從非緩衝輸入流到 Linux 控制檯的歷史

目錄

Contents

    - 如果想感受一下 raw mode
- 終端上的字符編程
    - Linux控制檯的歷史

這篇也是源自於水源C板上板友的一個問題，涉及Linux上的控制檯的實現方式和歷史原因。因爲內容比較長，所以在這裏再排版一下發出來。 原帖在這裏 。

可以設置不帶緩衝的標準輸入流嗎？

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

請問對於標準輸入流可以設置不帶緩衝嗎？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_FI
LENO, "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", buf
fer);
11      return 0;
12  }
```

似乎還是需要在命令行輸入後按回車纔會讓 fgets 返回，不帶緩衝究竟體現在哪裏？

這和緩存無關，是控制檯的實現方式的問題。

再講細節一點，這裏有很多個程序和設備。以下按 linux 的情況講：

1. 終端模擬器窗口（比如xterm）收到鍵盤事件
2. 終端模擬器(xterm)把鍵盤事件發給虛擬終端 pty1
3. pty1 檢查目前的輸入狀態，把鍵盤事件轉換成 stdin 的輸入，發給你的程序
4. 你的程序的 c 庫從 stdin 讀入一個輸入，處理

標準庫說的輸入緩存是在 4 的這一步進行的。而行輸入是在 3 的這一步被緩存起來的。

終端pty有多種狀態，一般控制檯程序所在的狀態叫「回顯行緩存」狀態，這個狀態的意思是:

1. 所有普通字符的按鍵，會回顯到屏幕上，同時記錄在行緩存區裏。
2. 處理退格(`BackSpace`)，刪除(`Delete`)按鍵爲刪掉字符，左右按鍵移動光標。
3. 收到回車的時候把整個一行的內容發給stdin。

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

同時在Linux/Unix下可以發特殊控制符號給pty讓它進入「raw」狀態，這種狀態下按鍵 不會被回顯，顯示什麼內容都靠你程序自己控制。 如果你想得到每一個按

鍵事件需要用raw狀態，這需要自己控制回顯自己處理緩衝， 簡單點的方法是用 readline 這樣的庫（基本就是「回顯行緩存」的高級擴展，支持了 Home/End，支持歷史）或者 ncurses 這樣的庫（在raw狀態下實現了一個簡單的窗口/ 事件處理框架）。

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

除此之外， `Ctrl-C` 轉換到 SIGINT， `Ctrl-D` 轉換到 EOF 這種也是在 3 這一步做的。

以及，有些終端模擬器提供的 `Ctrl-Shift-C` 表示複製這種是在 2 這一步做的。

以上是 Linux/unix 的方式。 Windows的情況大體類似，只是細節上有很多地方不一樣：

1. 窗口事件的接收者是創建 cmd 窗口的 Win32 子系統。
2. Win32子系統接收到事件之後，傳遞給位於 命令行子系統 的 cmd 程序
3. cmd 程序再傳遞給你的程序。

Windows上同樣有類似行緩存模式和raw模式的區別，只不過實現細節不太一樣。

strace查看了下

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

感謝FC的詳盡解答。

用strace查看了下，設置標準輸入沒有緩存的話讀每個字符都會調用一次 read 系統調用， 比如輸入 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
```

如果有緩存的話就只調用一次了 read 系統調用了：

```
1  read(0, abc
2  "abc\n", 1024)
 = 4
```

如果想感受一下 raw mode

沒錯，這個是你的進程內C庫做的緩存，tty屬於字符設備所以是一個一個字符塞給你的 程序的。

如果想感受一下 raw mode 可以試試下面這段程序（沒有檢測錯誤返回值）

```
 1  #include <stdio.h>
 2  #include <unistd.h>
 3  #include <termios.h>
 4
 5  static int ttyfd = STDIN_FILENO;
 6  static struct termios orig_termios;
 7
 8  /* reset tty - useful also for resto
ring the terminal when this process
 9     wishes to temporarily relinquish
the tty
10  */
11  int tty_reset(void){
12      /* flush and reset */
13      if (tcsetattr(ttyfd,TCSAFLUSH,&o
rig_termios) < 0) return -1;
14      return 0;
15  }
16
17
18  /* put terminal in raw mode - see te
rmio(7I) for modes */
```

```
19  void tty_raw(void)
20  {
21      struct termios raw;
22
23      raw = orig_termios;  /* copy ori
ginal and then modify below */
24
25      /* input modes - clear indicated
 ones giving: no break, no CR to NL,
26         no parity check, no strip cha
r, no start/stop output (sic) control */

27      raw.c_iflag &= ~(BRKINT | ICRNL |
 INPCK | ISTRIP | IXON);
28
29      /* output modes - clear giving:
no post processing such as NL to CR+NL
*/
30      raw.c_oflag &= ~(OPOST);
31
32      /* control modes - set 8 bit cha
rs */
33      raw.c_cflag |= (CS8);
34
35      /* local modes - clear giving: e
choing off, canonical off (no erase with

36         backspace, ^U,...),  no exten
ded functions, no signal chars (^Z,^C)
*/
37      raw.c_lflag &= ~(ECHO | ICANON |
IEXTEN | ISIG);
```

```
38
39      /* control chars - set return co
ndition: min number of bytes and timer
*/
40      raw.c_cc[VMIN] = 5; raw.c_cc[VTI
ME] = 8; /* after 5 bytes or .8 seconds
41
            after first byte seen
*/
42      raw.c_cc[VMIN] = 0; raw.c_cc[VTI
ME] = 0; /* immediate - anything
*/
43      raw.c_cc[VMIN] = 2; raw.c_cc[VTI
ME] = 0; /* after two bytes, no timer
*/
44      raw.c_cc[VMIN] = 0; raw.c_cc[VTI
ME] = 8; /* after a byte or .8 seconds
*/
45
46      /* put terminal in raw mode afte
r 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  }
```

終端上的字符編程

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

學習了！

進一步想請教一下fc大神。如果我在Linux上做終端上的字符編程，是否除了用ncurses庫 之外，也可以不用該庫而直接與終端打交道，就是你所說的直接在raw模式？ 另外，終端類型vt100和linux的差別在哪裏？爲什麼Kevin Boone的KBox配置手冊裏面說必 須把終端類型設成linux，而且要加上terminfo文件，才能讓終端上的vim正常工作？term info文件又是幹什麼的？

Linux控制檯的歷史

嗯理論上可以不用 ncurses 庫直接在 raw 模式操縱終端。

這裏稍微聊一下terminfo/termcap的歷史，詳細的歷史和吐槽參考 Unix hater's Handbook 第6章 Terminal Insanity。

首先一個真正意義上的終端就是一個輸入設備（通常是鍵盤）加上一個輸出設備（打印 機或者顯示器）。很顯然不同的終端的能力不同，比如如果輸出設備是打印機的話，顯 示出來的字符就不能刪掉了（但是能覆蓋），而且輸出了一行之後就不能回到那一行了 。再比如顯示器終端有的支持粗體和下劃線，有的支持顏色，而有的什麼都不支持。 早期Unix工作在電傳打字機（TeleTYpe）終端上，後來Unix被port到越來越多的機器上 ，然後越來越多類型的終端會被連到Unix上，很可能同一臺Unix主機連了多個不同類型 的終端。由於是不同廠商提供的不同的終端，能力各有不同，自然控制他們工作的方式 也是不一樣的。所有終端都支持回顯行編輯模式，所以一般的面向行的程序還比較好寫 ，但是那時候要撰寫支持所有終端的「全屏」程序就非常痛苦，這種情況就像現在瀏覽 器沒有統一標準下寫HTML要測試各種瀏覽器兼容性一樣。 通常的做法是

1. 使用最小功能子集
2. 假設終端是某個特殊設備，不管別的設備。

水源的代碼源頭 Firebird2000 就是那樣的一個程序，只支持固定大小的vt102終端。

這時有一個劃時代意義的程序出現了，就是 vi，試圖要做到「全屏可視化編輯」。這在 現在看起來很簡單，但是在當時基本是天方夜譚。 vi 的做法是提出一層抽象，記錄它所需要的所有終端操作，然後有一個終端類型數據庫 ，把那些操作映射到終端類型的具體指令上。當然並不是所有操作在所有終端類型上都 支持，所

以會有一堆 fallback，比如要「強調」某段文字，在彩色終端上可能 fallback 到紅色，在黑白終端上可能 fallback 到粗體。

vi 一出現大家都覺得好頂讚，然後想要寫更多類似 vi 這樣的全屏程序。然後 vi 的作 者就把終端抽象的這部分數據庫放出來形成一個單獨的項目，叫 termcap（Terminal Capibility），對應的描述終端的數據庫就是 termcap 格式。然後 termcap 只是一個 數據庫（所以無狀態）還不夠方便易用，所以後來又有人用 termcap 實現了 curses 。

再後來大家用 curses/termcap 的時候漸漸發現這個數據庫有一點不足：它是爲 vi 設 計的，所以只實現了 vi 需要的那部分終端能力。然後對它改進的努力就形成了新的 terminfo 數據庫和 pcurses 和後來的 ncurses 。 然後 VIM 出現了自然也用 terminfo 實現這部分終端操作。

然後麼就是 X 出現了， xterm 出現了，大家都用顯示器了，然後 xterm 爲了兼容各種 老程序加入了各種老終端的模擬模式。不過因爲最普及的終端是 vt100 所以 xterm 默 認是工作在兼容 vt100 的模式下。然後接下來各種新程序（偷懶不用*curses的那些） 都以 xterm/vt100 的方式寫。

嗯到此爲止是 Unix 世界的黑歷史。

知道這段歷史的話就可以明白爲什麼需要 TERM 變量配合 terminfo 數據庫纔能用一些 Unix 下的全屏程序了。類比一下的話這就是現代瀏覽器的 user-agent。

然後話題回到 Linux。 大家知道 Linux 早期代碼不是一個 OS， 而是 Linus 大神想 在他的嶄新蹭亮的 386-PC 上遠程登錄他學校的 Unix 主機，接收郵件和逛水源（咳咳）。於是 Linux 最早的那部分代碼並不是一個通用 OS 而只是一個 bootloader 加一個 終端模擬器。所以現在 Linux 內核裏還留有他當年實現的終端模擬器的部分代碼，而這 個終端模擬器的終端類型就是 linux 啦。然後他當時是爲了逛水源嘛所以 linux 終端 基本上是 vt102 的一個接近完整子集。

說到這裏脈絡大概應該清晰了， xterm終端類型基本模擬 vt100，linux終端類型基本模 擬 vt102。這兩個的區別其實很細微，都是同一個廠商的兩代產品嘛。有差別的地方差 不多就是 `Home` / `End` / `PageUp` / `PageDown` / `Delete` 這些不在 ASCII 控制字符表裏的按鍵的映射關係不同。

嗯這也就解釋了爲什麼在linux環境的圖形界面的終端裏 telnet 上水源的話，上面這些 按鍵會錯亂…… 如果設置終端類型是 linux/vt102 的話就不會亂了。在 linux 的 TTY 裏 telnet 也不會亂的樣子。

寫到這裏纔發現貌似有點長…… 總之可以參考 Unix hater's Handbook 裏的相關歷史評論和吐槽，那一段非常有意思。