



下载APP



02 | 理解进程（1）：为什么我在容器中不能kill 1号进程？

2020-11-18 李程远

容器实战高手课

[进入课程 >](#)**讲述：李程远**

时长 20:01 大小 18.34M



你好，我是程远。

今天，我们正式进入理解进程的模块。我会通过 3 讲内容，带你了解容器 init 进程的特殊之处，还有它需要具备哪些功能，才能保证容器在运行过程中不会出现类似僵尸进程，或者应用程序无法 graceful shutdown 的问题。

那么通过这一讲，我会带你掌握 init 进程和 Linux 信号的核心概念。

问题再现



接下来，我们一起再现用 kill 1 命令重启容器的问题。

我猜你肯定想问，为什么要在容器中执行 `kill 1` 或者 `kill -9 1` 的命令呢？其实这是我们团队里的一位同学提出的问题。

这位同学当时遇到的情况是这样的，他想修改容器镜像里的一个 bug，但因为网路配置的问题，这个同学又不想为了重建 pod 去改变 pod IP。

如果你用过 Kubernetes 的话，你也肯定知道，Kubernetes 上是没有 `restart pod` 这个命令的。这样看来，他似乎只能让 pod 做个原地重启了。**当时我首先想到的，就是在容器中使用 `kill pid 1` 的方式重启容器。**

为了模拟这个过程，我们可以进行下面的这段操作。

如果你没有在容器中做过 `kill 1`，你可以下载我在 GitHub 上的这个 [例子](#)，运行 `make image` 来做一个容器镜像。

然后，我们用 Docker 构建一个容器，用例子中的 **init.sh** 脚本作为这个容器的 init 进程。

最后，我们在容器中运行 `kill 1` 和 `kill -9 1`，看看会发生什么。

[复制代码](#)

```
1 # docker stop sig-proc;docker rm sig-proc
2 # docker run --name sig-proc -d registry/sig-proc:v1 /init.sh
3 # docker exec -it sig-proc bash
4 [root@5cc69036b7b2 /]# ps -ef
5 UID          PID    PPID    C  STIME TTY          TIME CMD
6 root           1         0  0  07:23 ?        00:00:00 /bin/bash /init.sh
7 root           8         1  0  07:25 ?        00:00:00 /usr/bin/coreutils --coreutils
8 root           9         0  6  07:27 pts/0    00:00:00 bash
9 root          22         9  0  07:27 pts/0    00:00:00 ps -ef
10
11 [root@5cc69036b7b2 /]# kill 1
12 [root@5cc69036b7b2 /]# kill -9 1
13 [root@5cc69036b7b2 /]# ps -ef
14 UID          PID    PPID    C  STIME TTY          TIME CMD
15 root           1         0  0  07:23 ?        00:00:00 /bin/bash /init.sh
16 root           9         0  0  07:27 pts/0    00:00:00 bash
17 root          23         1  0  07:27 ?        00:00:00 /usr/bin/coreutils --coreutils
18 root          24         9  0  07:27 pts/0    00:00:00 ps -ef
```

当我们完成前面的操作，就会发现无论运行 `kill 1`（对应 Linux 中的 SIGTERM 信号）还是 `kill -9 1`（对应 Linux 中的 SIGKILL 信号），都无法让进程终止。

那么问题来了，这两个常常用来终止进程的信号，都对容器中的 init 进程不起作用，这是怎么回事呢？

要解释这个问题，我们就要回到容器的两个最基本概念——init 进程和 Linux 信号中寻找答案。

知识详解

如何理解 init 进程？

init 进程的意思并不难理解，你只要认真听我讲完，这块内容基本就不会有问题了。我们下面来看一看。

使用容器的理想境界是一个容器只启动一个进程，但这在现实应用中有时是做不到的。

比如说，在一个容器中除了主进程之外，我们可能还会启动辅助进程，做监控或者 rotate logs；再比如说，我们需要把原来运行在虚拟机（VM）的程序移到容器里，这些原来跑在虚拟机上的程序本身就是多进程的。

一旦我们启动了多个进程，那么容器里就会出现一个 pid 1，也就是我们常说的 1 号进程或者 init 进程，然后**由这个进程创建出其他的子进程**。

接下来，我带你梳理一下 init 进程是怎么来的。

一个 Linux 操作系统，在系统打开电源，执行 BIOS/boot-loader 之后，就会由 boot-loader 负责加载 Linux 内核。


Linux 内核执行文件一般会放在 /boot 目录下，文件名类似 vmlinuz*。在内核完成了操作系统的各种初始化之后，**这个程序需要执行的第一个用户态程就是 init 进程**。

内核代码启动 1 号进程的时候，在没有外面参数指定程序路径的情况下，一般会从几个缺省路径尝试执行 1 号进程的代码。这几个路径都是 Unix 常用的可执行代码路径。


系统启动的时候先是执行内核态的代码，然后在内核中调用 1 号进程的代码，从内核态切换到用户态。

目前主流的 Linux 发行版，无论是 RedHat 系的还是 Debian 系的，都会把 /sbin/init 作为符号链接指向 Systemd。Systemd 是目前最流行的 Linux init 进程，在它之前还有 SysVinit、UpStart 等 Linux init 进程。

但无论是哪种 Linux init 进程，它最基本的功能都是创建出 Linux 系统中其他所有的进程，并且管理这些进程。具体在 kernel 里的代码实现如下：

 复制代码

```
1  init/main.c
2
3      /*
4       * We try each of these until one succeeds.
5       *
6       * The Bourne shell can be used instead of init if we are
7       * trying to recover a really broken machine.
8       */
9
10     if (execute_command) {
11         ret = run_init_process(execute_command);
12         if (!ret)
13             return 0;
14         panic("Requested init %s failed (error %d).",
15             execute_command, ret);
16     }
17
18     if (!try_to_run_init_process("/sbin/init") ||
19         !try_to_run_init_process("/etc/init") ||
20         !try_to_run_init_process("/bin/init") ||
21         !try_to_run_init_process("/bin/sh"))
22         return 0;
23
24
25     panic("No working init found.  Try passing init= option to kernel.  "
26         "See Linux Documentation/admin-guide/init.rst for guidance.");
```

 复制代码

```
1  $ ls -l /sbin/init
2  lrwxrwxrwx 1 root root 20 Feb  5 01:07 /sbin/init -> /lib/systemd/systemd
```

在 Linux 上有了容器的概念之后，一旦容器建立了自己的 Pid Namespace（进程命名空间），这个 Namespace 里的进程号也是从 1 开始标记的。所以，容器的 init 进程也被称为 1 号进程。

怎么样，1 号进程是不是不难理解？关于这个知识点，你只需要记住：**1 号进程是第一个用户态的进程，由它直接或者间接创建了 Namespace 中的其他进程。**


如何理解 Linux 信号？

刚才我给你讲了什么是 1 号进程，要想解决“为什么我在容器中不能 kill 1 号进程”这个问题，我们还得看看 kill 命令起到的作用。

我们运行 kill 命令，其实在 Linux 里就是发送一个信号，那么信号到底是什么呢？这就涉及到 Linux 信号的概念了。

其实信号这个概念在很早期的 Unix 系统上就有了。它一般会从 1 开始编号，通常来说，信号编号是 1 到 31，这个编号在所有的 Unix 系统上都是一样的。

在 Linux 上我们可以用 `kill -l` 来看这些信号的编号和名字，具体的编号和名字我给你列在了下面，你可以看一看。

 复制代码

```
1 $ kill -l
2  1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
3  6) SIGABRT     7) SIGBUS     8) SIGFPE     9) SIGKILL    10) SIGUSR1
4 11) SIGSEGV    12) SIGUSR2   13) SIGPIPE   14) SIGALRM   15) SIGTERM
5 16) SIGSTKFLT  17) SIGCHLD  18) SIGCONT   19) SIGSTOP   20) SIGTSTP
6 21) SIGTTIN    22) SIGTTOU  23) SIGURG    24) SIGXCPU   25) SIGXFSZ
7 26) SIGVTALRM  27) SIGPROF  28) SIGWINCH  29) SIGIO     30) SIGPWR
8 31) SIGSYS
```

用一句话来概括，**信号（Signal）其实就是 Linux 进程收到的一个通知。**这些通知产生的源头有很多种，通知的类型也有很多种。

比如下面这几个典型的场景，你可以看一下：

如果我们按下键盘“Ctrl+C”，当前运行的进程就会收到一个信号 SIGINT 而退出；

如果我们的代码写得有问题，导致内存访问出错了，当前的进程就会收到另一个信号 SIGSEGV；

我们也可以通过命令 `kill <pid>`，直接向一个进程发送一个信号，缺省情况下不指定信号的类型，那么这个信号就是 SIGTERM。也可以指定信号类型，比如命令 `"kill -9 <pid>"`，这里的 9，就是编号为 9 的信号，SIGKILL 信号。

在这一讲中，我们主要用到 **SIGTERM (15) 和 SIGKILL (9) 这两个信号**，所以这里你主要了解这两个信号就可以了，其他信号以后用到时再做介绍。

进程在收到信号后，就会去做相应的处理。怎么处理呢？对于每一个信号，进程对它的处理都有下面三个选择。

第一个选择是**忽略 (Ignore)**，就是对这个信号不做任何处理，但是有两个信号例外，对于 SIGKILL 和 SIGSTOP 这两个信号，进程是不能忽略的。这是因为它们的主要作用是为 Linux kernel 和超级用户提供删除任意进程的特权。

第二个选择，就是**捕获 (Catch)**，这个是指让用户进程可以注册自己针对这个信号的 handler。具体怎么做我们目前暂时涉及不到，你先知道就行，我们在后面课程会进行详细介绍。

对于捕获，SIGKILL 和 SIGSTOP 这两个信号也同样例外，这两个信号不能有用户自己的处理代码，只能执行系统的缺省行为。

还有一个选择是**缺省行为 (Default)**，Linux 为每个信号都定义了一个缺省的行为，你可以在 Linux 系统中运行 `man 7 signal`来查看每个信号的缺省行为。

对于大部分的信号而言，应用程序不需要注册自己的 handler，使用系统缺省定义行为就可以了。

进程处理信号的三种选择

忽略 (Ignore)	对信号不做任何处理，但 SIGKILL 和 SIGSTOP 例外。
捕获 (Catch)	让用户进程可以注册自己针对这个信号的 handler，但 SIGKILL 和 SIGSTOP 例外。
缺省 (Default)	Linux 为每个信号都定义了一个缺省的行为。对于大部分的信号，应用程序不需要注册自己的 handler，使用系统缺省定义行为即可。

我刚才说了，SIGTERM（15）和 SIGKILL（9）这两个信号是我们重点掌握的。现在已经讲解了信号的概念和处理方式，我就拿这两个信号为例，再带你具体分析一下。

首先我们来看 SIGTERM（15），这个信号是 Linux 命令 kill 缺省发出的。前面例子中的命令 kill 1，就是通过 kill 向 1 号进程发送一个信号，在没有别的参数时，这个信号类型就默认为 SIGTERM。

SIGTERM 这个信号是可以被捕获的，这里的“捕获”指的就是用户进程可以为这个信号注册自己的 handler，而这个 handler，我们后面会看到，它可以处理进程的 graceful-shutdown 问题。

我们再了解一下 SIGKILL（9），这个信号是 Linux 里两个**特权信号**之一。什么是特权信号呢？

前面我们已经提到过了，**特权信号就是 Linux 为 kernel 和超级用户去删除任意进程所保留的，不能被忽略也不能被捕获。**那么进程一旦收到 SIGKILL，就要退出。

在前面的例子中，我们运行的命令 kill -9 1 里的参数 “-9”，其实就是指发送编号为 9 的这个 SIGKILL 信号给 1 号进程。

现象解释


现在, 你应该理解 init 进程和 Linux 信号这两个概念了, 让我们回到开头的问题上来: “为什么我在容器中不能 kill 1 号进程, 甚至 SIGKILL 信号也不行?”

你还记得么, 在课程的最开始, 我们已经尝试过用 bash 作为容器 1 号进程, 这样是无法把 1 号进程杀掉的。那么我们再一起来看一看, 用别的编程语言写的 1 号进程是否也杀不掉。

我们现在用 **C 程序作为 init 进程**, 尝试一下杀掉 1 号进程。和 bash init 进程一样, 无论 SIGTERM 信号还是 SIGKILL 信号, 在容器里都不能杀死这个 1 号进程。

 复制代码

```
1 # cat c-init-nosig.c
2 #include <stdio.h>
3 #include <unistd.h>
4
5 int main(int argc, char *argv[])
6 {
7     printf("Process is sleeping\n");
8     while (1) {
9         sleep(100);
10    }
11
12    return 0;
13 }
```

 复制代码

```
1 # docker stop sig-proc;docker rm sig-proc
2 # docker run --name sig-proc -d registry/sig-proc:v1 /c-init-nosig
3 # docker exec -it sig-proc bash
4 [root@5d3d42a031b1 /]# ps -ef
5 UID          PID  PPID  C STIME TTY          TIME CMD
6 root           1     0  0 07:48 ?           00:00:00 /c-init-nosig
7 root           6     0  5 07:48 pts/0       00:00:00 bash
8 root          19     6  0 07:48 pts/0       00:00:00 ps -ef
9 [root@5d3d42a031b1 /]# kill 1
10 [root@5d3d42a031b1 /]# kill -9 1
11 [root@5d3d42a031b1 /]# ps -ef
12 UID          PID  PPID  C STIME TTY          TIME CMD
13 root           1     0  0 07:48 ?           00:00:00 /c-init-nosig
14 root           6     0  0 07:48 pts/0       00:00:00 bash
15 root          20     6  0 07:49 pts/0       00:00:00 ps -ef
```


我们是不是这样就可以得出结论——“容器里的 1 号进程，完全忽略了 SIGTERM 和 SIGKILL 信号了”呢？你先别着急，我们再拿其他语言试试。


接下来，我们用 **Golang 程序作为 1 号进程**，我们再在容器中执行 `kill -9 1` 和 `kill 1`。

这次，我们发现 `kill -9 1` 这个命令仍然不能杀死 1 号进程，也就是说，SIGKILL 信号和之前的两个测试一样不起作用。

但是，我们执行 `kill 1` 以后，SIGTERM 这个信号把 init 进程给杀了，容器退出了。

 复制代码

```
1 # cat go-init.go
2 package main
3
4 import (
5     "fmt"
6     "time"
7 )
8
9 func main() {
10     fmt.Println("Start app\n")
11     time.Sleep(time.Duration(1000000) * time.Millisecond)
12 }
```

 复制代码

```
1 # docker stop sig-proc;docker rm sig-proc
2 # docker run --name sig-proc -d registry/sig-proc:v1 /go-init
3 # docker exec -it sig-proc bash
4
5
6 [root@234a23aa597b /]# ps -ef
7 UID          PID  PPID  C STIME TTY          TIME CMD
8 root           1      0  1 08:04 ?           00:00:00 /go-init
9 root          10      0  9 08:04 pts/0       00:00:00 bash
10 root          23     10  0 08:04 pts/0       00:00:00 ps -ef
11 [root@234a23aa597b /]# kill -9 1
12 [root@234a23aa597b /]# kill 1
13 [root@234a23aa597b /]# [~]# docker ps
14 CONTAINER ID          IMAGE                COMMAND              CREATED
```

对于这个测试结果，你是不是反而觉得更加困惑了？

为什么使用不同程序，结果就不一样呢？接下来我们就看看 kill 命令下达之后，Linux 里究竟发生了什么事，我给你系统地梳理一下整个过程。

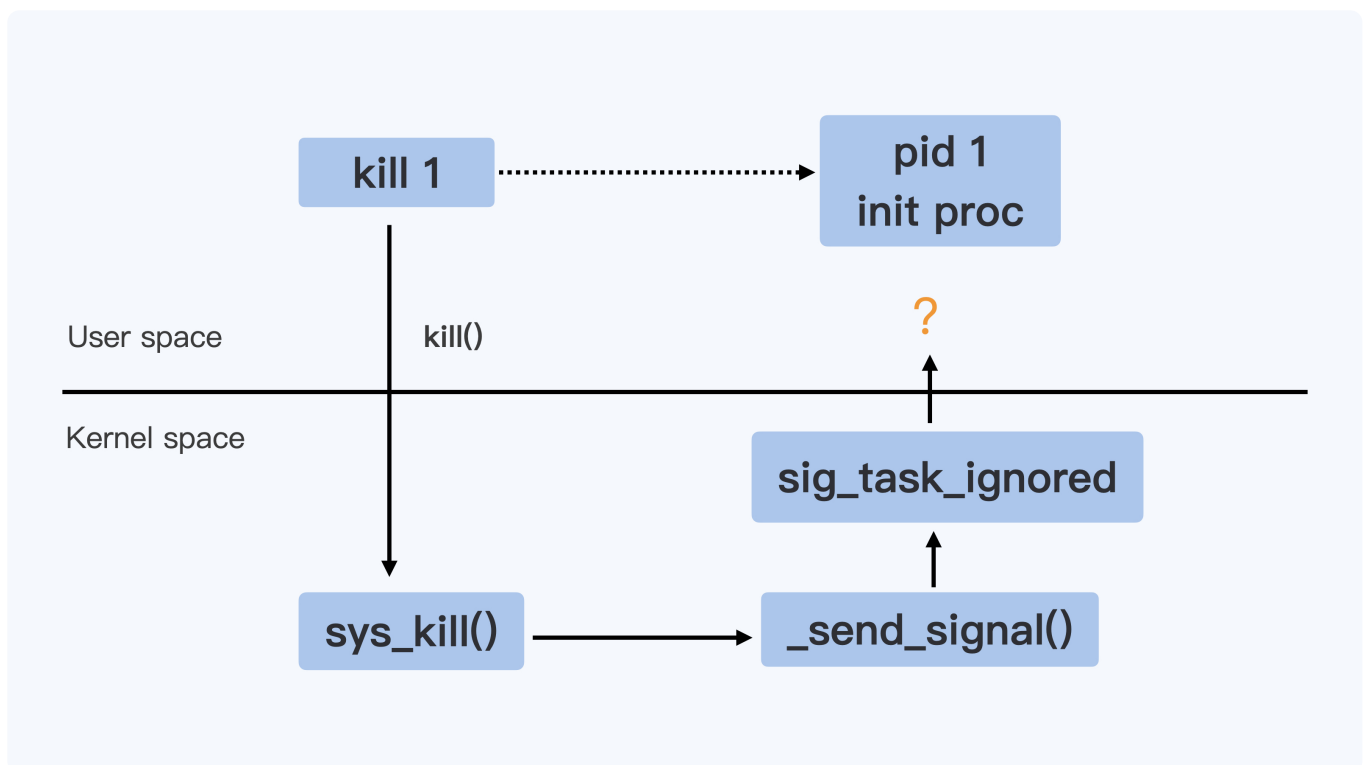
在我们运行 `kill 1` 这个命令的时候，希望把 SIGTERM 这个信号发送给 1 号进程，就像下面图里的**带箭头虚线**。

在 Linux 实现里，kill 命令调用了 **kill() 的这个系统调用**（所谓系统调用就是内核的调用接口）而进入到了内核函数 `sys_kill()`，也就是下图里的**实线箭头**。

而内核在决定把信号发送给 1 号进程的时候，会调用 `sig_task_ignored()` 这个函数来做判断，这个判断有什么用呢？

它会决定内核在哪些情况下会把发送的这个信号给忽略掉。如果信号被忽略了，那么 init 进程就不能收到指令了。

所以，我们想要知道 init 进程为什么收到或者收不到信号，都要去看看 **`sig_task_ignored()` 的这个内核函数的实现**。




sig_task_ignored()内核函数实现示意图

在 `sig_task_ignored()` 这个函数中有三个 `if{}判断`，第一个和第三个 `if{}判断` 和我们的问题没有关系，并且代码有注释，我们就不讨论了。

我们重点来看第二个 `if{}判断`。我来给你分析一下，在容器中执行 `kill 1` 或者 `kill -9 1` 的时候，这第二个 `if{}判断` 里的三个子条件是否可以被满足呢？

我们来看下面这串代码，这里表示**一旦这三个子条件都被满足，那么这个信号就不会发送给进程。**

 复制代码

```
1 kernel/signal.c
2 static bool sig_task_ignored(struct task_struct *t, int sig, bool force)
3 {
4     void __user *handler;
5     handler = sig_handler(t, sig);
6
7     /* SIGKILL and SIGSTOP may not be sent to the global init */
8     if (unlikely(is_global_init(t) && sig_kernel_only(sig)))
9
10         return true;
11
12     if (unlikely(t->signal->flags & SIGNAL_UNKILLABLE) &&
13         handler == SIG_DFL && !(force && sig_kernel_only(sig)))
14         return true;
15
16     /* Only allow kernel generated signals to this kthread */
17     if (unlikely((t->flags & PF_KTHREAD) &&
18         (handler == SIG_KTHREAD_KERNEL) && !force))
19         return true;
20
21     return sig_handler_ignored(handler, sig);
22 }
23
```

接下来，我们就逐一分析一下这三个子条件，我们来说说这个"`!(force && sig_kernel_only(sig))`"。

第一个条件里 `force` 的值，对于同一个 Namespace 里发出的信号来说，调用值是 0，所以这个条件总是满足的。

我们再来看一下第二个条件 “`handler == SIG_DFL`”，第二个条件判断信号的 handler 是否是 `SIG_DFL`。

那么什么是 `SIG_DFL` 呢？**对于每个信号，用户进程如果不注册一个自己的 handler，就会有一个系统缺省的 handler，这个缺省的 handler 就叫作 `SIG_DFL`。**


对于 `SIGKILL`，我们前面介绍过它是特权信号，是不允许被捕获的，所以它的 handler 就一直是 `SIG_DFL`。这第二个条件对 `SIGKILL` 来说总是满足的。

对于 `SIGTERM`，它是可以被捕获的。也就是说如果用户不注册 handler，那么这个条件对 `SIGTERM` 也是满足的。

最后再来看一下第三个条件，“`t->signal->flags & SIGNAL_UNKILLABLE`”，这里的条件判断是这样的，进程必须是 `SIGNAL_UNKILLABLE` 的。

这个 `SIGNAL_UNKILLABLE` flag 是在哪里置位的呢？

可以参考我们下面的这段代码，在每个 Namespace 的 init 进程建立的时候，就会打上 **`SIGNAL_UNKILLABLE`** 这个标签，也就是说只要是 1 号进程，就会有这个 flag，这个条件也是满足的。

 复制代码

```
1 kernel/fork.c
2             if (is_child_reaper(pid)) {
3                 ns_of_pid(pid)->child_reaper = p;
4                 p->signal->flags |= SIGNAL_UNKILLABLE;
5             }
6
7  /*
8   * is_child_reaper returns true if the pid is the init process
9   * of the current namespace. As this one could be checked before
10  * pid_ns->child_reaper is assigned in copy_process, we check
11  * with the pid number.
12  */
13
14 static inline bool is_child_reaper(struct pid *pid)
15 {
16     return pid->numbers[pid->level].nr == 1;
17 }
```

我们可以看出来，其实**最关键的一点就是 `handler == SIG_DFL`**。Linux 内核针对每个 **Nnamespace** 里的 **init** 进程，把只有 **default handler** 的信号都给忽略了。

如果我们自己注册了信号的 handler（应用程序注册信号 handler 被称作"Catch the Signal"），那么这个信号 handler 就不再是 `SIG_DFL`。即使是 `init` 进程在接收到 `SIGTERM` 之后也是可以退出的。

不过，由于 `SIGKILL` 是一个特例，因为 `SIGKILL` 是不允许被注册用户 handler 的（还有一个不允许注册用户 handler 的信号是 `SIGSTOP`），那么它只有 `SIG_DFL` handler。

所以 `init` 进程是永远不能被 `SIGKILL` 所杀，但是可以被 `SIGTERM` 杀死。

说到这里，我们该怎么证实这一点呢？我们可以做下面两件事来验证。


第一件事，你可以查看 1 号进程状态中 `SigCgt` Bitmap。

我们可以看到，在 Golang 程序里，很多信号都注册了自己的 handler，当然也包括了 `SIGTERM(15)`，也就是 bit 15。

而 C 程序里，缺省状态下，一个信号 handler 都没有注册；bash 程序里注册了两个 handler，bit 2 和 bit 17，也就是 `SIGINT` 和 `SIGCHLD`，但是没有注册 `SIGTERM`。

所以，C 程序和 bash 程序里 `SIGTERM` 的 handler 是 `SIG_DFL`（系统缺省行为），那么它们就不能被 `SIGTERM` 所杀。

具体我们可以看一下这段 `/proc` 系统的进程状态：

 复制代码

```
1  ### golang init
2  # cat /proc/1/status | grep -i SigCgt
3  SigCgt:      ffffffff7fc1feff
4
5  ### C init
6  # cat /proc/1/status | grep -i SigCgt
7  SigCgt:      0000000000000000
8
9  ### bash init
10 # cat /proc/1/status | grep -i SigCgt
```

```
11 SigCgt:      00000000000010002
12
```

第二件事，给 C 程序注册一下 SIGTERM handler，捕获 SIGTERM。

我们调用 `signal()` 系统调用注册 SIGTERM 的 handler，在 handler 里主动退出，再看看容器中 `kill 1` 的结果。

这次我们就可以看到，在进程状态的 SigCgt bitmap 里，bit 15 (SIGTERM) 已经置位了。同时，运行 `kill 1` 也可以把这个 C 程序的 init 进程给杀死了。

[复制代码](#)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include <unistd.h>
6
7 void sig_handler(int signo)
8 {
9     if (signo == SIGTERM) {
10         printf("received SIGTERM\n");
11         exit(0);
12     }
13 }
14
15 int main(int argc, char *argv[])
16 {
17     signal(SIGTERM, sig_handler);
18
19     printf("Process is sleeping\n");
20     while (1) {
21         sleep(100);
22     }
23     return 0;
24 }
```

[复制代码](#)

```
1 # docker stop sig-proc;docker rm sig-proc
2 # docker run --name sig-proc -d registry/sig-proc:v1 /c-init-sig
3 # docker exec -it sig-proc bash
4 [root@043f4f717cb5 /]# ps -ef
5 UID          PID   PPID    C  STIME TTY          TIME CMD
```

```
6 root          1          0   0 09:05 ?           00:00:00 /c-init-sig
7 root          6          0  18 09:06 pts/0       00:00:00 bash
8 root         19          6   0 09:06 pts/0       00:00:00 ps -ef
9
10 [root@043f4f717cb5 /]# cat /proc/1/status | grep SigCgt
11 SigCgt: 000000000000004000
12 [root@043f4f717cb5 /]# kill 1
13 # docker ps
14 CONTAINER ID          IMAGE                                COMMAND                                CREATED
```

好了，到这里我们可以确定这两点：

1. `kill -9 1` 在容器中是不工作的，内核阻止了 1 号进程对 SIGKILL 特权信号的响应。
2. `kill 1` 分两种情况，如果 1 号进程没有注册 SIGTERM 的 handler，那么对 SIGTERM 信号也不响应，如果注册了 handler，那么就可以响应 SIGTERM 信号。

重点总结

好了，今天的内容讲完了。我们来总结一下。

这一讲我们主要讲了 init 进程。围绕这个知识点，我提出了一个真实发生的问题：“为什么我在容器中不能 kill 1 号进程？”。

想要解决这个问题，我们需要掌握两个基本概念。

第一个概念是 Linux 1 号进程。它是第一个用户态的进程。它直接或者间接创建了 Namespace 中的其他进程。

第二个概念是 Linux 信号。Linux 有 31 个基本信号，进程在处理大部分信号时有三个选择：忽略、捕获和缺省行为。其中两个特权信号 SIGKILL 和 SIGSTOP 不能被忽略或者捕获。

只知道基本概念还不行，我们还要去解决问题。我带你尝试了用 bash, C 语言还有 Golang 程序作为容器 init 进程，发现它们对 kill 1 的反应是不同的。

因为信号的最终处理都是在 Linux 内核中进行的，因此，我们需要对 Linux 内核代码进行分析。

容器里 1 号进程对信号处理的两个要点，这也是这一讲里我想让你记住的两句话：

1. 在容器中，1 号进程永远不会响应 SIGKILL 和 SIGSTOP 这两个特权信号；
2. 对于其他的信号，如果用户自己注册了 handler，1 号进程可以响应。

思考题

这一讲的最开始，有这样一个 C 语言的 init 进程，它没有注册任何信号的 handler。如果我们从 Host Namespace 向它发送 SIGTERM，会发生什么情况呢？

欢迎留言和我分享你的想法。如果你的朋友也对 1 号进程有困惑，欢迎你把这篇文章分享给他，说不定就帮他解决了一个难题。

提建议

© 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 01 | 认识容器：容器的基本操作和实现原理

下一篇 03 | 理解进程（2）：为什么我的容器里有这么多僵尸进程？

精选留言 (36)

写留言



Helios

2020-11-19

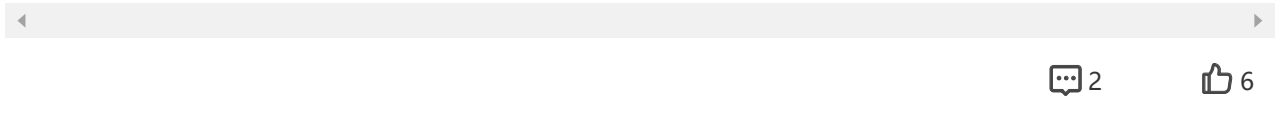
有一种老师说了一大圈，但是没有说容器的本质就是宿主机上的一个进程这个本质。

作者回复: @Helios,
很好的一个问题。

很多介绍容器的文章可能都会强调容器是进程，不过它们讨论的背景应该是和虚拟机做比较之后这么说的，因为在容器之前虚拟机是云平台上最流行的技术。强调容器是进程的目的是区分容器与虚拟机的差别，但是我不认为这个是容器的本质。

其实无论是namespace (pid namespace)还是cgroups都是在管理进程， 容器中运行是进程， 这个是个明显的特征了， 但不是本质。

我们如果换一个角度去思考， 如果容器流行先于虚拟机技术， 我们是否还会强调容器是进程了呢？

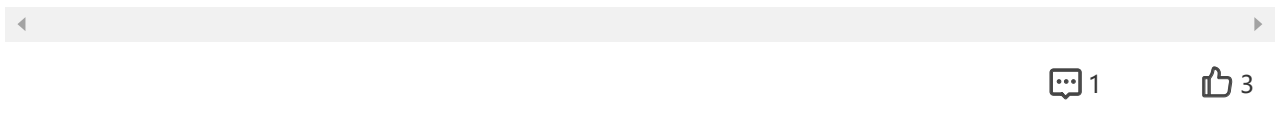
**1900**

2020-11-19

在容器中不能响应SIGKILL 和 SIGSTOP，但是在宿主机中可以响应，因为在宿主机中所看到的“容器1号进程”在宿主机上只是一个普通进程

展开 ∨

作者回复: @1900, 如果可以的话，你可以动手试一下，看看结果是不是和你分析的一样 :-)

**赵守忠[开心每一天]**

2020-11-21

kill 1 分两种情况，如果 1 号进程没有注册 SIGTERM 的 handler，那么对 SIGTERM 信号也不响应，如果注册了 handler，那么就可以响应 SIGTERM 信号。

---在k8s的容器环境内测试，基于tini。和老师讲的有些出入：

```
bash-5.0# ps -ef
```

```
PID USER TIME COMMAND...
```

展开 ∨

作者回复: @赵守忠[开心每一天]

非常好的发现啊！

tini的确没有注册SIGTERM，它的退出并不是因为SIGTERM的信号让它退出的，而是它发现它子进程都退出之后，主动退出的，这样容器也退出了。

可以看一下tini的源代码，它把所有接收到的信号(除了SIGHILD)都转发给了子进程，也包括了SIGTERM, 那么子进程收到SIGTERM就退出了，而tini自己可以收到SIGHILD, 然后tini自己退出，并且容器退出。

tini 的源代码和主循环：

<https://github.com/krallin/tini/blob/master/src/tini.c>

```
while (1) {
```

```
/* Wait for one signal, and forward it */
if (wait_and_forward_signal(&parent_sigset, child_pid)) {
    return 1;
}

/* Now, reap zombies */
if (reap_zombies(child_pid, &child_exitcode)) {
    return 1;
}

if (child_exitcode != -1) {
    PRINT_TRACE("Exiting: child has exited");
    return child_exitcode;
}
}
```

 1 2**朱雯**

2020-11-19

关于思考题：

这一讲的最开始，有这样一个 C 语言的 init 进程，它没有注册任何信号的 handler。如果我们从 Host Namespace 向它发送 SIGTERM，会发生什么情况呢？

啥叫从host namespace向他发送sigterm，这是啥意思，是宿主机对他发送sigterm吗，宿主机发送，那就直接把他杀了，不仅法sigterm会杀，发kill也多半会杀，因为在宿主...
展开

作者回复: > 啥叫从host namespace向他发送sigterm，这是啥意思，是宿主机对他发送sigterm吗

是的，在宿主机上的缺省namespace是host namespace.

> 不同的namespace，force不一定为0

是的，signal sender不在同一个namespace的时候，force不为0.

> 我的问题在于SIGNAL_UNKILLABLE 标签还会不会打上，打上以后是对宿主机这个标签也生效吗。

进程创建后这个标签是一直有的，只是pid在容器namespace里看到的是1，而在宿主机的namespace里是另外一个pid值

 2

**Geek_71d4ac**

2020-11-18

关于特权信号的那一段表述，我觉得是有问题的。当任务处于task uninterrupt状态时，是不能接收任何信号的。

展开 ▾

作者回复: @Geek_71d4ac

你说的没错，D state (uninterruptible) 进程 还有 Zombie进程都是不能接受任何信号的。我在后面的章节里还有介绍。



👍 2

**上邪忘川**

2020-11-21

从网上找到了不错的关于SigCgt 掩码位的解释，不懂的可以看一下，豁然开朗。<https://qastack.cn/unix/85364/how-can-i-check-what-signals-a-process-is-listening-to>

作者回复: 谢谢 @上邪忘川，很好的补充材料！



👍 1

**良凯尔**

2020-11-20

虽然在容器内kill 1号进程行不通，但是我可以在宿主上kill容器的1号进程来达到重启容器的目的，是这样吗？

作者回复: 在宿主机上kill容器的1号进程是可以的。不过，有时候容器的用户没有宿主机登陆的权限。



👍 1

**JianXu**

2020-11-19

```
if (unlikely(t->signal->flags & SIGNAL_UNKILLABLE) && handler == SIG_DFL && !
(force && sig_kernel_only(sig))) { return true;}
```

老师, 我是不是可以这样理解，在容器内部的时候对于没有安装SIGTERM handler的情况下，force=0 并且SIGNAL_UNKILLABLE 也是置位的，所以这个if 语句返回真，所以SI...

展开 ✓

作者回复: @Jlanxu,

> 因为不是宿主机上的第一个进程所以 UNKILLABLE 也没有置位

这个SIGALRM_UNKILLABLE 是目标进程（容器里的1号进程）在创建的时候置位的，所以无论发送信号的进程在容器namespace还是在host namespace, 这个flag都是存在的。

> 能不能进一步介绍一下为什么在Kernel 里面要放置这三个 if 语句来 ignore signal 呢？

这里可以看一下这段代码最初check-in的comments. 我的理解是如果1号进程被杀，会是整个系统处于混乱并且难调试的状态，我们要尽量的避免这种情况。

commit 86989c41b5ea08776c450cb759592532314a4ed6

Author: Eric W. Biederman <ebiederm@xmission.com>

Date: Thu Jul 19 19:47:27 2018 -0500

signal: Always ignore SIGKILL and SIGSTOP sent to the global init

If the first process started (aka /sbin/init) receives a SIGKILL it will panic the system if it is delivered. Making the system unusable and undebugable. It isn't much better if the first process started receives SIGSTOP.

So always ignore SIGSTOP and SIGKILL sent to init.

This is done in a separate clause in sig_task_ignored as force_sig_info can clear SIG_UNKILLABLE and this protection should work even then.

Reviewed-by: Thomas Gleixner <tglx@linutronix.de>

Signed-off-by: "Eric W. Biederman" <ebiederm@xmission.com>

◀



Helios

2020-11-19

不知道什么要在容器内执行，直接去宿主机上docker kill不行么。或者直接edit一下编排文件加个环境变量啥的，不就能触发原地升级么。

比较新的信号应该不止31个了，还增加了31个可靠信号，为了解决以前linux中信号堆积忽

略信号的问题。

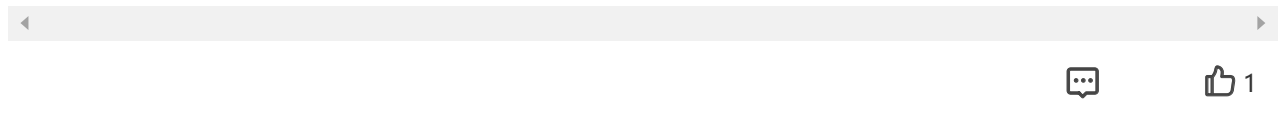
展开 ∨

作者回复: > 不知道什么要在容器内执行，直接在去宿主机上docker kill不行么。或者直接edit一下编排文件加个环境变量啥的，不就能触发原地升级么。

在生产环境中，用户是没有权限登陆宿主机的。在Pod spec部分，runtime允许修改的只有 container image了。

> 比较新的信号应该不止31个了，还增加了31个可靠信号，为了解决以前linux中信号堆积忽略信号的问题。

你说的没错，还有新的 32-64 可靠信号。因为和这一讲的问题没有太大关系，在这里就不展开了。



无名

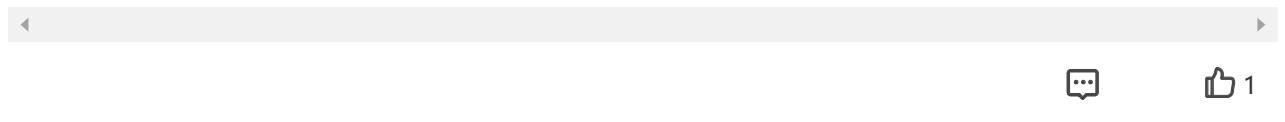
2020-11-19

man pid_namespace 提到了老师在文中强调的两个细节。

Only signals for which the "init" process has established a signal...

展开 ∨

作者回复: 谢谢 @无名!
仔细读文档也是很有用的!



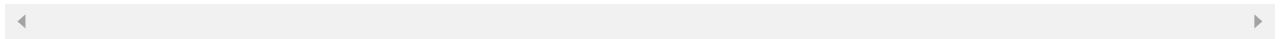
争光 Alan

2020-11-19

能不能讲完后，讲下社区的最佳实践，比如docker现在提供了docker run --init参数避免这个问题，内核层面有没有相关的优化跟进

展开 ∨

作者回复: @争光Alan, 谢谢，很好的建议!
我在这一章的最后，也拿了tini做了example来做一些简单best practice的说明。

**朱雯**

2020-11-19

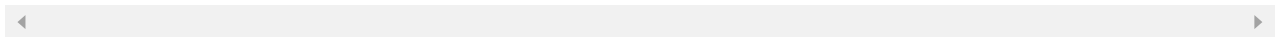
关于SigCgt bitmap 其实我是有些疑惑的，第一为什么是16位？我最开始是这样猜测的，就是一个位置代表一个信号量，那最多只能说明第一到第十六的信号量。后面看到加了handle处理的函数是这个样子

```
if (signo == SIGTERM) { printf("received SIGTERM\n"); exit(0); }...
```

展开 ∨

作者回复: @朱雯

很好的问题！我在文中只是说了31个posix标准里的signal. Linux 里还有编号32-64的real-time signal的扩展定义。因为和这一讲的问题没有太大的关系，我没有具体展开了。

**维c**

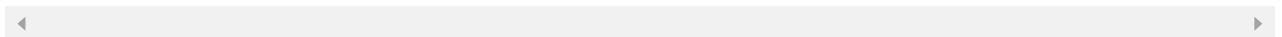
2020-12-03

查了一下资料，貌似sig_kernel_only函数是用了判断信号是不是kill或者stop的，是这两个信号才会返回true，这就意味着force不为0，同时信号是kill或者stop的时候信号是不会被忽略的，这也就解释了为什么宿主机是可以通过kill信号来杀掉容器里的进程，而sigterm由于force的值可能会被忽略，那么force的值又是又什么决定的呢？

展开 ∨

作者回复: @维c

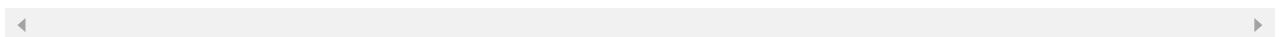
很好的分析！force 由发送信号的进程和接受信号进程是否在同一个namespace里决定。你可以再看一下代码。

**夜空中最亮的星 (华仔...)**

2020-12-01

讲解的很深入，这是我学到讲解最深的，跟着老师好好学

作者回复: 谢谢，华仔！



**daydreamer**

2020-11-28

思考题：

kill <pid> 不可以杀掉容器init进程

kill -9 <pid> 可以

不同点在于SIGTERM不是内核信号，所以!(force && sig_kernel_only(sig))为True，加上前面两个if也为true，所以忽略；SIGKILL是内核信号 !(force && sig_kernel_only(sig))...
展开 ∨

**po**

2020-11-25

老师，我做了个测试，我发现kill 1杀不掉进程，kill -9 1把进程杀了，步骤如下：

1. 首先启动一个容器，使用sleep命令启动，然后使用kill去杀：

```
[root@254fb2b3843a /]# ps -ef
```

```
UID PID PPID C STIME TTY TIME CMD...
```

展开 ∨

作者回复: @po,

- 对于 "kill 1"，你可以看一下 "cat /proc/1/status | grep SigCgt"吗？ Bit 15 (SIGTERM) 上的值应该是0.

- 对于"kill -9 1"的问题，你可以share 一下你的image 和 docker启动命令吗？



2

**王小飞**

2020-11-23

1. /sys/fs/cgroup/memory/docker/[container_id]/cgroups.proc 可以查看当前容器内所有进程的 ID

2. 宿主机上 kill -9 [进程号] 可以杀掉容器的 1 号进程。

作者回复: 对的宿主机上kill -9去杀容器的1 号进程没有问题，那么用kill <pid> (SIGTERM)呢？

**思维决定未来**

2020-11-23

回到这一节最前面的场景，我也有同样的问题，如何在pod不退出的情况下重载容器，不是重启，重启即使pod还在，对应容器也有一次restart次数的

展开 ▾

**谢哈哈**

2020-11-22

已在linux下确认C与GO的程序，如果不注册默认的handler，在host namespace空间对容器里的C只响应kill -9，而容器里的GO都可以响应

展开 ▾

作者回复: @谢哈哈

是这样的！

> 如果不注册默认的handler，在host namespace空间对容器里的C只响应kill -9

可以再想想，为什么这种情况，在host namespace还是不能相应SIGTERM？



1

**closer**

2020-11-20

我觉得思考题的答案是这样的 从宿主机找到docker 映射到物理机的进程pid 直接 kill -9 就行

作者回复: 对的，SIGKILL肯定可以。那么SIGTERM呢？

