

08 | 案例篇：系统中出现大量不可中断进程和僵尸进程怎么办？（下）

2018-12-07 倪朋飞



讲述：冯永吉

时长 10:53 大小 9.98M



你好，我是倪朋飞。

上一节，我给你讲了 Linux 进程状态的含义，以及不可中断进程和僵尸进程产生的原因，我们先来简单复习下。

使用 ps 或者 top 可以查看进程的状态，这些状态包括运行、空闲、不可中断睡眠、可中断睡眠、僵尸以及暂停等。其中，我们重点学习了不可中断状态和僵尸进程：

不可中断状态，一般表示进程正在跟硬件交互，为了保护进程数据与硬件一致，系统不允许其他进程或中断打断该进程。

僵尸进程表示进程已经退出，但它的父进程没有回收该进程所占用的资源。


上一节的最后，我用一个案例展示了处于这两种状态的进程。通过分析 top 命令的输出，我们发现了两个问题：

第一，iowait 太高了，导致系统平均负载升高，并且已经达到了系统 CPU 的个数。

第二，僵尸进程在不断增多，看起来是应用程序没有正确清理子进程的资源。

相信你一定认真思考过这两个问题，那么，真相到底是什么呢？接下来，我们一起顺着这两个问题继续分析，找出根源。

首先，请你打开一个终端，登录到上次的机器中。然后执行下面的命令，重新运行这个案例：

 复制代码

```
1 # 先删除上次启动的案例
2 $ docker rm -f app
3 # 重新运行案例
4 $ docker run --privileged --name=app -itd feisky/app:iowait
```


iowait 分析

我们先来看一下 iowait 升高的问题。

我相信，一提到 iowait 升高，你首先会想要查询系统的 I/O 情况。我一般也是这种思路，那么什么工具可以查询系统的 I/O 情况呢？

这里，我推荐的正是上节课要求安装的 dstat，它的好处是，可以同时查看 CPU 和 I/O 这两种资源的使用情况，便于对比分析。

那么，我们在终端中运行 dstat 命令，观察 CPU 和 I/O 的使用情况：

 复制代码


```
1 # 间隔 1 秒输出 10 组数据
2 $ dstat 1 10
3 You did not select any stats, using -cdngy by default.
4 --total-cpu-usage-- -dsk/total- -net/total- ---paging-- ---system--
5 usr sys idl wai stl| read writ| recv send| in out | int csw
6  0  0 96  4  0|1219k 408k|  0    0 |  0    0 | 42  885
7  0  0  2 98  0| 34M    0 |198B 790B|  0    0 | 42  138
```

8	0	0	0	100	0	34M	0	66B	342B	0	0	42	135
9	0	0	84	16	0	5633k	0	66B	342B	0	0	52	177
10	0	3	39	58	0	22M	0	66B	342B	0	0	43	144
11	0	0	0	100	0	34M	0	200B	450B	0	0	46	147
12	0	0	2	98	0	34M	0	66B	342B	0	0	45	134
13	0	0	0	100	0	34M	0	66B	342B	0	0	39	131
14	0	0	83	17	0	5633k	0	66B	342B	0	0	46	168
15	0	3	39	59	0	22M	0	66B	342B	0	0	37	134

从 dstat 的输出，我们可以看到，每当 iowait 升高（wai）时，磁盘的读请求（read）都会很大。这说明 iowait 的升高跟磁盘的读请求有关，很可能就是磁盘读导致的。

那到底是哪个进程在读磁盘呢？不知道你还记不记得，上节在 top 里看到的不可中断状态进程，我觉得它就很可疑，我们试着来分析下。

我们继续在刚才的终端中，运行 top 命令，观察 D 状态的进程：

 复制代码

```

1 # 观察一会儿按 Ctrl+C 结束
2 $ top
3 ...
4  PID USER      PR  NI    VIRT    RES    SHR  S  %CPU  %MEM     TIME+  COMMAND
5  4340 root        20   0   44676   4048   3432  R   0.3   0.0   0:00.05  top
6  4345 root        20   0   37280   33624   860  D   0.3   0.0   0:00.01  app
7  4344 root        20   0   37280   33624   860  D   0.3   0.4   0:00.01  app
8  ...
9

```

我们从 top 的输出找到 D 状态进程的 PID，你可以发现，这个界面里有两个 D 状态的进程，PID 分别是 4344 和 4345。

接着，我们查看这些进程的磁盘读写情况。对了，别忘了工具是什么。一般要查看某一个进程的资源使用情况，都可以用我们的老朋友 pidstat，不过这次记得加上 -d 参数，以便输出 I/O 使用情况。

比如，以 4344 为例，我们在终端里运行下面的 pidstat 命令，并用 -p 4344 参数指定进程号：

```

1 # -d 展示 I/O 统计数据，-p 指定进程号，间隔 1 秒输出 3 组数据
2 $ pidstat -d -p 4344 1 3
3 06:38:50      UID      PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
4 06:38:51        0    4344      0.00      0.00      0.00      0    app
5 06:38:52        0    4344      0.00      0.00      0.00      0    app
6 06:38:53        0    4344      0.00      0.00      0.00      0    app

```

在这个输出中，kB_rd 表示每秒读的 KB 数，kB_wr 表示每秒写的 KB 数，iodelay 表示 I/O 的延迟（单位是时钟周期）。它们都是 0，那就表示此时没有任何的读写，说明问题不是 4344 进程导致的。

可是，用同样的方法分析进程 4345，你会发现，它也没有任何磁盘读写。

那要怎么知道，到底是哪个进程在进行磁盘读写呢？我们继续使用 pidstat，但这次去掉进程号，干脆就来观察所有进程的 I/O 使用情况。

在终端中运行下面的 pidstat 命令：

```

1 # 间隔 1 秒输出多组数据（这里是 20 组）
2 $ pidstat -d 1 20
3 ...
4 06:48:46      UID      PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
5 06:48:47        0    4615      0.00      0.00      0.00      1  kworker/u4:1
6 06:48:47        0    6080  32768.00      0.00      0.00     170    app
7 06:48:47        0    6081  32768.00      0.00      0.00     184    app
8
9 06:48:47      UID      PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
10 06:48:48        0    6080      0.00      0.00      0.00     110    app
11
12 06:48:48      UID      PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
13 06:48:49        0    6081      0.00      0.00      0.00     191    app
14
15 06:48:49      UID      PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
16
17 06:48:50      UID      PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
18 06:48:51        0    6082  32768.00      0.00      0.00      0    app
19 06:48:51        0    6083  32768.00      0.00      0.00      0    app
20
21 06:48:51      UID      PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay  Command
22 06:48:52        0    6082  32768.00      0.00      0.00     184    app
23
24 06:48:52        0    6083  32768.00      0.00      0.00     175    app

```

```
25 06:48:52      UID      PID   kB_rd/s   kB_wr/s kB_ccwr/s iodelay Command
26 06:48:53        0    6083     0.00     0.00     0.00    105  app
27 ...
```

观察一会儿可以发现，的确是 app 进程在进行磁盘读，并且每秒读的数据有 32 MB，看来就是 app 的问题。不过，app 进程到底在执行啥 I/O 操作呢？

这里，我们需要回顾一下进程用户态和内核态的区别。进程想要访问磁盘，就必须使用系统调用，所以接下来，重点就是找出 app 进程的系统调用了。

strace 正是最常用的跟踪进程系统调用的工具。所以，我们从 pidstat 的输出中拿到进程的 PID 号，比如 6082，然后在终端中运行 strace 命令，并用 -p 参数指定 PID 号：

[📄 复制代码](#)

```
1 $ strace -p 6082
2 strace: attach: ptrace(PTRACE_SEIZE, 6082): Operation not permitted
```

这儿出现了一个奇怪的错误，strace 命令居然失败了，并且命令报出的错误是没有权限。按理来说，我们所有操作都已经是以 root 用户运行了，为什么还会没有权限呢？你也可以先想一下，碰到这种情况，你会怎么处理呢？

一般遇到这种问题时，我会先检查一下进程的状态是否正常。比如，继续在终端中运行 ps 命令，并使用 grep 找出刚才的 6082 号进程：

[📄 复制代码](#)


```
1 $ ps aux | grep 6082
2 root      6082  0.0  0.0    0    0 pts/0    Z+   13:43   0:00 [app] <defunct>
```

果然，进程 6082 已经变成了 Z 状态，也就是僵尸进程。僵尸进程都是已经退出的进程，所以就没法儿继续分析它的系统调用。关于僵尸进程的处理方法，我们一会儿再说，现在还是继续分析 iowait 的问题。

到这一步，你应该注意到了，系统 iowait 的问题还在继续，但是 top、pidstat 这类工具已经不能给出更多的信息了。这时，我们就应该求助那些基于事件记录的动态追踪工具

了。

你可以用 `perf top` 看看有没有新发现。再或者，可以像我一样，在终端中运行 `perf record`，持续一会儿（例如 15 秒），然后按 `Ctrl+C` 退出，再运行 `perf report` 查看报告：

 复制代码

```
1 $ perf record -g
2 $ perf report
```

接着，找到我们关注的 `app` 进程，按回车键展开调用栈，你就会得到下面这张调用关系图：


Samples: 143K of event 'cpu-clock', Event count (approx.): 35954750000					
Children	Self	Command	Shared Object	Symbol	
+ 99.17%	0.00%	swapper	[kernel.vmlinux]	[k]	0x00000000002000d5
+ 99.17%	0.00%	swapper	[kernel.vmlinux]	[k]	cpu_startup_entry
+ 99.17%	0.00%	swapper	[kernel.vmlinux]	[k]	do_idle
+ 99.15%	0.00%	swapper	[kernel.vmlinux]	[k]	default_idle_call
+ 99.15%	0.00%	swapper	[kernel.vmlinux]	[k]	arch_cpu_idle
+ 99.15%	0.00%	swapper	[kernel.vmlinux]	[k]	default_idle
+ 99.15%	99.11%	swapper	[kernel.vmlinux]	[k]	native_safe_halt
+ 67.79%	0.00%	swapper	[kernel.vmlinux]	[k]	start_secondary
+ 31.38%	0.00%	swapper	[kernel.vmlinux].init.text	[k]	x86_64_start_kernel
+ 31.38%	0.00%	swapper	[kernel.vmlinux].init.text	[k]	x86_64_start_reservations
+ 31.38%	0.00%	swapper	[kernel.vmlinux].init.text	[k]	start_kernel
+ 31.38%	0.00%	swapper	[kernel.vmlinux]	[k]	rest_init
- 0.66%	0.00%	app	[kernel.vmlinux]	[k]	entry_SYSCALL_64
entry_SYSCALL_64					
- do_syscall_64					
- 0.64% sys_read					
vfs_read					
__vfs_read					
new_sync_read					
blkdev_read_iter					
generic_file_read_iter					
- blkdev_direct_IO					
- 0.57% bio_iov_iter_get_pages					
- iov_iter_get_pages					
+ get_user_pages_fast					
- 0.66%	0.00%	app	[kernel.vmlinux]	[k]	do_syscall_64
- do_syscall_64					
- 0.64% sys_read					
vfs_read					
__vfs_read					
new_sync_read					
blkdev_read_iter					
generic_file_read_iter					
- blkdev_direct_IO					
+ 0.57% bio_iov_iter_get_pages					
+ 0.65%	0.00%	app	[unknown]	[k]	0x10be258d4c544155
+ 0.65%	0.00%	app	libc-2.27.so	[.]	__libc_start_main

这个图里的 `swapper` 是内核中的调度进程，你可以先忽略掉。

我们来看其他信息，你可以发现，app 的确在通过系统调用 `sys_read()` 读取数据。并且从 `new_sync_read` 和 `blkdev_direct_IO` 能看出，进程正在对磁盘进行**直接读**，也就是绕过了系统缓存，每个读请求都会从磁盘直接读，这就可以解释我们观察到的 `iowait` 升高了。

看来，罪魁祸首是 app 内部进行了磁盘的直接 I/O 啊！


下面的问题就容易解决了。我们接下来应该从代码层面分析，究竟是哪里出现了直接读请求。查看源码文件 [app.c](#)，你会发现它果然使用了 `O_DIRECT` 选项打开磁盘，于是绕过了系统缓存，直接对磁盘进行读写。

 复制代码

```
1 open(disk, O_RDONLY|O_DIRECT|O_LARGEFILE, 0755)
```


直接读写磁盘，对 I/O 敏感型应用（比如数据库系统）是很友好的，因为你可以 在应用中，直接控制磁盘的读写。但在大部分情况下，我们最好还是通过系统缓存来优化磁盘 I/O，换句话说，删除 `O_DIRECT` 这个选项就是了。

[app-fix1.c](#) 就是修改后的文件，我也打包成了一个镜像文件，运行下面的命令，你就可以启动它了：

 复制代码

```
1 # 首先删除原来的应用
2 $ docker rm -f app
3 # 运行新的应用
4 $ docker run --privileged --name=app -itd feisky/app:iowait-fix1
```

最后，再用 `top` 检查一下：

 复制代码

```
1 $ top
2 top - 14:59:32 up 19 min,  1 user,  load average: 0.15, 0.07, 0.05
3 Tasks: 137 total,  1 running,  72 sleeping,   0 stopped, 12 zombie
4 %Cpu0  :  0.0 us,  1.7 sy,  0.0 ni, 98.0 id,  0.3 wa,  0.0 hi,  0.0 si,  0.0 st
5 %Cpu1  :  0.0 us,  1.3 sy,  0.0 ni, 98.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
6 ...
7
```


	PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
9	3084	root	20	0	0	0	0	Z	1.3	0.0	0:00.04	app
10	3085	root	20	0	0	0	0	Z	1.3	0.0	0:00.04	app
11	1	root	20	0	159848	9120	6724	S	0.0	0.1	0:09.03	systemd
12	2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd
13	3	root	20	0	0	0	0	I	0.0	0.0	0:00.40	kworker/0:0
14	...											

你会发现，iowait 已经非常低了，只有 0.3%，说明刚才的改动已经成功修复了 iowait 高的问题，大功告成！不过，别忘了，僵尸进程还在等着你。仔细观察僵尸进程的数量，你会郁闷地发现，僵尸进程还在不断的增长中。

僵尸进程

接下来，我们就来处理僵尸进程的问题。既然僵尸进程是因为父进程没有回收子进程的资源而出现的，那么，要解决掉它们，就要找到它们的根儿，**也就是找出父进程，然后在父进程里解决。**

父进程的找法我们前面讲过，最简单的就是运行 pstree 命令：

 复制代码

```


1 # -a 表示输出命令行选项
2 # p 表 PID
3 # s 表示指定进程的父进程
4 $ pstree -aps 3084
5 systemd,1
6   └─dockerd,15006 -H fd://
7       └─docker-containe,15024 --config /var/run/docker/containerd/containerd.toml
8           └─docker-containe,3991 -namespace moby -workdir...
9               └─app,4009
10                   └─(app,3084)

```

运行完，你会发现 3084 号进程的父进程是 4009，也就是 app 应用。

所以，我们接着查看 app 应用程序的代码，看看子进程结束的处理是否正确，比如有没有调用 wait() 或 waitpid()，抑或是，有没有注册 SIGCHLD 信号的处理函数。

现在我们查看修复 iowait 后的源码文件 [app-fix1.c](#)，找到子进程的创建和清理的地方：

 复制代码


```

1  int status = 0;
2  for (;;) {
3      for (int i = 0; i < 2; i++) {
4          if(fork()== 0) {
5              sub_process();
6          }
7      }
8      sleep(5);
9  }
10
11 while(wait(&status)>0);

```

循环语句本来就容易出错，你能找到这里的问题吗？这段代码虽然看起来调用了 wait() 函数等待子进程结束，但却错误地把 wait() 放到了 for 死循环的外面，也就是说，wait() 函数实际上并没被调用到，我们把它挪到 for 循环的里面就可以了。

修改后的文件我放到了 [app-fix2.c](#) 中，也打包成了一个 Docker 镜像，运行下面的命令，你就可以启动它：


 复制代码

```

1 # 先停止产生僵尸进程的 app
2 $ docker rm -f app
3 # 然后启动新的 app
4 $ docker run --privileged --name=app -itd feisky/app:iowait-fix2

```

启动后，再用 top 最后来检查一遍：

 复制代码

```

1 $ top
2 top - 15:00:44 up 20 min,  1 user,  load average: 0.05, 0.05, 0.04
3 Tasks: 125 total,  1 running,  72 sleeping,   0 stopped,   0 zombie
4 %Cpu0  :  0.0 us,  1.7 sy,  0.0 ni, 98.3 id,   0.0 wa,   0.0 hi,   0.0 si,   0.0 st
5 %Cpu1  :  0.0 us,  1.3 sy,  0.0 ni, 98.7 id,   0.0 wa,   0.0 hi,   0.0 si,   0.0 st
6 ...
7
8  PID USER      PR  NI   VIRT    RES    SHR S  %CPU  %MEM    TIME+  COMMAND
9  3198 root        20   0   4376    840    780 S   0.3   0.0   0:00.01 app
10    2 root        20   0      0      0      0 S   0.0   0.0   0:00.00 kthreadd
11    3 root        20   0      0      0      0 I   0.0   0.0   0:00.41 kworker/0:0
12 ...

```

好了，僵尸进程（Z 状态）没有了，iowait 也是 0，问题终于全部解决了。

小结

今天我用一个多进程的案例，带你分析系统等待 I/O 的 CPU 使用率（也就是 iowait%）升高的情况。

虽然这个案例是磁盘 I/O 导致了 iowait 升高，不过，**iowait 高不一定代表 I/O 有性能瓶颈。当系统中只有 I/O 类型的进程在运行时，iowait 也会很高，但实际上，磁盘的读写远没有达到性能瓶颈的程度。**

因此，碰到 iowait 升高时，需要先用 dstat、pidstat 等工具，确认是不是磁盘 I/O 的问题，然后再找是哪些进程导致了 I/O。

等待 I/O 的进程一般是不可中断状态，所以用 ps 命令找到的 D 状态（即不可中断状态）的进程，多为可疑进程。但这个案例中，在 I/O 操作后，进程又变成了僵尸进程，所以不能用 strace 直接分析这个进程的系统调用。

这种情况下，我们用了 perf 工具，来分析系统的 CPU 时钟事件，最终发现是直接 I/O 导致的问题。这时，再检查源码中对应位置的问题，就很轻松了。

而僵尸进程的问题相对容易排查，使用 pstree 找出父进程后，去查看父进程的代码，检查 wait() / waitpid() 的调用，或是 SIGCHLD 信号处理函数的注册就行了。

思考

最后，我想邀请你一起来聊聊，你碰到过的不可中断状态进程和僵尸进程问题。你是怎么分析它们的根源？又是怎么解决的？在今天的案例操作中，你又有什么新的发现吗？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 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）

下一篇 09 | 基础篇：怎么理解Linux软中断？

精选留言 (61)

写留言



zecho

2018-12-07

21

提一个建议，案例的讲解过于简单，与预期有些差距，很多时候我们实际遇到的要比这个复杂，这会带来不是简单的几个命令就可以，特别需要更深入的工具，比如brendan中火焰图，perf-tools，或者systemtap等等；希望能找些实际的案例，谢谢。

展开

作者回复: 嗯嗯，实际上这里的案例是我故意设计的比较简单，这样初学者可以把重点放到理解当前讲的原理和指标上，而不是看着一堆的新工具和内核函数而感到害怕。当然了，火焰图、perf-tools、systemtap这些工具以后也会讲到，只是还是让我们先把基础的东西铺垫好。

如果你已经对这些比较熟悉了，推荐去把这些思路应用到实际的系统中去分析，然后在这里跟大家分享你的所得。我相信，你有可能会发现不同的理解，甚至是更好的分析思路。



王涛

2018-12-07

👍 15

d8打卡。看完这部分，作为一名运维人员就尴尬了，当开发跟你说机器性能有问题时，这个问题就变成了甩锅问题。开发说代码没问题，你又看不懂开发的代码。。。

作者回复: 找出进程就可以甩锅了☺

从磁盘IO的角度来说，其实很容易找出那些进程在消耗IO资源。因为这里侧重的是CPU使用的分析，所以IO讲的不是特别深入。后续的IO部分还会有更细致的拆解。



walker

2018-12-07

👍 7

有时候直接杀死僵尸进程的时候会导致服务不可用，或是崩溃。在线上运行的服务出现僵尸进程时，怎样处理比较好呢？



柯锦玲(侠...)

2018-12-07

👍 6

dock镜像等资料在哪里下载？

展开 ▾



每天晒白牙

2018-12-07

👍 4

【D8打卡】

今天主要学习的是系统中出现了大量不可中断进程和僵尸进程的处理方法
现象：

- ①iowait太高，导致平均负载升高，并且达到了系统CPU的个数
- ②僵尸进程不断增多...

展开 ▾



ninuxer

2018-12-07

👍 3

打卡day9

在用perf展开的分析详情中，有个children和self都90%多的swapper进程，如果我看到这个，我肯定会先去围绕这个进程展开，而会忽略占用才0.6%得app进程，关于这个情况，是我理解上有什么偏差么？



白华

2018-12-07

👍 3

今天进行实验看来还是不会成功的，上次在你的docker hub仓库中看到了iowait镜像，试了最新的几个，在centos7虚拟机中还是不行



姜小鱼

2018-12-07

👍 2

老师：iowait%生高并不能得出存在io性能的结论，还要继续看io量(dstat)和io并发等情况.那么这个io量到底达到多少才能说明存在性能瓶颈？有一个量化指标吗？期待回复，谢谢

作者回复: IO部分会讲的



我来也

2018-12-07

👍 2

[D8打卡]

今天又学了两个乖, dstat可以同时看cpu和io. (上篇文章安装后只看了下效果,没想到这一层).strace可以追踪系统调用.

虽然我之前也写过linux c程序,但是看到sys_read/new_sync_read/blkdev_direct_IO确实不知道是正在对磁盘进行直接读, 即使看了代码 也不知道 O_DIRECT 这个参数就是直接...
展开

作者回复: 嗯嗯 可以再思考一点，比如假如父进程已经是init了又该怎么办？注意init进程是不会退出的



运维小司机

2018-12-07

👍 2

后面那两个镜像，我不知道是不是我系统的问题，我运行iowait直接飚满了，系统直接卡

死。

展开 ▾



发条橙子 ...

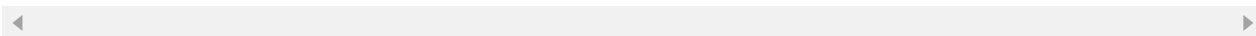
2018-12-16

👍 1

老师，我这边使用 stress 来模拟 io 的请求，发现 stress 的进程时而 D 时而 R。但是用 pidstat、iostat 以及 dstat 都没有看到对磁盘有读数据请求和写数据请求。所以这个是 stress 工具本身的问题么？？

展开 ▾

作者回复: 可能跟stress的选项有关，你的命令是什么？



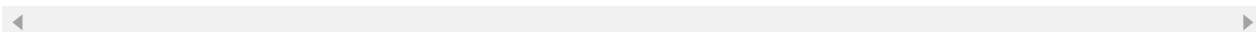
赵强强

2018-12-08

👍 1

倪老师.公司一个服务调用量很大，top命令输出显示php-fpm worker进程cpu使用率很高，但是pidstat输出确找不到该进程，进程确实一直存在的，是什么原因呢

作者回复: 试试分析下线程



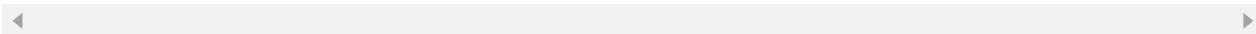
路过

2018-12-07

👍 1

听课程，还会需要有点编程的功底，否则这个问题排查起来只能粗暴简单处理了。把锅丢给开发了。:)

作者回复: 嗯嗯 了解一些编程的基本功很有帮助，特别是复杂问题的最后很可能要去分析函数调用栈等，更需要一定的编程功底。



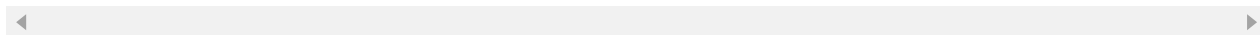
破晓

2018-12-07

👍 1

老师想问下关于cpu使用率这块的概念，system的cpu使用率，包不包含iowait的？他们之间是什么关系？

作者回复: 不包括，这是两个不同的指标



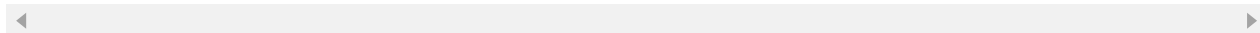
帆帆帆帆帆...

2018-12-07

👍 1

@jeff，数据库一般都有自己的数据缓冲池bufferpool，在合适的时间，数据库会从磁盘读入数据到bufferpool，或者从bufferpool写出数据到磁盘。在这种情况下，再使用文件系统缓存，反而不会有性能的提升，而且数据库写出数据到磁盘的时候，必须写到了磁盘才算真的完成了数据的持久化。

作者回复: 嗯嗯，谢谢分享



hsggj

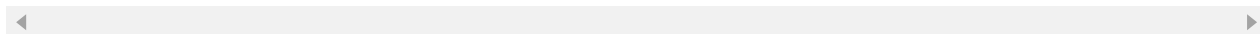
2018-12-07

👍 1

老师，请问一下，perf除了可以检测C++之外，其他的代码如java,php的问题可以检测吗？

展开 ∨

作者回复: 可以的，我们案例里面已经有一个PHP的示例，我还会在答疑中介绍Java的使用方法。



路过

2018-12-07

👍 1

用ubuntu 做实验很顺利。请问老师，曾经发生僵尸进程的父进程是1，服务器又不方便重启，如何清理这样的僵尸进程？谢谢老师！



tianyitan...

2019-03-01

👍

老师：

运行 perf top ，提示打开老师的测试案例 app 失败





信

2019-03-01



老师，我在使用strace命令 查看java 进程的时候，出现futex_wait，然后没别的输出了？
像java 这种想分析内核调用情况的工具还有哪些推荐呢



lin

2019-02-25



打卡

展开 ∨