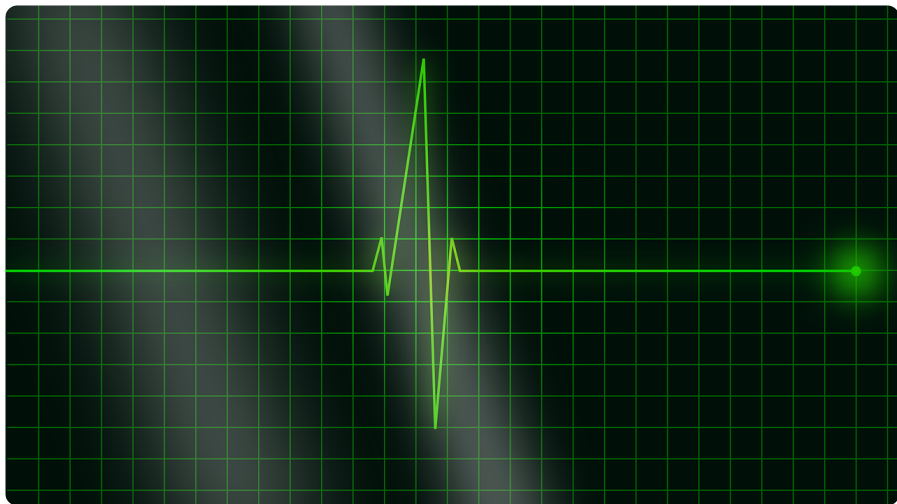


## 37 | 信号（上）：项目组A完成了，如何及时通知项目组B？

2019-06-21 刘超

趣谈Linux操作系统

[进入课程 >](#)



**讲述：刘超**

时长 12:03 大小 11.04M



上一节最后，我们讲了信号的机制。在某些紧急情况下，我们需要给进程发送一个信号，紧急处理一些事情。


这种方式有点儿像咱们运维一个线上系统，为了应对一些突发事件，往往需要制定应急预案。就像下面的列表中一样。一旦发生了突发事件，马上能够找到负责人，根据处理步骤进行紧急响应，并且在限定的事件内搞定。

ID	故障	产生现象	影响	处理步骤	负责人	是否演练	响应时间
1	管控服务异常						
2	MQ故障						
3	数据库故障						
4	计算节点宕机						
5	云主机资源池容量耗尽						
...	...						

我们现在就按照应急预案的设计思路，来看一看 Linux 信号系统的机制。

首先，第一件要做的事情就是，整个团队要想一下，线上到底能够产生哪些异常情况，越全越好。于是，我们就有了上面这个很长很长的列表。

在 Linux 操作系统中，为了响应各种各样的事件，也是定义了非常多的信号。我们可以通过 `kill -l` 命令，查看所有的信号。


 复制代码

```
1 # kill -l
```

```
2  1) SIGHUP          2) SIGINT          3) SIGQUIT         4) SIG
3  6) SIGABRT         7) SIGBUS          8) SIGFPE          9) SIG
4 11) SIGSEGV        12) SIGUSR2        13) SIGPIPE        14) SIG
5 16) SIGSTKFLT      17) SIGCHLD        18) SIGCONT        19) SIG
6 21) SIGTTIN        22) SIGTTOU        23) SIGURG         24) SIG
7 26) SIGVTALRM      27) SIGPROF        28) SIGWINCH       29) SIG
8 31) SIGSYS         34) SIGRTMIN       35) SIGRTMIN+1     36) SIG
9 38) SIGRTMIN+4     39) SIGRTMIN+5     40) SIGRTMIN+6     41) SIG
10 43) SIGRTMIN+9     44) SIGRTMIN+10    45) SIGRTMIN+11    46) SIG
11 48) SIGRTMIN+14    49) SIGRTMIN+15    50) SIGRTMAX-14    51) SIG
12 53) SIGRTMAX-11    54) SIGRTMAX-10    55) SIGRTMAX-9     56) SIG
13 58) SIGRTMAX-6     59) SIGRTMAX-5     60) SIGRTMAX-4     61) SIG
14 63) SIGRTMAX-1     64) SIGRTMAX
```



这些信号都是什么作用呢？我们可以通过 `man 7 signal` 命令查看，里面会有一个列表。

 复制代码

1	Signal	Value	Action	Comment
2				
3	SIGHUP	1	Term	Hangup detected on contrc
4				or death of controlling p
5	SIGINT	2	Term	Interrupt from keyboard
6	SIGQUIT	3	Core	Quit from keyboard
7	SIGILL	4	Core	Illegal Instruction
8				
9				
10	SIGABRT	6	Core	Abort signal from abort(3
11	SIGFPE	8	Core	Floating point exception
12	SIGKILL	9	Term	Kill signal

13	SIGSEGV	11	Core	Invalid memory reference
14	SIGPIPE	13	Term	Broken pipe: write to pipe readers
15				
16	SIGALRM	14	Term	Timer signal from alarm(2)
17	SIGTERM	15	Term	Termination signal
18	SIGUSR1	30,10,16	Term	User-defined signal 1
19	SIGUSR2	31,12,17	Term	User-defined signal 2
20	.....			



就像应急预案里面给出的一样，每个信号都有一个唯一的ID，还有遇到这个信号的时候的默认操作。

一旦有信号产生，我们就有下面这几种，用户进程对信号的处理方式。


**1.执行默认操作。** Linux 对每种信号都规定了默认操作，例如，上面列表中的 Term，就是终止进程的意思。Core 的意思是 Core Dump，也即终止进程后，通过 Core Dump 将当前进程的运行状态保存在文件里面，方便程序员事后进行分析问题在哪里。

**2.捕捉信号。** 我们可以为信号定义一个信号处理函数。当信号发生时，我们就执行相应的信号处理函数。

**3.忽略信号。**当我们不希望处理某些信号的时候，就可以忽略该信号，不做任何处理。有两个信号是应用进程无法捕捉和忽略的，即 SIGKILL 和 SEGSTOP，它们用于在任何时候中断或结束某一进程。

接下来，我们来看一下信号处理最常见的流程。这个过程主要是分成两步，第一步是注册信号处理函数。第二步是发送信号。这一节我们主要看第一步。


如果我们不想让某个信号执行默认操作，一种方法就是对特定的信号注册相应的信号处理函数，设置信号处理方式的是 **signal 函数**。

 复制代码

```
1 typedef void (*sighandler_t)(int);  
2 sighandler_t signal(int sigum, sighandler_t handler);
```


这其实就是定义一个方法，并且将这个方法 and 某个信号关联起来。当这个进程遇到这个信号的时候，就执行这个方法。

如果我们在 Linux 下面执行 man signal 的话，会发现 Linux 不建议我们直接用这个方法，而是改用 sigaction。定义如下：

 复制代码

```
1 int sigaction(int signum, const struct sigaction *act,  
2               struct sigaction *oldact);
```


这两者的区别在哪里呢？其实它还是将信号和一个动作进行关联，只不过这个动作由一个结构 `struct sigaction` 表示了。

 复制代码

```
1 struct sigaction {  
2     __sighandler_t sa_handler;  
3     unsigned long sa_flags;  
4     __sigrestore_t sa_restorer;  
5     sigset_t sa_mask;           /* mask last fc  
6 };
```

和 `signal` 类似的是，这里面还是有 `__sighandler_t`。但是，其他成员变量可以让你更加细致地控制信号处理的行为。而 `signal` 函数没有给你机会设置这些。这里需要注意的是，`signal` 不是系统调用，而是 `glibc` 封装的一个函数。这样就像 `man signal` 里面写的一样，不同的实现方式，设置的参数会不同，会导致行为的不同。

例如，我们在 glibc 里面会看到了这样一个实现：

 复制代码

```
1 # define signal __sysv_signal
2 __sighandler_t
3 __sysv_signal (int sig, __sighandler_t handler)
4 {
5     struct sigaction act, oact;
6     .....
7     act.sa_handler = handler;
8     __sigemptyset (&act.sa_mask);
9     act.sa_flags = SA_ONESHOT | SA_NOMASK | SA_INTERRUPT;
10    act.sa_flags &= ~SA_RESTART;
11    if (__sigaction (sig, &act, &oact) < 0)
12        return SIG_ERR;
13    return oact.sa_handler;
14 }
15 weak_alias (__sysv_signal, sysv_signal)
```

在这里面，sa\_flags 进行了默认的设置。SA\_ONESHOT 是什么意思呢？意思就是，这里设置的信号处理函数，仅仅起作用一次。用完了一次后，就设置回默认行为。这其实并不是我们想看到的。毕竟我们一旦安装了一个信号处理函数，肯定希望它一直起作用，直到我显式地关闭它。

另外一个设置就是**SA\_NOMASK**。我们通过 \_\_sigemptyset，将 sa\_mask 设置为空。这样的设置表示在

这个信号处理函数执行过程中，如果再有其他信号，哪怕相同的信号到来的时候，这个信号处理函数会被中断。如果一个信号处理函数真的被其他信号中断，其实问题也不大，因为当处理完了其他的信号处理函数后，还会回来接着处理这个信号处理函数的，但是对于相同的信号就有点尴尬了，这就需要这个信号处理函数写的比较有技巧了。

例如，对于这个信号的处理过程中，要操作某个数据结构，因为是相同的信号，很可能操作的是同一个实例，这样的话，同步、死锁这些都要想好。其实一般的思路应该是，当某一个信号的信号处理函数运行的时候，我们暂时屏蔽这个信号。后面我们还会仔细分析屏蔽这个动作，屏蔽并不意味着信号一定丢失，而是暂存，这样能够做到信号处理函数对于相同的信号，处理完一个再处理下一个，这样信号处理函数的逻辑要简单得多。

还有一个设置就是设置了**SA\_INTERRUPT**，清除了**SA\_RESTART**。这是什么意思呢？我们知道，信号的到来时间是不可预期的，有可能程序正在调用某个漫长的系统调用的时候（你可以在一台 Linux 机器上运行 `man 7 signal` 命令，在这里找 `Interruption of system calls and library functions by signal handlers` 的部分，里面说的非常详细），这个时候一个信号来了，会中断这个系统调用，去执行信号处理函数，那执行完了以后呢？系统调用怎么办呢？




这时候有两种处理方法，一种就是 `SA_INTERRUPT`，也即系统调用被中断了，就不再重试这个系统调用了，而是直接返回一个 `-EINTR` 常量，告诉调用方，这个系统调用被信号中断了，但是怎么处理你看着办。如果是这样的话，调用方可以根据自己的逻辑，重新调用或者直接返回，这会使得我们的代码非常复杂，在所有系统调用的返回值判断里面，都要特殊判断一下这个值。

另外一种处理方法是 `SA_RESTART`。这个时候系统调用会被自动重新启动，不需要调用方自己写代码。当然也可能存在问题，例如从终端读入一个字符，这个时候用户在终端输入一个 `'a'` 字符，在处理 `'a'` 字符的时候被信号中断了，等信号处理完毕，再次读入一个字符的时候，如果用户不再输入，就停在那里了，需要用户再次输入同一个字符。

因而，建议你使用 `sigaction` 函数，根据自己的需要定制参数。


接下来，我们来看 `sigaction` 具体做了些什么。

还记得在学习系统调用那一节的时候，我们知道，`glibc` 里面有个文件 `syscalls.list`。这里面定义了库函数调用哪些系统调用，在这里我们找到了 `sigaction`。

 复制代码

```
1 sigaction      -      sigaction      i:ipp  __sigactic
```

接下来，在 glibc 中，\_\_sigaction 会调用 \_\_libc\_sigaction，并最终调用的系统调用是 rt\_sigaction。

 复制代码

```
1 int
2 __sigaction (int sig, const struct sigaction *act, stru
3 {
4 .....
5     return __libc_sigaction (sig, act, oact);
6 }
7
8
9 int
10 __libc_sigaction (int sig, const struct sigaction *act,
11 {
12     int result;
13     struct kernel_sigaction kact, koact;
14
15
16     if (act)
17     {
18         kact.k_sa_handler = act->sa_handler;
19         memcpy (&kact.sa_mask, &act->sa_mask, sizeof (sig
20         kact.sa_flags = act->sa_flags | SA_RESTORER;
21
22
23         kact.sa_restorer = &restore_rt;
```

```

24     }
25
26
27     result = INLINE_SYSCALL (rt_sigaction, 4,
28                             sig, act ? &kact : NULL,
29                             oact ? &koact : NULL, _NSIG
30
31     if (oact && result >= 0)
32     {
33         oact->sa_handler = koact.k_sa_handler;
34         memcpy (&oact->sa_mask, &koact.sa_mask, sizeof (s
35         oact->sa_flags = koact.sa_flags;
36         oact->sa_restorer = koact.sa_restorer;
37     }
38     return result;
39 }

```

这也是很多人看信号处理的内核实现的时候，比较困惑的地方。例如，内核代码注释里面会说，系统调用 `signal` 是为了兼容过去，系统调用 `sigaction` 也是为了兼容过去，连参数都变成了 `struct compat_old_sigaction`，所以说，我们的库函数虽然调用的是 `sigaction`，到了系统调用层，调用的可不是系统调用 `sigaction`，而是系统调用 `rt_sigaction`。

```
1 SYSCALL_DEFINE4(rt_sigaction, int, sig,  
2         const struct sigaction __user *, act,  
3         struct sigaction  user *, oact,
```

```

4             size_t, sigsetsize)
5 {
6     struct k_sigaction new_sa, old_sa;
7     int ret = -EINVAL;
8     .....
9     if (act) {
10         if (copy_from_user(&new_sa.sa, act, siz
11             return -EFAULT;
12     }
13
14
15     ret = do_sigaction(sig, act ? &new_sa : NULL, c
16
17
18     if (!ret && oact) {
19         if (copy_to_user(oact, &old_sa.sa, size
20             return -EFAULT;
21     }
22 out:
23     return ret;
24 }

```

在 `rt_sigaction` 里面，我们将用户态的 `struct sigaction` 结构，拷贝为内核态的 `k_sigaction`，然后调用 `do_sigaction`。`do_sigaction` 也很简单，还记得进程内核的数据结构里，`struct task_struct` 里面有一个成员 `sighand`，里面有一个 `action`。这是一个数组，下标是信号，内容就是信号处理函数，`do_sigaction` 就是设置 `sighand` 里的信号处理函数。

```
1 int do_sigaction(int sig, struct k_sigaction *act, stru
2 {
3     struct task_struct *p = current, *t;
4     struct k_sigaction *k;
5     sigset_t mask;
6     .....
7     k = &p->sighand->action[sig-1];
8
9
10    spin_lock_irq(&p->sighand->siglock);
11    if (oact)
12        *oact = *k;
13
14
15    if (act) {
16        sigdelsetmask(&act->sa.sa_mask,
17                      sigmask(SIGKILL) | sigmas
18        *k = *act;
19    .....
20    }
21
22
23    spin_unlock_irq(&p->sighand->siglock);
24    return 0;
25 }
```

至此，信号处理函数的注册已经完成了。

## 总结时刻

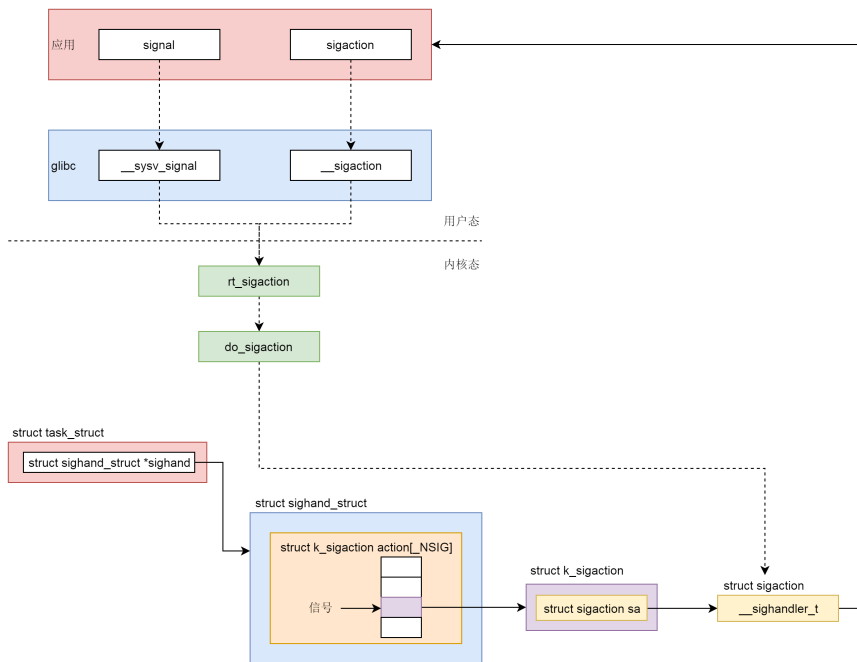
这一节讲了如何通过 API 注册一个信号处理函数，整个过程如下图所示。

在用户程序里面，有两个函数可以调用，一个是 `signal`，一个是 `sigaction`，推荐使用 `sigaction`。

用户程序调用的是 Glibc 里面的函数，`signal` 调用的是 `__sysv_signal`，里面默认设置了一些参数，使得 `signal` 的功能受到了限制，`sigaction` 调用的是 `__sigaction`，参数用户可以任意设定。

无论是 `__sysv_signal` 还是 `__sigaction`，调用的都是统一的一个系统调用 `rt_sigaction`。

在内核中，`rt_sigaction` 调用的是 `do_sigaction` 设置信号处理函数。在每一个进程的 `task_struct` 里面，都有一个 `sighand` 指向 `struct sighand_struct`，里面是一个数组，下标是信号，里面的内容是信号处理函数。



## 课堂练习

你可以试着写一个程序，调用 `sigaction` 为某个信号设置一个信号处理函数，在信号处理函数中，如果收到信号则打印一些字符串，然后用命令 `kill` 发送信号，看是否字符串被正常输出。

欢迎留言和我分享你的疑惑和见解，也欢迎可以收藏本节内容，反复研读。你也可以把今天的内容分享给你的朋友，和他一起学习和进步。

# 趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院

云计算技术部首席架构师



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 36 | 进程间通信：遇到大项目需要项目组之间的合作才...

下一篇 38 | 信号（下）：项目组A完成了，如何及时通知项目...

## 精选留言 (4)

写留言



Luke

2019-06-24

信号是不是操作系统的一个原语，在Windows端，对应的实现是消息Message循环





**Leon** 

2019-06-21

c语言开发者路过，表示以前从来不知道signal不是系统调用



**免费的人**

2019-06-21

关于SA\_ONESHOT，为什么我平时用signal的时候，处理函数可以被重复调用呢



**Sharry**

2019-06-21

收获满满, 不过看了标题以为是信号发送的流程, 结果学到了信号的注册和处理

