# Linux 2.6 中的页面回收与反向映射

**为什么要进行页面回收**

操作系统管理内存中的物理页面，同时也担任着内存分配的职责。应用程序可以通过内存分配函数向操作系统申请物理页面；在使用完这些物理页面之后，应用程序可以通过相应的内存释放函数释放这些物理页面。但是，对于内存中的某些物理页面来说，页面的使用者并不会主动释放它们，如果这些物理页面一直被占用而得不到释放，那么无论计算机上可用的物理内存有多少，物理内存迟早都有被用完的时候。所以，对于无法被主动释放的物理页面来说，操作系统就需要提供相应的功能去释放它们，Linux 中提供页面回收算法这样一种机制进行页面回收。

一般来说，用于页缓存的物理页面无法被页面的使用者主动释放，因为它们不知道这些页面何时应该被释放。Linux 中页缓存存在的最大好处就是可以让程序从缓存中快速获取数据，从而提升系统的性能。在系统负载不重的情况下，Linux 操作系统会分配较多的物理页面用于页缓存，从而提高程序的运行效率；但是在系统负载较重的情况下，Linux 操作系统就可能会适当回收用于缓存的页面，并减少用于缓存的页面的分配，从而满足系统中优先级更高的内存分配请求。对于用户进程来说，Linux 操作系统可以在它需要的时候为它分配物理内存，但是当用户进程不再需要这些物理页面的时候，如果用户进程不主动释放占用的页面，Linux 操作系统也不会强制用户进程去释放这些物理页面。基于上述这些情况，当内存中可用的物理页面越来越少，并最终导致内存的使用捉襟见肘的时候，为了保证系统的顺利运行，Linux 操作系统就会根据一定的算法去回收那些长期被占用并且没有得到有效使用的物理页面。

由操作系统内核本身使用的物理页面不在 Linux 操作系统进行页面回收的考虑范围之内，这是因为与用户进程相比，内核不需要占用非常多的内存，回收内核占用的物理页面会显著增加内核代码的复杂性，潜在收益非常低。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-cn-pagerecycle/#ibm-pcon)

**如何进行页面回收**

**哪些页面可以被回收**

内存中并非所有物理页面都是可以进行回收的，总的来说，以下这些种物理页面可以被 Linux 操作系统回收：

* 文件读写操作过程中用于缓冲数据的页面
* 用户地址空间中用于文件内存映射的页面
* 匿名页面：进程用户模式下的堆栈或者是使用 mmap 匿名映射的内存区
* 特殊的用于 slab 分配器的缓存，比如用于缓存文件目录结构 dentry 的 cache，以及用于缓存索引节点 inode 的 cache

在页面被操作系统回收之前，所有与之关联的进程页表项必须要断开与该页面之间的映射关系。对于匿名页面来说，在页面被回收之前，匿名页面中的内容首先需要先被交换到交换区中去；如果要回收的页面是“脏”页面，那么该页面在被回收之前需要先将页面中的数据写回。

除此之外，其他的页面要么不可以被回收，要么根本不必进行回收。比如，内核占用的页面不会被回收；映射到内核空间中的页面也不会被回收；内核在执行的过程中动态生成的页面需要永驻内存；被锁住的页面不能被回收；而没有被占用的物理页面则根本不需要被回收。

**进行页面回收的时机**

Linux 操作系统使用如下这两种机制检查系统内存的使用情况，从而确定可用的内存是否太少从而需要进行页面回收。

* + 周期性的检查：这是由后台运行的守护进程 kswapd 完成的。该进程定期检查当前系统的内存使用情况，当发现系统内空闲的物理页面数目少于特定的阈值时，该进程就会发起页面回收的操作。
  + “内存严重不足”事件的触发：在某些情况下，比如，操作系统忽然需要通过伙伴系统为用户进程分配一大块内存，或者需要创建一个很大的缓冲区，而当时系统中的内存没有办法提供足够多的物理内存以满足这种内存请求，这时候，操作系统就必须尽快进行页面回收操作，以便释放出一些内存空间从而满足上述的内存请求。这种页面回收方式也被称作“直接页面回收”。

如果操作系统在进行了内存回收操作之后仍然无法回收到足够多的页面以满足上述内存要求，那么操作系统只有最后一个选择，那就是使用 OOM( out of memory )killer，它从系统中挑选一个最合适的进程杀死它，并释放该进程所占用的所有页面。

上面介绍的内存回收机制主要依赖于三个字段：pages\_min，pages\_low 以及 pages\_high。每个内存区域（ zone ）都在其区域描述符中定义了这样三个字段，这三个字段的具体含义如下表 1 所示。

**表 1. 字段含义**

|  |  |
| --- | --- |
| **名称** | **字段描述** |
| pages\_min | 区域的预留页面数目，如果空闲物理页面的数目低于 pages\_min，那么系统的压力会比较大，此时，内存区域中急需空闲的物理页面，页面回收的需求非常紧迫。 |
| pages\_low | 控制进行页面回收的最小阈值，如果空闲物理页面的数目低于 pages\_low，那么操作系统内核会开始进行页面回收。 |
| pages\_high | 控制进行页面回收的最大阈值，如果空闲物理页面的数目多于 pages\_high，则内存区域的状态是理想的。 |

**页面回收算法**

Linux 中的页面回收是基于 LRU(least recently used，即最近最少使用 ) 算法的。LRU 算法基于这样一个事实，过去一段时间内频繁使用的页面，在不久的将来很可能会被再次访问到。反过来说，已经很久没有访问过的页面在未来较短的时间内也不会被频繁访问到。因此，在物理内存不够用的情况下，这样的页面成为被换出的最佳候选者。

LRU 算法的基本原理很简单，为每个物理页面绑定一个计数器，用以标识该页面的访问频度。操作系统内核进行页面回收的时候就可以根据页面的计数器的值来确定要回收哪些页面。然而，在硬件上提供这种支持的体系结构很少，Linux 操作系统没有办法依靠这样一种页计数器去跟踪每个页面的访问情况，所以，Linux 在页表项中增加了一个 Accessed 位，当页面被访问到的时候，该位就会被硬件自动置位。该位被置位表示该页面还很年轻，不能被换出去。此后，在系统的运行过程中，该页面的年龄会被操作系统更改。在 Linux 中，相关的操作主要是基于两个 LRU 链表以及两个标识页面状态的标志符，下文会逐一介绍这些相应的数据结构以及 Linux 如何使用这些数据结构进行页面回收。

**LRU 链表**

在 Linux 中，操作系统对 LRU 的实现主要是基于一对双向链表：active 链表和 inactive 链表，这两个链表是 Linux 操作系统进行页面回收所依赖的关键数据结构，每个内存区域都存在一对这样的链表。顾名思义，那些经常被访问的处于活跃状态的页面会被放在 active 链表上，而那些虽然可能关联到一个或者多个进程，但是并不经常使用的页面则会被放到 inactive 链表上。页面会在这两个双向链表中移动，操作系统会根据页面的活跃程度来判断应该把页面放到哪个链表上。页面可能会从 active 链表上被转移到 inactive 链表上，也可能从 inactive 链表上被转移到 active 链表上，但是，这种转移并不是每次页面访问都会发生，页面的这种转移发生的间隔有可能比较长。那些最近最少使用的页面会被逐个放到 inactive 链表的尾部。进行页面回收的时候，Linux 操作系统会从 inactive 链表的尾部开始进行回收。

用于描述内存区域的 struct zone() 中关于这两个链表以及相关的关键字段的定义如下所示：

struct zone {

……

spinlock\_t lru\_lock;

struct list\_head active\_list;

struct list\_head inactive\_list;

unsigned long nr\_active;

unsigned long nr\_inactive;

……

}

各字段含义如下所示：

lru\_lock：active\_list 和 inactive\_list 使用的自旋锁。

active\_list：管理内存区域中处于活跃状态的页面。

inactive\_list：管理内存区域中处于不活跃状态的页面。

nr\_active：active\_list 链表上的页面数目。

nr\_inactive：inactive\_list 链表上的页面数目。

**如何在两个 LRU 链表之间移动页面**

Linux 引入了两个页面标志符 PG\_active 和 PG\_referenced 用于标识页面的活跃程度，从而决定如何在两个链表之间移动页面。PG\_active 用于表示页面当前是否是活跃的，如果该位被置位，则表示该页面是活跃的。PG\_referenced 用于表示页面最近是否被访问过，每次页面被访问，该位都会被置位。Linux 必须同时使用这两个标志符来判断页面的活跃程度，假如只是用一个标志符，在页面被访问时，置位该标志符，之后该页面一直处于活跃状态，如果操作系统不清除该标志位，那么即使之后很长一段时间内该页面都没有或很少被访问过，该页面也还是处于活跃状态。为了能够有效清除该标志位，需要有定时器的支持以便于在超时时间之后该标志位可以自动被清除。然而，很多 Linux 支持的体系结构并不能提供这样的硬件支持，所以 Linux 中使用两个标志符来判断页面的活跃程度。

Linux 2.6 中这两个标志符密切合作，其核心思想如下所示：

* 如果页面被认为是活跃的，则将该页的 PG\_active 置位；否则，不置位。
* 当页面被访问时，检查该页的 PG\_referenced 位，若未被置位，则置位之；若发现该页的 PG\_referenced 已经被置位了，则意味着该页经常被访问，这时，若该页在 inactive 链表上，则置位其 PG\_active 位，将其移动到 active 链表上去，并清除其 PG\_referenced 位的设置；如果页面的 PG\_referenced 位被置位了一段时间后，该页面没有被再次访问，那么 Linux 操作系统会清除该页面的 PG\_referenced 位，因为这意味着这个页面最近这段时间都没有被访问。
* PG\_referenced 位同样也可以用于页面从 active 链表移动到 inactive 链表。对于某个在 active 链表上的页面来说，其 PG\_active 位被置位，如果 PG\_referenced 位未被置位，给定一段时间之后，该页面如果还是没有被访问，那么该页面会被清除其 PG\_active 位，挪到 inactive 链表上去。

Linux 中实现在 LRU 链表之间移动页面的关键函数如下所示（本文涉及的源代码均是基于 Linux 2.6.18.1 版本的）：

* mark\_page\_accessed()：当一个页面被访问时，则调用该函数相应地修改 PG\_active 和 PG\_referenced。
* page\_referenced()：当操作系统进行页面回收时，每扫描到一个页面，就会调用该函数设置页面的 PG\_referenced 位。如果一个页面的 PG\_referenced 位被置位，但是在一定时间内该页面没有被再次访问，那么该页面的 PG\_referenced 位会被清除。
* activate\_page()：该函数将页面放到 active 链表上去。
* shrink\_active\_list()：该函数将页面移动到 inactive 链表上去。

**LRU 缓存**

前边提到，页面根据其活跃程度会在 active 链表和 inactive 链表之间来回移动，如果要将某个页面插入到这两个链表中去，必须要通过自旋锁以保证对链表的并发访问操作不会出错。为了降低锁的竞争，Linux 提供了一种特殊的缓存：LRU 缓存，用以批量地向 LRU 链表中快速地添加页面。有了 LRU 缓存之后，新页不会被马上添加到相应的链表上去，而是先被放到一个缓冲区中去，当该缓冲区缓存了足够多的页面之后，缓冲区中的页面才会被一次性地全部添加到相应的 LRU 链表中去。Linux 采用这种方法降低了锁的竞争，极大地提升了系统的性能。

LRU 缓存用到了 pagevec 结构，如下所示 :

struct pagevec {

unsigned long nr;

unsigned long cold;

struct page \*pages[PAGEVEC\_SIZE];

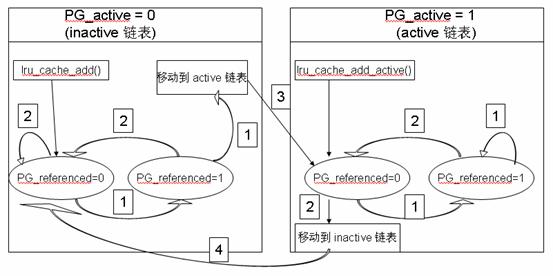
};

pagevec 这个结构就是用来管理 LRU 缓存中的这些页面的。该结构定义了一个数组，这个数组中的项是指向 page 结构的指针。一个 pagevec 结构最多可以存在 14 个这样的项（PAGEVEC\_SIZE 的默认值是 14）。当一个 pagevec 的结构满了，那么该 pagevec 中的所有页面会一次性地被移动到相应的 LRU 链表上去。

用来实现 LRU 缓存的两个关键函数是 lru\_cache\_add() 和 lru\_cache\_add\_active()。前者用于延迟将页面添加到 inactive 链表上去，后者用于延迟将页面添加到 active 链表上去。这两个函数都会将要移动的页面先放到页向量 pagevec 中，当 pagevec 满了（已经装了 14 个页面的描述符指针），pagevec 结构中的所有页面才会被一次性地移动到相应的链表上去。

下图概括总结了上文介绍的如何在两个链表之间移动页面，以及 LRU 缓存在其中起到的作用：

**图 1. 页面在 LRU 链表之间移动示意图**

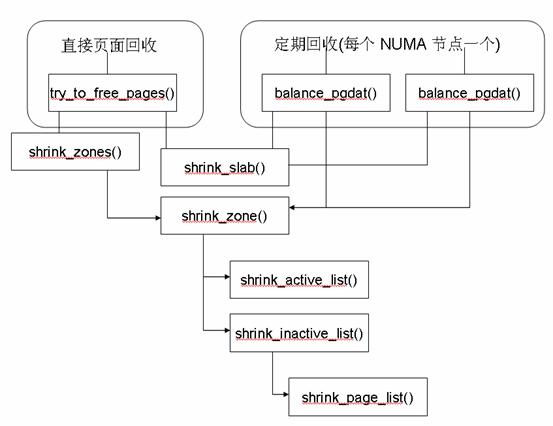


其中，1 表示函数 mark\_page\_accessed()，2 表示函数 page\_referenced()，3 表示函数 activate\_page()，4 表示函数 shrink\_active\_list()。

**页面回收的实现**

Linux 操作系统进行页面回收需要考虑的方面很多，下图列出了 Linux 操作系统进行页面回收的关键代码流程图，该图给出了实现页面回收的关键代码函数名，并说明它们之间是如何彼此链接的。

**图 2. 页面回收关键代码流程图**



上文提到 Linux 中页面回收主要是通过两种方式触发的，一种是由“内存严重不足”事件触发的；一种是由后台进程 kswapd 触发的，该进程周期性地运行，一旦检测到内存不足，就会触发页面回收操作。对于第一种情况，系统会调用函数 try\_to\_free\_pages() 去检查当前内存区域中的页面，回收那些最不常用的页面。对于第二种情况，函数 balance\_pgdat() 是入口函数。

当 NUMA 上的某个节点的低内存区域调用函数 try\_to\_free\_pages() 的时候，该函数会反复调用 shrink\_zones() 以及 shrink\_slab() 释放一定数目的页面，默认值是 32 个页面。如果在特定的循环次数内没有能够成功释放 32 个页面，那么页面回收会调用 OOM killer 选择并杀死一个进程，然后释放它占用的所有页面。函数 shrink\_zones() 会对内存区域列表中的所有区域分别调用 shrink\_zone() 函数，后者是从内存回收最近最少使用页面的入口函数。

对于定期页面检查并进行回收的入口函数 balance\_pgdat() 来说，它主要调用的函数是 shrink\_zone() 和 shrink\_slab()。从上图中我们也可以看出，进行页面回收的两条代码路径最终汇合到函数 shrink\_zone() 和函数 shrink\_slab() 上。

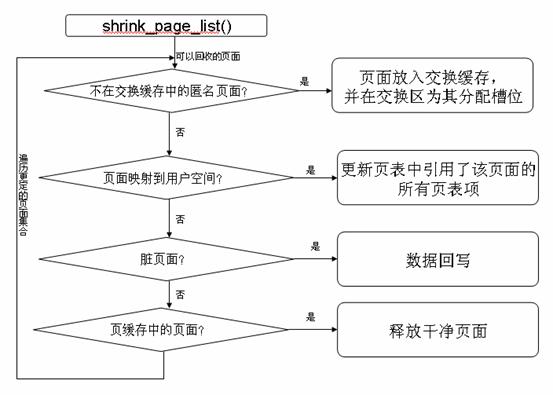
**函数 shrink\_zone()**

其中，shrink\_zone() 函数是 Linux 操作系统实现页面回收的最核心的函数之一，它实现了对一个内存区域的页面进行回收的功能，该函数主要做了两件事情：

* 将某些页面从 active 链表移到 inactive 链表，这是由函数 shrink\_active\_list() 实现的。
* 从 inactive 链表中选定一定数目的页面，将其放到一个临时链表中，这由函数 shrink\_inactive\_list() 完成。该函数最终会调用 shrink\_page\_list() 去回收这些页面。

函数 shrink\_page\_list() 返回的是回收成功的页面数目。概括来说，对于可进行回收的页面，该函数主要做了这样几件事情，其代码流程图如下所示：

**图 3. 函数 shrink\_page\_list() 实现的关键功能**



* 对于匿名页面来说，在回收此类页面时，需要将其数据写入到交换区。如果尚未为该页面分配交换区槽位，则先分配一个槽位，并将该页面添加到交换缓存。同时，将相关的 page 实例加入到交换区，这样，对该页面的处理就可以跟其他已经建立映射的页面一样；
* 如果该页面已经被映射到一个或者多个进程的页表项中，那么必须找到所有引用该页面的进程，并更新页表中与这些进程相关的所有页表项。在这里，Linux 2.6 操作系统会利用反向映射机制去检查哪些页表项引用了该页面，关于反向映射的内容在后边会有介绍；
* 如果该页面中的数据是脏的，那么数据必须要被回写；
* 释放页缓存中的干净页面。

**函数 shrink\_slab()**

函数 shrink\_slab() 是用来回收磁盘缓存所占用的页面的。Linux 操作系统并不清楚这类页面是如何使用的，所以如果希望操作系统回收磁盘缓存所占用的页面，那么必须要向操作系统内核注册 shrinker 函数，shrinker 函数会在内存较少的时候主动释放一些该磁盘缓存占用的空间。函数 shrink\_slab() 会遍历 shrinker 链表，从而对所有注册了 shrinker 函数的磁盘缓存进行处理。

从实现上来看，shrinker 函数和 slab 分配器并没有固定的联系，只是当前主要是 slab 缓存使用 shrinker 函数最多。

注册 shrinker 是通过函数 set\_shrinker() 实现的，解除 shrinker 注册是通过函数 remove\_shrinker() 实现的。当前，Linux 操作系统中主要的 shrinker 函数有如下几种：

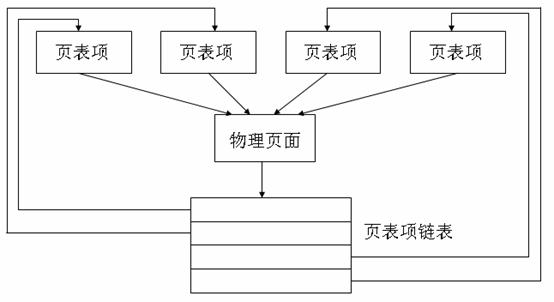
* shrink\_dcache\_memory()：该 shrinker 函数负责 dentry 缓存。
* shrink\_icache\_memory()：该 shrinker 函数负责 inode 缓存。
* mb\_cache\_shrink\_fn()：该 shrinker 函数负责用于文件系统元数据的缓存。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-cn-pagerecycle/#ibm-pcon)

**反向映射（reverse mapping）**

前文介绍过，在回收一个物理页面之前，需要查找到所有关联了该物理页面的页表项，并逐一更新这些页表项。Linux 2.6 使用了反向映射这种机制用于快速定位那些引用了某个物理页面的所有页表项。Linux 操作系统为物理页面建立一个链表，用于指向引用了该物理页面的所有页表项。其基本思想如下图所述：

**图 4. 反向映射的基本思想**



**反向映射技术的发展历史**

在 Linux 2.4 中，为了确定某个要回收的物理页面都被哪些页表项引用，必须要遍历所有进程，这是一项非常耗资源和时间的工程。为了更加有效地回收一个共享页面，Linux 在 2.5 版本的开发期间引入了反向映射这样一种机制。这种机制建立了物理页面和所有映射了该物理页面的页表项之间的一种关联，从而让操作系统可以快速定位引用了该物理页面的所有页表项。在 Linux 2.6 版本中，反向映射算法又经历了大量改进。

在 Linux 2.5 版本中，反向映射技术的实现主要是基于页表项链表。操作系统为每一个物理页面都维护了一个链表，所有与该物理页面关联的页表项都会被放到这个链表上。这种方法会存在一些问题：

* 空间资源的消耗：为每个物理页面维护这样一个链表，需要占用大量的内存空间。
* 时间资源的消耗：回收一个物理页面的时候，需要先获取该链表上的锁，然后遍历相应的反向映射链表，链表上的项越多，需要的时间就越多。

后来，Linux 2.6 引入了基于对象的反向映射机制。这种方法也是为物理页面设置一个用于反向映射的链表，但是链表上的节点并不是引用了该物理页面的所有页表项，而是相应的虚拟内存区域（ vm\_area\_struct 结构），虚拟内存区域通过内存描述符（ mm\_struct 结构）找到页全局目录，从而找到相应的页表项。相对于前一种方法来说，用于表示虚拟内存区域的描述符比用于表示页面的描述符要少得多，所以遍历后边这种反向映射链表所消耗的时间也会少很多。

**基于对象的反向映射的实现**

**数据结构**

page 结构中与基于对象的反向映射相关的关键字段有两个：\_mapcount 和 mapping。

struct page {

atomic\_t \_mapcount;

union {

……

struct {

……

struct address\_space \*mapping;

};

……

};

* 字段 \_mapcount 表明共享该物理页面的页表项的数目。该计数器可用于快速检查该页面除所有者之外有多少个使用者在使用，初始值是 -1，每增加一个使用者，该计数器加 1。
* 字段 mapping 用于区分匿名页面和基于文件映射的页面，如果该字段的最低位被置位了，那么该字段包含的是指向 anon\_vma 结构（用于匿名页面）的指针；否则，该字段包含指向 address\_space 结构的指针（用于基于文件映射的页面）。

匿名页面和文件映射页面分别采用了不同的底层数据结构去存放与页面相关的虚拟内存区域。对于匿名页面来说，与该页面相关的虚拟内存区域存放在结构 anon\_vma 中定义的双向链表中。结构 anon\_vma 定义很简单，如下所示：

struct anon\_vma {

spinlock\_t lock;

struct list\_head head;

};

而对于基于文件映射的页面来说，与匿名页面不同的是，与该页面相关的虚拟内存区域的存放是利用了优先级搜索树这种数据结构的。这是因为对于匿名页面来说，页面虽然可以是共享的，但是一般情况下，共享匿名页面的使用者的数目不会很多；而对于基于文件映射的页面来说，共享页面的使用者的数目可能会非常多，使用优先级搜索树这种结构可以更加快速地定位那些引用了该页面的虚拟内存区域。操作系统会为每一个文件都建立一个优先级搜索树，其根节点可以通过结构 address\_space 中的 i\_mmap 字段获取。

struct address\_space {

……

struct prio\_tree\_root i\_mmap;

……

}

Linux 2.6 中使用 (radix,size,heap) 来表示优先级搜索树中的节点。其中，radix 表示内存区域的起始位置，heap 表示内存区域的结束位置，size 与内存区域的大小成正比。在优先级搜索树中，父节点的 heap 值一定不会小于子节点的 heap 值。在树中进行查找时，根据节点的 radix 值进行。程序可以根据 size 值区分那些具有相同 radix 值的节点。

在用于表示虚拟内存区域的结构 vm\_area\_struct 中，与上边介绍的双向链表和优先级搜索树相关的字段如下所示：

struct vm\_area\_struct {

struct mm\_struct \* vm\_mm;

……

union {

struct {

struct list\_head list;

void \*parent;

struct vm\_area\_struct \*head;

} vm\_set;

struct raw\_prio\_tree\_node prio\_tree\_node;

} shared;

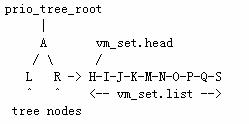
struct list\_head anon\_vma\_node;

struct anon\_vma \*anon\_vma;

};

与匿名页面的双向链表相关的字段是 anon\_vma\_node 和 anon\_vma。union shared 则与文件映射页面使用的优先级搜索树相关。字段 anon\_vma 指向 anon\_vma 表；字段 anon\_vma\_node 将映射该页面的所有虚拟内存区域链接起来；union shared 中的 prio\_tree\_node 结构用于表示优先级搜索树的一个节点；在某些情况下，比如不同的进程的内存区域可能映射到了同一个文件的相同部分，也就是说这些内存区域具有相同的（radix,size,heap）值，这个时候 Linux 就会在树上相应的节点（树上原来那个具有相同 （radix,size,heap） 值的内存区域）上接一个双向链表用来存放这些内存区域，这个链表用 vm\_set.list 来表示；树上那个节点指向的链表中的第一个节点是表头，用 vm\_set.head 表示；vm\_set.parent 用于表示是否是树结点。下边给出一个小图示简单说明一下 vm\_set.list 和 vm\_set.head。

**图 5. vm\_set.list 和 vm\_set.head**

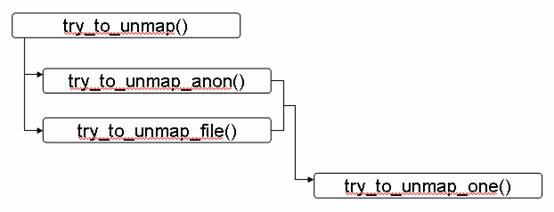


通过结构 vm\_area\_struct 中的 vm\_mm 字段可以找到对应的 mm\_struct 结构，在该结构中找到页全局目录，从而定位所有相关的页表项。

**使用反向映射**

在进行页面回收的时候，Linux 2.6 在前边介绍的 shrink\_page\_list() 函数中调用 try\_to\_unmap() 函数去更新所有引用了回收页面的页表项。其代码流程如下所示：

**图 6. 实现函数 try\_to\_unmap() 的关键代码流程图**



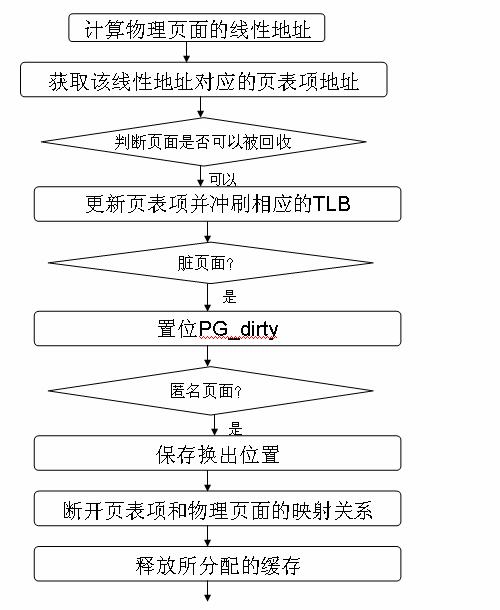
函数 try\_to\_unmap() 分别调用了两个函数 try\_to\_unmap\_anon() 和 try\_to\_unmap\_file()，其目的都是检查并确定都有哪些页表项引用了同一个物理页面，但是，由于匿名页面和文件映射页面分别采用了不同的数据结构，所以二者采用了不同的方法。

函数 try\_to\_unmap\_anon() 用于匿名页面，该函数扫描相应的 anon\_vma 表中包含的所有内存区域，并对这些内存区域分别调用 try\_to\_unmap\_one() 函数。

函数 try\_to\_unmap\_file() 用于文件映射页面，该函数会在优先级搜索树中进行搜索，并为每一个搜索到的内存区域调用 try\_to\_unmap\_one() 函数。

两条代码路径最终汇合到 try\_to\_unmap\_one() 函数中，更新引用特定物理页面的所有页表项的操作都是在这个函数中实现的。该函数实现的关键功能如下图所示：

**图 7. 函数 try\_to\_unmap\_one() 实现的关键功能**



对于给定的物理页面来说，该函数会根据计算出来的线性地址找到对应的页表项地址，并更新页表项。对于匿名页面来说，换出的位置必须要被保存下来，以便于该页面下次被访问的时候可以被换进来。并非所有的页面都是可以被回收的，比如被 mlock() 函数设置过的内存页，或者最近刚被访问过的页面，等等，都是不可以被回收的。一旦遇上这样的页面，该函数会直接跳出执行并返回错误代码。如果涉及到页缓存中的数据，需要设置页缓存中的数据无效，必要的时候还要置位页面标识符以进行数据回写。该函数还会更新相应的一些页面使用计数器，比如前边提到的 \_mapcount 字段，还会相应地更新进程拥有的物理页面数目等。

**使用反向映射的优缺点**

使用反向映射机制所带来的好处是显而易见的：可以快速定为引用了某个物理页面的所有页表项，这极大地方便了操作系统进行页面回收。相对于之前的遍历方法来说，反向映射机制在很大程度上减少了操作系统在页面回收上所占用的 CPU 时间。

但是，使用反向映射所面临的挑战也是很明显的，不管采用上述介绍的哪种方法建立反向映射，都不可避免地要消耗掉一定的内存空间，区别就在于用哪种方法占用的空间会更少，整体性能会更好。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-cn-pagerecycle/#ibm-pcon)

**总结**

页面回收是 Linux 内存管理中比较复杂的一个部分，涉及到的相关内容非常多，本文也不是面面俱到。反向映射是 Linux 2.5 开发过程中一个比较大的亮点，该技术在后续 Linux 2.6 版本中又得到了更进一步的发展。本文的目的是想帮助读者理清 Linux 2.6 中的页面回收和反向映射机制，本文通过相关的数据结构和关键的代码流程介绍了 Linux 操作系统如何利用反向映射机制有效地进行页面回收。关于 Linux 操作系统如何建立反向映射的内容，本文没有做详尽介绍，感兴趣的读者可以自行参考内核源代码。

**参考资料**

**学习**

* [内核比较: 2.6 内核中改进了内存管理](http://www.ibm.com/developerworks/cn/linux/l-mem26/)上可以看到 Linux 2.6 中关于内存管理的一些改进。
* 在 Understanding the Linux Kernel(3rdEdition) 和 Linux 内核情景分析中，有关于 Linux 中内存管理的介绍。
* 本文中引用的内核代码来自于 2.6.18.1 版本的内核，请参考内核源码：<http://www.kernel.org/>。
* 在 [developerWorks Linux 专区](http://www.ibm.com/developerworks/cn/linux/) 寻找为 Linux 开发人员（包括 [Linux 新手入门](http://www.ibm.com/developerworks/cn/linux/newto/)）准备的更多参考资料，查阅我们 [最受欢迎的文章和教程](http://www.ibm.com/developerworks/cn/linux/best2009/index.html)。
* 在 developerWorks 上查阅所有 [Linux 技巧](http://www.ibm.com/developerworks/cn/views/linux/libraryview.jsp?search_by=Linux+%E6%8A%80%E5%B7%A7) 和 [Linux 教程](http://www.ibm.com/developerworks/cn/views/linux/libraryview.jsp?type_by=%E6%95%99%E7%A8%8B)。
* 随时关注 developerWorks [技术活动](http://www.ibm.com/developerworks/cn/offers/techbriefings/)和[网络广播](http://www.ibm.com/developerworks/cn/swi/)。

**讨论**

* 欢迎加入 [developerWorks 中文社区](http://www.ibm.com/developerworks/cn/community/)。