从非缓冲输入流到 Linux 控制台的历史

目录

这篇也是源自于水源C板上板友的一个问题，涉及Linux上的控制台的实现方式和历史原因。因为内容比较长，所以在这里再排版一下发出来。 原帖在这里 。

可以设置不带缓冲的标准输入流

吗？

WaterElement(UnChanged) 于 2014年12月09日23:29:51 星期二 问到：

> 请问对于标准输入流可以设置不带缓冲吗？比如以下程序
>
> ```
> 1  #include <stdio.h>
> 2  #include <unistd.h>
> 3
> 4  int main(int argc, char *argv[]) {
> 5      FILE *fp = fdopen(STDIN_FILENO, "r");
> 6      setvbuf(fp, NULL, _IONBF, 0);
> 7      char buffer[20];
> 8      buffer[0] = 0;
> 9      fgets(buffer, 20, fp);
> 10     printf("buffer is:%s", buffer);
> 11     return 0;
> 12 }
> ```
>
> 似乎还是需要在命令行输入后按回车才会让 fgets 返回，不带缓冲究竟体现在哪里？

这和缓存无关，是控制台的实现方式的问题。

再讲细节一点，这里有很多个程序和设备。以下按 linux 的情况讲：

1. 终端模拟器窗口（比如xterm）收到键盘事件
2. 终端模拟器(xterm)把键盘事件发给虚拟终端 pty1
3. pty1 检查目前的输入状态，把键盘事件转换成 stdin 的输入，发给你的程序

4. 你的程序的 c 库从 stdin 读入一个输入，处理

标准库说的输入缓存是在 4 的这一步进行的。而行输入是在 3 的这一步被缓存起来的。

终端pty有多种状态，一般控制台程序所在的状态叫「回显行缓存」状态，这个状态的意思是:

1. 所有普通字符的按键，会回显到屏幕上，同时记录在行缓存区里。
2. 处理退格(BackSpace)，删除(Delete)按键为删掉字符，左右按键移动光标。
3. 收到回车的时候把整个一行的内容发给stdin。

参考： http://en.wikipedia.org/wiki/Cooked_mode

同时在Linux/Unix下可以发特殊控制符号给pty让它进入「raw」状态，这种状态下按键 不会被回显，显示什么内容都靠你程序自己控制。如果你想得到每一个按键事件需要用raw状态，这需要自己控制回显自己处理缓冲， 简单点的方法是用 readline 这样的库（基本就是「回显行缓存」的高级扩展，支持了 Home/End，支持历史）或者 ncurses 这样的库（在raw状态下实现了一个简单的窗口/ 事件处理框架）。

参考：
http://en.wikipedia.org/wiki/POSIX_terminal_interface#History

除此之外， Ctrl-C 转换到 SIGINT， Ctrl-D 转换到 EOF 这种也是在 3 这一步做的。

以及，有些终端模拟器提供的 Ctrl-Shift-C 表示复制这种是在 2 这一步做的。

以上是 Linux/unix 的方式。 Windows的情况大体类似，只是细节上有很多地方不一样：

1. 窗口事件的接收者是创建 cmd 窗口的 Win32 子系统。
2. Win32子系统接收到事件之后，传递给位于 命令行子系统 的 cmd 程序
3. cmd 程序再传递给你的程序。

Windows上同样有类似行缓存模式和raw模式的区别，只不过实现细

节不太一样。

strace查看了下

WaterElement(UnChanged) 于 2014年12月10日21:53:54 星期三 回复：

> 感谢FC的详尽解答。
>
> 用strace查看了下，设置标准输入没有缓存的话读每个字符都会调用一次 read 系统调用， 比如输入abc：
>
> ```
> 1  read(0, abc
> 2  "a", 1)                = 1
> 3  read(0, "b", 1)                = 1
> 4  read(0, "c", 1)                = 1
> 5  read(0, "\n", 1)                = 1
> ```
>
> 如果有缓存的话就只调用一次了 read 系统调用了：
>
> ```
> 1  read(0, abc
> 2  "abc\n", 1024)            = 4
> ```

如果想感受一下 raw mode

没错，这个是你的进程内C库做的缓存，tty属于字符设备所以是一个一个字符塞给你的 程序的。

如果想感受一下 raw mode 可以试试下面这段程序（没有检测错误

返回值）

```
#include <stdio.h>
#include <unistd.h>
#include <termios.h>

static int ttyfd = STDIN_FILENO;
static struct termios orig_termios;

/* reset tty - useful also for restoring the terminal when this process
   wishes to temporarily relinquish the tty
*/
int tty_reset(void){
    /* flush and reset */
    if (tcsetattr(ttyfd,TCSAFLUSH,&orig_termios) < 0) return -1;
    return 0;
}


/* put terminal in raw mode - see termio(7I) for modes */
void tty_raw(void)
{
    struct termios raw;

    raw = orig_termios;  /* copy original and then modify below */

    /* input modes - clear indicated ones giving: no break, no CR to NL,
       no parity check, no strip char, no start/stop output (sic) control */
    raw.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);

    /* output modes - clear giving: no post processing such as NL to CR+NL */
    raw.c_oflag &= ~(OPOST);

    /* control modes - set 8 bit chars */
    raw.c_cflag |= (CS8);
```

```
34
35      /* local modes - clear giving: echoing off, canonical off (no er
ase with
36         backspace, ^U,...),  no extended functions, no signal chars (
^Z,^C) */
37      raw.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);
38
39      /* control chars - set return condition: min number of bytes a
nd timer */
40      raw.c_cc[VMIN] = 5; raw.c_cc[VTIME] = 8; /* after 5 bytes or .8 s
econds
41                                      after first byte seen      */
42      raw.c_cc[VMIN] = 0; raw.c_cc[VTIME] = 0; /* immediate - anyth
ing      */
43      raw.c_cc[VMIN] = 2; raw.c_cc[VTIME] = 0; /* after two bytes, no
 timer  */
44      raw.c_cc[VMIN] = 0; raw.c_cc[VTIME] = 8; /* after a byte or .8 s
econds */
45
46      /* put terminal in raw mode after flushing */
47      tcsetattr(ttyfd,TCSAFLUSH,&raw);
48   }
49
50
51   int main(int argc, char *argv[]) {
52      atexit(tty_reset);
53      tty_raw();
54      FILE *fp = fdopen(ttyfd, "r");
55      setvbuf(fp, NULL, _IONBF, 0);
56      char buffer[20];
57      buffer[0] = 0;
58      fgets(buffer, 20, fp);
59      printf("buffer is:%s", buffer);
60      return 0;
61   }
```

终端上的字符编程

vander(大青蛙) 于 2014年12月12日08:52:20 星期五 问到：

> 学习了！
>
> 进一步想请教一下fc大神。如果我在Linux上做终端上的字符编程，是否除了用ncurses库 之外，也可以不用该库而直接与终端打交道，就是你所说的直接在raw模式？ 另外，终端类型vt100和linux的差别在哪里？为什么Kevin Boone的KBox配置手册里面说必 须把终端类型设成linux，而且要加上terminfo文件，才能让终端上的vim正常工作？term info文件又是干什么的？

Linux控制台的历史

嗯理论上可以不用 ncurses 库直接在 raw 模式操纵终端。

这里稍微聊一下terminfo/termcap的历史，详细的历史和吐槽参考 Unix hater's Handbook 第6章 Terminal Insanity。

首先一个真正意义上的终端就是一个输入设备（通常是键盘）加上一个输出设备（打印 机或者显示器）。很显然不同的终端的能力不同，比如如果输出设备是打印机的话，显 示出来的字符就不能删掉了（但是能覆盖），而且输出了一行之后就不能回到那一行了 。再比如显示器终端有的支持粗体和下划线，有的支持颜色，而有的什么都不支持。 早期Unix工作在电传打字机（TeleTYpe）终端上，后来Unix被port到越来越多的机器上 ，然后越来越多类型的终端会被连到Unix上，很可能同一台Unix主机连了多个不同类型 的终端。由于是不同厂商提供的不同的终端，能力各有不同，自然控制他们工作的方式 也是不一样的。所有终端

都支持回显行编辑模式，所以一般的面向行的程序还比较好写，但是那时候要撰写支持所有终端的「全屏」程序就非常痛苦，这种情况就像现在浏览 器没有统一标准下写HTML要测试各种浏览器兼容性一样。 通常的做法是

1. 使用最小功能子集
2. 假设终端是某个特殊设备，不管别的设备。

水源的代码源头 Firebird2000 就是那样的一个程序，只支持固定大小的vt102终端。

这时有一个划时代意义的程序出现了，就是 vi，试图要做到「全屏可视化编辑」。这在 现在看起来很简单，但是在当时基本是天方夜谭。 vi 的做法是提出一层抽象，记录它所需要的所有终端操作，然后有一个终端类型数据库，把那些操作映射到终端类型的具体指令上。当然并不是所有操作在所有终端类型上都 支持，所以会有一堆 fallback，比如要「强调」某段文字，在彩色终端上可能 fallback 到红色，在黑白终端上可能 fallback 到粗体。

vi 一出现大家都觉得好顶赞，然后想要写更多类似 vi 这样的全屏程序。然后 vi 的作 者就把终端抽象的这部分数据库放出来形成一个单独的项目，叫 termcap （Terminal Capibility），对应的描述终端的数据库就是 termcap 格式。然后 termcap 只是一个 数据库（所以无状态）还不够方便易用，所以后来又有人用 termcap 实现了 curses 。

再后来大家用 curses/termcap 的时候渐渐发现这个数据库有一点不足：它是为 vi 设 计的，所以只实现了 vi 需要的那部分终端能力。然后对它改进的努力就形成了新的 terminfo 数据库和 pcurses 和后来的 ncurses 。 然后 VIM 出现了自然也用 terminfo 实现这部分终端操作。

然后么就是 X 出现了， xterm 出现了，大家都用显示器了，然后 xterm 为了兼容各种 老程序加入了各种老终端的模拟模式。不过因为最普及的终端是 vt100 所以 xterm 默 认是工作在兼容 vt100 的模式下。然后接下来各种新程序（偷懒不用*curses的那些） 都以 xterm/vt100 的方式写。

嗯到此为止是 Unix 世界的黑历史。

知道这段历史的话就可以明白为什么需要 TERM 变量配合 terminfo 数据库才能用一些 Unix 下的全屏程序了。类比一下的话这就是现代浏览

器的 user-agent。

然后话题回到 Linux。 大家知道 Linux 早期代码不是一个 OS， 而是 Linus 大神想 在他的崭新蹭亮的 386-PC 上远程登录他学校的 Unix 主机，接收邮件和逛水源（咳咳）。于是 Linux 最早的那部分代码并不是一个通用 OS 而只是一个 bootloader 加一个 终端模拟器。所以现在 Linux 内核里还留有他当年实现的终端模拟器的部分代码，而这 个终端模拟器的终端类型就是 linux 啦。然后他当时是为了逛水源嘛所以 linux 终端 基本上是 vt102 的一个接近完整子集。

说到这里脉络大概应该清晰了， xterm终端类型基本模拟 vt100，linux终端类型基本模 拟 vt102。这两个的区别其实很细微，都是同一个厂商的两代产品嘛。有差别的地方差 不多就是 Home / End / PageUp / PageDown / Delete 这些不在 ASCII 控制字符表里的按键的映射关系不同。

嗯这也就解释了为什么在linux环境的图形界面的终端里 telnet 上水源的话，上面这些 按键会错乱…… 如果设置终端类型是 linux/vt102 的话就不会乱了。在 linux 的 TTY 里 telnet 也不会乱的样子。

写到这里才发现貌似有点长…… 总之可以参考 Unix hater's Handbook 里的相关历史评论和吐槽，那一段非常有意思。