



下载APP



02 基础篇（二）| Page Cache是怎样产生和释放的？

2020-08-17 邵亚方

Linux内核技术实战课

[进入课程 >](#)**讲述：邵亚方**

时长 09:51 大小 9.04M



你好，我是邵亚方。

上一讲，我们主要讲了“什么是 Page Cache”（What）， “为什么需要 Page Cache”（Why），我们这节课还需要继续了解一下“How”：**也就是 Page Cache 是如何产生和释放的。**

在我看来，对 Page Cache 的“What-Why-How”都有所了解之后，你才会对它引发的问题，比如说 Page Cache 引起的 load 飙高问题或者应用程序的 RT 抖动问题更加了如指掌，从而防范于未然。



其实，Page Cache 是如何产生和释放的，通俗一点的说就是它的“生”（分配）与“死”（释放），即 Page Cache 的生命周期，那么接下来，我们就先来看一下它是如

何“诞生”的。

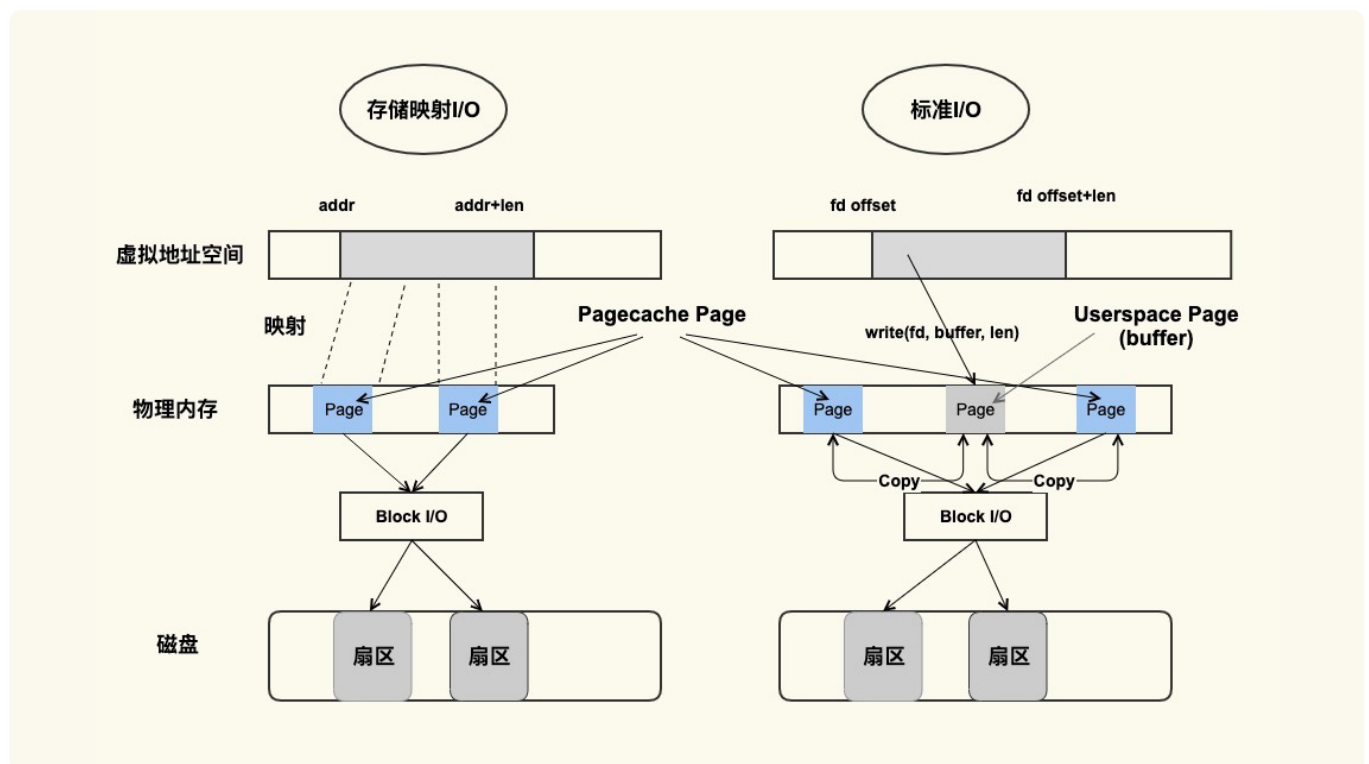
Page Cache 是如何“诞生”的？

Page Cache 的产生有两种不同的方式：

Buffered I/O（标准 I/O）；

Memory-Mapped I/O（存储映射 I/O）。

这两种方式分别都是如何产生 Page Cache 的呢？来看下面这张图：



Page Cache产生方式示意图


从图中你可以看到，虽然二者都能产生 Page Cache，但是二者的还是有些差异的：

标准 I/O 是写的 (write(2)) 用户缓冲区 (Userspace Page 对应的内存)，然后再将用户缓冲区里的数据拷贝到内核缓冲区 (Pagecache Page 对应的内存)；如果是读的 (read(2)) 话则是先从内核缓冲区拷贝到用户缓冲区，再从用户缓冲区读数据，也就是 buffer 和文件内容不存在任何映射关系。

对于存储映射 I/O 而言，则是直接将 Pagecache Page 给映射到用户地址空间，用户直接读写 Pagecache Page 中内容。

显然，存储映射 I/O 要比标准 I/O 效率高一些，毕竟少了“用户空间到内核空间互相拷贝”的过程。这也是很多应用开发者发现，为什么使用内存映射 I/O 比标准 I/O 方式性能要好一些的主要原因。

我们来用具体的例子演示一下 Page Cache 是如何“诞生”的，就以其中的标准 I/O 为例，因为这是我们最常使用的一种方式，如下是一个简单的示例脚本：

 复制代码

```
1  #!/bin/sh
2
3  #这是我们用来解析的文件
4  MEM_FILE="/proc/meminfo"
5
6  #这是在该脚本中将要生成的一个新文件
7  NEW_FILE="/home/yafang/dd.write.out"
8
9  #我们用来解析的Page Cache的具体项
10 active=0
11 inactive=0
12 pagecache=0
13
14 IFS=' '
15
16 #从/proc/meminfo中读取File Page Cache的大小
17 function get_filecache_size()
18 {
19     items=0
20     while read line
21     do
22         if [[ "$line" =~ "Active:" ]]; then
23             read -ra ADDR <<<"$line"
24             active=${ADDR[1]}
25             let "items=$items+1"
26         elif [[ "$line" =~ "Inactive:" ]]; then
27             read -ra ADDR <<<"$line"
28             inactive=${ADDR[1]}
29             let "items=$items+1"
30         fi
31     done < $MEM_FILE
32
33     if [ $items -eq 2 ]; then
34         break;
35     fi
36 done < $MEM_FILE
37 }
38
39 #读取File Page Cache的初始大小
40 get_filecache_size
```

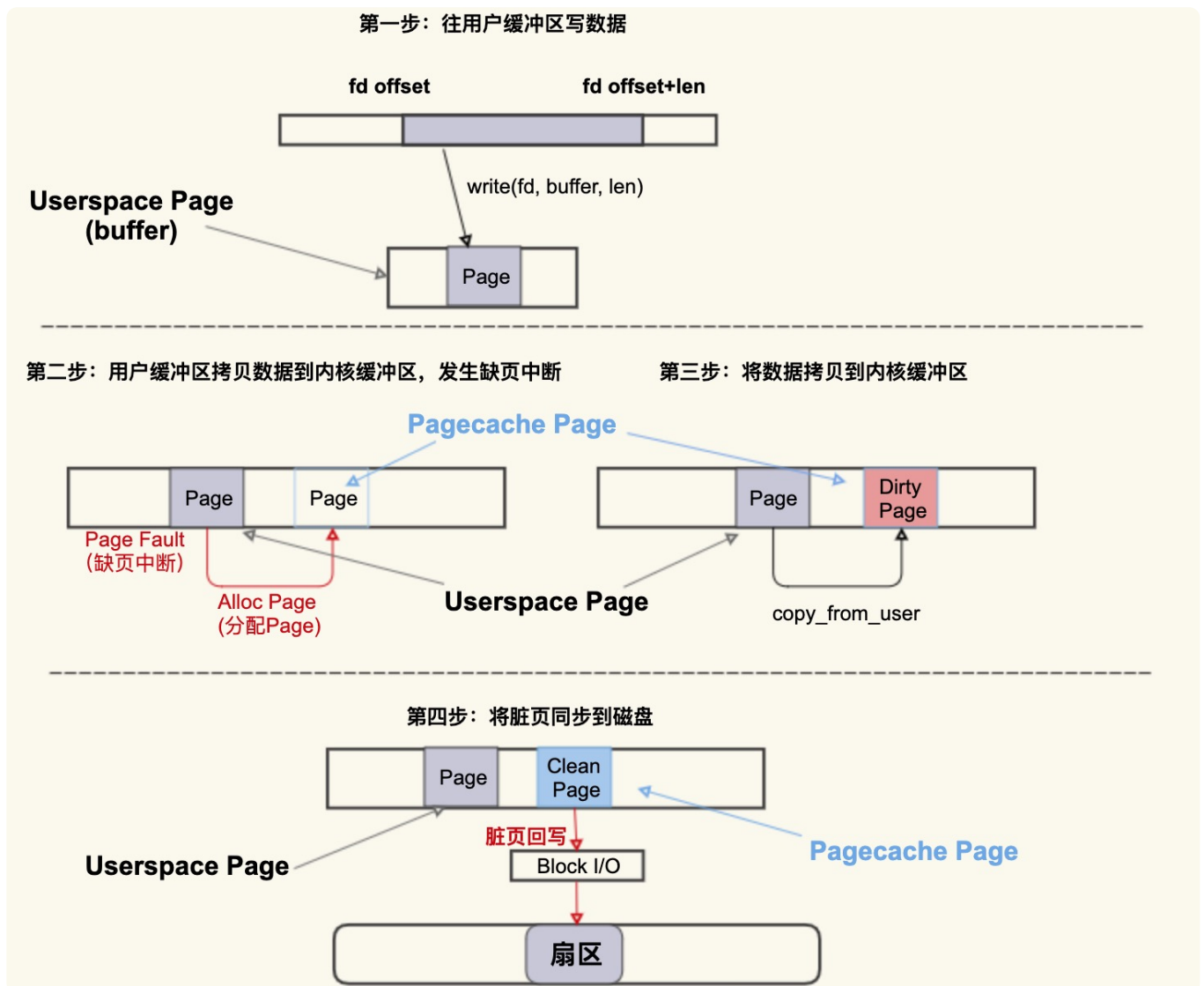
```
41 let filecache="$active + $inactive"
42
43 #写一个新文件，该文件的大小为1048576 KB
44 dd if=/dev/zero of=$NEW_FILE bs=1024 count=1048576 &> /dev/null
45
46 #文件写完后，再次读取File Page Cache的大小
47 get_filecache_size
48
49 #两次的差异可以近似为该新文件内容对应的File Page Cache
50 #之所以用近似是因为在运行的过程中也可能会有其他Page Cache产生
51 let size_increased="$active + $inactive - $filecache"
52
53 #输出结果
54 echo "File size 1048576KB. File Cache increased" $size inc
```

在这里我提醒你一下，在运行该脚本前你要确保系统中有足够多的 free 内存（避免内存紧张产生回收行为），最终的测试结果是这样的：

```
File size 1048576KB, File Cache increased 1048648KB
```

通过这个脚本你可以看到，在创建一个文件的过程中，代码中 /proc/meminfo 里的 Active(file) 和 Inactive(file) 这两项会随着文件内容的增加而增加，它们增加的大小跟文件大小是一致的（这里之所以略有不同，是因为系统中还有其他程序在运行）。另外，如果你观察得很仔细的话，你会发现增加的 Page Cache 是 Inactive(File) 这一项，**你可以去思考一下为什么会这样？**这里就作为咱们这节课的思考题。

当然，这个过程看似简单，但是它涉及的内核机制还是很多的，换句话说，可能引起问题的地方还是很多的，我们用一张图简单描述下这个过程：



这个过程大致可以描述为：首先往用户缓冲区 buffer(这是 Userspace Page) 写入数据，然后 buffer 中的数据拷贝到内核缓冲区（这是 Pagecache Page），如果内核缓冲区中还没有这个 Page，就会发生 Page Fault 会去分配一个 Page，拷贝结束后该 Pagecache Page 是一个 Dirty Page（脏页），然后该 Dirty Page 中的内容会同步到磁盘，同步到磁盘后，该 Pagecache Page 变为 Clean Page 并且继续存在系统中。

我建议你可以将 Alloc Page 理解为 Page Cache 的“诞生”，将 Dirty Page 理解为 Page Cache 的婴幼儿时期（最容易生病的时期），将 Clean Page 理解为 Page Cache 的成年时期（在这个时期就很少会生病了）。

但是请注意，并不是所有人都有童年的，比如孙悟空，一出生就是成人了，Page Cache 也一样，如果是读文件产生的 Page Cache，它的内容跟磁盘内容是一致的，所以它一开始是 Clean Page，除非改写了里面的内容才会变成 Dirty Page（返老还童）。

就像我们为了让婴幼儿健康成长，要悉心照料他 / 她一样，为了提前发现或者预防婴幼儿时期的 Page Cache 发病，我们也需要一些手段来观测它：

[复制代码](#)

```
1 $ cat /proc/vmstat | egrep "dirty|writeback"
2 nr_dirty 40
3 nr_writeback 2
```

如上所示，nr_dirty 表示当前系统中积压了多少脏页，nr_writeback 则表示有多少脏页正在回写到磁盘中，他们两个的单位都是 Page(4KB)。

通常情况下，小朋友们（Dirty Pages）聚集在一起（脏页积压）不会有什么问题，但在非常时期比如流感期间，就很容易导致聚集的小朋友越多病症就会越严重。与此类似，Dirty Pages 如果积压得过多，在某些情况下也会容易引发问题，至于是哪些情况，又会出现哪些问题，我们会在案例篇中具体讲解。

明白了 Page Cache 的“诞生”后，我们再来看一下 Page Cache 的“死亡”：它是如何被释放的？

Page Cache 是如何“死亡”的？

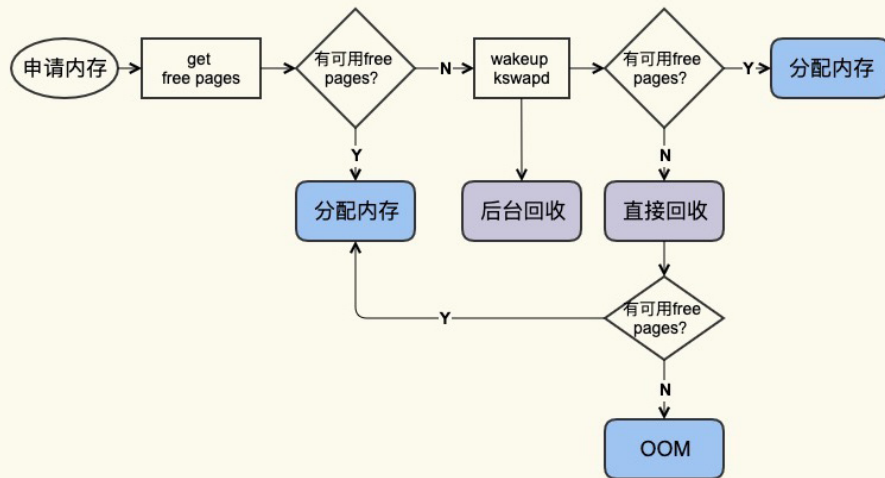
你可以把 Page Cache 的回收行为 (Page Reclaim) 理解为 Page Cache 的“自然死亡”。

言归正传，我们知道，服务器运行久了后，系统中 free 的内存会越来越少，用 free 命令来查看，大部分都会是 used 内存或者 buff/cache 内存，比如说下面这台生产环境中服务器的内存使用情况：

[复制代码](#)

```
1 $ free -g
2      total    used    free   shared  buff/cache   available
3 Mem:     125     41      6         0         79         82
4 Swap:      0      0      0
```

free 命令中的 buff/cache 中的这些就是“活着”的 Page Cache，那它们什么时候会“死亡”（被回收）呢？我们来看一张图：



你可以看到，应用在申请内存的时候，即使没有 free 内存，只要还有足够可回收的 Page Cache，就可以通过回收 Page Cache 的方式来申请到内存，**回收的方式主要是两种：直接回收和后台回收。**

那它是具体怎么回收的呢？你要怎么观察呢？其实在我看来，观察 Page Cache 直接回收和后台回收最简单方便的方式是使用 sar：

复制代码

```

1 $ sar -B 1
2 02:14:01 PM pgpgin/s pgpgout/s fault/s majflt/s pgfree/s pgscank/s pgscan
3
4
5 02:14:01 PM      0.14      841.53 106745.40      0.00 41936.13      0.00      0
6 02:15:01 PM      5.84      840.97  86713.56      0.00 43612.15     717.81      0
7 02:16:01 PM     95.02      816.53 100707.84      0.13 46525.81    3557.90      0
8 02:17:01 PM     10.56      901.38 122726.31      0.27 54936.13    8791.40      0
9 02:18:01 PM    108.14      306.69  96519.75      1.15 67410.50   14315.98     31
10 02:19:01 PM      5.97      489.67  88026.03      0.18 48526.07    1061.53      0
  
```

借助上面这些指标，你可以更加明确地观察内存回收行为，下面是这些指标的具体含义：

pgscank/s : kswapd(后台回收线程) 每秒扫描的 page 个数。

pgscand/s: Application 在内存申请过程中每秒直接扫描的 page 个数。

pgsteal/s: 扫描的 page 中每秒被回收的个数。

%vmeff: $\text{pgsteal}/(\text{pgscank}+\text{pgscand})$, 回收效率，越接近 100 说明系统越安全，越接近 0 说明系统内存压力越大。

这几个指标也是通过解析 /proc/vmstat 里面的数据来得出的，对应关系如下：

sar -B	/proc/vmstat
pgscank	pgscan_kswapd
pgscand	pgscan_direct
pgsteal	pgsteal_kswapd + pgsteal_direct

关于这几个指标我说一个小插曲，要知道，如果 Linux Kernel 本身设计不当会给你带来困扰。所以，如果你观察到应用程序的结果跟你的预期并不一致，也有可能是因为内核设计上存在问题，你可以对内核持适当的怀疑态度哦，下面这个是我最近遇到的一个案例。

如果你对 Linus 有所了解的话，应该会知道 Linus 对 Linux Kernel 设计的第一原则是 “never break the user space”。很多内核开发者在设计内核特性的时候，会忽略掉新特性对应用程序的影响，比如在前段时间就有人 (Google 的一个内核开发者) 提交了一个 patch 来修改内存回收这些指标的含义，但是最终被我和另外一个人 (Facebook 的一个内核开发者) 把他的这个改动给否决掉了。具体的细节并不是咱们这节课的重点，我就不多说了，我建议你在课下看这个讨论：[🔗 \[PATCH\] mm: vmscan: consistent update to pgsteal and pgscan](#)，可以看一下内核开发者们在设计内核特性时是如何思考的，这会有利于你更加全面的去了解整个系统，从而让你的应用程序更好地融入到系统中。

课堂总结

以上就是本节课的全部内容了，本节课，我们主要讲了 Page Cache 是如何“诞生”的，以及如何“死亡”的，我要强调这样几个重点：

Page Cache 是在应用程序读写文件的过程中产生的，所以在读写文件之前你需要留意是否还有足够的内存来分配 Page Cache；

Page Cache 中的脏页很容易引起问题，你要重点注意这一块；

在系统可用内存不足的时候就会回收 Page Cache 来释放出来内存，我建议你可以通过 `sar` 或者 `/proc/vmstat` 来观察这个行为从而更好的判断问题是否跟回收有关

总的来说，Page Cache 的生命周期对于应用程序而言是相对比较透明的，即它的分配与回收都是由操作系统来进行管理的。正是因为这种“透明”的特征，所以应用程序才会难以控制 Page Cache，Page Cache 才会容易引发那么多问题。在接下来的案例篇里，我们就来看看究竟会引发什么样子的的问题，以及你正确的分析思路是什么样子的。

课后作业

因为每个人的关注点都不一样，对问题的理解也不一样。假如你是一个应用开发者，你会更加关注应用的性能和稳定性；假如你是一个运维人员，你会更加关注系统的稳定性；假如你是初学内核的开发者，你会想要关注内核的实现机制。

所以我留了不同的作业题，主题是围绕 “Inactive 与 Active Page Cache 的关系” 当然了，对应的难度也不同：

如果你是一名应用开发者，那么我想问问你为什么第一次读写某个文件，Page Cache 是 Inactive 的？如何让它变成 Active 的呢？在什么情况下 Active 的又会变成 Inactive 的呢？明白了这个问题，你会对应用性能调优有更加深入的理解。

如果你是一名运维人员，那么建议你思考一下，系统中有哪些控制项可以影响 Inactive 与 Active Page Cache 的大小或者二者的比例？

如果你是一名初学内核的开发者，那么我想问你，对于匿名页而言，当产生一个匿名页后它会首先放在 Active 链表上；而对于文件页而言，当产生一个文件页后它会首先放在 Inactive 链表上。请问为什么会这样子？这是合理的吗？欢迎在留言区分享你的看法。

感谢你的阅读，如果你认为这节课的内容有收获，也欢迎把它分享给你的朋友，我们下一讲见。

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 01 基础篇（一） | 如何用数据观测Page Cache？

精选留言 (1)

写留言



0xFE
2020-08-17

/proc/sys/vm/pagecache_limit_mb 限制使用大小
展开

