



下载APP



10 分析篇 | 内存泄漏时，我们该如何一步步找到根因？

2020-09-10 邵亚方

Linux内核技术实战课

[进入课程 >](#)**讲述：邵亚方**

时长 08:45 大小 8.03M



你好，我是邵亚方。

通过我们前面的基础篇以及案例篇的学习，你对内存泄漏应该有了自己的一些理解。这节课我来跟你聊一聊系统性地分析内存泄漏问题的方法：也就是说，在面对内存泄漏时，我们该如何一步步去找到根因？

不过，我不会深入到具体语言的实现细节，以及具体业务的代码逻辑中，而是会从 Linux 系统上通用的一些分析方法入手。这样，不论你使用什么开发语言，不论你在开发什么，它总能给你提供一些帮助。



如何定位出是谁在消耗内存？

内存泄漏的外在表现通常是系统内存不够，严重的话可能会引起 OOM (Out of Memory)，甚至系统宕机。那在发生这些现象时，惯用的分析套路是什么呢？

首先，我们需要去找出到底是谁在消耗内存，/proc/meminfo 可以帮助我们来快速定位出问题所在。

/proc/meminfo 中的项目很多，我们没必要全部都背下来，不过有些项是相对容易出问题的，也是你在遇到内存相关的问题时，需要重点去排查的。我将这些项列了一张表格，也给出了每一项有异常时的排查思路。

/proc/meminfo	含义以及排查思路
Active(anon)	在active anon lru上的page，它和下一项之间会互相转换。
Inactive(anon)	在inactive anon lru上的page，特征是只可以被交换到swap分区，不可以被回收。这一项和前一项就是应用程序使用malloc()或者mmap()匿名方式来申请并且写后的内存，如果这两项过大，那需要去排查下应用程序的这两类内存申请方式。 这两项如果有异常，通常的排查思路如下： 1. 使用top来对进程消耗内存进行排序，找出哪些进程内存开销大； 2. 找出来内存异常的进程后，就可以使用pmap来分析该进程了； 3. 如果没有任何进程内存开销大，那需要去重点排查tmpfs。
Unevictable	这部分内存存在系统内存紧张时不能被回收，所以如果这部分内存持续增长，是很容易引起严重问题的。 这一项主要是由下面这些内存组成： 1. ram disk或者ramfs消耗的内存； 2. 以SHM_LOCK方式来申请的Shmem； 3. 使用mlock()系列函数来管理的内存。 所以当这一项过大时，你需要重点去看系统以及应用程序中这几种内存使用方式是否正确。
Mlocked	Mlocked其实是Unevictable的一种，只是它是最常使用的方式，所以被单独列了出来。这一项的值小于等于Unevictable。 针对这一项，你可以去排查mlock()方式保护的内存。
AnonPages	匿名映射页，请注意AnonPages != Active(anon) + Inactive(anon)，因为Shmem（包括tmpfs）虽然属于Active(anon) 或者Inactive(anon)，但是它们有对应的内存文件，所以不属于AnonPages。这是很容易困惑人的地方。总之，请记住，Active(anon) 和 Inactive(anon)是用来表示不可回收但是可以被交换到swap分区的内存，而AnonPages则是指没有对应文件的内存，二者的角度不一样。这也是很容易迷惑人的地方。 如果这一项有异常，你可以去排查以malloc()方式申请的内存，或者以mmap(PROT_WRITE, MAP_ANON MAP_PRIVATE)方式来申请的内存。
Mapped	应用程序使用mmap(2)方式来申请，并且还没有被unmap的内存。这个unmap包括应用程序主动调用munmap(2)，以及内核内存回收时的unmap。 如果这一项有异常，你需要去排查以mmap()方式申请的内存。
Shmem	共享内存，这里面特别需要注意的是tmpfs，你可以参考“08讲”来看更多的细节。 这一项的排查思路通常是： 1. 使用top来查看进程的SHR这一项，找出哪些进程的SHR较大； 2. 找出来内存异常的进程后，就可以使用pmap来分析该进程了； 3. 如果没有进程的SHR大，那需要去重点排查tmpfs。
Slab	Slab分为可以被回收的(SReclaimable)以及不可以被回收的(SUnreclaim)，其中不可被回收的Slab如果发生泄漏，比如kmalloc申请的内存没有释放，那问题会非常严重，具体你可以参考“09讲”来看更多的细节。这一项的排查思路通常是： 1. 使用slabtop来查看哪一项slab较大； 2. 排查驱动程序以kmalloc()方式申请的内存。
VmallocUsed	通过vmalloc方式来分配的内核内存，这部分内存泄漏的危害是很严重的，具体你可以参考“09讲”来看更多的细节。针对这一项，你可以去查看/proc/vmallocinfo，来判断哪些驱动程序以vmalloc()方式申请的内存较多。如果驱动程序已经被卸载，那驱动信息不会体现在/proc/vmallocinfo中，这就需要排查系统中曾经运行过的驱动程序了。

总之，如果进程的内存有问题，那使用 top 就可以观察出来；如果进程的内存没有问题，那你可以从 /proc/meminfo 入手来一步步地去深入分析。

接下来，我们分析一个实际的案例，来看看如何分析进程内存泄漏是什么原因导致的。

如何去分析进程的内存泄漏原因？

这是我多年以前帮助一个小伙伴分析的内存泄漏问题。这个小伙伴已经使用 top 排查出了业务进程的内存异常，但是不清楚该如何去进一步分析。

他遇到的这个异常是，业务进程的虚拟地址空间（VIRT）被消耗很大，但是物理内存（RES）使用得却很少，所以他怀疑是进程的虚拟地址空间有内存泄漏。

我们在“[🔗06 讲](#)”中也讲过，出现该现象时，可以用 top 命令观察（这是当时保存的生产环境信息，部分信息做了脱敏处理）：

[📄 复制代码](#)

```
1  PID USER      PR  NI  VIRT  RES  SHR S %CPU  %MEM    TIME+  COMMAND
2  31108 app        20   0   285g  4.0g  19m S  60.6  12.7   10986:15 app_server
```

可以看到 app_server 这个程序的虚拟地址空间（VIRT 这一项）很大，有 285GB。

那该如何追踪 app_server 究竟是哪里存在问题呢？

我们可以用 pidstat 命令（关于该命令，你可以[🔗man pidstat](#)）来追踪下该进程的内存行为，看看能够发现什么现象。

[📄 复制代码](#)

```
1  $ pidstat -r -p 31108 1
2
3
4  04:47:00 PM      31108      353.00      0.00 299029776 4182152  12.73  app_server
5  ...
6  04:47:59 PM      31108      149.00      0.00 299029776 4181052  12.73  app_server
7  04:48:00 PM      31108      191.00      0.00 299040020 4181188  12.73  app_server
8  ...
9  04:48:59 PM      31108      179.00      0.00 299040020 4181400  12.73  app_server
10 04:49:00 PM      31108      183.00      0.00 299050264 4181524  12.73  app_server
11 ...
12 04:49:59 PM      31108      157.00      0.00 299050264 4181456  12.73  app_server
13 04:50:00 PM      31108      207.00      0.00 299060508 4181560  12.73  app_server
14 ...
15 04:50:59 PM      31108      127.00      0.00 299060508 4180816  12.73  app_server
16 04:51:00 PM      31108      172.00      0.00 299070752 4180956  12.73  app_server
```

如上所示，在每个整分钟的时候，VSZ 会增大 10244KB，这看起来是一个很有规律的现象。然后，我们再来看下增大的这个内存区域到底是什么，你可以通过 `/proc/PID/smmaps` 来看（关于 `/proc` 提供的信息，你可以回顾我们课程的“[🔗05 讲](#)”）：

增大的内存区域，具体如下：

[📄 复制代码](#)

```
1 $ cat /proc/31108/smmaps
2 ...
3 7faae0e49000-7faae1849000 rw-p 00000000 00:00 0
4 Size:                10240 kB
5 Rss:                  80 kB
6 Pss:                  80 kB
7 Shared_Clean:         0 kB
8 Shared_Dirty:         0 kB
9 Private_Clean:        0 kB
10 Private_Dirty:       80 kB
11 Referenced:          60 kB
12 Anonymous:           80 kB
13 AnonHugePages:        0 kB
14 Swap:                 0 kB
15 KernelPageSize:       4 kB
16 MMUPageSize:          4 kB
17 7faae1849000-7faae184a000 ---p 00000000 00:00 0
18 Size:                 4 kB
19 Rss:                   0 kB
20 Pss:                   0 kB
21 Shared_Clean:         0 kB
22 Shared_Dirty:         0 kB
23 Private_Clean:        0 kB
24 Private_Dirty:        0 kB
25 Referenced:           0 kB
26 Anonymous:            0 kB
27 AnonHugePages:        0 kB
28 Swap:                 0 kB
29 KernelPageSize:       4 kB
30 MMUPageSize:          4 kB
```

可以看到，它包括：一个私有地址空间，这从 `rw-p` 这个属性中的 `private` 可以看出来；以及一个保护页，这从 `---p` 这个属性可以看出来，即进程无法访问。对于有经验的开发者而言，从这个 4K 的保护页就可以猜测出应该跟线程栈有关了。

然后我们跟踪下进程申请这部分地址空间的目的是什么，通过 `strace` 命令来跟踪系统调用就可以了。因为 VIRT 的增加，它的系统调用函数无非是 `mmap` 或者 `brk`，那么我们只需要 `strace` 的结果来看下 `mmap` 或 `brk` 就可以了。

用 `strace` 跟踪如下：

[复制代码](#)

```
1 $ strace -t -f -p 31108 -o 31108.strace
```

线程数较多，如果使用 `-f` 来跟踪线程，跟踪的信息量也很大，逐个搜索日志里面的 `mmap` 或者 `brk` 真是眼花缭乱，所以我们来 `grep` 一下这个大小 (10489856 即 10244KB)，然后过滤下就好了：

[复制代码](#)

```
1 $ cat 31108.strace | grep 10489856
2 31152 23:00:00 mmap(NULL, 10489856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON
3 31151 23:01:00 mmap(NULL, 10489856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON
4 31157 23:02:00 mmap(NULL, 10489856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON
5 31158 23:03:00 mmap(NULL, 10489856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON
6 31165 23:04:00 mmap(NULL, 10489856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON
7 31163 23:05:00 mmap(NULL, 10489856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON
8 31153 23:06:00 mmap(NULL, 10489856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON
9 31155 23:07:00 mmap(NULL, 10489856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON
10 31149 23:08:00 mmap(NULL, 10489856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON
11 31147 23:09:00 mmap(NULL, 10489856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON
12 31159 23:10:00 mmap(NULL, 10489856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON
13 31157 23:11:00 mmap(NULL, 10489856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON
14 31148 23:12:00 mmap(NULL, 10489856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON
15 31150 23:13:00 mmap(NULL, 10489856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON
16 31173 23:14:00 mmap(NULL, 10489856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON
```

从这个日志我们可以看到，出错的是 `mmap()` 这个系统调用，那我们再来看下 `mmap` 这个内存的目的：

[复制代码](#)

```
1 31151 23:01:00 mmap(NULL, 10489856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON
2 31151 23:01:00 mprotect(0x7fa94bbc0000, 4096, PROT_NONE <unfinished ...> <<<
3 31151 23:01:00 clone(<unfinished ...> <<< 创建线程
4 31151 23:01:00 <... clone resumed> child_stack=0x7fa94c5afe50, flags=CLONE_VM|
5 |CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID
```

```
6 |CLONE_CHILD_CLEARTID, parent_tidptr=0x7fa94c5c09d0, tls=0x7fa94c5c0700, child
```

可以看出，这是在 clone 时申请的线程栈。到这里你可能会有一个疑问：既然线程栈消耗了这么多的内存，那理应有很多才对啊？

但是实际上，系统中并没有很多 app_server 的线程，那这是为什么呢？答案其实比较简单：线程短暂执行完毕后就退出了，可是 mmap 的线程栈却没有被释放。

我们来写一个简单的程序复现这个现象，问题的复现是很重要的，如果很复杂的问题可以用简单的程序来复现，那就是最好的结果了。

如下是一个简单的复现程序：mmap 一个 40K 的线程栈，然后线程简单执行一下就退出。

[复制代码](#)


```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/mman.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
6 #define _SCHED_H
7 #define __USE_GNU
8 #include <bits/sched.h>
9
10 #define STACK_SIZE 40960
11
12 int func(void *arg)
13 {
14     printf("thread enter.\n");
15     sleep(1);
16     printf("thread exit.\n");
17
18     return 0;
19 }
20
21
22 int main()
23 {
24     int thread_pid;
25     int status;
26     int w;
27
28     while (1) {
29         void *addr = mmap(NULL, STACK_SIZE, PROT_READ|PROT_WRITE, MAP_PRIVATE|
30             if (addr == NULL) {
31
```

```

32         perror("mmap");
33         goto error;
34     }
35     printf("creat new thread...\n");
36     thread_pid = clone(&func, addr + STACK_SIZE, CLONE_SIGHAND|CLONE_FS|CL
37     printf("Done! Thread pid: %d\n", thread_pid);
38     if (thread_pid != -1) {
39         do {
40             w = waitpid(-1, NULL, __WCLONE | __WALL);
41             if (w == -1) {
42                 perror("waitpid");
43                 goto error;
44             }
45         } while (!WIFEXITED(status) && !WIFSIGNALED(status));
46     }
47     sleep(10);
48 }
49
50 error:
51     return 0;

```

然后我们用 pidstat 观察该进程的执行，可以发现它的现象跟生产环境中的问题是一致的：

 复制代码

```

1 $ pidstat -r -p 535 5
2 11:56:51 PM      UID      PID minflt/s  majflt/s     VSZ    RSS   %MEM  Command
3 11:56:56 PM      0       535     0.20     0.00    4364    360   0.00  a.out
4 11:57:01 PM      0       535     0.00     0.00    4364    360   0.00  a.out
5 11:57:06 PM      0       535     0.20     0.00    4404    360   0.00  a.out
6 11:57:11 PM      0       535     0.00     0.00    4404    360   0.00  a.out
7 11:57:16 PM      0       535     0.20     0.00    4444    360   0.00  a.out
8 11:57:21 PM      0       535     0.00     0.00    4444    360   0.00  a.out
9 11:57:26 PM      0       535     0.20     0.00    4484    360   0.00  a.out
10 11:57:31 PM      0       535     0.00     0.00    4484    360   0.00  a.out
11 11:57:36 PM      0       535     0.20     0.00    4524    360   0.00  a.out
12 ^C
13 Average:         0       535     0.11     0.00    4435    360   0.00  a.out

```

你可以看到，VSZ 每 10s 增大 40K，但是增加的那个线程只存在了 1s 就消失了。

至此我们就可以推断出 app_server 的代码哪里有问题了，然后小伙伴去修复该代码 Bug，很快就把该问题给解决了。

当然了，应用程序的内存泄漏问题其实是千奇百怪的，分析方法也不尽相同，我们讲述这个案例的目的是为了告诉你一些通用的分析技巧。我们掌握了这些通用分析技巧，很多时候就可以以不变来应万变了。

课堂总结

这节课我们讲述了系统性分析 Linux 上内存泄漏问题的分析方法，要点如下：

top 工具和 /proc/meminfo 文件是分析 Linux 上内存泄漏问题，甚至是所有内存问题的第一步，我们先找出来哪个进程或者哪一项有异常，然后再针对性地分析；

应用程序的内存泄漏千奇百怪，所以你需要掌握一些通用的分析技巧，掌握了这些技巧很多时候就可以以不变应万变。但是，这些技巧的掌握，是建立在你的基础知识足够扎实的基础上。你需要熟练掌握我们这个系列课程讲述的这些基础知识，熟才能生巧。

课后作业

请写一个内存泄漏的程序，然后观察 /proc/[pid]/maps 以及 smaps 的变化（pid 即内存泄漏的程序的 pid）。欢迎你在留言区与我讨论。

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

更多课程推荐

程序员的数学基础课

在实战中重新理解数学

黄申

LinkedIn 资深数据科学家



涨价倒计时 🕒

今日秒杀 **¥79**, 9月11日涨价至 **¥129**

© 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 09 分析篇 | 如何对内核内存泄漏做些基础的分析？

下一篇 11 基础篇 | TCP连接的建立和断开受哪些系统配置影响？

精选留言 (1)

写留言



ermaot

2020-09-10

曲径通幽，豁然开朗。基础要扎实，但工具也要熟啊

展开 ∨

