Содержание

терминал

Терминал и потоки ввода/вывода

Мы знаем, что программы, работающие в командной строке, могут быть ориентированы на работу с терминалом (такие как top, vim, mutt), и программы, ориентированные на работу с потоками ввода/вывода (такие как cat, sort, ls). И если с первыми понятно, что забери у них терминал, и работать они нормально не будут, то со вторыми, как будто бы, разницы быть не должно. Работа выполняется всё равно через файловые дескрипторы (fd 0, fd 1 и fd 2), и куда они подключены не имеет значения.

Так подсказывает логика и жизненный опыт. Однако в действительности всё не так просто.

Сравните вывод:

```
$ ls
$ ls | cat
```

Казалось бы, разницы быть не должно. cat работает просто повторителем и должен выдавать то, что даёт ему ls. Однако это не так.

В первом случае будет красивый вывод с цвета и по столбикам, а во втором всё в один ряд и без цветов. Почему?

Дело в том, что ls определяет, работает ли он непосредственно с терминалом, и если да, он опрашивает его ширину, форматирует вывод по ней и использует цвета. Если нет, то он выдаёт всё как есть и не напрягается.

Этот фокус использует довольно много программ. Как правило, всё ограничивается украшательством и раскрашиванием вывода, но могут и более интересные случаи, когда программа при отключении её от терминала начинает вести себя совершенно иначе, вплоть до того, что вообще перестаёт работать.

Вот вам пример:

Простейшая программа на python:

```
print u"Привет!"
```

Программа выводит на экран (а точнее, на стандартный поток вывода) слово «Привет» и завершается.

Так вот она не будет работать, если её отключить от терминала! Сравните:

```
$ python
Python 2.7.10+ (default, Oct 10 2015, 09:11:24)
[GCC 5.2.1 20151003] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print u"Привет"
```

```
Привет
>>>

$ python | cat
Python 2.7.10+ (default, Oct 10 2015, 09:11:24)
[GCC 5.2.1 20151003] on linux2
Type "help", "copyright", "credits" or "license" for more information.
»> print u"Привет"
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-5:
ordinal not in range(128)
```

В первом случае всё прекрасно. Во втором — программа вылетает с сообщением об ошибке. Но почему???

Понятно, что наша программа содержит где-то ошибку, но почему она себя так по-разному ведёт?

Дело в том что, что строка u«Привет» это юникодная строка находящаяся во внутреннем представлении языка.

При выводе она должна быть представлена в виде обычной последовательности байтов, закодирована.

Python делает тут немного магии и при выводе пытается закодировать её автоматически. В первом случае у него получается, потому что он подключен к терминалу и угадывает его кодировку; во втором — он работает с обычным потоком, считай с файлом, и у файла никакой кодировки нет.

```
$ python -c 'import sys; print sys.stdout.encoding'
UTF-8
$ python -c 'import sys; print sys.stdout.encoding' | cat
None
```

Правильнее было бы при выводе строки её закодировать (print u«Привет».encode('utf8')), и тогда никаких проблем бы не возникло. Другое решение:

```
import sys
reload(sys) # Весь прикол в reload
sys.setdefaultencoding('UTF8')
```

Это, в общем-то, микрохак, который не предполагался создателями языка, отсюда и хитрые манипуляции с модулем sys.

Кодировка, правда, не является каким-то свойством терминала, как можно было бы подумать, и Python не считывает её с терминала. Она задана в переменной окружения, которая доступна независимо от того, подключен поток к терминалу или нет. Если локаль не настроена, кодировку терминалу определить Python не сможет всё равно:

```
$ python -c 'import sys; print sys.stdin.encoding'
```

```
UTF-8
$ LANG=C python -c 'import sys; print sys.stdin.encoding'
ANSI_X3.4-1968
```

Как можно определить, подключен поток к терминалу, чтобы менять поведение программы в зависимости от этого? В каждом языке для этого есть соответствующие методы, в шелле это делается с помощью test (другое название: [, квадратная скобка открывается):

```
$ [ -t 0 ] ; echo $?
0
$ [ -t 0 ] < /dev/null ; echo $?
1
```

В первом случае поток ввода (fd 0) подключен к терминалу (код завершения $? программы test равен 0), а во втором мы переключаем поток на /dev/null, и это больше не терминал (код завершения $? равен 1).

Мы знаем, как отключить программу от терминала, но как сделать наоборот? Программа работает без терминала, а нам нужно, чтобы она думала, что терминал у неё есть. Как мы знаем, многие программы могут вообще не работать без терминала, а многие ведут себя совершенно неправильно. Программы, запущенные из cron или из других программ демонов не будут иметь терминала.

Нужно использовать программу, предоставляющую терминал. Вариантов много, простейший из них — программа script. Её плюс в том, что она очень простая, и что она, будучи разработанной ещё на заре юниксостроения, есть практически во всех UNIX/Linux-системах:

```
$ tty
/dev/pts/4
$ tty < /dev/null
not a tty
$ script -q -c tty < /dev/null
/dev/pts/6
```

Программа tty, которая выводит имя терминала, к которому подключен стандартный поток ввода, в первом случае указывает имя текущего терминала из которого она запущена, во втором — говорит что это вообще не терминал, правильно, потому что мы перенаправили поток ввода на /dev/null; но в третьем — опять терминал! — несмотря на перенаправление потока на /dev/null.

Запись терминальной сессии

Программа script предназначена для создания лога терминальной сессии. Она выделяет дочерней программе терминал, происходящее на котором попадает в файл.

```
$ script
Script started, file is typescript
$ # действия которые записываются
$ exit
```

```
Script done, file is typescript
```

В данном случае имя файла с логом работы — typescript.

Позже его можно просматривать, анализировать, делиться им с другими.

Есть несколько интересных проектов, которые развивают идею script:

- ttyrecord, где записывается не только поток символов, но и временные метки, что позволяет в будущем не просто просмотреть журнал, а прокрутить кино, как это было;
- asciinema, развивает эту идею и представляет кино прямо в веб — запись терминала можно просматривать не только в терминале, а прямо в браузере, и что ещё более важно — делиться ею; этакой youtube для действий в терминале;
- lilalo (сокращённо от live lab log) [http://xgu.ru/lilalo], где запись представлена не просто неструктурированным потоком байтов, проходящих через терминал, а высокоуровневым потоком, где команды, их вывод, условия их запуска хранятся отдельно.

Перехват вывода терминала

Эти все программы хорошо известны и многими используются. Тут ничего нового. А вот более интересный вопрос:

А можно ли вести запись уже открытого терминала, если программа уже работает, или если нет возможности запустить script перед ней?

Да, такая возможность есть, даже несколько, хотя и не очевидны.

В BSD-системах это можно сделать с помощью watch (там watch делает совсем другое чем в Linux), правда это работает только с виртуальными терминалами, но не работает с pty.

В Linux это можно сделать через SystemTap, прямо руками:

```
  - !/usr/bin/stap

if (kernel_string($tty->name) == @1) {
printf("%s", kernel_string_n($buf, $c))
}
}
```

SystemTap вещь очень мощная, но это уже слишком, из пушки по воробьям. Кроме того, что нужно устанавливать собственно SystemTap, нужны ещё отладочные символы ядра (kernel*dbg) и его заголовочные файлы. И, конечно, нужны права рута.

Намного более удобно использовать ispy https://github.com/dellis23/ispy, который не требует прав рута для того чтобы следить за собственными процессами. А с правами рута позволяет следить за любым терминалом системы.

```
$ ispy PID
```

показывает то, что выводится на терминал, к котором подключен процесс PID. Это вывод не

только самого процесса, но и его потомков.

Не только посмотреть, но и записать, что там происходит:

```
$ script -c 'ispy PID'
```

ispy использует ptrace (трассировку системных вызовов) и пытается на основе перехваченных вызовов воссоздать картину происходящего на терминале. Получается более-менее, но не идеально (попробуйте понаблюдать за процессами использующими терминал от всей души).

Все описанные решения (systemtap, watch, ispy и другие неназванные) являются хаками, влезающими в терминал каким-то непредусмотренным способом. Их следует всегда по возможности избегать, если есть возможность легальной записи происходящего на терминале, с помощью таких программ как script.

Мультиплексирование терминала

создать на одном терминале несколько можно с помощью одной из этих программ:

- screen (GNU Screen);
- tmux;
- pymux (клон tmux на Python) или другие клоны.

Поговорим о них, и рассмотрим их преимущества и недостатки.

GNU Screen — программа наиболее старая и заслуженная из этого списка. Она была написана в 1987 году (ещё в долинуксовые времена), в Берлине, и потом долго время, наверное, не меньше 20 лет, была единственным полноценным консольным мультиплексором терминала.

Решала она собственно две задачи:

1. Создание нескольких терминалов на одном и удобное переключение между ними;
2. Работа в отключенном (detached) режиме, когда от терминала можно отключиться, а приложение продолжает работать невидимым. После этого можно подключиться и продолжить работу с ним.

Вот из-за этих двух фич screen и используется. Фичи эти достаточно очевидные и всем широко известные.

Но что может screen ещё? Какие фокусы можно делать с ним?

Одна из интересных возможностей — это запуск приложений в detached-режиме, то есть по сути превращение консольной программы в программу-демон (хотя это, формально говоря, не так, потому что процесс имеет терминал, хоть и не видимый без подключения к нему):

```
$ screen -S prg1 -d -m top
$
```

Здесь интерактивная программа top запускается в невидимом, отключенном (detached) сеансе screen, и сама команда, вызванная в шелле, мгновенно завершается (а top продолжает

работать в фоне). Конечно, это не имеет особого смысла для top, но есть много программ и (в особенности самодельных) скриптов, которые вы хотите запустить и забыть, периодически подключаясь к ним и посматривая, что они делают, и взаимодействуя с ними интерактивно.

(аналогично можно было бы поступить запустив программу с помощью nohup, но во-первых, она не предоставляет терминала, а во-вторых, к ней потом нельзя подключиться и как-то повлиять, можно только смотреть log-файл)

Если программа запускается из загрузочного скрипта, она будет работать под root'ом, если же такие мощные права не нужны, ограничить их можно с помощью su:

```
$ screen -S prg2 -d -m su - nobody /usr/bin/prg2
```

Сейчас программа будет выполняться не под root'ом, а под пользователем nobody.

Имена сеансов (prg1, prg2 и т.д.) нужны для того чтобы удобнее впоследствии подключаться к работающей программе.

```
$ screen -ls # просмотр работающих сеансов
$ screen -r -D -S prg1 # подключиться к сеасну prg1 (отключиться потом: ctrl-a
d)
```

screen имеет возможности логирования происходящего в нём, но ещё удобнее это сделать, скомбинировав вызов программы с программой script, о которой мы говорили в прошлый раз:

```
$ screen -S prg1 -d -m script -c my-mega-script.sh screen-log
```

Что ещё может screen хорошего, о чём стоит знать?

- Организовывать совместный доступ к терминалу нескольким пользователям (или из нескольких мест);
- Открывать несколько терминалов одновременно в одном окне (ключевое слово: регионы);
- Вводить текст в консоль от имени пользователя.

Однако, мы не будем останавливаться на этих его фичах — делает он это хорошо, но есть программы, которые делают это ещё лучше. Переходим к ним, если же вы хотите больше узнать о screen: xgu.ru/wiki/man:screen (man screen в русском переводе).

Появившись в 1987 году, ещё даже за 4 года до появления Linux, screen очень быстро стал стандартом де-факто как консольный оконный менеджер (или мультиплексор терминалов). Он полностью занимал эту нишу, и работал настолько хорошо, что альтернатив ему не требовалось. Написать альтернантиву было не сложно, как мы увидим чуть позже, потом их появилось сразу несколько (и некоторые из них настолько простые, что реализовать их смогут многие из вас буквально за пару дней кодинга), но они были не нужны.

В 2007 году, 20 годами позже, появилась реальная альтернатива screen'у, которая набирает всё большую и большую популярность: tmux.

В чём её принципиальные отличия?

1. tmux очень простой, написан с нуля, и из-за этого легко расширяется и быстро

развивается (в отличие от screen который остался практически в первозданном виде);
2. tmux имеет BSD-лицензию, а не лицензию GNU как screen, что импонирует пользователям BSD-систем и фанатам полный свободы (в BSD-шном смысле этого слова);
3. tmux разделяет понятия окна и панели, делая управление ими намного удобнее и гибче (screen пытался делать это с помощью регионов, но получилось довольно плохо);
4. (и это, на мой взгляд, самое главное) tmux имеет полноценную клиент-серверную архитектуру, что позволяет независимо взаимодействовать со всеми его панелями, окнами и сеансами.

Автоматизированная отправка данных на терминал

Допустим, у нас работает сеанс tmux:

```
$ tmux new -s session1
```

А в нём работает какое-то приложение (для простоты, bash).

Как набрать в нём какую-то команду?

```
$ tmux send -t session1.0 ls / ENTER
```

Вы посылаете в нулевое окно сеанса session1 символы ls /etc/ и после этого ENTER (если нужно действительно отправить слово ENTER, нужно отключить интепретацию специальных названий опцией -l). В окне наберутся данные символы как будто их набирали вы.

GNU Screen тоже умеет это делать (-X), но делает это довольно неуклюже, требует активизации окна, перед тем как отправлять текст, тупит. Несмотря на всё моё глубочайшее уважение к screen, я должен признать, что tmux тут просто красавчик.

Сравните:

```
$ screen -X select 1
$ screen -X readreg p /tmp/file
$ screen -X paste p
```

(выбрать окно 1, считать в регистр p содержимое /tmp/file, вставить содержимое регистра p). tmux — красавчик.

Разумеется, вводить таким образом текст можно не только в локальную, но и удалённую систему (если в окне до этого запустить ssh/telnet). Это позволяет вводить сгенерированный текст в системы не имеющие нормальных механизмов автоматизации.

Конечно, при однократном вводе это можно сделать и другими разными способами:

```
sh generate-vlans | socat - EXEC:"ssh ${SWITCH}",setsid,pty,ctty
```

Здесь мы заходим на ${SWITCH} по ssh и передаём на него конфигурацию VLAN'ов (подробнее: http://xgu.ru/wiki/VLAN ищите по слову socat). Сгенерированные команды отправляются на коммутатор, как будто бы их набрали вы. Но потом происходит отключение, как будто ничего и не было.

Если же вы хотите совместить интерактивный и неинтерактивный, автоматизированный, ввод, здесь лучше подойдёт tmux.

Можно пойти ещё дальше и засунуть этот слой автоматизации не в шелл, а в более мощный интерпретатор, например, в ipython, сделать там библиотеку нужных функций и прямо вызывать их оттуда:

```
>>> routers = ['router1', 'router2', 'router3']
>>> configure_ospf(routers, '10.0.0.0/8')
```

Подобный механизм реализован в проекте Xentaur. Xentaur был, правда, написан ещё в до-tmux-ную эру, поэтому использует screen, но принципы остаются теми же.

tmux send крут, но он стреляет вслепую. tmux не анализирует то, что ему отвечают в терминале, и хотят вывод он, естественно, видит, ибо всё происходит в нём, нормальных встроенных механизмов для доступа к этому выводу нет. Как же быть?

Автоматизация двустороннего взаимодействия с терминалом

Существует множество программ и библиотек для взаимодействия с терминальными программами (такими, например, как ssh, которые считывают пароль с терминала): expect, pexpect (python), suppose (для node.js), PTY и ruby_expect (для ruby) и множество других. Программы эти работают по принципу: вопрос — ответ (ожидание вывода, ввод текста).

Можно ли совместить их с screen/tmux? То есть сделать так, чтобы работая в screen/tmux, можно было переходить в автоматический режим, где ввод и вывод обрабатывает программа (вспомните наш пример с конфигурированием коммутатора, и представьте что нам нужно не просто постить данные вслепую, а использовать каким-то образом вывод):

```
  - !/usr/bin/expect -f

spawn screen -x -S server
send "ssh igor@127.0.0.1\r"
expect "assword" { send "mega-secret-pass\r" }
expect -re "(\\\$ |# )"
send "ls\r"
expect -re "(\\\$ |# )"
send "\01d"
```

Здесь мы подключаемся к скрину server, запускаем там ssh, который коннектится локально под пользователем igor, ждёт требования ввести пароль, вводит его, потом ждёт приглашение шелла и исполняет там ls. После этого отключается (ctrl-a d).

Код ctrl-a d определён следующим образом:

В expect, как и во многих других программах, можно использовать восьмеричный код клавиши.

Определить код клавиши можно так:

1. Запустить od -c
2. Нажать клавишу или комбинацию клавиш (в нашем случае ctrl-a)
3. Нажать enter
4. Нажать ctrl-d

```
$ od -c
^A
0000000 001 \n
0000002
```

Получается, что в данном случае это код \001.

Перенос работающей программы с одного терминала на другой

Во всех случаях программа запускается внутри screen/tmux, то есть подумать об этом нужно заранее. Что же делать, если программа уже работает, а её нужно переносить в screen/tmux по-горячему? Возможно ли это вообще?

Как перенести работающую программу на другой терминал?

```
$ reptyr PID
```

Программа reptyr меняет терминал процесса с идентификатором PID на текущий. Меняет он его не напрямую, а делает хитрые перехват ввода и вывода с помощью уже известной нам по прошлому посту трассировки системных вызовов ptrace (хотя может подменить терминал и по-настоящему, перехватывая мастер-часть pty-файла с помощью так называемого TTY-stealing, который работает надёжнее, но требует прав рута).