

Шелл и утилиты

Здесь речь пойдет о командных оболочках, или командных интерпретаторах, именуемых также просто шеллами (*shell*), а также о мощных и часто неожиданных возможностях, предоставляемых классическими UNIX-утилитами и, особенно, их комбинациями.

Шеллы и их роль в POSIX-системах

Алексей Федорчук

Шелл (*Shell*), именуемый по-русски командной оболочкой, командным интерпретатором, командным процессором или иными, столь же неизящными словосочетаниями, - это первая программа, с которой сталкивается пользователь любой POSIX-совместимой ОС. И с ним же последним он расстается, выходя из системы. А все его действия во время работы - суть прямые или опосредованные команды, выполняемые в среде шелла. И даже если основная часть работы пользователя проходит в графическом режиме, в окружении интегрированных сред или оконных менеджеров, - окно терминала с приглашением командной строки быстро станет неотъемлемым атрибутом любого десктона. Ибо именно команды оболочки - самый простой и эффективный путь к выполнению всех операций по управлению файлами, многих задач обработки текста, да и просто запуска любых программ, что в консоли, что - в графическом режиме. Кроме того, всегда следует помнить, что вообще функционирование Unix-подобной системы в значительной мере происходит в шелл-среде. Ибо шелл-сценариями являются все скрипты инициализации системы, многие общесистемные и пользовательские конфигурационные файлы, такие системы управления пакетами, как порты FreeBSD и портежи Gentoo Linux. Ну и то, что любой пользователь свободных ОС рано или поздно начинает писать собственные сценарии оболочки - неизбежно, как распад мировой системы социализма (кто еще не начал - уж поверьте мне на слово, хотя скажи мне такое лет пять назад - сам бы не поверил).

О шеллах вообще

Сказанного в преамбуле, думаю, достаточно, чтобы отнести к выбору шелла со всей ответственностью. Причем при выборе этого должно учитываться два аспекта применения шелла - как среды для пользовательской работы, так и системы для исполнения общесистемных и пользовательских сценариев. Первый аспект охватывается понятием **интерактивный шелл**. Это - любой экземпляр командной оболочки, запущенный пользователем непосредственно. Если этот экземпляр запускается при входе пользователя в систему, его называют *login shell*. Очевидно, что *login shell* - также интерактивен, однако в сеансе работы каждого пользователя он будет единственным. Просто же интерактивных шеллов можно запустить сколько угодно - например, в каждом окне эмулятора терминала в Иксах будет функционировать собственная копия интерактивного шелла.

Имя исполняемого файла, запускающего *login shell* (вместе с полным путем к оному - например, `/bin/sh`) - атрибут учетной записи каждого реального пользователя системы. Теоретически в этом качестве могут выступать не только собственно шеллы (то есть командные оболочки), но и интерпретатор какого-либо языка программирования (например, *Tcl*), программа типа *Midnight Commander* или даже текстовый редактор. Однако эти случаи - специальные, и ниже далее рассматриваться не будут.

Второй аспект использования шелла - **неинтерактивный**. Неинтерактивный шелл - это экземпляр командной оболочки, вызываемый при выполнении пользователем любой команды или любого сценария. Он может быть вызван неявным или явным образом. Первый случай имеет место быть при выполнении команды из строки оболочки - ведь в этом случае создается (посредством системного вызова *fork*) точная копия породившего процесса (то есть той же интерактивной оболочки), а уже она вызовом *exec* запускает на исполнение введенную пользователем команду.

При составлении пользовательского или системного сценария шелл, вызываемый для его исполнения, можно указать явным образом - и настоятельно рекомендуется этой возможностью пользоваться. Делается это в первой строке скрипта, называемой **sha-bang**, которая по правилам должна иметь вид вроде `#!/bin/bash`

То есть после символов решетки (#) и восклицания (! - видимо, для пущей экспрессии:-)) указывать полный абсолютный путь к исполнимому файлу, оболочку запускающему. Делается это, в том числе, и во избежание недоразумений - так, просто единичный символ решетки в первой строке может быть интерпретирован не как комментарий, а как указание на запуск командной оболочки *csh*. И если сценарий был написан не для нее (а, например, для того же `/bin/sh`) - он, в силу различия синтаксиса, просто не будет выполнен.

Подчеркнем еще раз: хотя команда, запускающая интерактивный или неинтерактивный шелл, может носить то же имя, что и команда на запуск login shell (а в Linux, скажем, обычно так и есть), это - разные экземпляры программы, которые в общем случае могут быть настроены независимо. А значит - и вести себя разным образом. И потому не следует удивляться, когда bash в эмуляторе терминала не реагирует на управляющие клавиши, привычные для того же bash в виртуальной консоли. A Midnight Commander отказывается в своей командной строке опознавать пути к исполняемым файлам.

Очевидно, что претензии пользователя к интерактивному (особенно к пользовательскому login shell) и неинтерактивному шеллам могут быть разными. В первом случае, как явствует из названия, важнее всего удобство интерактивной работы - развитые средства автодополнения, работы с историей команд, возможности гибкой и информативной настройки приглашения командной строки. Для неинтерактивного же шелла на первый план выходят быстродействие и совместимость.

С быстродействием все понятно - когда единственной задачей командной оболочки является вызов на исполнение другой команды (или, в случае скрипта, серии связанных команд) - тратить ресурсы на обеспечение дополнительных возможностей может быть излишеством. А вот о совместимости следует сказать особо. Но сначала - несколько слов о том,

Какие бывают шеллы

Большая часть командных оболочек делится, на основе синтаксиса интерпретируемого ими языка, на две группы - sh- и csh-совместимые (о специфических шеллах, базирующихся, например, на диалекте LISP, я говорить не буду за их незнанием). На самом деле различия между ними синтаксисом команд не исчерпываются, а лежат глубже - в подходе к обработке командных конструкций, однако сейчас это не существенно.

Оболочки, относимые к sh-совместимым, происходят от первой командной оболочки первых Unix-систем, которую так и называют - shell или Bourne Shell (то есть шелл Борна). В ней были заложены многие возможности для интерпретации команд и их конструкций, то есть составления системных и пользовательских сценариев. Однако по своим интерактивным возможностям она оставляла желать лучшего, и потому на базе ее была создана оболочка Корна (Korn Shell).

Шелл Корна сохранил совместимость с борновским шеллом на уровне синтаксиса, однако привнес в него как дополнительные возможности интерпретации, так и приемы, направленные на удобство интерактивной работы. И потому именно он лег в основу стандарта POSIX (Portable Operation System Interface), которому должны удовлетворять командные оболочки POSIX-совместимых систем.

Следует заметить, что соответствие этому стандарту - один из критериев отнесения некоей ОС к семейству Unix или Unix-подобных. В частности, именно такой стандартизованный шелл должен вызываться при исполнении общесистемных сценариев (например, инициализации) любой POSIX-системы. Сам же по себе POSIX shell - некая мифическая абстракция, ближе к которой подходят /bin/sh - умолчальная пользовательская оболочка из FreeBSD, и ash, принятая в качестве стандартной в NetBSD (а также широко используемая во всяких мини-дистрибутивах Linux).

Шеллы и Борна, и Корна не были свободно распространяемыми программами в понимании FSF или BSD. Поэтому ни тот, ни другой не могли использоваться в таких ОС, как *BSD или Linux, хотя свободная реализация оболочки Корна (pdksh) и была создана. Однако, как и ее прототип, шелл Корна, она, несколько развившись относительно первозданного Bourne shell, обладала интерактивными достижениями, далекими от идеала. Столь же слабы они были и в ash. И потому наиболее широкое распространение получила оболочка bash (что расшифровывается как "еще одна оболочка Борна", " заново рожденный шелл" и тому подобным образом), разработанная в рамках [проекта GNU](#).

Популярность bash в немалой степени была обусловлена его интерактивными возможностями - он аккумулировал все достижения интерактивной мысли sh- и csh-совместимых оболочек, прибавив к ним немало собственных. И умудрившись при этом сохранить базовую совместимость с POSIX shell. Конечно, многие его собственные фичи ("bash'измы") выходили за рамки этого стандарта. Однако для соответствия оному был предусмотрен режим совместимости - то есть bash был способен эмулировать стандартный POSIX shell.

Однако главным для bash было то, что эта оболочка оказалась тесно интегрирована с операционной системой Linux: именно bash волею судеб стал одной из первых программ, которые Линус запустил поверх своего новосозданного ядра. И потому идеи bash-скриптинга пронизали Linux до самых его оснований. Ну а для совместимости использовался тот самый режим эмуляции: в Linux файл, запускающий POSIX shell (по стандарту он должен именоваться /bin/sh), является собой жесткую или символьическую ссылку на реальный /bin/bash (или /usr/bin/bash). Последний же, будучи вызванным таким образом, полностью воспроизводит стандартный POSIX shell (разумеется, путем утраты своих продвинутых фич).

Клан csh-совместимых оболочек развивался параллельно сыном и пасынкам Борна. Собственно оболочка csh была создана в Университете Беркли в ходе реализации проекта BSD Unix (тогда - именно так). Она получила дополнительные интерактивные возможности, во многом превосходящие таковые у современного ей шелла Корна. Главное же - языку, ею интерпретируемому, были приданы черты синтаксического сходства с языком Си (откуда, собственно, и название - C-Shell, хотя не следует думать, что на всамделишний Си ее

интерпретируемый язык похож). В результате оболочка csh оказалась весьма эффективной как для интерактивной работы, так и при создании сценариев. Только вот сценарии эти не были совместимы со скриптами POSIX shell, обретшего уже силу стандарта. То есть для создания общесистемных сценариев она оказалась практически не пригодной.

В отличие от большинства прочих достижений берклианской мысли, оболочка csh, по не вполне ясным для меня причинам, не обрела статуса свободной программы. Поэтому она не могла использоваться даже в своих родных пределах - в BSD-системах. Однако на замену ей была изобретена свободная оболочка tcsh - не просто функциональное воспроизведение, но дальнейшее развитие оболочки csh. По интерактивным возможностям она, как минимум, не уступает bash и потому утвердилась в стане свободных BSD-клонов. В частности, оболочка tcsh принята в качестве login shell для суперпользователя во FreeBSD. Правда, вызывается она в режиме совместимости с csh, однако /bin/csh - не более чем жесткая ссылка на /bin/tcsh, в чем легко убедиться, выведя идентификаторы их исполнимых файлов:

```
$ ls -i /bin/*csh  
16548 /bin/csh*      16548 /bin/tcsh*
```

Оболочка tcsh используется в качестве универсального "умолчального" пользовательского шелла также в OpenBSD. Однако характерно, что все общесистемные сценарии в обеих ОС написаны, тем не менее, в соответствие с требованиями POSIX Shell.

Проблема выбора

Из приведенного краткого обзора можно видеть, что в плане шеллов выбор пользователя достаточно обширен. А ведь я остановился только на самых распространенных. Однако рискну предположить, что большинство начинающих пользователей Linux'a об этом не особо задумываются. Ведь по умолчанию во всех его дистрибутивах в качестве шелла по умолчанию принят bash, обладающий как развитыми средствами интерпретации, так и продвинутыми интерактивными возможностями, да еще при сохранении совместимости со стандартом. Так зачем, казалось бы, искать добра от добра?

Первый вариант ответа очевиден - добро это дополнительное ищется в тех случаях, когда необходима минимизация ресурсов. Когда bash, занимающий более полумегабайта на диске (и около полутора - в памяти) оказывается излишне громоздким и медленным. Впрочем, первое играет роль только для всяких rescue-систем на дискетах (и прочих "мелких" носителях), а второе на современных машинах нечувствительно. Тем не менее, маленький и быстрый шелл Альмквиста (ash) может оказаться подходящим не только в спасательных целях, но и для всякого рода скрипtingа. Хотя работать в строке ash регулярно мне бы не хотелось...

Вторая причина поиска нового шелла - неудовлетворенность возможностями имеющегося. Эта проблема остро встает перед пользователями FreeBSD - уж очень убог его умолчальный login shell для обычного пользователя в плане интерактивной работы. Что особенно наглядно проступает в сравнении с tcsh, каковой по определению получает в свое распоряжение root-оператор. А потому и простой юзер может поддаться искушению и выбрать себе тот же tcsh - хотя бы ради единства стиля работы (ведь на настольной машине тот же юзер, как правило, сам себе root).

Должен заметить, что именно при первом приобщении к FreeBSD я и перепробовал целый ряд доступных в ней шеллов. Благо через систему портов или коллекцию пакетов они столь же легко удалялись, как и устанавливались. Не обошел я своим вниманием и tcsh - и в целом остался им доволен. Как интерактивная оболочка tcsh, на мой взгляд, несколько превосходит bash (хотя и не принципиально). Однако непривычность синтаксиса (или необходимость его изучения) способны отвратить от этого шелла. Тем паче, что редактирование системных скриптов или, тем более, сочинение собственных сценариев все равно требует обращения к sh-скрипtingу.

И, наконец, третья причина для изысканий - поиски идеала (а не они ли движут, осознанно или нет, изрядной долей пользователей свободных POSIX-систем?). Спору нет, bash - оболочка хорошая, но до такого идеала явно не дотягивающая, слишком много в нем направлено на достижения баланса компактности и функциональности. Должен сказать, что мои поиски идеала успехом увенчались. И потому наступает время обратиться к рассмотрению того шелла, который я полагаю лучшим в качестве интерактивного инструмента - Z-Shell, или просто zsh.

Zsh, или мой любимый шелл

Алексей Федорчук

В [прошлой статье](#) мы немало времени уделили проблеме выбора командной оболочки. И такой выбор сделали - пока просто поверив мне на слово. В настоящем же разделе я рассчитываю дать ему обоснование.

Представление главного героя

Как можно догадаться из названия заметки, главным героем ее является командная оболочка **Z-Shell** или, сокращенно, zsh, о которой на протяжении всего предшествующего повествования не было сказано ни слова. Настало время исправить это упущение.

Итак, zsh - оболочка из клана sh-совместимых, первоначально разрабатывавшаяся Паулом Фальстадом (Paul Falstad), ныне развивается в рамках [самостоятельного проекта](#) сообществом энтузиастов (Zsh Development Group) при координации Петера Стефенсона (Peter Stephenson). В отличие от bash, прямого (как, впрочем, и косвенного) отношения к GNU zsh не имеет, распространяется под собственной лицензией BSD-стиля, а, следовательно, является полностью свободной программой.

Существует мнение (и не только мое), что в zsh нашли свое воплощение все прогрессивные тенденции таких развитых оболочек, как bash и tcsh. И, ознакомившись с его возможностями, с этим трудно не согласиться - в zsh есть все, что было хорошего в тех обеих оболочках, но, если так можно выразиться, в превосходной степени.

Действительно, какими особенностями определяется в первую очередь удобство интерактивной работы в командной оболочке? В порядке, котором с ними сталкивается пользователь, это будут:

- автодополнение командной строки;
- возможности навигации по ней и ее редактирования;
- просмотр буфера истории команд;
- возможность минимизации ввода за счет использования псевдонимов.

Автодополнением клавишей **Tab** команд, частично введенных в ответ на приглашение оболочки, трудно удивить пользователей bash или tcsh. Столь же естественно, что при возможности безальтернативного дополнения именно оно и происходит, а при наличии некоторых альтернатив выводятся возможные варианты. Однако zsh идет дальше - и после вывода таковых в ответ на последующие нажатия клавиши табулятора начинает автоматический их перебор.

Автодополнение путей к файлам, выступающим в качестве аргументов команд, - тоже не бог весть какое новшество. И единственное отличие zsh от прочих оболочек выражается здесь в том, что и для путей действует автоматический перебор вариантов клавишей табулятора. А вот то, что автодополнение работает также и для аргументов команд, например, команды man - окажется приятной неожиданностью. Так, чтобы вызвать полное экранное руководство по zsh, достаточно набрать

```
$ man zsha  
и нажать табулятор, чтобы развернуть его до полного  
$ man zshall
```

Более того, автодополнению (и автоматическому перебору его возможностей) подвержены даже опции многих команд. Это особенно показательно для таких синтаксически сложных команд, как find. Так, последовательность

```
$ find / -na  
будет автоматически дополнена до  
$ find / -name
```

А после указания этого для указания опции действия можно ограничиться вводом символа дефиса
\$ find / -name filename -

и выбрать необходимое действие из предложенного списка. Например, print - вывод на экран, или exec - исполнение сторонней команды.

И еще один частный случай автодополнения уникален для zsh: развертывание путей при сокращенном их наборе. Так, что просмотреть содержимое каталога /usr/portage/distfiles, достаточно набрать в командной строке

```
$ ls /u/p/di  
и нажать клавишу табуляции: сокращенный ввод пути (разумеется, при безальтернативности онного) будет
```

автоматически развернут до полного.

Автодополнение в zsh гармонично сочетается с автокоррекцией (т.н. spelling командной строки). Конечно, и это само по себе не уникально. Однако проверка правильности ввода и автокоррекция в zsh распространяются не только на встроенные (как в bash) и даже внешние (как в tcsh) команды, но даже на пути и аргументы. Причем если автокоррекция становится назойливой (например, для команд типа cp или mv она порывается исправить аргументы на имена существующих файлов), ее можно отключить - и именно только для определенных команд.

Средства навигации по командной строке и ее автоматического редактирования - необходимое условие комфорта в интерактивной работе внутри оболочки. Здесь говорить, казалось бы, не о чем - управляющие клавишные последовательности для таких действий давно уже вошли в обиход всех командных оболочек,

претендующих на развитость. Однако и в этой области zsh есть чем похвастаться - в нем задействованы все комбинации клавиш для перемещения и удаления (как посимвольного, так и командными "словами" и фрагментами строки), которые существуют в bash и tcsh, причем - построенные по тем же принципам.

Управляющие последовательности в zsh построены по принципу сочетания клавиши **Control**+литера или **Meta**+литера, причем вторая комбинация обычно выступает в качестве "усиленного" варианта первой. Так, если **Control+D** удаляет символ в позиции курсора, то сочетание **Meta+D** проделывает это для всех символов от позиции курсора до конца командного слова.

Предусмотрены в zsh и клавищные комбинации для таких действий, как преобразование регистра литерных символов, "перетасовки" символов и командных "слов" в строке, заключения строки в кавычки (а при необходимости - и экранирования оных символами обратного слэша). Есть, конечно же, и комбинация для многоуровневой отмены ввода.

Действие большинства простых (двухклавищных) последовательностей дублируется "сложными", вида **Meta+литера-**Control**+литера**, которые прекрасно работают и при переключении на кириллическую раскладку клавиатуры.

Легко заметить, что управляющие последовательности в zsh реализованы с стилем emacs. Однако это - лишь один из возможных режимов, тот, который принят по умолчанию. При желании ничего не стоит переключить навигацию и управление в режим vi, если таковой кажется более привычным для пользователя. Оболочка zsh обладает всеми стандартными средствами доступа к буферу истории команд - перехода к началу и концу буфера истории, просмотра оного вперед и назад (как клавишами управления курсором, так и соответствующими управляющими последовательностями), обычного и т.н. наращиваемого поиска в обоих направлениях, исполнения выуженной из буфера команды с автоматическим переходом к следующей. Плюс - весьма изощренные способы вывода в строку отдельных фрагментов команд из буфера истории - например, отдельного командного "слова", начиная с последнего, с дальнейшим перебором "слов" буферизованных команд назад. Или - вывод полного списка команд из буфера с их последовательным перебором в том или ином направлении.

И, наконец, такое мощное средство минимизации пользовательского ввода, как псевдонимы команд (*aliases*). Разумеется, в zsh (как и в bash или tcsh) псевдоним может быть присвоен любой команде со сколь угодно длинным набором опций. Так, куда как проще раз и навсегда определить команду ls как псевдоним самой же себя, но с опциями -FG, нежели каждый раз вспоминать, как отличить в ее выводе каталоги от обычных файлов.

Однако zsh идет дальше: в нем это дополняется возможностью определения псевдонимов для командных конвейеров в форме опции -g (от *global aliases* - именно так именуется эта возможность). Так, всем известно, что для обеспечения постраничного вывода любой команды (например, той же ls) вывод этот нужно передать по конвейеру (pipe) программе-pager'у (less или more). Однако не лениво ли - каждый раз вводить что-нибудь вроде

```
$ ls | less
```

да еще и не забывать это делать? Если лениво - на помощь придут глобальные псевдонимы. Опять же раз и навсегда определяем, что опция -g со значением L есть псевдоним для конвейера '| less':

```
$ alias -g L='| less'
```

после чего имеем возможность, указывая ее после команды, требующей постраничного ввода, именно его и получать:

```
$ ls -g L
```

Казалось бы, не намного проще? Ах нет: ведь ничто не препятствует нам создать еще один, обычный, псевдоним для команды ls с этой опцией, например:

```
$ alias lp='ls -g L'
```

чтобы при необходимости постраничного вывода списка каталога именно к нему и прибегнуть.

Важный момент облегчения существования пользователя в любом шелле - вид приглашения командной строки, должна настройка которого может часто избавить от лишнего набора команд (уж от команды pwd я по возможности стараюсь избавиться именно таким образом). Так вот, zsh поддерживает несколько независимо настраиваемых обычных приглашений - обычное, или первичное, вторичное - для многострочных команд, "выделенное", приглашение при выводе вариантов автокоррекции и даже специальное "приглашение" в правой части командной строки. Которое, конечно, собственно приглашением не является, но позволяет вывести полезную информацию, например, текущее время или дату, номер виртуальной консоли, и т.д. Для пущей экспрессии настраивается также передача символов в любом приглашении - цветом ли, выделением, инверсией, подчеркиванием.

Очень полезная возможность - различение вида приглашения для обычного пользователя, получившего права root'a в результате команды su без опций (то есть без перечитывания профильных файлов администратора), от собственно пользовательского приглашения - дабы не забывал юзер о временностии своих полномочиях.

Надеюсь, что мне удалось убедить читателя в превосходных интерактивных возможностях оболочки zsh. Теперь стоит поговорить о функциональности, которая проявляется не только в интерактивной работе (но и, скажем, при сочинении скриптов). Функциональность же любой оболочки можно в первом приближении оценить по количеству встроенных в нее команд. То есть - команд, выполняемых внутри самого шелла, без порождения новых процессов, как это происходит при выполнении внешних команд. Очевидно, что такие

встроенные команды будут выполняться быстрее и отъедать меньше ресурсов. Что, конечно, не скажется при интерактивной работе на современных машинах, но вот в сложных сценариях - вполне может. В zsh поддерживается весь набор встроенных команд, стандартизованный для POSIX shell, большинство команд из развитых оболочек bash и tcsh, ну и, разумеется, специфичные для этого шелла команды. Общее число их превышает 80 - примерно столько же, сколько встроено в tcsh и bash, вместе взятые.

Очень интересна (и удобна) в zsh работа с командными конструкциями перенаправления. Здесь и множественное перенаправление вывода, когда результат выполнения команды направляется сразу в несколько файлов, и множественное перенаправление ввода - когда команда, напротив, получает аргументы последовательно из более чем одного файла, перенаправление без команды, когда конструкция типа

```
$ < filename
```

просто выведет на экран содержимое указанного файла - без привлечения команд типа cat или less.

При перенаправлении возможна группировка команд по шаблону. Так, файлы с именами вида file1 и file2 можно просмотреть одной командой

```
$ < file{1,2}
```

Перенаправление ввода/вывода может иногда заменять конвейеризацию команд. Так, конструкция вида

```
$ sort < file{1,2}
```

отсортирует содержимое обоих файлов точно так же, как это сделал бы конвейер команд

```
$ cat file1 file2 | sort
```

Наконец, еще одна специфическая особенность zsh - т.н. пред-исполнимая модификация команд (precommand modifier), осуществляющаяся перед их интерпретацией. Именно таким образом можно отменить чрезмерно навязчивую автокоррекцию аргументов для одной отдельно взятой команды, например, копирования:

```
$ nocorrect cp file1 file2
```

Легко видеть, что все изобилие возможностей zsh далеко выходит за рамки стандарта POSIX для командных оболочек. Однако, в подтверждение своего соответствия оному, zsh, наступая на горло собственной песне, способен к эмуляции POSIX Shell - для этого достаточно создать файл /bin/sh как символическую ссылку на исполнимый файл zsh, например:

```
$ ln -s /usr/bin/zsh /bin/sh
```

Впрочем, делать это следует только в случае полной уверенности, что все общесистемные скрипты полностью совместимы с zsh - а такая уверенность может быть только в том случае, если они написаны собственноручно:-). Я успешно применял zsh в качестве общесистемного шелла в самостройном Linux'e (по мотивам Linux from Scratch). Однако в других дистрибутивах (и тем более во FreeBSD) от этого лучше воздержаться.

Кроме того, имеется и некий режим совместимости с командными оболочками csh-клана (в нем я, впрочем, не разбирался).

Здесь перечислена лишь небольшая часть возможностей оболочки zsh, в частности, я не останавливался на его встроенных функциях, хотя именно они и есть та база, что обеспечивает все описанное (и не описанное) богатство возможностей. Не говорил я и о подгружаемых модулях (по типу plug-ins) - а ведь среди последних есть даже собственный ftp-клиент. Ибо для этого потребовалось бы пересказать всю экранную документацию к нему - более дюжины тап-страниц общим объемом (в *.gz-виде) свыше 250 Кбайт, плюс официальное руководство с сайта проекта, включающее в pdf-формате 260 страниц (к слову - автором этой документации является тот же Петер Стеффенсон).

Тем не менее надеюсь, что я убедил вас в том, что zsh - штука стоящая. Если так - то

Приступаем к установке

Оболочка zsh стандартно входит в большинство (я бы сказал - во все из мне известных) полнофункциональных дистрибутивов Linux, в качестве портов и пакетов доступна во FreeBSD и OpenBSD (на счет NetBSD - просто не помню), портирована даже в AtheOS (вернее - ее отприска Syllable). Так что проблем с ее получением быть не должно. Устанавливаем zsh штатным для данного дистрибутива методом и переходим к следующему разделу.

Если же почему-либо zsh в составе дистрибутива не обнаружился - отправляемся на [ftp://ftp.zsh.org/pub/](http://ftp.zsh.org/pub/) или какое-либо его зеркало (список - на <http://www.zsh.org>), можно - в каталог distfiles ftp-серверов FreeBSD или Gentoo. И качаем в свое удовольствие исходники последней стабильной версии (на данный момент - 4.07), или, при желании, разрабатываемой (ныне - 4.2) - различий в стабильности между ними я не заметил. Исходники zsh распаковываются и собираются обычным образом - ./configure, make, make install, никаких неожиданностей здесь не предвидится. Единственно, я предварительно, чисто для интереса, поинтересовался бы опциями конфигурирования -

```
$ ./configure --help
```

Из которых не побрезговал бы опцией --bindir=/bin - это будет полезно, если zsh будет использоваться как login shell.

В дистрибутивах Linux, предусматривающих автоматическое обновление общесистемных профильных файлов (например, в Gentoo), может возникнуть необходимость в том, чтобы zsh брал свои переменные окружения из какого-либо общего конфига, например, /etc/profile. Для этого при начальном конфигурировании исходников следует указать

```
$ ./configure --enable-zprofile=/etc/profile
```

И еще: ручная сборка zsh может потребоваться в некоторых пакетных дистрибутивах, в которых, будучи установленным из бинарников, он может работать неподобающим образом (о причинах я скажу позже). Теперь же, установив zsh тем или иным образом, делаем его своей пользовательским шеллом по умолчанию - login shell (одной из команд типа usermod, pw, chsh) и -

Начинаем настройку

Без этого нам, скорее всего, не обойтись. Дело в том, что в свежеустановленном zsh, мы имеем шанс не увидеть почти ничего из описанных выше прелестей - ни развертывания сокращений путей, ни автодополнений опций и аргументов, ни выразительных приглашений командной строки. Перед нами будет безликая строка с именем машины (типа localhost%), которая едва-то будет справляться с обычным автодополнением команд и путей (и то - не обязательно). Могут возникнуть проблемы даже с вызовом собственной экранной документации man zsh. Почему?

Дело в том, что zsh имеет очень богатый набор собственных конфигурационных файлов, о которых я скажу чуть ниже. Но при установке его эти файлы не помещаются автоматически ни в каталог /etc, ни в домашний каталог пользователя - то есть ни в одно из тех мест, где их можно было бы ожидать. Конечно, совсем без первичных настроек zsh не останется: он прекрасно воспринимает их из таких общесистемных профильных файлов, как /etc/profile, /etc/login и т.д. Однако, во-первых, для этого он должен быть собран должным образом. А во-вторых, и вести себя при этом он будет почти точно также, как и соответствующие оболочки (bash или tcsh, а то и /bin/sh - почему, например, во FreeBSD zsh по первости не способен даже к автодополнению - от одного из этих профильных файлов zsh унаследует и переменные окружения, включая MANPATH - почему подчас не сможет найти и свою собственную документацию).

Так что для придания zsh полного блеска следует прибегнуть к его собственным конфигурационным файлам. Правда, сначала придется отыскать их примеры из штатной поставки - должен заметить, что в разных системах они могут обнаружиться в весьма неожиданных местах. Так, во FreeBSD их штатное место - каталог /usr/local/share/examples/zsh, в дистрибутиве Gentoo Linux примеры оказываются в /usr/share/doc/zsh-XXX-rX/StartupFiles, в иных - вполне могут оказаться где-нибудь в районе /usr/share/zsh, и т.д.

Для минимизации времени на поиски выдам секрет - примеры конфигурационных файлов zsh, входящие в штатный комплект, называются - zlogin, zshenv и zshrc (возможно, с расширением .gz), и благодаря моей доброте :-) их не трудно будет отыскать командой find - за пределы каталога /usr файлы эти попасть не должны (даже в том случае, если, как в Gentoo, сам исполняемый файл zsh оказывается в каталоге /bin).

После изыска конфигов для сердца вольного есть два пути. Первый - простой, копируем их в наш домашний каталог в качестве dot-файлов (~/.zlogin, ~/.zshenv и ~/.zshrc, соответственно), после чего наслаждаемся жизнью. В ряде случаев этого достаточно, чтобы получить доступ к базовым (но очень даже расширенным, сравнительно с сородичами) возможностям этого шелла.

Разумеется, суперпользователь может скопировать эти файлы и в каталог /etc (без точек в имени) - в этом случае они будут определять конфигурацию zsh для всех пользователей, его запускающих (любым образом, о чем - чуть ниже).

Второй путь - попытаться разобраться, за что отвечают скопированные файлы и как они устроены. Это понадобится в двух случаях - а) для идеальной настройки своего нового шелла и б) если zsh работает не совсем так хорошо, как вы ожидали (например, не так, как описано выше).

Первый шаг на этом пути - задаться вопросом, а зачем zsh'у так много конфигов, если другие шеллы спокойно обходятся двумя (а то и одним, как /bin/sh). На это я отвечу, что конфигов в zsh вовсе не много, а очень много: в дополнение к трем примерным в разделе FILES его man-страницы можно найти упоминание еще о zprofile и zlogout (и, соответственно, ~/.zprofile и ~/.zlogout). А в ходе пользования им вы, скорее всего, увидите в своем каталоге еще и такие файлы, как ~/.zcompdump и ~/.zhistory.

Так для чего нам такое богатство? Чтобы разобраться в этом, вспомним о трех видах функционирования любого шелла - неинтерактивном, интерактивном и подвиде последнего - главном пользовательском (login shell). Так вот, файл /etc/zshenv (или ~/.zshenv) считывается при каждом запуске любого экземпляра zsh, независимо от того, происходит он интерактивно или опосредованно. Настройки файла /etc/zshrc (и ~/.zshrc) имеют силу для любого интерактивного запуска zsh. И, наконец, файлы /etc/zlogin и /etc/zprofile оба вместе (как и соответствующая им пара ~/.zlogin и ~/.zprofile) относятся только к тому экземпляру интерактивно запущенного zsh, который выступает в качестве login shell.

Зачем так сложно? А для того, чтобы можно было гибко (и индивидуально) настроить неинтерактивные, интерактивные и пользовательские экземпляры шеллов. Действительно, очевидно, что настройку неинтерактивного шелла влияет только содержимое файла /etc/zshenv (и ~/.zshenv), на настройку любого интерактивно запущенного экземпляра - уже он же вкупе с /etc/zshrc (и ~/.zshrc), тогда как поведение login shell определяется кумулятивным эффектом всех трех (или даже четырех) их пар.

Хорошо, но зачем же нам два конфига для login shell? - спросите вы меня. Ответ прост - из соображений совместимости с bash и tcsh. Для пояснения чего вернемся к истории вопроса. В первозданном шелле Борна существовал только один конфиг - /etc/profile (~/.profile) для любых экземпляров шелла. В bash к нему прибавился еще и файл /etc/bashrc (~/.bashrc) для интерактивного использования (считываемые, естественно, после предыдущего - как более молодой по происхождению).

В csh же набор конфигов был совсем иной. Там изначально существовали два конфига - /etc/csh.env (~/.csh.env) на все случаи жизни и /etc/login (~/.login) - в качестве конфигуратора login shell, считываемые именно в таком порядке.

В zsh же, дабы удовлетворить привычки пользователей любых предшествовавших шеллов, были включены оба "логируемых" конфига, причем порядок их считывания был унаследован от каждого из родителей. В результате получилась довольно сложная последовательность при запуске login shell:

zshenv -> zprofile -> zshrc -> zlogin

Причем каждый конфиг сначала, естественно, считывается из каталога /etc, а затем из домашнего каталога пользователя берется его аналог. Разумеется, если все четыре файла присутствуют (и там, и там). Что, сразу скажем, отнюдь не обязательно. Очевидно, что совместное использование zprofile и zlogin ни малейшего смысла не имеет. Просто бывшим пользователям bash привычней первая схема запуска login shell, бывшим приверженцам tcsh - вторая. Забегая вперед, замечу, что вообще пользователь может обойтись только одним конфигом в своем домашнем каталоге (например, ~/.zshrc для любого интерактивного экземпляра шелла - ведь login shell также будет интерактивным), а все общие настройки получать из общесистемного конфига (например, /etc/profile). Более того, в дистрибутиве Gentoo Linux именно так поступить лучше всего - к причинам вернусь позднее.

Осталось объяснить смысл остальных dot-файлов из пользовательского каталога. Каковой, впрочем, ясен из названий: ~/.zlogout - это сценарий, отрабатываем при выходе из login shell, ~/.zhistory хранит историю команд (это и есть ее буфер), а ~/.zcompdump (насколько я понимаю) - делает то же самое, но для встроенных функций zsh. Два последних файла возникают (при выполнении некоторых условий) сами собой, и речи о них больше почти не будет.

Разобравшись с назначением dot-файлов, можно, наконец, выполнить

Собственно конфигурирование

Процесс этот начнем с того, что отделим зерна от плевел, то есть решим: а какие же именно файлы нужно настроить. Для начала я оставил бы в покое все файлы из каталога /etc - вернее, просто не стал бы копировать туда примеры. Почему? Да потому, что во многих дистрибутивах Linux имеются либо предварительно настроенные общесистемные конфиги, либо предусмотрены средства для их автоматического создания и обновления (ниже я продемонстрирую это на примере Gentoo). А во FreeBSD в каталоге /etc принято хранить только профильные файлы общесистемного шелла (сиречь /bin/sh, тогда как настройка пользовательского шелла - сугубо личное дело пользователя).

Так что ограничиваемся только нашим домашним каталогом. Однако и здесь многое зависит от ОС и дистрибутива. Так, в user-ориентированных дистрибутивах Linux первый кандидат на удаление (или не-копирование) - файл ~/.zshenv. Конечно, его можно создать в предельно облегченном виде (например, настройка приемов неинтерактивной работы здесь абсолютно лишняя). Характерно, что в штатном примере zshenv присутствует только переменная PATH. Но ведь ее можно получить из общесистемного /etc/profile, не так ли?

Особенно лишним выглядит ~/.zshenv для пользователей Gentoo. Как можно прочитать в [соответствующей документации](#), здесь существует прекрасный механизм установки общесистемных переменных - env-update, автоматически обновляющий главный профильный конфиг /etc/profile. Так что при отсутствии ~/.zshenv все необходимое (в актуальном виде) будет браться оттуда. В присутствии же его - zsh откажется от считывания /etc/profile, в результате чего такие переменные, как локаль, EDITOR, PAGER и т.д., придется определять дополнительно.

А вот во FreeBSD и тех дистрибутивах Linux, которые (как, например, CRUX и Archlinux) не имеют средств автоматизации установки общесистемных переменных, файл ~/.zshenv оказывается практически необходимым. И ниже я попытаюсь обосновать это (а заодно и продемонстрировать, что в нем может содержаться).

Далее, решаем, а нужен ли нам отдельный конфиг для login shell, и если нужен - то какой из них - ~/.zlogin или ~/.zprofile. Теоретически, можно обойтись и без того, и без другого. Уж без последнего - так наверняка, не зря же его нет ни среди штатных примеров, ни среди многочисленных конфигов, которыми поделились с народом активные пользователи zsh (см. источники информации). А файл ~/.zlogin у меня, например, есть, хотя содержание его может варьировать от системы к системе.

Не помешает и файл ~/.zlogout - его возможное содержание также будет рассмотрено позднее.

Остается главный (по крайней мере, самый большой) конфиг - ~/.zshrc. Что с ним делать? Это легко уяснить, ознакомившись со штатным примером.

А в примере этом мы найдем и настройку приглашений командной строки, и определение псевдонимов - обычных и глобальных, для конвейеров команд, и определение переменной cdpath, и величину буфера истории команд, и многое, многое другое.

Описывать все опции из примера было бы долго и скучно. А потому здесь я остановлюсь только на тех, которые создают удобство при интерактивной работе и определяют внешний вид нашего шелла.

Начнем с последнего, то есть приглашения командной строки. В этом качестве могут использоваться:

- полное или сокращенное имя хост-машины (последнее принято по умолчанию для первичного);
- путь к текущему каталогу в различных формах;
- номер текущей команды в буфере истории или строки в данном сеансе работы;
- имя пользователя, или командной оболочки;
- номер текущего терминала;
- дата и время в разных форматах;
- индикация работы от лица суперпользователя);
- любые символы типа стрелок, крышечек, скобочек;
- текстовые сообщения (например, поздравление с началом трудового процесса);
- многое другое (подробности - в man zshmisc).

Плюс к этому приглашения могут быть оформлены визуально различно: выделением жирным шрифтом (boldface mode) или повышенной яркостью (underline mode), инвертированием текста/фона (standout mode), а также цветами. Все это позволяет добиться максимальной информативности приглашения и его внешней выразительности.

Из опций настройки интерактивности - вспомним о том, что обеспечивает расширенные возможности автодополнения. А обеспечиваются они строками, следующими после комментария:

```
# Setup new style completion system. To see examples of the old style (compctl
# based) programmable completion, check Misc/compctl-examples in the zsh
# distribution.
```

и имеющими вид
autoload -U compinit
compinit

Вообще, штатный пример файла zshrc очень неплохо прокомментирован - по аглицки, но разобраться можно:-).

Далее - опции управления историей команд. Здесь для начала следует определить файл, в котором эта история будет храниться (в примере по умолчанию таковой отсутствует, и никакая история по умолчанию не сохраняется по выходе из сеанса zsh):
HISTFILE=~/.zhistory

Теперь - объем нашей исторической памяти, задаваемый двумя опциями -
HISTSIZE=1000

определяющей память текущего сеанса, и
SAVEHIST=1000

устанавливающая размер буфера, сохраняемого в HISTFILE. Очевидно, что первую бессмысленно делать больше второй (рекомендуются равные значения - в моем примере они взяты просто с потолка).

Для обеспечивая наращивания файла истории после каждого сеанса работы (в пределах установленной квоты) определяем:

```
setopt APPEND_HISTORY
```

Объем исторической памяти можно установить любым, однако не стоит расходовать его на повторение дублирующихся команд, ошибочными нажатиями **Enter** в пустой строке и т.д.:

```
setopt HIST_IGNORE_ALL_DUPS
setopt HIST_IGNORE_SPACE
setopt HIST_REDUCE_BLANKS
```

Можно настроить и еще много чего, но на этом пока закончим (некоторые опции будут подробнее рассмотрены в следующем разделе). Потому что наступило время подумать, а что представляют из себя установленные нами опции (типа `compinit` и т.д.)? А представляют они собой по преимуществу встроенные функции из комплекта `zsh`, которые в изобилии можно обнаружить в каталоге `/usr/share/zsh/4.0.6/functions`. И именно благодаря этим функциям и удается все настроить.

А еще - становится понятным, почему иногда `zsh` в пакетных дистрибутивах, устанавливаясь из бинарников, ведет себя не вполне подобающим образом. Да именно потому, что комплект функций в поставке и их использование в файлах примеров не вполне идентичны (по крайней мере, мне такие случаи встречались). При сборке же из исходников с сайта проекта, насколько мне довелось наблюдать, идентичность эта гарантирована. Тот же результат, естественно, достигается и в Source Based дистрибутивах (или, скажем, в системе портов FreeBSD).

Вообще говоря, я долго не мог понять, почему пользователи не любят `zsh`. И в конце концов пришел к выводу - именно потому, что сборщики дистрибутивов подчас не умеют его готовить. Это - не только мое мнение: многие пользователи пакетных дистрибутивов свидетельствуют, что установленный из "коробки" `zsh` в их системах работает весьма криво. Так что позволю себе привести

Личный рецепт приготовления

Как-то на одном форуме промелькнул вопрос - а как сделать, чтобы в командной строке `zsh` клавиши типа **Delete**, **End**, **Home** вели себя нормально (по умолчанию они этого делать не собираются). У меня до этого долго не доходили руки - я в этих целях привык к клавишным комбинациям (см. следующий раздел). Однако некоторое чувство дискомфорта преследовало: как же так, какой-то там `bash` умеет нормально обращаться с клавишами, а любимый `zsh` - не умеет. А тут и случай представился: во время затянутой в рамках мегатестирования тотальной пересборки Qt/KDE/иже с ними временем образовалось - вагон и маленькая тележка. И я наконец-то разобрался с клавишами в `zsh`. О чем с удовольствием рапортую в этом разделе. Из многочисленных `zsh`-конфигов я (в настоящее время) использую: `~/.zshenv`, считываемый при каждом запуске экземпляра оболочки (интерактивном и неинтерактивном), `~/.zshrc`, считываемый при каждом интерактивном ее запуске, и `~/.zlogin`, считываемый при каждом запуске `zsh` в качестве `login shell`. При этом установка переменных окружения происходит в стиле C-Shell: сначала из `~/.zshenv`, затем из `~/.zshrc` и, наконец, из `~/.zlogin`.

Так что первым в моей схеме идет `~/.zshenv`. Он оказывает воздействие только при shell-скрипtinge. Поэтому у меня он очень мал. В Linux-варианте (для Archlinux) он выглядит так:

```
#  
# My ~/.zshenv for Linux  
#export PATH="/bin:/usr/bin:\  
/usr/local/bin:/usr/X11R6/bin:\  
/opt/bin:/opt/kde/bin"
```

Bo FreeBSD все примерно то же, но, естественно, `/opt` в `PATH` включать не требуется:

```
#  
# My ~/.zshenv for FreeBSD  
#  
PATH="/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin"  
QTDIR=/usr/local/qt  
PATH="$PATH:$QTDIR"  
MANPATH="$QTDIR/doc/man:$MANPATH"  
LD_LIBRARY_PATH="$QTDIR/lib:$LD_LIBRARY_PATH"  
export QTDIR PATH MANPATH LD_LIBRARY_PATH
```

Понятно, что оба файла можно записать поизящней (с точки зрения формы), но - так уж они исторически скились.

Основное внимание я уделяю файлу `~/.zshrc`, так как им определяется поведение `zsh` не только при авторизации в консоли, но и при запуске терминальных окон в Иксах.

```
#  
# My ~/.zshrc  
#  
# Path для поиска командой cd: то есть вместо cd $HOME/docs/editors/  
# можно набирать просто cd editors  
cdpath=(~/media ~/docs)
```

```
## Установка нормального поведения клавиш Delete, Home, End и т.д.:  
case $TERM in  
    linux)  
        bindkey "^[2~" yank  
        bindkey "^[3~" delete-char  
        bindkey "^[5~" up-line-or-history  
        bindkey "^[6~" down-line-or-history  
        bindkey "^[1~" beginning-of-line  
        bindkey "^[4~" end-of-line  
        bindkey "^[e" expand-cmd-path ## C-e for expanding path of typed command  
        bindkey "^[A" up-line-or-search ## up arrow for back-history-search
```

```

bindkey "^[B" down-line-or-search ## down arrow for fwd-history-search
bindkey " " magic-space ## do history expansion on space
;;
*xterm*|rxvt|(dt|k|E)term)
bindkey "^[2~" yank
bindkey "^[3~" delete-char
bindkey "^[5~" up-line-or-history
bindkey "^[6~" down-line-or-history
bindkey "^[7~" beginning-of-line
bindkey "^[8~" end-of-line
bindkey "^[[e" expand-cmd-path ## C-e for expanding path of typed command
bindkey "^[A" up-line-or-search ## up arrow for back-history-search
bindkey "^[B" down-line-or-search ## down arrow for fwd-history-search
bindkey " " magic-space ## do history expansion on space
;;
esac
# Примечание: если, скажем, в KDE для konsole
# выбрать тип Linux console, необходимости
# во второй секции нет. Консоль Linux в KDE можно

спокойно установить и во FreeBSD.
# К слову сказать, совсем уж нормального
# поведения клавиш в syscons я до сих пор
# не добился:-)

# Use hard limits, except for a smaller stack and no core dumps
unlimit
limit stack 8192
limit core 0
limit -s

# Установка атрибутов доступа для
# вновь создаваемых файлов
umask 022

# Исправление поведения less - для ликвидации
# лишних Esc и прочего безобразия при
# выводе тап-страниц.
# Насколько мне известно, нужно только в некоторых
# дистрибутивах Linux
export LESS="-R"

# Установка alias'ов

## alias'ы для команд, не требующих коррекции, но требующих подтверждения

alias mv='nocorrect mv -i'
# переименование-перемещение с подтверждением

alias cp='nocorrect cp -iR'
# рекурсивное копирование с подтверждением

alias rm='nocorrect rm -i'
# удаление с подтверждением

alias rmf='nocorrect rm -f'
# принудительное удаление

alias rmrf='nocorrect rm -fR'
# принудительное рекурсивное удаление

alias mkdir='nocorrect mkdir'
# создание каталогов без коррекции
## Примечание: если не определить здесь nosorrect,
## zsh будет настойчиво предлагать подстановку
## существующих имен при создании каталогов,
## копировании и т.д.

## Разные полезные (IMHO) alias'ы
alias h=history
alias grep=egrep

### вывод свободного дискового пространства
### в мегабайтах
alias df=df -m'

### Представление вывода less в more-подобном виде
### (с именем файла и процентом вывода)
alias less='less -M'

```

```

### Русский словарь для ispell по умолчанию
alias ispell='ispell -d russian'

## aliases для команды ls<

### показ классификации файлов в цвете и
### символически (Linux)
alias ls='ls -F --color=auto'
### Bo FreeBSD достаточно
alias ls='ls -F'

### вывод в глинистом формате
alias ll='ls -l'

### вывод всех файлов, включая dot-файлы, кроме . и ..
alias la='ls -A'

### вывод всех файлов в глинистом формате, включая inodes
alias li='ls -ial'

### вывод только каталогов
alias lsd='ls -ld */(DN)'

### вывод только dot-файлов
alias lsa='ls -ld .*'

# Установка глобальных псевдонимов
# для командных конвейеров
alias -g M='|more'
alias -g L='|less'
alias -g H='|head'
alias -g T='|tail'
alias -g N='2>/dev/null'

# Ниже даны опции, относящиеся к функциям zsh,
# которыми собственно и определяется мощь этой оболочки

# Shell functions
setenv() { typeset -x "${1} ${1:+} ${((@))} ${2:#}" }

# csh compatibility
freeload() { while (( $# )); do; unfunction $1; autoload -U $1; shift; done }

# Where to look for autoloaded function definitions

fpath=($fpath ~/.zfunc)

# Autoload all shell functions from all directories in $fpath (following
# symlinks) that have the executable bit on (the executable bit is not
# necessary, but gives you an easy way to stop the autoloading of a
# particular shell function). $fpath should not be empty for this to work.
for func in ${^fpath/*(N-.x:t)}; autoload $func

# automatically remove duplicates from these arrays
typeset -U path cdpath fpath manpath

# Указание путей к man-страницам.
## Linux:
manpath="/usr/man:/usr/share/man:\n/usr/local/man:/usr/X11R6/man:/opt/qt/doc"

## FreeBSD:
manpath="/usr/share/man:/usr/local/man:/usr/X11R6/man"

export MANPATH

# Список хостов, к которым будет применяться
# автодополнение при наборе в командной строке
# например, как аргументов браузера или
# ftp-клиента (see later zstyle)
hosts=(`hostname` ftp.freebsd.org ftp.archlinux.org)

# Установка вида приглашения

## Обычное приглашение вида ~%=>
## (каталог от домашнего - пользователь/root - стрелка)
PROMPT='%~%#=> '


## Приглашения для второй линии многострочных команд

```

```

## вида #_строки>
PROMPT2=%i%U> '

## Приглашение с правой стороны экрана вида
## 19:15 vc/5 (время - номер консоли)
RPROMPT=' %T %y%b'

# Всякие переменные

## файл истории команд
## если не указан, история не будет сохраняться
## при выходе из сеанса
HISTFILE=~/zhistory

## Число команд, сохраняемых в HISTFILE
SAVEHIST=5000

## Число команд, сохраняемых в сеансе
HISTSIZE=5000
## Примечание:
## рекомендуются равные значения для
## SAVEHIST и HISTSIZE

DIRSTACKSIZE=20

# Опции истории команд

## Дополнение файла истории
setopt APPEND_HISTORY

##忽 置略所有重复命令
setopt HIST_IGNORE_ALL_DUPS

##忽 置略所有空格
setopt HIST_IGNORE_SPACE

## Удалять из файл истории пустые строки
setopt HIST_REDUCE_BLANKS

# Установка-снятие опций шелла
setopt notify globdots correct pushdtohome cdablevars autolist
setopt correctall autocd recexact longlistjobs
setopt autoresume histignoredups pushdsilent noclobber
setopt autopushd pushdminus extendedglob rquotes mailwarning
unsetopt bgnice autoparamsslash

## Отключение звукового сигнала
## при ошибках
setopt No_Beep

## Не считать Control+C за выход из оболочки
setopt IGNORE_EOF

# Autoload zsh modules when they are referenced
zmodload -a zsh/stat stat
zmodload -a zsh/zpty zpty
zmodload -a zsh/zprof zprof
zmodload -ap zsh/mapfile mapfile

# Опции общего поведения
# bindkey -v # режим навигации в стиле vi
bindkey -e # режим навигации в стиле emacs

bindkey '' magic-space # also do history expansion on space
bindkey '^I' complete-word # complete on tab, leave expansion to _expand

# Для разворота сокращенного ввода типа cd d/e в docs/editors
autoload -U compinit
compinit

# Completion Styles

# list of completers to use
zstyle ':completion:*:::' completer _expand _complete _ignored _approximate

# allow one error for every three characters typed in approximate completer
zstyle -e ':completion:*:approximate::*' max-errors \
'reply=( $((( ${#PREFIX}+${#SUFFIX}/3 )) numeric )'

# insert all expansions for expand completer

```

```

zstyle ':completion:*:expand:' tag-order all-expansions
# formacodeing and messages
zstyle ':completion:' verbose yes
zstyle ':completion:*:descriptions' format "%B%d%b"
zstyle ':completion:*:messages' format "%d"
zstyle ':completion:*:warnings' format 'No matches for: %d'
zstyle ':completion:*:corrections' format "%B%d (errors: %e)%b"
zstyle ':completion:' group-name ""
# match uppercase from lowercase
zstyle ':completion:' matcher-list 'm: {a-z}={A-Z}'
# offer indexes before parameters in subscripts
zstyle ':completion:*:-subscript-:' tag-order indexes parameters

# command for process lists, the local web server details and host completion
#zstyle ':completion*:processes' command 'ps -o pid,s,nice,stime,args'
#zstyle ':completion*:urls' local 'www' '/var/www/htdocs' 'public_html'
zstyle '*' hosts $hosts

# Filename suffixes to ignore during completion (except after rm command)
zstyle ':completion:*:(rm):*:files' ignored-pacodeerns '*?.o'*??.c~' \
'*?.old'*?.pro'
# the same for old style completion
#fignore=(.o .c~ .old .pro)

# ignore completion functions (until the _ignored completer)
zstyle ':completion*:functions' ignored-pacodeerns '_'

# Флаги оптимизации для gcc
CFLAGS="-O3 -march=pentium4 -fomit-frame-pointer \
-funroll-loops -pipe -mfpmath=sse -mmmx -msse2 -fPIC"
CXXFLAGS="$CFLAGS"
BOOTSTRAPCFLAGS="$CFLAGS"
export CFLAGS CXXFLAGS BOOTSTRAPCFLAGS

И, наконец, файл ~/.zlogin. Что осталось неохваченным в ~/.zshrc и требуется только при авторизации в системе? Правильно, пользовательские переменные для определения терминала, редактора, пейджера и т.д.

#
# My ~/.zlogin for FreeBSD
#
from 2>/dev/null
EDITOR=joe
PAGER=less
TERM=${TERM:-cons25r}
export EDITOR PAGER TERM

Кроме того, в Linux'е здесь же резонно установить locale (во FreeBSD локаль лучше определять через класс пользователя). И потому для Linux - дополнение:
# My ~/.zlogin appendix for Linux
#
# Установка всех локально-зависимых переменных,
# кроме LC_ALL
export LANG="ru_RU.koi8r"

# Установка десятичной точки
# вместо запятой
# (требуется для некоторых счетных программ)
export LC_NUMERIC="POSIX"

Да, самое последнее - файл ~/.zlogout, отрабатываемый по завершении сеанса пользовательского shell'a. У меня от включает две строчки
sync
clear
назначение которых более чем понятно (синхронизация дисковых кэшей и очистка экрана).

Это мои пользовательские конфиги. Почти те же самые я использую и для root'a, с минимальными корректировками. Так, переменная path в /root/.zshenv дополняется значениями
/sbin:/usr/sbin:/usr/local/sbin

В /root/.zshrc опция cdpath имеет вид
cdpath=(/etc /usr)

Кроме того, во FreeBSD здесь я исключаю переменные CFLAGS, CXXFLAGS и BOOTSTRAPCFLAGS, так как от лица root'a все собирается через систему портов, где их аналоги определяются в файле /etc/make.conf.

А в /root/.zlogin локаль (в Linux) установлена как
export LANG="POSIX"

так как некоторые программы упорно не желают собираться при какой-либо иной.

```

Заключительные замечания

Я уже упоминал, что проект zsh прекрасно документирован, один User's Guide чего стоит. Однако внимание прессы, как онлайновой, так и бумажной, к нему явно недостаточно. И потому из дополнительных источников информации на ум приходит только (если не считать отрывочных упоминаний в паре книжек по Linux) статья Мэтта Чапмена (Macode Chapman) - [Curtains up: introducing the Z shell](#). Да еще недавно появился на русском языке материал Алексея Отта [Командный процессор Zsh](#)

Много полезного можно узнать из анализа dot-файлов различных пользователей, ссылки на которые отыскиваются как на www.zsh.org, так и в иных местах. Собирать все эти ссылки мне было лениво (да подчас я и не помню, где их отыскал), так что все свои находки я собрал воедино в виде отдельного [тарбала](#).

Кое что о csh и tcsh

Алексей Федорчук

Должен сразу оговориться, что я не являюсь знатоком csh и tcsh. Правда, в период моего приобщения к BSD-системам я некоторое время попользовался последней. И она даже мне понравилась. Однако потом я прочно перешел на zsh, и до tcsh руки уже не доходили,

Тем не менее, кое-какие материалы по tcsh у меня тогда поднакопились. Их-то я и решил собрать на этой странице. В расчете на то, что кто-нибудь из любителей этой оболочки напишет сочинение, восполняющее мои многочисленные пробелы...

Командная оболочка csh пришла к нам из BSD-мира и традиционно пользуется популярностью среди пользователей этих систем. Правда, изначальный ее вариант (собственно csh) не принадлежит к силу свободно распространяемых программ. И потому в открытых BSD-клонах используется ее Open Sources разновидность - оболочка tcsh. Которую, впрочем, никто не запрещает применять и в Linux'e.

Правда, теоретически во FreeBSD в качестве стандартной пользовательской принятой оболочки /bin/sh. Однако для суперпользователя там по умолчанию устанавливается именно tcsh. Каковая выступает в качестве стандартной и OpenBSD.

Оболочка csh в русскоязычной литературе почти не описана (единственный известный мне документ по этому поводу - статья [М.П.Крутикова](#)). Что же касается tcsh - по ней в Рунете, пожалуй, что и вообще ничего не найти. Хотя она объединяет, похоже, преимущества синтаксиса csh и удобные средства настройки и интерактивной работы bash. Что и подвигло меня на написание этой заметки.

Особенности семейства C-Shell

Как уже говорилось, внешние различия между основными семействами командных оболочек лежат в первую очередь в области синтаксиса собственного их языка. В клонах Bourne Shell язык этот ни на что, насколько я понимаю, особенно не похож. В оболочке C-Shell и ее производных синтаксис встроенного языка, как и следует из названия, схож с таковым всамделишнего языка программирования C (хотя не следует думать, что от этого сам по себе язык имеет что-то общее с C).

C-подобный синтаксис встроенного языка csh (и tcsh) обеспечивает существенно больший лаконизм сценариев, чем в скриптах shell-совместимых интерпретаторов. Правда, именно при создании сценариев использование оболочки csh может быть ограничено ее несовместимостью со стандартом POSIX.

Различия в синтаксисе между семействами sh и csh отражают различие их обращения с условными выражениями. В классических шеллах это - просто последовательности команд (подобные конвејерам), в которых выполнение каждой последующей определяется успешным или неуспешным завершением предыдущей. В csh же они представляют собой вычисляемые арифметические или логические выражения. Далее, важное с точки зрения пользователя различие - обращение с путями к исполняемым файлам. Клоны шелла Борна при вводе команды перечитывают состав каталогов, включенных в качестве значений переменной PATH. В клонах же csh эти значения хранятся в чем-то типа собственного буфера, именуемого хэш-таблицей. В результате чего достигается выигрыш в быстродействии исполнения внешних команд. Обратная сторона - при добавлении к одному из каталогов переменной PATH нового файла (типичный случай - при установке новой программы) оболочка csh его просто не увидит; то есть для вызова такой новой программы в текущем сеансе придется указывать полный путь к ее исполнимому файлу. Или - перестроить хэш-таблицу, для чего предназначена специальная команда `rehash`.

Наконец, в csh по иному определяются переменные. Если во всех POSIX-шеллах для этого достаточно задать имя и значение, то здесь этой цели служит специальная команда `set`, например:

```
set EDITOR=joe  
определит редактор по умолчанию. Впрочем, таким образом будет задана только переменная оболочки -  
средств экспорт их в среду в csh не имеется. Для задания переменных среды существует отдельная команда  
setenv:  
setenv EDITOR=joe
```

Общая характеристика tcsh

Командная оболочка tcsh определяется как вариант C-Shell с возможностями дополнения имен файлов и редактирования командной строки - нельзя не отметить, что в оригинальной csh они развиты существенно слабее, чем, скажем, в bash. Тем не менее, tcsh полностью совместима со своим прототипом.

Как и все другие оболочки, tcsh объединяет в себе интерактивный командный процессор и интерпретатор собственного языка сценариев, превращающий ее в простую в обращении, но весьма мощную среду программирования.

Как следует из названия, одна из основных функций любой командной среды - исполнение команд, внешних (то есть независимых программ) и встроенных. Встроенные и внешние команды могут иногда дублировать функции друг друга, но при прочих равных условиях применение первых - предпочтительней, они выполняются быстрее. И набор встроенных команд - это то, что, помимо всего прочего, отличает командные среды друг от друга и определяет их функциональность.

Среда tcsh содержит достаточно большое (говорят, больше 50, я считаю поленился) количество встроенных команд. Полный их список можно получить с помощью команды builtins (к слову сказать, также встроенной), ответом на которую будет список вроде этого:

```
: @ alias alloc bg bindkey break  
breaksw builtins case cd chdir complete continue  
default dirs echo echotc else end endif  
endsw eval exec exit fg filetest foreach  
glob goto hashstat history hup if jobs  
kill limit log login logout ls-F nice  
nohup notify onintr popd printenv pushd rehash  
repeat sched set setenv secodec secodey shift  
source stop suspend switch telltc time umask  
unalias uncomplete unhash unlimit unset unsetenv wait  
where which while
```

Все эти команды могут использоваться как в интерактивном режиме, так и в составе сценариев (скриптов).

Описание команд можно найти в экранной документации. На некоторых наиболее используемых из них я буду останавливаться по ходу дела.

Основные возможности, предоставляемые tcsh в интерактивном режиме, помимо собственно исполнения команд, включают:

- редактирование командной строки;
- дополнение слов (word completion) - как для путей, так и для команд;
- хранение и воспроизведение истории команд;
- управление текущими задачами (job control).

Редактор командной строки предоставляет средства навигации внутри нее, с возможностью изменения отдельных знаков и компонентов команд, их опций и аргументов.

Навигация по командной строке и ее редактирование осуществляется двумя различными способами. Первый - использование стандартных клавиш управления курсором, таких, как **Left** и **Right**, **Home** и **End**, для навигации, и клавиш **Delete** и **Backspace** - для редактирования. Достоинство его - в простоте, вернее, в привычности: в ряде случаев эти клавиши ведут себя так же, как и в программах для DOS/Windows. Однако - далеко не всегда: на многих типах терминалов хоть какая-то из этих клавиш (а то и все сразу) обнаруживают аномальные особенности поведения.

Этого недостатка лишен второй способ навигации и редактирования - с использованием специальных клавишных комбинаций. Каковые на всех известных мне консолях ведут себя практически идентично. И к тому же предоставляют, по сравнению со стандартными клавишами, дополнительные возможности.

Управляющие комбинации (bindkeys) в большинстве случаев имеют вид **Control+литера** (то есть литерная клавиша нажимается при нажатой клавише **Control**) или **Escape-литера** (когда литерная клавиша нажимается непосредственно сразу клавиши **Escape**). Все они не чувствительны к регистру и, насколько мне удалось выяснить, также и к раскладке клавиатуры (то есть работают, вне зависимости от переключения, например, с латиницы на кириллицу и обратно).

Полный список управляющих комбинаций для tcsh может быть получен командой binkdkey

а наиболее употребимые из них я собрал в следующем параграфе. Здесь отмечу лишь, что наряду с обеспечиваемыми стандартными клавишами перемещениями курсора, такими, как:

- **Control+A** - перемещение курсора в начало строки;

- **Control+E** - перемещение курсора в конец строки;
- **Control+F** - перемещение курсора на один знак вперед;
- **Control+B** - перемещение курсора на один знак назад;

управляющие комбинации дают возможность перемещаться на одно слово вперед или назад (**Escape-F** и **Escape-B**, соответственно), в предыдущую позицию курсора (**Control+X-X**), удалять целиком слово (**Escape-D**) или часть строки после курсора, перемещать знаки (**transpose-chars**, **Control+T**), изменять регистр знаков и многое другое. А поскольку под все эти операции задействованы только клавиши основной части клавиатуры, скорость их выполнения - непревзойденная (при наличии некоторого навыка, доведенного, желательно, до рефлекторного уровня).

Следующая неоценимая возможность tcsh - дополнение слов. Которое работает как для команд, так и для имен файлов и путей к ним. То есть при наборе первых знаков команды (или файла) соответствующее действие (например, нажатие клавиши табуляции) автоматически дополняет недостающие знаки. Если, конечно, набранных знаков хватает для однозначной идентификации. Если же не хватает - есть возможность просмотреть списков доступных вариантов и выбрать из них подходящий (эта функция называется **autolist**). Как обычно, дополнение слов выполняется двояким способом - или клавишей **TAB**, или управляемыми комбинациями. Из них отметим **Control+I** - собственно дополнение слова, и **Control+D** - вызов списка вариантов для дополнения; в последнем случае курсор обязательно должен стоять после последнего введенного символа - иначе эта комбинация сработает на удаление знака под курсором.

История команд подразумевает, что некоторое количество ранее введенных команд сохраняется в специальном буфере, и может быть вызвано для просмотра, исполнения или редактирования.

Для этого можно использовать клавиши управления курсором - **Up** (назад) и **Down** (вперед), с помощью которых как бы "пролистываются" по одной все ранее введенные команды. Аналогичного результата можно добиться и управляемыми комбинациями - **Control+P** и **Control+N** или **Escape+P** и **Escape+N**, каждая пара из которых является аналогом пары **Up** и **Down**, соответственно.

Кроме этого, историю эту можно просмотреть с помощью встроенной команды **history**, которая выдаст нумерованный список всех выводившихся ранее (в количестве, определенном в файле конфигурации среды) команд, например:

```
> history
1 17:19 logout
2 17:57 history
3 17:57 pwd
4 17:57 ls
5 17:57 ls -laFG
6 17:57 history
```

Любая из них вызывается в командную строку с помощью конструкции
!#

где # - порядковый номер команды в списке. Как и во всех прочих развитых оболочках, команду из истории можно отредактировать и (или) запустить на исполнение.

Далее я приведу нечто вроде краткого справочника по оболочке tcsh. В нем рассмотрены управляемые последовательности, встроенные команды и конфигурационные файлы.

Управляющие последовательности

Список управляемых клавишных комбинаций (последовательностей, bindkeys) вызывается встроенной командой **bindkey**. Существует три типа последовательностей: **Control+символ**, **Meta+символ**, **Meta+Control+символ**. **Meta+символ** как бы "усиливает" действие последовательности **Control+символ**. Например, **Control+b** - перемещение на один символ назад, **Meta+b** - перемещение на одно "слово" назад. В качестве клавиши **Meta** на обычных клавиатурах работает нажатие и отпуск клавиши **Escape**.

Навигация по строке

Control+F

Перемещение курсора на один символ вперед.

Control+B

Перемещение курсора на один символ назад.

Meta+F

Перемещение курсора на одно "слово" команды (разделитель командного слова - пробел или слэш) вперед.

Meta+B

Перемещение курсора на одно "слово" назад.

Control+A

Перемещение курсора в начало строки.

Control+E

Перемещение курсора в конец строки.

Control+X

Sequence-lead-in (последовательное перемещение курсора в начало/конец строки и обратно).

Редактирование строки

Control+D

Это многозначная последовательность. Удаление символа в позиции курсора, если последний находится внутри строки (не в пустой строке и не в конце ее, см. далее).

Control+H

Удаление символа перед позицией курсора.

Control+W Meta+Control+H

Удаление "слова" перед позицией курсора.

Meta+D

Удаление "слова" после позиции курсора.

Control+K

Удаление части строки после позиции курсора (то есть вправо).

Control+U

Полная очистка командной строки, в случае многострочной команды - только текущей (на которой находится курсор).

Control+I Meta+Control+I Meta+/-

Дополнение "слова" (команды или пути) целиком или до последнего совпадающего для нескольких команд (путей) символа.

Control+D

При положении курсора в конце строки (но не в пустой строке) - вывод списка возможных автодополнений "слова" (autolist).

Meta+Control+D

Вывод списка возможных автодополнений "слова" (autolist). В отличие от последовательности Control+D, работает вне зависимости от положения курсора в строке.

Meta+_

Копирование в текущую строку последней введенной команды.

Control+J Control+M

Новая линия. Обычно эквивалентно нажатию клавиши Enter (то есть запуску команды на исполнение).

Однако но если в конце строки стоит символ \ (обратный слэш), эта комбинация вызывает переход на новую строку с выводом вторичного приглашения, например:

```
$fromwin -v -o ~/dir\  
? file_name
```

где \$ - первичное приглашение, ? - вторичное, после которого можно продолжать ввод команд, опций и аргументов.

Control+T

Перестановка символов (справа налево).

Meta+C

Капитализация (перевод в верхний регистр) единичного символа под курсором.

Meta+L

Перевод символов от позиции курсора до конца "слова" в нижний регистр.

Meta+U

Перевод символов от позиции курсора до конца "слова" в верхний регистр.

Meta+\$

Спеллинг введенной команды с автоматическим исправлением.

Управление процессами и история команд

Control+D

В пустой строке - завершение сеанса оболочки, аналогично exit или logout. Такое действие этой последовательности можно отменить в настройках.

Control+C

Завершение текущей задачи с возвращением приглашения командной строки.

Control+Z

Перевод текущей задачи в фоновый режим.

Control+Y

Вроде бы повторение последней выполненной команды. Не всегда срабатывает.

Meta+Y

Перебор ранее вводившихся команд.

Control+N

Пролистывание истории команд вниз (вперед).

Control+P

Пролистывание истории команд вверх (назад).

Meta+N

Т.н. наращиваемый поиск в истории команд вперед.

Meta+P

Нарашиваемый поиск в истории команд назад.

Прочие

Control+R

Перерисовка экрана (например, при смене экранного шрифта командой vidcontrol или аналогичной).

Control+L Meta+Control+L

Очистка экрана (эквивалентно команде clean).

Meta+?

После ввода команды дает следующие сообщения: для встроенной команды оболочки - что таковой является (например - bindkey: shell built-in command), для внешней команды - путь к исполнимому файлу (/usr/bin/man), для псевдонима - его значение (/bin/ls -FG).

Meta+H

Запуск справки для введенной команды (если таковая имеется).

Control+X-Control+D

Вывод списка всех доступных в системе команд.

Встроенные команды

Данный список не полон - в свое время я его не доделал, а нынче tcsh не пользуюсь. Дополнения и комментарии приветствуются. Полный список встроенных команд можно получить с помощью команды builtins (к слову сказать, также встроенной).

@ - вывод всех переменных оболочки

@ name = expr - присвоение выражению (expr) значения name

alias - команда управления псевдонимами; без опций и аргументов - вывод значений всех псевдонимов;

другие варианты:

alias имя_псевдонима значение - присвоение значению псевдонима

alias имя_псевдонима - вывод значения псевдонима, например

alias ll

ls -la

alloc - вывод информации о памяти.

bg %job - перевод фоновой задачи в активный режим (по номеру или имени задачи).

bindkey - вывод всех клавиатурных команд; варианты:

bindkey -l - список команд редактирования с кратким описанием.

bindkey -d - переключение в режим команд по умолчанию.

bindkey -e - переключение в стиль команд emacs

bindkey -v - переключение в стиль команд vi

bindkey -a - вывод альтернативных команд (для режима vi)

break - прекращение цикла foreach или while

builtins - вывод всех встроенных команд

breaksw - causes a break from a switch, resuming after the endsw.

bye - синоним команды logout; доступна только в том случае, если shell собран с этой опцией; опции компиляции можно просмотреть командой echo \$version.

cd путь - переход в каталог; варианты:

cd - - переход в последний рабочий каталог

cd -p путь - выводит путь, используемый как подстановка для заданного

cd -l путь - выводит абсолютный путь для указанного каталога.

chdir - синоним команды cd.

dirs - выводит подстановку для текущего каталога, например:

dirs -l - полный путь для текущего каталога

echo 'Привет' - вывод сообщения (то, что в кавычках).
echotc - перевод курсора на линию или колонку, к началу и т.д.
else end endif endsw
Условные операторы.
exec command - исполнение указанной команды в текущем shell'e.
exit - выход из оболочки, при повторной авторизации после этой команды - возврат в последний рабочий каталог.
fg - перевод процесса (по имени или номеру) в активный режим.
hashstat - вывод статистики о хеш-таблице.
history - вывод истории команд; варианты:
history -h - без номера и времени
history -r - в обратном порядке
history -c - очистка истории команд
hup [command] - перезапуск текущей команды
jobs - вывод списка текущих заданий; варианты:
jobs -l - вывод списка текущих заданий с ID задачи
%job - перевод задачи в активный режим; синоним встроенной команды fg.
%job & - перевод задачи в фоновый режим; синоним встроенной команды bg.
kill - прекращение процесса.
login - выход из текущего shell'a и приглашение к повторной авторизации.
logout - выход из текущего сеанса работы.
ls - просмотр содержимого каталога.
nice - установка приоритета процесса.
rehash - перестройка хеш-таблицы
set - установка переменной оболочки
setenv - установка переменной окружения
time - вывод текущего времени
umask - установка маски доступа
unalias - отмена установленного псевдонима.

Конфигурационные файлы

По умолчанию во FreeBSD для csh (tcsh) предусмотрен (во имя совместимости с классическим csh) только один стартовый файл - /etc/cshrc (/cshrc), аналогом которого для пользователя будет ~/.cshrc. Однако вообще tcsh конфигурируется такими файлами - ~/.tcshrc (читывается первым при авторизации), ~/.history (читается после ~/.tcshrc), ~/.login (читается при каждом вызове оболочки), ~/.cshdirs (читается последним). Порядок считывания конфигурационных файлов может быть изменен при компиляции соответствующими опциями.

Очень коротко о bash

Алексей Федорчук

Bash - наиболее распространенная среди пользователей Linux командная оболочка, выступающая в этой ОС к тому же общесистемной и умолчальной. Популярна она, насколько мне известно, и среди пользователей иных POSIX-систем, по крайней мере свободных. И потому не сказать о ней хоть пару слов здесь - нельзя. Однако о bash существует немерянное количество источников информации. Достаточно заметить, что в любой толстой книге про Linux, когда речь заходит о командных оболочках, имеется ввиду именно bash. Немало сведений о ней есть и в Сети, в том числе в русскоязычном ее сегменте. Не будучи ни знатоком, ни любителем этой оболочки (и даже ее пользователем - вот уже более трех лет), я ничего не могу добавить к написанному другими. И потому на этой странице ограничусь ссылками на [источники информации](#).

Классические Unix-утилиты. Введение для пользователя

Алексей Федорчук

Классические утилиты Unix обеспечивают базовую функциональность любой POSIX-совместимой системы. Собственно говоря, в свободных POSIX-системах они в чистом виде не встречаются из-за лицензионных соображений. Однако тут обнаруживаются их аналоги, разработанные в рамках проекта GNU (их обычно так и называют - GNU-утилиты) или различных BSD-проектов. Причем ни те, ни другие ничуть не уступают

своим проприетарным прототипам в функциональности, а зачастую и превосходят их во многих отношениях.

По секрету скажу, что в большинстве коммерческих Unix'ов уже давно не используются собственно утилиты из собственно Unix (то есть System V) - обычно они заменены на BSD-аналоги, благо лицензия последних это позволяет. И потому это семейство программ мы будем называть далее базовыми утилитами, без различия их происхождения. Хотя в некоторых случаях придется оговаривать особенности BSD- или GNU-утилит.

Как уже было замечено, базовые утилиты служат для обеспечения базовой функциональности системы (в частности, они - неотъемлемая часть всех общесистемных сценариев). С этим, надеюсь, спорить никто не будет. Однако этим значение базовых утилит не исчерпывается: среди них содержится достаточное количество инструментов для решения самых различных задач пользователя, администратора, разработчика. Именно они символизируют собой целостность POSIX-систем, делая их самодостаточными. Не то чтобы в реальной жизни можно обойтись без различных дополнительных приложений, в том числе и тяжелых. Но базовый инструментарий, при должном его знании, позволяет легко и просто (подчас проще, чем специализированные программы) решить множество вполне реальных пользовательских задач.

В обоснование этого тезиса и начат цикл статей о возможностях базовых утилит, входящих в base-комплект любой BSD-системы (во FreeBSD это называется Distributions) или в стандартный набор Base Linux (о котором говорится в соответствующем разделе).

Речь в этом цикле пойдет, в сущности, о тех самых командах, список которых выдается в ответ на нажатие клавиши табуляции в командной строке bash свежеустановленной базовой Linux-системы. И которые можно обнаружить в каталогах /bin, /sbin, /usr/bin и /usr/sbin BSD-системы.

Количество этих команда (а в свежеустановленном Gentoo, например, их более 670) способно обескуражить. Если, конечно, хоть как-то не упорядочить (в уме) это богатство. И упорядочивание это, в почти соответствие с заветами председателя Мао, можно провести в двух стилях (великий кормчий, правда, говорил о трех стилях, но меня, не столь великого, на это не хватило).

Первый стиль упорядочивания - по назначению. Здесь можно выделить команды пользовательские, административные и разработческие. И не нужно думать, что первые нужны только пользователю, вторые - только сисадмину, а третьи - лишь программеру.

Под пользовательскими можно понимать команды управления данными (файлами и их содержимым). А ведь и самые крутые админы или программеры время от времени копируют свои файлы или просматривают их содержимое. Административные команды предназначены для управления системой. Но на настольной машине каждый пользователь обычно - сам себе root, и проблему подключения нового диска (и даже монтирования дискеты) за него никто не решит. Ну а разработческие команды нужны не только для создания нового гениального софта, но и для сборки уже созданного. И как это делать - неплохо понимать любому юзеру.

Именно такой способ реализуется во Free- и прочих BSD. Где все компоненты базового набора размещаются в корневом каталоге и основных ветвях каталога /usr (а весь дополнительно устанавливаемый софт - исключительно в каталоге /usr/local).

Второй стиль упорядочивания, ортогональный первому, реализуется в Linux (и особенно последовательно проводится в source based его дистрибутивах). Это - упорядочивание по принадлежности к именованным наборам утилит, которые часто именуются пакетам (packages); в частности, именно как packages они фигурируют в [Linux from Scratch](#) Герарда Бикманса.

И еще одна оговорка. Стили упорядочивания по назначению и по принадлежности - ортогональны, но не в Евклидовом пространстве. И потому одни команды из некоего пакета могут попасть в юзерскую группу, некоторые - в админскую. А иногда одна и та же команда может выступать в разных ипостасях.

Таким образом, в этом цикле необходимо рассмотреть, как соотносятся именованные наборы базовых утилит с командами, классифицированными по их назначению. Как и положено пользователю, начну с той рубашки, что ближе моему телу - с пользовательских возможностей.

Конечно, для пользователя (как и администратора, и разработчика, и кого угодно еще) базовые утилиты начинаются с командной оболочки, сиречь shella. Однако дело это столь сложно, что выделено в [отдельное производство](#). Как и дело о базовых текстовых редакторах - этот разговор пойдет в [соответствующем разделе](#), а куда он способен завести - я и сам не знаю. И потому пока ограничусь только описанием некоторых отдельных пользовательских из состава базовых утилит.

Задач перед пользователем стоит великое множество. Однако две из них встают с неизбежностью Рагнарека: манипулирование файлами и манипулирование их контентом.

Под манипулированием файлами понимается, ясное дело, их:

- создание,
- копирование, перемещение и переименование,

- удаление,
- разделение,
- архивирование и компрессия.

Однако любая файловая операция начинается с просмотра текущего состояния файловой системы и (или) поиска необходимого файла. Несколько позже начнем с этого и мы - ближайшая из статей цикла будет посвящена именно командам управления файлами.

А пока вспомним, что файлы создаются (и хранятся) не ради самих себя, а ради того контента, который в них содержится. И потому вторая из вековечных пользовательских задач - просмотр содержимого файлов, его модификация, а главное - поиск файла по его (весьма приближенно запомненному) содержимому. Так что следующий (после предыдущего) материал - об управлении контентом файлов (для определенности - текстовых, ибо их в POSIX-системах подавляющее большинство).

Наконец, есть у пользователя и третья группа задач - ее можно объединить понятием *всякая всячина*. Именно всяким полезным (как мне кажется) утилитам самого разного назначения и будет посвящена заключительная из юзерских статей о базовых утилитах.

Файловые утилиты: создание и атрибуция

Алексей Федорчук

В этой заметке и нескольких последующих будут рассмотрены команды, предназначенные для операций с файлами как целостными сущностями, не затрагивающие их содержимого. Работа с файлами сводится к:

- их созданию;
- навигации по файловой системе;
- получению информации о файлах;
- манипулированию существующими файлами;
- поиску файлов.

Кроме того, к сфере файловых операций следует отнести и средства их архивирования, компрессии, да и вообще резервного копирования, - но это тема обширная, и потому отдельная.

Характеристика средств

Средства управления файлами в Base Linux входят в состав пакета coreutils (в него с некоторого времени объединены три ранее самостоятельных комплекта - fileutils, sh-utils и textutils). Он включает в себя множество команд для создания файлов различных типов, установки атрибутов файлов, копирования, перемещения, переименования и удаления файлов, а также получения информации о файлах. Последней цели служит также набор file, включающий единственную, одноименную, команду.

Кроме того, к этому же кругу утилит относится пакет findutils, в который входят команды: bigrm, code, find, frcode, locate, updatedb and xargs. Большая их часть предназначена для определения местоположения файлов, а команда find (особенно в сочетании с xargs) - практически универсальное средство обработки файлов.

Практически все перечисленные команды имеются и в BSD-системах, выступая там под теми же именами и обладая теми же функциями. Там они, разумеется, не делятся на пакеты, попадая целиком под понятие Base Distributions.

Ниже я рассмотрю основные команды из этих наборов, предназначенные для файловых операций, вместе с их наиболее используемыми опциями. Чтобы не возвращаться далее к этому вопросу, скажу сразу: почти все

команды этой заметки (и большинство из заметок последующих) имеет три стандартные опции (т.н. GNU Standard Options, впрочем, они же имеются и в BSD-утилитах, и, более того, именно оттуда и происходят:-)): --help (иногда также -h или -?) для получения помощи, --verbose (иногда также -v) для вывода информации о текущей версии, и --, символизирующей окончание перечня опций (т.е. любой символ или их последовательность после нее интерпретируются как аргумент). Так что далее эти опции в описания фигурировать не будут.

И еще. В этих заметках будет говориться только о самостоятельных командах и утилитах - тех, которым соответствует собственный исполняемый файл (обычно в каталогах /bin, /sbin, /usr/bin и /usr/sbin). Для встроенных команд оболочки (а они подчас совпадают с внешними утилитами по назначению и имени) в свое время найдется и свое место.

Создание

Работа с файлами начинается с их создания. Конечно, в большинстве случаев файлы (вместе с их контентом) создаются соответствующими приложениями (текстовыми редакторами, word-процессорами и т.д.). Однако в набор coreutils входит несколько команд, специально предназначенных для создания файлов. Это - touch, mkdir, ln, mknod, mkfifo. Кроме того, с этой же целью могут быть использованы команды cat и tee. И еще раз напомню, что все это хозяйство имеется и в любой BSD-системе.

Команды touch, cat, tee

Первая из указанных команд в форме

```
$ touch filename
```

просто создает обычный (регулярный) файл с именем filename и без всякого содержимого. Кроме того, с помощью специальных опций она позволяет устанавливать временные атрибуты файла, о чем я скажу чуть ниже.

Для чего может потребоваться создание пустого файла? Например, для создания скелета web-сайта с целью проверки целостности ссылок. Поскольку число аргументов команды touch не ограничено ничем (вернее, ограничено только максимальным количеством символов в командной строке:-)), это можно сделать одной командой:

```
$ touch index.html about.html content.html [...]
```

Можно, воспользовавшись приемом группировки аргументов (об этом говорится в [care о POSIX](#)), и заполнить файлами все подкаталоги текущего каталога:

```
$ touch dirname1/{filename1,filename2} \
      dirname2/{filename3,filename4}
```

и так далее. Правда, сама команда touch создавать подкаталоги не способна - это следует сделать предварительно командой mkdir (о которой - абзацем ниже).

Команда cat также может быть использована для создания пустого регулярного файла. Для этого нужно просто перенаправить ее вывод в файл:

```
$ cat > filename
```

создать новую строку (нажатие клавиши **Enter**) и ввести символ конца файла (комбинацией клавиш **Control+Z**). Разумеется, предварительно в этот файл можно и ввести какой-нибудь текст, однако это уже относится к управлению контентом. Почему мы и рассмотрим команду cat подробнее в соответствующем разделе.

Интересно создание файлов с помощью команды tee. Смысл ее - в раздвоении выходного потока, выводимого одновременно и на стандартный вывод, и в файл, указанный в качестве ее аргумента. То есть если использовать ее для создания файла с клавиатуры, это выглядит, будто строки удваиваются на экране. Но это не так: просто весь вводимый текст копируется одновременно и на экран, и в файл. И потому ее удобно применять в командных конструкциях, когда требуется одновременно и просмотреть результаты исполнения какой-либо команды, и запечатлеть их в файле:

```
$ls dir | tee filename
```

По умолчанию команда tee создает новый файл с указанным именем, или перезаписывает одноименный, если он существовал ранее. Однако данная с опцией -a, она добавляет новые данные в конец существующего файла.

Команда mkdir

Команда mkdir создает файл особого типа - каталог, содержимым которого является список входящих в него файлов. Очевидно, что список этот в момент создания каталога должен быть пуст, однако это не совсем так: любой, даже пустой, каталог содержит две ссылки - на каталог текущий, обозначаемый как ./ (т.е. сам на себя) и на каталог родительский, ../ (т.е тот, в список файлов которого он включается в момент создания). Не исключение тут даже корневой каталог - просто для него индексные дескрипторы (уникальные числа, однозначно идентифицирующие любой файл) текущего и родительского каталогов совпадают.

Команда `mkdir` требует обязательного аргумента - имени создаваемого каталога. Аргументов может быть больше одного - в этом случае будет создано два или больше поименованных каталогов. По умолчанию они создаются как подкаталоги каталога текущего. Можно создать также подкаталог в существующем подкаталоге:

```
$ mkdir parentdir/newdir
```

Если же требуется создать подкаталог в каталоге, отличном от текущего, - путь к нему требуется указать в явном виде, в относительной форме:

```
$ mkdir ..dirname1 dirname2
```

или в форме абсолютной:

```
$ mkdir /home/username dirname1 dirname2
```

И в удаленном каталоге можно одной командой создать несколько подкаталогов, для чего нужно прибегнуть к группировке аргументов:

```
$ mkdir ..parentdir/{dirname1,dirname2,...,dirname#}
```

Такой прием позволяет одной командой создать дерево каталогов проекта. Например, скелет web-сайта, который потом можно наполнить пустыми файлами с помощью команды `touch`.

А опций у команды `mkdir` - всего две (за исключением стандартных опций GNU): `--mode` (или `-m`) для установки атрибутов доступа и `--parents` (или `-p`) для создания как требуемого каталога, так и родительского по отношению к нему (если таковой ранее не существовал). Первая опция используется в форме

```
$ mkdir --mode=### dirname
```

или

```
$ mkdir -m ### dirname
```

Здесь под `###` понимаются атрибуты доступа для владельца файла, группы и прочих, заданные в численной нотации (например, 777 - полный доступ на чтение, изменение и исполнение для всех). Не возбраняется и использование символьной нотации: команда

```
$ mkdir -m a+rwx dirname
```

создаст каталог с теми же атрибутами полного доступа для всех (об этом подробно говорится в [саге о POSIX](#)).

Опция `--parents` (она же `-p`) позволяет создавать иерархическую цепочку подкаталогов любого уровня вложенности. Например,

```
$ mkdir -p dirlevel1/dirlevel2/dirlevel3
```

в один заход создаст в текущем каталоге цепочку вложенных друг друга подкаталогов. Разумеется, и здесь с помощью группировки аргументов можно создать несколько одноранговых подкаталогов:

```
$ mkdir -p dirlevel1/dirlevel2/{dirlevel31,...,dirlevel3#}
```

Команда `ln`

Командой `ln` создаются ссылки обоих видов - жесткие и символические. Первые - просто иное имя (то есть иная запись в каком-либо каталоге) для того же набора данных. И создается просто -

```
$ ln filename linkname
```

где первый аргумент (`filename`) - имя существующего файла, а второй (`linkname`) - имя создаваемой ссылки (то есть просто второе имя для того же набора данных). Жесткая ссылка может быть создана только на обычный (регулярный) или специальный файл (например, файл устройства), но не на каталог. Кроме того, она не может пересекать границы физической файловой системой.

Символическая ссылка создается той же командой `ln`, и с теми же аргументами, но со специальной опцией `-s`:

```
$ ln -s filename linkname
```

Символическая ссылка, в отличие от жесткой, - файл особого типа, и таких ограничений не имеет: она может указывать на любой файл или каталог, в том числе расположенный в другой файловой системе. При создании символической ссылки на каталог автоматически создаются символические ссылки на все входящие в его состав файлы и вложенные подкаталоги.

В некоторых системах, говорят, допустимо и создание жестких ссылок на каталог (с помощью опций `--directory`, `-d` или `-F`), однако - исключительно от лица суперпользователя.

В качестве источника символической ссылки может выступать другая символическая ссылка. То есть команда

```
$ ln -s linkname linkname2
```

создаст символическую ссылку `linkname2`, ссылающуюся на ссылку же `linkname`, и только последняя будет указывать на обычный файл или каталог `filename`. Однако если дать команду

```
$ ln -n linkname linkname2
```

то новообразованная ссылка `linkname2` будет указывать не на ссылку `linkname`, а на исходный для последней файл `link_scr`.

Если имя символической ссылки, заданной в качестве второго аргумента команды `ls -s`, совпадает с именем существующего файла (регулярного, каталога, или символической ссылки на файл, отличный от первого аргумента команды), то новая символическая ссылка создана не будет. Однако такую ссылку, совпадающую с существующим именем, можно принудительно создать посредством опции `-f` (или `--force`).

При этом, разумеется, содержание оригинального файла (будь то другая символическая ссылка, регулярный файл или каталог), одноименного создаваемой символической ссылке, будет утеряно безвозвратно, причем в случае каталога - вместе со всеми его файлами и подкаталогами. Поэтому при форсированном создании

символических ссылок на каталоги, содержащие большое количество разноименных файлов, существует вероятность случайной (при простом совпадении имен) утери важных данных. Чтобы предотвратить это, используется опция `--interactive (-i)`: благодаря ей команда `ln` будет выдавать запрос на подтверждение действий, если обнаружит совпадения имен создаваемых ссылок и существующих файлов.

Есть и другие способы сохранить исходный файл при совпадении его имени с именем создаваемой ссылки. Так, опция `-b (--backup)`, прежде чем переписать замещаемый файл, создаст его резервную копию - файл вида `filename~`. А опция `-S (--suffix)` не только создаст такую копию, но и припишет к ее имени какой-либо осмысленный суффикс. Так, в результате команды

```
$ ln -sf --S=.old filename1 filename2
```

существовавший ранее регулярный файл `filename2` будет замещен символьской ссылкой на файл `filename1`, однако содержимое оригинала будет сохранено в файле `filename2.old`. Опция же `-V METHOD (--version-control=METHOD)` позволяет последовательно нумеровать такие копии замещаемых симлинками файлов.

Значения `METHOD` могут быть:

- `t, numbered` - всегда создавать нумерованную копию;
- `nil, existing` - создавать нумерованную копию только в том случае, если хоть одна таковая уже существует, если же нет - создавать обычную копию;
- `never, simple` - всегда создавать обычную копию;
- `none, off` - не создавать копию замещаемого файла вообще.

Команда `mknod`

Команда `mknod` предназначается для создания файлов специального типа - файлов устройств (о них речь пойдет в грядущих статьях Intro-цикла). В качестве аргументов она требует:

- имени устройства в форме `/dev/dev_name`;
- указания его типа - символьного (`c`), с побитным доступом (например, последовательные порты) или блочного (`b`), с которым возможен обмен данными блоками (например, дисковые накопители);
- старшего номера (`major`) - уникального идентификатора, характеризующего родовую группу устройств (например, 4 - идентификатор виртуальных терминалов);
- младшего номера (`minor`), являющегося идентификатором конкретного устройства в своей группе (например, младший номер 0 в группе старшего номера 4 - идентификатор первой, системной, виртуальной консоли, а 63 - идентификатор последней теоретически возможной из них).

Кроме стандартных, команда `mknod` имеет только одну опцию `-m (--mode)`, с помощью которой устанавливаются атрибуты доступа к создаваемому файлу устройства (точно также, как это было описано для команды `mkdir`). Таким образом, команда

```
$ mknod --mode=200 /dev/tty63 c 4 63
```

создаст файл устройства для последней теоретически возможной виртуальной консоли. Что, впрочем, не значит, будто бы сразу после этого в нее можно переключаться - предварительно следует внести должные изменения в скрипты инициализации, о чем будет рассказано в соответствующем месте.

На практике команда `mknod` часто используется в опосредованном виде - в составе сценария `/dev/MAKEDEV`, автоматизирующего процесс создания файлов устройств. В частности, он делает ненужным знание старшего и младшего номеров устройств. Правда, только для тех из них, которые были предусмотрены при разработке сценария. Если же требуемое устройство не входит в число охваченных сценарием `/dev/MAKEDEV`, команды `mknod` в явном виде не избежать. Правда, все более широкое внедрение файловой системы `devfs`, создающей

файлы устройств при загрузке (причем - только реально существующих в системе), скоро сделает команду mknod анахронизмом.

Команда mkfifo

Если создавать файлы устройств (и, соответственно, пользоваться командой mknod) приходится достаточно редко, то необходимости применения команды mkfifo у юзера может не возникнуть никогда (у меня, например, такой необходимости не возникало ни разу за все время знакомства с Linux'ом). Тем не менее, отметим для полноты картины, что такая команда существует и создает именованные каналы (named pipes) - специальные файлы, предназначенные для обмена данными между процессами. Вообще-то, именованные каналы играют большую роль во всех Unix-системах, но роль эта от пользователя хорошо замаскирована. Так что полноты картины для просто приведу формат команды без комментариев:

```
$ mkfifo [options] filename
```

А опция здесь - всего одна -m (--mode), и, как нетрудно догадаться по аналогии, устанавливает она атрибуты доступа.

Атрибуция

К слову сказать - об атрибутах, следующая группа команд предназначена именно для атрибутирования файлов. В ней - chmod, chown, chgrp, а также уже затронутая команда touch.

Команды chown и chgrp

Команды chown и chgrp служат для изменения атрибутов принадлежности файла - хозяину и группе: очевидно, что все, не являющиеся хозяином файла, и не входящие в группу, к которой файл приписан, автоматически попадают в категорию прочих (other).

Формат команды chown - следующий:

```
$ chown newowner filename
```

По причинам безопасности, достаточно очевидным, изменить хозяина файла может только суперпользователь. Пользователь обычный в подавляющем большинстве случаев автоматически становится хозяином всех им созданных (и скопированных) файлов, и избавиться от этого бремени, как и от родительского долга, не в состоянии.

А вот изменить принадлежность своих файлов (т.е. тех, в атрибутах принадлежности он прописан как хозяин) пользователь вполне может. Команда:

```
$ chgrp newgroup filename
```

от его лица припишет файл filename к группе newgroup. Однако и здесь есть ограничение - результат будет достигнут, только если хозяин файла является членом группы newgroup, иначе опять придется прибегнуть к полномочиям администратора.

Можно также одной командой сменить (только суперпользователю, конечно) и хозяина файла, и группу, к которой он приписан. Делается это так:

```
$ chown newowner:newgroup filename
```

Или так:

```
$ chown newowner:filename
```

Где, понятное дело, под именем newowner выступает новый хозяин файла, а под именем newgroup - новая группа, к которой он приписан.

В обеих командах вместо имени хозяина и группы могут фигурировать их численные идентификаторы (UID и GID, соответственно).

Для команд chown и chgrp поддерживается один и тот же набор опций. Наиболее интересны (и важны) две из них. Опция --reference позволяет определить хозяина файла и его принадлежность к группе не явным образом, а по образу и подобию файла, имя которого выступает в качестве значения опции. Так, команда

```
$ chown --reference=ref_filename filename
```

установит для файла filename те же атрибуты принадлежности (хозяина и группу), что были ранее у файла ref_filename.

Опция -R (или --recursive) распространяет действие обеих команд не только на файлы текущего каталога (излишне напоминать, что в качестве аргументов команд могут использоваться маски типа *, *.ext, name.* и т.д.), но и на все вложенные подкаталоги, вместе с входящими в них файлами. То есть пользователь может поменять групповую принадлежность всех файлов в своем домашнем каталоге одной командой:

```
$ chgrp -R newgroup ~/
```

А суперпользователь тем же способом может установить единообразные атрибуты принадлежности "по образцу" для всех компонентов любого каталога:

```
$ chown -R --reference=ref_filename \
  /somepath/somecat/*
```

Команда chmod

Как и следует из ее имени, команда chmod предназначена для смены атрибутов доступа - чтения, изменения и исполнения. В отношении единичного файла делается это просто:

```
$ chmod [атрибуты] filename
```

Атрибуты доступа могут устанавливаться с использование как символьной, так и цифровой нотации (подробнее об этом - в [POSIX-cale](#)). Первый способ - указание, для каких атрибутов принадлежности (хозяина, группы и всех остальных) какие атрибуты доступа задействованы. Атрибуты принадлежности обозначаются символами u (от *user*) для хозяина файла, g (от *group*) - для группы, o (от *other*) для прочих и a (от *all*) - для всех категорий принадлежности вообще. Атрибуты доступа символизируются литерами r (от *read*), дающей право чтения, w (от *write*) - право изменения и x (от *execute*) - право исполнения.

Атрибуты принадлежности соединяются с атрибутами доступа символами + (присвоение атрибута доступа), - (отнятие атрибута) или = (присвоение только данного атрибута доступа с одновременным отнятием всех остальных). Одновременно в строке можно указать (подряд, без пробелов) более чем один из атрибутов принадлежности и несколько (или все) атрибуты доступа.

Для пояснения сказанного приведу несколько примеров. Так, команда

```
$ chmod u+w filename
```

установит для хозяина (u) право изменения (+w) файла filename, а команда

```
$ chmod a-x filename
```

отнимет у всех пользователей вообще (a) право его исполнения (-x). В случае, если некоторый атрибут доступа присваивается всем категориям принадлежности, символ a можно опустить. Так, команда

```
$ chmod +x filename
```

в противоположность предыдущей, присвоит атрибут исполнения файла filename всем категориям принадлежности (и хозяину, и группе, и прочим).

С помощью команды

```
$ chmod go=rx filename
```

можно присвоить группе принадлежности файла filename и всем прочим (не хозяину и не группе) право на его чтение и исполнение с одновременным отнятием права изменения.

Наконец, команда chmod в состоянии установить и дополнительные атрибуты доступа к файлу, такие, как биты SUID и GUID, или, скажем, атрибут sticky (и о них будет разговор в [POSIX-cale](#)). Так, в некоторых системах (например, во FreeBSD - в Linux-дистрибутивах я с таким не встречался) XFree86 подчас по умолчанию устанавливается без атрибута суидности на исполняемом файле X-сервера. Это влечет за собой невозможность (без специальных wrapper'ов) запуска Иксов от лица обычного пользователя. Один из способов борьбы - просто присвоить этому файлу бит суидности:

```
$ chmod u+s /usr/X11R6/bin/XFree86
```

Приведенные примеры можно многократно умножить, но, думается, их достаточно для понимания принципов работы команды chmod с символьной нотацией атрибутов.

Цифровая нотация - еще проще. При ней достаточно указать сумму присваиваемых атрибутов в восьмеричном исчислении (4 - атрибут чтения, 2 - атрибут изменения и 1 - атрибут исполнения; 0 символизирует отсутствие любых атрибутов доступа) для хозяина (первая позиция), группы (вторая позиция) и прочих (третья позиция). Все атрибуты доступа, оставшиеся вне этой суммы, автоматически отнимаются у данного файла. То есть команда

```
$ chmod 000 filename
```

означает снятие с файла filename всех атрибутов доступа для всех категорий принадлежности (в том числе и хозяина) и эквивалентна команде

```
$ chmod =rwx filename
```

в символьной нотации. А команда

```
$ chmod 777 filename
```

напротив, устанавливает для всех полный доступ к файлу filename. Для установки дополнительных атрибутов доступа в численной нотации потребуется указать значение четвертого, старшего, регистра. Так, команда для рассмотренного выше примера - присвоения атрибута суидности исполняемому файлу X-сервера, - в численной нотации будет выглядеть как

```
$ chmod 4711 /usr/X11R6/bin/XFree86
```

Как и для команд chown и chgrp, наиболее значимые опции команды chmod - это --reference и -R. И смысл их - идентичен. Первая устанавливает для файла (файлов) атрибуты доступа, идентичные таковым референсного файла, вторая - распространяет действие команды на все вложенные подкаталоги и входящие в них файлы. Рекурсивное присвоение атрибутов доступа по образцу требует внимания. Так, если рекурсивно отнять для всего содержимого домашнего каталога атрибут исполнения (а он без соблюдения некоторых условий монтирования автоматом присваивается любым файлам, скопированным с носителей файловой структуры FAT или ISO9660 без расширения RockRidge, что подчас мешает), то тем самым станет невозможным вход в любой из вложенных подкаталогов. Впрочем, в [заметке про утилиту find](#) будет показан один из способов борьбы с таким безобразием.

Команда touch для атрибуции

Кроме атрибутов принадлежности и доступа, файлам свойственны еще и атрибуты времени - времени доступа (atime), времени изменения метаданных (ctime) и времени изменения данных (mtime) файла. Они устанавливаются автоматически, в силу самого факта открытия файла (atime), смены атрибутов любых атрибутов, например, доступа (ctime) или редактирования содержимого файла (mtime).

Однако бывают ситуации, когда автоматически установленные временные атрибуты требуется изменить. Случай продления жизни trial-версии программы не рассматриваем - настоящий POSIX'ивист до такого не опускается, не так ли? А вот сбой системных часов, в результате которого временные атрибуты создаваемых и модифицируемых файлов перестанут соответствовать действительности - штука вполне вероятная.

Казалось бы, чего страшного? Ах нет, фактор времени играет в Unix-системах очень существенную роль.

Во-первых, команда make (а под ее управлением компилируются программы из исходников) проверяет временные атрибуты файлов (в первую очередь - атрибут mtime) и при их несоответствии может работать с ошибками. Ну и более обычная ситуация - на основе временных меток файлов можно эффективно осуществлять, скажем, резервное копирование (см. [раздел о той же утилите find](#)). И потому желательно, чтобы они отражали реальное время создания и модификации файла.

Так вот, для изменения временных атрибутов файлов и предназначена в первую очередь команда touch, которую ранее мы использовали просто для создания пустого файла. Данная же с именем существующего файла в качестве аргумента -

```
$ touch exist_file
```

она присвоит всем его временным атрибутам (atime, ctime, mtime) значения текущего момента времени.

Изменение временных атрибутов можно варыировать с помощью опций. Так, если указать только одну из опций -a, -c, -m, то текущее значение времени будет присвоено только атрибуту atime, ctime или mtime, соответственно. Если при этом использовать еще и опцию -d [значение], то любому из указанных атрибутов (или им всем) можно присвоить любую временную метку, в том числе и из далекого будущего. А посредством опции -r filename файл-аргумент получит временные атрибуты, идентичные таковым референсного файла filename.

Файловые утилиты: навигация, информация, манипулирование

Алексей Федорчук

Навигация по файловой системе

Следующее, что необходимо пользователю после создания файлов - ориентация среди существующего их изобилия. Для начала при этом неплохо определиться со своим текущим положением в файловой системе. Для этого предназначена команда pwd. В ответ на нее выводится полный путь к текущему каталогу.

Например, если текущим является домашний каталог пользователя, в ответ на:

```
$ pwd
```

последует

```
/home/username
```

Команда pwd имеет всего две опции: -L и -P. Первая выводит т.н. логический путь к текущему каталогу. То есть, таковым является, скажем, каталог /usr/src/linux, являющий собой символьическую ссылку на каталог /usr/src/linux-номер_версии, то в ответ на

```
$ pwd -L
```

так и будет выведено

```
/usr/src/linux
```

Впрочем, тот же ответ последует и на команду pwd без опций вообще. Если же дать эту команду в форме

```
$ pwd -P
```

то будет выведен путь к физическому каталогу, на который ссылается текущий, например:

```
/usr/src/linux-2.4.19-gentoo-r9
```

Далее, по каталогам неплохо как-то перемещаться. Что делается командой cd. В отличие от прочих команд, рассматриваемых в этом разделе, это - внутренняя команда, встроенная во все командные оболочки - бесполезно было бы искать соответствующий ей исполняемый файл. Однако это не уменьшает ее важности. Использование ее очень просто -

```
$ cd pathname
```

где pathname - путь к искомому каталогу в абсолютной (относительно корня) или относительной (относительно текущего каталога) форме.

Определить местоположение команды (и вообще исполняемых файлов) в структуре файловой системы можно с помощью команды which (это также встроенная команда оболочки). В качестве аргумента ее можно указать одно или несколько имен файлов, в ответ на что будет выведен полный путь к каждому из них:

```
$ which tcsh zsh bash
```

```
/bin/tcsh
```

```
/bin/zsh  
/bin/bash
```

При наличии одноименных исполняемых файлов в разных каталогах по умолчанию будет выведен путь только к первому из них: для вывода всех файлов-“тезок” можно прибегнуть к опции -a. При этом не важно, будут это жесткие или символьические ссылки.

Более широкие возможности поиска - у команды whereis. По умолчанию, без опций, она для заданного в качестве аргумента имени выводит список бинарных файлов, man-страниц и каталогов с исходными текстами:

```
$ whereis zsh  
zsh: /bin/zsh /etc/zsh /usr/lib/zsh /usr/share/zsh  
/usr/man/man1/zsh.1.gz /usr/share/man/man1/zsh.1.gz
```

Соответствующими опциями можно задать поиск файлов одного из этих типов: -b - бинарных, -m - страниц руководств, -s - каталогов с исходниками. Дополнительные опции -B, -M, -S (в сочетании с опцией -f) позволяют определить исходные каталоги для их поиска.

Наконец, команда locate осуществляет поиск всех файлов и каталогов, содержащих компонент имени, указанный в качестве аргумента и осуществляет вывод содержимого найденных каталогов. Так, в ответ на команду

```
$ locate zsh
```

будет выведен список вроде следующего:

```
/bin/zsh  
/bin/zsh-4.0.6  
/etc/zsh  
/etc/zsh/zlogin  
/etc/zsh/zshenv  
/etc/zsh/zshrc
```

и так далее. Команда locate при этом обращается к базе данных, расположенной в каталоге /var/spool/locate/locatedb (точный путь в разных системах может варьироваться). По умолчанию эта база данных пуста - и перед использованием команды locate должна быть наполнена содержанием. Для этого предназначен скриптовый /usr/bin//updatedb, извлекающий сведения из базы данных установленных пакетов - /var/db/pkg. При активной доустановке программ база эта нуждается в периодическом обновлении.

Приведенные команды относятся к поиску исполняемых файлов и программных компонентов. Однако чаще перед пользователем возникает необходимость поиска неких произвольных файлов. На сей предмет существует команда find. Однако возможности ее не сводятся к поиску - это практически универсальный инструмент для файловых операций. И потому она будет подробно рассмотрена отдельно - в [следующей заметке](#).

Получение информации о файлах

Наиболее универсальным средством получения практически исчерпывающей информации о файлах является команда ls. Общая форма ее запуска -

```
$ ls [options] names
```

где в качестве аргумента names могут выступать имена файлов или каталогов в любом количестве. Команда эта имеет многочисленные опции, основные из которых мы и рассмотрим.

Начать с того, что команда ls, данная без всяких опций, по умолчанию выводит только имена файлов, причем опуская т.н. dot-файлы, имена которых начинаются с точки (это - некие аналоги hidden-файлов в MS DOS). Кроме того, если в качестве аргумента указано имя каталога (или аргумент не указан вообще, что подразумевает текущий каталог), из списка имен его файлов не выводятся текущий (.) и родительский (..) каталог.

Для вывода всех без исключения имен файлов (в том числе и скрытых) предназначена опция -a. Смысл опции -A близок - она выводит список имен всех файлов, за исключением имени текущего (.) и родительского (..) каталога.

Кроме имени, любой файл идентифицируется своим номером inode. Для его вывода используется опция -i:

```
$ ls -i  
12144 content.html    12149 gentoo02.html
```

и так далее. Как и многие другие, команда ls обладает способностью рекурсивной обработки аргументов, для чего предназначена опция -R, выводящая список имен файлов не только текущего каталога, но и всех вложенных подкаталогов:

```
$ ls -R  
unixforall:  
about/ apps/ diffimages/ distro/ signature.html sys/  
anons/ content/ difftext/ gentoo/ statistics/ u4articles/
```

```
unixforall/about:  
about_lol.html about_lol.txt index.html
```

```
unixforall/anons:  
anons_dc.html
```

В выводе команды ls по умолчанию имена файлов разных типов даются абсолютно одинаково. Для их визуального различия используется опция -F, завершающая имена каталогов символом слэша, исполняемых файлов - символом звездочки, символьических ссылок - "собакой"; имена регулярных файлов, не имеющих атрибута исполнения, никакого символа не включают:

```
$ ls -F  
dir1/ dir2/ dir3@ file1 file2* file3@
```

Другое средство для визуального различия типов файлов - колоризация, для чего применяется опция -G. Цвета шрифта, воспроизводящего имена, по умолчанию - синий для каталогов, лиловый (magenta) для символьических ссылок, красный - исполняемых файлов, и так далее. Для файлов устройств, исполняемых файлов с атрибутом "сущности", каталогов, имеющих атрибут sticky, дополнительно колоризуется и фон, на котором выводится шрифта, воспроизводящий их имена. Подробности можно посмотреть секции ENVIRONMENT man-страницы для команды ls. Впрочем, колоризация работает не при всех настройках терминалов (и не во всех командных оболочках).

По умолчанию команда ls выводит список файлов в порядке ASCII-кода первого символа имени. Однако есть возможность его сортировки в обратном порядке (-r), в порядке времени модификации (-i) или времени доступа (-tu). Кроме того, опция -f отменяет какую-либо сортировку списка вообще.

Информацию об объеме файлов можно получить, используя опцию -s, выводящую для имени каждого файла его размер в блоках, а также суммарные объем всех выведенных файлов:

```
$ ls -s ..book  
total 822  
656 book.html  
4 content1.html  
86 var_part2.html  
24 command.html  
38 part2.html  
6 command.txt  
8 shell_tmp.html
```

Добавление к опции -s еще и опции -k (то есть ls -sk) выведет всю ту же информацию в килобайтах. Очевидно, что если размер блока файловой системы, как это подчас бывает, составляет 1024 байта, вывод ls с этими опциями будет одинаков.

Как можно видеть из всех приведенных выше примеров, списки файлов по команде ls выводятся в многоколоночном виде (чему соответствует опция -C, однако указывать ее нет необходимости - многоколоночный вид принят для краткого формата по умолчанию). Но можно задать и одноколоночное представление списка посредством опции -1:

```
$ ls -1  
dir1  
dir2  
dir3  
file1  
file2  
file3
```

До сих пор речь шла о кратком формате команды ls. Однако более информативным является т.н. длинный ее формат, вывод в котором достигается опцией -l и автоматически влечет за собой одноколоночное представление списка:

```
$ ls -l  
total 8  
drwxr-xr-x 2 alv alv 512 8 май 18:04 dir1  
drwxr-xr-x 3 alv alv 512 8 май 17:43 dir2  
lrwxr-xr-x 1 alv alv 4 9 май 07:59 dir3 -> dir2  
-rw-r--r-- 1 alv alv 14 8 май 10:39 file1  
-rwxr-xr-x 1 alv alv 30 9 май 08:02 file2  
lrwxr-xr-x 1 alv alv 2 8 май 10:57 file3 -> f1
```

Можно видеть, что по умолчанию в длинном формате выводятся:

- сведения о типе файла (- - регулярный файл, d - каталог, l - символьическая ссылка, c - файл символьного устройства, b - файл блочного устройства) и атрибуты доступа для различных атрибутов принадлежности (о чем было сказано достаточно);
- количество жестких ссылок на данный идентификатор inode;
- имя пользователя - владельца файла, и группы пользователей, которой файл принадлежит;
- размер файла в блоках;

- время модификации файла с точностью до месяца, дня, часа и минуты (в формате, принятом в данной locale);
- имя файла и (для символьических ссылок) имя файла-источника.

Однако это еще не все. Добавив к команде ls -l еще и опцию -i, можно дополнительно получить идентификатор inode каждого файла, опция -n заменит имя владельца и группу на их численные идентификаторы (UID и GUID, соответственно), а опция -T выведет в поле времени модификации еще и годы, и секунды:

```
$ ls -linT
total 8
694402 drwxr-xr-x 2 1000 1000 512 8 май 18:04:56 2002 dir1
694404 drwxr-xr-x 3 1000 1000 512 8 май 17:43:31 2002 dir2
673058 lrwxr-xr-x 1 1000 1000 4 9 май 07:59:08 2002 dir3 -> dir2
673099 -rw-r--r-- 1 1000 1000 14 8 май 10:39:38 2002 file1
673059 -rwxr-xr-x 1 1000 1000 30 9 май 08:02:23 2002 file2
673057 lrwxr-xr-x 1 1000 1000 2 8 май 10:57:07 2002 file3 -> f1
```

Разумеется, никто не запрещает использовать в длинном формате и опции визуализации (-F и -G), и опции сортировки (-r, t, tu), и любые другие, за исключением опции -C - указание ее ведет к принудительному выводу списка в многоколоночной форме, что естественным образом подавляет длинный формат представления.

Я столь подробно остановился на описании команды ls потому, что это - основное средство визуализации файловых систем любого Unix, при умелом использовании ничуть не уступающее развитым файловым менеджерам (типа Midnight Commander или Konqueror) по своей выразительности и информативности. И отнюдь не требующее для достижения таковых вбивания руками многочисленных опций: в разделе о [командных оболочках](#) будет показано, что соответствующей настройкой последних можно добиться любого "умолчального" вывода команды ls.

Существуют и другие команды для получения информации о файлах. Например, команда под характерным именем file (единственная в одноименном наборе) с аргументом в виде имени файла в состоянии определить тип его с большой детальностью. Так, для регулярных файлов она распознает:

- исполняемые бинарные файлы с указанием их формата (например, ELF), архитектуры процессора, для которых они скомпилированы, характер связи с разделяемыми библиотеками (статический или динамический)
- исполняемые сценарии с указанием оболочки, для которой они созданы;
- текстовые и html-документы, часто с указанием используемого набора символов.

Последнему, впрочем, для русскоязычных документов доверять особо не следует: кодировка KOI8-R в них вполне может быть обозвана ISO-8859.

Определяет она также каталоги, символьические ссылки, специальные файлы устройств, указывая для последних старшие и младшие номера устройств.

Наконец, команда stat (и это - встроенная команда оболочки), с именем файла в качестве аргумента, выводит большую часть существенных сведений о файле в удобном для восприятия виде, например, включая идентификатор inode, режим доступа (в символьной форме), идентификаторы владельца и группы, временные атрибуты, количество жестких и символьических ссылок.

Приведенных способов получения информации о файле, как кажется, пользователю должно быть достаточно. Переходим к манипуляциям с существующими файлами - копированию, перемещению, переименованию, удалению.

Манипулирование файлами

Манипулирование файлами осуществляется командами, входящими в состав набора coreutils. Начнем с копирования - это выполняется очень простой командой, ср, имеющей, однако, весьма разнообразные аспекты применения. В самом простом своем виде она требует всего двух аргументов - имени файла-источника на первом месте и имени целевого файла - на втором:

```
$ cp file_source file_target
```

Этим в текущем каталоге создается новый файл (`file_target`), идентичный по содержанию копируемому (`file_source`). То есть область данных первого будет дублировать таковую последнего. Однако области метаданных у них будут различны изначально. Целевой файл - это именно новый файл, со своим идентификатором `inode`, заведомо иными временными атрибутами; его атрибуты доступа и принадлежности в общем случае также не обязаны совпадать с таковыми файла-источника.

Новый файл может быть создан и в произвольном каталоге, к которому пользователь имеет соответствующий доступ: для этого следует только указать полный путь к нему:

```
$ cp file_source dir/subdir/file_target
```

Если в качестве второго аргумента команды указано просто имя каталога, то новый файл будет создан в нем с именем, идентичным имени файла-источника. Однако подчеркну, что в любом случае копирования создается именно новый файл, никак после этого не связанный с файлом исходным.

Если в качестве последнего аргумента выступает имя каталога, он может предваряться любым количеством аргументов - имен файлов:

```
$ cp file1 file2 ... file3 dir/
```

В этом случае в целевом каталоге `dir/` будут созданы новые файлы, идентичные по содержанию файлам `file1`, `file2` и т.д.

Если в целевом (или текущем) каталоге уже имеется файл с именем, совпадающим с именем вновь создаваемого файла, он в общем случае будет без предупреждения заменен новым файлом. Единственное средство для предотвращения этого - задание опции `-i` (от *interactive*) - при ее наличии последует запрос на перезапись существующего файла:

```
$ cp -i file1 file2  
overwrite file2? (y/n [n])
```

Как было показано в разделе о командных оболочках, некоторые из них (например, `zsh`) могут быть настроены так, чтобы по умолчанию не допускать перезаписи существующих файлов. Однако если такая потребность осознанно возникнет, это можно выполнить с помощью опции `-f` (от *force*). К слову сказать, она также аннулирует действие опции `-i`, например, при использовании ее в псевдониме команды `cp`.

Имя каталога может выступать и в качестве первого аргумента команды `cp`. Однако это потребует опции `-R` (иногда допустима и опция `-r` - в обоих случаях от *recursive*). В этом случае второй аргумент также будет воспринят как имя каталога, который не только будет создан при этом, но в нем также будет рекурсивно воспроизведено содержимое каталога источника (включая и вложенные подкаталоги).

При копировании файлов, представляющих собой символические ссылки, они будут преобразованы в регулярные файлы, копирующие содержимое файлов - источников ссылки. Однако при рекурсивном копировании каталогов, содержащих символические ссылки, возможно их воспроизведение в первозданном виде. Для этого вместе с опцией `-R` должна быть указана одна из опций `-H` или `-L`. Однако обе они при отсутствии `-R` игнорируются.

Как уже было сказано, создаваемые при копировании целевые файлы по умолчанию получают атрибуты доступа и времени, не зависящие от таковых файла-источника. Обычно они определяются значением переменной `umask`, заданной глобально, в профильном файле командной оболочки пользователя (по умолчанию значение `umask` обычно - 022). Однако при желании атрибуты исходного файла можно сохранить в файле целевом - для этого предназначена опция `-p`. Разумеется, атрибуты эти будут сохранены только в том случае, это это допустимо целевой файловой системой: не следует ожидать, что атрибуты доступа и принадлежности будут сохранены при копировании на носитель с файловой системой FAT.

Для выполнения операции копирования файла он должен иметь атрибут чтения для пользователя, выполняющего копирование; кроме того, последний должен обладать правом на изменение каталога, в который производится копирование.

Кроме простого копирования файлов, существует команда для копирования с преобразованием - `dd`.

Обобщенный ее формат весьма прост

```
$ dd [options]
```

то есть она просто копирует файл стандартного ввода в файл стандартного вывода, а опции описывают условия преобразования входного потока данных в выходной. Реально основными опциями являются `if=file1`, подменяющая стандартный ввод указанным файлом, и `of=file2`, проделывающая ту же операцию со стандартным выводом.

А далее - прочие условия преобразования, весьма обильные. Большинство из них принимают численные значения в блоках:

- опции `ibs=n` и `obs=n` устанавливают размер блока для входного и выходного потоков, `bs=n` - для обоих сразу;
- опция `skip=n` указывает, сколько блоков нужно пропустить перед записью входного потока;
- опция `count=n` предписывает скопировать из входного потока лишь указанное количество блоков.

Имеется и опция conv=value, которая преобразует входной поток в соответствие с принятыми значениями, например, из формата ASCII в формат EBCDIC, рекомендуемый для использования в ОС на базе Unix System V.

Сфера применения команды dd далеко выходит за рамки простого копирования файлов. Например, именно с ее помощью изготавливаются загрузочные дискеты.

Следующие две часто требуемые файловые операции - переименование и перемещение, - выполняются одной командой, mv. Она требует минимум двух аргументов - имени источника и целевого имени. Если оба они - имена файлов, происходит переименование первого файла во второй. Если последним аргументом выступает имя уже существующего каталога, то файл или каталог, указанные в качестве первого аргумента, перемещаются в каталог назначения. Причем если первый аргумент - файл, между первым и последним аргументами может быть сколько угодно аргументов - имен файлов (но не каталогов).

Как и при копировании, при перемещении и переименовании одноименные файлы, ранее существовавшие в целевом каталоге, затираются, замещаясь файлами-источниками без предупреждения. Чтобы этого не случилось, используется опция -i, требующая запрос на подтверждение действия. Напротив, опция -f в принудительном порядке перезаписывает существующий файл.

Операции копирования и перемещения/переименования выглядят сходными, однако по сути своей глубоко различны. Начать с того, что команда mv не совершает никаких действий с перемещаемыми или переименовываемыми файлами - она модифицирует каталоги, к которым присвоены имена этих файлов. Это имеет два важных следствия. Во-первых, при перемещении/переименовании файлы сохраняют первозданные атрибуты доступа, принадлежности и даже времени изменения метаданных (ctime) и модификации данных (mtime) - ведь ни те, ни другие при перемещении/переименовании файла не изменяются.

Во-вторых, для выполнения этих действий можно не иметь никаких вообще прав доступа к файлам - достаточно иметь право на изменение каталогов, в которых они переименовываются или перемещаются: ведь имя файла фигурирует только в составе каталога, и нигде более.

Аналогичный смысл имеет и удаление файлов, выполняемое командой

```
$ rm filename
```

в которой аргументов, означающих имена подлежащих удалению файлов, может быть произвольное количество. Как и при перемещении, при этом не затрагиваются ни метаданные, ни данные файлов, а только удаляются их имена из родительских каталогов. И потому для удаления файлов опять же не обязательно иметь какие-либо права в их отношении - достаточно прав на изменение содержащих их каталогов.

Командой rm файлы-аргументы будут удалены в общем случае без предупреждения. Подобно командам cp и mv, для команды rm предусмотрены опции -i (запрос на подтверждение) и -f (принудительное удаление вне зависимости от настроек оболочки).

Интересный момент - удаление случайно созданных файлов с именами, "неправильными" с точки зрения системы или командной оболочки. Примером этого могут быть имена, начинающиеся с символа дефиса.

Если попробовать сделать это обычным образом

```
$ rm -file
```

в ответ последует сообщение об ошибке типа

```
rm: illegal option -- l
```

то есть имя файла будет воспринято как опция. Для предотвращения этого такое "неправильное" имя следует предварить символом двойного дефиса и пробелом, означающими конец списка опций:

```
$ rm -- -file
```

В принципе, команда rm ориентирована на удаление обычных и прочих файлов, но не каталогов. Однако с опцией -d она в состоянии справиться и с этой задачей - в случае, если удаляемый каталог пуст. Наконец, опция -R (или -r) производит рекурсивное удаление каталогов со всеми их файлами и вложенными подкаталогами.

Это делает использование опции -R весьма опасным: возможно, набивший оскоину пример

```
$ rm -R /
```

когда при наличии прав суперпользователя уничтожается вся файловая система, и утириован, но в локальном масштабе такая операция более чем реальная.

Специально для удаления каталогов предназначена команда

```
$ rmdir
```

которая способна удалить только пустой каталог. Кроме того, с опцией -r она может сделать это и в отношении каталогов родительских - но также только в том случае, если они не содержат файлов.

Архивация и компрессия

Архивация и компрессия - это уже не только манипулирование файлами, но и, некоторым образом, изменение их контента. Тем не менее рассмотрим их в этом разделе - ведь с позиций пользователя их смысл близок копированию файлов. И, собственно, целям резервного копирования и архивации, и компрессии призваны служить.

Для пользователя DOS/Windows, привыкшего к программам типа Zip/WinZip, архивация и компрессия неразрывны, как лошади в упряжке. Однако это - разные действия. Архивация - это сборка группы файлов

или каталогов в единый файл, содержащий не только данные файлов-источников, но и информацию о них - имена файлов и каталогов, к которым они приписаны, атрибуты принадлежности, доступа и времени, что позволяет восстановить как данные, так и их структуру из архива в первозданном виде. Компрессия же предназначена исключительно для уменьшения объема, занимаемого файлами на диске (или ином носителе). Для архивации и компрессии предназначены самостоятельные команды. Хотя архивацию и компрессию можно объединить в одной конструкции или представить так, будто они выполняются как бы в едином процессе.

Традиционные средства архивации Unix-систем - команды `cpio` и `tar`. Суть первой, как можно понять их названия - копирование файлов в файл архива и из файла архива. Используется она в трех режимах.

Первый режим, `copy-out`, определяемый опцией `-o` (или `--create`), предусматривает считывание списка файлов (`name list`) со стандартного ввода и объединяет их в архив, который может быть направлен в архивный файл или на устройство для записи резервных копий. Список файлов для архивирования может представлять собой вывод какой-либо иной команды. Так, в примере

```
$ find ./* | cpio -o > arch.cpio
```

файлы текущего каталога, найденные командой `find`, при посредстве команды `cpio` будут направлены в архивный файл `arch.cpio`.

Второй режим (`copy-in`, опция `-i`, или `--extract`) осуществляет обратную процедуру: развертывание ранее созданного архива в текущем каталоге:

```
$ cpio -i < arch.cpio
```

Здесь нужно заметить, что если разворачиваемый архив включает подкаталоги, автоматически они созданы не будут, и последует сообщение об ошибке. Для создания промежуточных каталогов команда `cpio` должна использоваться с опцией `-d` (`--make-directories`).

В третьем режиме (`copy-pass`, опция `-p`, или `--pass-through`) команда `cpio` выполняет копирование файлов из одного дерева каталогов в другой, комбинируя режимы `copy-out` и `copy-in`, но без образования промежуточного архива. Список файлов для копирования (`name list`) считывается со стандартного ввода, а каталог назначения указывается в качестве аргумента:

```
$ cpio -p dir2 < name_list
```

Команда `cpio` имеет множество опций, позволяющих создавать, в частности, архивы в различных форматах (для межплатформенной переносимости). Однако я на них останавливаться не буду, отсылая заинтересованных к соответствующей [ман-странице](#): она не кажется мне удобной в применении. И упомянута здесь, во-первых, для полноты картины, во-вторых - универсальности ранее (архивы `cpio` понимаются абсолютно всеми Unix'ами), в третьих - как одно из средств преобразования пакетов, используемых в различных дистрибутивах Linux, друг в друга. Например, утилита `rpm2cpio` преобразует широко распространенный формат пакетов `rpm` в еще более универсальный `cpio`.

Основным же средством архивирования во всех Unix-системах является команда `tar`. Обобщенный формат ее

```
$ tar [options] archiv_name [arguments]
```

где `archiv_name` - обязательный аргумент, указывающий на имя архивного файла, с которым производятся действия, определяемые главными опциями. Формы указания опций для команды `tar` очень разнообразны. Исторически первой была краткая форма без предваряющего дефиса, что поддерживается и поныне. Однако в текущих версиях команды в целях единообразия утверждена краткая форма с предваряющим дефисом или дублирующая ее полная форма, предваряемая двумя дефисами. Некоторые опции (например `--help` - получение справки об использовании команды) предусмотрены только в полной форме.

Главные опции и указывают на то, какие действия следует выполнить над архивом в целом:

- создание архива (опция `c`, `-c` или `--create`);
- просмотр содержимого существующего архива (опция `t`, `-t` или `--list`);
- распаковка архива (опция `x`, `-x`, `--extract` или `--get`).

Легко понять, что при работе с архивом как целым одна из этих главных (т.н. функциональных) опций обязательна. При манипулировании же фрагментами архива они могут подменяться другими функциональными опциями, как то:

- `r` (`-r` или `--append`) - добавление новых файлов в конец архива;
- `u` (`-u` или `--update`) - обновление архива с добавлением не только новых, но и модифицированных (с меньшим значением атрибута `mtime`) файлов;

- -A (–catenate или –concatenate) - присоединение одного архива к другому;
- --delete - удаление именованных файлов из архива;
- --compare - сравнение архива с его источниками в файловой системе.

Прочие (очень многочисленные) опции можно отнести в разряд дополнительных - они определяют условия выполнения основных функций команды. Однако одна из таких дополнительных опций - f (-f или --file), значение которой - имя файла (в том числе файла устройства, и не обязательно на локальной машине), также является практически обязательной. Дело в том, что команда tar (от *tape archiv*) изначально создавалась для прямого резервного копирования на стримерную ленту, и именно это устройство подразумевается в качестве целевого по умолчанию. Так что если это не так (а в нынешних условиях - не так почти наверняка), имя архивного файла в качестве значения опции f следует указывать явно. Причем некоторые реализации команды tar требуют, чтобы в списке опций она стояла последней.

Проиллюстрируем сказанное несколькими примерами. Так, архив из нескольких файлов текущего каталога создается следующим образом:

```
$ tar cf arch_name.tar file1 ... file#
```

Если задать дополнительную опцию v, ход процесса будет отображаться на экране - это целесообразно, и в дальнейших примерах эта опция будет использоваться постоянно.

С помощью команды tar можно заархивировать и целый каталог, включая его подкаталоги любого уровня вложенности, причем - двояким образом. Так, если дать команду

```
$ tar cvf arch_name.tar *
```

файлы каталога текущего каталога (включая подкаталоги) будут собраны в единый архив, но без указания имени каталога родительского. А командой

```
$ tar cvf arch_name.tar dir
```

каталог dir будет упакован с полным сохранением его структуры.

С помощью команды

```
$ tar xvf arch_name.tar
```

будет выполнена обратная процедура - распаковка заархивированных файлов в текущий каталог. Если при архивировании в качестве аргумента было указано имя каталога, а не набора файлов (пусть даже в виде шаблона) - этот каталог будет восстановлен в виде корневого для всех разархивируемых файлов.

При извлечении файлов из архива никто не обязывает нас распаковывать весь архив - при необходимости это можно сделать для одного нужного файла, следует только указать его имя в качестве аргумента:

```
$ tar xvf arch_name.tar filename
```

Правда, если искомый файл находился до архивации во вложенном подкаталоге, потребуется указать и путь к нему - от корневого для архива каталога, который будет различным для двух указанных схем архивации.

Ну а для просмотра того, каким образом был собран наш архив, следует воспользоваться командой

```
$ tar tf arch_name.tar
```

Если архив собирался по первой схеме (с именами файлов в качестве аргументов, вывод ее будет примерно следующим:

```
dir2/
```

```
dir2/file1
```

```
example
```

```
new
```

```
newfile
```

```
tee.png
```

При втором способе архивации мы увидим на выводе нечто вроде

```
dir1/
```

```
dir1/example
```

```
dir1/new
```

```
dir1/newfile
```

```
dir1/tee.png
```

```
dir1/dir2/
```

```
dir1/dir2/file1
```

В данном примере опция v была опущена. Включение ее приведет к тому, что список файлов будет выведен в длинном формате, подобном выводу команды ls -l:

```
drwxr-xr-x alv/alv      0 10 май 11:03 2002 dir2/
-rw-r--r-- alv/alv      0 10 май 11:03 2002 dir2/file1
```

...

Команда tar имеет еще множество дополнительных опций, призванных предотвращать перезапись существующих файлов, осуществлять верификацию архивов, учитывать при архивации разного рода временные атрибуты, вызывать для исполнения другие программы. К некоторым опциям я еще вернусь после рассмотрения команд компрессии, другие же предлагается изучить самостоятельно, воспользовавшись страницей экранной документации.

Команд для компрессии файлов несколько, но реальный интерес ныне представляют две парные утилиты - gzip/gunzip и bz2/bunzip2. Первый член каждой пары, как легко догадаться из названия, отвечает преимущественно за компрессию, второй - за декомпрессию файлов (хотя посредством должных опций они легко меняются ролями).

Команда gzip - это традиционный компрессор Unix-систем, сменивший в сей роли более старую утилиту compress. Простейший способ ее использования -

```
$ gzip filename
```

где в качестве аргументов будет выступать имя файла. При этом (внимание!) исходный несжатый файл подменяется своей сжатой копией, которой автоматически присваивается расширение *.gz.

В качестве аргументов может выступать и произвольное количество имен файлов - каждый из них будет заменен сжатым файлом *.gz. Более того, посредством опции -r может быть выполнено рекурсивное сжатие файлов во всех вложенных подкаталогах. Подчеркну, однако, что никакой архивации команда gzip не производит, обрабатывая за раз только единичный файл. Фактически форма

```
$ gzip file1 file2 ... file#
```

просто эквивалент последовательности команд

```
$ gzip file1
```

```
$ gzip file2
```

```
...
```

```
$ gzip file#
```

Правда, объединение компрессированных файлов возможно методом конкатенации (с помощью команды cat) или посредством архивирования командой tar - и о том, и о другом будет сказано чуть позже.

Команда gzip имеет и другие опции, указываемые в краткой (однобуквенно) или полной нотации. В отличие от tar, знак дефиса (или, соответственно, двойного дефиса) обязателен в обоих случаях. Так, опциями -1 ... -9 можно задать степень сжатия и, соответственно, время процедуры: -1 соответствует минимальному, но быстрому сжатию, -9 - максимальному, но медленному. По умолчанию в команде gzip используется опция -6, обеспечивающая компромисс между скоростью и компрессией.

Благодаря опции -d (--decompress) команда gzip может выполнить развертывание сжатого файла, заменяя его оригиналом без расширения *.gz. Хотя в принципе для этого предназначена команда gunzip:

```
$ gunzip file.gz
```

Использование этой команды настолько прозрачно, что я задерживаться на ней не буду.

В последнее время широкое распространение получил компрессор bzip2, обеспечивающий большую (на 10-15%) степень сжатия, хотя и менее быстродействующий. Использование его практически идентично gzip, с деталями его можно ознакомиться с помощью страницы экранной документации man bzip2. Итоговый компрессированный файл получает имя вида *.bz2 и может быть распакован командой bunzip2 (или командой bzip2 -d). Следует только помнить, что форматы *.gz и *.bz2 не совместимы между собой. Соответственно, первый не может быть распакован программой bunzip2, и наоборот.

Поскольку программы tar и gz обеспечивают каждая свою сторону обработки файлов, возникает резонное желание использовать их совместно. Самый простой способ сделать это - воспользоваться командой tar с опцией z. Например, команда

```
$ tar cvzf dir.tar.gz dir/
```

Обратите внимание, что расширение *.gz в этом случае нужно указывать в явном виде - автоматически оно к имени архива не присоединяется и компрессированный архив будет иметь вид dir.tar. Поскольку в Unix расширения имен файлов не играют той сакральной роли, что в MS DOS, это не помешает распаковке такого файла командой

```
$ tar xvzf dir.tar
```

Опция z сама по себе никакой компрессии не выполняет - она просто вызывает компрессор gzip для сжатия каждого из архивируемых файлов. Аналогичный смысл имеет и опция j - только ею для этой цели привлекается команда bzip2.

При использовании команды tar с опцией z (или j) исходные файлы остаются в неприкосновенности. Следует, однако, помнить, что архив сжатых файлов не может быть обновлен командой tar с параметрами g или u.

Есть и другой способ совместной архивации и компрессии - просто последовательность команд

```
$ tar cf dir.tar *
```

```
$ gzip dir.tar
```

В результате образуется сжатый архив - внешне такой же файл dir.tar.gz. Хотя в принципе архив сжатых файлов и сжатый архивный файл - это разные вещи (можно заметить, что они даже различаются по объему, хотя и всего на несколько байт), сжатый архив также может быть благополучно развернут командой tar с опцией z. И столь же очевидно, что он не может быть ни пополнен, ни обновлен средствами архиватора tar.

Компрессированные архивы, созданные сочетанием программ tar и gzip/bzip2 - общепринятый в Unix-системах метод распространения файлов. Однако иногда для совместимости с ОС, не допускающими двух точек в имени файла (знаете такую ОС?), компрессированным tar-архивам на присваивается расширение *.tgz.

Можно встретить и файлы с маской *.tbz2 (или даже *.tbz - именно такой вид имеют пакеты в 5-й ветке FreeBSD). Нетрудно догадаться, что это те же архивы *.tar.bz2.

Утилита find как универсальный файловый инструмент

Алексей Федорчук

В этой заметке речь пойдет о наборе исходников, известном в проекте GNU как findutils. И в первую голову - о команде find (как, впрочем, и о тесно связанной с ней команде xargs). Столь высокая честь выпадает им потому, что посредством этих двух команд можно решить если не все, то большинство (Buono Parte) проблем, возникающих при работе с файлами.

Кроме find и xargs, в состав набора findutils входят команды locate и updatedb, о которых говорилось [ранее](#), а также команды bigram, code, frcode, реального приложения которых я, честно говоря, не знаю (и, соответственно, говорить о них не буду).

Итак, файловый апфигей - команда find. Строго говоря, вопреки своему имени, команда эта выполняет не поиск файлов как таковой, но - рекурсивный обход дерева каталогов, начиная с заданного в качестве аргумента, отбирает из них файлы в соответствие с некоторыми критериями и выполняет над отбранным файловым хозяйством некоторые действия. Именно эту ее особенность подчеркивает резюме команды find, получаемое (в некоторых системах) посредством

```
$ whatis find  
find(1)           - walk a file hierarchy
```

что применительно слушаю можно перевести как "прогулка по файловой системе".

Команда find по своему синтаксису существенно отличается от большинства прочих Unix-команд. В обобщенном виде формат ее можно представить следующим образом:

```
$ find аргумент [опция_поиска] [значение] \  
    [опция_действия]
```

Аргумент - это путь поиска, то есть каталог, начиная с которого следует искать файлы, например, корневой

```
$ find / [опция_поиска] [значение] \  
    [опция_действия]
```

или домашний каталог пользователя

```
$ find ~/ [опция_поиска] [значение] \  
    [опция_действия]
```

Опция поиска - критерий, по которому следует искать файл (файлы). В качестве таковых могут выступать имя файла (-name), его тип (-type), атрибуты принадлежности, доступа или времени.

Ну а опция действия определяет, что же надлежит сделать с найденным файлом или файлами. А сделать с ними, надо заметить, можно немало - начиная с вывода на экран (-print) и кончая передаче в качестве аргументов любой другой команде (-exec).

Как можно заметить, опция поиска и опция действия предваряются знаком дефиса, значение первой отделяется от ее имени пробелом.

Однако начнем по порядку. Опции поиска команды find позволяют выполнить вышеозначенный поиск по следующим критериям (символ дефиса перед опциями ниже опущен, но не следует забывать его ставить):

- name - поиск по имени файла или по маске имени; в последнем случае метасимволы маски должны обязательно экранироваться (например, - name *.tar.gz) или заключаться в кавычки (одинарные или двойные, в зависимости от правил, принятых для данной командной оболочки); этот критерий чувствителен к регистру, но близкий по смыслу критерий iname позволяет производить поиск по имени без различия строчных и заглавных букв;
- type - поиск по типу файла; этот критерий принимает следующие значения - f (регулярный файл), d (каталог), s (символическая связь), b (файл блочного устройства), c (файл символьного устройства);
- user И group - поиск по имени или идентификатору владельца или группы, выступающим в качестве значения критерия; существует также критерии nouser и nogroup - они отыскивают файлы, владельцев не имеющие (то есть тех, учетные записи для которых отсутствуют в файлах /etc/passwd и /etc/group); эти критерии в значения, разумеется, не нуждаются;
- size - поиск по размеру, задаваемому в виде числа в блоках или в байтах - в виде числа с последующим символом с; возможны значения n (равно n блоков), +n (более n блоков), -n (менее n блоков);

- perm - поиск файлов по значениям их атрибутов доступа, задаваемых в символьной форме;
- atime, ctime, mtime - поиск файлов с указанными временными атрибутами; значения временных атрибутов указываются в сутках (точнее, в периодах, кратных 24 часам); возможны формы значений этих атрибутов: n (равно указанному значению n*24 часа), +n (ранее n*24 часа), -n (позднее n*24 часа);
- newer - поиск файлов, измененных после файла, указанного в качестве значения критерия (то есть имеющего меньшее значение mtime);
- maxdepth и mindepth позволяют конкретизировать глубину поиска во вложенных подкаталогах - меньшую или равную численному значению для первого критерия и большую или равную - для второго;
- depth - производит отбор в обратном порядке, то есть не от каталога, указанного в качестве аргумента, а с наиболее глубоко вложенных подкаталогов; смысл этого действия - получить доступ к файлам в каталоге, для которого пользователь не имеет права чтения и исполнения;
- prune - позволяет указать подкаталоги внутри пути поиска, в которых отбора файлов производить не следует.

Кроме этого, существует еще одна опция поиска - fstype, предписывающая выполнять поиск только в файловой системе указанного типа; очевидно, что она может сочетаться с любыми другими опциями поиска. Например, команда

```
$ find / -fstype ext3 -name zsh*
```

будет искать файлы, имеющие отношение к оболочке Z-Shell, начиная с корня, но только - в пределах тех разделов, на которых размещена файловая система Ext3fs (на моей машине - это именно чистый корень, за вычетом каталогов /usr, /opt, /var, /tmp и, конечно же, /home).

Критерии отбора файлов могут группироваться практически любым образом. Так, в форме

```
$ find ~/ -name *.tar.gz newer filename
```

она выберет в домашнем каталоге пользователя все компрессированные архивы, созданные после файла с именем filename. По умолчанию между критериями отбора предполагается наличие логического оператора "И". То есть будут отыскиваться файлы, удовлетворяющие и маске имени, и соответствующему атрибуту времени. Если требуется использование оператора "ИЛИ", он должен быть явно определен в виде дополнительной опции -o между опциями поиска. Так, команда:

```
$ find ~/ -mtime -2 -o newer filename
```

призвана отобрать файлы, созданные менее двух суток назад, или же - позднее, чем файл filename.

Особенность GNU-реализации команды find (как, впрочем, и ее тезки из числа BSD-утилит) - то, что она по умолчанию выводит список отобранных в соответствии с заданными критериями файлов на экран, не требуя дополнительных опций действия. Однако, как говорят, в других Unix-системах (помнится, даже и в некоторых реализациях Linux мне такое встречалось) указание какой-либо из таких опций - обязательно. Так что рассмотрим их по порядку.

Для вывода списка отобранных файлов на экран в общем случае предназначена опция -print. Вывод этот имеет примерно следующий вид:

```
$ find . -name f* -print
./file1
./file2
./dir1/file3
```

Сходный смысл имеет и опция -ls, однако она выводит более полные сведения о найденных файлах, аналогично команде ls с опциями -dglrs:

```
$ find / -fstype ext3 -name zsh -ls
88161 511 -rwxr-xr-x 1 root          root      519320 Ноя 23 15:50 /bin/zsh
```

Важное, как мне кажется, замечание. Если команда указанного вида будет дана от лица обычного пользователя (не root-оператора), кроме приведенной выше строки вывода, последуют многочисленные сообщения вроде

```
find: /root: Permission denied
```

указывающие на каталоги, закрытые для просмотра обычным пользователем, и весьма мешающие восприятию. Чтобы подавить их, следует перенаправить вывод сообщения об ошибках в файл /dev/null, то есть указать им "Дорогу никуда":

```
$ find / -fstype ext3 -name zsh -ls 2> /dev/null
```

Идем далее. Опция `-delete` уничтожит все файлы, отобранные по указанным критериям. Так, командой
\$ find ~ -atime +100 -delete

будут автоматически стерты все файлы, к которым не было обращения за последние 100 дней (из молчаливого предположения, что раз к ним три месяца не обращались - значит, они и вообще не нужны). Истреблению подвергнутся файлы в подкаталогах любого уровня вложенности - но не включающие их подкаталоги (если, конечно, последние сами не подпадают под критерии отбора).

И, наконец, опция `-exec` - именно ею обусловлено величие утилиты `find`. В качестве значения ее можно указать любую команду с необходимыми опциями - и она будет выполнена над отобранными файлами, которые будут рассматриваться в качестве ее аргументов. Проиллюстрируем это на примере.

Использовать для удаления файлов опцию `-delete`, как мы это только что сделали - не самое здоровое решение, ибо файлы при этом удаляются без запроса, и можно случайно удалить что-нибудь нужное. И потому достигнем той же цели следующим образом:

```
$ find ~/ -atime +100 -exec rm -i {} \;
```

В этом случае на удаление каждого отобранного файла будет запрашиваться подтверждение.

Обращаю внимание на последовательность символов `{}` \; (с пробелом между закрывающей фигурной скобкой и обратным слэшем) в конце строки. Пара фигурных скобок `{}` символизирует, что свои аргументы исполняемая команда (в примере - `rm`) получает от результатов отбора команды `find`, точка с запятой означает завершение команды-значения опции `-exec`, а обратный слэш экранирует ее специальное значение от интерпретации командной оболочки.

Кроме опции действия `-exec`, у команды `find` есть еще одна, близкая по смыслу, опция `-ok`. Она также вызывает некую произвольную команду, которой в качестве аргументов передаются имена файлов, отобранные по критериям, заданным опцией (опциями) поиска. Однако перед выполнением каждой операции над каждым файлом запрашивается подтверждение.

Приведенный пример, хотя и вполне жизненный, достаточно элементарен. Рассмотрим более сложный случай - собираем в один каталог всех скриншотов в формате PNG, разбросанных по дереву домашнего каталога:

```
$ find ~/ -name *.png -exec cp {} imagesdir \;
```

В результате все png-файлы будут изысканы и скопированы (или - перемещены, если воспользоваться командой `mv` вместо `cp`) в одно место.

А теперь - вариант решения задачи, которая казалась мне сначала трудно разрешимой: рекурсивное присвоение необходимых атрибутов доступа в разветвленном дереве каталогов - различных для регулярных файлов и каталогов.

Зачем и отчего это нужно? Поясню на примере. Как-то раз, обзаведясь огромным по тем временам (40 Гбайт) винчестером, я решил собрать на него все нужные мне данные, рассеянные по дискам CD-R/RW (суммарным объемом с полкубометра) и нескольким сменным винчестерам, одни из которых были отформатированы в FAT16, другие - в FAT32, третьи - вообще в ext2fs (к слову сказать, рабочей моей системой в тот момент была FreeBSD). Сгрузив все это богатство в один каталог на новом диске, я создал в нем весьма неприглядную картину.

Ну, во-первых, все файлы, скопированные с CD и FAT-дисков, получили (исключительно из-за неаккуратности монтажа), с помощью которых этого можно было бы избежать, но - спешка, спешка...) биты исполнения, хотя все это были файлы данных. Казалось бы, мелочь, но иногда очень мешающая; в некоторых системах это не позволяет, например, просмотреть html-файл в Midnight Commander простым нажатием **Enter**. Во-вторых, для некоторых каталогов, напротив, исполнение не было предусмотрено ни для кого - то есть я же сам перейти в них не мог. В третьих, каталоги (и файлы) с CD часто не имели атрибута изменения - а они нужны мне были для работы (в т.ч. и редактирования). Конечно, от всех этих артефактов можно было бы избавиться, предусмотрев должные опции монтажа накопителей (каждого накопителя - а их число, повторяю, измерялось уже объемом занимаемого пространства), да я об этом и не подумал - что выросло, то выросло. Так что ситуация явно требовала исправления, однако проделать вручную такую работу над данными более чем в 20 Гбайт виделось немыслимым.

Да так оно, собственно, и было бы, если бы не опция `-exec` утилиты `find`. Каковая позволила изменить права доступа требуемым образом. Итак, сначала отбираем все регулярные файлы и снимаем с них бит исполнения для всех, заодно присваивая атрибут изменения для себя, любимого:

```
$ find ~/dir_data -type f \  
-exec chmod a-x,u+w {} \;
```

Далее - поиск каталогов и обратная процедура над итоговой выборкой:

```
$ find ~/dir_data -type d \  
-exec chmod a+xr,u+w {} \;
```

И дело - в шляпе, все права доступа стали единообразными (и теми, что мне нужны). Именно после этого случая я, подобно митьковскому Максиму, проникся величием философии марксизма (пардон, утилиты `find`). А ведь это еще не предел ее возможностей - последний устанавливается только встающими задачами и собственной фантазией...

Так, с помощью команды `find` легко наладить периодическое архивирование результатов текущей работы.

Для этого перво-наперво создаем командой `tar` полный архив результатов своей жизнедеятельности:

```
$ tar cvf alldata.tar ~/*
```

А затем в меру своей испорченности (или, напротив, аккуратности), время от времени запускаем команду

```
$ find ~/ -newer alldata.tar \
    -exec tar uvf alldata.tar {} \;
```

Еще один практически полезный вариант использования команды `find` в мирных целях - периодическое добавление отдельно написанных фрагментов к итоговому труду жизни (например, мемуарам энтузиаста). Впрочем, чтобы сделать это, необходимо сначала ознакомиться с командами обработки файлов, к которым мы вскоре обратимся.

А пока - об ограничении возможностей столь замечательной сцепки команды `find` с опцией действия `-exec` (распространяющиеся и на опцию `-ok`). Оно достаточно очевидно: вызываемая любой из этих опций команда выполняется в рамках самостоятельного процесса, что на слабых машинах, как говорят, приводит к падению производительности (должен заметить, что на машинах современных заметить этого практически невозможно).

Тем не менее, ситуация вполне разрешима. И сделать это призвана команда `xargs`. Она определяется как построитель и исполнитель командной строки со стандартного ввода. А поскольку на стандартный ввод может быть направлен вывод команды `find` - `xargs` воспримет результаты ее работы как аргументы какой-либо команды, которую, в свою очередь, можно рассматривать как аргумент ее самоё (по умолчанию такой командой-аргументом является `/bin/echo`).

Использование команды `xargs` не связано с созданием изобилия процессов (дополнительный процесс создается только для нее самой). Однако она имеет другое ограничение - лимит на максимальную длину командной строки. Во FreeBSD, например (по крайней мере, в системе, на которой я нынче редактирую этот текст) этот лимит составляет 65536, что определяется командой следующего вида:

```
sysctl -a | grep kern.argmax
```

И способы изменить его мне не известны - был бы благодарен за соответствующую информацию.

Поэтому в реальности у меня не возникало необходимости в команде `xargs` и, соответственно, я не занимался ее глубоким изучением. Так что заинтересованных отсылаю к соответствующей `man`-странице (и - библиографии в следующем абзаце).

Библиография

Прекрасное описание работы BSD-реализации утилиты `find` (и `xargs`, заодно) - в паре статей Dru Lavigne, размещенных в Софтерре в переводах Станислава Лапшанского: [часть 1](#) и [часть 2](#).

А примеры ее использования в мирных целях совместно с утилитами обработки текстов - в [следующей статье](#).

Использование bash и утилит для обработки текста

Сергей Майков

От редактора: Этим начинается цикл заметок, который можно озаглавить примерно как "Готовые решения по shell-скрипtingу" - Алексей Федорчук.

Дана задача - изменить во всех html-файлах какие-либо атрибуты.

Например, надо изменить во всех тэгах `<body>` фоновый цвет на черный, цвет букв - на белый. Для определенности предположим, что нам не важно, какой цвет был раньше. При этом следует либо заменить значения атрибутов `bgcolor` и `text`, либо добавить их, но не "попортить" другие возможные атрибуты.

Начнем с наиболее простого случая - единичной замены с помощью `sed`, а потом уже завернем все это в `find` для замены множественной.

Мне кажется, что проще сперва удалить наши атрибуты цвета (если они есть), а потом уже добавить наши новые значения:

```
cat test.html | sed -e '#(0)
s/<body[^>]*\)\ bgcolor=[^>]*\1/i;
# (1) убираем атрибут bgcolor
s/<body[^>]*\)\ text=[^>]*\1/i;
# (2) убираем атрибут text
s/<body\)\1 bgcolor="#000000" text="#" #fffff/i
# (3) вставляем новые значения атрибутов
'
```

Попробуем понять, что же мы тут написали. Для начала (строка (0)) выводим содержимое обрабатываемого файла командой `cat` и передаем его по конвейеру утилите `sed`. Далее...

...Строка (1):

- `s/` - начинаем команду поиска и замены. Команда имеет вид `s/SEARCH/REPLACE/FLAGS`, где `SEARCH` - то, что ищем; `REPLACE` - то, на что заменяем; `FLAGS` - флаги режима поиска. Самые распространенные

флаги, это i - регистронезависимый поиск и g - искать во всем тексте, а не только первое совпадение. Вместо символов / в качестве разделителя могут выступать любые другие символы.

- `\(<body[^>]*\)` - ищем тэг body и засыпаем его и все что следует за ним до нужного атрибута (слово bgcolor) во внутреннюю переменную 1. Для этого мы используем скобки. Здесь мы применяем регулярное выражение `[^>]*` (ноль или больше любых символов, кроме >) вместо `.*` (ноль или больше любых символов) для того, чтобы наша звездочка `(.)` не "сожрала" весь текст до последнего вхождения ограничивающего атрибута bgcolor в одной строке. Это случится, например, если после тэга body будет идти тэг html с атрибутом bgcolor. А применяя `[>]` мы ограничиваем его аппетит первым же символом `>`, то есть концом тэга body. Заключение в скобки позволяет нам запомнить найденный текст (то есть то, что удалять не надо).
- `bgcolor=[^>]*` - ищем слово bgcolor= и любое количество любых символов, кроме пробела и `>`, после него. То есть, как раз то, что нам надо удалить.

`\1` - это часть REPLACE, в которой мы подставляем сохраненный с помощью скобок текст. i - это упоминаемый выше флаг игнорирования регистра.

Строка (2) - идентична первой.

Строка (3) - Ищется начало тэга body и подставляются наши атрибуты.

Собственно, непосредственную замену мы разобрали и переходим к обработке множества файлов. Тут позволю себе небольшое лирическое отступление.

Относительно недавно в sed появилась очень приятная опция `-i`. Она позволяет производить изменения непосредственно в файле. Если раньше для изменения файла приходилось делать что-то вроде:

```
$ sed -e 'что-то делаем' test.txt > test.txt.tmp ;  
mv -f test.txt.tmp test.txt
```

то теперь достаточно

```
$ sed -i -e 'что-то делаем' test.txt
```

Чтобы узнать, поддерживается ли опция `-i`, достаточно сделать `sed --help`.

Следовательно, если наш sed не поддерживает опции `-i`, то придется разбивать команду на две и это мешает нам использовать команду find с опцией действия `-exec`. Поэтому наш скрипт разбивается на два варианта: с использованием `find -exec` и использование find с циклом for.

Обработка всех файлов в каталоге и подкаталогах. Вариант 1.

Тут все просто. Используем возможность find задавать действие для каждого найденного файла и получаем окончательный вариант нашего скрипта:

```
$ find -name '*.html' -exec sed -i -e  
's/(<body[^>]*) bgcolor=[^>]*\1/i;  
s/(<body[^>]*) text=[^>]*\1/i;  
s/(<body)\1 bgcolor="#000000"  
text="#ffffff"/i' "{}"\;
```

Тут отмечаем, что имя файла передается в виде {}, которые мы заключили в двойные кавычки на тот случай, если в имени файла содержатся пробелы. Конец команды `-exec` - это ;, "заэкранованная" от bash'a обратным слэшем (подробности об использовании команды find - [здесь](#)).

Обработка всех файлов в каталоге и подкаталогах. Вариант 2.

Если у нас древний sed, то придется разбивать действие с файлом на две команды, как показано выше. Из-за этого мы не можем воспользоваться командой find в сочетании с `-exec`. Чтож, это не беда, будем использовать возможности bash'a, в частности его циклы.

Первое, что приходит в голову, это использовать что-то вроде:

```
for i in `find -name '*.html'`; do  
    sed -i -e 's/(<body[^>]*) bgcolor=[^>]*\1/i;  
    s/(<body[^>]*) text=[^>]*\1/i; s/(<body)\1/  
    bgcolor="#000000" text="#ffffff"/i' "$i" > $i.tmp  
    mv -f $i.tmp $i
```

done

Все замечательно, но если у нас есть файлы с пробелами в имени, то мы получаем кучу сообщений о том, что файлы не найдены. Так как при подстановке списка файлов от find ... bash обрабатывает его как список слов, разделенных пробелами (точнее, символами, указанными в переменной \$IFS), то вместо файла Name with spaces.html, мы получаем три "псевдоимени" файлов: Name, with и spaces.html (**от редактора: лишний стимул не следовать порочной практике бездумного формирования имен файлов из первой фразы текстового документа, принятой... ну сами знаете где - А.Ф.**).

Поэтому я предлагаю воспользоваться следующей конструкцией:

```
find -name '*.html' | while read i; do  
    sed -i -e 's/(<body[^>]*) bgcolor=[^>]*\1/i;  
    s/(<body[^>]*) text=[^>]*\1/i; s/(<body)\1/  
    bgcolor="#000000" text="#ffffff"/i' "$i" > "$i.tmp"
```

```
mv -f "$i.tmp" "$i"
done
```

Здесь read читает строки в переменную `i` и мы гарантированно получаем полное имя файла.

Сироты коммандера Нортона

Алексей Федорчук

Написано: 2001

Последняя редакция: 2004.08.24

-Кто твоя мать, рядовой Петров?

-Коммунистическая партия Советского Союза!

-Кто твой отец?

-Вы, товарищ коммандир!

-Какова твоя заветная мечта?

-Сиротой бы остаться...

Из старого советского анекдота

Всенощная наша любовь к файловому менеджеру коммандера Нортона и многочисленным его потомкам для любых платформ известна и в комментариях не нуждается (о причинах ее - в [заметке про konqueror](#)). Не миновала эта любовь и открытые Unix-подобные системы. Где роль главного наследника славного коммандера прочно закрепилась за Midnight Commander'ом (выступающим под псевдонимом `mc`), весьма точно воспроизводящим внешний облик своего родителя, но далеко превзошедшим его функционально. Должен сознаться, что на заре моего приобщения к Linux'у без `mc` я не обходился: именно он, наряду с KDE, помог сломать психологический барьер перед командной строкой, выросший за годы плодотворного влияния Windows.

Однако время шло, я постепенно проникался величием традиционных Unix-средств для управления файлами, далеко превосходящими по скорости и эффективности любые Commander'ы (не говоря уж об Explorer'ах). И к волшебному сочетанию символов `mc` обращался все реже и реже - хотя по привычке устанавливал его всегда, чтобы было. Ну и для визуализации результатов своих действий он часто оказывался не лишним.

Первое разочарование в `mc` постигло меня на стадии приобщения к FreeBSD. Если его версия для Linux могла носить имя сына коммандера Нортона с гордостью, то FreeBSD-версию иначе чем коммандирской сироткой язык назвать не поворачивался (речь идет о 2001 году - ныне были недостатки Free'шной версии изжиты, подобно пьянству при Михал Сергеиче). Начать с того, что по непонятным причинам вызывался он там не волшебной аббревиатурой `mc`, а весьма неуклюжим и непривычным буквосочетанием `midc`. Что, конечно, решалось просто введением дополнительного псевдонима в профильном файле, например, для `csh`

```
alias mc      midc
```

однако все равно раздражало. Но это еще полбеды. Далее обнаружилось отсутствие встроенного редактора: если соответствующая опция в `mc` (пардон, в `midc`) была включена, по нажатию на F4 вызывался не кто иной, как `vi`.

Конечно, и это поддавалось лечению - или переопределением переменной `EDITOR`, или использованием внешнего редактора. Благо последний мог имитироваться редактором `le`, идеологически и по интерфейсу весьма схожим с `mcedit` (и обладающим к тому же рядом полезных качеств).

Но с было трудно примириться любому старому коммандирскому комбатанту - так это с непотребным поведением клавишных комбинаций по умолчанию. Так, комбинация **Control+Enter** отнюдь не помешала файл под курсором в командную строку, а **Control+O** вместо отключения панелей вызывала просто немедленный (и - без предупреждения) выход из `midc`. И Midnight Commander под FreeBSD был мной окончательно заброшен.

Во избежание недоразумений повторяю - все сказанное относится именно к `midc` под FreeBSD. В Linux-своей версии `mc` - это мощный, полнофункциональный файловый менеджер, ничуть не уступающий FAR'у для Windows. Да и под FreeBSD, по сведениям Андрея Лаврентьева (<http://unix1.jinr.ru/~lavr/>), умолчальные его недостатки могут быть ликвидированы путем пересборки с соответствующими настройками, однако об этом я узнал позднее.

И тут наступил психологический момент вспомнить о другом коммандерском отпрыске для FreeBSD - Demos Commander'e (или, сокращенно, deco). Получившего свое имя, разумеется, не от древнегреческих демократов, а от пионера российского провайдерства (автор - Serge Vakulenko). И конечно, ничего специфически FreeBSD'шного не содержащего: просто в дистрибутивах Linux он обычно не встречается, а во FreeBSD (и - OpenBSD также) входит стандартно, как в виде пакета, так и порта.

Надо заметить, что с первого взгляда deco производит еще более сиротское впечатление: съежившиеся на пол-экрана панели удручающе-черного цвета (на некоторых терминалах приобретающие почему-то окраску в династических цветах дома Романовых), крайне ограниченные возможности интерактивной настройки, возможности файловых манипуляций - на уровне 1-х версий Norton Commander (в частности, невозможно скопировать, переместить или удалить каталог целиком, с содержащимися в нем файлами).

Возможности просмотра и редактирования также на первый взгляд разнообразием не блещут: можно воспользоваться встроенным вышером и редактором (с весьма ограниченными возможностями) или подключить внешние аналоги. Ну а о таких излишествах, как встроенный ftp-клиент - и говорить не приходится.

Однако если набраться некоторого терпения, начинаешь понимать всю сермяжную правду, заложенную в deco. Во-первых, поведение клавишных комбинаций в нем привычно (пользователю mc) и предсказуемо. Во-вторых, возможности настройки его не столь уж бедны, как кажется на первый взгляд. Есть возможность создания пользовательского меню (вызываемого по клавише **F2**) - неотъемлемого атрибута "командирских" файловых менеджеров. К любым типам файлов, определяемых по маскам, можно привязать не только команды, но и сколь угодно сложные их конструкции. А самое главное - в deco обнаружился замечательный режим командной строки.

Включение этого режима (через меню или комбинацией клавиш **Control+P**) приводит к тому, что действия в панелях становятся невозможными, но зато открываются самые широкие возможности манипулирования непосредственно командами оболочки. После чего становится ясным, почему по умолчанию панели deco свернуты на половину экрана (что, впрочем, легко изменить - включив через меню или комбинацией **Control+F** режим Full Screen: нижняя его часть, в сущности, представляет собой обычное терминальное окно).

Сам по себе deco предлагает на выбор одну из двух встроенных командных оболочек - /bin/sh и /bin/csh. Правда, интерактивные их возможности очень бедны: не поддерживаются ни пролистывание истории команд (хотя таковая само по себе и поддерживается), ни автодополнение, ни псевдонимы, ни прочие давно уже привычные радости. Попытки настройки оболочек также остаются безуспешными. по крайней мере, никакие мои манипуляции с профильными файлами (типа ~/.csh или ~/.profile) эффекта не возымели ни малейшего. Честно говоря, для меня так и осталось загадкой, откуда deco черпает сведения о переменных оболочки и окружения.

Тем не менее, работа в режиме командной строки оказывается весьма комфортной. При этом начинаешь понимать, что программа эта, в сущности, никакой не файловый менеджер; или, мягче сказать, эта ее функция - лишь дополнение к основной, а именно: визуализации действий в командной строке. То есть именно то, что восхитило меня в свое время в Konqueror'e - файловом менеджере-браузере из KDE. Кстати сказать, в экранной документации deco так и называется - visual shell, а отнюдь не файловый менеджер. Конечно, хотелось бы, чтобы терминальное окно под визуализационными панелями использовалось наиболее эффективно. То есть - подключить к deco какую-либо мощную современную командную среду (типа bash, tcsh или zsh). Однако возможности для этого не предусмотрено: так называемые /bin/sh и /bin/csh встраиваются в него статически, и изменить это мне не удалось.

К сожалению, ожидать совершенствования deco не приходится: судя по всему, развитие программы прекратилось во второй половине 90-х годов. Однако сам по себе факт ее существования вселяет надежду: а вдруг кому придет в голову создать аналогичный визуализатор командных оболочек, но уже на современном этапе их развития?