

我的进程去哪儿了，谁杀了我的进程

目录

- [Linux Signal](#)
- [到底是什么信号](#)
 - [OOM](#)
- [谁发的信号](#)
 - [systemtap](#)
 - [audit](#)
- [案例与总结](#)
- [references](#)

正文

曾经在我眼前，却又消失不见，这是今天的第六遍。

一段感情，我们希望它能够天长地久，越稳定越好，最怕的就是中途夭折，无疾而终。

即使不能到海枯石烂的一天，我们也希望好聚好散，大家理智的告别，然后笑着活下去。

笑着活下去



况且，我们时候，我们只是希望给这段感情按下一个暂停键，大家先冷静思考自己的问题，然后重新开始。

最让人抓狂的是突然就联系不上另一半，不接你电话、不回你短信、消息在你的世界。而你根本不知道原因，可能是因为在约会的时候你没有夸她新买的鞋子漂亮；也许是你多看了路边的美眉两眼；也许是出现了更强有力的竞争者；也许是她的父母觉得程序员太屌丝，不愿意你们继续来往；也许是她得了不治之症，不想连累你。总之，你不知道哪里出了问题，惶惶不可终日。

好吧，这里不是情感专栏，我是一个程序员，咱说点有关服务器程序的事情，准确一点，是指服务端进程，后文不加区分。

服务端的程序，一般都希望比较长时间的运行，比如7*24小时。不过，程序也需要更新，比如修Bug，比如增加新功能，比如修复增加新功能引入的bug。

一个大型系统一般包含多个进程，同一个服务也可能是有多个进程组成，那么可以将这组进程逐步更新：先让一部分进程停止提供服务，等待已有的请求都完毕之后，重启这些服务，然后再更新替他进程。

即使，我们需要关闭所有服务，也需要优雅（graceful）地停止这些进程。所谓的优雅，就是保证已有的请求都能处理完，需要持久化的状态、数据都保存成功，然后再结束进程，一般来说，可以通过发送信号或者通过哨兵（sentinel）来结束。

只要是预期内的进程结束，那么都是ok的。而预期之外的进程结束往往令程序员抓狂，线上服务器的问题往往意味着分分钟几位数的损失、KPI、年终奖.....想想就很恐怖。

本文地址：<http://www.cnblogs.com/xybaby/p/8098229.html>

Linux Signal

[回到顶部](#)

一个进程的异常终止，通常有两种情况，一种是crash，另一种是被kill掉了。

crash是指程序出现了自己无法解决的异常情况，只能终止，比如Python语言抛出了一个未被捕获的异常，就会结束程序。对于C、C++，最有名的就是段错误（segmentation fault），如果在Linux下面，那么会生成coredump，程序员通过gdb（有可能）可以分析出crash的原因。当然，要生成coredump也是需要正确的设置，可以通过ulimit（ulimit -c）查看或者设置。

而进程被kill掉，就是其他进程给目标进程发送了信号（signal），当然也可以是自己给自己发的信号，而目标进程没有正确处理这些信号，或者根本没有机会（权力）处理这些信号，那么目标进程就有可能终止。

信号是Unix-like系统进程间通信（IPC）的一种方式，这种通知是异步的，信号是一种软中断，操作系统会将目标进程的正常执行流程暂停，然后处理信号。如果目标进程注册了相应的信号处理函数（signal handler），那么就会调用这个signal handler，否则会执行默认的信号处理函数。

不同的操作系统，支持的信号可能略有差异，可以使用kill -l 查看系统所有的信号。下面是Linux上常见的信号以及处理机制

信号	值	处理动作	发出信号的原因
SIGHUP	1	A	终端挂起或者控制进程终止
SIGINT	2	A	键盘中断（如break键被按下）
SIGQUIT	3	C	键盘的退出键被按下
SIGILL	4	C	非法指令
SIGABRT	6	C	由abort(3)发出的退出指令
SIGFPE	8	C	浮点异常
SIGKILL	9	AEF	Kill信号
SIGSEGV	11	C	无效的内存引用
SIGPIPE	13	A	管道破裂: 写一个没有读端口的管道
SIGALRM	14	A	由alarm(2)发出的信号
SIGTERM	15	A	终止信号
SIGUSR1	30,10,16	A	用户自定义信号1
SIGUSR2	31,12,17	A	用户自定义信号2
SIGCHLD	20,17,18	B	子进程结束信号
SIGCONT	19,18,25		进程继续（曾被停止的进程）

SIGSTOP 17,19,23 DEF 终止进程
SIGTSTP 18,20,24 D 控制终端 (tty) 上按下停止键
SIGTTIN 21,21,26 D 后台进程企图从控制终端读
SIGTTOU 22,22,27 D 后台进程企图从控制终端写

处理动作一项中的字母含义如下

- A 缺省的动作是终止进程
- B 缺省的动作是忽略此信号
- C 缺省的动作是终止进程并进行内核映像转储 (dump core)
- D 缺省的动作是停止进程
- E 信号不能被捕获
- F 信号不能被忽略

如果默认处理动作是C (coredump)，那么就会生成coredump，然后终止进程，在上一篇文章《[啊，我的程序为啥卡住啦](#)》中，提到用kill -11 pid 来终止、调试卡住的程序，这个11就是指信号SIGSEGV

注意SIGKILL SIGSTOP这两个信号，既不可以被捕获，也不能被忽略，就是说收到这两个信号，程序就会不留痕迹地终止。

到底是什么信号

[回到顶部](#)

从上面，我们可以看到，有很多信号都可以终止进程，如果我们没有针对某种信号指定处理函数，那么我们怎么知道进程是被哪一个进程kill掉了呢，那就是[strace](#)。

我们以一段简单的Python代码为例：



```
1 # -*- coding: utf-8 -*-
2
3 def func():
4     while True:
5         pass
6
7 if __name__ == '__main__':
8     func()
```



运行该代码：

```
python run_forever.py &
[1] 1035
```

可以看到pid是1035，我们使用strace查看该进程：\$ strace -T -tt -e trace=all -p 1035，然后另起终端kill这个Python进程：\$kill -9 1035。在strace可以看到：

```
$ strace -T -tt -e trace=all -p 1035
Process 1035 attached
16:19:52.092202 +++ killed by SIGKILL +++
```

也可以使用其他信号kill，比如，重新运行Python程序，进程为32599，kill -10 32599，那么strace的结果是

```
$strace -T -tt -e trace=all -p 32599
Process 32599 attached
16:09:20.264993 --- SIGUSR1 {si_signo=SIGUSR1, si_code=SI_USER, si_pid=29373,
si_uid=3010} ---
16:09:20.265656 +++ killed by SIGUSR1 +++
```

对比kill -9， kill -10多了一行信息，这是非常有用的信息，因为有了两个非常重要的字段，*si_pid*、*si_uid*

```
pid_t si_pid; // sending process ID
pid_t si_uid; // Real user ID of sending process
```

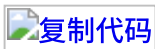
即*si_pid*说明是哪一個進程發送的信息，*si_uid*是哪一個用戶發出的信息，使用id *si_uid*就能看出詳細的用戶信息。这样就直接定位了谁发送的信息。

不过，很多时候，进程被干掉，都是使用了kill -9（SIGKILL），那么即使使用strace也只是知道被干掉了，也没法知道被谁干掉了。而且，除非一个程序经常被kill掉，否则很少有人有strace长期监控这个进程。

OOM

关于进程收到了SIGKILL信号，有一种不得不提的情况，那就是[OOM \(out of memory\)](#)，简单来说，就是当Linux系统内存不足，在大量使用swap之后，会kill掉内存占用最大的进程。这应该算操作系统系统自身的一种保护机制，以防更多的进程不能正常工作。关于OOM killer，网上有详尽的资料介绍，在这里只是简单看看现象。

下面是一个简单的Python程序，该程序会无限的分配内存，直到内存不足：



```
1 # -*- coding: utf-8 -*-
2
3 def func():
4     l = [1]
5     while True:
6         l.append(l * 2)
7
8 if __name__ == '__main__':
9     func()
```



运行该程序，然后用strace追踪，前面都是正常的内存分配相关的系统调用，直到最后

```
16:02:19.192409 mprotect(0x7f4aa886b000, 552960, PROT_READ|PROT_WRITE) = 0
<0.000014>
16:02:19.744000 mmap(NULL, 552960, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = -1 ENOMEM (Cannot allocate memory) <0.000108>
16:02:19.755545 mprotect(0x7f4aa88f2000, 552960, PROT_READ|PROT_WRITE) = 0
<0.000019>
16:02:23.404805 mmap(NULL, 552960, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0 <ptrace(SYSCALL):No such process>
16:02:25.325272 +++ killed by SIGKILL +++
[1]+ Killed python will_be_killed.py
```

stackoverflow上[who-killed-my-process-and-why](#)一文指出，由于outofmemory被kill掉的进程，会在/var/log下的某个文件中留下最终的遗迹，在笔者使用的debian系统中，可以通过dmesg查看：

```
dmesg -T | grep -E -i -B100 'killed process'
```

日志如下：

```
[月 12日 24 16:02:24 2017] [ pid ] uid tgid total_vm rss nr_ptes swapents oom_score_adj name
[月 12日 24 16:02:24 2017] [ 172] 0 172 10553 722 23 1375 0 systemd-journal
[月 12日 24 16:02:24 2017] [ 181] 0 181 10201 398 22 134 -1000 systemd-udevd
[月 12日 24 16:02:24 2017] [ 518] 0 518 8738 361 21 70 0 cron
[月 12日 24 16:02:24 2017] [ 519] 0 519 4756 314 13 45 0 atd
[月 12日 24 16:02:24 2017] [ 520] 0 520 13795 412 31 143 -1000 sshd
[月 12日 24 16:02:24 2017] [ 525] 0 525 4964 161 14 65 0 systemd-logind
[月 12日 24 16:02:24 2017] [ 536] 105 536 10531 172 26 98 -900 dbus-daemon
[月 12日 24 16:02:24 2017] [ 580] 0 580 1064 362 8 34 0 acpid
[月 12日 24 16:02:24 2017] [ 602] 0 602 4926 373 13 37 0 agetty
[月 12日 24 16:02:24 2017] [ 605] 0 605 4881 373 14 37 0 agetty
[月 12日 24 16:02:24 2017] [ 643] 108 643 8346 283 22 134 0 ntpd
[月 12日 24 16:02:24 2017] [ 889] 104 889 12794 369 26 158 0 exim4
[月 12日 24 16:02:24 2017] [11640] 0 11640 4188 1118 15 0 0 atop
[月 12日 24 16:02:24 2017] [29370] 0 29370 14434 420 30 177 0 sshd
[月 12日 24 16:02:24 2017] [29372] 3010 29372 14434 189 29 147 0 sshd
[月 12日 24 16:02:24 2017] [29373] 3010 29373 7813 491 20 587 0 bash
[月 12日 24 16:02:24 2017] [30731] 0 30731 14434 429 30 177 0 sshd
[月 12日 24 16:02:24 2017] [30733] 3010 30733 14434 328 29 154 0 sshd
[月 12日 24 16:02:24 2017] [30734] 3010 30734 7810 432 19 606 0 bash
[月 12日 24 16:02:24 2017] [30746] 3010 30746 13967 408 30 102 0 su
[月 12日 24 16:02:24 2017] [30747] 0 30747 7389 422 19 194 0 bash
[月 12日 24 16:02:24 2017] [31688] 0 31688 13967 408 31 101 0 su
[月 12日 24 16:02:24 2017] [31689] 3010 31689 7808 482 19 566 0 bash
[月 12日 24 16:02:24 2017] [32128] 3010 32128 7761 445 19 32 0 top
[月 12日 24 16:02:24 2017] [32132] 3010 32132 2357921 1868878 4581 467803 0 python
[月 12日 24 16:02:24 2017] [32133] 3010 32133 1255 152 7 57 0 strace
[月 12日 24 16:02:24 2017] Out of memory: Kill process 32132 (python) score 957 or sacrifice
child
[月 12日 24 16:02:24 2017] Killed process 32132 (python) total-vm:9431684kB, anon-
rss:7473936kB, file-rss:1576kB
```

由于进程使用内存最多，而且没有设置保护机制，所以32122这个python进程就被kill掉了。

所以，如果机器内存使用率较高，那么当进程消失的时候不妨用dmesg看看。

谁发的信号

[回到顶部](#)

更普遍的，即使我们知道进程被SIGKILL干掉了，这没有什么用。关键是得找出谁发送的这个信号，是故意的还是意外，也许是新手写的脚本误伤，也许是老手故意搞破坏。

最简单的，那就是查看[last](#)与[history](#)，last看看谁登陆到了系统，然后再用history看看相关的操作记录。可是，操作记录是可以被清除的，history -c可以清除本终端上的操作记录，也可以直接清空、删除.bash_history，也许有更高级的手段来禁止清空操作记录，但到底是道高一尺魔高一丈，还是魔高一尺道高一丈，我也就不清楚了。

systemtap

在[who-killed-my-process-and-why](#)中，指出了其中的一种办法，使用[systemtap](#)，笔者也是第一次听说这个家伙，据说非常强大。官方给出了捕获信号的[示例代码](#)。



```
probe begin
{
    printf("%-8s %-16s %-5s %-16s %6s %-16s\n",
           "SPID", "SNAME", "RPID", "RNAME", "SIGNUM", "SIGNAME")
}

probe signal.send
{
    if (sig_name == @1 && sig_pid == target())
        printf("%-8d %-16s %-5d %-16s %-6d %-16s\n",
               pid(), execname(), sig_pid, pid_name, sig, sig_name)
}
```



将上述代码保存为sigmon.stp文件，然后查看待监控的进程的pid，使用下述命令监控发送到该进程的SIGKILL信号

```
stap -x pid sigmon.stp SIGKILL
```

不过这种方法，我并没有实验成功，systemtap的安装太麻烦了。。。

audit

在[Who sends a SIGKILL to my process mysteriously on ubuntu server](#)中，提到了另外一个更加简单的方法，那就是使用audit

安装很简单：sudo apt-get install auditd

启动服务并查看状态：service auditd start & service auditd status

然后通过auditctl添加规则：auditctl -a exit,always -F arch=b64 -S kill -F a1=9

启动然后kill掉Python程序

```
$ python run_forever.py &
[1] 24067
$ kill -9 24067
```

使用ausearch搜索结果：ausearch -sc kill。结果如下

```
time->Mon Dec 25 19:52:55 2017
type=PROCTITLE msg=audit(1514202775.088:351): proctitle="bash"
type=OBJ_PID msg=audit(1514202775.088:351): opid=24067 日uid=-1 ouid=3010 oses=-1
ocomm="python"
type=SYSCALL msg=audit(1514202775.088:351): arch=c000003e syscall=62 success=yes exit=0
a0=5e03 a1=9 a2=0 a3=7ffc0d9f7b90 items=0 ppid=1349 pid=1350 uid=3010 gid=3010
euid=3010 suid=3010 fsuid=3010 egid=3010 sgid=3010 fsgid=3010 tty=pts0 comm="bash"
exe="/bin/bash" key=(null)
```

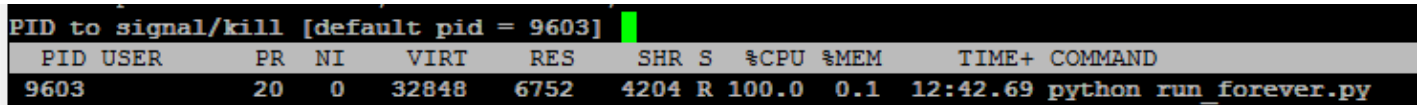
可以看到，信号的目标进程是一个python程序，pid是24067，启动该进程的用户id是3010。kill进程的信号被用户3010在pid为1350的bash中发出。

案例与总结

[回到顶部](#)

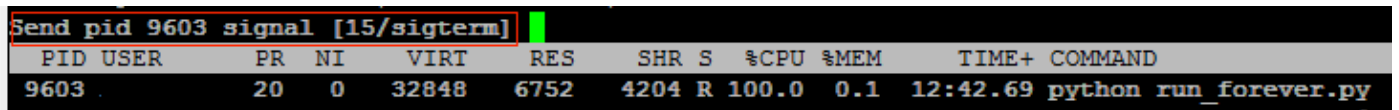
我遇到过的，进程悄无声息消失的情况，有以下几种

- (1) 进程确实是crash了，不过用于core file size设置的问题，没有生成coredump，这里可以通过ulimit -c确认
- (2) oom，代码bug导致进程占用内存过多，被操作系统干掉
- (3) 进程被父进程，或者监控进程（watchdog）给kill掉，这个在使用框架的时候容易出现
- (4) 进程被误杀，诸如这样的脚本 `kill -9 `ps aux | grep python | awk '{print $2}'``，杀掉所有的python进程，这是非常粗暴的方法，非常容易误杀
- (5) top，top命令也能杀掉进程，are you kidding me? No



PID to signal/kill [default pid = 9603]										
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+ COMMAND
9603		20	0	32848	6752	4204	R	100.0	0.1	12:42.69 python run_forever.py

如上图所示，进程9603是一个Python程序，top -c默认按照CPU使用量排序，所以这个CPU 100%的进程在最前面。当按下K键的时候，就会给进程发信号，default pid就是在第一行的进程，当然这里也可以输入pid。直接回车，结果如下图



Send pid 9603 signal [15/sigterm]										
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+ COMMAND
9603		20	0	32848	6752	4204	R	100.0	0.1	12:42.69 python run_forever.py

可以看到，默认的信号是SIGTERM（15），也可以输入信号。在敲回车之后，这个进程就被kill了

即使查看history命令，也只有一个top命令，完全看不出什么，所以使用top命令也要小心啊，捕获SIGTERM也是一个不错的主意。

当然，做好权限控制，就能减少进程被意外Kill，特别是在线上服务器，适当的监控也是必要的。

那么当进程消失的时候，可以按照下列步骤排查

- (1) 看日志
- (2) 查看有没有coredump，查看ulimit -c
- (3) 看系统内存使用量，用dmesg看是不是OOM
- (4) 看last与history，crontab
- (5) 也许一切都在运维的掌控之中



references

[回到顶部](#)

[Taming the OOM killer](#)

[who-killed-my-process-and-why](#)

[Who sends a SIGKILL to my process mysteriously on ubuntu server](#)

本文版权归作者xybaby（博文地址：<http://www.cnblogs.com/xybaby/>）所有，欢迎转载和商用，请在文章页面明显位置给出原文链接并保留此段声明，否则保留追究法律责任的权利，其他事项，可留言咨询。

标签: [linux](#)

[好文要顶](#) [关注我](#) [收藏该文](#)  



[xybaby](#)

[关注 - 7](#)

[粉丝 - 142](#)

[+加关注](#)

10

1

[« 上一篇：啊，我的程序为啥卡住啦](#)

posted @ 2017-12-27 09:05 [xybaby](#) 阅读(1678) 评论(2) [编辑](#) [收藏](#)