

## 26 | 案例篇：如何找出狂打日志的“内鬼”？

2019-01-18 倪朋飞



朗读人：冯永吉

时长12:37 大小11.56M



你好，我是倪朋飞。

前两节，我们学了文件系统和磁盘的 I/O 原理，我先带你复习一下。

文件系统，是对存储设备上的文件进行组织管理的一种机制。为了支持各类不同的文件系统，Linux 在各种文件系统上，抽象了一层虚拟文件系统 VFS。

它定义了一组所有文件系统都支持的数据结构和标准接口。这样，应用程序和内核中的其他子系统，就只需要跟 VFS 提供的统一接口进行交互。

在文件系统的下层，为了支持各种不同类型的存储设备，Linux 又在各种存储设备的基础上，抽象了一个通用块层。

通用块层，为文件系统和应用程序提供了访问块设备的标准接口；同时，为各种块设备的驱动程序提供了统一的框架。此外，通用块层还会对文件系统和应用程序发送过来的 I/O 请求进行排队，并通过重新排序、请求合并等方式，提高磁盘读写的效率。

通用块层的下一层，自然就是设备层了，包括各种块设备的驱动程序以及物理存储设备。

文件系统、通用块层以及设备层，就构成了 Linux 的存储 I/O 栈。存储系统的 I/O，通常是整个系统中最慢的一环。所以，Linux 采用多种缓存机制，来优化 I/O 的效率，比方说，

为了优化文件访问的性能，采用页缓存、索引节点缓存、目录项缓存等多种缓存机制，减少对下层块设备的直接调用。

同样的，为了优化块设备的访问效率，使用缓冲区来缓存块设备的数据。

不过，在碰到文件系统和磁盘的 I/O 问题时，具体应该怎么定位和分析呢？今天，我就以一个最常见的应用程序记录大量日志的案例，带你来分析这种情况。

## 案例准备

本次案例还是基于 Ubuntu 18.04，同样适用于其他的 Linux 系统。我使用的案例环境如下所示：

机器配置：2 CPU，8GB 内存

预先安装 docker、sysstat 等工具，如 `apt install docker.io sysstat`

这里要感谢唯品会资深运维工程师阳祥义帮忙，分担了今天的案例。这个案例，是一个用 Python 开发的小应用，为了方便运行，我把它打包成了一个 Docker 镜像。这样，你只要运行 Docker 命令，就可以启动它。


接下来，打开一个终端，SSH 登录到案例所用的机器中，并安装上述工具。跟以前一样，案例中所有命令，都默认以 root 用户运行。如果你是用普通用户身份登陆系统，请运行 `sudo su root` 命令，切换到 root 用户。

到这里，准备工作就完成了。接下来，我们正式进入操作环节。

温馨提示：案例中 Python 应用的核心逻辑比较简单，你可能一眼就能看出问题，但实际生产环境中的源码就复杂多了。所以，我依旧建议，操作之前别看源码，避免先入为主，要把它当成一个黑盒来分析。这样，你可以更好把握住，怎么从系统的资源使用问题出发，分析出瓶颈所在的应用，以及瓶颈在应用中大概的位置。


## 案例分析

首先，我们在终端中执行下面的命令，运行今天的目标应用：

 复制代码

```
1 $ docker run -v /tmp:/tmp --name=app -itd feisky/logapp
```

然后，在终端中运行 `ps` 命令，确认案例应用正常启动。如果操作无误，你应该可以在 `ps` 的输出中，看到一个 `app.py` 的进程：

 复制代码


```
1 $ ps -ef | grep /app.py
2 root      18940 18921 73 14:41 pts/0    00:00:02 python /app.py
```

接着，我们来看看系统有没有性能问题。要观察哪些性能指标呢？前面文章中，我们知道 CPU、内存和磁盘 I/O 等系统资源，很容易出现资源瓶颈，这就是我们观察的方向了。我们来观察一下这些资源的使用情况。

当然，动手之前你应该想清楚，要用哪些工具来做，以及工具的使用顺序又是怎样的。你可以先回忆下前面的案例和思路，自己想一想，然后再继续下面的步骤。

我的想法是，我们可以先用 `top`，来观察 CPU 和内存的使用情况；然后再用 `iostat`，来观察磁盘的 I/O 情况。

所以，接下来，你可以在终端中运行 `top` 命令，观察 CPU 和内存的使用情况：

 复制代码

```
1 # 按 1 切换到每个 CPU 的使用情况
2 $ top
```

```

3 top - 14:43:43 up 1 day,  1:39,  2 users,  load average: 2.48, 1.09, 0.63
4 Tasks: 130 total,   2 running,  74 sleeping,   0 stopped,   0 zombie
5 %Cpu0  :  0.7 us,  6.0 sy,   0.0 ni,  0.7 id, 92.7 wa,   0.0 hi,   0.0 si,   0.0 st
6 %Cpu1  :  0.0 us,   0.3 sy,   0.0 ni, 92.3 id,  7.3 wa,   0.0 hi,   0.0 si,   0.0 st
7 KiB Mem : 8169308 total,  747684 free,   741336 used,  6680288 buff/cache
8 KiB Swap:          0 total,          0 free,          0 used. 7113124 avail Mem
9
10  PID USER      PR  NI   VIRT   RES    SHR S  %CPU %MEM    TIME+  COMMAND
11 18940 root      20   0 656108 355740  5236 R   6.3  4.4   0:12.56 python
12 1312  root      20   0 236532 24116   9648 S   0.3  0.3   9:29.80 python3

```


观察 top 的输出，你会发现，CPU0 的使用率非常高，它的系统 CPU 使用率（sys%）为 6%，而 iowait 超过了 90%。这说明 CPU0 上，可能正在运行 I/O 密集型的进程。不过，究竟是什么原因呢？这个疑问先保留着，我们先继续看完。

接着我们来看，进程部分的 CPU 使用情况。你会发现，python 进程的 CPU 使用率已经达到了 6%，而其余进程的 CPU 使用率都比较低，不超过 0.3%。看起来 python 是个可疑进程。记下 python 进程的 PID 号 18940，我们稍后分析。

最后再看内存的使用情况，总内存 8G，剩余内存只有 730 MB，而 Buffer/Cache 占用内存高达 6GB 之多，这说明内存主要被缓存占用。虽然大部分缓存可回收，我们还是得了解下缓存的去处，确认缓存使用都是合理的。

到这一步，你基本可以判断出，CPU 使用率中的 iowait 是一个潜在瓶颈，而内存部分的缓存占比较大，那磁盘 I/O 又是怎么样的情况呢？

我们在终端中按 Ctrl+C，停止 top 命令，再运行 iostat 命令，观察 I/O 的使用情况：

 复制代码

```

1 # -d 表示显示 I/O 性能指标，-x 表示显示扩展统计（即所有 I/O 指标）
2 $ iostat -x -d 1
3 Device            r/s      w/s    rkB/s    wkB/s   rrqm/s   wrqm/s  %rrqm  %wrqm  r_await
4 loop0             0.00     0.00     0.00     0.00     0.00     0.00    0.00    0.00     0
5 sdb                0.00     0.00     0.00     0.00     0.00     0.00    0.00    0.00     0
6 sda               0.00    64.00     0.00  32768.00     0.00     0.00    0.00    0.00     0

```

还记得这些性能指标的含义吗？先自己回忆一下，如果实在想不起来，查看上一节的内容，或者用 `man iostat` 查询。

观察 `iostat` 的最后一列，你会看到，磁盘 `sda` 的 I/O 使用率已经高达 99%，很可能已经接近 I/O 饱和。


再看前面的各个指标，每秒写磁盘请求数是 64，写大小是 32 MB，写请求的响应时间为 7 秒，而请求队列长度则达到了 1100。

超慢的响应时间和特长的请求队列长度，进一步验证了 I/O 已经饱和的猜想。此时，`sda` 磁盘已经遇到了严重的性能瓶颈。

到这里，也就可以理解，为什么前面看到的 `iowait` 高达 90% 了，这正是磁盘 `sda` 的 I/O 瓶颈导致的。接下来的重点就是分析 I/O 性能瓶颈的根源了。那要怎么知道，这些 I/O 请求相关的进程呢？

不知道你还记不记得，上一节我曾提到过，可以用 `pidstat` 或者 `iostat`，观察进程的 I/O 情况。这里，我就用 `pidstat` 来看一下。

使用 `pidstat` 加上 `-d` 参数，就可以显示每个进程的 I/O 情况。所以，你可以在终端中运行如下命令来观察：

 复制代码

```
1 $ pidstat -d 1
2
3 15:08:35      UID      PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
4 15:08:36        0    18940      0.00  45816.00      0.00     96    python
5
6 15:08:36      UID      PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
7 15:08:37        0      354      0.00      0.00      0.00     350  jbd2/sda1-8
8 15:08:37        0    18940      0.00  46000.00      0.00     96    python
9 15:08:37        0    20065      0.00      0.00      0.00    1503  kworker/u4:2
```

从 `pidstat` 的输出，你可以发现，只有 `python` 进程的写比较大，而且每秒写的数据超过 45 MB，比上面 `iostat` 发现的 32MB 的结果还要大。很明显，正是 `python` 进程导致了 I/O 瓶颈。



再往下看 iodelay 项。虽然只有 python 在大量写数据，但你应该注意到了，有两个进程（kworker 和 jbd2）的延迟，居然比 python 进程还大很多。

这其中，kworker 是一个内核线程，而 jbd2 是 ext4 文件系统中，用来保证数据完整性的内核线程。他们都是保证文件系统基本功能的内核线程，所以具体细节暂时就不用管了，我们只需要明白，它们延迟的根源还是大量 I/O。


综合 pidstat 的输出来看，还是 python 进程的嫌疑最大。接下来，我们来分析 python 进程到底在写什么。

首先留意一下 python 进程的 PID 号，18940。看到 18940，你有没有觉得熟悉？其实前面在使用 top 时，我们记录过的 CPU 使用率最高的进程，也正是它。不过，虽然在 top 中使用率最高，也不过是 6%，并不算高。所以，以 I/O 问题为分析方向还是正确的。

知道了进程的 PID 号，具体要怎么查看写的情况呢？

其实，我在系统调用的案例中讲过，读写文件必须通过系统调用完成。观察系统调用情况，就可以知道进程正在写的文件。想起 strace 了吗，它正是我们分析系统调用时最常用的工具。

接下来，我们在终端中运行 strace 命令，并通过 -p 18940 指定 python 进程的 PID 号：

 复制代码


```
1 $ strace -p 18940
2 strace: Process 18940 attached
3 ...
4 mmap(NULL, 314576896, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f0f682e8000
5 mmap(NULL, 314576896, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f0f682e8000
6 write(3, "2018-12-05 15:23:01,709 - __main"... , 314572844
7 ) = 314572844
8 munmap(0x7f0f682e8000, 314576896)      = 0
9 write(3, "\n", 1)                          = 1
10 munmap(0x7f0f7aee9000, 314576896)     = 0
11 close(3)                             = 0
12 stat("/tmp/logtest.txt.1", {st_mode=S_IFREG|0644, st_size=943718535, ...}) = 0
```

从 `write()` 系统调用上，我们可以看到，进程向文件描述符编号为 3 的文件中，写入了 300MB 的数据。看来，它应该是我们要找的文件。不过，`write()` 调用中只能看到文件的描述符编号，文件名和路径还是未知的。

再观察后面的 `stat()` 调用，你可以看到，它正在获取 `/tmp/logtest.txt.1` 的状态。这种“点 + 数字格式”的文件，在日志回滚中非常常见。我们可以猜测，这是第一个日志回滚文件，而正在写的日志文件路径，则是 `/tmp/logtest.txt`。

当然，这只是我们的猜测，自然还需要验证。这里，我再给你介绍一个新的工具 `lsof`。它专门用来查看进程打开文件列表，不过，这里的“文件”不只有普通文件，还包括了目录、块设备、动态库、网络套接字等。

接下来，我们在终端中运行下面的 `lsof` 命令，看看进程 18940 都打开了哪些文件：

 复制代码

```
1 $ lsof -p 18940
2 COMMAND  PID USER  FD   TYPE DEVICE SIZE/OFF      NODE NAME
3 python   18940 root   cwd   DIR    0,50      4096 1549389 /
4 python   18940 root   rtd   DIR    0,50      4096 1549389 /
5 ...
6 python   18940 root    2u   CHR  136,0        0t0        3 /dev/pts/0
7 python   18940 root    3w   REG    8,1 117944320    303 /tmp/logtest.txt
```


这个输出界面中，有几列我简单介绍一下，`FD` 表示文件描述符号，`TYPE` 表示文件类型，`NAME` 表示文件路径。这也是我们需要关注的重点。

再看最后一行，这说明，这个进程打开了文件 `/tmp/logtest.txt`，并且它的文件描述符是 3 号，而 3 后面的 `w`，表示以写的方式打开。

这跟刚才 `strace` 完我们猜测的结果一致，看来这就是问题的根源：进程 18940 以每次 300MB 的速度，在“疯狂”写日志，而日志文件的路径是 `/tmp/logtest.txt`。

既然找出了问题根源，接下来按照惯例，就该查看源代码，然后分析为什么这个进程会狂打日志了。


你可以运行 `docker cp` 命令，把案例应用的源代码拷贝出来，然后查看它的内容。（你也可以点击[这里](#)查看案例应用的源码）：

 复制代码

```
1 # 拷贝案例应用源代码到当前目录
2 $ docker cp app:/app.py .
3
4 # 查看案例应用的源代码
5 $ cat app.py
6
7 logger = logging.getLogger(__name__)
8 logger.setLevel(level=logging.INFO)
9 rHandler = RotatingFileHandler("/tmp/logtest.txt", maxBytes=1024 * 1024 * 1024, backup(
10 rHandler.setLevel(logging.INFO)
11
12 def write_log(size):
13     '''Write logs to file'''
14     message = get_message(size)
15     while True:
16         logger.info(message)
17         time.sleep(0.1)
18
19 if __name__ == '__main__':
20     msg_size = 300 * 1024 * 1024
21     write_log(msg_size)
```

分析这个源码，我们发现，它的日志路径是 `/tmp/logtest.txt`，默认记录 INFO 级别以上的所有日志，而且每次写日志的大小是 300MB。这跟我们上面的分析结果是一致的。

一般来说，生产系统的应用程序，应该有动态调整日志级别的功能。继续查看源码，你会发现，这个程序也可以调整日志级别。如果你给它发送 SIGUSR1 信号，就可以把日志调整为 INFO 级；发送 SIGUSR2 信号，则会调整为 WARNING 级：

 复制代码


```
1 def set_logging_info(signal_num, frame):
2     '''Set logging level to INFO when receives SIGUSR1'''
3     logger.setLevel(logging.INFO)
4
5 def set_logging_warning(signal_num, frame):
6     '''Set logging level to WARNING when receives SIGUSR2'''
7     logger.setLevel(logging.WARNING)
8
9 signal.signal(signal.SIGUSR1, set_logging_info)
```



```
10 signal.signal(signal.SIGUSR2, set_logging_warning)
```


根据源码中的日志调用 `logger.info(message)`，我们知道，它的日志是 INFO 级，这也正是它的默认级别。那么，只要把默认级别调高到 WARNING 级，日志问题应该就解决了。

接下来，我们就来检查一下，刚刚的分析对不对。在终端中运行下面的 kill 命令，给进程 18940 发送 SIGUSR2 信号：


 复制代码

```
1 $ kill -SIGUSR2 18940
```

然后，再执行 top 和 iostat 观察一下：

 复制代码

```
1 $ top
2 ...
3 %Cpu(s):  0.3 us,  0.2 sy,  0.0 ni, 99.5 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
```

 复制代码

```
1 $ iostat -d -x 1
2 Device            r/s        w/s        kB/s        kB/s        rrqm/s        wrqm/s        %rrqm        %wrqm        r_await
3 loop0              0.00        0.00         0.00         0.00         0.00         0.00         0.00         0.00         0
4 sdb                 0.00        0.00         0.00         0.00         0.00         0.00         0.00         0.00         0
5 sda                 0.00        0.00         0.00         0.00         0.00         0.00         0.00         0.00         0
```

观察 top 和 iostat 的输出，你会发现，稍等一段时间后，iowait 会变成 0，而 sda 磁盘的 I/O 使用率也会逐渐减少到 0。

到这里，我们不仅定位了狂打日志的应用程序，并通过调高日志级别的方法，完美解决了 I/O 的性能瓶颈。

案例最后，当然不要忘了运行下面的命令，停止案例应用：

```
1 $ docker rm -f app
```

## 小结

日志，是了解应用程序内部运行情况，最常用、也最有效的工具。无论是操作系统，还是应用程序，都会记录大量的运行日志，以便事后查看历史记录。这些日志一般按照不同级别来开启，比如，开发环境通常打开调试级别的日志，而线上环境则只记录警告和错误日志。

在排查应用程序问题时，我们可能需要，在线上环境临时开启应用程序的调试日志。有时候，事后一不小心就忘了调回去。没把线上的日志调高到警告级别，可能会导致 CPU 使用率、磁盘 I/O 等一系列的性能问题，严重时，甚至会影响到同一台服务器上运行的其他应用程序。

今后，在碰到这种“狂打日志”的场景时，你可以用 `iostat`、`strace`、`lsof` 等工具来定位狂打日志的进程，找出相应的日志文件，再通过应用程序的接口，调整日志级别来解决问题。

如果应用程序不能动态调整日志级别，你可能还需要修改应用的配置，并重启应用让配置生效。

## 思考

最后，给你留一个思考题。

在今天的案例开始时，我们用 `top` 和 `iostat` 查看了系统资源的使用情况。除了 CPU 和磁盘 I/O 外，剩余内存也比较少，而内存主要被 Buffer/Cache 占用。

那么，今天的问题就是，这些内存到底是被 Buffer 还是 Cache 占用了呢？有没有什么方法来确认你的分析结果呢？

欢迎在留言区和我讨论，也欢迎把这篇文章分享给你的同事、朋友。我们一起在实战中演练，在交流中进步。



# Linux 性能优化实战

10 分钟帮你找到系统瓶颈

倪朋飞

微软资深工程师  
Kubernetes 项目维护者



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得转载

上一篇 25 | 基础篇：Linux 磁盘I/O是怎么工作的（下）

下一篇 27 | 案例篇：为什么我的磁盘I/O延迟很高？

## 精选留言

写留言



hua168

2019-01-19

7

大神，能问一个题外话吗，关于自己人生规划，水平和眼界所限，想不通，都说大神级见识很广也多，能给我这个35岁只维护过四五十台linux服务器的运维指条路吗？ ...

展开

作者回复: 看前面的经历，技术面还是挺广的，但可能很多地方都不太深入。

建议还是先找份工作吧，长期待业可能会加重焦虑。然后可以根据实际工作需要，先把工作需要的知识技能加深掌握。等到可以从容应对工作的时候，再考虑在某个领域加强深入。

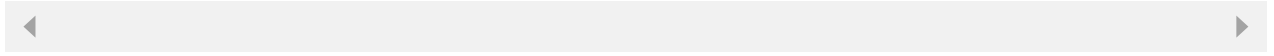
**Christmas**

2019-01-18

👍 5

pcstat(page cache stat)这个可以查看目标log文件在cache中的大小

作者回复: 嗯嗯 是的，并且这个工具需要知道哪个文件

**郭江伟**

2019-01-18

👍 3

buffers/cached使用情况可以从proc文件系统看：

```
gjw@gjw:~$ cat /proc/meminfo
```

```
MemTotal: 7588504 kB...
```

展开 ▾

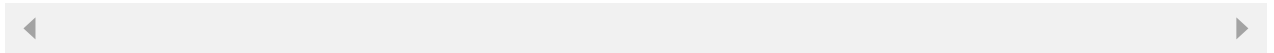
**J**

2019-01-20

👍 1

logger.info(message)的情况下，还可以使用logger.setLevel修改日志级别吗？

作者回复: 可以的，比如调高到警告级别，那么 info() 实际上就不写了

**我来也**

2019-01-18

👍 1

[D26打卡]

又是老套路了，哈哈。

先是top看%iowait到升高，再看pidstat是哪个进程在操作磁盘，再strace看进程的调...

展开 ▾

**Geek\_41dcba**

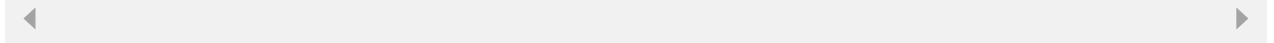
2019-01-18

👍 1

在回答今天的思考题前，我想需要明确两个前提，一个是Buffer到底在整个系统结构的哪一层，会不会是不是在IO调度器的下一层，我想应该会，理由是Buffer缓存磁盘内容调度器合并后再去写磁盘效率更好；另一个是之前有看到留言对于文件系统使用带Cache的IO...

展开 ▾

作者回复: 嗯 可以实际操作验证一下



**划时代**

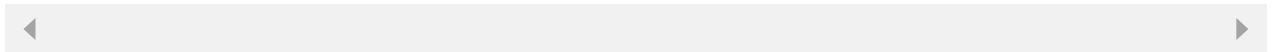
2019-01-18

👍 1

指出老师的一个问题，“日志回滚文件”，打印日志的过程中从直觉来看很容易误认为日志是在“回滚”，我也犯过这样的错误；rotating英文直译为“旋转”或“轮流”，实际的日志打印过程中，日志名称是“旋转”的，例如log.1(当前打印的日志文件并且一直...

展开 ▾

作者回复: 谢谢指出，确实只是文件名称的旋转，内容并没有回滚



**wtcctw**

2019-01-18

👍 1

逻辑清晰，步骤详细，赞



**无名老卒**

2019-01-22

👍 0

查看buffer/cache占用，建议使用pcstat或者hcache，hcache是基于pcstat的，pcstat可以查看某个文件是否被缓存和根据进程pid来查看都缓存了哪些文件。hcache在其基础上增加了查看整个操作系统Cache和根据使用Cache大小排序的特性。...

展开 ▾



**xfan**

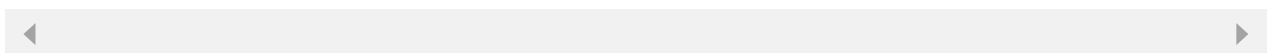
2019-01-21

👍 0

我开启了交换swap，和实验不同的是我的Buffer/cache占用不高，反倒是swap很高。还有一个是基本没有写，读很高。奇怪

展开 ▾

作者回复: 检查下swapiness的设置



**渡渡鸟\_linux**

2019-01-21

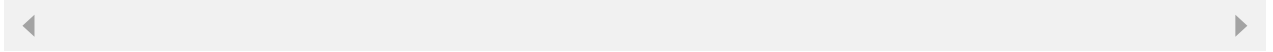
👍 0

我这边使用centos7 2c 8g 实验结果与文章中有些区别：

1. 在运行容器后，使用top命令发现sys与iowait各占单个CPU的10-50%
2. 针对iowait，使用dstat发现磁盘每秒写入约为300M; vmstat中bo也验证了写请求较...

展开 ▾

作者回复: sys使用高是正常的，任何IO都要经过系统调用完成，自然要占用sys。

**小老鼠**

2019-01-20

👍 0

我觉的应该是cache，写日志，日志是文件

**仲鬼**

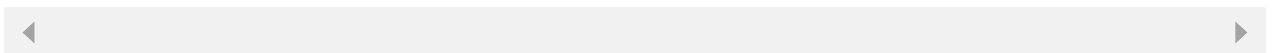
2019-01-18

👍 0

“每秒写的数据超过 45 MB，比上面 iostat 发现的 32MB 的结果还要大”  
老师好，没明白这里的比较要说明什么问题？

展开 ▾

作者回复: 这儿是要说 IO 的来源，肯定是 I/O 大的进程

**安小依**

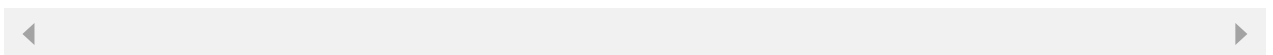
2019-01-18

👍 0

Ubuntu 16.04, 使用 strace 应该需要先临时修改系统一个配置：echo 0 | sudo tee /proc/sys/kernel/yama/ptrace\_scope。另外用 java 模拟写文件，strace 发现还是不行，一直卡在这个地方： ...

展开 ▾

作者回复: 你这个例子是向文件 /tmp/aaa 写，不是标准输出







**唯美** 01-18

👍<sup>0</sup>

打卡Day26  
day day up!

展开 ▾

---



**ninuxer**

2019-01-18

👍 0

打卡day27

应该是被cache占用，因为内存篇提到cache主要是负责文件的读写缓存，buffer是负责块设备读写缓存，而案例中写的是文件...

展开 ▾