TAG_VALID 实际上意味着缓冲区散列表中有一条与本 tag 关联的项目
 */
#define BM_LOCKED                (1U << 22)    /* 缓冲区首部被锁定 */
#define BM_DIRTY                 (1U << 23)    /* 数据需要写入 */
#define BM_VALID                 (1U << 24)    /* 数据有效 */
#define BM_TAG_VALID             (1U << 25)    /* 标签有效，已经分配 */
#define BM_IO_IN_PROGRESS        (1U << 26)    /* 读写进行中 */
#define BM_IO_ERROR              (1U << 27)    /* 先前的 I/O 失败 */
#define BM_JUST_DIRTIED          (1U << 28)    /* 写之前已经脏了 */
#define BM_PIN_COUNT_WAITER      (1U << 29)    /* 有人等着钉住页面 */
#define BM_CHECKPOINT_NEEDED     (1U << 30)    /* 必须在检查点时写入 */
```

```
#define BM_PERMANENT            (1U << 31)    /* 永久缓冲 */


/* BufferDesc -- 单个共享缓冲区的共享描述符/共享状态
 *
 * 注意，读写 tag、state、wait_backend_pid 等字段时必须持有缓冲区首部锁 (BM_LOCKED 标记位)
 * 简单地说，refcount、usagecount 标记位组合起来被放入一个原子变量 state 中，而缓冲区首部锁
 * 实际上是嵌入标记位中的一个 bit。这种设计允许我们使用单个原子操作，而不是获取/释放自旋锁
 * 来实现一些操作。例如 refcount 的增减。buf_id 字段在初始化之后再也不会改变，所以不需要锁
 * freeNext 是通过 buffer_strategy_lock 而非 buf_hdr_lock 来保护的。LWLocks 字段可以自己管好自
 * 己。注意，buf_hdr_lock *不是* 用来控制对缓冲区内数据的访问的
 *
 * 我们假设持有首部锁时，没人会修改 state 字段。因此持有缓冲区首部锁的人可以在一次写入
 * 中对 state 变量进行很复杂的更新，包括更新完的同时释放锁（清理 BM_LOCKED 标记位）。此外，不持有
 * 缓冲区首部锁而对 state 进行更新仅限于 CAS 操作，它能确保操作时，没有设置 BM_LOCKED 标记位
 * 不允许使用原子自增/自减、OR/AND 等操作
 *
 * 一个例外是，如果我们钉住了该缓冲区，它的标签除了我们自己之外不会被偷偷修改
 * 所以我们无须锁定自旋锁就可以检视该标签。此外，一次性的标记读取也无须锁定自旋锁，
 * 当我们期待测试标记位不会改变时，这种做法很常见
 *
 * 如果另一个后端钉住了该缓冲区，我们就无法从磁盘页面上物理移除项目。因此后端需要等待
 * 所有其他的钉被移除。移除时它会得到通知，这是通过将它的 PID 存到 wait_backend_pid，并设置
 * BM_PIN_COUNT_WAITER 标记位而实现的。就目前而言，每个缓冲区只能有一个等待者
 *
 * 对于本地缓冲区，我们也使用同样的首部，不过锁字段就没用了，一些标记位也没用。为了避免不必要
 * 的开销，对 state 字段的操作不需要用实际的原子操作（即 pg_atomic_read_u32，
 * pg_atomic_unlocked_write_u32）
 *
 * 增加该结构的尺寸，增减、重排该结构的成员时需要特别小心。保证该结构体小于 64B 对于性能
 * 至关重要 (最常见的 CPU 缓存尺寸)
 */
typedef struct BufferDesc
{
    BufferTagtag;           /* 存储在缓冲区中页面的标识 */
    int           buf_id;            /* 缓冲区的索引编号 (从 0 开始) */

    /* 标记的状态，包含标记位、引用计数、使用计数 */
    /* 9.6 版本使用原子操作替换了很多字段的功能 */
    pg_atomic_uint32 state;

    int           wait_backend_pid;  /* 等待钉页计数的后端进程 PID */
```

```
    int             freeNext;          /* 空闲链表中的链接 */

    LWLock          content_lock;      /* 访问缓冲区内容的锁 */
} BufferDesc;
```

8.4　缓冲区管理器的工作原理

本节介绍缓冲区管理器的工作原理。当后端进程想要访问所需页面时，它会调用 ReadBufferExtended 函数。

函数 ReadBufferExtended 的行为因场景而异，在逻辑上具体可以分为三种情况。每种情况都将用一小节介绍。第 8.4.4 节将介绍 PostgreSQL 中基于时钟扫描的页面置换算法。

8.4.1　访问存储在缓冲池中的页面

当从缓冲池槽中的页面里读取行时，PostgreSQL 进程获取相应缓冲区描述符的共享 content_lock，因而缓冲池槽可以同时被多个进程读取。

当向页面插入（及更新、删除）行时，该 postgres 后端进程获取相应缓冲区描述符的独占 content_lock（注意，这里必须将相应页面的脏位置设为"1"）。

访问完页面后，相应缓冲区描述符的引用计数值减 1。

图 8.8 是访问存储在缓冲池中的页面示意图。

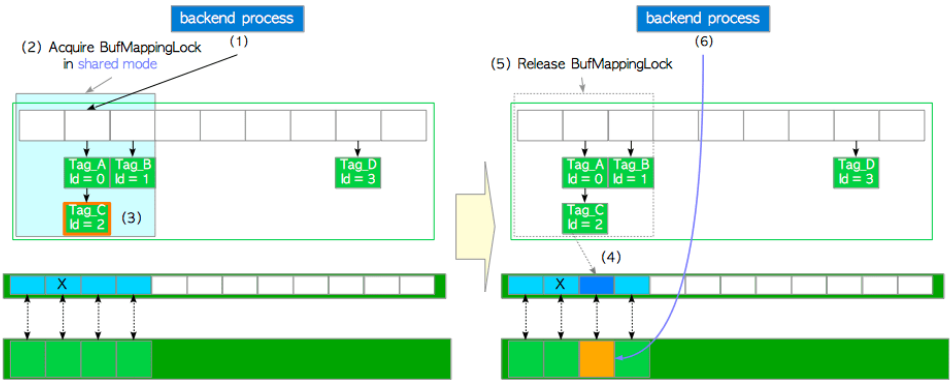

图 8.8　访问存储在缓冲池中的页面

我们来介绍最简单的情况，即所需页面已经存储在缓冲池中。在这种情况下，缓冲区管理器会执行以下步骤：

（1）创建所需页面的 buffer_tag（在本例中 buffer_tag 是'Tag_C'），并使用散列函数计算与描述符相对应的散列桶槽。

（2）获取相应散列桶槽分区上的 BufMappingLock 共享锁。

（3）查找标签为'Tag_C'的条目，并从条目中获取 buffer_id。本例中 buffer_id 为 2。

（4）将 buffer_id=2 的缓冲区描述符钉住，即将描述符的 refcount 和 usage_count 增加 1。

（5）释放 BufMappingLock。

（6）访问 buffer_id=2 的缓冲池槽。

8.4.2　将页面从存储加载到空槽

图 8.9 是将页面从存储加载到空槽的示意图。

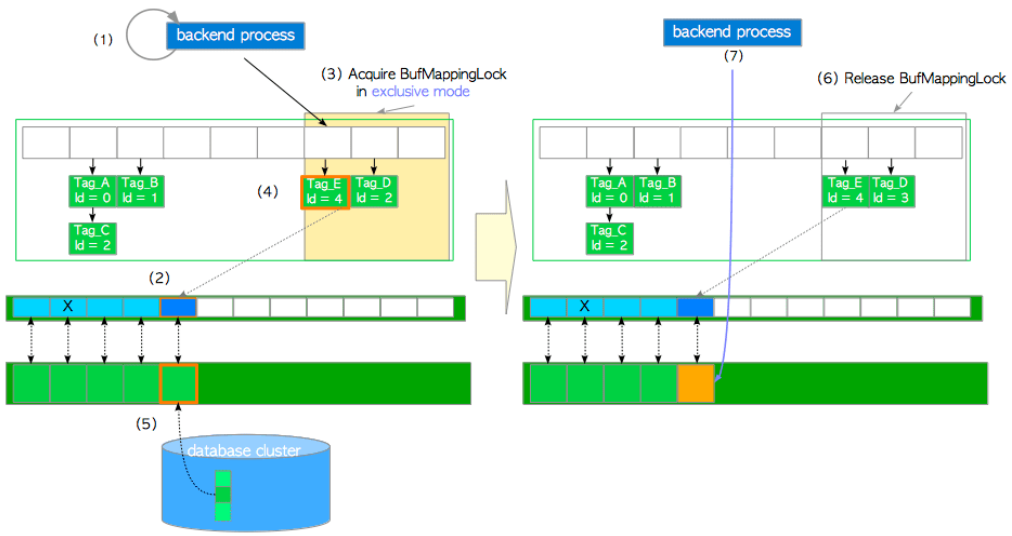

图 8.9　将页面从存储加载到空槽

在第二种情况下，假设所需页面不在缓冲池中，且 freelist 中有空闲元素（空描述符）。

这时，缓冲区管理器将执行以下步骤：

（1）查找缓冲区表（本节假设页面不存在，找不到对应页面）。

第一，创建所需页面的 buffer_tag（本例中 buffer_tag 为'Tag_E'）并计算其散列桶槽。

第二，以共享模式获取相应分区上的 BufMappingLock。

第三，查找缓冲区表（根据假设，这里没找到）。

第四，释放 BufMappingLock。

（2）从 freelist 中获取空缓冲区描述符，并将其钉住。在本例中所获的描述符：buffer_id=4。

（3）以独占模式获取相应分区的 BufMappingLock（此锁将在步骤（6）中被释放）。

（4）创建一条新的缓冲表数据项：buffer_tag='Tag_E'，buffer_id=4，并将其插入缓冲区表中。

（5）将页面数据从存储加载至 buffer_id=4 的缓冲池槽中，如下所示：

第一，以排他模式获取相应描述符的 io_in_progress_lock。

第二，将相应描述符的 IO_IN_PROGRESS 标记位设置为 1，以防其他进程访问。

第三，将所需的页面数据从存储加载到缓冲池插槽中。

第四，更改相应描述符的状态，将 IO_IN_PROGRESS 标记位设置为"0"，且 VALID 标记位设置为"1"。

第五，释放 io_in_progress_lock。

（6）释放相应分区的 BufMappingLock。

（7）访问 buffer_id=4 的缓冲池槽。

8.4.3　将页面从存储加载到受害者缓冲池槽

在这种情况下，假设所有缓冲池槽位都被页面占用，且未存储所需的页面。图 8.10、图 8.11 是将页面从存储加载到受害者缓冲池槽的示意图。

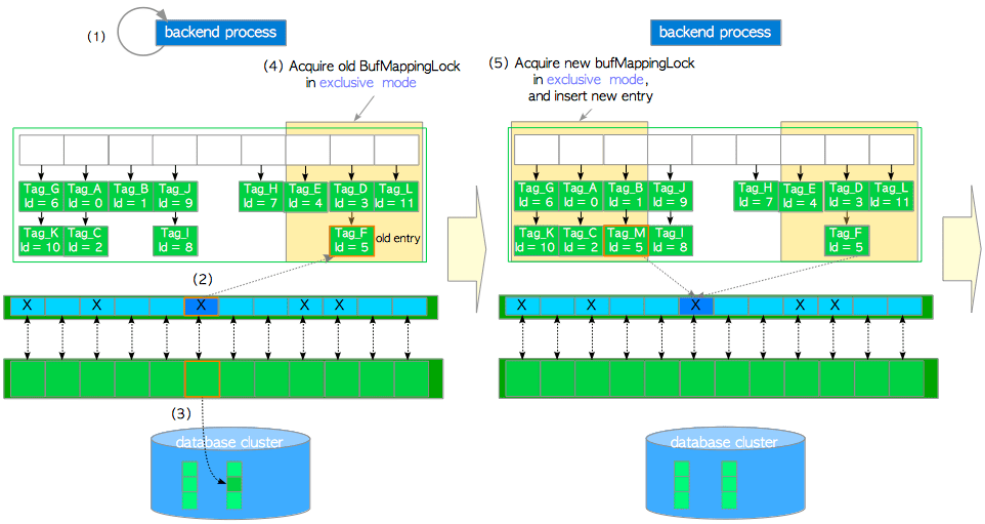

图 8.10　将页面从存储加载到受害者缓冲池槽

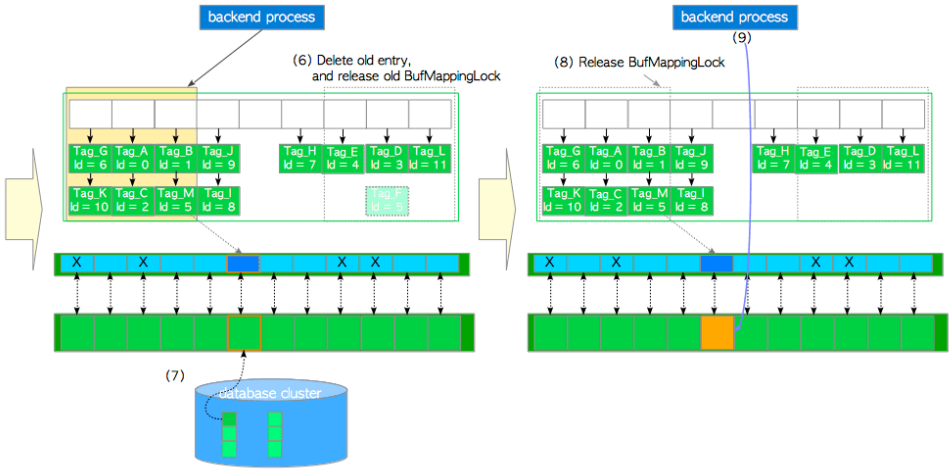

图 8.11　将页面从存储加载到受害者缓冲池槽（接图 8.10）

缓冲区管理器将执行以下步骤：

（1）创建所需页面的 `buffer_tag` 并查找缓冲表。在本例中假设 `buffer_tag` 是`'Tag_M'`（且相应的页面在缓冲区中找不到）。

（2）使用时钟扫描算法选择一个受害者缓冲池槽位，从缓冲表中获取包含着受害者槽位 `buffer_id` 的旧表项，并在缓冲区描述符层将受害者槽位的缓冲区描述符钉住。本例中受害者

槽的 buffer_id=5，旧表项为 Tag_F, id = 5。时钟扫描将在下一节介绍。

（3）如果受害者页面是脏页，则将其刷盘（write & fsync），否则进入步骤（4）。

在使用新数据覆盖脏页之前，必须将脏页写入存储中。脏页的刷盘步骤如下：

第一，获取 buffer_id=5 描述符上的共享 content_lock 和独占 io_in_progress_lock。

第二，更改相应描述符的状态：相应 IO_IN_PROCESS 位设置为"1"，JUST_DIRTIED 位设置为"0"。

第三，根据具体情况，调用 XLogFlush()函数将 WAL 缓冲区上的 WAL 数据写入当前 WAL 段文件（WAL 和 XLogFlush 函数将在第 9 章中介绍）。

第四，将受害者页面的数据刷盘至存储中。

第五，更改相应描述符的状态；将 IO_IN_PROCESS 位设置为"0"，将 VALID 位设置为"1"。

第六，释放 io_in_progress_lock 和 content_lock。

（4）以排他模式获取缓冲区表中旧表项所在分区上的 BufMappingLock。

（5）获取新表项所在分区上的 BufMappingLock，并将新表项插入缓冲表：

第一，创建由新表项：由 buffer_tag='Tag_M'与受害者的 buffer_id 组成的新表项。

第二，以独占模式获取新表项所在分区上的 BufMappingLock。

第三，将新表项插入缓冲区表中。

（6）从缓冲表中删除旧表项，并释放旧表项所在分区的 BufMappingLock。

（7）将目标页面数据从存储加载至受害者槽位，然后用 buffer_id=5 更新描述符的标识字段，将脏位设置为 0，并按流程初始化其他标记位。

（8）释放新表项所在分区上的 BufMappingLock。

（9）访问 buffer_id=5 对应的缓冲区槽位。

8.4.4　页面替换算法：时钟扫描

本节的其余部分介绍了时钟扫描算法。该算法是 NFU（Not Frequently Used）算法的变体，

开销较少，能高效地选出较少使用的页面。

我们将缓冲区描述符想象为一个循环列表，如图 8.12 所示。缓冲区描述符为黑色或灰色的方框，框中的数字显示每个描述符的 usage_count。而 nextVictimBuffer 是一个 32 位的无符号整型变量，它总是指向某个缓冲区描述符并按顺时针顺序旋转。

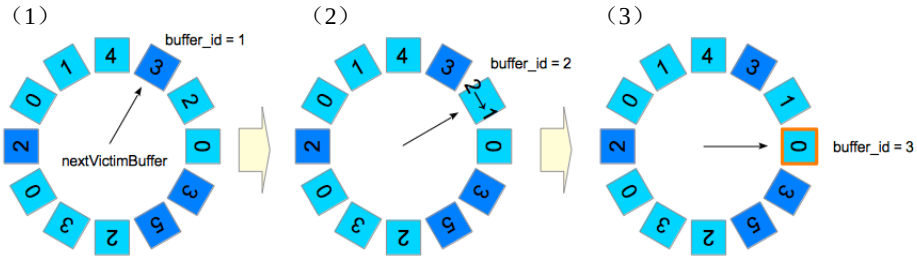

图 8.12　时钟扫描

伪代码：时钟扫描

```
    WHILE true
(1)     获取 nextVictimBuffer 指向的缓冲区描述符
(2)     IF 缓冲区描述符没有被钉住 THEN
(3)         IF 候选缓冲区描述符的 usage_count == 0 THEN
                BREAK WHILE LOOP  /* 该描述符对应的槽就是受害者槽 */
            ELSE
                将候选描述符的 usage_count - 1
            END IF
        END IF
(4)     迭代 nextVictimBuffer，指向下一个缓冲区描述符
    END WHILE
(5) RETURN 受害者页面的 buffer_id
```

（1）获取 nextVictimBuffer 指向的候选缓冲区描述符。

（2）如果候选描述符未被钉住，则进入步骤（3），否则进入步骤（4）。

（3）如果候选描述符的 usage_count 为 0，则选择该描述符对应的槽作为受害者，并进入步骤（5），否则将此描述符的 usage_count 减 1，并继续执行步骤（4）。

（4）将 nextVictimBuffer 迭代至下一个描述符（如果到末尾则回绕至头部）并返回步骤（1），重复直到找到受害者。

（5）返回受害者的 buffer_id。

该算法的伪代码与算法描述如下：

1．nextVictimBuffer 指向第一个描述符（buffer_id = 1），但因为该描述符被钉住了，所以跳过。

2．extVictimBuffer 指向第二个描述符（buffer_id = 2），该描述符未被钉住，但其 usage_count 为 2，因此该描述符的 usage_count 将减 1，而 nextVictimBuffer 迭代至第三个候选描述符。

3．nextVictimBuffer 指向第三个描述符（buffer_id = 3），该描述符未被钉住，但其 usage_count = 0，因而成为本轮的受害者。

当 nextVictimBuffer 扫过未钉住的描述符时，其 usage_count 会减 1。因此只要缓冲池中存在未钉住的描述符，该算法总能在旋转若干次 nextVictimBuffer 后，找到一个 usage_count 为 0 的受害者。

8.5　环形缓冲区

在读写大表时，PostgreSQL 会使用环形缓冲区而不是缓冲池。环形缓冲器是一个很小的临时缓冲区域。当满足下列任一条件时，PostgreSQL 将在共享内存中分配一个环形缓冲区。

1．批量读取。当扫描关系读取数据的大小超过缓冲池的四分之一时，环形缓冲区的大小为 256 KB。

2．批量写入，当执行下列 SQL 命令时，环形缓冲区大小为 16 MB。

- COPY FROM 命令。
- CREATE TABLE AS 命令。
- CREATE MATERIALIZED VIEW 或 REFRESH MATERIALIZED VIEW 命令。
- ALTER TABLE 命令。

3．清理过程，当自动清理守护进程执行清理过程时，环形缓冲区大小为 256 KB。

分配的环形缓冲区将在使用后被立即释放。

环形缓冲区的好处显而易见，如果后端进程在不使用环形缓冲区的情况下读取大表，则所有存储在缓冲池中的页面都会被移除，这会导致缓存命中率降低。环形缓冲区可以避免此问题。

> **为什么批量读取和清理过程的默认环形缓冲区大小为 256 KB**
>
> 源代码中缓冲区管理器目录下的 README 中解释了这个问题。
>
> 顺序扫描使用 256KB 的环形缓冲区，它足够小，因而能放入 L2 缓存中，从而使得操作系统缓存到共享缓冲区的页面传输变得高效。通常更小一点也可以，但环形缓冲区需要足够大到能同时容纳扫描中被钉住的所有页面。

8.6 脏页刷盘

除了置换受害者页面之外，检查点进程和后台写入器进程也会将脏页刷盘至存储中。尽管两个进程都具有相同的功能（脏页刷盘），但是它们有着不同的角色和行为。

检查点进程将检查点记录写入 WAL 段文件，并在检查点开始时进行脏页刷盘。第 9.7 节介绍了检查点和检查点开始的时机。

后台写入器的目的是通过少量多次的脏页刷盘，减少检查点带来的密集写入的影响。后台写入器会一点点地将脏页落盘，尽可能减少对数据库活动造成的影响。在默认情况下，后台写入器每 200ms 被唤醒一次（由参数 `bgwriter_delay` 定义），且最多刷写 `bgwriter_lru_maxpages` 个页面（默认为 100 个页面）。

> **为什么检查点进程与后台写入器相分离**
>
> 在 9.1 及更低版本中，后台写入器会规律性地执行检查点进程。在 9.2 版本中，检查点进程从后台写入被单独剥离出来。原因在一篇题为“将检查点进程与后台写入器相分离”的提案中有介绍。下面是一些摘录：
>
> 当前（在 2011 年）后台写入器进程既执行后台写入，又负责检查点，还处理一些其他的职责。这意味着我们没法在不停止后台写入的情况下执行检查点最终的 fsync。因此，在同一个进程中做两件事会有负面的性能影响。
>
> 此外，在 9.2 版本中，我们的一个目标是通过将轮询循环替换为锁存器，从而降低功耗。bgwriter 中的循环复杂度太高了，以至于无法找到一种简单的使用锁存器的方法。