

17 | 案例篇：如何利用系统缓存优化程序的运行效率？

2018-12-28 倪朋飞

Linux性能优化实战

[进入课程 >](#)



讲述：冯永吉

时长 15:30 大小 14.20M



你好，我是倪朋飞。

上一节，我们学习了内存性能中 Buffer 和 Cache 的概念。简单复习一下，Buffer 和 Cache 的设计目的，是为了提升系统的 I/O 性能。它们利用内存，充当起慢速磁盘与快速 CPU 之间的桥梁，可以加速 I/O 的访问速度。

Buffer 和 Cache 分别缓存的是对磁盘和文件系统的读写数据。

从写的角度来说，不仅可以优化磁盘和文件的写入，对应用程序也有好处，应用程序可以在数据真正落盘前，就返回去做其他工作。

从读的角度来说，不仅可以提高那些频繁访问数据的读取速度，也降低了频繁 I/O 对磁盘的压力。

既然 Buffer 和 Cache 对系统性能有很大影响，那我们在软件开发的过程中，能不能利用这一点，来优化 I/O 性能，提升应用程序的运行效率呢？

答案自然是肯定的。今天，我就用几个案例帮助你更好地理解缓存的作用，并学习如何充分利用这些缓存来提高程序效率。

为了方便你理解，Buffer 和 Cache 我仍然用英文表示，避免跟“缓存”一词混淆。而文中的“缓存”，通指数据在内存中的临时存储。

缓存命中率

在案例开始前，你应该习惯性地先问自己一个问题，你想要做成某件事情，结果应该怎么评估？比如说，我们想利用缓存来提升程序的运行效率，应该怎么评估这个效果呢？换句话说，有没有哪个指标可以衡量缓存使用的好坏呢？

我估计你已经想到了，**缓存的命中率**。所谓缓存命中率，是指直接通过缓存获取数据的请求次数，占有所有数据请求次数的百分比。

命中率越高，表示使用缓存带来的收益越高，应用程序的性能也就越好。

实际上，缓存是现在所有高并发系统必需的核心模块，主要作用就是把经常访问的数据（也就是热点数据），提前读入到内存中。这样，下次访问时就可以直接从内存读取数据，而不需要经过硬盘，从而加快应用程序的响应速度。

这些独立的缓存模块通常会提供查询接口，方便我们随时查看缓存的命中情况。不过 Linux 系统中并没有直接提供这些接口，所以这里我要介绍一下，cachestat 和 cachetop，它们正是查看系统缓存命中情况的工具。


cachestat 提供了整个操作系统缓存的读写命中情况。

cachetop 提供了每个进程的缓存命中情况。

这两个工具都是 [bcc](#) 软件包的一部分，它们基于 Linux 内核的 eBPF (extended Berkeley Packet Filters) 机制，来跟踪内核中管理的缓存，并输出缓存的使用和命中情况。

这里注意，eBPF 的工作原理不是我们今天的重点，记住这个名字即可，后面文章中我们会详细学习。今天要掌握的重点，是这两个工具的使用方法。


使用 cachestat 和 cachetop 前，我们首先要安装 bcc 软件包。比如，在 Ubuntu 系统中，你可以运行下面的命令来安装：

 复制代码

```
1 sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 4052245BD4284CDD
2 echo "deb https://repo.iovisor.org/apt/xenial xenial main" | sudo tee /etc/apt/sources.list.d/bcc.list
3 sudo apt-get update
4 sudo apt-get install -y bcc-tools libbcc-examples linux-headers-$(uname -r)
```

注意：bcc-tools 需要内核版本为 4.1 或者更新的版本，如果你用的是 CentOS，那就需要手动[升级内核版本后再安装](#)。

操作完这些步骤，bcc 提供的所有工具就都安装到 /usr/share/bcc/tools 这个目录中了。不过这里提醒你，bcc 软件包默认不会把这些工具配置到系统的 PATH 路径中，所以你得自己手动配置：

 复制代码

```
1 $ export PATH=$PATH:/usr/share/bcc/tools
```

配置完，你就可以运行 cachestat 和 cachetop 命令了。比如，下面就是一个 cachestat 的运行界面，它以 1 秒的时间间隔，输出了 3 组缓存统计数据：

 复制代码

```
1 $ cachestat 1 3
2      TOTAL    MISSES      HITS  DIRTIES  BUFFERS_MB  CACHED_MB
3         2         0         2         1         17         279
4         2         0         2         1         17         279
5         2         0         2         1         17         279
```

你可以看到，cachestat 的输出其实是一个表格。每行代表一组数据，而每一列代表不同的缓存统计指标。这些指标从左到右依次表示：

TOTAL，表示总的 I/O 次数；

MISSES，表示缓存未命中的次数；


HITS，表示缓存命中的次数；

DIRTIES，表示新增到缓存中的脏页数；

BUFFERS_MB 表示 Buffers 的大小，以 MB 为单位；

CACHED_MB 表示 Cache 的大小，以 MB 为单位。

接下来我们再来看一个 cachetop 的运行界面：

 复制代码

```
1 $ cachetop
2 11:58:50 Buffers MB: 258 / Cached MB: 347 / Sort: HITS / Order: ascending
3 PID      UID      CMD           HITS      MISSES    DIRTIES  READ_HIT%  WRITE_HIT%
4    13029  root      python          1         0         0      100.0%     0.0%
```

它的输出跟 top 类似，默认按照缓存的命中次数（HITS）排序，展示了每个进程的缓存命中情况。具体到每一个指标，这里的 HITS、MISSES 和 DIRTIES，跟 cachestat 里的含义一样，分别代表间隔时间内的缓存命中次数、未命中次数以及新增到缓存中的脏页数。

而 READ_HIT 和 WRITE_HIT，分别表示读和写的缓存命中率。

指定文件的缓存大小

除了缓存的命中率外，还有一个指标你可能也会很感兴趣，那就是指定文件在内存中的缓存大小。你可以使用 [pcstat](#) 这个工具，来查看文件在内存中的缓存大小以及缓存比例。

pcstat 是一个基于 Go 语言开发的工具，所以安装它之前，你首先应该安装 Go 语言，你可以点击[这里](#)下载安装。

安装完 Go 语言，再运行下面的命令安装 pcstat：

```
1 $ export GOPATH=~/.go
2 $ export PATH=~/.go/bin:$PATH
3 $ go get golang.org/x/sys/unix
4 $ go get github.com/tobert/pcstat/pcstat
```

全部安装完成后，你就可以运行 pcstat 来查看文件的缓存情况了。比如，下面就是一个 pcstat 运行的示例，它展示了 /bin/ls 这个文件的缓存情况：

```
1 $ pcstat /bin/ls
2 +-----+-----+-----+-----+-----+
3 | Name      | Size (bytes) | Pages      | Cached      | Percent      |
4 |-----+-----+-----+-----+-----+
5 | /bin/ls   | 133792       | 33         | 0           | 000.000      |
6 +-----+-----+-----+-----+-----+
```

这个输出中，Cached 就是 /bin/ls 在缓存中的大小，而 Percent 则是缓存的百分比。你看到它们都是 0，这说明 /bin/ls 并不在缓存中。

接着，如果你执行一下 ls 命令，再运行相同的命令来查看的话，就会发现 /bin/ls 都在缓存中了：

```
1 $ ls
2 $ pcstat /bin/ls
3 +-----+-----+-----+-----+-----+
4 | Name      | Size (bytes) | Pages      | Cached      | Percent      |
5 |-----+-----+-----+-----+-----+
6 | /bin/ls   | 133792       | 33         | 33          | 100.000      |
7 +-----+-----+-----+-----+-----+
```

知道了缓存相应的指标和查看系统缓存的方法后，接下来，我们就进入今天的正式案例。

跟前面的案例一样，今天的案例也是基于 Ubuntu 18.04，当然同样适用于其他的 Linux 系统。

机器配置：2 CPU，8GB 内存。

预先按照上面的步骤安装 bcc 和 pcstat 软件包，并把这些工具的安装路径添加到 PATH 环境变量中。

预先安装 Docker 软件包，比如 apt-get install [docker.io](https://docs.docker.com/engine/install/ubuntu/)


案例一

第一个案例，我们先来看一下上一节提到的 dd 命令。

dd 作为一个磁盘和文件的拷贝工具，经常被拿来测试磁盘或者文件系统的读写性能。不过，既然缓存会影响到性能，如果用 dd 对同一个文件进行多次读取测试，测试的结果会怎么样呢？


我们来动手试试。首先，打开两个终端，连接到 Ubuntu 机器上，确保 bcc 已经安装配置成功。

然后，使用 dd 命令生成一个临时文件，用于后面的文件读取测试：

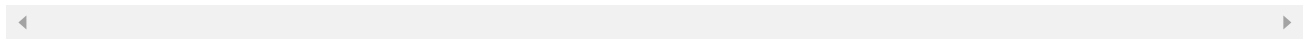
 复制代码

```
1 # 生成一个 512MB 的临时文件
2 $ dd if=/dev/sda1 of=file bs=1M count=512
3 # 清理缓存
4 $ echo 3 > /proc/sys/vm/drop_caches
```

继续在第一个终端，运行 pcstat 命令，确认刚刚生成的文件不在缓存中。如果一切正常，你会看到 Cached 和 Percent 都是 0:

 复制代码

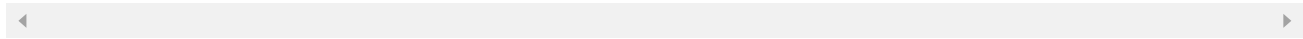
```
1 $ pcstat file
2 +-----+-----+-----+-----+-----+
3 | Name   | Size (bytes) | Pages   | Cached | Percent |
4 |-----+-----+-----+-----+-----+
5 | file   | 536870912    | 131072  | 0       | 000.000 |
```



还是在第一个终端中，现在运行 `cachetop` 命令：

复制代码

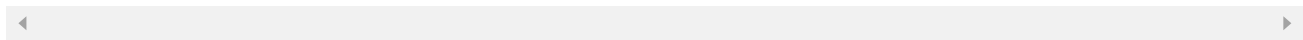
```
1 # 每隔 5 秒刷新一次数据
2 $ cachetop 5
```



这次是第二个终端，运行 `dd` 命令测试文件的读取速度：

复制代码

```
1 $ dd if=file of=/dev/null bs=1M
2 512+0 records in
3 512+0 records out
4 536870912 bytes (537 MB, 512 MiB) copied, 16.0509 s, 33.4 MB/s
```



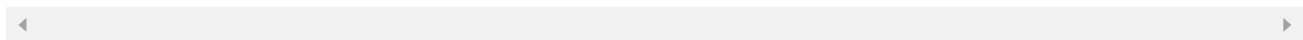
从 `dd` 的结果可以看出，这个文件的读性能是 33.4 MB/s。由于在 `dd` 命令运行前我们已经清理了缓存，所以 `dd` 命令读取数据时，肯定要通过文件系统从磁盘中读取。

不过，这是不是意味着，`dd` 所有的读请求都能直接发送到磁盘呢？

我们再回到第一个终端，查看 `cachetop` 界面的缓存命中情况：


复制代码

PID	UID	CMD	HITS	MISSES	DIRTIES	READ_HIT%	WRITE_HIT%	
\\.\.								
3	3264	root	dd	37077	37330	0	49.8%	50.2%



从 `cachetop` 的结果可以发现，并不是所有的读都落到了磁盘上，事实上读请求的缓存命中率只有 50%。

接下来，我们继续尝试相同的测试命令。先切换到第二个终端，再次执行刚才的 dd 命令：

 复制代码

```
1 $ dd if=file of=/dev/null bs=1M
2 512+0 records in
3 512+0 records out
4 536870912 bytes (537 MB, 512 MiB) copied, 0.118415 s, 4.5 GB/s
```

看到这次的结果，有没有点小惊讶？磁盘的读性能居然变成了 4.5 GB/s，比第一次的结果明显高了太多。为什么这次的结果这么好呢？


不妨再回到第一个终端，看看 cachetop 的情况：

 复制代码

```
1 10:45:22 Buffers MB: 4 / Cached MB: 719 / Sort: HITS / Order: ascending
2 PID      UID      CMD              HITS      MISSES    DIRTIES  READ_HIT%  WRITE_HIT%
3 \.\.\.
4 32642 root    dd                131637      0         0        100.0%     0.0%
```

显然，cachetop 也有了不小的变化。你可以发现，这次的读的缓存命中率是 100.0%，也就是说这次的 dd 命令全部命中了缓存，所以才会看到那么高的性能。

然后，回到第二个终端，再次执行 pcstat 查看文件 file 的缓存情况：

 复制代码

```
1 $ pcstat file
2 +-----+-----+-----+-----+
3 | Name   | Size (bytes) | Pages   | Cached   | Percent  |
4 |-----+-----+-----+-----+
5 | file   | 536870912    | 131072  | 131072   | 100.000  |
6 +-----+-----+-----+-----+
```

从 pcstat 的结果你可以发现，测试文件 file 已经被全部缓存了起来，这跟刚才观察到的缓存命中率 100% 是一致的。

这两次结果说明，系统缓存对第二次 dd 操作有明显的加速效果，可以大大提高文件读取的性能。

但同时也要注意，如果我们把 dd 当成测试文件系统性能的工具，由于缓存的存在，就会导致测试结果严重失真。

案例二


接下来，我们再来看一个文件读写的案例。这个案例类似于前面学过的不可中断状态进程的例子。它的基本功能比较简单，也就是每秒从磁盘分区 /dev/sda1 中读取 32MB 的数据，并打印出读取数据花费的时间。

为了方便你运行案例，我把它打包成了一个 [Docker 镜像](#)。跟前面案例类似，我提供了下面两个选项，你可以根据系统配置，自行调整磁盘分区的路径以及 I/O 的大小。

-d 选项，设置要读取的磁盘或分区路径，默认是查找前缀为 /dev/sd 或者 /dev/xvd 的磁盘。


-s 选项，设置每次读取的数据量大小，单位为字节，默认为 33554432（也就是 32MB）。

这个案例同样需要你开启两个终端。分别 SSH 登录到机器上后，先在第一个终端中运行 cachetop 命令：

 复制代码


```
1 # 每隔 5 秒刷新一次数据
2 $ cachetop 5
```

接着，再到第二个终端，执行下面的命令运行案例：

 复制代码

```
1 $ docker run --privileged --name=app -itd feisky/app:io-direct
```


案例运行后，我们还需要运行下面这个命令，来确认案例已经正常启动。如果一切正常，你应该可以看到类似下面的输出：

 复制代码

```
1 $ docker logs app
2 Reading data from disk /dev/sdb1 with buffer size 33554432
3 Time used: 0.929935 s to read 33554432 bytes
4 Time used: 0.949625 s to read 33554432 bytes
```

从这里你可以看到，每读取 32 MB 的数据，就需要花 0.9 秒。这个时间合理吗？我想你第一反应就是，太慢了吧。那这是不是没用系统缓存导致的呢？

我们再来检查一下。回到第一个终端，先看看 cachetop 的输出，在这里，我们找到案例进程 app 的缓存使用情况：

 复制代码

```
1 16:39:18 Buffers MB: 73 / Cached MB: 281 / Sort: HITS / Order: ascending
2 PID      UID      CMD           HITS      MISSES    DIRTIES  READ_HIT%  WRITE_HIT%
3    21881  root      app           1024         0         0        100.0%     0.0%
```

这个输出似乎有点意思了。1024 次缓存全部命中，读的命中率是 100%，看起来全部的读请求都经过了系统缓存。但是问题又来了，如果真的都是缓存 I/O，读取速度不应该这么慢。

不过，话说回来，我们似乎忽略了另一个重要因素，每秒实际读取的数据大小。HITS 代表缓存的命中次数，那么每次命中能读取多少数据呢？自然是一页。


前面讲过，内存以页为单位进行管理，而每个页的大小是 4KB。所以，在 5 秒的时间间隔里，命中的缓存为 $1024 \times 4K / 1024 = 4MB$ ，再除以 5 秒，可以得到每秒读的缓存是 0.8MB，显然跟案例应用的 32 MB/s 相差太多。

至于为什么只能看到 0.8 MB 的 HITS，我们后面再解释，这里你先知道怎么根据结果来分析就可以了。

这也进一步验证了我们的猜想，这个案例估计没有充分利用系统缓存。其实前面我们遇到过类似的问题，如果为系统调用设置直接 I/O 的标志，就可以绕过系统缓存。

那么，要判断应用程序是否用了直接 I/O，最简单的方法当然是观察它的系统调用，查找应用程序在调用它们时的选项。使用什么工具来观察系统调用呢？自然还是 strace。

继续在终端二中运行下面的 strace 命令，观察案例应用的系统调用情况。注意，这里使用了 pgrep 命令来查找案例进程的 PID 号：

 复制代码


```
1 # strace -p $(pgrep app)
2 strace: Process 4988 attached
3 restart_syscall(<...\.. resuming interrupted nanosleep \..>) = 0
4 openat(AT_FDCWD, "/dev/sdb1", O_RDONLY|O_DIRECT) = 4
5 mmap(NULL, 33558528, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f448d:
6 read(4, "8vq\213\314\264u\373\4\336K\224\25@\371\1\252\2\262\252q\221\n0\30\225bD\252\26:
7 write(1, "Time used: 0.948897 s to read 33"...\.., 45) = 45
8 close(4)                                = 0
```

从 strace 的结果可以看到，案例应用调用了 openat 来打开磁盘分区 /dev/sdb1，并且传入的参数为 O_RDONLY|O_DIRECT（中间的竖线表示或）。

O_RDONLY 表示以只读方式打开，而 O_DIRECT 则表示以直接读取的方式打开，这会绕过系统的缓存。

验证了这一点，就很容易理解为什么读 32 MB 的数据就都要那么久了。直接从磁盘读写速度，自然远慢于对缓存的读写。这也是缓存存在的最大意义了。

找出问题后，我们还可以在再看看案例应用的[源代码](#)，再次验证一下：


 复制代码

```
1 int flags = O_RDONLY | O_LARGEFILE | O_DIRECT;
2 int fd = open(disk, flags, 0755);
```

上面的代码，很清楚地告诉我们：它果然用了直接 I/O。


找出了磁盘读取缓慢的原因，优化磁盘读的性能自然不在话下。修改源代码，删除 O_DIRECT 选项，让应用程序使用缓存 I/O，而不是直接 I/O，就可以加速磁盘读取速度。

[app-cached.c](#) 就是修复后的源码，我也把它打包成了一个容器镜像。在第二个终端中，按 Ctrl+C 停止刚才的 strace 命令，运行下面的命令，你就可以启动它：

 复制代码

```
1 # 删除上述案例应用
2 $ docker rm -f app
3
4 # 运行修复后的应用
5 $ docker run --privileged --name=app -itd feisky/app:io-cached
```


还是第二个终端，再来运行下面的命令查看新应用的日志，你应该能看到下面这个输出：

 复制代码

```
1 $ docker logs app
2 Reading data from disk /dev/sdb1 with buffer size 33554432
3 Time used: 0.037342 s s to read 33554432 bytes
4 Time used: 0.029676 s to read 33554432 bytes
```

现在，每次只需要 0.03 秒，就可以读取 32MB 数据，明显比之前的 0.9 秒快多了。所以，这次应该用了系统缓存。

我们再回到第一个终端，查看 cachetop 的输出来确认一下：

 复制代码

```
1 16:40:08 Buffers MB: 73 / Cached MB: 281 / Sort: HITS / Order: ascending
2 PID      UID      CMD           HITS      MISSES    DIRTIES  READ_HIT%  WRITE_HIT%
3    22106  root     app           40960         0         0       100.0%     0.0%
```

果然，读的命中率还是 100%，HITS（即命中数）却变成了 40960，同样的方法计算一下，换算成每秒字节数正好是 32 MB（即 $40960 \times 4k / 5 / 1024 = 32M$ ）。

这个案例说明，在进行 I/O 操作时，充分利用系统缓存可以极大地提升性能。但在观察缓存命中率时，还要注意结合应用程序实际的 I/O 大小，综合分析缓存的使用情况。

案例的最后，再回到开始的问题，为什么优化前，通过 cachetop 只能看到很少一部分数据的全部命中，而没有观察到大量数据的未命中情况呢？这是因为，cachetop 工具并不把直接 I/O 算进来。这也又一次说明了，了解工具原理的重要。

cachetop 的计算方法涉及到 I/O 的原理以及一些内核的知识，如果你想了解它的原理的话，可以点击[这里](#)查看它的源代码。

总结

Buffers 和 Cache 可以极大提升系统的 I/O 性能。通常，我们用缓存命中率，来衡量缓存的使用效率。命中率越高，表示缓存被利用得越充分，应用程序的性能也就越好。

你可以用 cachestat 和 cachetop 这两个工具，观察系统和进程的缓存命中情况。其中，

cachestat 提供了整个系统缓存的读写命中情况。

cachetop 提供了每个进程的缓存命中情况。

不过要注意，Buffers 和 Cache 都是操作系统来管理的，应用程序并不能直接控制这些缓存的内容和生命周期。所以，在应用程序开发中，一般要用专门的缓存组件，来进一步提升性能。

比如，程序内部可以使用堆或者栈明确声明内存空间，来存储需要缓存的数据。又或者，使用 Redis 这类外部缓存服务，优化数据的访问效率。

思考

最后，我想给你留下一道思考题，帮你更进一步了解缓存的原理。

今天的第二个案例你应该很眼熟，因为前面不可中断进程的文章用的也是直接 I/O 的例子，不过那次，我们是从 CPU 使用率和进程状态的角度来分析的。对比 CPU 和缓存这两个不同角度的分析思路，你有什么样的发现呢？

欢迎在留言区和我讨论，写下你的答案和收获，也欢迎你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 16 | 基础篇：怎么理解内存中的Buffer和Cache？

下一篇 18 | 案例篇：内存泄漏了，我该如何定位和处理？

精选留言 (60)

写留言



我来也

2018-12-28

22

[D17打卡]

想不到Buffer 和 Cache还有专门的工具分析, 长见识了!

暂时只能在自己的机器上玩玩, 生产环境连root权限都没有,更别提升级CentOS内核版本了.

关于思考题,我是这样想的:...

展开

作者回复: 总结的很好，其实两个思路都可以，不过具体实践时可能会受限于可用的性能工具



白华

2018-12-29

👍 11

centos7系统安装bcc-tools的教程我写在了简书上：

<https://www.jianshu.com/p/997e0a6d8e09>大家如果有安装不下来的可以看看

展开 ▾



noma

2019-01-02

👍 10

pcstat 不使用代理安装：

```
if [ $(uname -m) == "x86_64" ]; then
```

```
    curl -L -o pcstat https://github.com/tobert/pcstat/raw/2014-05-02-
```

```
01/pcstat.x86_64
```

```
else...
```

展开 ▾



Tech

2018-12-31

👍 8

有个疑问，既然app那个案例是直接i/o，那为什么还是有缓存了4MB呢？



Johnson

2018-12-28

👍 6

dd命令也支持直接IO的 有选项oflag和iflag 所以dd也可以用来绕过cache buff做测试

作者回复: 对的



念你如昔

2019-01-04

👍 4

(我是小白)

1.dd命令前清理了缓存，为什么缓存命中率是百分之49.8呢？（都已经没有缓存了，按理

说是0)

2.绕过了系统缓存，为什么缓存命中率还百分之百？

作者回复: 1. 预读，2. 还有元数据缓存



渡渡鸟_lin...

2019-01-01

👍 4

补充下centos7使用yum 安装bcc-tools:

```
[root@centos-80 ~]# yum update
```

```
[root@centos-80 ~]# rpm --import https://www.elrepo.org/RPM-GPG-KEY-elrepo.org && rpm -Uvh http://www.elrepo.org/elrepo-release-7.0-2.el7.elrepo.noarch.rpm...
```

展开 ▾



往事随风, ...

2018-12-28

👍 4

要是centos验证一下就好了，不同系统很多问题不一样，操作上遇到问题很奇怪

作者回复: 大部分案例我都在centos7验证了，不过文章中有些地方没有列出来详细的步骤，比如安装或者升级软件包的步骤，这些其实都是些基本功了。如果碰到实在无法解决的问题，请具体描述下。



xfan

2019-01-11

👍 3

我没有遇到老师的情况，我的hits为40960而不是1024，刚好是32M，我可能是固态？



jssfy

2019-01-04

👍 3

请问第一次读50%的命中率如何解释呢？

展开 ▾

作者回复: 预读



苹果xixi
2019-01-01

👍 3

1024*4K/1024=4m这是怎么算的



mj4ever
2018-12-31

👍 3

老师：

1、不知道是不是固态硬盘的原因，调整了参数至320MB

```
docker run --privileged --name=app -itd feisky/app:io-direct /app -d /dev/sdb -s 335544320
```

2、用命令观察，cachestat 5，可以达到409600， $409600 * 4 / 1024 = 320\text{MB}$...

展开 ▾



许山山
2018-12-28

👍 3

我也觉得这门课超级棒了，原理加时间，学到很多！

展开 ▾

作者回复: 谢谢 ☺



春暖花开
2018-12-28

👍 3

非常有价值，是我阅读的极客课程里面最棒的。

展开 ▾

作者回复: ☺ 谢谢



兔子先生
2019-02-17

👍 2

这里的direct i/o是不是上一节课里的直接操作磁盘的“裸i/o”呢？如果是的话是不是应该从buffer的角度分析实验二呢？

展开 ∨

作者回复: 不是的，直接IO是跳过Buffer，裸IO是跳过文件系统（还是有buffer的）



2xshu

2018-12-29

👍 2

老师你好，第一个案例我有不太明白的地方。希望能得到老师的指教。
既然执行了 `echo 3 > /proc/sys/vm/drop_caches`，为什么在 `dd if=file of=/dev/null bs=1M` 的时候，还有缓存能命中呢？我得理解是这些数据应该都没有在缓存啊。

展开 ∨



末班车

2018-12-28

👍 2

老师太厉害了，这个课程的价值远远高于这个价！！

展开 ∨



ninuxer

2018-12-28

👍 2

打卡day18

年终忙着各种总结，各种文字性东西，已经连着一礼拜12点之后下班了，再忙也要坚持跟着实践~ 🍻🍻🍻

展开 ∨

作者回复: 🍻加油



小松松

2019-04-13

👍 1

```
23:16:27 Buffers MB: 14 / Cached MB: 284 / Sort: HITS / Order: ascending
PID UID CMD HITS MISSES DIRTIES READ_HIT% WRITE_HIT%
2049 root dockerd 3 0 1 66.7% 0.0%
2064 root dockerd 3 0 1 66.7% 0.0%
```

422 root jbd2/sda2-8 3 3 1 33.3% 33.3%...

展开 ▾



1864633311...

2019-02-11

👍 1

```
[root@bogon ~]# cat /etc/redhat-release  
CentOS Linux release 7.3.1611 (Core)
```

```
[root@bogon ~]# docker ps...
```

展开 ▾