

## 06 | 案例篇：系统的 CPU 使用率很高，但为啥却找不到高 CPU 的应用？

2018-12-03 倪朋飞



朗读：冯永吉

时长15:18 大小14.03M



你好，我是倪朋飞。

上一节我讲了 CPU 使用率是什么，并通过一个案例教你使用 top、vmstat、pidstat 等工具，排查高 CPU 使用率的进程，然后再使用 perf top 工具，定位应用内部函数的问题。不过就有人留言了，说似乎感觉高 CPU 使用率的问题，还是挺容易排查的。

那是不是所有 CPU 使用率高的问题，都可以这么分析呢？我想，你的答案应该是否定的。

回顾前面的内容，我们知道，系统的 CPU 使用率，不仅包括进程用户态和内核态的运行，还包括中断处理、等待 I/O 以及内核线程等。所以，**当你发现系统的 CPU 使用率很高的时候，不一定能找到相对应的高 CPU 使用率的进程。**

今天，我就用一个 Nginx + PHP 的 Web 服务的案例，带你来分析这种情况。

# 案例分析

## 你的准备

今天依旧探究系统 CPU 使用率高的情况，所以这次实验的准备工作，与上节课的准备工作基本相同，差别在于案例所用的 Docker 镜像不同。

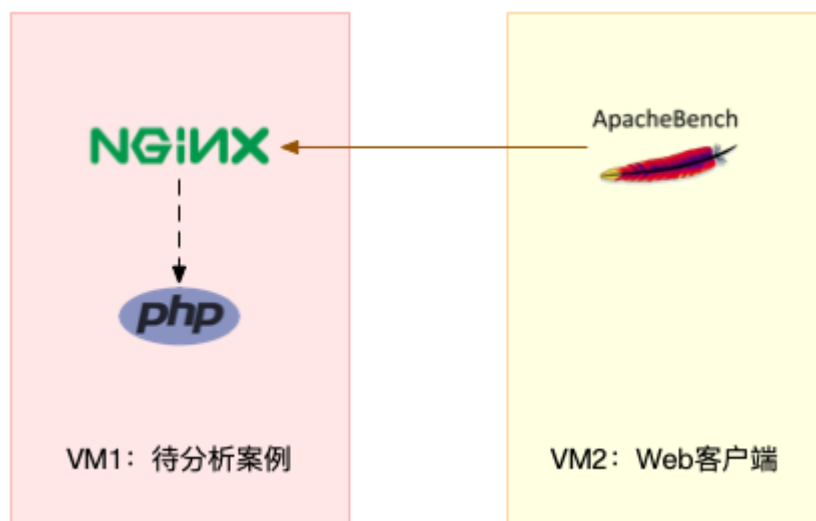
本次案例还是基于 Ubuntu 18.04，同样适用于其他的 Linux 系统。我使用的案例环境如下所示：

机器配置：2 CPU，8GB 内存

预先安装 docker、sysstat、perf、ab 等工具，如 `apt install docker.io sysstat linux-tools-common apache2-utils`

前面我们讲到过，ab (apache bench) 是一个常用的 HTTP 服务性能测试工具，这里同样用来模拟 Nginx 的客户端。由于 Nginx 和 PHP 的配置比较麻烦，我把它们打包成了两个 [Docker 镜像](#)，这样只需要运行两个容器，就可以得到模拟环境。

注意，这个案例要用到两台虚拟机，如下图所示：



你可以看到，其中一台用作 Web 服务器，来模拟性能问题；另一台用作 Web 服务器的客户端，来给 Web 服务增加压力请求。使用两台虚拟机是为了相互隔离，避免“交叉感染”。

接下来，我们打开两个终端，分别 SSH 登录到两台机器上，并安装上述工具。


同样注意，下面所有命令都默认以 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。

走到这一步，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 PHP 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，而要把它当成一个黑盒来分析。这样，你可以更好把握，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。


## 操作和分析

首先，我们在第一个终端，执行下面的命令运行 Nginx 和 PHP 应用：

 复制代码


```
1 $ docker run --name nginx -p 10000:80 -itd feisky/nginx:sp
2 $ docker run --name php-fpm -itd --network container:nginx feisky/php-fpm:sp
```

然后，在第二个终端，使用 curl 访问 `http://[VM1 的 IP]:10000`，确认 Nginx 已正常启动。你应该可以看到 It works! 的响应。

 复制代码

```
1 # 192.168.0.10 是第一台虚拟机的 IP 地址
2 $ curl http://192.168.0.10:10000/
3 It works!
```

接着，我们来测试一下这个 Nginx 服务的性能。在第二个终端运行下面的 ab 命令。要注意，与上次操作不同的是，这次我们需要并发 100 个请求测试 Nginx 性能，总共测试 1000 个请求。


 复制代码

```
1 # 并发 100 个请求测试 Nginx 性能，总共测试 1000 个请求
2 $ ab -c 100 -n 1000 http://192.168.0.10:10000/
3 This is ApacheBench, Version 2.3 <$Revision: 1706008 $>
4 Copyright 1996 Adam Twiss, Zeus Technology Ltd,
5 ...
6 Requests per second:      87.86 [#/sec] (mean)
7 Time per request:        1138.229 [ms] (mean)
8 ...
```

从 ab 的输出结果我们可以看到，Nginx 能承受的每秒平均请求数，只有 87 多一点，是不是感觉它的性能有点差呀。那么，到底是哪里出了问题呢？我们再用 top 和 pidstat 来观察一下。


这次，我们在第二个终端，将测试的并发请求数改成 5，同时把请求时长设置为 10 分钟（-t 600）。这样，当你在第一个终端使用性能分析工具时，Nginx 的压力还是继续的。

继续在第二个终端运行 ab 命令：

 复制代码

```
1 $ ab -c 5 -t 600 http://192.168.0.10:10000/
```

然后，我们在第一个终端运行 top 命令，观察系统的 CPU 使用情况：

 复制代码

```
1 $ top
2 ...
3 %Cpu(s): 80.8 us, 15.1 sy,  0.0 ni,  2.8 id,  0.0 wa,  0.0 hi,  1.3 si,  0.0 st
4 ...
5
6      PID USER      PR  NI   VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
7    6882 root        20   0   8456   5052   3884 S   2.7   0.1   0:04.78 docker-containe
8    6947 systemd+  20   0  33104   3716   2340 S   2.7   0.0   0:04.92 nginx
9    7494 daemon    20   0 336696  15012   7332 S   2.0   0.2   0:03.55 php-fpm
10   7495 daemon    20   0 336696  15160   7480 S   2.0   0.2   0:03.55 php-fpm
11  10547 daemon    20   0 336696  16200   8520 S   2.0   0.2   0:03.13 php-fpm
12  10155 daemon    20   0 336696  16200   8520 S   1.7   0.2   0:03.12 php-fpm
13  10552 daemon    20   0 336696  16200   8520 S   1.7   0.2   0:03.12 php-fpm
14  15006 root       20   0 1168608 66264  37536 S   1.0   0.8   9:39.51 dockerd
15   4323 root       20   0      0      0      0 I   0.3   0.0   0:00.87 kworker/u4:1
16 ...
```

观察 top 输出的进程列表可以发现，CPU 使用率最高的进程也只不过才 2.7%，看起来并不高。

然而，再看系统 CPU 使用率（%Cpu）这一行，你会发现，系统的整体 CPU 使用率是比较高的：用户 CPU 使用率（us）已经到了 80%，系统 CPU 为 15.1%，而空闲 CPU（id）则只有 2.8%。

为什么用户 CPU 使用率这么高呢？我们再重新分析一下进程列表，看看有没有可疑进程：


docker-containerd 进程是用来运行容器的，2.7% 的 CPU 使用率看起来正常；

Nginx 和 php-fpm 是运行 Web 服务的，它们会占用一些 CPU 也不意外，并且 2% 的 CPU 使用率也不算高；

再往下看，后面的进程呢，只有 0.3% 的 CPU 使用率，看起来不太像会导致用户 CPU 使用率达到 80%。

那就奇怪了，明明用户 CPU 使用率都 80% 了，可我们挨个分析了一遍进程列表，还是找不到高 CPU 使用率的进程。看来 top 是不管用了，那还有其他工具可以查看进程 CPU 使用情况吗？不知道你记不记得我们的老朋友 pidstat，它可以用来分析进程的 CPU 使用情况。

接下来，我们还是在第一个终端，运行 pidstat 命令：

 复制代码


```
1 # 间隔 1 秒输出一组数据（按 Ctrl+C 结束）
2 $ pidstat 1
3 ...
4 04:36:24      UID      PID    %usr %system  %guest  %wait   %CPU   CPU   Command
5 04:36:25        0    6882    1.00   3.00   0.00   0.00   4.00    0  docker-containe
6 04:36:25     101    6947    1.00   2.00   0.00   1.00   3.00    1   nginx
7 04:36:25        1   14834    1.00   1.00   0.00   1.00   2.00    0  php-fpm
8 04:36:25        1   14835    1.00   1.00   0.00   1.00   2.00    0  php-fpm
9 04:36:25        1   14845    0.00   2.00   0.00   2.00   2.00    1  php-fpm
10 04:36:25       1    14855    0.00   1.00   0.00   1.00   1.00    1  php-fpm
11 04:36:25       1    14857    1.00   2.00   0.00   1.00   3.00    0  php-fpm
12 04:36:25        0   15006    0.00   1.00   0.00   0.00   1.00    0  dockerd
13 04:36:25        0   15801    0.00   1.00   0.00   0.00   1.00    1  pidstat
14 04:36:25        1   17084    1.00   0.00   0.00   2.00   1.00    0  stress
15 04:36:25        0   31116    0.00   1.00   0.00   0.00   1.00    0  atopacctd
16 ...
```

观察一会儿，你是不是发现，所有进程的 CPU 使用率也都不高啊，最高的 Docker 和 Nginx 也只有 4% 和 3%，即使所有进程的 CPU 使用率都加起来，也不过是 21%，离 80% 还差得远呢！

最早的时候，我碰到这种问题就完全懵了：明明用户 CPU 使用率已经高达 80%，但我却怎么都找不到是哪个进程的问题。到这里，你也可以想想，你是不是也遇到过这种情况？还能不能再做进一步的分析呢？

后来我发现，会出现这种情况，很可能是因为前面的分析漏了一些关键信息。你可以先暂停一下，自己往上翻，重新操作检查一遍。或者，我们一起返回去分析 top 的输出，看看能不能有新发现。

现在，我们回到第一个终端，重新运行 top 命令，并观察一会儿：

 复制代码

```
1 $ top
2 top - 04:58:24 up 14 days, 15:47, 1 user, load average: 3.39, 3.82, 2.74
3 Tasks: 149 total, 6 running, 93 sleeping, 0 stopped, 0 zombie
4 %Cpu(s): 77.7 us, 19.3 sy, 0.0 ni, 2.0 id, 0.0 wa, 0.0 hi, 1.0 si, 0.0 st
5 KiB Mem : 8169348 total, 2543916 free, 457976 used, 5167456 buff/cache
6 KiB Swap: 0 total, 0 free, 0 used. 7363908 avail Mem
7
8  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM    TIME+  COMMAND
9  6947 systemd+ 20   0   33104    3764   2340 S   4.0   0.0   0:32.69 nginx
10  6882 root      20   0   12108    8360   3884 S   2.0   0.1   0:31.40 docker-containe
11 15465 daemon   20   0  336696   15256   7576 S   2.0   0.2   0:00.62 php-fpm
12 15466 daemon   20   0  336696   15196   7516 S   2.0   0.2   0:00.62 php-fpm
13 15489 daemon   20   0  336696   16200   8520 S   2.0   0.2   0:00.62 php-fpm
14  6948 systemd+ 20   0   33104    3764   2340 S   1.0   0.0   0:00.95 nginx
15 15006 root      20   0 1168608   65632  37536 S   1.0   0.8   9:51.09 dockerd
16 15476 daemon   20   0  336696   16200   8520 S   1.0   0.2   0:00.61 php-fpm
17 15477 daemon   20   0  336696   16200   8520 S   1.0   0.2   0:00.61 php-fpm
18 24340 daemon   20   0    8184    1616    536 R   1.0   0.0   0:00.01 stress
19 24342 daemon   20   0    8196    1580    492 R   1.0   0.0   0:00.01 stress
20 24344 daemon   20   0    8188    1056    492 R   1.0   0.0   0:00.01 stress
21 24347 daemon   20   0    8184    1356    540 R   1.0   0.0   0:00.01 stress
22 ...
```


这次从头开始看 top 的每行输出，咦？Tasks 这一行看起来有点奇怪，就绪队列中居然有 6 个 Running 状态的进程（6 running），是不是有点多呢？

回想一下 ab 测试的参数，并发请求数是 5。再看进程列表里，php-fpm 的数量也是 5，再加上 Nginx，好像同时有 6 个进程也并不奇怪。但真的是这样吗？

再仔细看进程列表，这次主要看 Running (R) 状态的进程。你有没有发现，Nginx 和所有的 php-fpm 都处于 Sleep (S) 状态，而真正处于 Running (R) 状态的，却是几个 stress 进程。这几个 stress 进程就比较奇怪了，需要我们做进一步的分析。


我们还是使用 pidstat 来分析这几个进程，并且使用 -p 选项指定进程的 PID。首先，从上面 top 的结果中，找到这几个进程的 PID。比如，先随便找一个 24344，然后用 pidstat 命令

看一下它的 CPU 使用情况：

 复制代码


```
1 $ pidstat -p 24344
2
3 16:14:55      UID      PID    %usr %system  %guest  %wait   %CPU   CPU  Command
```

奇怪，居然没有任何输出。难道是 pidstat 命令出问题了吗？之前我说过，**在怀疑性能工具出问题前，最好还是先用其他工具交叉确认一下**。那用什么工具呢？ps 应该是最简单易用的。我们在终端里运行下面的命令，看看 24344 进程的状态：

 复制代码

```
1 # 从所有进程中查找 PID 是 24344 的进程
2 $ ps aux | grep 24344
3 root      9628  0.0  0.0 14856 1096 pts/0    S+   16:15   0:00 grep --color=auto 24344
```

还是没有输出。现在终于发现问题，原来这个进程已经不存在了，所以 pidstat 就没有任何输出。既然进程都没了，那性能问题应该也跟着没了吧。我们再用 top 命令确认一下：

 复制代码

```
1 $ top
2 ...
3 %Cpu(s): 80.9 us, 14.9 sy,  0.0 ni,  2.8 id,  0.0 wa,  0.0 hi,  1.3 si,  0.0 st
4 ...
5
6  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM    TIME+  COMMAND
7  6882 root        20   0   12108    8360   3884 S   2.7   0.1   0:45.63 docker-containe
8  6947 systemd+  20   0   33104    3764   2340 R   2.7   0.0   0:47.79 nginx
9  3865 daemon    20   0  336696  15056   7376 S   2.0   0.2   0:00.15 php-fpm
10  6779 daemon     20   0    8184    1112    556 R   0.3   0.0   0:00.01 stress
11 ...
```

好像又错了。结果还跟原来一样，用户 CPU 使用率还是高达 80.9%，系统 CPU 接近 15%，而空闲 CPU 只有 2.8%，Running 状态的进程有 Nginx、stress 等。

可是，刚刚我们看到 stress 进程不存在了，怎么现在还在运行呢？再细看一下 top 的输出，原来，这次 stress 进程的 PID 跟前面不一样了，原来的 PID 24344 不见了，现在的是 6779。




进程的 PID 在变，这说明什么呢？在我看来，要么是这些进程在不停地重启，要么就是全新的进程，这无非也就两个原因：

第一个原因，进程在不停地崩溃重启，比如因为段错误、配置错误等等，这时，进程在退出后可能又被监控系统自动重启了。

第二个原因，这些进程都是短时进程，也就是在其他应用内部通过 `exec` 调用的外面命令。这些命令一般都只运行很短的时间就会结束，你很难用 `top` 这种间隔时间比较长的工具发现（上面的案例，我们碰巧发现了）。

至于 `stress`，我们前面提到过，它是一个常用的压力测试工具。它的 PID 在不断变化中，看起来像是被其他进程调用的短时进程。要想继续分析下去，还得找到它们的父进程。


要怎么查找一个进程的父进程呢？没错，用 `pstree` 就可以用树状形式显示所有进程之间的关系：

 复制代码

```
1 $ pstree | grep stress
2      | -docker-containe--php-fpm--php-fpm---sh---stress
3      |      | -3*[php-fpm---sh---stress---stress]
```

从这里可以看到，`stress` 是被 `php-fpm` 调用的子进程，并且进程数量不止一个（这里是 3 个）。找到父进程后，我们能进入 `app` 的内部分析了。

首先，当然应该去看看它的源码。运行下面的命令，把案例应用的源码拷贝到 `app` 目录，然后再执行 `grep` 查找是不是有代码再调用 `stress` 命令：


 复制代码

```
1 # 拷贝源码到本地
2 $ docker cp phpfpn:/app .
3
4 # grep 查找看看是不是有代码在调用 stress 命令
5 $ grep stress -r app
6 app/index.php:// fake I/O with stress (via write()/unlink()).
7 app/index.php:$result = exec("/usr/local/bin/stress -t 1 -d 1 2>&1", $output, $status);
```

找到了，果然是 `app/index.php` 文件中直接调用了 `stress` 命令。



再来看看 [app/index.php](#) 的源代码：


 复制代码

```
1 $ cat app/index.php
2 <?php
3 // fake I/O with stress (via write()/unlink()).
4 $result = exec("/usr/local/bin/stress -t 1 -d 1 2>&1", $output, $status);
5 if (isset($_GET["verbose"]) && $_GET["verbose"]==1 && $status != 0) {
6     echo "Server internal error: ";
7     print_r($output);
8 } else {
9     echo "It works!";
10 }
11 ?>
```

可以看到，源码里对每个请求都会调用一个 stress 命令，模拟 I/O 压力。从注释上看，stress 会通过 write() 和 unlink() 对 I/O 进程进行压测，看来，这应该就是系统 CPU 使用率升高的根源了。

不过，stress 模拟的是 I/O 压力，而之前在 top 的输出中看到的，却一直是用户 CPU 和系统 CPU 升高，并没见到 iowait 升高。这又是怎么回事呢？stress 到底是不是 CPU 使用率升高的原因呢？

我们还得继续往下走。从代码中可以看到，给请求加入 verbose=1 参数后，就可以查看 stress 的输出。你先试试看，在第二个终端运行：

 复制代码

```
1 $ curl http://192.168.0.10:10000?verbose=1
2 Server internal error: Array
3 (
4     [0] => stress: info: [19607] dispatching hogs: 0 cpu, 0 io, 0 vm, 1 hdd
5     [1] => stress: FAIL: [19608] (563) mkstemp failed: Permission denied
6     [2] => stress: FAIL: [19607] (394) <-- worker 19608 returned error 1
7     [3] => stress: WARN: [19607] (396) now reaping child worker processes
8     [4] => stress: FAIL: [19607] (400) kill error: No such process
9     [5] => stress: FAIL: [19607] (451) failed run completed in 0s
10 )
```

看错误消息 **mkstemp failed: Permission denied**，以及 **failed run completed in 0s**。原来 stress 命令并没有成功，它因为权限问题失败退出了。看来，我们发现了一个 PHP 调用外部 stress 命令的 bug：没有权限创建临时文件。

从这里我们可以猜测，正是由于权限错误，大量的 stress 进程在启动时初始化失败，进而导致用户 CPU 使用率的升高。

分析出问题来源，下一步是不是就要开始优化了呢？当然不是！既然只是猜测，那就需要再确认一下，这个猜测到底对不对，是不是真的有大量的 stress 进程。该用什么工具或指标呢？

我们前面已经用了 top、pidstat、pstree 等工具，没有发现大量的 stress 进程。那么，还有什么其他的工具可以用吗？

还记得上一期提到的 perf 吗？它可以用来分析 CPU 性能事件，用在这里就很合适。依旧在第一个终端中运行 perf record -g 命令，并等待一会儿（比如 15 秒）后按 Ctrl+C 退出。然后再运行 perf report 查看报告：

 复制代码

```
1 # 记录性能事件，等待大约 15 秒后按 Ctrl+C 退出
2 $ perf record -g
3
4 # 查看报告
5 $ perf report
```

这样，你就可以看到下图这个性能报告：


Samples: 137K of event 'cpu-clock', Event count (approx.): 34267500000					
Children	Self	Command	Shared Object	Symbol	
- 77.13%	0.00%	stress	stress	[.]	0x0000000000000168d
- 0x168d					
+ 25.97%		random_r			
+ 18.62%		random			
5.68%		rand			
+ 4.07%		0x2f25			
3.55%		0x2eff			
3.02%		0x2ee5			
2.26%		0xe80			
2.19%		0x2ef3			
2.16%		0x2f09			
1.87%		0x2f18			
1.56%		0x2f29			
1.36%		0x2f1e			
1.29%		0x2f1b			
1.22%		0x2f14			
0.97%		0x2f10			
0.71%		0x2f0d			
+ 25.97%	24.84%	stress	libc-2.24.so	[.]	random_r
+ 18.62%	17.86%	stress	libc-2.24.so	[.]	random
+ 5.68%	5.43%	stress	libc-2.24.so	[.]	rand
+ 5.66%	0.00%	swapper	[kernel.vmlinux]	[k]	0x000000000002000d5
+ 5.66%	0.00%	swapper	[kernel.vmlinux]	[k]	cpu_startup_entry
+ 5.66%	0.00%	swapper	[kernel.vmlinux]	[k]	do_idle
+ 5.60%	0.00%	swapper	[kernel.vmlinux]	[k]	default_idle_call
+ 5.60%	0.00%	swapper	[kernel.vmlinux]	[k]	arch_cpu_idle
+ 5.59%	0.00%	swapper	[kernel.vmlinux]	[k]	default_idle
+ 5.59%	5.42%	swapper	[kernel.vmlinux]	[k]	native_safe_halt
+ 4.37%	0.00%	php-fpm	[kernel.vmlinux]	[k]	entry_SYSCALL_64

你看，stress 占了所有 CPU 时钟事件的 77%，而 stress 调用调用栈中比例最高的，是随机数生成函数 random()，看来它的确就是 CPU 使用率升高的元凶了。随后的优化就很简单了，只要修复权限问题，并减少或删除 stress 的调用，就可以减轻系统的 CPU 压力。

当然，实际生产环境中的问题一般都要比这个案例复杂，在你找到触发瓶颈的命令行后，却可能发现，这个外部命令的调用过程是应用核心逻辑的一部分，并不能轻易减少或者删除。

这时，你就得继续排查，为什么被调用的命令，会导致 CPU 使用率升高或 I/O 升高等问题。这些复杂场景的案例，我会在后面的综合实战里详细分析。

最后，在案例结束时，不要忘了清理环境，执行下面的 Docker 命令，停止案例中用到的 Nginx 进程：

 复制代码


```
1 $ docker rm -f nginx php-fpm
```

## execsnoop

在这个案例中，我们使用了 top、pidstat、pstree 等工具分析了系统 CPU 使用率高的问题，并发现 CPU 升高是短时进程 stress 导致的，但是整个分析过程还是比较复杂的。对于这类问题，有没有更好的方法监控呢？

[execsnoop](#) 就是一个专为短时进程设计的工具。它通过 ftrace 实时监控进程的 exec() 行为，并输出短时进程的基本信息，包括进程 PID、父进程 PID、命令行参数以及执行的结果。

比如，用 execsnoop 监控上述案例，就可以直接得到 stress 进程的父进程 PID 以及它的命令行参数，并可以发现大量的 stress 进程在不停启动：

 复制代码

```
1 # 按 Ctrl+C 结束
2 $ execsnoop
3 PCOMM          PID      PPID    RET  ARGS
4 sh              30394   30393    0
5 stress         30396   30394    0 /usr/local/bin/stress -t 1 -d 1
6 sh              30398   30393    0
7 stress         30399   30398    0 /usr/local/bin/stress -t 1 -d 1
8 sh              30402   30400    0
9 stress         30403   30402    0 /usr/local/bin/stress -t 1 -d 1
10 sh             30405   30393    0
11 stress         30407   30405    0 /usr/local/bin/stress -t 1 -d 1
12 ...
```

execsnoop 所用的 ftrace 是一种常用的动态追踪技术，一般用于分析 Linux 内核的运行时行为，后面课程我也会详细介绍并带你使用。

## 小结

碰到常规问题无法解释的 CPU 使用率情况时，首先要想到有可能是短时应用导致的问题，比如有可能是下面这两种情况。

第一，应用里直接调用了其他二进制程序，这些程序通常运行时间比较短，通过 top 等工具也不容易发现。

第二，应用本身在不停地崩溃重启，而启动过程的资源初始化，很可能会占用相当多的 CPU。

对于这类进程，我们可以用 pstree 或者 execsnoop 找到它们的父进程，再从父进程所在的应用入手，排查问题的根源。

## 思考

最后，我想邀请你一起来聊聊，你所碰到的 CPU 性能问题。有没有哪个印象深刻的经历可以跟我分享呢？或者，在今天的案例操作中，你遇到了什么问题，又解决了哪些呢？你可以结合我的讲述，总结自己的思路。

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 05 | 基础篇：某个应用的CPU使用率居然达到100%，我该怎么办？

下一篇 07 | 案例篇：系统中出现大量不可中断进程和僵尸进程怎么办？（上）

## 精选留言 (68)

写留言



sotey

2018-12-03

36

对老师膜拜！今天一早生产tomcat夯住了，16颗cpu全部98%以上，使用老师的方法加上java的工具成功定位到了问题线程和问题函数。

作者回复: 举一反三动手实践的楷模😊 希望看到更多的人把这些思路用起来



**western**

2018-12-06

👍 5

感觉看侦探小说，遍看遍等老师说那句话——“真相只有一个”



**好好学习**

2018-12-10

👍 4

perf record -ag -- sleep 2;perf report  
一部到位

作者回复: 👍



**每天晒白牙**

2018-12-05

👍 3

【D6补卡】

如果碰到不好解释的CPU问题时，比如现象：

通过top观察CPU使用率很高，但是看下面的进程的CPU使用率好像很正常，通过pidstat命令查看cpu也很正常。但通过top查看task数量不正常，处于R状态的进程是可疑点。

...

展开 ▼



**looperX**

2018-12-04

👍 3

Day03留言。

Github上找到一个链接，里面有各种工具，包括execsnoop。



**walker**

2018-12-03

👍 3

execsnoop这个工具在centos里找不到，有类似的代替品吗

作者回复: 点击链接到github上。

完全一样的工具应该没有，但可以基于内核追踪技术（比如BPF）来自己写一个。



dexter

2018-12-22

👍 2

环境已搭建好，但现在不知道怎么通过github链接来安装execsnoop



划时代

2018-12-04

👍 2

今天早上收到线上一台机器的CPU使用率告警邮件，马上登陆机器查看告警进程号的/proc情况，使用uptime、pidstat和top命令查当前运行情况，后面定位到是crontab定时任务中的程序引起负载过高。



夜空中最亮的...

2018-12-03

👍 2

execsnoop

这个工具没找到

作者回复: 之前的链接有错误，已经修复了，可以点击链接跳过去



每天晒白牙

2018-12-03

👍 2

【D6打卡】这几天回丈母娘家，没带电脑，没能实战，只能看文章了，回来后要补上



风

2018-12-16

👍 1

老师好，我在实验的过程中，在最后使用 perf record -ag 的时候，发现记录下来的值，其中 stress 并不是消耗 CPU 最猛的进程，而是swapper，不知道什么原因？碰到这种情况时，该如何继续排查下去？以下是我的 perf report

Samples: 223K of event 'cpu-clock', Event count (approx.): 55956000000

Children Self Command Shared Object Symbol...

展开 ▼

作者回复: 嗯嗯 很好的问题。简单的说，swapper跟我们要分析的对象无关。这也是为什么我们不上来就用perf，而是先用其他方法缩小范围。我还会在答疑篇里解释swapper的作用





Griffin

2018-12-09

1

实际生产环境中的进程更多，stress藏在ps中根本不容易发现，pstree的结果也非常大。老师有空讲讲如何找到这些异常进程的方法和灵感。

作者回复: 嗯嗯，这是一个模块，侧重于CPU的分析，后面还会讲磁盘、网络等等



□

2018-12-04

1

400%cpu 150%user 3%nice 119%sys 127%idle 0% iow 0%irq 1% sirq 0%host

PID USER PR NI VIRT RES SHR S[%CPU] %MEM TIME+ARGS

4149 system 10 -10 1.8G 176M 113M 141 6.4 7:13.67

4127 root 20 0 0 0 0 98.3 0 7:23.93

老师你好，板卡上linux 系统好多工具无法使用，只能用top查看CPU使用率很高，...

展开



kakasir

2018-12-04

1

因为开放面比较窄，老师后面提到的一些php，docker，nginx都比较陌生，新的虚拟机环境都需要安装，操作起来就困难重重，不过确实能通过案例领会解决问题的思路，还是会坚持下去



mj4ever

2018-12-03

1

整理下今天学到的知识：

1、遇到CPU使用率高，但又无法通过top定位到某个进程时，就需要考虑是否是短时应用所导致，如：

(1) 应用本身问题，导致反复重启

(2) 应用通过exec调用了其他二进制程序，这些程序运行时间又比较短...

展开



阿蒙

2018-12-03

1

老师在某楼的回复中说 strace 没有统计功能，这里提一下，-c 可以统计某 pid 的系统调用

次数，但确实没法对整个系统进行统计。

---



**渡渡鸟\_linux**

2018-12-03

👍 1

这个案例当中，使用perf抓取CPU时钟信息，制作火焰图，放入web服务器目录，通过浏览器查看就能非常直观的看到是那个函数引起的问题。

---



**赵强强**

2018-12-03

👍 1

倪老师您好，我在网上看到一个关于iowait指标的解释，非常形象，但不确定是否准确，帮忙鉴别一下，谢谢，链接 <http://linuxperf.com/?p=33>

---



**bruceding**

2019-02-12

👍

对于内核函数的调试，4.0 的内核可以使用 eBPF 工具，2.6 或者 4.0 以下的工具，使用 systemtap。perf 是基于采样的原理。本文的例子 execsnoop 可以替换成 [https://sourceware.org/systemtap/SystemTap\\_Beginners\\_Guide/threadtimesect.html](https://sourceware.org/systemtap/SystemTap_Beginners_Guide/threadtimesect.html)。systemtap 中文资料比较少，本人也翻译了相关文档，参考：<http://systemtap.bruceding.com/>。

---



**bruceding**

2019-02-12

👍

sar -w 或者 sar -w 1 也能直观的看到每秒生成线程或者进程的数量。Brendan Gregg 确实是这个领域的大师，贡献了很多的技术理念和实践经验。他的《性能之巅》 可以和本课对比着看，会有更多的理解。